Groups

Problems

1. The current group design is based on message broadcast by the sender to all group members, requiring a fully connected graph between members. This results in the linear scaling of the traffic for senders, making it not viable for large groups.

2. As connection handshake is mediated by the introducer, there are multiple steps to establish direct connections between members, and while all these messages are asynchronous thanks to the relays, they require multiple steps, and can take arbitrarily long time to complete.

3. Group state is incosistent in all parts: group profile, group membership and group messages. While some efforts are made to avoid this inconsistency (pending messages while members are being connected, profile changes only allowed to group owners), there is no conflict resolution strategy, and no mechanism to validate group integrity. In addition to race conditions and slow connections between members that may result in inconsistent state between members, there is also nothing that prevents members from sending different messages to different members - integrity validation is only present withing each connection between members, but not for the whole group.

Design goals

1. Strong consistency of messages, membership and group configuration - with conflict resolution and integrity validation.
2. Availability of past messages, as defined by group policy. This implies the need for multi-hop delivery, and it also that a fully connected graph is not required.
3. Better scaling of sending of the group messages - as `O(log N)` or as `O(log log N)`, at the cost of increasing the cost to receive the messages. It will result on higher average traffic per member with much lower variance of that traffic.
4. Member roles would imply their different weight in conflict resolution and different per-member traffic - groups should be more costly for owners and admins, in terms of traffic, although there may be a separate flag for "high-traffic members".
5. Group integrity validation to ensure the consistent state for all members - message DAG (with support for irreversible deletion, but otherwise immutable), conversation state (mutable), membership roster, group profile and policies/preferences. That implies the requirement for deterministic conflict resolution strategy.
6. Efficient mechanism to catch up with missed messages (e.g., member was offline, or the group delivery was suspended - there are many

requests for such features) - via files, with explicit support for history gaps.

Design ideas / approaches

Abstractions for group state, integrity and conflict resolution

Group can be considered as a distributed state machinem, that requires some state machine replication protocol to achieve consistency, with the following components of the state:

Events history.

There are some recent attempts [1], [2] to agree on a single linear history of events, when initially due to a concurrency a state machine history forks into a tree, and then the job of participating nodes to agree on a single linear history of events. This is a general approach in most cryptocurrency blockchains where linear history of transations prevents double spends.

The consensus about linear history seems costly to achieve and in the context of group communication seems unnecessary in most cases. This is different from cryptocurrency event history, where all events represent the change of ownership in some asset, and therefore there should be a consensus about the sequence of all events. But even in cryptocurrency blockchains this could have been achieved without a single linear history if there were some additional restriction imposed on each asset record that there can be no more than one transaction per asset between some comparatively rare "finalization/commit" records that merge all forks, and no linearity required between these events. In case of cryptocurrency blockchain it would substantially increase throughput of all transactions at the cost of reducing the frequency of transactions with each given asset.

```
A (finalized state) / \ B C \ / \ D E \ / F (finalized state)
```

The diagram above is a sample of events graph (directed acyclic graph, or DAG) where each event references one or several parents by incuding their identifiers (consisting of peer identifier and sequential message number from this peer) and hashes to validate the integrity of this graph for all peers (Merkle DAG). If each event represents some irreversible action (e.g. transaction with the asset in a blockchain), then only one such event is allowed between finalized states A and F, and if more than one event happens between these states then all such events are rejected.

The advantage of group communication over blockchain is that there is no strong requirement to have an agreement on the linear history of events, except when events change some parts of group state (see below) in a contradictory way. Most events in the group communication are simple message additions to the conversation history, and while it has to be generally consistently ordered, some inconsistency can be preferable from the point of user experience.

Consider different approaches to the user experience in the presence of concurrently sent messages. Approach 1 dictates that messages can be only appended to the group history as it is seen by the user, but without gaps in DAG. That means that once the peer appends the message nothing can be appended above it, but the received messages can be inserted in linearised view of message history (this needs to be better formalized what "avoiding gaps in DAG" means, but in general, event parents should be above the child in the linearized history, even if they are received after the child, and the ordering between many parents of the same child, and between many children of the same set of parents, can be determined by some timestamp or some other attribute of these parents or children. It cannot be determined by the child itself, as there can be more than one child referencing the same set of parents, and they would define a different order).

```
A B | X | C D - both these childs would refer to parents A and B,
but they may define different order
```

Such user experience is present, for example, in WhatsApp, and its benefit is that a peer is less likely to miss messages added to the past of the message history.

Approach 2 dictates that all peers should have the same linear history of messages, even if it means inserting messages very early in the conversation history. In case of centralized system this order can be maintained quite easily by using the server timestamp. In case of decentralized system we cannot rely on timestamps at all, as clocks can be out of sync, and timestamp-based ordering (that SimpleX Chat uses now) would contradict parent-is-always-above principle (such violations are experienced in SimpleX Chat as replies to messages are shown above the messages themselves). There are multiple approaches on resolving these conflicts, including sorting messages by their hash.

The advantage of approach 2 is that events will be in same order for all peers, and in case some members see local peer replies under some messages (that may be unrelated to this reply), all other members, including the local peer. The disadvantage is that messages can be missed.

Imagine this conversation in both approaches:

``` Approach 1:

Alice view Bob view Charlie view ---------- ---------- ------------ B: let's make smth B: let's make smth B: let's make smth A: I'm in! A: I'm in! C: better let's break smth C: better let's break smth C: better let's break smth A: I'm in!

```
                    or Bob can see C, A
```

Approach 2:

Alice view Bob view Charlie view ---------- ---------- ------------ B: let's make smth B: let's make smth B: let's make smth C: better let's break smth C: better let's break smth C: better let's break smth A: I'm in! A: I'm in! A: I'm in!
```

` ` `

Alice clearly agreed to make something, but to Charlie it looks like she agreed to break something, and to Bob it can look either way, depending to the approach to ordering of two children. What is worse in this case, compared to the approach 2, is that Alice is unaware of it, but even if she were aware, she still cannot prove later to what she agreed without access to the parents of the event.

In case of approach 2 the conversation history changes in a way that it can be missed, but in the end it converges to the same linear view.

In both cases having only the linear history is insufficient for peers to prove what was the context of their messages, and this context defines the meaning - all peers should have visibility of the actual graph.

In the presense of such graph visibility, the Approach 1 seems better, as it reduces changes in the linear conversation history, provides a better user experience, and reduces the likelihood of missing the message.

Irrespective of how the DAG of events is presented and interpreted, DAG itself can be synchronised between members, as in the absense of malicious peers it has no conflicts by design - members can concurrently extend this graph at any time, referencing all peers visible to them at a time.

Some peers may fail to include all parents visible to them, and that would reduce graph linearity, but it won't by itself create conflicts. This can be addressed by a policy that messages failing to reference parents older than some threshold are rejected, so that group cannot be in a partitioned state indefinitely.

This graph acts as a proof that messages were delivered.

Group profile.

Group profile includes all any visible or invisible data about group available to all members, and also all conversation preferences and policies. Currently we only allow modifying the group profile to group owners, to reduce the possibility of the conflicts, without any conflict resolution strategy.

To achieve consistency of group profile changes the following can be considered:

1. Profile changes can be split to multiple event types, to reduce the possibility of conflicts. While it doesn't solve the problem of conflict resolution, it makes them less likely, and improves user experience.

2. Even without splitting profile changes to multiple vent types, the scope of the change can be determined as the diff between the previous and the new state.

3. Similar idea to having scoped or unscoped "commit" events can be applied to profile changes, so that if two changes happen between

commit events, only one of them will be accepted. This requires sending such "commit" events periodically, even in the absense of changes.

Group membership

This is the list of all members and their permissions. Events adding members do not create conflicts, even if they happen in different branches of the trea. Events removing members and changing members roles can create conflicts, for example in cases when two admin members attempt to remove each other concurrently.

To resolve conflict, there can be three approaches (that also applies to group profile changes):

- agree on event ordering. That will make one event succeed, and another rejected. This ordering has to be agreed consistently by all members, so either some consensus protocol is needed, which is complex and slow, or some arbitrary prioritization, e.g. ordering by peer ID, but this seems arbitrary, unfair or revealing additional metadata (e.g. if seniority of peers is determined by the time of joining the group).
- let both conflicting events succeed, when possible. This can be possible when members try to remove each other concurrently, or to downgrade the role. This approach seems bad, as it can leave the group in the unmanageable state (all owners removed), and it is not possible in general case when it is applied to the same member or to group profile.
- let both events fail. This seems the most reasonable approach, but it means that the action either should be delayed until "commit" event (e.g., when the action is destructive, irreversible, or can prevent some other actions before commit - e.g. removing member or role downgrade), or it may need to be reversed, and, possibly, subsequent actions need to be replaced.

Group operation

So, while most group events can be happening at any time (message additions, updates and deletions), some events require linearization and "proposed"/"accepted"/"committed" states.

Some sort of consensus protocol is still needed for all membership changes other than member addition, when the members who have appropriate permissions can vote for the change, requiring defined share of the members who can perform this action to accept this change, before it is committed.

Scalable message dissemination

The previous part presented approaches to the consistency of all parts of the group state. It relies on the efficient approach to distibuting each group event to all members of the group.

The current approach when the event creator simply sends it to all members is not viable:

- too much traffic for the sender.
- too large delay in message delivery in the period before the new member connected to all members.
- creating a fully connected graph is expensive and not scalable.

An open question is whether each member in the group should even have the full list of group members and their profile data - for some group sizes it may also be expensive. Even with the client-centric approach to group design it may be enough to have only "high capacity peers" storing the full member list, and the other members only having the knowledge of the active members - who contributed at least one message in the observable history.

[3] describes an efficient randomized algorithm for disseminating updates between distributed database sites that while doesn't guarantee the full propagation of the update, achieves a very high degree of the probability that all members receives the updates. The algorithm considers mechanics of distribution similar to epidemics, and apply the same approach to estimate the probability. When there is new update, the peer that created this update sends it to other randomly chosen peers with some fixed interval between sending these updates. As soon as a peer receives the update, they start start sending it other randomly chosen peers. In one of the variants of this algorithm, the peers that receive the update provide the feedback to the sender whether they saw this update before or not. If the recipient saw the update, then the sender stops broadcasting it with `1/k` probability. In the "blind" variant, the sender decides to stop broadcasting with `1/k` probability without the feedback. In addition to "feedback"/"blind" variants, there are also "coin" (described)/"counter" variants. In "counter" variant the sender sends the update exactly `k` times. The paper shows that "feedback" has little impact on the dissemination completeness, and that the "counter" is more efficient than "coin". It also shows that the "residue" - the share of the peers that didn't receive the update - is `e ^ (-k)` where `k` is the number of messages each peer sends (in case of "coin" variant - on average).

The suggestion is that we can apply a similar idea to disseminating the messages (events) in the group. Given that we validate the integrity of the group using Merkle DAG, the peers will be able to detect any missed updates, and request them. Instead of deciding which members to send the updates to we can first decide which members to connect to, and then choose members to receive the updates - randomly in both cases. If we aim to the average number of members **not** receiving updates being 0.1 (that is, one member misses one out of 10 messages, until the next one is sent), then depending on the group size `N` each member should broadcast the message to `k` peers, where `k` is:

N	**k**
100	~7
1000	~9
10k	~13

N	k
100k	~14
1m	~16

Reducing the group size 10x in above table would reduce probability of one member missing any given message to 1%.

We can also consider that some members will send a higher number of messages (e.g., group owners/admins or members with high bandwidth devices), but it has to be estimated whether this will make a material impact on the number of messages sent.

Whether to use "coin" or "feedback" approach can depend on whether delivery receipts are enabled, although that would complicate the algorithm, and might not work well in case of some members have delivery receipts disabled.

Achieving consensus for group level events

A small share of group events can create contraditions in group state:

- removing members
- changing members roles
- changing group profile

Currently these events are sent to all members as soon as they are created, and members act on these events as soon as they receive them. For group state to be consistent, all members will have to wait until the consensus between group admins is achieved.

We could delay broadcasts to all members only after the consensus between admins is achieved. We also could have admins using the old "direct mail" approach (using the terminology from [3] - which is the current approach). So one of the options could be to achieve the consensus between group admin and owners before broadcasting updates about group profile and membership to all members.

References

[1] Trees and Turtles: Modular Abstractions for State Machine Replication Protocols. https://arxiv.org/pdf/2304.07850.pdf

[2] Efficient Replication via Timestamp Stability. https://arxiv.org/pdf/2104.01142.pdf

[3] Epidemic Algorithms for Replicated Database Maintenance. https://dl.acm.org/doi/pdf/10.1145/43921.43922