Bachelor of Science

Decentral Group Management in Messengers with Delegated Proof of Stake

---

By And Hell
at 14. August 2019

Abstract

Mobile messaging applications are used widely for group communication using group chats. Most messenger platforms rely on their centralized infrastructure to maintain the group states. This can imply privacy issues and allow potential missus by the messenger.

The goal of this work is to design a protocol, which distributes the group managing between the group members directly. The protocol is based on the Delegated Proof of Stake (DPoS) consensus protocol and uses a blockchain to store the groups state. The designed protocol is implemented and tested using a simulated group. Further the properties of the protocol and its security are discussed.

Contents

Chapter 1

Introduction

Mobile messaging applications, also known as *messenger*, are used by over one billion people to connect to each other and exchange messages. One of the core features of applications like WhatsApp, Signal, Facebook Messenger, Threema and Telegram are group chats. Group chats allow multiple users to communicate with each other over a shared message thread. Users can be added to these threads or leave them any time.

To maintain the member list of such a group chat most messenger uses *administrators*, which are members with additional rights. While all populare messenger uses centralized servers to distributed messages, some also uses them for the group managing. This weakens the users privacy, even if the application uses end-to-end encryption to secure the messages content, and introduces a single point of failure. Current ideas from secret services such as the GCHQ plan to misuse this by forcing the services to add ghost users to the groups [13][7].

By using blockchain technology the group management can be distributed between all members and remove the need to a centralized management. Delegated Proof of Stake enables the members to elected their administrators and allow all members to contribute to the group equally.

This work focuses on Delegated Proof of Stake (DPoS) as consensus protocol and analyses the usage of such a system for distributed group management in secure messaging applications. To do so, a protocol is designed which uses Delegated Proof of Stake and is tested using an autonomous running group simulation.

Chapter 2

Backgroud

2.1 Messenger

Unlike SMS and Email, mobile messaging applications are not interoperable. In order to communicate with each other, users need to have the same applications installed. Compared to such distributed systems, applications like Whatsapp, Signal, Telegram or Facebook Messenger use a central server for message exchange and contact discovery.

To secure their users, many applications offer end-to-end encryption of the message content. Either the encryption is used by default, such as in WhatsApp, Signal or Threema or can be activated if needed such as in Telegram, or Facebook Messenger.

Messaging applications are used primarily on smartphones. This leads to asynchronous communication, since the connection can be unreliable and users can be contacted at any time, but may only response from time to time. In order to communicate it is not nesseary to be online at the same time. Messages are processed as soon as they can be received and the receiver decides to response.

Another feature of a messenger is the indication of message delivery. If a message is deliverd successfully, this is shown to the user through two checkmarks or similar methods. Furthermore, messengers can also exchange read receipients, that shows the sender, that the receiver has read the message.

Groups

The properties explained above, also hold true for group chats. Group chats are a shared message thread, that can be accessed by a limited set of users and is used for asynchronous communication.

Instead of temporary threads, like discussions on a specific topic in a mailing list, group chat in messaging applications are used for long term communication.

In order to identify groups, members can set a name and, optionally, a picture or a group description.

To manage group chats most systems use administrators. These are members of the group with additional rights. Depending on the messaging application, for example, only administrators can add or remove members.

Definition and Notation

In their analysis of different group messaging protocols Rösler et al. introduced definitions and notations that will be adopted partly in this work [1].

Given a group identifier ID_{gr}, a set of members $\mathcal{M}_{gr}$, a set of administrators $\mathcal{M}^*_{gr}$ and the group information $info_{gr}$, the state of a group gr can be described at any given point.

$$gr = (ID_{gr}, \mathcal{M}_{gr}, \mathcal{M}^*_{gr}, info_{gr})$$

- $\mathcal{U}$: The set of users of a messenger. (E.G. $\{A, B, C, ...Z\}$)
- $\mathcal{M}_{gr}$: The set of members of a group gr where $\mathcal{M}^*_{gr} \subseteq \mathcal{U}$. (E.G. $\{A, B, C, D\}$)
- $\mathcal{M}^*_{gr}$: The set of administrators of a group gr where $\mathcal{M}^*_{gr} \subseteq \mathcal{M}_{gr}$. (E.G. $\{A^*, B^*\}$)
- $info_{gr}$: Addtional information like a group name, picture or description

The users of a protocol are able to perform the following actions:

- $ContM(gr, m)$: Send a content message m to the group gr.

- $Add(gr, V)$: Add user V to the group gr.
- $Leave(gr)$: Sender leaves the group gr.
- $Rmv(gr, V)$: Remove user V from group gr.
- $Upd(gr, info)$: Update info of group gr.
- $Act(gr) \in \{ContM(gr, m), Add(gr, V), Leave(gr), Rmv(gr, V), Upd(gr, info)\}$
- $Snd(gr, Act)$: Send an action Act to a group gr.
- $Rec(gr, Act, U)$: Recieve an action Act for a group gr from a user U.

Users U can send $Snd(gr, Act)$ and receive $Rec(gr, Act, U)$ actions $Act(gr)$ to and from members $V_i \in \mathcal{M}_{gr}$. $Add(gr, V)$, $Leave(gr)$, $Rmv(gr, V)$ and $Upd(gr, info)$ leads to a change of the group state gr.

2.2 Security Model

Rösler et al. provided a security model to analyze group messaging protocols [1]. The goals and threat models are adopted in this work. However, due to the focus on the management itself, not all requirements are relevant, since they are reached by the used end-to-end encryption and message protocol itself.

2.2.1 Security Goals

Three security goals for group chats are defined by Rösler et al. 1) confidentiality of the conversations' content, 2) integrity of the conversation and 3) the confidentiality included by the group management. These goals can be extended with 4) privacy and 5) same state.

While 1) and 2) can be reached with the end-to-end encryption protocol, 3), 4) and 5) are relevant for group management.

Confidentiality by Group Management

Only members of a group gr should be able to change the group state gr.

Additive Closeness Only administrators should be able to add new members to a group.
If an action $Add(gr, V)$ is received with $Rec(gr, Add(gr, V), U)$, than $U \in M^*_{gr}$ must hold.

Subtractive Closeness Only administrators should be able to remove members from a group.
If an action $Rmv(gr, V)$ is received with $Rec(gr, Rmv(gr, V), U)$, than $U \in M^*_{gr}$ must hold.

Privacy

Group management should protect the users privacy. Besides confidentiality of message content, this includes confidentiality about the group itself. Only the members should know about the existence of this group and its state.

Same State

Members of a group need to be able to ensure that they act under the same assumptions of the groups state.

$$gr_A = gr_B = gr_C ...$$

2.2.2 Threat Model

Rösler et al. considered three types of adversaries against messaging protocols. *Malicious server, malicious user* and *network attacker*. In regards toward group managing also a *malicious member* needs to be considered.

Malicious Server

A malicious server attacks the connection between client and server or obtains control over the server. This adversary can modify and forge all communication that is only protected on the transport layer or drop messages.

Malicious User

This is an adversary with access to the system like a normal user that tries to forge messages to break the requirements defined above. This user is not a member of the target group.

Network Attacker

A network attacker has access to the communication network. He is able to modify unprotected traffic and drop connections.

Malicious Member

In this case, the adversary is a member of the target group. Either because the member lost control over his devices, or decides to harm the group himself, out of personal reasons.

2.3 Examples

The following section analysis the currently used group management protocols of Signal and WhatsApp.

2.3.1 Signal

Signal is an open source, end-to-end encrypted messenger. The application is available for Android and Apples iPhone. Desktop systems can be used as additional devices using the Electorn based desktop application.

End-to-end encryption is provided by the Signal Protocol, which provides *prefect forward secrecy* as well as *future secrecy*, also known as *post compromise security*. These properties are archived through using using the asynchronous X3DH, or *Extended Triple Diffie-Hellman* [18], for key exchange and the Double Ratchet algorithm [17] for message encryption.

In 2018 Signal introduced Sealed Sender, where the senders user identifier is also part of the encrypted message content. Therefore the Signal server can not know which user are is sending the message [14].

Group Messages

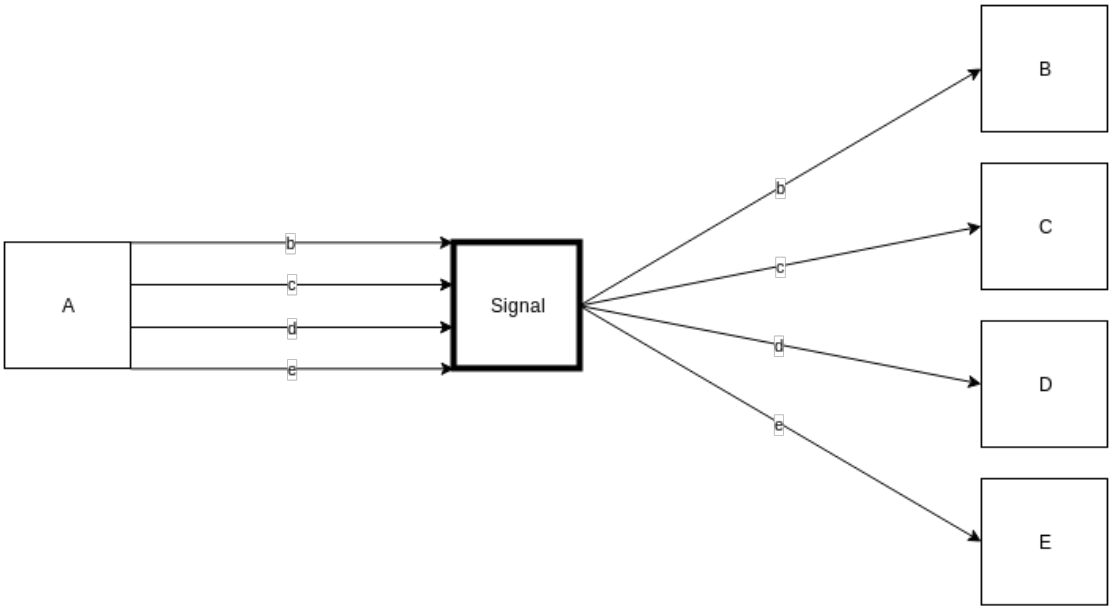

Figure 2.1: User A sends a group message as multiple messages (b, c, d, e) to the Signal server, where they are forwared to the receivers (B, C, D, E).

Group messages in Signal are handled the same as direct messages. As a result, properties like *prefect forward secrecy* and *future secrecy* are also achived for group messages [15]. As shown in figure 2.2, a seperate message is send to each member, instead of a single message to the group. Group messages include a group identifier ID_{gr}, that is used to assign the messages to the correct group thread by the receiver.

By handling group message this way, the server does not need to know, that groups exist in the first place.

Group Management

Unlike most other applications, Signal does not rely on *administrators* in order to manage groups. Every member can add new users or change the group information like name or picture [15]. Per the definition above, every member can be considered a administrator such that $\mathcal{M}_{gr}^{*} = \mathcal{M}_{gr}$. In Signal, members can not be removed by other members. Users can only leave a group by themselves. Signal groups are not limited in size [24].

Three group message types are defined for management: *Update, Quit* and *Request Info* [23]. *Update* is used to create a group, as well as, to add members or to update the group info. *Quit* is used to leave a group and *Request Info* to sync group info and member lists with other users, for example after joining a new group.

According to the user forum, Signal groups tend to become unusable over time [8]. This can happen due to users uninstalling Signal without being properly unregistered or by issues with handling *leave* messages. In this cases, users may not be removed from the member list of all users.

While Signal keeps groups private to the server, Rösler et al. showed that *addaptive closeness* could be broken by a malicious user that gains access to the groups identifier ID_{GR} [21]. As of 2019, this issue seems not to be fixed in the Signal Android codebase [22].

2.3.2 WhatsApp

With over 1.5 billion users WhatsApp is one of the most used messaging services. The application is available for Android, iOS and Windows Phone. Desktop systems can use the web based remote application.

After WhatsApp was acquired by Facebook in 2014, *Open Whisper Systems*[1] announced to be working with them to bring end-to-end encryption to their users. In 2016 the integration of the Signal Protocol was completed [9].

Group Messages

Even though WhatsApp uses the Signal Protocol, group messages are encrypted and send differently as compared to the Signal application. WhatsApp sends a single, end-to-end encrypted group message to the server. The server then distributes the message to each member [27]. Due to this design, messages can not be encrypted with the Double Ratchet algorithm. Instead WhatsApp uses *Sender Keys* which are also defined in the Signal Protocol [27]. Sender Key encryption only provides *perfect forward secrecy* but no *future secrecy*. Messages for group management are only encrypted on the transport layer [1].

[1] Open Whisper Systems was the organization behind Signal before the Signal Foundation was founded in 2018 [16].

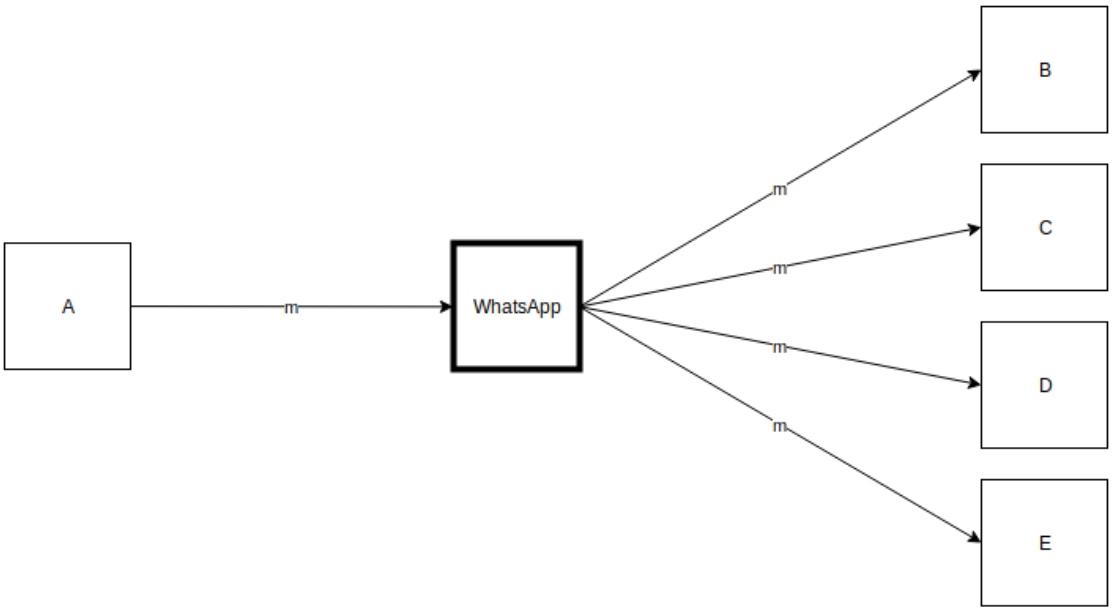

Figure 2.2: User A sends a group message m to the WhatsApp server, where it is forwared to the receivers (B, C, D, E).

Group Management

Like message distribution, the group management also relies on the WhatsApp server. Users send an action $Act(gr)$ to the WhatsApp server, where it is processed and forwarded to the members.

Unlike Signal, Whatsapp uses administrators. For new groups, the founder is the only administrator, but he can promote additional administrators. Administrators can demote each other. It is also possible to restrict the edition of the group info and the ability to send messages to administrators only. [26]

As shown by Rösler et al. WhatsApp groups can be attacked by a malicious server, since management messages could be forged by such an adversary [21]. The *privacy* goal is also not fulfilled. The server needs to know about the groups and its members in order to distribute messages and needs to be able to access all management messages, in order to maintain the current group state.

2.4 Blockchains

Blockchains are used to build decentral, digital currencies. Beside being decentral, blockchain systems can also be premissionless, immutable and trustless. These properties allow the systems to achive consensus without the need to a centralised authority.

The most widely known cryptographic currency is BitCoin. In BitCoin, everyone (premissionless) can contribute and try to build a new block. For these new blocks the miner needs to find a cryptographic hash that matches specific properties. The search for such a hash requires computational effort and is therefore called *Proof of Work*. Since all blocks are chain together by including the previous block hash, it is required to recompute all following blocks, if one would try to change an old block (immutable). Everyone can verify the blocks hashes and the found solution, since the blockchain is public (trustless).

2.4.1 Delegated Proof of Stake

As oposed to Proof of Work, in Delegated Proof of Stake (DPoS), no computational effort is needed to build new blocks and secure the chain. Instead stakeholders can elect a set of representatives that then build and distribute new blocks according to a predefined schedule.

In BitShares, stakeholders add a list of chosen delegates, also called witnesses or representatives, to their transactions [11] as their vote. These votes are weighted by the amount of coins the voter holds [2]. After a defined period of time, all votes are recounted and the result is used for the next delegates. There is no fixed election day, on which users are required to vote. Instead voting is possible at any time, through commiting a transaction to the chain.

Each delegate is assigned at a specific point in time that is defined by the schedule to build his block. After every representative has signed a block, a new round starts. The elected delegates are scheduled in a random order, such that block producer B cannot discriminate block producer A by ignoring his blocks [12]. To build new blocks, delegates collect transactions and combine them to a block, that is signed with their public key. If a block producer fails to build his block on time, he does not get the block reward and might not be elected again if he repeats to fail. By this design Delegated Proof of Stake can be considered a synchronous protocol, since delegates need to be online all the time.

Delegated Proof of Stake enables blockchains with block times form 3 seconds, as in BitShares [3], down to 0.5 seconds, as in EOS [6].

Transactions as Proof of Stake

Transactions can contain a hash of the most recent block of the chain. This proofs that the transaction was made under a given assumption about the chains state. If such a block is not part of the chain, the assumption was invalid and the transaction will not be included in new blocks. This also prevents that blocks are included in secret chains, that an attacker might hold back for a double spending attack [12].

Chapter 3

Idea

Decentralized group management, like in Signal, provides more protection towards the users privacy. But since a central server is missing, it is harder to use administrators to manage the group and ensure consensus about the current group state. The idea of this thesis is to use a distributed consensus protocol that is used to decentralise the management, while keeping the group states consistent and secure.

By using Delegated Proof of Stake, group members could elect their delegates, that act as administrators. These delegates then validate and confirm *suggestions*, to add or remove a user. These suggestions are made by the other members and would replace the transactions from traditional blockchains. To confirm suggestions, delegates could sign them and distribute them as a new block. If delegates do not want to approve the suggestion, they can ignore them. Since such management actions are less frequent than transactions on a public blockchain, blocks can be build for each suggestion, instead of collecting many transactions first.

Adding the latest block hash in messages or suggestions, users can ensure that they are using the same group state, by just comparing the hashes. If these hashes do not match, the user can request the chain from the other group members to resolve the issue to continue with the correct group state. Like *Transactions as Proof of Stake*, this method can also prevent attacks against the chain.

Due to the synchronous block schedule, Delegated Proof of Stake does not fit in the asynchronous requirements of mobile messaging applications. However, this could be resolved by dropping this schedule and allow delegates to sign and confirm suggestions whenever they are online.

If delegates fail to accept suggestions in a timely manner, or if suggestions are accepted that are not supported by the majority of members, a new election can be triggered and the misbehaving delegates can be replaced.

To motivate members to participate in the group, users votes can be weighted. If a member gets many of his suggestions accepted, or builds more blocks as a delegate than others, their vote can be valued more. This also motivates delegates to sign suggestions faster.

Such a system would empower all members to take control over the group, instead of depending on the decisions of single administrators. Privacy and integrity of the group state are also secured, since noone out side of the group has knowlege about the group and all actions are secured and verified with the help of the blockchain.

Chapter 4

Design

4.1 Blockchain with Delegated Proof of Stake

As explained above, a blockchain could be used together with Delegated Proof of Stake to maintain the groups state. Each group would use its own blockchain to store its history and validate the state and actions. The following section explains the components and actors, that are needed in such a system.

4.1.1 Voting

As shown above, DPoS systems, such as BitShares, allow users to vote at any time, by appending the vote to a transaction. This removes the need for *election days*. However, for group management, these elections days bring benefits compared to the other approach. Unlike in traditional blockchains, it is possible that suggestions may not be included in the chain. Therefore the votes would also not be included. Furthermore, if a member never or only infrequently shares suggestions, no or only old votes are available from this member. With election days, every member is able to make an active decision that is included in the voting process. Such an election can be actively triggered by a member, if the trust in one delegate is lost. As a result, elections are only held if needed and the group can react faster to issues. In the BitShares approach, members would need to wait for the end of the current legislation period, in order to have the vote taking an action.

Votes

Votes are shared as a list of members, together with a signature of the voter. The list contains the members the voter votes for and needs to contain as many items as the amount of delegates that need to be elected. The number of required delegated depends on the group size, such that, for larger groups, members need to vote for more delegates. The signature is the signed Merkle root of this vote list.

The votes are weighted by a factor that can be calculated for each member and is based on the participants in the group management. The needed information to compute this factor can be retrieved from the chain. If a member submits more suggestions, that are accepted or signs more blocks as a delegate, he gets a higher factor.

Voting Process

The voting process is split into two time frames:

1. Voting
2. Confirmation

The voting frame is used by the members to share the vote with the group, while the confirmation frame is used to ensure that every member proceeds with the same election result.

Due to the asynchronous nature of mobile messengers, member can come online at any time during the time frames, or miss the election completely. The protocol needs to ensure, that every member still ends up with the result.

In both time frames, the amount of members that need to contribute is defined. This way it can be ensured that the result represents the majority of the members. If not enough members share their vote or confirm the result, the election must be declared invalid. At least 50% of the members need to share a vote, while 66% should confirm the result.

Voting Frame The voting frame is started if a member shares a *StartVote* suggestion and is open for a predefined time window. If a member gets online during this window, he can share his vote by sending it to the group. However, the member can also decide not to vote.

Every vote that is send and received during this time frame needs to be stored by each member to compute the final result later.

Confirmation Frame If a member gets online during this frame, he collects all votes from the voting window. The users then computes the voting result and sends the result together with a signature and all received votes to the group. Unlike the vote itself, this step should not be optional. However, it can be handled automatically by the application, since an user interaction is not needed.

Every result that is send and received during this window, also needs to be stored.

After the Election If a member gets online after both time frames are closed, he needs to compare all received results. If they are all the same, the member builds a new *Voting Block* for this result and includes all received signatures as confirmation. If the results are not equal, the appended votes need to be compared as well. This allows the members to locate the issue that leads to the different results and take further actions.

This last step is performed by every member, even if the client has not been online during the election itself.

4.1.2 Suggestions as Transactions

Traditional blockchain systems use transactions to transfer coins from one account to another. To be usable for group management, transactions are considered as suggestion for a specific action. If a member wants to update the group name, he can share a suggestion for this change. The suggestion is applied if a delegate builds a valid block that is appended to the blockchain. It is also possible that a valid suggestion is dismissed. To do so, every delegate would need to decide to not sign and build no block for this suggestion. This is the difference to a transaction, that should be included always as long as they are valid.

In this protocol, four types of suggestion are defined:

- Add user
- Remove user
- Update info

- Start new vote

Based on the messengers use case, further suggestions could be added.

Each suggestion includes a reference to the latest block, together with the blocks hash, of the time the suggestion is created. This allows the receivers to validate that the sender uses the correct assumptions about that state.

4.1.3 Delegates

Delegates are elected by the group members and can act as administrators. But unlike admins with direct *write* access, delegates can only confirm suggestion shared by other members. However, if the delegate does not like the suggestion, he can simply ignore the message and no further action is needed.

A delegate signs a received suggestion and distributes it as new block. This new block also contains a proof that the signer is indeed a valid delegate. This proof is the *Merkle verification path* for a Merkle tree [1], which is created from the latest voting result. The group members can then verify the correctness of the delegates new block and append it to their local chain to apply the suggestion.

Compared to the synchronous delegates in a Delegated Proof of Stake protocol, which needs to follow a fixed time schedule, delegates in this system can act freely and build blocks whenever they are online and the user makes the active decision to confirm the suggestion. There is also no fixed order in which order delegates need to operate.

The number of delegates per group should depend on the group size.

4.1.4 Blocks

Blocks are the data structure used to create a verifiable history of the groups state. This is achived by chaining blocks together. Every new block contains the hash of the previous block *phb* and an unique block identifier ID_b. Blocks are build and distributed by the elected delegates.

[1] A Merkle tree is a hash tree. Each node is the hash of its children for non leafs and leafs are the hash of the data at this leaf. Merkle trees can be used to compare large list by comparing the root hash or to efficiently verify the existence of information in a large list, by checking the Merkle verification path.

This work uses four types of blocks to cover different information that needs to be stored in the chain.

Genesis Block

The genesis block gB is used as first block and is shared to create a new group. This block contains the initial group state gr. For a new group, every member acts as a delegate such that the initial group state gr $\mathcal{M}^*_{gr} = \mathcal{M}_{gr}$.

The genesis block is signed sig by the groups founder and uses the group identifier ID_{gr} as block ID.

$$gB = (ID_{gr}, \mathcal{M}_{gr}, info_{gr}, M_{founder}, sig)$$

Voting Block

The Voting Block vB is used to store the election results. Instead of a single signature, this block can contain multiple signatures $[sig]$. One from each member that confirms the voting result. Unlike the other block, each member of the group builds this block on his own. The block needs to be build in such a way, that every member sill computes the same hash.

The signatures are created for the Merkle root of the list of elected delegates.

$$vB = (Id_b, \mathcal{M}^*_{gr}, [sig], pbh)$$

Suggestion Block

Suggestion blocks sB are used for confirming the suggestions sug that are explained above. Each block is build for a single suggestion. Beside the signed suggestions, blocks contain the delegates signature sig and the Merkle path mvp needed for the verification.

$$sB = (Id_b, sug, mvp, sig, pbh)$$

Confirmation Block

In periods where no suggestions are shared, and therefore no new blocks would be created, delegates can build a new confirmation block cB. These blocks confirm

the current group state. This allows members to ensure that no previous block was missed and their group state is still valid.

$$cB = (Id_b, mvp, sig, pbh)$$

4.1.5 Forks

Forks occur if two or more block are build on top of the same previous block. This can happen if two delegates build a block at the same time, and therefore use the same previous block.

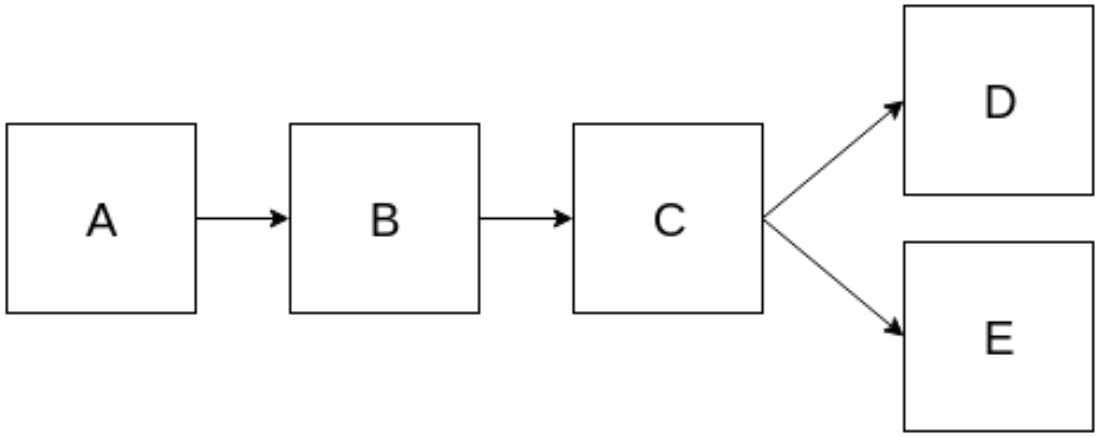

Figure 4.1: Block *D* and *E* created two branches.

If such a fork appears, the clients need to store both blocks. To resolve the fork, the delegate which builds the next block chooses the block he received first as previous block. This is no issue if both blocks are build for the same suggestion, since the information is still included in the chain. Otherwise the suggestion of the drop block is not part of the chain any longer and therefore the action is not applied to the group state. To reapply the action, a new suggestion needs to be committed.

However, it is not assumed that forks occur very frequently. This is due to the centralized messaging which ensures fast delivery of new blocks.

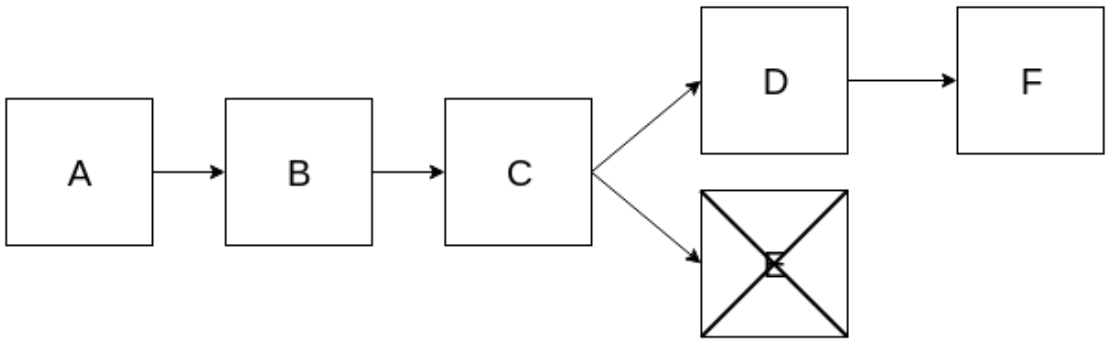

Figure 4.2: The fork is resloved by continuing block D. Block E is dropped.

4.2 Protocol

The following section describes the processes and messages that are needed to implement the Delegates Proof of Stake from above.

It is assumed that each user controls a public key pair, that can be used to sign messages and blocks. Every user has access to the public keys of other users, in order to verify this signatures. The required key and identity management is not part of this protocol. Such features are assumed to be solved by the messaging system itself and are not specific to group management.

4.2.1 Messages

Several message types are defined for this protocol.

- Text
- NewBlock
- Suggestions
- Vote
- VoteResult
- GroupAddition

These messages are send with a shared message container which contains information like the receiving group identifier ID_{gr}, the sender S, the message type

$type_{msg}$, the time the message was sent t, the messages signature sig, and finally the content c itself.

$$msg = (ID_{gr}, S, type_{msg}, t, sig, c)$$

If a message is received, the user checks if the sender S is a member of the group ($S \in \mathcal{M}_{gr}$), otherwise the message is dropped and a warning could be displayed.

4.2.2 Start New Group

Any user can start a new group at any time. To do so, the client needs to build and sign a new genesis block and share it with a *NewBlock* message. On receival, a user checks the blocks signature. If the block is valid, the client creates a new group with the members and group information from the genesis block.

4.2.3 Suggestions

Suggestions can be shared by every group member. A suggestion message contains the user M making the suggestion, a timestamp $time$, the suggestion type $type_{sug}$ and a field for the action $action$. Valid types are *Add*, *Remove*, *Update* and *StartVote*. For *Add* or *Remove* suggestions the *action* field contains the affected user, while it contains the groups *info* for *Update* suggestions. The message also needs to include a reference to the latest block and its block hash (ID_{bl}, bh) as well as a valid signature sig from the proposer.

$$msg_{sug} = (M, time, type_{sug}, action, ID_{bl}, bh, sig)$$

If a member receives a new suggestion and is a currently elected delegate, the receiver verifies the message and includes the suggestion into a new block if the the suggestion should be accepted. Before doing so, he needs to check that no other delegate already shared a valid block for this suggestion. The *StartVote* suggestion is an exception to this behavior. If such a suggestion is received and no election is currently running, all members start a new election. *StartVote* suggestions don't need to be signed by a delegate and be distributed as a new block first.

4.2.4 Confirm Suggestions

To confirm a suggestion, a delegate shares a signed *Suggestion Block* using a *NewBlock* message. Before doing so, the delegate needs to verify the suggestion. This includes the suggestions signature, as well as if the block reference is included in the chain and the action is valid. For example an action is invalid if the suggestion would remove an user, that is not member of the group.

If a *NewBlock* message is received, the members need to verify the block as well as the included suggestion. This verification contains the following steps:

1. Verify the message signature.
2. Verify the block signature.
   (a) Verify the election proof (Merkle verification path).
   (b) Verify the signature.
3. Compare the previous block hash.
4. Check if the new block creates an fork.
5. Verify that the included suggestion was not shared by the delegate himself.
6. Check if the suggestions block reference is less than N blocks old.
7. Verify that the suggestions block reference is included in the chain.

If the block passes all steps, the block is appended to the chain, and each member updates the group state.

If an *Add* or *Remove* suggestion is approved, all members need to update their member lists. For an *Add* suggestion, all delegates send their current chain to the new member using the *GroupAddition* message. If the block contains a *Remove* suggestion, each member checks if he is the user that got removed and removes the group from his group list if needed.

4.2.5 Adding New Members

If a new user is added to the group after a *Suggestion Block* was appended for an *Add* suggestion, each delegate shares his chain with the new member.

For the first *GroupAddition* message, the added member verifies the whole blockchain using the steps from above. If the chain is valid, the client creates the current group state from the chain and adds the group to the group list. For the messages that are send later by the other delegates, the new member just needs to check the latest block hash to compare the chains. If all chains matche, the added user knows that non of the delegates shared a wrong group state, otherwise the new member would need to contact the group to reslove the issue.

4.2.6 Voting

Per the definition above, the voting process needs two more messages, beside the *StartVote* suggestion. The *Vote* and *VoteResult* messages are used to share the vote itselt and, in the second step, the confirmation. However even if the messages use different types, the information that are exchanged are mostly the same.

Each message includes the voters user identifier V, a list of votes (either the own vote or the computed result) $[vote]$, the Merkle root of the votes mr as well as the signature of the Merkle root sig. Each message also contains the hash of the h_{sug} that started the election.

$$msg_{vote} = (V, [vote], mr, sig, h_{sug})$$

The *VoteResult* message also contains the list of all received *Vote* messages $rVote$ for this election.

$$msg_{VoteResult} = (V, [vote], mr, sig, h_{sug}, [rVote])$$

If voting messages are received, they need to be validate and stored for later procession. The validation contains the following steps.

1. Compare the suggestion hash h_{sug}.
2. Verify that the voter has not already voted.
3. Check that the vote list is valid.
   (a) Contains the correct amount of votes.
   (b) Contains no duplicated votes.
4. Verify that the Merkle root mr matches the votes $[vote]$.
5. Verify the Merkle root signature sig.

4.2.7 Confirmation Blocks

If no new block was build in the last N hours, the first delegate to get online after this time, can build and distribute a new *Confirmation Block*. A delegate therefore needs to check if such a block is required each time he gets online and has processed all received messages.

4.3 Simulation

The designed protocol is tested using a simulated group chat. The goal of this implementation is to see if such a protocol can work in an asynchronous environment. It is not tested how the implementation behaves under attacks from malicious users. Every client acts according to the protocol.

In this setup several clients are connected to a simple message server and exchange random group management messages. The clients simulate offline phases in which they do not process new messages, to test the ability of the system to work in an asynchronous environment.

4.3.1 Assumptions

The simulation is designed with several assumptions. These are made to simplify the messenger part of the system.

A messenger system can provide contact discovery as well as key exchange and user verification. Therefore this functionality is simplified in the simulation by using a pre shared user list that is included in the clients. This list also contains the users public keys, that can be used to verify signatures.

Furthermore it is assumed that, message encryption, or further protection such as protection from message replay attacks are handled by the used messaging and encryption protocol, such as the Signal protocol. These properties are considered solved and not part of this implementation. The implementation also assumes that all messages are delivered and are received in the correct order.

4.3.2 Message Distribution

The client can exchange messages using a centralized message queue server. Group messages are delivered like in Signal. Each client can be reached via his own channel on this server. A group message is then send to the channel of every member.

Each client sleeps a random amount of time, to simluate asynchronicity. If the client gets back online, it accesses the messages from its channel and starts processing them.

4.3.3 Actions

Each time a client gets online, a random decision is made, if and what action the client should perform. For example the client can decide to share an *Add*, *Remove*, *Update* or *StartVote* suggestion. The affected users or group information are also chosen randomly.

Chapter 5

Implementation

5.1 Introduction

Due to the focus on Delegated Proof of Stake, this chapter only covers the parts of the implementation which are relevant to this parts in detail. Supporting code like contact managing, messages exchange and key handling is only covered on a higher level.

5.2 Environment

The programming language for this implementation is C# together with dotnet core 2.2. C# is an object oriented language that can be used on Windows, macOS and Linux with the dotnet core framework. This combination is chosen, since the author has experience with dotnet core development under Linux.

For the cryptographic tasks, like hashing and digital signatures, libsodium [5] is used. This is an easy to use, cryptography library which is available in dotnet core via the Sodium.Core [4] nuget package. The author has worked with libsodium before.

RabbitMQ [20] is used for message exchange. RabbitMQ is a message broker that is easy to install and has client-side libraries for many languages and systems. The broker is installed inside an Virtual Box virtual machine [25] running Ubuntu Server 18.04. This allows the server to be easily restarted if needed and allows the installation to be moved to another system. The server installation uses a

configuration that allows any client to connect and open channels without the need for authorization or authentication. Since the system is only used for development and is not connected to any external sources, this is not considered as a security issue that is relevant for this implementation.

5.3 Foundations

The following section explains parts of the implementation that are not directly relevant to the blockchain code, but are needed to provide the necessary infrastructure.

User Identites

Each client has a prefilled `ContactDB`. This is a static class with 50 user IDs and private/public key pairs. The user IDs are integers running from 1 to 50.

Each client is started with an ID as command line parameter. This ID is the clients user identifier. On startup the clients pick their private key from the `ContactDB` and stores it together with his ID inside the static class `MySelf`. The other users public keys are loaded into a *Dictionary* where they can be retrieved using the user IDs. Every user also uses the ID to register a channel at the message broker to receive the messages. This way, in order to send a message, the clients just need the others user IDs.

The key pairs where generated using a temporary dotnet program with libsodiums `crypto_sign_keypair` and pasted into the main implementation.

Using the `ContactDB` all clients already know each other and no further contact discovery and key exchange needs to be implemented.

Messages Exchange

All messages are exchanged as JSON serialized objects. This allows fast and easy serialization and deserialization into the correct object using the Newtonsoft.JSON nuget package[19]. As defined above there are several message types that are exchanged using the same container. To be able to decode the content into the correct object, the `Type` field, which is implemented as an `Enum`, is used to select

the correct message object. The content of the message is encoded an JSON string inside the message container.

The JSON messages can be send by using the `Send` method of a static class. The `MessageConnector` class handles the connection to the broker and maintains the channels. The class is initialized at startup using the user id to open the channel and the host address of the RabbitMQ installation.The class can be called from anywhere in the code to send messages. The `MessageConnector` is also used to register a `consumer` which invokes a event when a new message is delivered to the clients channel. The consumer is of the type `EventingBasicConsumer` which is defined in the RabbitMQ nuget library.

When the event for a new message is invoked, the received message is stored inside a temporary `MessageDB`. This is a class with a `Dictionary` containing a list of messages per group. The MessageDB can be used to get the next message for procession or check if a specific type of message, like *Votes* or *NewBlocks* are still stored in the message backlog.

Main Routine

The main routine is implemented in the `BaseChatApp` class. It is invoked using the `Run` method from the programs entry point, the `Main` method, after the connection to the message broker is established. `Run` starts a `Do While()` loop, which loops until a text message with the content "end" is received. This allows all running clients to be shutdown at once using a single message that is send to every client.

Inside the loop four steps are performed:

- Sleep
- Process messages
- Check further actions
- Send a new random action

Sleep

To simulate asynchronicity, each client sleeps for a random amount of time before it gets *online*. The sleep time is chosen between three different time slices with a

different probability x.

$$time = \begin{cases} rand(0.1s, 6s), & 0.0 \leq x < 0.7 \\ rand(6s, 12s), & 0.7 \leq x < 0.9 \\ rand(12s, 24s), & 0.9 \leq x < 1.0 \end{cases}$$

[1]

12 seconds inside the simulation are considered to be 24 hours in real time, such that the maximal sleep time would equal 48 hours.

The different time slices are included, such that it is possible for a client to miss whole voting, while keeping the overall offline time shorter. If picking a random time from the whole range, higher sleeping times would be more common, and such the simulation would run slower and with less participation.

Process Messages

After the clients get online, or rather wake up from sleeping, the messages receivedand stored in the `MessageDB` during this time are processed one by one.

Based on the messages type, the matching `MessageProcessor` is initialized and called. Each `MessageProcessor` implements the `IMessageProcessor` interface, which defines the `Process` method. This method takes the message and returns a tupel `(bool res, string msg)`. The Boolean is true if the message could be processed without issues and the string includes a message that is printed to the console, to either show the messages content or an error description. Each message type uses its own `IMessageProcessor` implemenation. The `Process` method is used to verify and validate the message and forward it to the correct group, where it can be further processed.

Further Actions

If no messages are left in the `MessageDB`, each group is checked for open actions, such as sending a vote or confirming new suggestion. Open votes are signalized using a Boolean flag, while open suggestions are stored in a list.

Voting and suggestion handling is explained in detail later.

[1] $rand(a, b)$ retruns a random number between a and b.

New Random Action

Each time a client is online, it is decided randomly, if the client should send a suggestion to a group. There is a probability form 15% (lagre groups) to 35% (small groups), that such a message is send. The probability is implemented like this, to keep smaller groups still make progress, since the probability that at least on client shares a suggestion is higher.

If a suggestion should be shared, the client picks a random type and builds a new suggestion wich either changes the group name randomly, add a new random user from the `ContactDB` or removes a random member from the group. A *StartVote* suggestion is only shared in 20% of the cases where it is selected. New votes should be less likely than a change of the group name.

5.4 Blockchain

The blockchain implementation can be split into three components. Suggestions, blocks and the final blockchain.

5.4.1 Suggestions

As mentioned above, each client can share a new suggestion if he gets online. To do so, he calls one of the `BuildAndShare*` methods of the `Group` class. The methods are located there, since this class includes all required information such as the latest blocks or the member list, which is needed to pick a valid user for add or remove suggestions.

To build a new suggestion, the client first needs to retrieve the latest block hash as well as the block ID from the chain. If the chain is currently in a forked state, the hashes of the forks also need to be included in the suggestion. This allows the suggestion to be valid, independent of the branch that gets continued.

After the suggestion object is initialized, with its type and action message, it needs to be signed using the `Sign` method. This method uses the `UserId` and `PrivateKey` from the `MySelf` class. The signature is generated over a `byte[]` which is build by combining the action messages hash and the byte representation of the user Id, the suggestions type, the creation time ans the block ID appended with the latest block hashes.

After signing, the suggestion is serialized into JSON and fed into a new message container, which also needs to be signed. Now this message can be send to the other group members.

Listing 5.1: Build and send a new *Update* suggestion for a random name

```
public void BuildAndSendNewNameMessage()
{
    var chainHeader = Blockchain.GetLatestBlock();

    var byteName = new byte[4];
    new Random().NextBytes(byteName);
    var name = Convert.ToBase64String(byteName);

    var nameSuggestion = new Suggestion(Actions.Update, name
        , chainHeader.bId, chainHeader.hash);
    nameSuggestion.Sign();
    var suggestionJson = JsonConvert.SerializeObject(
        nameSuggestion);
    var msg = new Message(MessageType.Suggestion,
        suggestionJson, GroupId);
    msg.Sign();

    Send(msg);

    AddSuggestionForConfirmation(nameSuggestion);
}
```

Listing 5.2: *Update* suggestion message example

```
{
    "SenderId": 1,
    "GroupId": 1337,
    "Content": {
        "SuggestedBy": 1,
        "SuggestedAt": "2019-07-19T11:11:33.6763106+02:00",
        "Action": 2,
        "ActionMsg": "bqFA+Q==",
        "ChainHeader": [
            "sKfr/UR8IeLfaFKkoGA0jtiE7UJvyYYgzGPNy700dRU="
```

```
        ],
        "BlockId": 1337,
        "Signature": "jvLb8+
           i10wvFh1wj29PHFyXqjVyrRujsR4W0QfHkTcEMpPZt5fjiENkWn
           /p1exhO6WrrskhCKz6bM+seY9zcAg=="
    },
    "Type": 4,
    "SentAt": "2019-07-19T11:11:33.6838976+02:00",
    "Signature": "Qe2c3iAXMNI6rQQ7iQfI8g3iN/
       gEdAifIuMHHawKfN4BG01GnW08hNeXyPGA3zY+4
       zIn5CNEbDYiG0gAY6bLBA=="
}
```

If a client then receives the suggestion, it is forwarded to the `SuggestionMessageProcessor`. There it is checked that the receiver knows the group the message was intended for and if so, the sender is indeed a member of this group. If that is the case, the suggestions JSON string is deserialized from the message content and the signature is validated using libsodiums `PublicKeyAuth.VerifyDetached`. The senders public key for this verification is taken from the `ContactDB`.

If the suggestion is of type *StartVote*. the `StartNewVote` method of the group is called, where a new vote is initiated. Otherwise `AddSuggestionForConfirmation` is called. This method checks if the client is currently an elected delegate for the group. If so, the unprocessed messages in the `MessageDB` are checked for a valid new block that includes this suggestion. Since the suggestion was already confirmed by another delegate in this case, it does not need to be processed further. If no new block is found, the suggestion is appended to the `OpenSuggestions` list of the group. This list is later accessed to build new blocks, after all open messages are processed.

For any valid suggestion, a message is printed to the screen, including the senders ID, and the suggestions information.

5.4.2 Blocks

The following section describes the implementation and usage of the three blocks: `SuggestionBlock`, `ConfirmationBlock` and `GenesisBlock`.

The fourth block `VotingBlock` is explained in the voting section 5.5.

Implemenation

All different block types derive from the `BaseBlock` class.

All blocks are stored inside the `Dicionary<int, BaseBlock> Blocks` of the `Chain` class. The block ID is used as key. As opposed to using the hash as key, this allows faster access time, since the hash does not need to be computed before accessing a block or when iterating over the chain. Further the `Chain` class is used to retrieve information from the blocks or to verify and append new blocks.

Listing 5.3: `BaseBlock` class

```
class BaseBlock
{
   public int BlockId {get;set;}
   public BlockType Type {get;set;}
   public byte[] VotingMerkleRoot {get;set;}
   public byte[] PreviousBlockHash {get;set;}
   public virtual byte[] Hash() => throw new
      NotImplementedException();
   public virtual void Sign() => throw new
      NotImplementedException();
}
```

BlockId The `BlockId` is the unique block identifier, that is increased by one for each new block.

Type `Type` is a `enum` which representes the block type, such that the `BaseBlock` can be casted into the correct block after a `BaseBlock` object is received or loaded from the chain.

VotingMerkleRoot The `VotingMerkleRoot` contains the Merkle root of the latest voting result. By including the root into each block, every information that is needed for verification is, together with the delegates verification path, directly available and does not need to be cached or retrieved from the chain. The root

just needs to be compared with the `VotingMerkleRoot` of the previous block to ensure the correct election result ist used.

PreviousBlockHash As the name suggested, this is the field used to store the previous block hash. This hash *chains* the blocks together and make the chain immutable.

Suggestion Blocks

`SuggestionBlock` extend the `BaseBlocks` with all fields that are needed to add suggestions to the chain. This includes the suggestion object itself (`Suggestion`) as well as the delegates Merkle verification path (`DelegateVerificationPath`) and the signature (`Signature` and `SignedBy`).

The blocks signature is build over the suggestions hash, the `VotingMerkleRoot`, the `PreviousBlockHash` and the byte representation of the block ID and the signers user ID. These byte arrays are concated into a single array which is then signed with the clients privat key. The block hash create over the same information as the signature, but also includes the signatures bytes.

Building After all messages are proceeded in the main routine, the `ConfirmSuggestion` method is called for each item in the `OpenSuggestion` list per group. As mentioned above, this list is only filled if the client is a delegate. Inside the loop, the client also checks is a new block is available in the `MessageDB` to prevent forks. If a new block is available, the confirmation is canceled until the next time the client get online. In `ConfirmSuggestion` the client makes a random decision if the suggestion should be accepted. For testing purposes a confirmation rate of 65% was chosen in this simulation. In a real world implementation this decision would be made manually by the elected user. If the suggestion is accepted, the groups method `SendAndApplySuggestion` is called. This method checks if the suggestion is still valid with the current chain state. The following properties are verified in this step:

- The suggestion does not try to remove the user himself.
- The suggestion does not try to add a user that is already a member.
- The suggestion does not try to remove a user that is not a member.

- The referenced block is less than three[2] blocks old.
- The referenced block hash is included in the chain.

Beside the check to not remove oneself, it is more useful to verify this properties just before confirming the suggestion, since the chain could have changed if *NewBlock* messages followed the addition of the suggestion into the `OpenSuggestion` list.

If the verification succeeds, the delegate builds a new `SuggestionBlock`. The new block is then signed and converted into a JSON string. The JSON message is included into the message container with the *NewBlock* type set and send to the groups members.

Receiving If a new `SuggestionBlock` is received, it needs to be verified, appended to the chain and, finally, applied to the group state.

The `NewBlockMessage` content is first casted into a `BaseBlock` to access the `Type` property and cast the block into the correct type. The block is then fed into the `Chain` class for verification and addition using the `ValidateAndAdd` method. Firstly the `ValidateBlock` method is called to check the properties that are independent from the chain, such as:

- The blocks signature
- The Merkle verification path
- That the suggestion is not signed by the block signer

If this verification succeeds, the block is compared against the chain. Now it is checked that the included `VotingMerkleRoot` and `PreviousBlockHash` matches the data from the latest block and the correct `BlockId` is used. If the hash or the ID are not as expected, it is checked if the block opens a new fork. If the `BlockId` equals the latest block ID the new block is appendend to the `CurrentFork` list. If the new ID is not equal the latest block but the new block hash is included in the `CurrentFork` list, it is needed to switch to this branch. Branches are switched by replacing the latest block from the chain with the matching item from the fork list. The last verification step, is to validate the suggestion

[2]The block depth of three was chosen to test the suggestion expiration inside the simulation. For real world usage, a solution better than a fixed depth, needs to be defined.

using the criteria from the suggestion validation described above. If the block is valid it can be added to the `Blocks` dictionary. Also the `LatestBlock` reference and `LatestBlockTime` are updated and the `CurrentForks` list is cleared.

Finally the `ApplyBlock` method of the group is called to update the groups state. For *Add* or *Remove* suggestions the member list is reloaded from the chains data. In case of an `Add`, the client also sends a `NewGroupMessage` to the added user if he is a delegate.

Confirmation Blocks

Confirmation blocks reuse the `SuggestionBlock` class. If this block is initialized with `null` as suggestion parameter, the block type is set to `Confirmation` and the `Suggestion` property is left empty.

Delegates build new confirmation blocks, if no new block was appended to the chain in the last 6 seconds, which would map to 12 hours in reality, using the `LatestBlockTime` field of the chain. Such a block is only accepted if the signer differs from the latest block signer. Otherwise the block is validated using the same methods than the `SuggestionBlock`.

Genesis Blocks

The `GenesisBlock` is used to start a new group and is the first block in the groups chain. Like all blocks, this block type also is derived from the BaseBlock and extends it with a `GroupInitiator` property as well as the list of the first members user ID. The signature created by the initiator is stored in the `Signature` byte array. To allow all members to act as delegates, the `VotingMerkleRoot` is build over user IDs in the `Members` list.

The blocks signature includes the `BlockId`, the `GroupInitiator` and the `VotingMerkleRoot`, since this root is created over all members, the member list does not need to be included itself. The block hash includes also the `Signature` bytes.

Building To create a new `GenesisBlock` the client calls the blocks constructor with a new, preferabely random, group ID. This group ID is used as first block ID and, in this simulation, as first group name. Further the contructor take the intial

member list.

After signing, the block id serialied to JSON and send to the members as a *NewBlock* message.

In this simulation, such a block is only build once. If a client is initialized with the ID 1, it creates this block at startup with the group ID *1337* and a defined number N of new members. This member list contains simply the range of the first N members. E.G. for a group with $N = 5$, the group startes with members $\{1, 2, 3, 4, 5\}$.

Receiving If a block of type *Genesis* is recieved in the `NewBlockMessageProcessor` the block is verfied by checking the `Signature` and the `VotingMerkleRoot`. For a valid block, a new group is initialized using the block ID as group ID and as group name. Also the chain is created using the received block as first block. The new group is finally added to the `GroupDb`, if not already a group with the same ID exists.

5.4.3 Chain Operations

There are several methods that are used to analyse the chain. This section explains two of them shortly.

Load Members

To update the member list, the clients iterate over the chain, in order to retrieve the current Members. This is done by using a dictionary with an entry for each user ID that appears in the chain as key and a Boolean as value. At first entries for the members included in the `GenesisBlock` are created an initiated with `true`. While iterating over the chain, for each *Add* and *Remove* suggestions the boolean value is inverted. If the whole chain is consumed, all dictionary entries with the value `true` are currently a member of the group.

To iterate over the chain, a while loop is used, that runs from the latest block ID, to the ID of the genesis block and accesses the blocks from the `Blocks` dictionary using this block ID. In each loop the current block ID is decreased by 1.

Verify

If a user gets added to a group, the delegates send him the whole blockchain. The new member then needs to verify this chain. The `Verify` method iterates over the chain and verifies each blocks signature, the previous block hashes and for `SuggestionBlocks` the included suggestion.

5.5 Voting

Beside the blockchain and blocks itself, voting is the other key part of a Delegate Proof of Stake protocol.

Start Election

An election is started by sharing a *StartVote* suggestion. In this simulation, such a suggestion can be shared at random by every member, or must be shared, if a delegate approves a suggestion that removes a currently elected delegate from the group.

Every time such a suggestion is received, all group members initiate a new election (if not a previous election is still running). To start an election in the `VotingState` class, the client first needs to compute the amount of delegates to elect and the vote weight for each member.

The amount of delegates needed depends on the group size.

$$\#\mathcal{M}_{gr}^{*} = \begin{cases} 2, & \#\mathcal{M}_{gr} \leq 11 \\ 3, & 12 < \#\mathcal{M}_{gr} \leq 18 \\ \sqrt{\#\mathcal{M}_{gr}}, & 19 < \#\mathcal{M}_{gr} \leq 200 \\ 15, & \#\mathcal{M}_{gr} > 200 \end{cases}$$

The square root of the members is used, since the amount of delegates does not need to grow linear with the amount of members. Two is chosen as lower bound, such that both delegates still can share suggestions, that then needs to be signed by the other. The square root would return member-to-delegate distribution rates of $\frac{1}{3}$ or $\frac{1}{4}$ for small groups. This amount of delegates would not be needed in such groups, therefore the amount is limited to 2 or 3. The upper bound of 15 was

chosen to make voting decision easier in large groups and limit the amount of delegates that need to be trusted.

The voting weights are calculated from the blockchain. Each user has it is own weight, a factor that is applied to their votes to increase the votes relevance. The weights are calculated from two parts. The amount of approved suggestions and *SuggestionBlocks* build by each user.

Two dictionaries are used to count the amount of suggestions s or relevant blocks b for each member stored in the chain. The final factor is then computed using these dictionaries.

$$w = 1 + \frac{\frac{s}{2max(S)} + \frac{b}{2max(B)}}{2}$$

[3]

As result, the user that contributes most to the group can get a factor up to 1.5, while other factors are calculated relative to this work.

This formula was chosen, since it also rewards users that are not elected delegates and "only" contribute through suggestions.

Voting Window

Using the `StartNewVote` method of `VotingState` a new election is started for the suggestion. If the method is called, while the voting is still in *VotingWindow*, the `SendOwnVote` flag is set to true. The *VotingWindows* is defined as the suggestions send time plus 12 seconds (would equal 24 hours). After processing all new messages, the client checks the flag, and sends a new `VoteMessage` if needed. Such a vote includes the user IDs of each member the client wants to vote for. If the group needs to elect 5 delegates, the user also needs to vote for 5 delegates. In this simulation the votes are picked randomly and the client can also decide to not vote at all.

Beside the votes, the message also contains the hash of the suggestion and the signature of the Merkle root build from the vote list.

For each `VoteMessage` the clients verify that the vote was send during the

[3] $max(S)$ and $max(B)$ are the maximum of build signatures or block by a single user stored in the dictionaries.

`VotingWindow` using the messages time stamp, that the included signature is valid and the the voter has not already shared a vote before. Further it is verified that the vote list only contains unique entries and is of the expected length (amount of delegates to elect). Valid votes are stored in a dictionary after applying the members voting weight.

Each time a client gets online, the voting state needs to be checked for new steps first. The `CheckForNextStep` method checks if the *VotingWindow* or the `ConfirmationWindow` are now ended, that the next action like sending the voting result or build the final `VotingBlock` can be invoked.

Confirmation Window

If the *VotingWindow* is closed and the *ConfirmationWindow* has started, the next call of `CheckForNextStep` invokes the `OnVoteFinished` event, if no new relevant message are left in the `MessageDB`. Otherwise this event is invoked after this votes where verified and stored. On the `OnVoteFinished` event, the client computes his own voting result over all valid votes he received. This is done by taking the N users keys with the heighest value from the `Votes` dictionary, where N is the number of delegates which need to be elected. This result is only computed, if more than 50 % of the members shared a vote, otherwise the election is closed and no further action are taken. If enough members contributes, the clients must build a `VoteResultMessage`. Beside the result list and its Merkle root signature, this message also needs to include all received votes and the latest block hash. Unlike as for the suggestion, this time only the latest block is needed and possible forks can be ignored. The votes need to be included, such that every client can verify that it received the same votes. The signatures ensure that the votes are forwarded without being changed by the `VoteResultMessage` sender. This step is needed to resolve the *Byzantine generals problem* [10], which describe the issue of one or more parties lying in a distributed system. The usage of signatures to compare all received messages can prevent this issue.

The latest block hash needs to be included to resolve possible forks. The branch which is referenced most, is going to be used as previous block. The other branches are dropped. This step is needed since every client builds the *VotingBlock* himself. Therefore there needs to be a way to ensure, that every client uses the same previous block.

Like the `VoteMessage` the `VoteResultMessage` are also validated through

checking that the message contains the correct suggestion hash and a valid vote list (correct length and no duplicated entries). If the message is send before the *ConfirmationWindow* is closed and the Merkle root signature is valid, the result is stored. The different result combinations are stored together with the signatures in the `VotingResults` dictionary, where the Merkle root is used as the key.

Result Confirmation and new Block

The check for the end of the *ConfirmationWindow* works the same as the check for the *VotingWindow*. If the window is closed and all messages are processed, the *OnCollectionFinished* event is invoked. In this event, the final result is computed if at least 2/3 of the members shared their result. If the `VotingResults` dictionary contains only one entry, every client that shared his result, computed the same output. In this case this result can be used to build the new *VotingBlock* directly. Otherwise the members need to compare the attached votes of each submitted message to spot the difference that leaded to the different result. In this simulation, every client behaves according to the protocol, such that different result should not occur. However, real world implementations would need to take further actions like excluding the votes of members that shared different votes [4].

However, at the end every member needs to build his own new *VotingBlock*, even if he missed both windows. This block includes the previous block hash, that was mostly shared with the *VotingResultMessages*, the suggestion that started the election and a list with the elected user IDs. This list Merkle root is stored in the `VotingMerkleRoot`, such that further blocks can be validated against it. Further the block includes every signature of this root. Such that the block is not only confirmed by a single member, but by every member that computed the election result and shared it using the `VotingResultMessage`.

Finally the new block can be appended to the chain, where it is first checked if it is needed to switch the branch. If the elected delegate list contains the clients user ID, the client computes his own Merkle verification path and stores it in the `MyDelegateVerificationPath`, where it can be loaded to build new blocks.

[4] E.G. Member A sends the vote $\{B, C\}$ to B and $\{C, D\}$ to C and D.

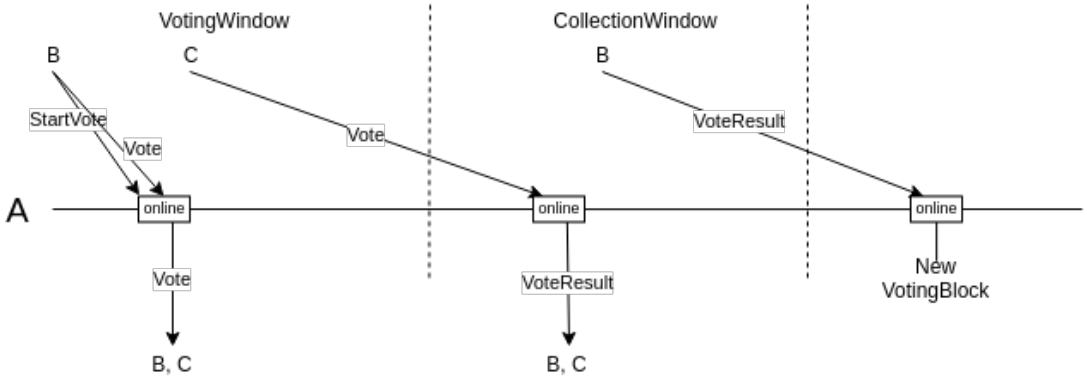

Figure 5.1: The election process out of A's point of view.

5.6 Merkle Tree

In this impelmentation the Merkle tree is used whenever a list needs to be signed, or to verify the delegates. The Merkle tree is a hash tree, a datastucture that allows to verify that lists are equal by comparing the root, or to verify that an item is included in a list, without checking every item. Instead a path is defined. If following the path results in the correct Merkle root, the item is prooven to be item in the list.

This implemenation uses three methods: `GetRoot`, `GetVerificationPath` and `VerifyPath`.

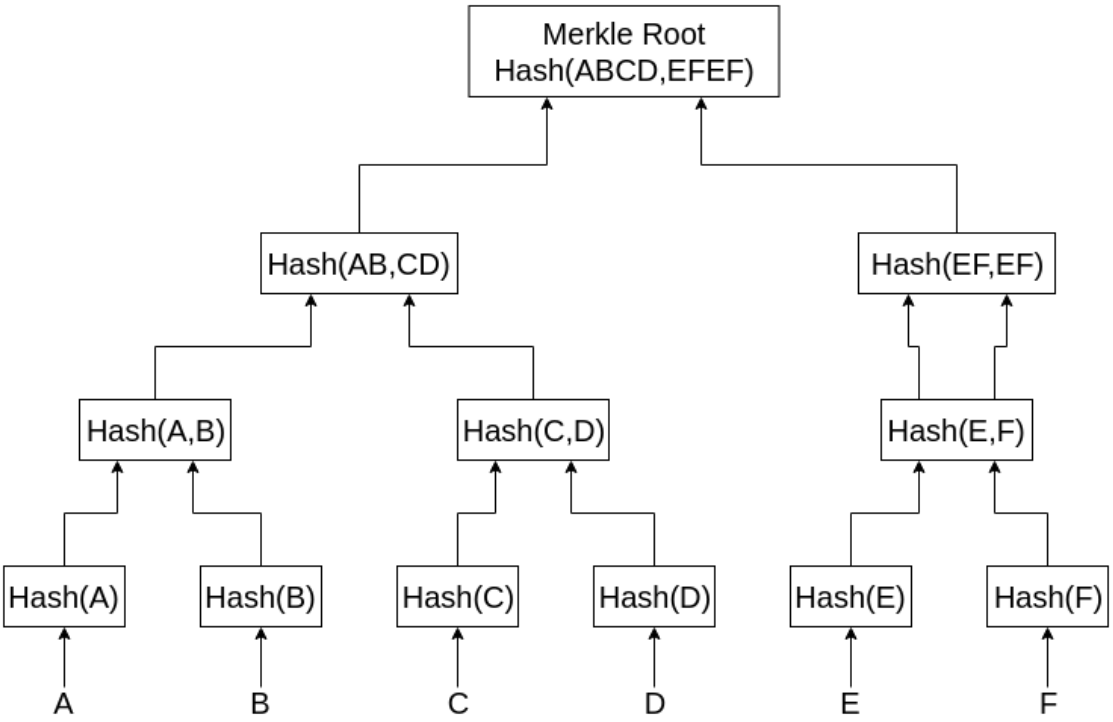

Figure 5.2: A Merkle tree with 6 leafs.

Get Root

The root of a Merkle tree is computed iterativly. The first step is to hash every value. This is done using libsodiums `GenericHash` BLAKE2 implemenation. The new hashes are stored in the `hashes` list. A `while` loop is used to iterate until this list only contains one item, which would be the root. The inner loop iterates over the `hashes` list with a step width of two and combines two items to a new hash. As shown in figure 5.2, if the number of hashes is odd, for the last item the same hash is used twice, instead of its neighbor. The result is stored in the `nxtLevel` list, which replaces the `hashes` list after the iteration is finished, such that the next level of the tree can be computed. Before hashing the nodes, they are swapped, such that the smaller item is the first. This removes the need to remember the position of a node when buildung the verification path.

Listing 5.4: Compute the Merkle root from a list of byte arrays.

```
1 public byte[] GetRoot(List<byte[]> Values)
2 {
3     var nxtLevel = new List<byte[]>();
4     var hashes = new List<byte[]>();
```

```
5
6     foreach (var value in Values)
7        hashes.Add(Hash(value, HASH_LENGTH));
8
9     byte[] children = new byte[HASH_LENGTH*2];
10    // iterate until one item is left, last item is the root
11    while (hashes.Count > 1)
12    {
13       for (int i = 0; i < hashes.Count; i += 2)
14       {
15          byte[] A = hashes[i];
16          byte[] B;
17
18          // reuse first if last item and odd number of
               nodes
19          if(i + 1 >= hashes.Count) B = A;
20          else                B = hashes[i+1];
21
22          // Swap nodes, such that the smallest is the first
23          if(ToInt64(A) < ToInt64(B))
24             Swap(A, B);
25
26          children = Combine(A,B);
27
28          // Add new hash to the next level of the tree
29          nxtLevel.Add(Hash(children, HASH_LENGTH));
30       }
31       // replace the hashes with the next level
32       hashes.Clear();
33       hashes.AddRange(nxtLevel);
34       nxtLevel.Clear();
35    }
36
37    return hashes[0];
38 }
```

Get Verification Path

The `GetVerificationPath` method uses the same loops as `GetRoot`, but further remembers an `index`. This `index` is initialized with the index of the item, which the path is created for and updated each time, the inner loop hits this index. If this happens, either node `A` or `B` are appended to the path, and `index` is set to the index of the new hash in the `nxtLevel` list. If node `A` or `B` is selected depends on the index. The opposite of the matching index needs to be selected, such that the path contains the *neighbor* nodes. E.G. if the index of node *A* is hit, *B* needs to be added to the chain.

If the final root is consumed, the `path` list contains every hash that is needed.

Verify Path

`VerifyPath` takes three arguments: 1) the start item, 2) the path and 3) the root hash.

After the start item is hashed into `hash`, it is iterated over each hash in path. Both hashes are, again, swapped, such that the smaller hash is first in the combined array, which is than hashed into the `hash` value. The path is correct, if the last computed hash equals the root hash.

Listing 5.5: Verify a Merkle path.

```
1
2  public static bool VerifyPath(byte[] start, List<byte[]>
       path, byte[] root)
3  {
4      var hash = Hash(start, HASH_LENGTH);
5
6      byte[] children = new byte[HASH_LENGTH*2];
7      foreach (var item in path)
8      {
9          var A = hash;
10         var B = item;
11
12         if (ToInt64(A) < ToInt64(B))
13             Swap(A, B);
14
```

```
15          children = Combine(A,B);
16
17          hash = Hash(children, HASH_LENGTH);
18      }
19
20      return Compare(hash, root);
21  }
```

Chapter 6

Analysis

6.1 Security Analysis

The following section analyses the security of the designed protocol, against the properties and adversaries defined in the security model in section 2.2. Possible weaknesses in this implementation are not included in this analysis, since the simulations goal is to see if such a system can work and is not designed and intended for real world usage.

6.1.1 Closeness

Both *Additive* and *Subtractive* closeness are achived by the protocol. None of the adversaries would be able to forge a message that leads to a violation of this rule. External adversaries, such as malicious user or server, can not build message since they lack information such as the group ID or the latest block hash. Further the messages would be blocked since the sender is not in the member list of the group. A malicious member would need to trick a delegate to sign his malicious suggestion, since he would not be able to sign and build a new block for his own suggestion. To perform a successful attack, the adversary would need access to two members, with one being a delegate.

6.1.2 Privacy

Privacy toward the network or the server are achived if the messenger uses end-to-end encryption. The server only needs to see the receiver in order be able to exchange group messages, unlike WhatsApps system, where the whole group state is known. However, using traffic analysis, the server would still be able to retrieve information about the group. For example, if one user sends messages of the same size to the members B, C and D at the same time and B later sends messages to A, C and D, the server can assume that A, B, C and D form a group. The server is even able to perform this analysis if further protection such as Signals Sealed Sender is used.

Furthermore the privacy of former group members is violated. Information about the membership are stored inside the groups chain, and are therefore accessable by all new members. By encrypting the user IDs inside the suggestions and exchanging the decryption key outside of the chain with the current members, this information can be protected from new members. Such measurements would further increase the protocols complexity and prevent the ability to build the member list from the chain.

6.1.3 Same State

That all members use the same state is ensured by including and verifying the latest block hash in suggestions and blocks. If the hash mismatches the client can request the chain from the others members and resync its state.

A mismatch can occur if a malicious network or server drops messages to a user. With end-to-end encryption the adversaries can not detect group management messages and needs to drop all messages. With *Traceable Delivery* such an attack can be detected and the state can be recovered if the client is able to receive messages again.

The same state would also be violated if a malicious delegate builds different blocks for different users. This can be detected with the next suggestion or block from another member, such that the malicious delegate would be detected. Further actions, such as a new vote to replace the delegate and dropping the delegates latest blocks, can then be taken.

The latest block hash can be included in every group message to speed up the detection of issues.

6.1.4 Further Issues

The usage of static keys for signing and verifying signatures can result in further issues.

Post Compromise Security

In the current design and implementation, one user always uses the same key to sign messages and blocks. Once this key is compromised, an adversary would be able to impersonate the user. As a result, the protocol provides no *post compromise security* (or *future secrecy*).

New or lost Device

If a user looses his device, or switches to a new mobile phone, he can not access his old private key. A new key pair needs to be generated and distributed. However, the old public key is still needed to verify old blocks or suggestions. This further increases the requirements for the key management in a real world deployment.

6.2 Blockchain

The following section take a look on the blockchains components and their open issues.

6.2.1 Suggestions

The concept of *Suggestions* allows every member to contribute transparently to the group. Furthermore this concept helps to prevent attacks from malicious users. Since no member is allowed to sign his own suggestions, a second elected member is needed to verify and confirm the suggestion.

An attacker could build a private chain which removes or changes suggestions, compared to the groups real chain. If he would be able to introduce this chain to the group, he could manipulate the state. This attack is prevented using *Transactions-as-Proof-of-Stake*, where each suggestion references the current state of the groups state. A malicious actor is not able to create such a private chain, that removes old

blocks and rebuild the chain, since doing so breaks the other suggestions chain reference. The block hash included in the suggestion, is not present in the attackers chain.

In this protocol, no *Leave* messages are defined. Instead a member would need to send a *Remove* suggestion for himself. The delegates then could delay or prevent the leave of the user by ignoring this suggestion. As a result, the *Leave* use case is not coverd by a suggestion based design. To allow users to leave groups directly, a new block type would need to be included, which can be signed by the leaver himself. This increases the protocols complexity and the possible attack surface since more verifications need to be implemented.

6.2.2 Voting

Voting is the most complex part of the protocol. This complexity results out of the need to solve the *Byzantine Generals Problem* which requires a second round of messages to confirm the result and detect issues. Another source of complexity is the synchronous setup with two time frames, which needs to be implemented in a asynchronous environment where it is possible that messages needs to be processed outside of the original time frame. However the designed voting protocol worked within the asynchronous scope of the simulation.

Election Days

The *Election days* with the fixed time frames add a synchronous component to a otherwise asynchronous protocol. This requires the system to ensure that states and messages are checked and processed in the correct order, and the client can not just pick up their work from where they left. Further this approach opens new attack vectors. A malicious member can forge the send time of a vote, such that it is include by members that haven't computed their *VoteResult* yet, even if the *VotingWindow* is already closed. Even thought such an attack can be detected, since the included votes in the result messages differ, it increases the risk of a election to result with an invalid result. This kind of attacks can be prevented if all messages are attached with a *server-receive-time*, which than can be used to check if the message was send within the correct time frame, instead of trusting the senders time stamp.

Another possible attack is that the suggestion message, which initiate the vote,

could be prevented from reaching a member, either because it is lost in transit, or was not send to the member by the creator of the suggestion in the first place. In this case, the member left out would fail to validate the following vote messages, since he has not initiated an election. Nor would he be able to share his own vote in the election. To resolve this issue, the protocol would need to be tweak in such a way, that it's also possible to initiated an election from the suggestion attached to the votes.

Furthermore the need to include all votes in the *VoteResult* message requires large data usage for large groups. In group with 50 members, a vote process can include up to 4.900 messages in total (50 Members send 2 messages to the 49 members $50 * 49 * 2$). The *VoteResult* method would include 3200 bytes ($64 * 50$) of signatures (each users vote signature). A single client needs to handle a total of 163.200 bytes ($64 * 50 + 64 * 50 * 50$) of signatures per election. The amount of messages ($\#\mathcal{M}_{gr} * (\#\mathcal{M}_{gr} - 1) * 2$) and used bandwidth ($64 * \#\mathcal{M}_{gr} + 64 * \#\mathcal{M}_{gr}^2$) grow exponential. Therefore the chosen protocol does not scale properly for large groups.

Privacy

Votes are not private in this design. Every member knows who the other members voted for. Beside privacy implications, this also can have effects on the voting behavior of the members, who may rather not vote instead of sharing an "unpopular" vote.

Alternative Approach

To simplify the voting process and decrease message bandwidth, the approach from DPoS systems such as BitShares can be adopted, where the votes can be shared at any time and are attached to the transactions. However the protocol would need to be changed, such that votes are also taken into account, even if the corresponding suggestion is not approved and not added to the chain. It could be possible to introduce a new message type for votes, such that they don't need to be attached to a suggestion and are required to be always included into the chain. If each time a member updates his vote, the delegates can be calculated newly, such that changes in the voting result are applied instantly.

However, this system is also not able to provide privacy for the voters.

6.2.3 Forks

With the currently used forking mechanism it is possible that information are lost form the chain. While this is no issue, if the blocks in the branches contains the same suggestion. For different suggestion the information from the dropped branch is no longer valid. If a user was added to the group, the result is, that his membership is no longer valid.

This issue can be used by a malicious delegate to drop blocks he don't like. If he creates a fork, e. G. with an *ConfirmationBlock* and directly expands his branch with another *SuggestionBlock* the original valid block is lost.

Furthermore the current protocol and implementation does not define, how the clients should behave if both branches get extended. This can happen, when two clients confirm, not only one, but multiple suggestions in a row at the same time. However, in the simulations, this case never occurred, since clients check there messages open messages for *NewBlock* messages before signing a new block.

One way to prevent this issues, is to prevent forks in the first place. By introducing *server-receive-timestamp* for all messages, every client can check which block was shared first and only use this block and no fork occur. In order to work, the timestamp would also need to be send back to the sender, such that he can also do the comparison.

6.3 Simulation

The designed system was implemented and tested using a simulation.

6.3.1 Implementation Goals

The implementation is able to simulate a group within the defined properties and assumptions. Using the designed protocol, clients are able to maintain groups with all 50 simulated members. Clients simulate asynchronicity and hold elections. Delegate are able to sign suggestions that add and remove users and update the group name. Thous, the goal to test the protocol in a simplified simulation is reached and the protocol is shown to work within the tested scope.

Asynchronicity

The clients are able to send and receive message asynchronously. Each client sleeps for a random amount of seconds and processes all open messages when he wakes up and is *online*.

Elections

The simulation is able to hold election and append the result as *VotingBlock* to the chain. Since no client acts malicious, all clients compute the same result. The implementation is able to compute the results, even if the client was not online during the voting, and still add the correct block at the correct position.

Forks

In the current simulation forks occur nearly never. Due to the low message delivery time, since all applications run on the same machine, a fork only occur if two clients build a block at exactly the same time. Before building a block, each clients checks his *MessageDB* for new messages and cancels the current building if a new block is available. Since the others block can be seen directly, no second block is build and the chain is able to solve the fork with the next block. Branches with a depth of 2 or more blocks are prevented in the simulation. In a real world implementation, this check would not be sufficient to prevent such forks. Without this check the simulation would fail if such branches occur.

6.3.2 Results

Attached are the test results for three runs with different group sizes. Table 6.1 shows the number of created blocks and the average blocktime which is between 2.652 and 3.8 seconds. Due to the increasing number of delegates, larger groups have shorter block times, since the probability that a delegate get online sooner is higher. During all test runs only two forks occur and four elections failed. These elections failed because not enough members shared a vote or confirmed the result.

However, these result allow no conclusions about the performance of the protocol, since the result strongly depends on the sleep and confirmation rate parameter of the simulation.

Table 6.1: Results for 8, 24 and 50 possible group members.

Max Members	**8**	**24**	**50**
Initial Group Size	3	6	8
Blocks	102	106	210
avg. Blocktime	3.800 s	2.663 s	2.652 s
Confirmed Sug.	84	91	197
Total Sug.	139	120	268
Elections	9	9	11
Failed Elections	2	1	2
Forks	0	1	1

6.3.3 Challenges

The election implementation depends on the assumptions that ever message is received in the correct order and no messages are lost since the implementation needs to follow a strict order in which states are checked and votes are processed. Without the fixed order of these steps the clients are not able to compute the results. The first implementation approach used timers and events to set flags which indicated the current window. This concept increased the implementations complexity to deal with message that where received outside of the window they where send in. Dropping this event based implementation added the need to check each groups voting state as soon as a client gets online.

Chapter 7

Conclusion

In this work the author presented a protocol for decentral group management in mobile messaging applications. The designed protocol is able to maintain a stable group inside a simulation and is able to protect the group from several adversaries such as a malicious server, users or members.

The analysis has shown, that the protocol has some open issues, that would prevent the use in a real world application at the current state, but also possible solutions where described that can be worked on in further work. The election can be changed, such that votes can be shared asynchronously at any time and the forks can be resloved with *server-side-timestamps*. By tweaking the simulation, the protocol can be tested with connection issues such as lost messages.

However, it is also worth taking a look at simplifying the solution, by removing some complex parts like the blockchain or the elections. The groups history do not need to be stored to compare the latest state and suggestions could also be confirmed by administrators. A distributed protocol with suggestions, *traditional* administrators and a properly confirmed state change, still can provide protection from malicious servers and members.

A final comparison between some possible options can be done by building prototypes based on existing open source messengers like Signal.

Bibliography

[1] Joël Alwen, Sandro Coretti, and Yevgeniy Dodis. The Double Ratchet: Security Notions, Proofs, and Modularization for the Signal Protocol. Cryptology ePrint Archive, Report 2018/1037, 2018. https://eprint.iacr.org/2018/1037.

[2] BitShares. BitShares Whitepaper. https://github.com/bitshares-foundation/bitshares.foundation/blob/master/download/articles/BitSharesBlockchain.pdf. Accessed Juni 30, 2019.

[3] BitShares. Delegated Proof of Stake (DPOS). https://docs.bitshares.org/en/master/technology/dpos.html#voting-algorithm. Accessed Juni 30, 2019.

[4] Trond Arne Bråthen. libsodium-core. https://github.com/tabrath/libsodium-core/. Accessed July 3, 2019.

[5] Frank Denis. libsodium. https://download.libsodium.org/doc/. Accessed July 3, 2019.

[6] EOS. TechnicalWhitePaper. https://github.com/EOSIO/Documentation/blob/master/TechnicalWhitePaper.md. Accessed Juni 30, 2019.

[7] Sharon Bradford Franklin and Andi Wilson Thompson. Open Letter to GCHQ on the Threats Posed by the Ghost Proposal. https://www.lawfareblog.com/open-letter-gchq-threats-posed-ghost-proposal. Accessed Juni 22, 2019.

[8] janvlug. Large groups tend to become unusable over time. https://community.signalusers.org/t/large-groups-tend-to-become-unusable-over-time/952. Accessed Juni 24, 2019.

[9] Jan Koun and Brian Action. end-to-end encryption. https://blog.whatsapp.com/10000618/end-to-end-encryption. Accessed Juni 26, 2019.

[10] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, pages 382–401, July 1982.

[11] Dan Larimer. BitShares History: Delegated Proof-of-Stake (DPOS). https://steemit.com/bitshares/@testz/bitshares-history-delegated-proof-of-stake-dpos. Accessed Juni 30, 2019.

[12] Dan Larimer. DPOS Consensus Algorithm - The Missing White Paper. https://steemit.com/dpos/@dantheman/dpos-consensus-algorithm-this-missing-white-paper. Accessed Juni 30, 2019.

[13] Ian Levy and Crispin Robinson. Principles for a More Informed Exceptional Access Debate. https://www.lawfareblog.com/principles-more-informed-exceptional-access-debate. Accessed Juni 22, 2019.

[14] Joshua Lund. Technology preview: Sealed sender for Signal. https://signal.org/blog/sealed-sender/. Accessed Juni 24, 2019.

[15] Moxie Marlinspike. Private Group Messaging. https://signal.org/blog/private-groups/. Accessed Juni 21, 2019.

[16] Moxie Marlinspike. Signal Foundation. https://signal.org/blog/signal-foundation/. Accessed Juni 24, 2019.

[17] Moxie Marlinspike and Trevor Perrin. The Double Ratchet Algorithm. https://signal.org/docs/specifications/doubleratchet/. Accessed Juni 24, 2019.

[18] Moxie Marlinspike and Trevor Perrin. The X3DH Key Agreement Protocol. https://signal.org/docs/specifications/x3dh/. Accessed Juni 24, 2019.

[19] Newtonsoft.JSON. Json.NET. https://www.newtonsoft.com/json. Accessed Juni 30, 2019.

[20] RabbitMQ. RabbitMQ. https://www.rabbitmq.com/. Accessed Juni 30, 2019.

[21] Paul Rösler, Christian Mainka, and Jörg Schwenk. More is Less: On the End-to-End Security of Group Chats in Signal, WhatsApp, and Threema. Cryptology ePrint Archive, Report 2017/713, 2017. https://eprint.iacr.org/2017/713.

[22] Signal. GroupMessageProcessor.java. https://github.com/signalapp/Signal-Android/blob/e603162ee767d56fa16f56701cd29010f22ed22d/src/org/thoughtcrime/securesms/groups/GroupMessageProcessor.java. Accessed July 2, 2019.

[23] Signal. libsignal-service-java. https://github.com/signalapp/libsignal-service-java/. Accessed Juni 24, 2019.

[24] Signal. Start and Manage Groups. https://support.signal.org/hc/en-us/articles/360007319331-Start-and-Manage-Groups. Accessed Juni 24, 2019.

[25] VirtualBox. VirtualBox. https://www.virtualbox.org/. Accessed Juni 30, 2019.

[26] WhatsApp. WhatsApp FAQ. https://faq.whatsapp.com/. Accessed Juni 21, 2019.

[27] WhatsApp. WhatsApp Security Whitepaper. https://www.whatsapp.com/security/WhatsApp-Security-Whitepaper.pdf. Accessed Juni 21, 2019.