Chair of Network Architectures and Services
School of Computation, Information, and Technology
Technical University of Munich

TECHNICAL UNIVERSITY OF MUNICH

SCHOOL OF COMPUTATION, INFORMATION, AND TECHNOLOGY

INFORMATICS

BACHELOR'S THESIS IN INFORMATICS

Private Group Management for Mix Networks

Christoph Schnabl

TECHNICAL UNIVERSITY OF MUNICH

SCHOOL OF COMPUTATION, INFORMATION, AND TECHNOLOGY

INFORMATICS

Bachelor's Thesis in Informatics

Private Group Management for Mix Networks

Privates Gruppen Management für Mix Netzwerke

Author:	Christoph Schnabl
Supervisor:	Prof. Dr.-Ing. Georg Carle
Advisor:	Daniel Hugenroth, M. Sc. (University of Cambridge)
	Filip Rezabek, M. Sc. (TUM)
	Richard von Seck, M. Sc. (TUM)
Date:	February 15, 2023

I confirm that this Bachelor's Thesis is my own work and I have documented all sources and material used.

Garching, February 15, 2023	
Location, Date	Signature

Abstract

Popular group messaging applications, such as Signal or WhatsApp, provide end-to-end encryption protecting the rights of individuals. However, they lack metadata privacy, which means that an attacker could learn about group membership by analyzing communication patterns. Mix networks provide metadata privacy but their decentralized nature makes certain group management features challenging to implement.

This thesis introduces the decentralized Private Group Management (PGM) protocol that provides both message confidentiality and metadata privacy. The first property is accomplished by encrypting each message with an ephemeral key using symmetric ratchets. The ratchets are regularly updated by one static admin. I also show that having only one admin is sufficient. The second property is achieved by deploying the PGM protocol to Loopix, a type of mix network and utilizing Rollercoaster, an efficient group multicast scheme. To synchronize group state and ensure message order the PGM protocol introduces the hash chain datatype. The protocol prioritizes group control messages hence making membership changes propagate more quickly. This work extends a simulator, introduced in the Rollercoaster paper, to evaluate PGM empirically. The benchmarks show that the latency of membership operations is roughly 50% higher compared to payload latency for a typical scenario. The share of the time, where the group state is synced, varies from 96% to 99.3% depending on the scenario. Prioritizing group control messages decreases mean latency for group membership updates from 10.9 to 9.9 seconds for intensive scenarios.

First I want to express my appreciation to my advisors for making this thesis possible and introducing me to the world of research. I would like to thank Daniel Hugenroth for exposing me to the captivating world of security and privacy research, providing many new ideas, and always coming up with quick fixes for any python related inquiries. Special thanks to Filip Rezabek and Richard von Seck for making this collaboration possible, and for their valuable and regular feedback during our meetings.

I would also like to express my gratitude to Prof. Carle for supervising this thesis and providing valuable feedback and motivation during my introductory and midterm talk and to the Chair of Network Architectures and Services at TUM for fostering scientific exchange through their Oberseminar.

Finally, I want to thank my family, Grete, Christian, Teresa, Minka, and my friends Daniel M. Swoboda, Sven Olaf Rohr, Lukas Bock, Zeno Fox, Stefan Zheng, Celine Röder, Aamin Dev, Nejira Hadzalic, Thomas Kaar, Magda Bachinger, Linda Lu, Patrick Annen, and Brianna Fan for their continued support throughout my personal and professional life.

Contents

List of Figures

Chapter 1

Introduction

"Just metadata."

Almost everyone communicates online daily and relies on applications to keep their data private. Users use chat applications that employ End-to-End (E2E) encryption, such that only the recipient of a message can read it. The content of messages is not the only type of data that needs to be protected.

Data about who is talking to whom (and when, for how long, ...) is metadata and also important to be kept private. Consider the case where the behavior of a user is leaked, e.g., video chatting with a psychologist every Tuesday at 2 p.m. These two users could be friends having a regular coffee chat rather than patients and therapists. However, those with access to metadata may infer that the user is in therapy. This is problematic beyond health-associated applications, including protecting sources in journalism, avoiding censorship, and many other cases. Current messaging applications only support E2E encryption, but limited metadata privacy.

Users may want to communicate with more than one party, leading us to the first part of this work — group messaging on an application level. In this work, I develop a protocol that can be used by group messaging apps to guarantee more metadata privacy. Several group messaging applications have proposed protocols with stronger privacy guarantees, Signal [1] being one of the most prominent examples. Signal relies on a centralized architecture. However, this allows for an attacker to observe communication patterns (traffic analysis) in this one place, e.g. to find out users that are online at the same time.

The second part of this work covers privacy-preserving approaches on the networking layer. This is interesting since it resolves the underlying problem of current architectures that allow adversaries to run this kind of attack. An example of a decentralized network architecture are mix networks. They have properties that allow for more metadata privacy. In particular, they resist traffic analysis by using a mixing strategy that makes it harder for attackers to link the incoming and outgoing messages of a node in a network. Anonymity networks like mix networks increase privacy, but certain features of group messaging that users expect are harder to implement. This includes ensuring a consistent group membership view for all users and implementing encryption. Thus, my main contribution is designing a decentralized group protocol that can be deployed on top of a mix network.

1.1 Research Questions

To guarantee stricter metadata privacy I developed the Private Group Management (PGM) protocol and deployed it to Loopix [2], one type of decentralized anonymous communication network. This makes certain group messaging features harder to implement. To tackle this I split the problem of enabling private group communication into the following four research questions (Q):

- **Q1 Decentralization**
  What properties does a decentralized group management protocol entail compared to a trusted server?
- **Q2 Group Management**
  How can a group management protocol on top of mix networks be designed?
- **Q3 Metadata Privacy and Security**
  What extent of metadata privacy and message confidentiality can be guaranteed?
- **Q4 Performance**
  How fast is the PGM protocol theoretically and practically?

1.2 Outline

I have motivated the need for privacy and security from a user's perspective. Chapter 2 builds upon this by introducing cryptographic primitives and the definitions of relevant terms, such as privacy, anonymity, and unlinkability. The chapter also focuses on how these concepts are achieved by mix networks and their challenges when they are employed for group messaging. In Chapter 3 I present relevant research in the area of group messaging protocols and distributed key exchange. Chapter 4 follows up on the use case of this work and derives assumptions and requirements from that. This helps in defining an abstract view of group messaging operations. I analyze properties that ensure common security goals and compare relevant protocols. In Chapter 5 I introduce my group management protocol and its core parts and a working prototype implementation. Chapter 6 includes evaluations of my protocol theoretically about my previous comparison, and practically in the form of benchmarks. Lastly, I conclude with Chapter 7 answering the research questions, and stating my core contributions and future work.

1.3 Contributions

My work aims to advance private group messaging and I make the following contributions:

1. A comparison of different group messaging protocols regarding common security goals in Chapter 4.
2. A group messaging protocol consisting of, decentralized state management, encryption, and key distribution in Chapter 5.
3. An implementation based on Loopix, one type of mix network, including benchmarks, proving practical feasibility in Chapter 6.

Chapter 2

Background

This chapter introduces the relevant background in privacy, cryptography and anonymity networks. Many different cryptographic building blocks are needed to achieve privacy, most importantly encryption and key exchange. With regards to encryption, it is crucial to define forward secrecy (FS) and post-compromise security (PCS), as these terms are used varyingly throughout the literature. Within the introduction, I have referred very informally to metadata privacy and how mix networks [3] can guarantee it. Defining metadata privacy on a technical level is important for the design and analysis of anonymity networks. Understanding the limitations of anonymity networks is crucial in deriving requirements and constraints for the PGM protocol.

2.1 Privacy

Privacy refers to an individual being able to keep personal information private. Metadata privacy and anonymity are closely related to each other as protecting one's identity enhances privacy. In particular, for group messaging this means protecting the identity of group members from non-group members.

2.1.1 Anonymity

To understand and analyze anonymity networks and their guarantees it is essential to define anonymity on a technical level. Figure 2.1 depicts the concepts of FS, PCS, unlinkability, and unobservability.

Definition 1. *(Anonymity) Anonymity is defined as keeping one's identity private from different parties depending on the context. The anonymity of individuals can only exist if the individual is indistinguishable from a group of peers, the so-called anonymity set [4].*

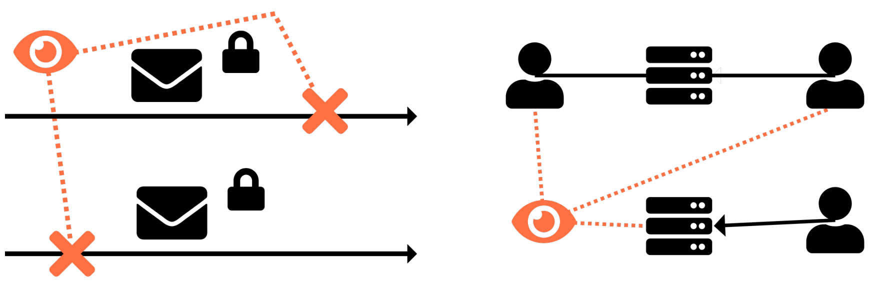

Figure 2.1: FS (top left), PCS (bottom left), unlinkability (top right), unobservability (bottom right)

For anonymity networks, it is interesting to not only consider the set of possible users that communicate with each other but rather the probability distribution of these users for one particular message. This is an information-theoretic approach to analyzing guarantees of anonymity networks [5].

Definition 2. *(Unlinkability) Two subjects, e.g. users, are said to be unlinkable if an adversary cannot link these two after observing the system.*

In the context of messaging, unlinkability is relevant between senders and the messages they sent, as well as sender-receiver unlinkability. In that way, a person is said to be anonymous if they cannot be linked to e.g. messages they send and receive [4].

Definition 3. *(Unobservability) Unobservability is the inability of an adversary to infer whether a subject has performed a certain action [4].*

For messaging, it is relevant to consider whether a user is online and sending messages (sender-online unobservability) and whether they are receiving messages (receiver unobservability). This is relevant since an adversary could link users that are repeatedly online at the same time.

2.2 Cryptography

Cryptography provides concepts that help to ensure anonymity and privacy. This includes protecting information through encryption and ensuring authorized access through authentication. Mitigating compromised keys in encryption is important, especially when keys are reused multiple times. This is especially relevant in group messaging, where users may communicate over longer periods.

Forward and Post-Compromise Secrecy

Encryption protects the confidentiality of data, such that only holders of a key can access it. As keys can be compromised additional mechanisms for stronger confidentiality guarantees should be employed.

Definition 4. *FS guarantees that past messages (messages received before a compromise) cannot be decrypted by an attacker [6].*

FS in Authenticated Key Exchange refers to an adversary that monitors all traffic and is unable to distinguish a new key from a random string.

Definition 5. *PCS guarantees that the confidentiality of future messages is not compromised even if the key is compromised [7].*

In particular, protecting against attackers that compromise long-term keys is defined as total PCS. The case where, the attacker is only able to execute cryptographic operations, but not access underlying keys, is defined as weak PCS.

2.3 Anonymity Networks

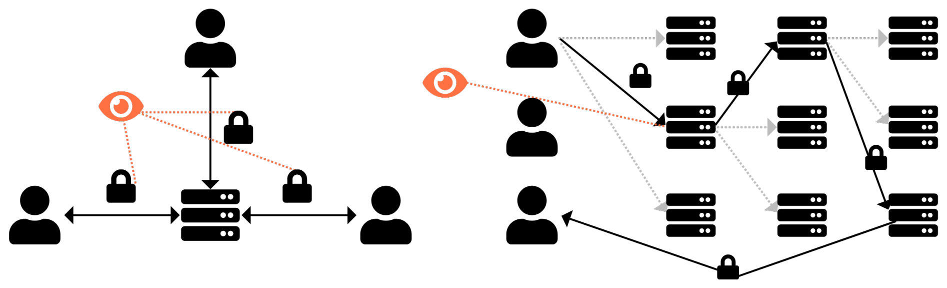

Figure 2.2: Active attack on a central server (left) vs. active attack on a mix network (right)

Anonymity networks are designed to protect the anonymity of users. These networks typically achieve this by routing messages through multiple nodes which makes it more difficult for an attacker to identify individual messages at each node in the network. Figure 2.2 visualizes how an adversary might run active attacks on a centralized server architecture and a mix network. The following mechanisms have been proposed to provide anonymity on this network level. I briefly expand on layered encryption before diving into attacks on anonymity networks, mix networks and different mixing strategies, as one example of an anonymity network. I am also introducing Loopix [2], one instance of a mix network, which is later used for my work as well as Rollercoaster, a multicast protocol that operates on top of Loopix.

2.3.1 Mix Networks

A mix network [3] consists of multiple layers of nodes. They use a so-called mixing strategy which makes it harder for attackers to trace messages within the network. One example is a threshold mixing strategy where nodes wait for messages until a preset threshold is reached to then relay them in random (mixed-up) order. However, mixing comes at the expense of higher latency, especially if multiple layers of mixes are used. The choice of mixing strategy influences this trade-off. Danezis and Serjantov [5] have shown that purely using the anonymity set can result in false security when evaluating a mixing strategy. Therefore, they are proposing a more effective metric using an information theoretic approach. The intuition behind this is to consider how hard it is for an attacker to distinguish one particular message from all the others they are observing. A more detailed, axiomatic description of how entropy as a measure solves this is eluded here and can be found in Danezis' and Serjantov's paper.
To demonstrate this difference in practice consider another mixing strategy, the Pool Mix [5]. In the Pool Mix strategy, every node stores a pool of a fixed amount of messages n. Upon receiving the N-th new message, the node forwards n random messages. Since a message may stay in the pool for possibly many rounds, an adversary using has to consider the sender of every message when using the anonymity set metric. However, one might argue that this mix does not offer the level of anonymity it would promise under an anonymity set consideration, which is confirmed using the aforementioned model of using a particular probability distribution instead of assuming uniformity [5].
Additionally, messages are routed randomly through the network and encrypted in multiple layers. The node currently handling a message is only able to decrypt one layer of encryption before forwarding the messages. This is often referred to as onion routing, layered or multiple encryptions because multiple encryption steps are applied to one packet. This is usually used in networks such as mix networks, where packets are sent through multiple different nodes before being delivered to the actual recipient. The sender chooses a route and then encrypts their message with the public key of each intermediate node and the recipient. Upon receiving the message each node decrypts one layer of encryption with its private key and passes it on to the next node. Since the current node can only decrypt one layer, it also only knows the previous and the next node [8]. This makes it more difficult for an adversary to trace messages throughout the network.

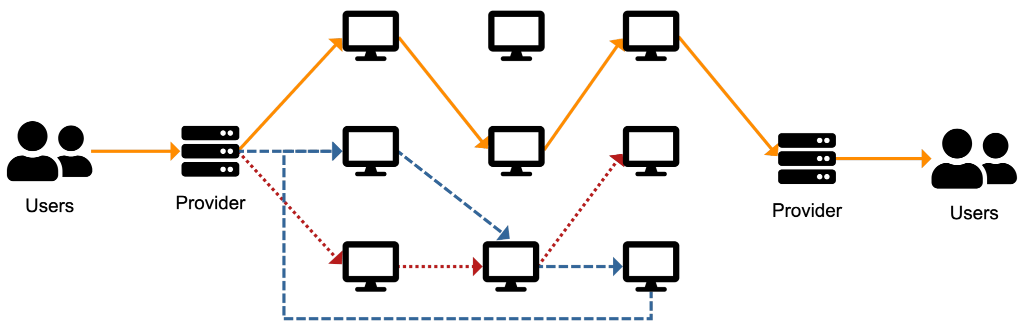

Figure 2.3: Loopix [2] architecture with three forward-connected layers of Mix Nodes. Users ingest messages through their providers into the network where the messages are routed through the network (solid orange line). Drop messages are visualized as dotted red lines, loop messages as dashed blue lines.

2.3.2 Attacks

Attacks on anonymity networks can be distinguished by the capabilities of an adversary. An attack is called passive if the adversary can only observe messages. Passive attacks are typically harder to defend against the attacker who does not perform any observable action. In the case where the adversary does send malicious messages an attack is called active [9]. With a passive attack, an adversary can observe traffic to destroy the unobservability and Unlinkability properties of users. One example of a passive attack is traffic analysis. There the adversary might try to link the sender and receiver of one particular message. This is one example of traffic analysis.

2.3.3 Loopix

Loopix [2] is one type of mix network with a fixed amount of layers of mix nodes, where nodes in one layer only forward messages to nodes in the next layer. Users access the network through so-called providers. The provider sends messages to the first layer of the network and receives messages from the last layer. In case a user is not online, their provider stores the message for them. This architecture is visualized in Figure 2.3. To guarantee privacy Loopix employs cover traffic and uses a continuous mixing strategy called Poisson mixing.
Users send messages by choosing a random route through the network, then encrypting it in layers with each node's public key and includes a delay for each stage. This dealy is sampled from an exponential distribution. Then the message is injected into the network by the user's provider. The first mix node decrypts the message and forwards it with the specified delay to the next layer. The receiving mix node in the last layer of the network then sends the message to the user's provider. Messages are E2E encrypted

with a fixed length using the Sphinx packet format [10]. Applications built on top of Loopix need to take into account that this encryption scheme is not forward-secure.
Loopix assumes a powerful Global Passive Adversary (GPA) that can observe all network traffic, and the state of some mix nodes and compromises some users. Loopix provides sender-receiver unlinkability, and sender and receiver unobservability by employing cover traffic and Poisson mixing.
Loopix distinguishes three different types of messages where the first two act as cover traffic: loop, drop, and content. Drop messages are dropped by providers upon receiving while loop messages are looped back to the sender.
In Poisson mixing, the sender draws the delays for each hop of the message from an exponential distribution. The parameter of the distribution is public and the same for every node and thus directly influences the end-to-end latency as well as privacy guarantees of a network configuration. Loopix neither implements reliable transport, nor efficient multicasts. The latter poses a latency problem for group messaging.

2.3.4 Rollercoaster

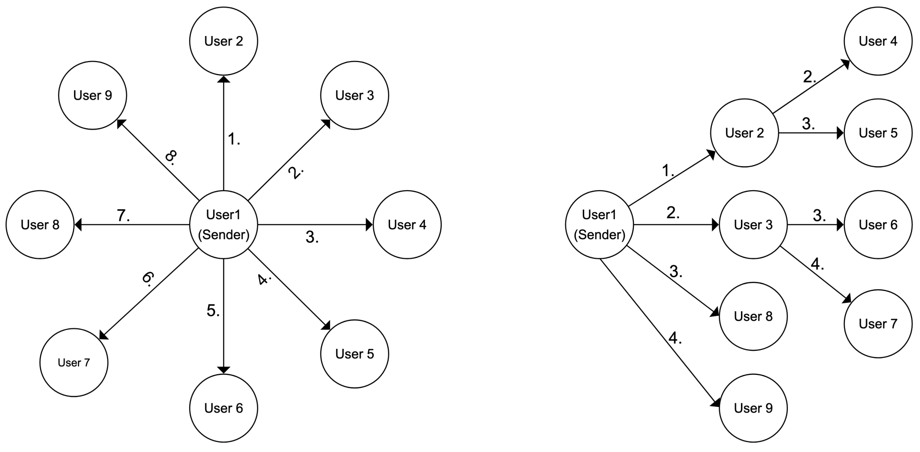

Figure 2.4: Message distribution graphs of Sequential Unicast (left) and Rollercoaster [11] (right). In Sequential Unicast the sender has to ingest one message after the other, while Users that already received the message help to fan out in Rollercoaster.

Loopix does not implement efficient multicasts needed for group messaging. Rollercoaster [11] is a protocol for more efficient group multicasts layered on top of Loopix. Rollercoaster reduces latency from $\mathcal{O}(n)$ (in the case of naive multicasts, with n participants) to $\mathcal{O}(\log n)$ by having recipients help in distributing the messages in a tree-like structure. This can be seen in Figure 2.4 where a Sequential Unicast to 8 other people leads to 8 messages in the sender's outbox, and a maximum of 4 for the sender and 2 for the receiving children in Rollercoaster.

Practically, this reduces the 99th latency for groups of over 100 users from 75.6 seconds with Loopix to 12.3 seconds with Rollercoaster. The main reason for this high latency with sequential unicasts is the congested outbox of the user that sent the broadcast message. This congestion exists since messages in Loopix are forwarded with a delay drawn from an exponential distribution, and this delay scales linearly with the number of messages in the outbox, and thus also with the group size.
The Rollercoaster paper includes a simulator [12] of both Rollercoaster and Loopix, which the authors used to benchmark their protocol. I have extended the simulator with the PGM protocol to benchmark it.

Chapter 3

Related Work

This work aims to enable private group messaging by combining group chat on an application-level with anonymous communication networks from an infrastructure perspective. On an application level related work has been conducted mostly in the context of E2E encryption in groups and distributed key exchange. I introduce how two popular group chat applications, Signal [13] and WhatsApp [14], provide encryption but lack metadata privacy. This chapter also elaborates on centralized group messaging protocols showing how WhatsApp has been influenced by Signal and how Signal has been influenced by Multi-party Off-the-Record Messaging (mpOTR) [15]. I include two decentralized approaches for key exchange: Lockboxes [16] and Decentralized Continuous Group Key Agreement (DCGKA) [17].

3.1 Centralized Group Messaging Protocols

Centralized group messaging protocols rely on a server to manage state as well as to facilitate messaging and key exchanges between users.

3.1.1 Signal

Signal is one of the most prominent and widely used examples of secure messaging. Signal implements E2E encrypted direct messaging and group messaging. The first one is used in the group protocol as a pairwise secure channel to send messages and exchange group keys. Signal relies on a centralized architecture to achieve metadata privacy for groups, but is vulnerable to passive attacks, e.g. an observer can infer that users are online at the same time and hence breaks sender-receiver unlinkability.

The Signal Protocol

Signal makes use of the X3DH key exchange [18] to initiate communication between two parties. This initial shared secret key is used to initialize the Double Ratchet [19] algorithm. A cryptographic ratchet can conceptually be viewed as a one-way function where the input key derives two new keys, one key that is used for encrypting messages, and the other one that is used in the next ratchet step. For more details refer to Appendix A. Within the Double Ratchet algorithm [19], each user utilizes symmetric ratchets (one for sending and one for receiving) as well as one Diffie Hellman (DH) ratchet. The two parties update their two symmetric ratchets with every message sent/received and update their DH ratchet.

Signal Groups

Signal [13] first introduced group messaging in 2014 as part of the Text Secure v2 protocol [13]. The protocol used pairwise channels known from the Signal protocol for one-to-one communication with each user locally storing group membership lists. This resulted in several issues besides performance considerations. Because group membership lists were managed decentrally, concurrent group updates created inconsistencies, e.g., B sending a message to a user that has previously been removed. The protocol also did not include role-based access control. In 2019, Signal introduced the Private Group System [1], a new protocol for group management. In this protocol, the group membership list is managed by the server and encrypted such that only members of the group can access it. To achieve this, the server needs to authorize a user to access the group's state without knowing the group the user belongs to. This is achieved by utilizing HMAC-based Zero-Knowledge Proofs (ZKP). That way, group membership information is kept private in case of a compromised server state. I elude an explanation of how this formally works, but a good practical analogy may be found in the following article by Naor et al. [20].

3.1.2 WhatsApp Sender Keys

WhatsApp [14] uses a protocol called Sender Keys for secure group messaging. Similar to Signal each user stores one sending chain, and $n-1$ receiving chains for all other group members, which the user updates with every message sent/received. In Sender Keys, users initialize their chains by exchanging the root keys through pairwise encrypted channels. For these channels, WhatsApp uses an implementation of the Signal protocol. This leads to a new member joining the group having to distribute the key of their sending chain to all other group members. In the case where a member is removed, all other members also have to update and distribute their keys.

3.2 Decentralized Group Messaging Protocols

For decentralized group messaging protocols, it is harder to manage state, and update keys, because there is no central server to federate this.

3.2.1 Multi-party Off-the-Record Messaging

mpOTR [15] generalizes the Off-the-Record protocol (OTR) for the asynchronous case. OTR introduced the DH ratchet mentioned previously that is also being used in mpOTR. mpOTR partitions time into sessions, with each session consisting of three distinct phases. First, during setup users agree on a shared key. During the second phase, users can communicate. Finally, the shutdown phase happens as soon as an existing user is removed, or a new user is added. In this phase users also exchange hash values of all messages for transcript consistency. After that, to continue communication a session is started.

3.2.2 Local-First-Web Auth

The Local-First-Web Auth project [16] introduces a library to build decentralized collaborative apps, such as messaging apps. It includes authentication, key management via so-called Lockboxes, and decentralized state management via a signature chain. Lockboxes can essentially be viewed as an abstraction of authenticated encryption. This allows Alice to share a private key with Bob. The signature chain is used for authorization, group membership changes, and state synchronization. Every node depicts an action that has been administrated to the group (e.g. creation, adding and removing members). Every node refers to a previous node using a cryptographic hash of the predecessor's content. FS can be achieved by regularly rotating lockbox keys. No formal security analysis has been performed so far.

3.2.3 Decentralized Continuous Group Key Agreement

DCGKA [17] is a protocol to share symmetric group keys in a decentralized setting. Similar to WhatsApp and Signal DCGKA also uses symmetric ratcheting for FS. PCS updates are performed with every message that is sent. The sender generates a random seed secret and shares it through a secure direct channel. The recipients then use this secret to derive keys to update their sending chain and the respective receiving chain. The recipient broadcasts the update to the group, and all recipients who receive the acknowledgment apply the update to both the original sender's chain and their own. This ensures that all members of the group update the ratchet for every other user.

Chapter 4

Analysis

This chapter explores, how people use group messaging and how this influences the design of the protocol. I explain specific use cases and based on that derive the protocol's requirements and assumptions. Finally, I analyze security and privacy goals and compare, how and to what extent various centralized and decentralized protocols achieve these.

4.1 Problem Statement

Users use group messaging applications to create and delete groups, add and remove users, as well as to send and receive messages. Additionally, users also expect their application to provide them with secure communication and respect their privacy. In particular, only designated recipients should be able to read their messages and know who is chatting with each other. As described in the introductory Chapter 1, existing group chat applications provide the first characteristic but lack the second one. In more general terms, the second requirement refers to group membership information being anonymous for attackers outside the group. This means that an attacker is not able to distinguish two random users from two users that share a communication relationship. Many users are reliant on their mobile phones to communicate with each other, resulting in clients being offline for prolonged periods. My work prioritizes single-device over multi-device support as this simplifies implementing state synchronization and encryption.

I assume that users already know and trust each other, as they know each other either from the offline world or through another party. This is usually, how they also verify each other's identity when one invites the other to the group.

In this case, users are mostly communicating in small-to-medium-sized groups (of roughly a maximum of 100 people) mainly due to the following two reasons. First, they want to communicate with other users they already know and second, larger groups cannot practically be considered private, as ensuring trust between all users gets increasingly harder for larger groups. This is different from the case of large groups, which require anonymity also within the group. Research on practical user behavior has also shown that larger groups do require different guarantees, such as the individual's anonymity [21].
Groups, of different natures, exist and this influences the usage of the group. With messaging behavior, I am referring to two distinct types of behavior. First, when and how often they send messages and how sent messages are distributed by the sender (think send-heavy users, vs. read-only users). Second, it is also important to consider, when and how often users are added and removed to and from the group. This not only influences the group size but also the performance of the group messaging protocol.
I could not find useful user studies investigating different group messaging scenarios and how to partition users. Hence this would require extensive user research and goes beyond the scope of this work.
Therefore, my assumptions lead to valuable insights and show how different trends in usage affect the performance of the PGM protocol. I come up with three group archetypes into the following three types. I am assuming that behavior roughly varies along the following two dimensions: when people are added and removed to and from the group and how often these group membership changes happen. Below I explain these scenarios and give common examples of how one might encounter these in daily life. A concrete model and its parameters can be found in Chapter 6.

Heavy-dynamic groups. For these groups, many constant group membership changes happen over the entire time. The size of groups of this nature does vary, but users are added and removed frequently. In practice, such groups could be a newsletter with high fluctuations.

Low-dynamic groups. Similar to the first scenario, membership changes happen less often. This can be seen as a less active group.

Invite-heavy groups. Most people are invited soon after the creation of the group. After that, the group is saturated and not many changes happen. Such a scenario could refer to people organizing an event, a meeting, or a group of close friends.

4.2 Assumptions

Within this section, I explain my assumptions for group messaging and outline how they relate to the use cases described above. These assumptions greatly influence the design and the performance of the protocol.

Existing trust relationships. On a practical level, this requires A to be able to verify B's digital identity. This can be achieved by A and B meeting offline to exchange keys, A and B already using another trusted online communication channel, or assuming a trusted Public-Key Infrastructure (PKI). All of these can be added to accomplish authentication and I thus elude this step from the design of the protocol.

Pairwise secure channels. The admin and the user that is invited not only already know each other, but they also already share a communication channel. This is important, to invite users to a group and later to update group keys [22].

Small-to-medium sized groups. The PGM protocol is designed to be used with small-to-medium-sized groups and adheres to the use cases described above. As described before, a group of hundreds of participants cannot be considered private, even with perfect encryption and metadata guarantees.

Eventual delivery. Loopix [2] does not provide message delivery guarantees and hence requires an application to do so. For my protocol, I am assuming that some sort of eventual delivery mechanism has already been implemented. Exploring how to build eventual delivery on top of Loopix could be an interesting direction for future work.

Weak causal order broadcast. The order of messages is crucial in group messaging for two reasons. First, as the protocol uses different keys, assuming a shared clock, that guarantees a total order of all messages, is not practically feasible in a distributed setting. Thus I am assuming weaker order guarantees are called weak causal order broadcasts. With this order, a message m_1 is older than m_2 either by m_1 being sent by the same user before m_2, m_1 being processed by a user before sending m_2 or by transitivity [17]. The second case can be guaranteed in Loopix, since messages are received and processed before sending out new messages.

No malicious users. It can be assumed that group members or those being invited obey the protocol without changing the semantics of group messaging. In particular, this also holds for the case where the admin becomes malicious and arbitrarily dictates the groups, e.g. by removing members. As such I am also assuming that users can identify malicious behavior. If there was a malicious user, everyone else could agree to create a new group without the wrongdoer and copy the chat history. This agreement can easily be implemented with one of the following two options. First, one user manually takes action and creates a new group with the copied chat history, but excludes the malicious user. Second, all users conduct a majority vote in the messaging application on whether to kick out the malicious user. In the case where the majority becomes malicious, the non-malicious users are still left with the first option. This assumption is particularly important for the design choices of my protocol.

Adversary

I am assuming a GPA that can observe all messages of the group but does not interfere with the communication. This means in the context of the above assumptions that the adversary cannot alter the order of messages, nor pose as a user.

4.3 Requirements

Each user may be a member or admin of multiple groups that they regularly interact with. A group can be defined as the fourtuple $(G_{id}, G_{users}, G_{admin}, G_{state})$. The group id G_{id} is a random identifier and may be expanded to include a group name, description, etc. The admin of the group G_{admin} refers to the user that created the group such that $G_{admin} \in G_{users}$. To encapsulate group state (messages and users) G_{state} refers to a set of messages. Messages are either control (add, remove, etc.) or payload messages. An order over the set of messages can then be used for two things. First to determine the current set of group members and the messages that they are allowed to receive and send. Second, this can be used to order payload messages.
Protocols such as Signal or WhatsApp use a central server to handle out-of-order messages and synchronize the group state. My first idea, of deploying a centralized protocol by using a distributed consensus protocol (e.g. Raft or Paxos), has shown some flaws. First, leader election for asynchronous mobile clients itself is very hard to achieve reliably. Second, the centralized component might leak information, and lead to added latency in the mix network.

4.3.1 Security Goals

This section explains relevant security goals by first discussing their relevance and definition within private group messaging. I map these security goals to common security properties, namely integrity and confidentiality, and how these properties can be used to ensure the desired level of security. Additionally, I consider the assumptions that are

made and discuss which security properties can be relaxed.
In computer security, the CIA triad model [23] states the following three security goals: confidentiality, integrity, and availability. Confidentiality refers to protecting data from unauthorized access. Integrity refers to protecting that from unauthorized change or deletion. Availability allows users to access data and services. Below I am explaining security properties that are relevant for group messaging in general and map them to their respective security goal.

Message confidentiality. Messages must only be readable by their receivers. This can be achieved by encryption. Neither a server, an attacker, nor anyone else not designated to read it, has to be able to decrypt the message. More advanced encryption schemes do also provide FS or PCS guarantees.

Metadata privacy. Metadata privacy on a technical level refers to networks that can guarantee anonymity, unobservability and unlinkability as explained in Chapter 2. A protocol that can be deployed on top of Loopix, can utilize the network's guarantees for Metadata privacy.

Transcript consistency. In group messaging the transcript refers to the log or list of all messages. Transcript consistency ensures that all users have a complete and ordered (under the limitation of causal order) list of messages. This also means that no messages are missing or duplicated. In centralized protocols, the server usually takes on the authority on ordering messages.

Multi-device support. Users can use multiple devices to send/receive messages. Implementing multi-device support in an E2E encrypted group chat application is difficult as it requires synchronizing messages and updating keys across multiple (possibly non-online) devices, as well as ensuring the authenticity of different devices.

4.4 Comparison

The following section compares how and to what extent different protocols were introduced in Chapter 3 about the abovementioned security properties.

Table 4.1 compares Signal [1], WhatsApp [22], DCGKA [17], Lockboxes [16], and mpOTR [15] with regards to the following properties. Whether a protocol is decentralized or not refers to its design with or without a central server in consideration. FS and PCS for message encryption as explained in Chapter 2. All protocols but Lockboxes do provide FS. In Lockboxes a weak form of FS is achieved by regularly rotating

	Signal [1]	WhatsApp [22]	DCGKA [17]	Lockboxes [16]	mpOTR [15]
Decentralized	✗	✗	✓	✓	✓
FS	✓	✓	✓	✗*	✓
PCS	✓	✗	✓	✗	✗
Number of PCS updates	$\mathcal{O}(n^2)$	-	$\mathcal{O}(n^2)$	-	-
Metadata privacy	✗*	✗	✗	✗	✗
Multi-device support	✓*	✓*	✗	✓	✗
Transcript consistency	✓	✓	✓	✓	✗

Table 4.1: Comparison of relevant security protocols across the previously introduced protocols

keys. The number of PCS updates refers to the number of messages that are ingested to update everyone's state. For Signal and DCGKA this equals to $n-1$ messages per user resulting in a total of $\mathcal{O}(n^2)$ messages, where n denotes the current group size. Signal provides a limited form of metadata privacy, while all the others do not.
Signal uses a centralized protocol, which means that it depends on a server for maintaining the confidentiality of messages and protecting metadata privacy. It is important to note that while the server itself is not aware of group details, unlinkability (e.g. between users and their messages) and unobservability (especially online unobservability) cannot be fully guaranteed in the presence of an adversary, who can monitor the server traffic.
Signal offers multi-device support, with its linked devices feature, but it has certain restrictions. Firstly, it does not support syncing of previous messages, and secondly, it is limited to linking up to five tablet or desktop devices. Nevertheless, linked devices can still communicate even if the phone is offline [24]. Signal achieves FS and PCS through the Double Ratchet [19] protocol.
WhatsApp also relies on a server for transcript consistency, WhatsApps' design provides no metadata privacy. FS is achieved through the sender keys protocol as described previously. WhatsApp does not provide PCS and keys only change when group members are added or leave. To support multi-device functionality, WhatsApp employs a relay system similar to Signal. However, phones must be online for this system to work.
DCGKA is a decentralized group protocol, that provides FS and PCS with every message sent. The number of messages sent for key updates is $\mathcal{O}(n^2)$ as every user has to broadcast their new seed secret to every other user. DCGKA does provide transcript consistency, but metadata privacy or multi-device support are not implemented.

Lockboxes provide a very weak form of FS by rotating the lockbox keys regularly. mpOTR is a decentralized protocol and as such provides FS by generating new keys for each session and ratcheting before sending a message. Transcript consistency is only available after the end of each session.

Chapter 5

Private Group Management Protocol

The following chapter introduces the high-level design and operations of the PGM protocol and its core building blocks: managing state, encrypting messages, and admin-federated updates. It also includes an adaptation to Loopix, where control messages are prioritized.

5.1 Protocol Overview

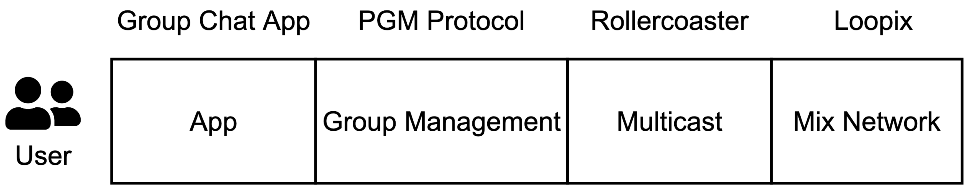

Figure 5.1: The PGM protocol builds on top of Rollercoaster and can be used by applications to implement group messaging

The PGM protocol can be used by application protocols to facilitate group communication. This includes the following operations: creating a group, sending messages, adding and removing members, and updating keys. The protocol covers each operation for both the sending and the receiving side.
To achieve the necessary metadata privacy guarantees I deploy it on top of Loopix [2], one implementation of a mix network. The PGM protocol uses Rollercoaster [11] for multicasts with lower latency. An overview of the different involved layers can be found in Figure 5.1.
Due to the decentralized nature of mix networks, two problems arise: synchronizing

group state and guaranteeing message order. State synchronization is important such that removed users cannot send or receive messages. Guaranteeing message order is needed to be able to correctly decrypt messages with the respective ephemeral key.

5.2 Building Blocks

This section introduces the following three building blocks that solve state management and encryption. The first building block is synchronizing group states and ordering messages. For this, I introduce a decentralized datatype called hash chain. The second building block covers encryption. Messages are encrypted with ephemeral keys derived from a symmetric ratchet. The third building block is managing group changes, that are federated by the admin. Only allowing the admin to make changes is sufficient as introduced in Chapter 4. The group admin also sends out key updates regularly and with every group control message to provide a PCS.

5.2.1 Synchronizing group state and message order

The decentralized architecture of mix networks creates two challenges. First, each user has local state that needs to be synced across users without a central server to resolve conflicts. Second, assuming a global total order (e.g. server time in the centralized setting) is impractical, and thus a different way to order messages is needed. This is important, first to be able to decrypt messages with their respective key, and second that removed users are not able to send messages, and messages shall not be sent to removed users either. In the following section, I present the hash chain datatype that has been inspired by the Signature Chains of Keybase [25] and the Local-First Auth [16] project.

In the PGM protocol users store messages they receive in a graph. Every node in the graph represents one message and has edges to its predecessors. This is analogous to the weak causal order over messages, where every message specifies its direct predecessors. There does not exist a total order global among users, but one for individual users such that every subgraph refers to one specific group member. All subgraphs for individual users between two control messages are called a session. A session specifies the set of current group members, that are allowed to send and receive messages.

Users keep track of heads for each subgraph of every sender. In the case where the last message was a control one, the admin's head is the only one updated. Upon sending a message, the user first appends the message to the chain by referring to all current heads as predecessors. Users then update their heads as follows. For control messages, all previous heads are reset. For all messages, the head for the user's subgraph is updated. Similarly, when receiving messages users also append the message to the chain and

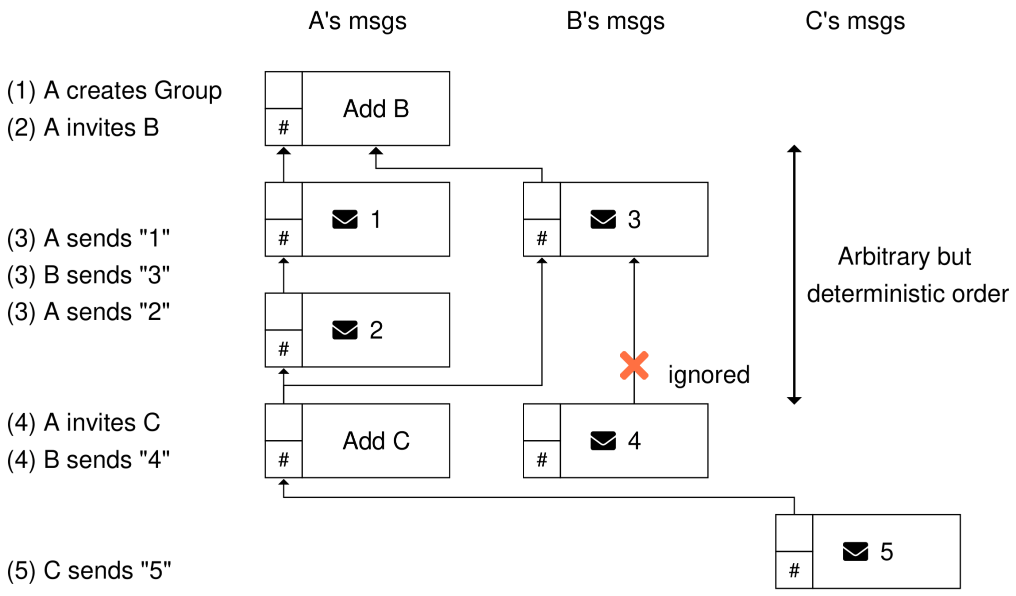

Figure 5.2: Users append their messages to the hash chain

update heads accordingly. To linearize the graph, any order that is deterministically computable for all users is possible. In the case of the PGM protocol, this is achieved by ordering all messages within one session by their sent timestamp and then ordering all sessions.

Since I assume eventual delivery the protocol does not have to handle lost messages but considers the case of out-of-order messages, e.g. m_1 depending on m_2, but m_2 arriving before m_1. This is achieved by keeping track of processed and unprocessed nodes. Upon receiving a message, it is added to the list of unprocessed nodes in case its predecessors have not been processed. If all predecessors are present the new message is processed accordingly (e.g. the user is added to the group) and inserted into the processed list. If some previous unprocessed messages now have all predecessors they depend on availability. This is done recursively until all depending messages have been resolved. One downside of this is, that delays can accumulate, e.g. by one early message with lots of dependants arriving late.

Figure 5.2 shows a simple case of a group with three users: admin A, B, and C. A creates the group and invites B. After that both A and B send messages 1,2, and 3 respectively by appending them to the chain. Then A also invites C to the group, while B tries to send a message. This node is eventually rejected by all other users since admin A did not include this message as a predecessor. The message is still displayed for one user if they have not yet received the admin's message. This way, messages sent by a member after the admin sent this message, but before the admin's message is received, are ignored. The order of messages 1,2 and 3 is not specified but any deterministic order suffices. Finally, C sends message 5 to the group. The final transcript includes messages 1,2,3, seen by A and B as well as message 5 seen by all three users.

5.2.2 Encryption

Similar to other protocols, PGM employs a symmetric ratchet for every user of the group. Figure 5.3 depicts the schematics of a symmetric ratchet. The local state for one user consists of their ratchet, and one for every other member. To send a message the user updates their ratchet. One part of the output is used as an ephemeral key to encrypt the message, and the other part is to update the ratchet state. To decrypt a message the recipient analogously updates the ratchet for the sender of the messages. This works since PGM assumes eventual delivery and ensures a total order for all messages sent by one user.
To add PCS the admin sends regular key updates and also refreshes the key upon group additions or removals. Similar to many other primitives in modern cryptography generating random keys from a truly random seed assumes the existence of a pseudorandom generator (PRG) [26]. Update keys are redacted from messages before being appended to the hash chain. The structure of the hash chain also allows users to sign nodes, such that the chain cannot be altered later on.

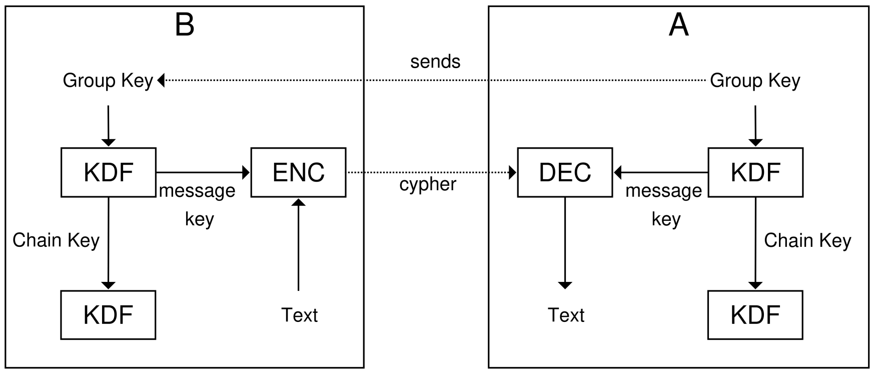

Figure 5.3: Symmetric ratchet used to provide FS in message encryption.

5.2.3 Adaptations to Rollercoaster

Since all members of a group help to distribute messages in the Rollercoaster multicast scheme, all users must have a consistent group view. This is not completely obvious since one might argue that it would be enough if their view is a subset of users. However, Rollercoaster messages do not include any information about the schedule, and thus every user regenerates the schedule before redistributing messages. This issue can be solved by including schedule information, or for the PGM protocol to include the identifier for the head the sender used to calculate the schedule. That way the user distributing the sender's message can calculate the same schedule. Users that did not receive the necessary group membership updates are handled similarly to rollercoaster timeouts, to allow the sender to redistribute the load.

5.3 Prioritizing control messages

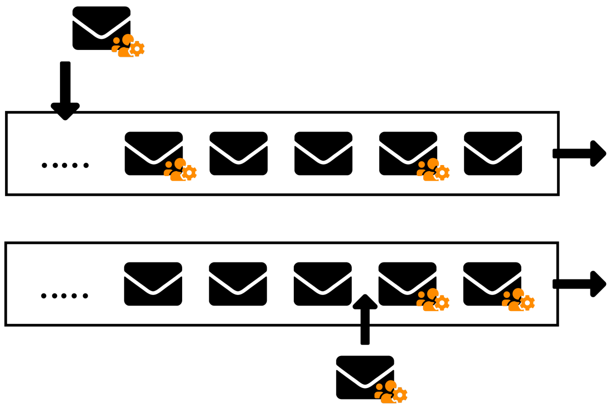

Figure 5.4: Order of control messages in a First-In First-Out (FIFO) outbox (top) versus a priority queue outbox (bottom).

In Loopix users send messages by first adding them to their outboxes. Since message delays are drawn from an exponential distribution this can result in high latencies for messages at the back of the queue, again due to multicast messages congesting the outbox. Since latencies are especially important for control operations to keep group state in sync, Loopix FIFO outbox can be replaced with a priority queue where control messages receive higher priority. This can also be viewed as two separate outboxes, such that control messages are sent, before any payload messages. Figure 5.4 shows how a control message is appended to Loopix's FIFO outbox and how the same message would be inserted further in the front using the priority queue approach.

Chapter 6

Evaluation

This chapter includes a practical and theoretical evaluation of my protocol. The practical evaluation consists of an explanation of the tested scenarios, how latency is measured, implementation aspects that influence latency, and finally an explanation and analysis of the benchmark results. The benchmarks measure different latencies for three different group sizes and models of three different group scenarios derived from the problem statement in Chapter 4.
The theoretical evaluation discusses, how the private group management protocol performs in the properties introduced in Chapter 4 and compares it to related work. I also discuss challenges and open problems.

6.1 Benchmark

The following section includes information about the setup of the simulator, implementation aspects, tested scenarios and group sizes as well as benchmark results.

6.1.1 Setup

For the setup of the simulator [12], I have chosen the standard configuration described in Rollercoaster [11] that satisfies Loopix's [2] privacy requirements. Users pull their inboxes at a rate of $\Delta_\mu = 1/s$ one message per second. Payload, drop, and loop messages for Users are sent at rates drawn from independent and identically distributed exponential random variables with parameters $\lambda_P = \lambda_D = \lambda_L = 2/s$. Mix nodes send loop messages at a rate of $\lambda_L = 2/s$. Users and mix nodes delay messages exponentially distributed with parameter $\lambda_\mu = 3/s$. From the independence of these two random variables follows that the total sending rate of mixes and users equals six messages per

Pull rate (User)	$\Delta_\mu = 1/s$	Rate at which users check the inbox
Payload/drop/loop rate (User)	$\lambda_{P,D,L} = 2/s$	Rate at which users send
Loop rate (Mix)	$\lambda_L = 2/s$	Rate at which mixes send loop messages
Delay rate (Mix and User)	$\lambda_\mu = 3/s$	Rate at which messages are delayed
Message split factor (RC)	$p = 1$	Split factor supports p-restricted multicast
Branch factor (RC)	$k = 2$	Nodes distribute messages to two kids
Timeouts (RC)	None	Re-delivery in the presence of timeouts

Table 6.1: Overview of the chosen configuration parameters for Loopix Users and Mixes as well as Rollercoaster (RC)

second (denote as $\lambda = 6s$). These configuration parameters are presented in Table 6.1. In Rollercoaster p-restricted multicast and split messages are used to increase the performance of the network. However to show viability for a standard Rollercoaster version these features are not used in the benchmark. Another reason for that is that split messages can result in users sending out messages before processing incoming messages which would violate the causal order. Thus p is set to one and so is the split factor. Nodes in the message distribution graph have two children to which they forward messages, which is called a branch factor of $k = 2$ in Rollercoaster. Since offline users do not influence the latency measurements as described later, all users are considered to be online in the simulation and thus timeouts also do not play a role.
All benchmarks span a simulated time of 24 hours with the simulator running at a granularity of 10 milliseconds. Since this refers to simulated time and not real-world elapsed time, the benchmarks can be executed to replicate the results on any machine with sufficient memory (this benchmark used a docker host with 12GB of available main memory) requirements. Network latency is the most interesting metric to look at for the protocol as computing times of typical milliseconds can be neglected for modern mobile clients in comparison to the network latency of seconds in Loopix.
Protocol messages are encoded before being sent across the simulated network to replicate a real-life scenario. As the Loopix simulator did assume shared memory for group state previously, this required some changes to the simulator's underlying architecture. The benchmarks only measure network latency, and thus encryption operations that have not been implemented within the simulator. Shared memory cannot be assumed, as the users are using apps on different devices and there is no server taking on the

role of managing state. Key updates for PCS are also not implemented. This does not influence the measured latency, since key updates are similar to adding or removing members from a group and thus a scenario with a few more group updates would emulate PCS updates. For a similar reason, snapshot nodes are not needed within the benchmark employed, as the only non-constant runtime hash chain operations are querying the group's current users and getting all messages. For querying group members this means that without snapshot nodes all admin messages have to be traversed every time. As mentioned before this is easily circumvented by either adding snapshot nodes, or caching the last result and only query until the head of the previous query.

6.1.2 Scenarios

For the following benchmarks, I consider the three scenarios introduced in Chapter 4. They are defined by two characteristics, first how long it takes for a group to reach its target group size, a phase which I refer to as the ramp-up period, or in other words the relative duration of the invite phase and second, the intensity of membership changes, or in other words how dynamic groups are and how many group membership changes happen.

For group sizes, I consider very small, small, and medium groups with target group sizes of 16, 32, and 64 members. Naturally, the user count of these groups varies time in the simulation, so a group size of 16 refers to an average during the second period of the simulation. Below follows an explanation of how this is modeled to resemble real-life scenarios.

As mentioned above, we need our simulation of group messaging to vary in two parameters, ramp-up time and dynamics. Recall from the simulator configuration that payload messages are sent every 30 seconds. Then dynamics can be modeled as the probability $p_{control}$ that a control event occurs where either one member is added or one is removed. For the ramp-up, we need to calculate the probabilities for add and remove events such that within the specified ramp-up period the specified group user count is reached. This is achieved with the following simple model.

First, I neglect the ramp-up period and assume that a group starts with its targeted user size. Upon sending a control message the admin decides with probability p_{add} to add a new member to the group or remove someone. The probability p_{add} depends on the current size of the group and the target group size such that $p_{add} \in [0.1, 0.9]$. To specify how the group size may vary from its target, p_{add} is increased the fewer users are currently in the group and decreased if there is an overhead in users. The innermost term equals 0.5 iff $target = current$, 0 iff $current = target + band$ and similarly 1 iff $current = target - band$. For all values in between this term is scaled linearly and bound by 0.1 and 0.9:

$$p_{add} \leftarrow min(0.9, max(0.1, 0.5 + \frac{0.5}{band} \times (target - current))) \tag{6.1}$$

To also include ramp-up time δ_{ram-up}, I model the user size growing linearly over time. This is done by adapting the target user size accordingly and works well for longer simulation periods with enough sent group messages.

$$target \leftarrow min(current, current \times \frac{time_{curr}}{\delta_{ramp-up} \times time_{total}}) \tag{6.2}$$

Below I explain the concrete configurations for the following scenarios. For the band parameter, I chose 25% of the target group size. In brackets is the name of how I refer to them in the benchmark figures:

- Heavy-usage groups **(Heavy)**: $p_{control} = 0.2$, $\delta_{ramp-up} = 0.5$
- Low-usage groups **(Low)**: $p_{control} = 0.0625$, $\delta_{ramp-up} = 0.5$
- Invite-heavy groups **(Invite)**: $p_{control} = 0.1$, $\delta_{ramp-up} = 0.05$

6.2 Results

Figure 6.1 depicts how the number of users changes over the simulated time of 24 hours for the aforementioned three scenarios and three group sizes. The rows of this diagram refer to the group sizes of 8, 16, and 32 while the columns denote each of the different scenarios of heavy-usage, low-usage, and invite-heavy groups. Each of the nine subfigures plots group membership changes over time. The x-axis shows the current group size and the y-axis shows the simulated time in hours. Additionally, the dotted red horizontal line depicts the mean group size for the respective plot, while the green dashed horizontal line visualizes the target group size.
The first observation is, how different ramp-up periods manifest. While for the heavy and low scenarios, it expectedly takes roughly 12 hours to first reach the target group size, and the invite scenario takes longer than the expected 1.2 hours since control messages are only sent every 6 minutes. Notice how this behavior for the invite scenario differs by group size, where the target size is reached easier for smaller group sizes. The second aspect to notice is how more dynamic groups have more points scattered and more variance, which also directly follows from the definition of the model. Due to the ramp-up happening faster and almost as many events happening as with the heavy scenario, the invite scenario's mean number of users is the closest to the target for all group sizes. It is interesting how for a group size of 8, the heavy and low scenarios

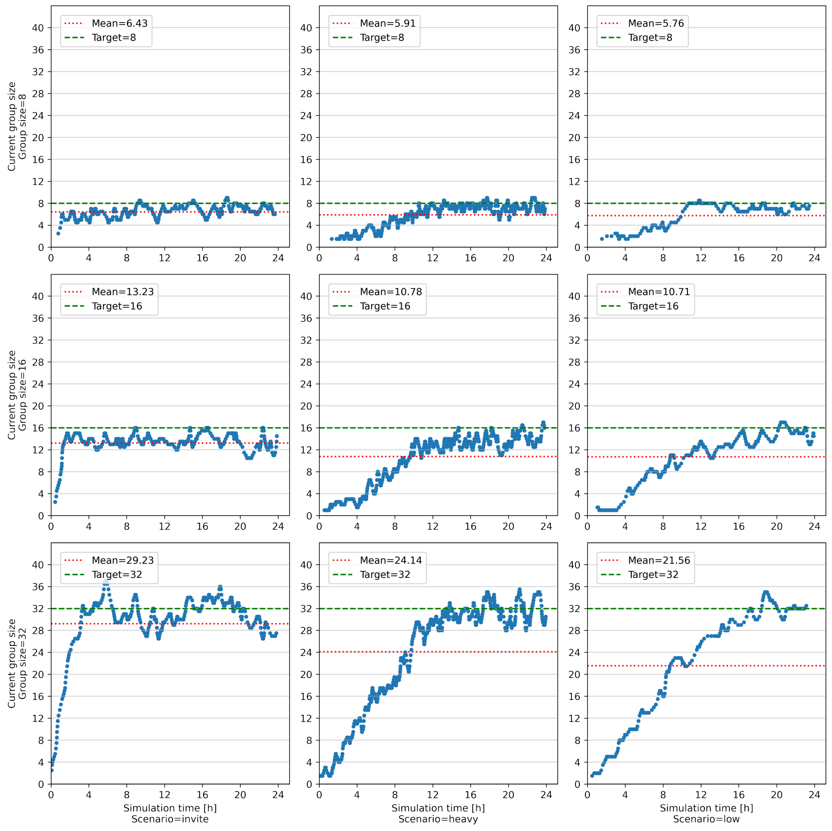

Figure 6.1: Rolling mean number of the last 4 points of users and target number of users over time for each combination of different group sizes and scenarios.

do not differ in their mean number of users and how they do not differ a lot for larger scenarios either.

In Figure 6.2 the latency distributions for all nine cases are plotted. Again, rows of three subfigures correspond to group sizes and columns to the different scenarios. Each subplot shows latency in seconds on the y-axis and the frequency messages with that latency have been sent on the x-axis. Latency is measured as the end-to-end delay from the time a user created the message and put it into their outbox until it is delivered to the recipient's inbox by their provider.
The mean latency is almost equivalent for different scenarios, but not for different group sizes. This is due to broadcasts being less efficient for larger group sizes. This can be observed especially for the p90 and p99 latencies. While the mean only differs in 2 seconds when comparing the group size of 8 with 32, it does by over 3 seconds for p90 and by almost 4 seconds for p99. The frequency of messages also increases which group size, which matches the expectation of larger groups leading to more individual messages.

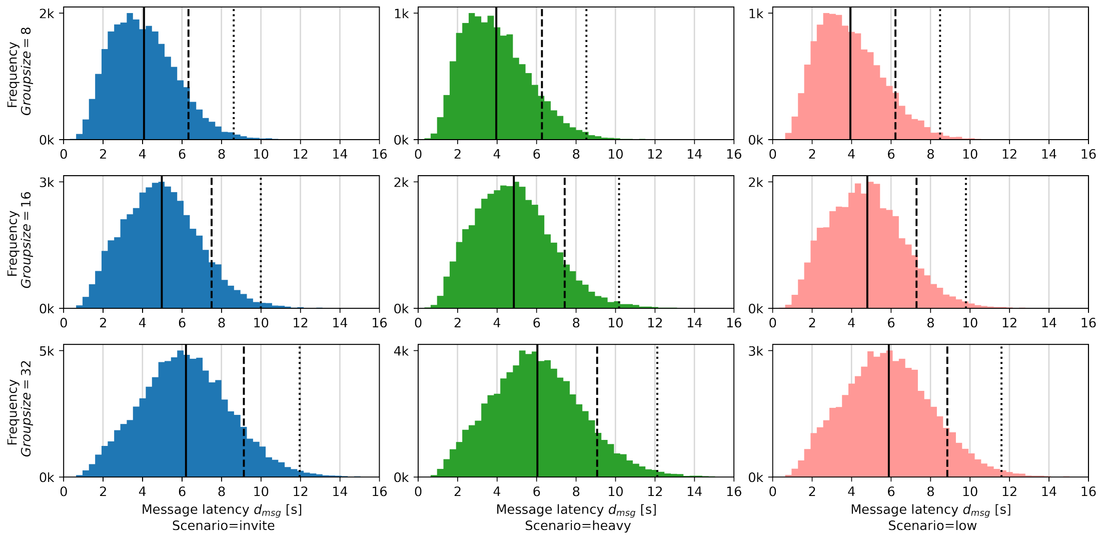

FIGURE 6.2: Latency histograms depicting mean, p95, and p99 latency, for each combination of different group sizes and scenarios including payload and group operations.

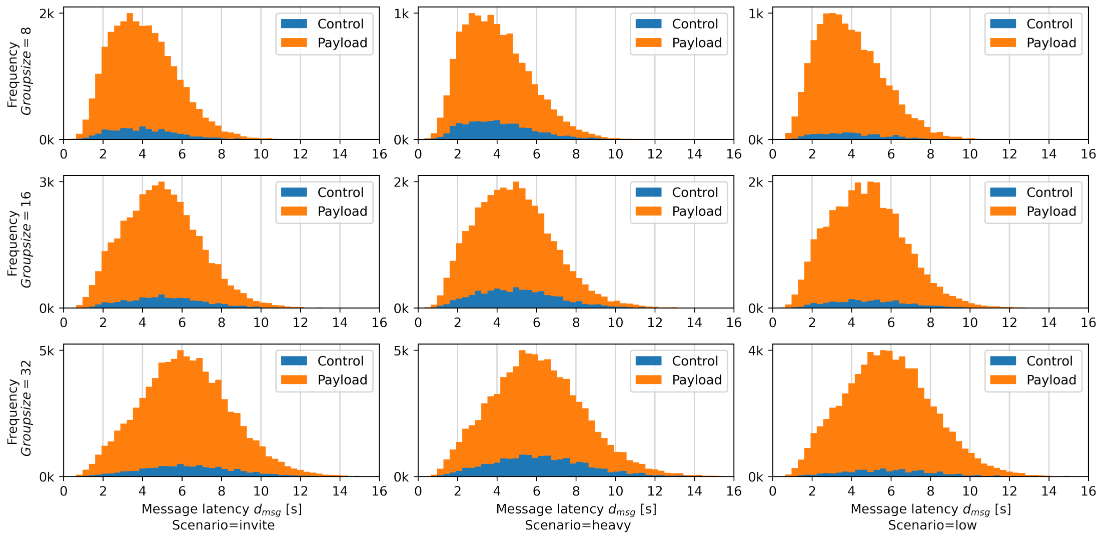

FIGURE 6.3: Stacked latency histograms for control messages (blue) and payload message (orange) for each combination of different group sizes and scenarios.

Figure 6.3 is very similar to Figure 6.2 in that it shows nine latency distributions. In addition, payload messages are visualized in orange and control messages (adds and removes) in blue. The first observation to make is that the share of control messages decreases for the invite and low scenario in comparison to the heavy scenario. The share of control messages is very similar even for different group sizes.

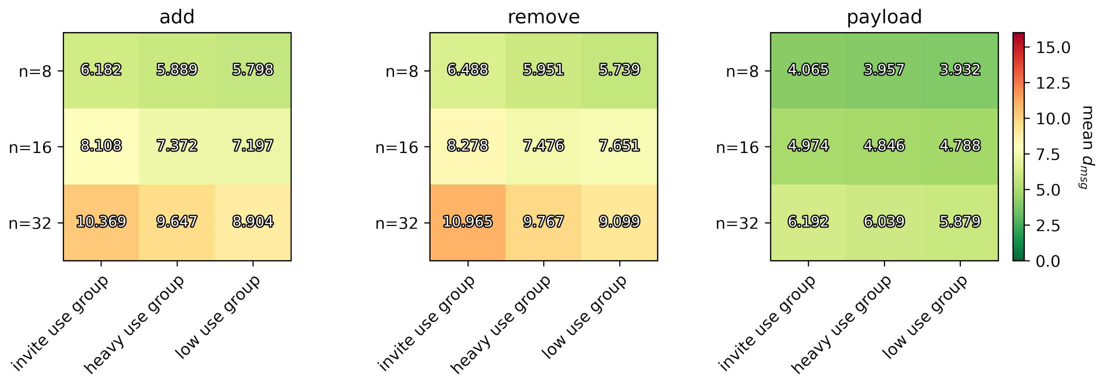

Figure 6.4: For each of the following three operations a heatmap depicting mean latencies in seconds is shown. Each heatmap itself shows the mean latency of every combination of group sizes and scenarios.

Figure 6.4 visualizes latencies for each of the different operations: add, remove, and payload. For the payload case the figure depicts the mean latency. For both control operations, the latency until every group member has received the message is shown. Since more add operations happen during the ramp-up period where the number of users is lower, the latency for the remove case is generally higher. Latencies for invite scenarios are worse than the heavy usage case, which follows from very high latencies during the ramp-up phase.

In Figure 6.5 the x-axis shows 24 hours of simulated time, and the y-axis shows the share of time (in percent) during which the group state was in sync for every member. The red line indicates the mean sync share. If every user would have the same state for the whole simulation the share would equal 100%. Practically, due to network latency this is not achievable. The share is calculated by comparing the delay and received time for every message. First, messages of different types (add, control) are added into one timeline, where overlapping intervals are merged. By doing so the total elapsed time is effectively partitioned into synced time and unsynced time. Using this information I calculate the share of synced state for every transition between the two different possibilities. This share is then used to update the weighted cumulative group state sync. These points are plotted in the figure, such that every point refers to one new unsynced interval and how it changes the share of synced group state.
For the invite scenario the share drops during the ramp-up period as many group messages are being sent and thus they congest the outbox queue for the admin user. This is less severe for the case where control messages are prioritized in the outbox, especially noticeable with a group size of 32 people. This also results in a slightly lower mean.
For the low and heavy scenarios, the share decreases over time. This is due to the ramp-up period and higher latencies with increasing group size. The share stabilizing

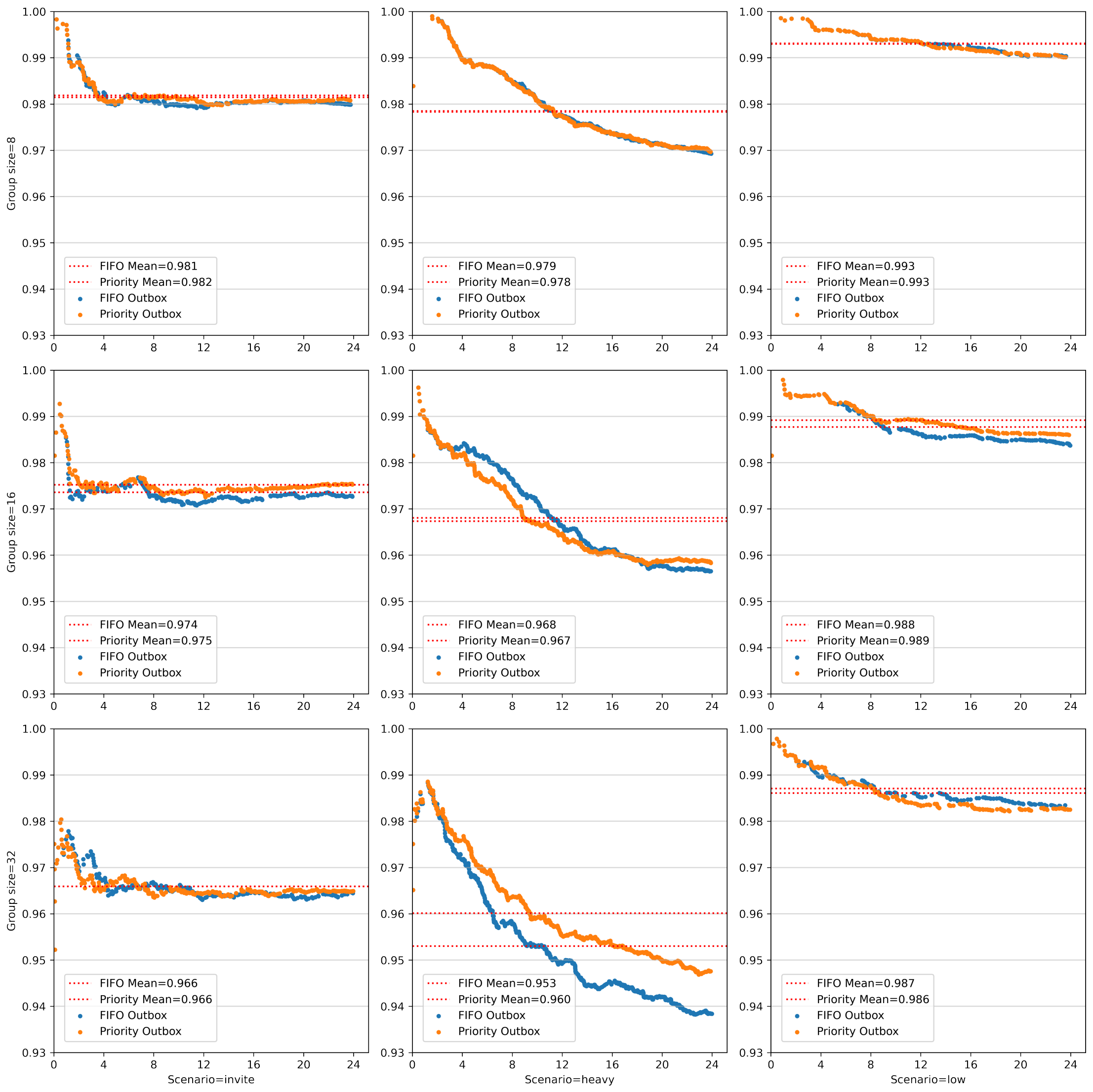

Figure 6.5: Share of state sync over 24 hours of simulation time for all different cases and two different outboxes. The results for using a priority queue are displayed in orange, and the standard FIFO outbox is in blue.

after the ramp-up period confirms this trend. The low-usage scenario is very stable with a mean sync share of around 99% for every group size. Small groups also tend to have a higher sync share as the mean drops from 97.9% to 96.8% for 16 users and to 95.3% for 32 users and behaves similarly for the other scenarios. For the case with heavy usage and 32 members share drop is the most significant, and so is the positive impact of using a priority queue as an outbox. That is a mean share of 95.3% for the FIFO outbox versus a 96% share for the priority queue outbox. For the heavy scenario the share shows a downward trend indicating a potential limit of the PGM protocol.

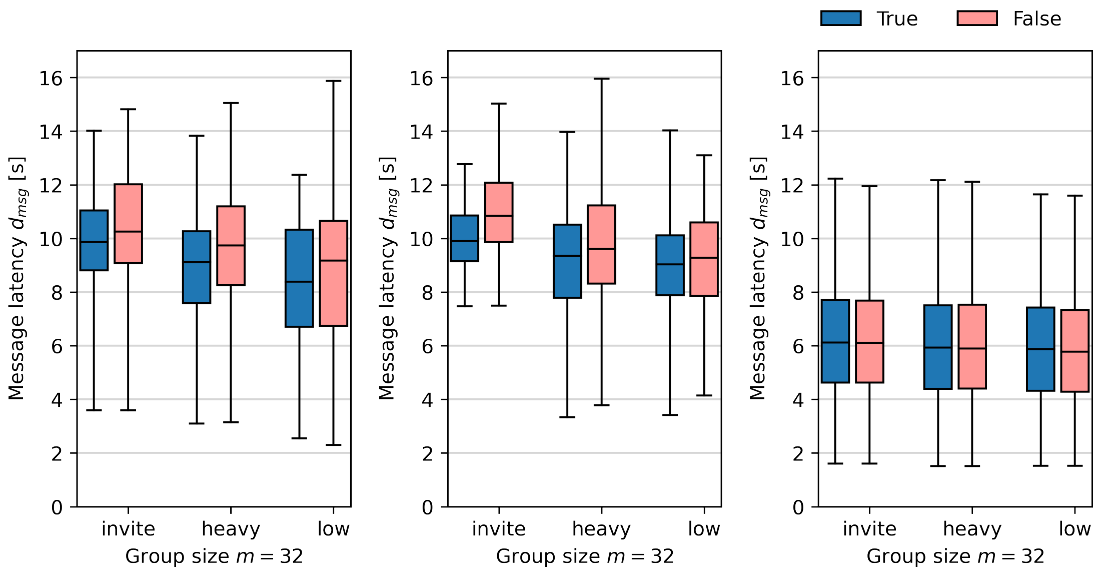

Figure 6.6: Boxplots showing the latency distribution for three different operations. Each subplot shows all six combinations of three scenarios and the choice of outbox. True indicates that a priority outbox and false that a FIFO outbox is used.

Figure 6.6 consists of three subfigures for the following three operations: add, remove, and payload. In each subfigure, six boxplots are displayed for each combination of the three scenarios invite, heavy and low and whether a priority queue or FIFO is used as an outbox. Each boxplot shows the following five latencies in seconds: minimum, p25, mean (p50), p75, and maximum latency. For the payload subfigure, the latency distribution does not differ a lot by the use of the type of outbox. This might be because control and payload messages are only sent together by one sender in the case where the admin sends a message and also decides to add or remove a member. Thus prioritizing control messages increases the latency of payload messages only for a smaller portion of messages, which explains the neglectable difference in mean latency. The biggest difference of one second in p75 latency can be observed for add and remove operations with the heavy and invite scenario. Interestingly, the low scenario has a high maximum latency of almost 16 seconds for the add operation. Overall, the priority queue outbox can be said to improve mean latency for control messages by up to one second, while only marginally influencing the latency for payload messages.

Figure 6.7 shows for a simulated time of 4 hours how the share of group state sync is different for the following three cases: Rollercoaster using a FIFO outbox and a priority outbox, and Sequential Unicast using a FIFO outbox. Observe, how the share stabilizes at 93% for the unicast scenario while this approaches 96% for both Rollercoaster cases.

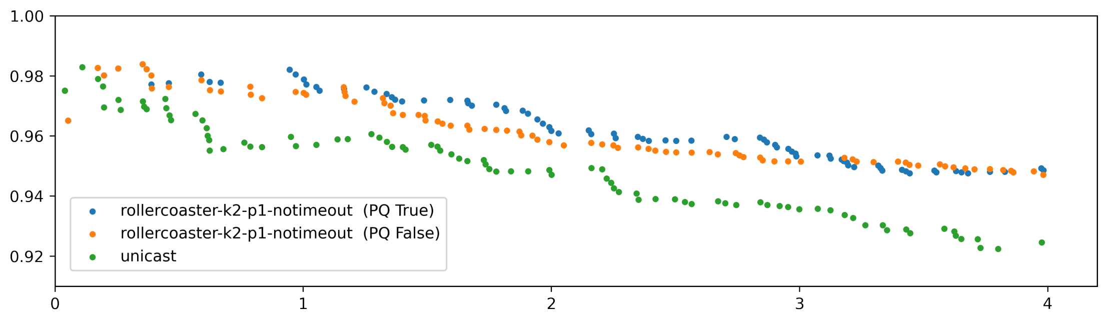

Figure 6.7: Share of state sync over 4 hours of simulation time for Rollercoaster using a priority outbox and FIFO outbox, as well as Sequential Unicast using a FIFO outbox.

This is explained by the mean latency of unicast being almost twice as high as compared to unicast for the heavy scenario with a target group size of 32.

The results of this research demonstrate the practical feasibility of the proposed protocol in addressing the issues of decentralized systems in group messaging. The latency for group control messages is roughly 50% higher compared to the payload for a typical medium group scenario. Since the latency for group operations refers to the time until everyone has the same group state this overhead grows with group size. The share of the time where the group state is synced varies from 96% for heavy-usage groups with 32 members to 99% low-usage groups with 8 members and 99.3% for low-usage groups with 32 members. Invite-heavy scenarios are particularly unsynced during the ramp-up period. All scenarios, but the heavy one, stabilize after the ramp-up period, but then the share slightly decreases with group size, again due to outliers occurring more frequently. The downward trend for the heavy scenario might indicate that sending control messages all three minutes for a group is a limitation, where the sync starts to suffer. Prioritizing group control messages decreases mean latency from 10.9 to 9.9 seconds for the invite-heavy scenario with 32 group members and in the weakest case prioritizing control messages only adds a maximum of 0.1 seconds of mean latency to payload messages in the worst case. The share of 96% state sync for a heavy-usage scenario seems low, it simulated the behavior of sending 24 control messages an hour and this over 24 hours. For scenarios with fewer messages than this, the latency overhead for group state sync will be less dramatic.

6.3 Encryption

This section argues how the PGM protocol can provide encryption guarantees, namely FS and PCS and how it compares to the introduced existing protocols.
As motivated in the introduction, to mitigate the risk of compromised keys for long-term

chats, FS is crucial. The PGM protocol provides FS by using symmetric ratchets to encrypt texts which can be found in many other group protocols. Formally, this primitive has been introduced as Forward-Secure Authenticated Encryption (with Additional Data) [27].
A user that has recently been removed, but has not yet received the remove message from the admin could still send messages that are accepted by other users that also have not received this information, either. Similarly, they could still end up receiving messages that they should not. This only poses a temporary threat as I am assuming eventual delivery.
PCS is provided by having the admin update keys with every control message and regularly every few messages. The root group key is exchanged via pairwise secure channels, a compromise would not lead to long-term compromise, since the admin would send a new update soon. Strictly speaking, this does not guarantee PCS with every message sent, especially when the admin is offline for longer periods but has the benefit of only having to send $n-1$ messages.
This is a weaker security guarantee when compared to DCGKA or Signal it is an improvement upon Sender Keys, Lockboxes, and mpOTR that do not provide PCS at all.

6.4 Challenges

The current approach faces the following challenges that pose interesting areas for future improvements. Users that are offline over long periods still receive messages, even if they will not ever be online again. This results in higher latency for all involved group members. Garbage-collecting offline users could solve this problem. Another issue is the difficulty posed by relying on the admin for federating group state, particularly when the admin is offline for an extended period. The leakage of admin update keys can also pose a significant security threat by compromising the confidentiality of all messages, not just those of the compromised sender in the group.

Additionally, the absence of shared private information among users weakens FS in the case of long-term key leakage. Regular key exchanges are carried out, but not for every message sent which limits the scope of PCS.

Chapter 7

Conclusion

In this work, I present the PGM decentralized private group management protocol and deployed it to an anonymity network. I outline my contributions and potential future research areas in the following conclusion.

7.1 Summary

I explored the challenges of ensuring secure and private group communication in messaging apps. Currently, no group messaging application provides strong metadata privacy guarantees. Anonymity networks can provide unlinkability and unobservability, but it is challenging to deploy a centralized protocol to them. Current anonymity networks like Loopix [2], do not offer efficient broadcasts required for group messaging. The Rollercoaster protocol can be used to achieve this but it does not feature dynamic groups. To overcome these challenges, I propose the PGM protocol that provides FS, as well as PCS and metadata privacy by being deployed to Loopix. This protocol features three central building blocks: a decentralized datatype called hash chain for ordering messages, a forward-secure symmetric ratchet for encrypting each message with a different key, and regular group key updates sent by the group admin. Ordering messages is necessary to decrypt messages with their respective key and to ensure a consistent group view for all members. The PGM protocol includes an adaption to Loopix, where users prioritize control messages to reduce group state latency.
I implemented the PGM protocol and deployed it to a Loopix simulator to run benchmarks with different group scenarios and different group sizes. The results of this research demonstrate that the latency for group operations is roughly 50% higher compared to the payload for a typical medium group scenario. The share of the time where

the group state is synced varies from 96% to 99.3% for the most to the least intensive scenario. Prioritizing group control messages decreases mean latency from 10.9 to 9.9 seconds in the best case while only adding 0.1 seconds of mean latency to payload messages in the worst case.

7.2 Research Questions

The following section answers the research questions that I introduced in Chapter 1

- **Q1 Decentralization**
  What properties does a decentralized group management protocol entail compared to a trusted server?
  For a decentralized group management protocol ensuring message order, synchronizing state synchronization, and federating key updates are important.
- **Q2 Group Management**
  How can a group management protocol on top of mix networks be designed?
  A group management protocol on top of Loopix has to allow for dynamic groups and synchronize the state between users. The PGM protocol introduces hash chain datatype where each user stores all the messages and group updates. In PGM only the admin performs group membership updates.
- **Q3 Metadata Privacy and Security**
  What extent of metadata privacy and message confidentiality can be guaranteed?
  The PGM protocol can guarantee the metadata privacy properties of Loopix. For FS every user employs symmetric ratchets for every other user. Additionally, the admin updates the initial ratchet key to protect against the compromise of long-term keys.
- **Q4 Performance**
  How fast is the PGM protocol theoretically and practically?
  The benchmarks show that for the PGM protocol group latency is roughly 50% higher compared to the payload, the group state sync share varies from 96% to 99.3%, and that prioritizing group control messages can decrease mean latency from 10.9 to 9.9 seconds.

7.3 Future Work

The first interesting avenue to expand this work is to include multi-admin support. This allows multiple admins to manage the group state, including adding and removing members and sending key updates. Multi-admin support should allow admins to change over time. Two benefits of this are that first people are used to having multiple admins, and secondly, this could also mitigate the problems occurring with the admin being offline. Admins still need to be trusted, but having multiple admins is a weaker assumption compared to PGM. One challenge with this approach is conflicting remove operations, e.g. A and B removing each other concurrently. Matrix [28] resolves such conflicts by first applying removals and sorting them by timestamp.

Another avenue for future research is the implementation of multi-device support, allowing users to seamlessly switch between devices while maintaining their group state and accessing previous conversations. This might not even be desired, since having the ability to transfer unecrypted chat histories poses a security risk and sharing ephemeral keys between two devices that are not online at the same time is hard. WhatsApp emulates multi-device support, where online phones act as intermediaries, allowing to use WhatsApp in the browser. Signal allows additional devices to be linked even if the main phone is offline, but this feature is only available on tablets or desktop devices.

Another promising topic for future work is to deploy the PGM protocol to a running mix network. Examples of mix network projects include Nym [29] and Panoramix [30]. While this allows for applications that users can use, it may not provide a much better understanding of the protocol's efficiency compared to the simulated version.

The way PCS updates are performed can be improved. First, everyone uses the same update secret provided by the admin in PGM. This poses an attack vector since in the event of a key compromise messages for every user are affected. Secondly, PCS may be provided with every sent message while not degrading latency too much. DCGKA [17] solves the first problem as described in Chapter 3, but leads to $\mathcal{O}(n^2)$ messages. It would be insightful to observe how this influences the latency for group scenarios. In addition to these protocol changes, an important future addition is to provide formal security games to verify the encryption and metadata claims.

Chapter A

Appendix

A.1 Cryptographic Primitives

This section mentions various cryptographic primitives that are referred to in the building blocks of the protocols. These primitives include symmetric and asymmetric encryption, key exchange protocols, and different ratcheting mechanisms.

Definition 6. *(Symmetric Encryption) Triple of algorithms* (Gen, Enc, Dec) *using the same key to encrypt and decrypt messages s.t.* $Pr_{k \leftarrow Gen()}[Enc(Dec(m,k),k) = t] = 1$ *for all possible keys [26].*

Definition 7. *(Asymmetric Encryption) Triple of algorithms* (Gen, Enc, Dec) *with Gen generating a key-pair, where* K_A,k_A *refers to the public and private key of A, s.t.* $Pr_{k \leftarrow Gen()}[Enc(Dec(m,k_a),K_a) = m] = 1$ *for all possible keys [26].*

With this scheme, A and B can share their respective public keys to encrypt the messages they send. Asymmetric encryption also plays a major role for A and B deriving a shared symmetric key, without exchanging any private information.

Definition 8. *(Authenticated Key Exchange - AKE) A protocol that allows two or more parties to share a common symmetric key and to authenticate their identities. An attacker observing all traffic must not be able to derive private keys from public information. Only involved parties can [31].*

As public key operations can be costly to compute, it is more efficient to first derive a new symmetric key using public key cryptography and then later use this key for encryption. Authentication is necessary to mitigate an attacker in the middle that is imitating A and B to derive shared secrets with them.

Definition 9. *(Message Authentication Code - MAC) Pair of algorithms* $(Tag, Verify)$ *s.t.* $Verify(k, Tag(k, m), m) = True$ *for all keys* k *[26].*

Tag generates a tag given a symmetric key k, and message m. The tag can be used by $Verify$ in combination with the key k and the message m, to verify the message's integrity and authenticity.

Definition 10. *(Authenticated Encryption with Associated Data - AEAD) Pair of symmetric encryption algorithms that use a MAC for authentication. Associated Data is included in the MAC, but not encrypted [26].*

Definition 11. *(CKA) A continuous key agreement is a key exchange protocol that allows participants to update the shared key continuously. This helps to mitigate key compromises [26].*

Definition 12. *(Diffie-Hellman Key Exchange)[31] - A and B agree on some* p *and* g *beforehand. A, B choose random secrets* a *and* b *before sharing* $K_a \equiv g^a \pmod p$ *and* $K_b \equiv g^b \pmod p$ *with each other.*

Due to commutativity A and B can compute the shared private secret $g^{ab} \pmod p$. By just knowing a, b, p, g an attacker cannot efficiently compute the shared private secret. This is referred to as the discrete-logarithm problem and is commonly believed to hold for traditional computers. In the case where A and B can verify each other's identity (e.g. out-of-band, or through a trusted public-key infrastructure) the DH protocol is considered an instance of an AKE protocol.

Definition 13. *(Extended Three Ways Diffie-Hellman Key Exchange (X3DH))*
X3DH [18] provides authenticated key exchange for the asynchronous case.

B is offline when A wants to initiate a key exchange. B has published multiple one-time pre-keys (to mitigate the compromises of long-term keys) as well as an identity key and signed pre-key to verify their identity. A uses B's published keys together with their identity key and an ephemeral public key to compute four DH secrets, which are then concatenated into one shared secret.

Definition 14. *(Key Derivation Function - KDF) A cryptographic hash function is called a KDF when a root key plus some other input are used to derive a new key (that is indistinguishable from random bits). This key is then used for the actual encryption [26].*

If one part of the KDF's key is used as input for a second KDF this constructs a so-called KDF chain.

```
1 (output key, new key)← KDF(input, root key);
2 secret← Enc(text, output key);
3 (output key, new key)← KDF(input, new key);
```

Definition 15. *(Symmetric-key Ratchet) Uses a KDF chain, with constants as input. A symmetric-key ratchet allows for encrypting each message with a new key. This allows message keys to be kept to handle out-of-sync messages [32].*

Definition 16. *(Diffie Hellman Ratchet) A Diffie Hellman Ratchet uses multiple steps of Diffie Hellman key exchange [33]. An attacker stealing A's receiving chains can decrypt all their future messages. To mitigate this A uses a DH ratchet and includes their new public key in every message. Upon receiving B computes the shared secret that is used to derive new receiving and sending chain keys.*

1. B sends A their public key K_B
2. A computes $sk \leftarrow DH(k_A, K_B)$
3. A sends B their public key K_A

A.2 Full Protocol Specification

In the following section, I present the full PGM protocol specification and elaborate on its details. First, I explain the structure of messages and the group state γ. Each user manages a set of groups they are part of.

This section covers group messaging for both the sending and receiving side of messages, encrypting and decrypting messages as well as forwarding the ratchet and updating the ratchet's key. Lastly, it also elaborates on the hash chain operations and how to synchronize the state, get the current set of users, and an ordered list of messages.

The group state γ is defined as a named tuple as follows $\gamma \leftarrow$ (group-id, payload-nonce, admin, user, heads, ratchets, processed-nodes, unprocessed-nodes). The group-id is a random integer identifying the group, the payload-nonce is a counter that is increased upon sending messages. Admin refers to the admin of the group, and user to the group member since γ is local to every member. Heads map every sender to their current head, while ratchets keep track of each user's current sending key. Processed-nodes and unprocess-nodes are used to handle out-of-order messages.

Messages are also named tuples of the following format msg $\leftarrow$ (type, body, prev, sender). Type is a set of possible message types: *Add*, *Remove*, *Update*, *Payload*. The content of the body depends on the type. For *Add/Remove* it has the form of (member, group-key), and (member, nodes, heads, ratchets) for invites. For *Update* the body consists of the new group-key and Payload messages of the actual message content.

Functions take parameters and operate on the group state γ. I use common idiomatic pseudocode constructs with a bit of added notation for increased readability and understandability as defined below:

Named values For a tuple t instead of accessing its first value by t_0 I am referring to the value by the name given by the format t`.nonce`. This is useful while handling γ and message.

Priority queue A priority queue with the following operations `get`, `put`, `empty` and their typical semantics

Dictionaries I am using sets of key-value pairs as a dictionary-like datatype with the notation of accessing the value of a key as follows `dict[key] = value`. This can also be done for multiple keys and is equivalent to having a loop iterate through keys and access them one by one.

A.2.1 Group Messaging

The following section introduces all functions provided to implement a group messaging app. For the sending side, this refers to how the admin can create the group, add and remove members as well as update keys and how users can send messages. The receiving end includes one entry point that handles out-of-order messages and then processes messages based on their type. For each of the types, this section includes a function handling this type of message.

Sender's side

For the side of the sender, the case where the admin sends messages and initializes the state is the most interesting, while for non-admin users sending a message to the group is very simple. This part of the protocol uses `broadcast` provided by Rollercoaster.

```
input: admin, user
1 (ratchets[admin], group-id, payload-nonce) ← (random(), random(), 0);
2 (heads, processed-nodes, unprocessed-nodes) ← (∅, ∅, ∅);
3 γ ← (group-id, payload-nonce, admin, user, ratchets, heads, processed-nodes,
    unprocessed-nodes);
4 return γ;
```

Algorithm 1: `create-group`

```
input: $\gamma$, payload, sender, type, prev
1 $\gamma$.payload-nonce $\leftarrow$ $\gamma$.payload-nonce + 1;
2 prev $\leftarrow$ ($\gamma$.heads[get-users($\gamma$)]);
3 sender $\leftarrow$ $\gamma$.user;
4 msg $\leftarrow$ (payload, sender, type, prev);
  // Send nonce unencrypted, and encrypted message
5 return ($\gamma$.payload-nonce, encrypt(msg, $\gamma$.payload-nonce));
```

Algorithm 2: `create-message`

```
input: $\gamma$
1 key $\leftarrow$ random();
2 $\gamma$ $\leftarrow$ update-ratchet($\gamma$, key);
3 msg $\leftarrow$ create-message($\gamma$, key, UpdateKey);
4 broadcast(get-users($\gamma$), msg);
5 return $\gamma$;
```

Algorithm 3: `update-key`

```
input: $\gamma$, member
  // Send an invite to user
1 update-key $\leftarrow$ random();
2 $\gamma$ $\leftarrow$ update-ratchet($\gamma$, update-key);
3 broadcast(member, create-message($\gamma$, (member, update-key, ($\gamma$.heads, $\gamma$.ratchets)),
   Add));
4 msg $\leftarrow$ create-message ($\gamma$, (member, update-key), Add);
5 broadcast(get-users($\gamma$), msg);
6 return append($\gamma$ redact-keys(msg));
```

Algorithm 4: `add-to-group`

```
input: $\gamma$, member
1 broadcast(member, create-message($\gamma$, (member, 0), Remove));
2 update-key $\leftarrow$ random();
3 $\gamma$ $\leftarrow$ update-ratchet($\gamma$, update-key);
4 msg $\leftarrow$ create-message($\gamma$, (member, update-key), Remove);
5 $\gamma$ $\leftarrow$ append($\gamma$, redact-keys(msg)) broadcast(get-users($\gamma$), msg);
6 return $\gamma$;
```

Algorithm 5: `remove-from-group`

```
input: $\gamma$, body
1 msg $\leftarrow$ create-message($\gamma$, body, Payload);
2 broadcast($\gamma$, msg);
```

Algorithm 6: `send-message`

`create-group`
Initializes all parameters for the group state γ for the respective user.

`create-message`
Create a message for the provided payload. Returns the encrypted payload and the unencrypted nonce.

`update-key`
Lines 1-2: Get a new group key and update the ratchet for the admin.
Line 3-4: Create a key update message and broadcast it to the current group.

`add-to-group`
Lines 1-2: Get a new group key and update the ratchet for the admin.
Line 3: Send the prospective member a direct message with the current state of the hash chain and the new group key.
Lines 4-5: Broadcast the new addition to the rest of the group.
Line 6: Add the new member to the local group state by appending it to the hash chain.

`remove-from-group`
Line 1: Send the removed member a direct message without including the new key.
Lines 2-3: Get a new group key and update the ratchet for the admin.
Line 4-5: Remove the member from the local group state by appending it to the hash chain.
Lines 6: Broadcast the removal to the group without the removed member.

`send-message`
Lines 1-2: Create a message of type payload and broadcast it to the group.

Receivers's side

```
input: $\gamma$, msg
// Ideally, we already received all previous messages
1 if $msg.prev \in \gamma.processed\text{-}nodes$ then
2     msg ← decrypt($\gamma$, msg) on-receive($\gamma$, msg) // Unprocessed node might have
          arrived, reprocess their successors
3     for $successors \in \gamma.unprocessed\text{-}nodes[received\text{-}node]$ do
4         $\gamma$.unprocessed-nodes[received-node] ← $\gamma$.unprocessed-nodes[received-node] \
           successors;
5         receive-message($\gamma$, successors)
6     end
7 else
      // Reprocess nodes as soon as predecessors arrive later
8     $\gamma$.unprocessed-nodes[msg.prev] ← msg;
9 end
```

Algorithm 7: `recieve-message`

```
input: $\gamma$, msg
// Message in its decrypted form
1 if $msg.type = Payload$ then
      // Process payload
2 end
3 if $msg.type = Add$ then
4     $\gamma$ ← on-add($\gamma$, msg);
5 end
// Similarly for on-update-key, on-remove
6 return $\gamma$;
```

Algorithm 8: `on-receive`

```
input: $\gamma$, msg
1 (member, group-key, init) ← msg.body if $\gamma.user = member$ then
      // user is invited to the group
2     (heads, ratchets) ← init ($\gamma.heads$, $\gamma.ratchets$) = (heads, ratchets);
3 else

4 end
5 append($\gamma$, redact-keys(msg));
6 return update-ratchet( $\gamma$, group-key);
```

Algorithm 9: `on-add`

```
input: $\gamma$, msg
1 (member, group-key) $\leftarrow$ msg.body
2 if $\gamma.user = member$ then
      // Remove member from group
3     $\gamma = \emptyset$;
4 else
5     append($\gamma$, redact-keys(msg));
6 end
7 return update-ratchet($\gamma$, group-key);
```

Algorithm 10: `on-remove`

```
input: $\gamma$, key
1 return update-ratchet($\gamma$, key);
```

Algorithm 11: `on-update-key`

`receive-message`
This function is invoked upon the provider delivering a message to the user and handles out-of-order messages. As soon as a message is ready to be processed it is going to be decrypted and then passed to `on-receive`.
Lines 1-2: Handles the case where all predecessors for a message have already arrived and pass the message on.
Lines 3-6: Recursively reprocess nodes that have waited for the current message to arrive.
Lines 7-8: Handles the case where the node waits for predecessors to arrive.

`on-receive`
On receive routes decrypted messages according to their tag. Payload messages might also be processed here to then be displayed to the user.

`on-add`
Line 1: Unpack the msg.body into the information about the member that is added and the new group-key.
Lines 2-4: Handles the case where the recipient is added. For this case, the body also includes init information for the newly added member's state.
Lines 5-6: Handles the case where users add a new member by appending the received message to the hash chain.

`on-remove`
Line 1: Unpack the msg.body into the information about the member that is removed and the new group-key.
Lines 2-4: Handles the case where the recipient is removed from the group.

Lines 5: Handles the case where users remove the member from the group by appending the received message to the hash chain.

`on-update-key`
Updates the key ratchet for all users by invoking *update-ratchet*.

A.2.2 Encryption

The following part of the protocol provides functions to update ratchet keys, as well as to encrypt and decrypt messages and uses an implementation of the Authenticated Encryption scheme [34].

```
input: $\gamma$, msg
// Update the ratchet for the sender of the message
1 (msg-key, $\gamma$.ratchets[msg.sender]) $\leftarrow$ KDF($\gamma$.ratchets[msg.sender]);
2 return ($\gamma$, msg-key)
```

Algorithm 12: `ratchet`

```
input: $\gamma$, update-key
1 $\gamma$.ratchets[get-users($\gamma$)] $\leftarrow$ update-key;
2 return $\gamma$;
```

Algorithm 13: `update-ratchet`

```
input: $\gamma$, msg, nonce
1 (msg-key, $\gamma$) $\leftarrow$ ratchet($\gamma$, msg);
2 return (nonce, AEAD-encrypt(msg, nonce, msg-key))
```

Algorithm 14: `encrypt`

```
input: $\gamma$, msg, nonce
1 (msg-key, $\gamma$) $\leftarrow$ ratchet($\gamma$, msg);
2 return AEAD-decrypt(msg, nonce, msg-key);
```

Algorithm 15: `decrypt`

`ratchet`
Lines 1-2: Derives and returns a message key and derives a new key for the sender's ratchet.

`update-ratchet`
Lines 1-2: Initializes the ratchet with a new update key for all users.

`encrypt/decrypt`
Lines 1-2: Gets an ephemeral message key from the ratchet and returns the AEAD encrypted ciphertext, or the decrypted plaintext respectively.

A.2.3 Hash Chain

The following section includes detailed constructions as well as explanations for the hash chain operations. This includes querying the set of current users, getting a list of messages, and appending new messages to the chain.

```
input: $\gamma$
users ← $\emptyset$ ;
curr ← $\gamma$.heads[$\gamma$.admin];
while curr do
    member ← msg.body[0];
    if curr.msg.type = Add then
        users ← curr.msg.bod ∪ users;
    end
    if curr.msg.type = Remove then
        users ← users \ {curr.msg.member};
    end
    if curr.prev then
        curr ← curr.prev[$\gamma$.admin];
    end
end
return users;
```

Algorithm 16: `get-users`

```
input: $\gamma$, msg, prev
prev-map = $\emptyset$;
for node ∈ prev do
    prev-map[node.msg.sender] = node;
end
if msg.type ∈ (Add, Remove, UpdateKey) then
    $\gamma$.heads = $\emptyset$;
    // Reset heads
end
node ← (hash(msg, prev), prev-map, msg);
$\gamma$.heads[msg.sender] ← node;
return $\gamma$;
```

Algorithm 17: `append`

```
input: $\gamma$, node, until
// Ignores messages sent after a merge by only considering
   predecessors
(history, to-search) $\leftarrow$ ($\emptyset$, $\emptyset$);
// Priority is based on session first then timestamp
put((to-search, 0, node));
while not empty(to-search) do
    curr-sess, curr-node $\leftarrow$ get(to-search);
    if curr-node $\notin$ history // Add node to history if not yet discovered
     then
        history $\leftarrow$ history $\cup$ curr-node;
        if curr-node = until // Found node thus return the message history
         then
            return history;
        end
        if curr-node.msg.type = Payload then
            // Create new session if prev node was an admin merge
            if curr-node.prev.msg.type $\in$ (Add, Remove, UpdateKey) then
                // Only one previous node since it is a payload node
                put(to-search, (curr-sess + 1, curr-node.prev));
                continue;
            end
        end
        for node $\in$ curr-node.prev do
            put(to-search, curr-sess, node);
        end
    end
end
// No new messages found
return $\emptyset$;
```

Algorithm 18: `search`

```
input: $\gamma$, until
1 first-admin-node ← $\gamma$.heads[$\gamma$.admin];
2 while first-admin-node do
3     if first-admin-node.type ∈ (Add, Remove, UpdateKey) then
4         break;
5     end
6     first-admin-node ← first-admin-node.prev[$\gamma$.admin];
7 end
8 return search(first-admin-node, until);
```

Algorithm 19: `get-messages`

`get-users`
Linearly traverses all messages sent by the admin and executes add/remove operations. Upon sending messages to a group this procedure is invoked. To circumvent its runtime to grow linearly with the total number of messages sent, it is simple to store previous results and their respective head node. This allows to only compute new changes and can be seen as some sort of snapshot mechanism. Additionally, control nodes could include pointers to previous control messages.
Lines 1-2: initialize the set of users and the current node as the admin's head.
Lines : Curr always points to the current admin node.*Lines 4-9:* Adds or removes the respective user to or from the set of users and ignores other messages.
Lines 11-13: Traverese to the admin's previous operations.

`append`
Lines 1-4: Map nodes to the user that sent the message.
Lines 5-7: Since the new message is a merge the heads for all users are reset.
Line 8: Creates a new node for the provided message and its predecessors. Hash the message content and predecessors to obtain a unique id for the node. This could include a digital signature.
Line 9: Updates the head for the sender to the newly created node.

`search`
Traverses the graph as follows. Consider all messages sent between two admin control messages as one session. Messages are then ordered first by session and second by their sent date. This means that all nodes of the current session are traversed first. The second property can be swapped as long as it is deterministic for all users to ensure transcript consistency and does not violate causal order. At some point the until node is found or the whole graph traversed. Users store their last processed admin head and the value to avoid reprocessing the whole message graph every time.
Lines 1-2: Initializes the message history as an empty list, the to-search as a priority

queue and adds the first node to search with session 0.
Lines 3-7: While there are existing nodes to search for, append them to the history in that order if they have not been seen before.
Lines 8-11: Stop the search and return the history of nodes if the until the node is found.
Lines 12-17 Ensures that all messages sent between two control operations are processed before proceeding to the next session.
This is achieved by increasing the session if the previous node contains a control message. Note that the session depends on the priority of the current node. Using session as the priority parameter ensures processing all messages of the current session before moving to the new session.
Lines 18-20: Append all previous nodes for the current node.

`get-messages`
Finds the first control node created by the admin as a starting point and passes it to `search`.

A.2.4 Snapshots

Users might request *get-users* and *get-messages* quite often and would naively traverse the whole graph. To avoid linear complexity in the number of messages it is easy to adapt the protocol to use checkpoints. In the case of *get-messages* this is as simple as passing the last checkpoint as the until parameter. The *get-users* function is also designed such that adding a similar mechanism is straightforward.

A.3 List of acronyms

E2E	End-to-End
FS	forward secrecy
PKI	Public-Key Infrastructure
PCS	post-compromise security
GPA	Global Passive Adversary
DCGKA	Decentralized Continuous Group Key Agreement
mpOTR	Multi-party Off-the-Record Messaging
OTR	Off-the-Record protocol
DH	Diffie Hellman
PGM	Private Group Management
FIFO	First-In First-Out

Bibliography

[1] M. Chase, T. Perrin, and G. Zaverucha, *The Signal Private Group System and Anonymous Credentials Supporting Efficient Verifiable Encryption*, Cryptology ePrint Archive, Paper 2019/1416, `https://eprint.iacr.org/2019/1416`, 2019. DOI: `10.1145/3372297.3417887`. [Online]. Available: `https://eprint.iacr.org/2019/1416`.

[2] A. M. Piotrowska, J. Hayes, T. Elahi, S. Meiser, and G. Danezis, "The Loopix Anonymity System", in *26th USENIX Security Symposium (USENIX Security 17)*, Vancouver, BC: USENIX Association, Aug. 2017, pp. 1199–1216, ISBN: 978-1-931971-40-9. [Online]. Available: `https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/piotrowska`.

[3] D. L. Chaum, "Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms", *Communications of the ACM*, vol. 24, no. 2, pp. 84–90, Feb. 1981. DOI: `10.1145/358549.358563`. [Online]. Available: `https://doi.org/10.1145/358549.358563`.

[4] A. Pfitzmann and M. Köhntopp, "Anonymity, Unobservability, and Pseudeonymity — a Proposal for Terminology", in *International Workshop on Designing Privacy Enhancing Technologies: Design Issues in Anonymity and Unobservability*, Berkeley, California, USA: Springer-Verlag, 2001, 1–9, ISBN: 3540417249.

[5] A. Serjantov and G. Danezis, "Towards an Information Theoretic Metric for Anonymity", in *Privacy Enhancing Technologies*, R. Dingledine and P. Syverson, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 41–53, ISBN: 978-3-540-36467-2.

[6] C. Boyd and K. Gellert, "A Modern View on Forward Security", *The Computer Journal*, vol. 64, no. 4, pp. 639–652, Aug. 2020, ISSN: 0010-4620. DOI: `10.1093/comjnl/bxaa104`. eprint: `https://academic.oup.com/comjnl/article-pdf/64/4/639/37161647/bxaa104.pdf`. [Online]. Available: `https://doi.org/10.1093/comjnl/bxaa104`.

[7] 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.

[8] M. Reed, P. Syverson, and D. Goldschlag, “Anonymous connections and onion routing”, *IEEE Journal on Selected Areas in Communications*, vol. 16, no. 4, pp. 482–494, 1998. DOI: 10.1109/49.668972.

[9] A. Serjantov, R. Dingledine, and P. Syverson, “From a Trickle to a Flood: Active Attacks on Several Mix Types”, in *Information Hiding*, F. A. P. Petitcolas, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 36–52, ISBN: 978-3-540-36415-3.

[10] G. Danezis and I. Goldberg, *Sphinx: A compact and provably secure mix format*, Cryptology ePrint Archive, Paper 2008/475, https://eprint.iacr.org/2008/475, 2008. [Online]. Available: https://eprint.iacr.org/2008/475.

[11] D. Hugenroth, M. Kleppmann, and A. R. Beresford, “Rollercoaster: An Efficient Group-Multicast Scheme for Mix Networks”, in *30th USENIX Security Symposium (USENIX Security 21)*, USENIX Association, Aug. 2021, pp. 3433–3450, ISBN: 978-1-939133-24-3. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/hugenroth.

[12] ——, *Rollercoaster Simulation*, Accessed on 2022-09-09, 2021. [Online]. Available: https://github.com/lambdapioneer/rollercoaster/tree/main/simulation.

[13] M. Marlinspike, *Private Group Messaging*, Accessed on 2022-09-12, 2014. [Online]. Available: https://signal.org/blog/private-groups/.

[14] M. Platforms, *WhatsApp Security Whitepaper*, Acessed on 2023-01-29, 2023. [Online]. Available: https://www.whatsapp.com/security/WhatsApp-Security-Whitepaper.pdf.

[15] I. Goldberg, B. Ustaoğlu, M. D. Van Gundy, and H. Chen, “Multi-Party off-the-Record Messaging”, in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, ser. CCS ’09, Chicago, Illinois, USA: Association for Computing Machinery, 2009, 358–368, ISBN: 9781605588940. DOI: 10.1145/1653662.1653705. [Online]. Available: https://doi.org/10.1145/1653662.1653705.

[16] H. Caudill, *Local First Auth*, Accessed on 2022-09-27, 2022. [Online]. Available: https://github.com/local-first-web/auth/blob/main/docs/internals.md.

[17] M. Weidner, M. Kleppmann, D. Hugenroth, and A. R. Beresford, “Key Agreement for Decentralized Secure Group Messaging with Strong Security Guarantees”, in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and*

Communications Security, ser. CCS '21, Virtual Event, Republic of Korea: Association for Computing Machinery, 2021, 2024–2045, ISBN: 9781450384544. DOI: 10.1145/3460120.3484542. [Online]. Available: https://doi.org/10.1145/3460120.3484542.

[18] T. Perrin and M. Marlinspike, *The X3DH Key Agreement Protocol*, Acessed on 2023-01-01, 2016. [Online]. Available: https://signal.org/docs/specifications/x3dh/x3dh.pdf.

[19] ——, *The Double Ratchet Algorithm*, Acessed on 2023-01-01, 2016. [Online]. Available: https://signal.org/docs/specifications/doubleratchet/doubleratchet.pdf.

[20] M. Naor, Y. Naor, and R. Omer, *Applied Kid Cryptography*, Acessed on 2023-01-12, 1999. [Online]. Available: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=000c55c1b4ad3cf0ef91867b8e8e7a8d6150ee47.

[21] M. R. Albrecht, J. Blasco, R. B. Jensen, and L. Mareková, "Collective Information Security in Large-Scale Urban Protests: the Case of Hong Kong.", in *USENIX Security Symposium*, 2021, pp. 3363–3380.

[22] D. Balbas, D. Coliens, and G. Phillip, *Analysis and Improvements of the Sender Keys Protocol for Group Messaging*, Acessed on 2023-01-01, 2022. [Online]. Available: https://davidbalbas.github.io/files/Sender_Keys_RECSI2022.pdf.

[23] D. D. Clark and D. R. Wilson, "A comparison of commercial and military computer security policies", in *1987 IEEE Symposium on Security and Privacy*, IEEE, 1987, pp. 184–184.

[24] *Signal Messenger Features - Linked Devices*, Acessed on 2023-02-04. [Online]. Available: https://support.signal.org/hc/en-us/articles/360007320551-Linked-Devices.

[25] *Keybase Signature Chain*, Accessed on 2023-02-07, 2023. [Online]. Available: https://book.keybase.io/docs/teams/sigchain.

[26] J. Katz and Y. Lindell, in *Introduction to modern cryptography: Principles and protocols*. CRC Press, 2007.

[27] 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, https://eprint.iacr.org/2018/1037, 2018. [Online]. Available: https://eprint.iacr.org/2018/1037.

[28] *Matrix*, Accessed on 2023-02-09, 2023. [Online]. Available: https://spec.matrix.org/latest/.

[29] *Nym Mixnet*, Accessed on 2023-02-08, 2023. [Online]. Available: https://nymtech.net/.

[30] *Panoramix*, Accessed on 2023-02-08, 2023. [Online]. Available: `https://panoramix-project.eu/`.

[31] W. Diffie, P. C. V. Oorschot, and M. J. Wiener, "Authentication and authenticated key exchanges", *Designs, Codes and Cryptography*, vol. 2, no. 2, pp. 107–125, Jun. 1992. DOI: `10.1007/bf00124891`. [Online]. Available: `https://doi.org/10.1007/bf00124891`.

[32] M. Abdalla and M. Bellare, "Increasing the Lifetime of a Key: A Comparative Analysis of the Security of Re-keying Techniques", in *Advances in Cryptology — ASIACRYPT 2000*, T. Okamoto, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 546–559, ISBN: 978-3-540-44448-0.

[33] N. Borisov, I. Goldberg, and E. Brewer, "Off-the-Record Communication, or, Why Not to Use PGP", in *Proceedings of the 2004 ACM Workshop on Privacy in the Electronic Society*, ser. WPES '04, Washington DC, USA: Association for Computing Machinery, 2004, 77–84, ISBN: 1581139683. DOI: `10.1145/1029179.1029200`. [Online]. Available: `https://doi.org/10.1145/1029179.1029200`.

[34] P. Rogaway, "Authenticated-Encryption with Associated-Data", in *Proceedings of the 9th ACM Conference on Computer and Communications Security*, ser. CCS '02, Washington, DC, USA: Association for Computing Machinery, 2002, 98–107, ISBN: 1581136129. DOI: `10.1145/586110.586125`. [Online]. Available: `https://doi.org/10.1145/586110.586125`.