Asynchronous and Decentral Group Management

in Messengers with Delegated Proof of Stake

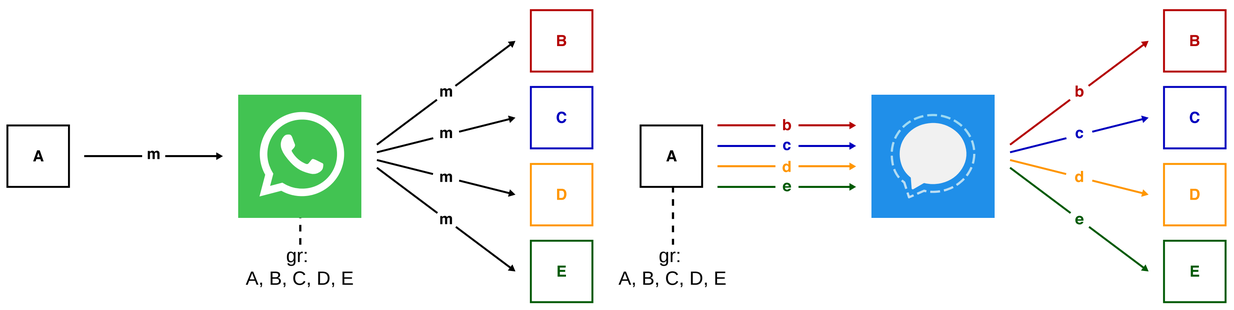
AndHell

Group Chats

Group chats in mobile messengers are mostly managed by *administrators* and rely on central servers to maintain the state:

$$\mathrm{gr} = (\mathrm{Id}_{gr}, \mathcal{M}_{gr}, \mathcal{M}^*_{gr}, \mathrm{info}_{gr})$$

Each group can be descried with the members *M*, the admins *M**, meta *info* (like the name) and an unique *Id*. To resolve the privacy implications resulting from the central management, a decentral approach can be implemented. Due to the mobile and asynchronous communication, messages can be lost or received in the wrong order. The protocol needs to be able to ensure *same state* under this conditions.



Delegated Proof of Stake

Delegated Proof of Stake *DPoS* is a consensus algorithm for **blockchains**. All participants can elect **Delegates**. These delegates can build blocks by signing transactions. In the context of group management, **Suggestions** are used as transactions.

Suggestions

Suggestions *S* can be sent by any member at any time. They are used to update the current group state. Three suggestions are defined as tuple:

- **(Add, X)**: Add a new member X to the group
- **(Remove, X)**: Remove member X from the group
- **(Update, *info*)**: Updates the group info.

Suggestions are confirmed, if a delegate builds a new block, that includes the suggestion, e.g. (Update, “Cats”). Delegates can also decide to not confirm a suggestion by not sharing a corresponding block, e.g. (Update, “Dogs”).

Blocks

Blocks *B* are built by the delegates to confirm a suggestion or the current state. Confirmation Blocks *Conf.* are used if no new suggestions are open for confirmation and are shared to confirm that members still use the correct state.

(Id, pBH, (S|Conf.), MVP, VMR, nVMR, Sender, Signature, [Vote])

Blocks are linked to each other by including the Previous Block Hash *pBH*. Delegates proof that they are currently elected by providing a Merkle Verification Path *MVP* that can be verified by the members using the Voting Merkle Root *VMR* of the Merkle Tree over the delegates. If the elected delegates change, the update is introduced with the next VMR *nVMR* field. Each block contains the votes *[Vote]*, that were shared since the last block. Finally a block needs to be singed by the delegated.

Voting

In order to elect the delegates, each member can send his votes *V* at any time. A vote can include from 1 to |M| votes, each weighted with 1/N. Votes can be verified with the signed Merkle Root *MR* of the votes.

(uID, [Votes], signed MR)

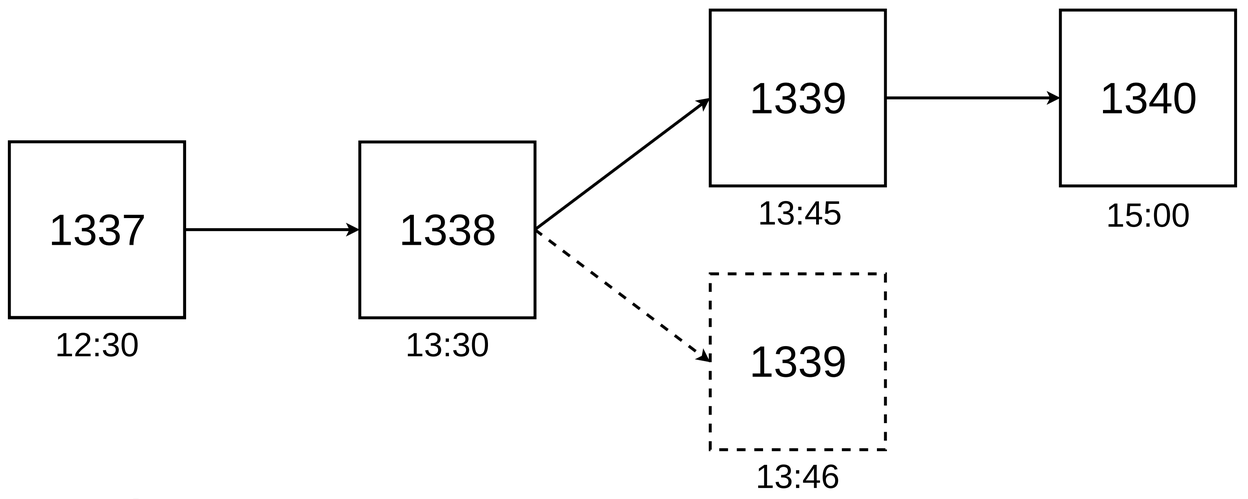
Votes are applied as soon as they are added to the blockchain by an new block. For each new block, all members compute the current elected delegates and compare the Merkle Root against the *nVMR*. To compute the result, the last vote of each member is retrieved from the blockchain and the votes are counted. The final results for each member can be displayed as:

(1: 2.4, [1,4,5])

Member 1 received 2.4 votes and voted for 1, 4, and 5.
The N members with the highest votes are chosen as delegates, where N is based on the number of members$(N = \sqrt{|\mathcal{M}_{gr}|})$.

Forks

Forks can occur if delegates share new blocks that are linked to the same previous block. In this case the blockchain opens into multiple branches. Most systems resolve forks by only using the longest chain, e.g. the branch that gets continued first. This approach is not suitable for group management. In phases with multiple branches, members might end in different states.
Forks can be prevented if messages get a *Server-Side-Timestamp*. Clients then can compare which block was shared first and only use this block.



Asynchronous Messaging

Messages can be lost or delayed in transit. If messages are received, that can't be appended to the blockchain directly (e.g. due to a missing block), these messages are stored in a cache. If the missing message does not arrive (delayed), the client requests the blockchain from the other group members to resync itself.

Simulation

Group chats are simulated in an dotnetcore application that exchanges messages via a RabbitMQ server. Asynchronicity is simulated by random sleeps of the clients. If a client is awake, the received messages are, sometimes randomly delayed or dropped and processed otherwise. Before going to sleep new suggestions, such as adding a new users are send to keep the group alive.