Design of an Efficient Distributed Delivery Service for Group Key Agreement Protocols

Ludovic Paillat, Claudia-Lavinia Ignat, Davide Frey, Mathieu Turuani, Amine Ismail

Design of an Efficient Distributed Delivery Service for Group Key Agreement Protocols

Ludovic Paillat[1][0000−0002−5428−8834], Claudia-Lavinia Ignat[2][0000−0002−0487−6128], Davide Frey[3][0000−0002−6730−5744], Mathieu Turuani[2], and Amine Ismail[1]

[1] Hive Computing Services, Cannes, France
{ludovic.paillat,amine.ismail}@hivenet.com
[2] Université de Lorraine, CNRS, Inria, LORIA, Nancy, France
{claudia.ignat,mathieu.turuani}@inria.fr
[3] Inria, IRISA, CNRS, Université de Rennes, France
davide.frey@inria.fr

Abstract. End-to-end encrypted messaging applications such as Signal became widely popular thanks to their capability to ensure the confidentiality and integrity of online communication. While the highest security guarantees were long reserved to two-party communication, solutions for n-party communication remained either inefficient or less secure until the standardization of the MLS Protocol (Messaging Layer Security). This new protocol offers an efficient way to provide end-to-end secure communication with the same guarantees originally offered by the Signal Protocol for two-party communication. However, both solutions still rely on a centralized component for message delivery, called the Delivery Service in the MLS Protocol. The centralization of the Delivery Service makes it an ideal target for attackers and threatens the availability of any protocol relying on MLS. In order to overcome this issue, we propose the design of a fully distributed Delivery Service that allows clients to exchange protocol messages efficiently and without any intermediary. It uses a Probabilistic Reliable-Broadcast mechanism to efficiently deliver messages and the Cascade Consensus Protocol to handle messages requiring an agreement. Our solution strengthens the availability of the MLS Protocol without compromising its security.

Keywords: Distributed systems · Group key agreement · Consensus protocols · Reliable broadcast.

1 Introduction

To protect the privacy of their users, a number of Internet-based services have started to develop solutions based on *end-to-end encryption (E2EE)* that prevent third parties from accessing user data transferred from one endpoint to another.

Secure messaging applications such as Signal and WhatsApp are well-known to advertise their use of *E2EE*. Indeed, the Signal Protocol was the first to propose *E2EE* for two-party conversations using the Double Ratchet algorithm [18].

However, the solutions proposed by these same messaging applications for n-party secure communication were either inefficient by requiring the establishment of encrypted communication between all pairs of group members, or less secure when using a common encryption key (e.g. Sender Keys Protocol [5]) which may not be refreshed as the group dynamically changes.

To address the topic of secure group communication, industrial and academic organizations such as Cisco, Mozilla, Facebook and Inria proposed the Messaging Layer Security Protocol (MLS) standardized as RFC 9420 [6]. This protocol relies on a Group Key Agreement Protocol called TreeKEM [9] allowing members of a group to derive a common secret called *group key* which serves as a basis to secure group communications. It is scalable in terms of the number of operations modifying the group and it supports periodic group-key renewals preventing compromised communication.

The MLS Protocol offers an efficient solution to guarantee the *confidentiality* and *integrity* of communication. However, the *availability* of the protocol depends on the *Delivery-Service* component, which remains centralized most of the time. The centralization of this component makes it an ideal target for attackers who wish to disrupt communication. Notably, with the help of a compromised Delivery Service, an attacker can prevent group members from refreshing their keys and resolving the compromise.

In order to overcome these limitations we propose a fully distributed Delivery Service. It combines two distributed communication mechanisms adapted to the need of the messages exchanged by the protocol. We use a Probabilistic Reliable Broadcast mechanism [13] to reliably deliver messages allowing users to propose changes to the group (i.e. *Proposal* messages) and the Cascade Consensus Protocol [1] to deliver the messages that actually modify the group (i.e. *Commit* messages) and thus require an agreement between members.

Our contribution is two-fold:

- the formalization of the MLS protocol's Delivery Service, detailing the necessary properties of this component;
- a novel algorithm describing a fully distributed Delivery Service.

We start by reviewing the state of the art of distributed communication mechanisms in Section 2. We then present the TreeKEM Protocol and formalize the Delivery Service in Section 3. Section 4 presents the details of our solution and discusses how it increases the security of the protocol with respect to a centralized approach. Finally, Section 5 concludes the paper and presents some future work directions.

2 Related work

Secure communication requires securely sharing encryption keys thus preventing attackers from gaining access to them. Additionally, as communications can remain established for a long time, protocols must provide a way to mitigate the eventual compromise of some keys. The following security properties need to be ensured:

- *Backward secrecy*: the knowledge of previous keys does not affect the security of the current key and future ones.
- *Forward secrecy*: the knowledge of a key does not affect the security of previous keys.
- *Post-Compromise security*: if a member is compromised, an UPDATE operation from this member will resolve the compromise and restore the security of the group in subsequent states.

The Signal protocol based on the Double Ratchet Algorithm [18] provides strong security guarantees for two-party communication. The Double Ratchet algorithm provides *Forward secrecy* by generating a new encryption key for each message while periodical Diffie-Hellman key exchanges provide fresh security material ensuring *Post-Compromise security*. However, this approach cannot be applied to group (i.e. n-party) communication and the use of pairwise communication channels (i.e. a group member sends a message to the group by using $n-1$ secure communication channels established with the other members) does not scale well.

An alternative, *Sender Keys* [5], allows one member to use pairwise communication channels to share a common encryption key that can be used to encrypt messages. While this key can be used to deterministically generate individual *message keys* to provide *Forward Secrecy*, *Post-Compromise security* is not ensured as the common key is only renewed in rare occasions such as member removal. Thus, the compromise of one member makes it possible to spy on future messages for a long time.

Secure group-communication protocols address these drawbacks. In the following, we first present the main existing protocols for secure group communication and we highlight the advantages of TreeKEM [9]. We then describe the distributed communication mechanisms we selected to make it distributed.

2.1 Secure Group Communication

First attempts to secure group communication were Conference Key Distribution Systems in which a member generates a conference key and distributes it to all group members. Different topologies for Distribution Systems [11] were proposed. However the *Star-Based* topology is inefficient as it requires $\mathcal{O}(n)$ key exchanges, while other topologies such as a *Tree* or a *Cyclic System* cannot tolerate the fault of even one participant.

Group Key Agreement protocols were introduced as a way to establish and manage dynamic groups whose members can derive a common encryption key called *group key*. These protocols mainly provide three operations: the ADD and REMOVE operations to manage the group and the UPDATE operation to allow group members to refresh their secret keys. Each operation leads to a new group key, thereby ADD and REMOVE operations guarantee *backward* and *forward secrecy* for the group key while periodical UPDATEs ensure *post-compromise security*. Thus, if a member does not renew its secret key, this member should be

removed from the group as a compromise of this member threatens the security of the group.

Collaborative Group Key Agreement protocols were the first to establish a *group key* built on the principle of the Diffie-Hellman key exchange. The GDH protocols [4,20] allow members to collaboratively build the group key presented in Equation 1. However, the computation of this key requires $\mathcal{O}(n)$ communication rounds. The TGDH (Tree-based Group Key Agreement) protocol [14] establishes a group key more efficiently in $\mathcal{O}(\log n)$ rounds. The underlying binary-tree structure contains the member keys in the leafs, intermediate keys exchanged between sub-groups (i.e. members sharing one node in their path to the root of the tree) and the root representing the group key. The resulting group key for a 5-member group is presented in Equation 2. Additionally, operations on the group only require the modification of $\mathcal{O}(\log n)$ intermediate keys.

$$K = \alpha^{r_1 \times r_2 \times \cdots \times r_n} \qquad (1) \qquad K = \alpha^{\left(\alpha^{\left(\alpha^{r_1 \times r_2}\right) \times \left(\alpha^{r_3 \times r_4}\right)}\right) \times r_5} \qquad (2)$$

Nevertheless, the TGDH protocol cannot tolerate the fault or disconnection of any member, as each member can only modify their path to the root of the tree. Thus, the removal of one member from the tree requires the participation of the closest neighbor to this member in the tree. The ART (Asynchronous Ratcheting Trees) protocol [12] enhances TGDH with the capability of creating a group in which all members but the group creator are not required to be online. In this protocol, the group creator can create the initial tree by using the X3DH key exchange algorithm [16] with ephemeral keys stored by each group member in a Public Key Infrastructure. This key exchange guarantees that each group member will be able to derive only the keys they should know with the exception of the group creator who initially knows the entire tree.

The more recent TreeKEM protocol [9] allows clients to issue asynchronous group operations. By replacing the tree structure based on Diffie-Hellman with a tree structure based on the principle of *Key Derivation*, the TreeKEM protocol allows any member to carry out operations on the tree without requiring the help of any particular group member. The TreeKEM protocol achieves good performance and was backed by multiple security analyses (i.e. [3], [10], ...). It is part of a complete solution for Secure Group Communication, the MLS Protocol [6] standardized in RFC 9420. For all these reasons we rely on TreeKEM as a basis for our proposed distributed Group Key Agreement mechanism.

2.2 Distributed Communication Mechanisms

We extend TreeKEM by focusing on two distributed communication mechanisms adapted for the delivery of Proposal and Commit messages as described in Section 3.1.

In the case of Proposal messages, which are sent by members only to propose modifications to the group, and thus do not require agreement, we use a Reliable Broadcast protocol. This protocol only focuses on the correct delivery of a message from a specific sender to all group members.

Specifically we adopt a gossip-based Scalable Reliable Broadcast protocol [13] with a $\mathcal{O}(\log(n))$ per-process communication and computation complexity where n is the number of processes. More precisely, we adopt MURMUR [13], a gossip-based dissemination protocol which only guarantees the delivery of messages without the ability to provide an order of messages for a given member. This protocol perfectly fits the requirements for Proposal messages as these messages do not need to be ordered in any way.

For dealing with Commit messages, which need to be totally ordered and thus require agreement between group members, we use the Cascade Consensus Protocol [1]. Below, we first provide a description of the Context-Adaptative Cooperation abstraction which is an important part of Cascade Consensus and then we detail the Cascade Consensus protocol.

Context-Adaptative Cooperation. Context-Adaptative Cooperation (CAC) is a new broadcast abstraction that sits between reliable broadcast and consensus [1]. It allows multiple senders to send concurrent messages and focuses on the ability to detect when some messages are indeed sent concurrently and conflict with each other.

In CAC, contrary to classical reliable broadcast protocols, the protocol will not only deliver a message but also a *conflict-set* indicating other messages that might conflict with the delivered one. Therefore, CAC can act as a reliable broadcast when there is not any conflict during a broadcast instance. Otherwise, in the case of a conflict between multiple senders, this conflict will be detected and CAC can trigger a classical consensus protocol to handle the conflict.

The protocol works by collecting signatures from other processes about the different broadcast messages. When a process broadcasts a protocol message to others, this process includes the signed messages received from other processes. The use of signatures prevents a malicious process from presenting different views to other processes as processes share their own views by means of these signed messages. Then, after reaching a sufficient count of signatures, the protocol can move forward with the associated messages. The CAC protocol is organized in two phases (Witness and Ready) detailed below.

In the *Witness* phase, each process broadcasts a WITNESS message targeting the first broadcast message that this process received. After receiving WITNESS signatures from a quorum of q_W processes, the current process can move on to the next phase for this given message. However, in the event of multiple broadcast messages, it is possible that none reaches the quorum of q_W processes. Thus, an *unlocking mechanism* is triggered when a process knows that it received a response from at least $n - t$ processes, n being the total number of processes with at most t Byzantine processes. In that case this process will also send a WITNESS message for all witnessed messages that satisfy a lower threshold. These messages are likely to conflict and appear in the *conflict-set* that will later be computed. The size of the quorum $q_W = 2t + k$ is based on a parameter k that tunes the sensitivity of the protocol to conflicting messages: a small k can

lead to multiple conflicts whereas with a larger k fewer messages will likely reach the quorum.

The *Ready* phase is reached for a given message when this message reaches the quorum of q_W distinct WITNESS signatures. Then, the process can broadcast a READY message and wait for q_R distinct READY signatures for the same message. The purpose of this phase is to ensure *totality* meaning that if the current process delivers the message then all the other correct processes also deliver it. Finally, as mentioned earlier, CAC associates the delivered messages with a *conflict-set*. This *conflict-set* contains messages that received enough WITNESS signatures to be considered in conflict.

Cascade Consensus Protocol. The CAC abstraction enables the construction of a consensus protocol named *Cascade Consensus*. In this protocol, a process starts by using CAC to broadcast a message. Then, if there are no conflicts, meaning that CAC delivers a single message with a *conflict-set* of size 1, the protocol can directly finish and deliver this message. Otherwise in case of a conflict, with a *conflict-set* containing two messages or more, the protocol triggers a *Restrained Consensus* between the senders associated with the messages in the *conflict-set*. This *Restrained Consensus* involves only the senders in conflict and thus is less costly than regular consensus provided that none of its participants is Byzantine and no period of asynchrony occurs during its execution. Thus, either one of the participants is chosen by *Restrained Consensus* and its message can be delivered, or in the event of a timeout, a classical Consensus algorithm is used to settle the *Cascade Consensus* protocol and reach a final decision.

We choose this protocol as the graceful conditions allowing the early termination of Cascade Consensus match the expected behavior of TreeKEM. Indeed, depending on the size and the dynamics of a group, conflicts may not occur very often, in which cases the consensus ends after a CAC broadcast. Additionally, these conflicts may usually be restricted to a few members of the group, reducing the complexity of consensus by triggering Restrained Consensus.

3 The TreeKEM Protocol and its Delivery Service

The TreeKEM protocol underlying Messaging Layer Security (MLS) [6] provides a standard and well-secured solution for end-to-end secure group communication such as group messaging and video conferencing. We first present the Propose and Commit approach used by TreeKEM to handle concurrent operations in the group. Next, we introduce the core structure of TreeKEM based on the ratcheting tree. We then describe the Delivery Service component of TreeKEM which is in charge of handling protocol communication and resolving conflicts between members. We present our formalization of the role and the properties of this Delivery Service component. Finally, we describe the centralized solution of the Delivery Service adopted by TreeKEM and its follow-up protocols.

3.1 Propose and Commit

The 8$^{\text{th}}$ draft of the MLS Protocol, introduces the principle of *Propose and Commit* in the TreeKEM Protocol which describes the current way of organizing the operations in the protocol. This allows group members to perform concurrent operations in the form of *Proposals*. Then, these proposals can be merged into a *Commit* which materializes these changes into the group and leads to a new *epoch* for the group. In TreeKEM, it is up to the application using the protocol to decide which client commits the *Proposals*.

In the case of conflicting *Proposals* (e.g. multiple add/remove/update *Proposals* for the same member), the *commiter* must choose between those proposals to solve the conflicts. Additionally, if one *Commit* contains *Proposals* to add one or multiple members, the protocol generates a *Welcome* message that allows them to effectively join the group.

Therefore, based on the MLS RFC [6] we propose the following formalization of the TreeKEM protocol:

- $s \leftarrow \textsc{Init}(ID)$: initiates the group state s for the user associated with the identity ID.
- $p \leftarrow \textsc{ProposeAdd}(s, ID')$: creates a proposal to add the user corresponding to ID' to the group with the current state s. The operation outputs a proposal p.
- $p \leftarrow \textsc{ProposeRemove}(s, ID')$: creates a proposal to remove the user corresponding to ID' from the group with current state s. The operation outputs a proposal p.
- $p \leftarrow \textsc{ProposeUpdate}(s, k)$: creates a proposal to update the current user's key in current state s with k. The operation outputs a proposal p.
- $(C, W) \leftarrow \textsc{Commit}(s, P)$: commits a set P of valid and non conflicting proposals (i.e. according to Section 12.2 of the RFC [6]) to make the current group state s progress. The operation outputs a commit message C to make current members transition to the new group state and possibly a welcome message W to allow proposed new users to join the group.
- $(s', K) \leftarrow \textsc{Process}(s, M)$: processes a commit or welcome message M to transition from the old group state s to the new group state s'. The transition additionally generates a new *group key* K.

3.2 Ratcheting Tree

One way to implement the protocol would be for a member to generate a new group key and encrypt this key with the public keys of each user. However this kind of approach would scale poorly in the case of large groups. Thus, TreeKEM uses the concept of the ratcheting tree in order to improve scalability.

The ratcheting tree is organized as a balanced binary tree of public-private key-pairs whose leaves represent members of the group. Group members do not share the same view of the tree as each member only knows a subset of the private keys: the keys on the path from the leaf associated with this member

to the root of the tree. Therefore the only private key known by all members is the root of the tree which gets to be used as the group key. Figure 1 presents the example of a ratcheting tree in the case of a 4-member group. In the tree, members 1 and 2 share the secret associated with A, 3 and 4 share the secret associated with B and C's secret is shared among all members and considered the group key.

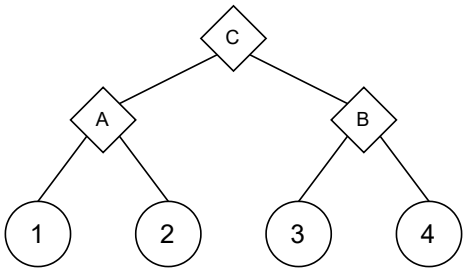

Fig. 1. Example of a TreeKEM ratcheting tree in the case of a 4-user group.

When a member creates a Commit message to advance the group state, this member modifies one or more leaves in the tree to apply the operations described by the associated proposals. Then, this member updates theirs path to the root in order to ensure the renewal of the group key. This can be done in two steps:

- First, a key derivation mechanism, similar to a hash function, makes it possible to generate the new keys. For example, member 1 would generate a new leaf key $k_{1'}$, and iterate a key derivation mechanism to generate the following keys: $k_{1'} \rightarrow k_{A'} \rightarrow k_{C'}$.
- Then, the tree structure can be used to limit the number of encryptions: $k_{A'}$ can be encrypted to member 2, and $k_{C'}$ can be encrypted to the remaining sub-group using the public key of B.

As members can complete the chain of keys using the key-derivation mechanism, the number of keys and needed encryptions to generate a Commit message is limited to $log(n)$ in a group of size n, which offers good scalability. However, the protocol does not provide a way to merge the modifications conveyed by multiple commits. Thus, in the event of concurrent commits, only one commit must be chosen to be applied by all group members. The task of resolving these conflicts falls under the responsibility of the *Delivery Service* that will be described in the following subsection.

3.3 The TreeKEM's Delivery Service

The MLS Architecture draft [8] describes the typical architecture in which clients implementing the MLS Protocol [6] interact with each other. In this architecture,

clients communicate with each other using a *Delivery Service* whose detailed description falls out of the scope of the MLS Protocol.

Thus, the role of the *Delivery Service* is essentially to ensure the delivery of the different types of messages: Application messages exchanged between users, *Proposal* messages and *Commit* messages. The only constraint for Application and *Proposals* messages is that they must eventually reach their recipients.

However, *Commit* messages present different challenges. In fact, the MLS Protocol imposes a linear history of epochs. This means that the members should process the *Commit* messages in the same order and only one *Commit* message can exist per epoch, as the protocol is not capable of merging multiple *Commits*. Ensuring this linear history can be a problem as the members may generate concurrent commits in different situations:

- On the one hand, some members might not take advantage of concurrent operations (i.e. send a *Proposal* directly followed by a corresponding *Commit*) and in dynamic and/or large groups, there is a high probability of members issuing multiple *Commits* simultaneously.
- On the other hand, even when members first send *Proposals* to allow operations to be executed concurrently, there is still the need for one of those members to *commit* the operations. This member can be determined deterministically to limit the cost, for example based on the set of received proposals. But, in a distributed system the presence of failures and asynchronous periods may prevent all members from deciding on the same *commiter*. Thus, even if the probability of conflicts is lowered, it is still possible for concurrent *commits* to be issued. Addressing this situation requires solving consensus.

Therefore, in case of conflicts, the *Delivery Service* must act as a reference, capable of deciding on one *Commit* that all members will consider as the right one for a given *epoch*.

Formalization. Based on the specifications provided in the MLS Protocol [6] and the MLS Architecture draft [8], we now present our formalization of the role and the properties of the TreeKEM Delivery Service.

The Delivery Service is a group communication mechanism that provides two operations and two callbacks:

- `ds_proposal_broadcast`(pm): a process p_i can invoke this operation to submit a proposal message pm.
- `ds_proposal_deliver`(p_i, pm): callback triggered to deliver a proposal message pm broadcast by process p_i.
- `ds_commit_propose`(ep, cm): a process p_i can invoke this operation to submit a commit message cm for the current epoch ep.
- `ds_commit_deliver`(ep, p_i, cm): callback triggered to deliver the commit message cm for the epoch ep and broadcast by process p_i, which allows members to progress to the next epoch and state.

The Delivery Service satisfies the following properties:

- **Proposal Validity.** If a correct process p_i ds-proposal-delivers a pair (p_j,pm), then pm is a valid proposal message, p_j is a correct process and p_j has previously ds-proposal-broadcast pm.
- **Proposal Totality.** If a correct process p_i ds-proposal-broadcasts a proposal message pm, then all correct processes eventually ds-proposal-deliver the pair (p_i,pm).
- **Epoch Validity.** If a correct process p_i ds-commit-delivers a tuple (ep,p_j,cm), then ep is the current epoch, cm is a valid commit message in the current epoch, p_j is a correct process and p_j has previously ds-commit-proposed cm.
- **Epoch Agreement.** If any two correct processes ds-commit-deliver respective tuples (ep,-,cm) and (ep',-,cm'), and if $ep = ep'$ then we have $cm = cm'$.
- **Epoch Termination.** If a correct process p_i ds-commit-proposes a commit message cm for the epoch ep, then all correct processes eventually ds-commit-deliver a pair (ep,-,-).
- **Epoch-Content Consistency** If a correct process ds-commit-delivers a tuple (-,-,cm), then for all proposal messages pm referenced by cm all correct processes have previously ds-proposal-delivered a tuple (-,pm).

3.4 Centralized Delivery Service

The approach adopted by TreeKEM and other protocols it inspired (e.g. [15], [2]) is to rely on a central authority to implement the Delivery Service. In this case, the Delivery Service would be operated by a central server. To keep confidentiality this central server is assumed to be *untrusted*. This means that, as the messages are end-to-end encrypted, the server cannot read or modify the messages exchanged between members, but has access to a limited set of metadata necessary to fulfill its task (i.e. group id and epoch).

Due to its limited view on the content of messages, an *untrusted* centralized Delivery Service can only accomplish the role of an ordering server, providing the different messages in the same order for all clients. Then, in case of conflicts, the members can choose the first valid commit message and coherently resolve the conflict.

However, this central server can easily become a target of attacks such as Denial of Service which can impact its availability. In more complex settings, a compromised Delivery Service can be used to mount attacks against specific groups. By blocking the operations of given members, one can prevent these members from refreshing their keys and thus eventually lead to the failure of the Post-Compromise Security property.

4 A Distributed Delivery Service

In this section, we describe our solution for a fully distributed Delivery Service. Instead of relying on a third-party untrusted server, the participants in the key agreement protocols, or a subset of them, directly run the delivery service. We illustrate our solution as a flowchart in Figure 2 and formally in Algorithm 1. The

flowchart in Figure 2 describes the verification process and the delays that messages can encounter between their receipt from the communication component and their delivery. The Algorithm 1 details how the execution unfolds.

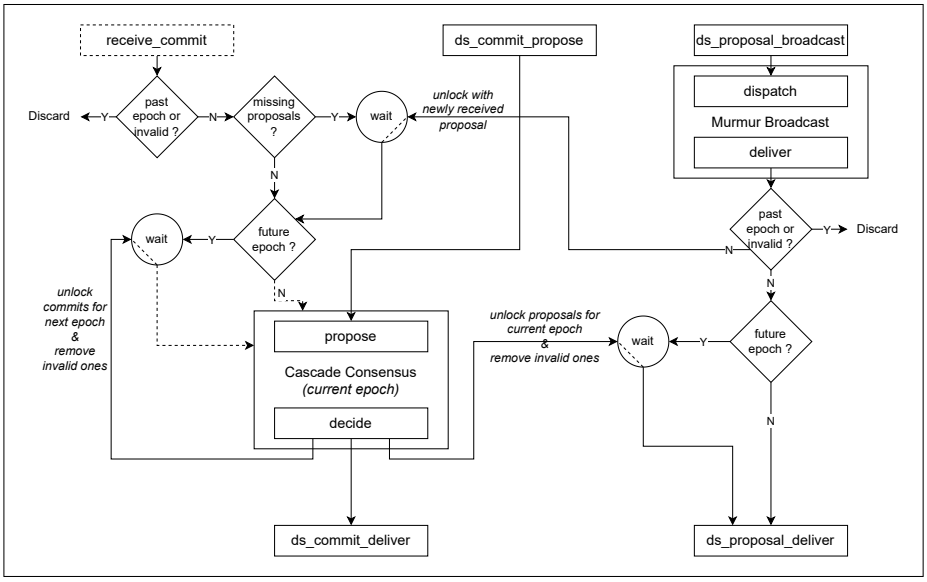

Fig. 2. Diagram detailing our solution for a Distributed Delivery Service

Our solution relies on two basic communication mechanisms adapted to the different messages of the TreeKEM protocol: a Probabilistic Reliable Broadcast (i.e. the Murmur protocol from [13]) for Proposal messages and the Cascade Consensus Protocol [1] for Commit messages. We introduce a slight modification to the Cascade Consensus Protocol and to the CAC broadcast mechanism on which it is based (see Section 2). Specifically, we delay the acknowledgement of a commit (i.e. the emission of a Witness message upon receipt of a first Witness message) until the client considers this commit as valid. We materialize this validation by a call to `ccb.receiveCommit` (l. 17). This ensures that the commit message can be introduced as a candidate in the consensus protocol only when all members agree that this commit is valid.

Algorithm 1 Formalization of our solution for a Distributed Delivery Service based on the Cascade Consensus Protocol and the Murmur broadcast protocol.

Implements:
1: DistributedDeliveryService, **instance** dds
Uses:
2: MurmurBroadcast, **instance** mb
3: CascadeConsensusBroadcast, **instance** ccb
4: TreeKEMProtocol, **instance** tkem
5:
6: **upon event** < Init > **do**
7: $proposals = \emptyset$; $incomplete\_commits = \emptyset$; $future\_commits = \emptyset$;
8: $future\_proposals = \emptyset$
9:
10: **upon event** < dds.BroadcastProposal | [`Proposal`, $proposal$] > **do**
11: **trigger** < mb.Dispatch | [`Proposal`, $proposal$] >
12:
13: **procedure** *handleCommit*($commit$) ▷ handle commits that are or became complete
14: **if** $commit$.epoch > $tkem$.currentEpoch **then**
15: $future\_commits \leftarrow future\_commits \cup commit$
16: **else**
17: ccb.receiveCommit($commit$)
18: **end if**
19:
20: **upon event** < mb.Deliver | [`Proposal`, $proposal$] > **do** ▷ ignore past and invalid proposals
21: $proposals \leftarrow proposals \cup proposal$
22: **for** $commit \in incomplete\_commits$ **do**
23: **if** $commit$.proposals $\subseteq proposals$ **then**
24: *handleCommit*($commit$)
25: $incomplete\_commits \leftarrow incomplete\_commits \setminus commit$
26: **end if**
27: **end for**
28: **if** $proposal$.epoch = $tkem$.currentEpoch **then**
29: **trigger** < dds.DeliverProposal | [`Proposal`, $proposal$] >
30: **else**
31: $future\_proposals \leftarrow future\_proposals \cup proposal$
32: **end if**
33:
34: **upon event** < dds.ProposeCommit | [`Commit`, $commit$] > **do**
35: **trigger** < ccb.Propose | $tkem$.currentEpoch,[`Commit`, $commit$] >

```
36: upon event < ccb.ReceiveCommit | [Commit, commit] > do      ▷ ignore past and invalid commits
37:     if commit.proposals ⊈ proposals then
38:         incomplete_commits ← incomplete_commits ∪ commit
39:     else
40:         handleCommit(commit)
41:     end if
42:
43: upon event < ccb.Deliver | [Commit, commit] > do
44:     tkem.apply(commit)
45:     for proposal ∈ future_proposals do
46:         if tkem.currentEpoch = proposal.epoch and tkem.isValid(proposal) then
47:             trigger < dds.DeliverProposal | [Proposal, proposal] >
48:             future_proposals ← future_proposals \ proposal
49:         end if
50:     end for
51:     for commit ∈ future_commits do
52:         if tkem.currentEpoch = commit.epoch and tkem.isValid(commit) then
53:             ccb.receiveCommit(commit)
54:             future_commits ← future_commits \ commit
55:         end if
56:     end for
57:     trigger < dds.DeliverCommit | [Commit, commit] >
```

Delaying commits allows our solution to control the flow of epochs, making sure that it only delivers messages belonging to the current epoch, and that all members have moved to the current epoch before deciding on the next one. This prevents an attacker from executing a *Denial of Service attack* by submitting a large number of commits in a short time. To implement this delay, Algorithm 1 employs two waiting queues (*incomplete_commits*, *future_commits*) initialized on line 7. Together, these queues allow our solution to wait until all the proposals referenced by a commit are received before starting an agreement (i.e. a consensus) on this commit. This leads to the satisfaction of the *Epoch-Content-Consistency* property.

In a similar manner, we also introduce a delay on proposal messages (by means of the *future_proposals* waiting queue initialized on line 8 of Algorithm 1). This makes it possible to delay proposals whose messages belong to a future epoch—that a member suffering from network delays cannot immediately verify—or that were created by an attacker in an attempt to block the progress of the group.

We handle proposal messages using the Murmur [13] protocol (l. 20), which guarantees *Proposal Totality*. When a member receives a proposal message, two cases can arise. If the proposal belongs to the current epoch, and thus satisfies *Proposal Validity*, we can directly deliver the message (l. 29). Otherwise, we wait for this message to become valid by inserting this message in the list of future proposals (l. 31). Additionally, we check if this proposal is referenced by any

commit previously received (l. 22). This check may unblock a commit whose acknowledgment was delayed to ensure *Epoch-Content Consistency*.

To handle commit messages, on the other hand, we employ the modified cascade-consensus mechanism (l. 36). Commits whose proposals are missing are delayed (l. 38), either if some of their proposals are missing to guarantee *Epoch-Content Consistency* or to guarantee *Commit Validity* if the commit belongs to a future epoch (l. 15) that cannot yet be verified. Then, valid commits are transmitted as candidates for the next epoch to the Cascade Consensus Protocol. This protocol ensures the *Epoch Agreement* and *Epoch Termination*.

Finally, when members all agree on the same commit, this commit is delivered by the consensus protocol (l. 43). This commit can be transmitted to the TreeKEM protocol to take effect. Additionally, the change of epoch can unblock proposals (l. 47) and commits (l. 53) that were previously delayed if the client received messages out of order.

5 Conclusion

In this paper we presented a completely distributed solution for the design of the Delivery-Service component of TreeKEM by combining a Probabilistic Broadcast [13] method with the Cascade Consensus Protocol [1]. We formalized the role and the necessary properties of this Delivery-Service component and we proposed a novel algorithm supporting its distributed counterpart. This design allows users to run the TreeKEM protocol or any similar Group Key Agreement protocol without requiring their communication to go through a central server. Our approach increases the security and the availability of group communication making it harder for an attacker to compromise the Delivery Service. Mounting attacks against the group requires compromising one third of the clients [1] instead of only one server in the standard centralized solution.

We plan to implement our Distributed Delivery Service solution and to conduct a performance evaluation in the context of peer-to-peer networks. Additionally, we plan to conduct a formal security analysis of our protocol. We will further extend our solution in order to explicitly support offline group members. Currently, the TreeKEM protocol supports offline members using a central server that stores messages for these users. We plan to design completely distributed alternative solutions for offline communication by relying on State-Machine-Replication [7] mechanisms such as State Transfer. This should allow previously offline members to get up to date with the group by directly synchronizing with some other members. In order to support a completely distributed group-key-management component for our real-time peer-to-peer collaborative editor MUTE [17], we plan to integrate the TreeKEM protocol and our distributed Delivery Service. We further plan to combine this distributed group-key-management mechanism with a distributed access control mechanism such as ACCURE [19].

Acknowledgments

This work is supported by the "Alvearium" Inria and hive partnership.

References

1. Albouy, T., Frey, D., Gestin, M., Raynal, M., Taïani, F.: Context adaptive cooperation (2023), https://arxiv.org/abs/2311.08776
2. Alwen, J., Auerbach, B., Noval, M.C., Klein, K., Pascual-Perez, G., Pietrzak, K., Walter, M.: Cocoa: Concurrent continuous group key agreement. In: Advances in Cryptology – EUROCRYPT 2022: 41st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Trondheim, Norway, May 30 – June 3, 2022, Proceedings, Part II. p. 815–844. Springer-Verlag, Berlin, Heidelberg (2022). https://doi.org/10.1007/978-3-031-07085-3_28
3. Alwen, J., Coretti, S., Dodis, Y., Tselekounis, Y.: Modular design of secure group messaging protocols and the security of mls. In: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security. p. 1463–1483. CCS '21, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3460120.3484820
4. Ateniese, G., Steiner, M., Tsudik, G.: Authenticated group key agreement and friends. In: Proceedings of the 5th ACM Conference on Computer and Communications Security. p. 17–26. CCS '98 (1998). https://doi.org/10.1145/288090.288097
5. Balbás, D., Collins, D., Gajland, P.: Analysis and improvements of the sender keys protocol for group messaging. XVII Reunión española sobre criptología y seguridad de la información. RECSI 2022 **265**, 25 (2022), https://arxiv.org/abs/2301.07045
6. Barnes, R., Beurdouche, B., Robert, R., Millican, J., Omara, E., Cohn-Gordon, K.: The Messaging Layer Security (MLS) Protocol. RFC 9420 (Jul 2023). https://doi.org/10.17487/RFC9420, https://www.rfc-editor.org/info/rfc9420
7. Bessani, A., Sousa, J., Alchieri, E.E.: State machine replication for the masses with bft-smart. In: 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks. pp. 355–362 (2014). https://doi.org/10.1109/DSN.2014.43
8. Beurdouche, B., Rescorla, E., Omara, E., Inguva, S., Duric, A.: The Messaging Layer Security (MLS) Architecture. Internet-Draft draft-ietf-mls-architecture-10, Internet Engineering Task Force (Dec 2022), https://datatracker.ietf.org/doc/draft-ietf-mls-architecture/10/, work in Progress
9. Bhargavan, K., Barnes, R., Rescorla, E.: TreeKEM: Asynchronous Decentralized Key Management for Large Dynamic Groups A protocol proposal for Messaging Layer Security (MLS). Research report, Inria Paris (May 2018), https://hal.inria.fr/hal-02425247
10. Brzuska, C., Cornelissen, E., Kohbrok, K.: Security analysis of the mls key derivation. In: 2022 IEEE Symposium on Security and Privacy (SP). pp. 2535–2553 (2022). https://doi.org/10.1109/SP46214.2022.9833678
11. Burmester, M., Desmedt, Y.: A secure and efficient conference key distribution system. In: De Santis, A. (ed.) Advances in Cryptology — EUROCRYPT'94 (1995), https://www.cs.fsu.edu/~langley/Eurocrypt/euro-pre.pdf
12. Cohn-Gordon, K., Cremers, C., Garratt, L., Millican, J., Milner, K.: On ends-to-ends encryption: Asynchronous group messaging with strong security guarantees.

In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. p. 1802–1819. CCS '18, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3243734.3243747
13. Guerraoui, R., Kuznetsov, P., Monti, M., Pavlovic, M., Seredinschi, D.A.: Scalable Byzantine Reliable Broadcast (Extended Version). In: 33rd International Symposium on Distributed Computing (DISC 2019) (2019), https://arxiv.org/abs/1908.01738
14. Kim, Y., Perrig, A., Tsudik, G.: Simple and fault-tolerant key agreement for dynamic collaborative groups. In: Proceedings of the 7th ACM Conference on Computer and Communications Security. p. 235–244. CCS '00 (2000). https://doi.org/10.1145/352600.352638
15. Klein, K., Pascual-Perez, G., Walter, M., Kamath, C., Capretto, M., Cueto, M., Markov, I., Yeo, M., Alwen, J., Pietrzak, K.: Keep the dirt: Tainted treekem, adaptively and actively secure continuous group key agreement. In: 2021 IEEE Symposium on Security and Privacy (SP). pp. 268–284 (2021). https://doi.org/10.1109/SP40001.2021.00035
16. Moxie, M., Trevor, P.: Signal - specifications - the x3dh key agreement protocol (2016), https://signal.org/docs/specifications/x3dh/
17. Nicolas, M., Elvinger, V., Oster, G., Ignat, C.L., Charoy, F.: MUTE: a peer-to-peer web-based real-time collaborative editor. In: ECSCW 2017 - 15th European Conference on Computer-Supported Cooperative Work. Proceedings of 15th European Conference on Computer-Supported Cooperative Work - Panels, Posters and Demos, vol. 1, pp. 1–4. EUSSET, Sheffield, United Kingdom (Aug 2017). https://doi.org/10.18420/ecscw2017_p5
18. Perrin, T., Marlinspike, M.: The double ratchet algorithm. Signal - Specifications (2016), https://signal.org/docs/specifications/doubleratchet/
19. Rault, P.A., Ignat, C.L., Perrin, O.: Access control based on CRDTs for Collaborative Distributed Applications. In: The International Symposium on Intelligent and Trustworthy Computing, Communications, and Networking (ITCCN-2023), Proceedings of the 22nd IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom-2023). Exeter, UK (Nov 2023), https://inria.hal.science/hal-04224855
20. Steiner, M., Tsudik, G., Waidner, M.: Key agreement in dynamic peer groups. IEEE Transactions on Parallel and Distributed Systems **11**(8), 769–780 (2000). https://doi.org/10.1109/71.877936