Byzantine fault tolerance for peer-to-peer collaboration

MARTIN KLEPPMANN

until Dec 2023
TU Munich

from Jan 2024
University of Cambridge

email martin@kleppmann.com
web https://martin.kleppmann.com
bluesky @martin.kleppmann.com
mastodon @martin@nondeterministic.computer

TUM VolkswagenStiftung Ink & Switch
Collaborative Applications

Google Docs
Office 365
Overleaf
Trello
Figma

Wide range of domain-specific collaboration software, e.g. for investigative journalism, medical records, data analysis, engineering/CAD, ...
Byzantine fault tolerance:
System continues to provide its advertised guarantees, even if some nodes are malicious (do not correctly follow protocol).
authentication
and input validation
Central Server

Authentication and input validation

Authoritative copy of the data
COLLABORATION IN ANY NETWORK TOPOLOGY

-one server

[Diagram showing a server connected to multiple users via arrows]
COLLABORATION IN ANY NETWORK TOPOLOGY

- one server
- multiple servers (maybe federated)
COLLABORATION IN ANY NETWORK TOPOLOGY

- one server
- multiple servers (maybe federated)
- peer-to-peer (including LAN, Bluetooth, etc.)
"The purchase price is £1,000,000"

"The purchase price is £100,000"
"Capital gains!"

seller

"The purchase price is £1,000,000"

estate agent

"The purchase price is £100,000"

buyer

Bargain!
Deal!  

“The purchase price is £1,000,000”

Capital gains!

“Bargain!”

“The purchase price is £100,000”
BLOCKCHAIN TO THE RESCUE?

Merkle tree for inclusion proofs
Blockchain to the Rescue?

Merkle tree for inclusion proofs

Total order

Byzantine consensus
BLOCKCHAIN TO THE RESCUE?

Total order required for cryptocurrencies (to prevent double-spending).

The wrong model for collaboration!
Total order required for cryptocurrencies (to prevent double-spending)

Consensus = pick one of several proposed values

Collaboration = keep all edits and merge them
partitioned
Byzantine Eventual Consistency and the Fundamental Limits of Peer-to-Peer Databases

Martin Kleppmann
University of Cambridge
Cambridge, UK
mk428@cst.cam.ac.uk

Heidi Howard
University of Cambridge
Cambridge, UK
hh360@cst.cam.ac.uk

ABSTRACT
Sybil attacks, in which a large number of adversary-controlled nodes join a network, are a concern for many peer-to-peer database systems, necessitating expensive countermeasures such as proof-of-work. However, there is a category of database applications that are, by design, immune to Sybil attacks because they can tolerate arbitrary numbers of Byzantine-faulty nodes. In this paper, we characterize this category of applications using a consistency model we call Byzantine Eventual Consistency (BEC). We introduce an algorithm that guarantees BEC based on Byzantine causal broadcast, prove its correctness, and demonstrate near-optimal performance in a prototype implementation.

1 INTRODUCTION
Peer-to-peer systems are of interest to many communities for a number of reasons: their lack of central control by a single party can make them more resilient, and less susceptible to censorship. However, they are also vulnerable to Sybil attacks, in which a large number of adversary-controlled nodes join a network. To tolerate Sybil attacks, many systems rely on proof-of-work. However, there is a category of applications that are immune to Sybil attacks because they can tolerate arbitrary numbers of Byzantine-faulty nodes. In this paper, we characterize this category of applications using a consistency model we call Byzantine Eventual Consistency (BEC). We introduce an algorithm that guarantees BEC based on Byzantine causal broadcast, prove its correctness, and demonstrate near-optimal performance in a prototype implementation.

The reason why permissioned blockchains must control membership is that they rely on Byzantine agreement, which assumes that at most $f$ nodes are Byzantine-faulty. To tolerate $f$ faults, Byzantine agreement algorithms typically require at least $3f + 1$ nodes [17]. If more than $f$ nodes are faulty, these algorithms can guarantee neither safety (agreement) nor liveness (progress). Thus, a Sybil attack that causes the bound of $f$ faulty nodes to be exceeded can result in the system’s guarantees being violated; for example, in a cryptocurrency, they could allow the same coin to be spent multiple times (a double-spending attack).

This state of affairs raises the question: if Byzantine agreement cannot be achieved in the face of arbitrary numbers of Byzantine-faulty nodes, what properties can be guaranteed in this case?

A system that tolerates arbitrary numbers of Byzantine-faulty nodes is immune to Sybil attacks: even if the malicious peers outnumber the honest ones, it is still able to function correctly. This makes such systems of large practical importance: being immune to Sybil attacks means neither proof-of-work nor the central control of permissioned blockchains is required.
Byzantine Eventual Consistency (BEC)

**Eventual update:**
One correct replica applies update \( u \)
\[ \Rightarrow \text{all correct replicas eventually apply } u \]
Byzantine Eventual Consistency (BEC)

**Eventual update:**
- One correct replica applies update $u$
- $\Rightarrow$ all correct replicas eventually apply $u$

**Convergence:**
- Two replicas have applied same set of updates
- $\Rightarrow$ they are in the same state
Byzantine Eventual Consistency (BEC)

Eventual update:
One correct replica applies update \( u \)
\( \Rightarrow \) all correct replicas eventually apply \( u \)

Convergence:
Two replicas have applied same set of updates
\( \Rightarrow \) they are in the same state

Invariant preservation:
The state of a correct replica always satisfies all of the app's declared invariants
(and a few other, more technical properties)
Version vectors are not safe

Byzantine

Correct

Correct

Correct
VERSION VECTORS ARE NOT SAFE

"equivocation"

C (Byzantine) updates u₁ with ID (C, 1) and u₂ with ID (C, 2). The correct outcome for A and B is indicated.
VERSION VECTORS ARE NOT SAFE

A
\{ (c, 1) \rightarrow u_1 \}$

C
Byzantine

B
\{ (c, 1) \rightarrow u_2 \}$
VERSION VECTORS ARE NOT SAFE

C
Byzantine

A
Hello, I have \{C \to 1\}

B
I also have \{C \to 1\}, we must be in sync

\{(c, 1) \to u_1\}

\{(c, 1) \to u_2\}
VERSION VECTORS ARE NOT SAFE

A never delivers $u_2$
B never delivers $u_1$
$\Rightarrow$ failure of eventual delivery

Byzantine

\[
\begin{align*}
\text{correct} & \quad \exists (c, 1) \rightarrow u_1 \?
A & \\
\text{correct} & \quad \exists (c, 1) \rightarrow u_2 \?
B & \\
\end{align*}
\]

Hello, I have $\{C \rightarrow 1\}$, we must be in sync

I also have $\{C \rightarrow 1\}$,
ENSURING EVENTUAL DELIVERY

Nodes connect pairwise, send each other updates that the other doesn't have

\[ \{u_1, u_2\} \]

A

\[ \{u_1, u_3\} \]

B
ENSURING EVENTUAL DELIVERY

Nodes connect pairwise, send each other updates that the other doesn't have
ENSURING EVENTUAL DELIVERY

How do nodes figure out what to send to each other?

Hash graph (like Git!):

A

\[ u_1 \quad \rightarrow \quad H(u_1) \quad \rightarrow \quad u_2 \]

B

\[ u_1 \quad \rightarrow \quad H(u_1) \quad \rightarrow \quad u_3 \]
ENSURING EVENTUAL DELIVERY

How do nodes figure out what to send to each other?

Hash graph (like Git!):

A

\[ u_1 \quad \text{null} \quad \text{data} \quad \text{H}(u_1) \quad \text{data} \]

B

\[ u_1 \quad \text{null} \quad \text{data} \quad \text{H}(u_1) \quad \text{data} \]

\[ \text{H}(u_2) \quad \text{x} \quad \text{H}(u_3) \]
ENSURING EVENTUAL DELIVERY

How do nodes figure out what to send to each other?

Hash graph (like Git!):

Assuming a collision-resistant hash function, a Byzantine node cannot cause two correct nodes to believe they are in sync when in fact they have diverged.
BEC replication

A

B

remember result of last sync between A and B
BEC replication

-added by A since last sync

remember result of last sync between A and B

-added by B since last sync
BEC replication

added by A since last sync

BloomFilter (hashes)

remember result of last sync between A and B

added by B since last sync

BloomFilter (hashes)
This sync protocol is implemented in Automerge

github.com/automerge/automerge

Thanks to Peter van Hardenberg & other contributors!

Blog post

martin.kleppmann.com/2020/12/02/bloom-filter-hash-graph-sync.html

Details in the paper!  arxiv.org/abs/2012.00472
Byzantine Eventual Consistency (BEC)

**Eventual update:**
One correct replica applies update $u$  
$\Rightarrow$ all correct replicas eventually apply $u$

**Convergence:**
- use CRDTs
  Two replicas have applied same set of updates  
  $\Rightarrow$ they are in the same state

**Invariant preservation:**
The state of a correct replica always satisfies all of the app's declared invariants  
(and a few other, more technical properties)
Who are the current members of a group chat/collaborators on a document?

Simple answer: all users who have been added and not removed again
Real answer: not as straightforward as you may think...

Work-in-progress with Annette Bieniuse and Herb Caudill
Recovering from key compromise in decentralised access control systems

Martin Kleppmann
University of Cambridge
Cambridge, UK
martin@kleppmann.com

Annette Bieniusa
TU Kaiserslautern
Kaiserslautern, Germany
bieniusa@cs.uni-kl.de

Abstract—In systems with multiple administrators, such as group chat applications, it can happen that two users concurrently revoke each other’s permissions. For example, this could occur because an administrator’s device was compromised, and an adversary is actively using stolen credentials from this device while another administrator is trying to revoke the compromised device’s access. In decentralised systems, the order of these mutual revocations may be unclear, leading to disagreement about who the current group members are. We present an algorithm for managing groups where members can add or remove other members. In the event of a compromise, our algorithm allows the legitimate users to reliably revoke all compromised devices and lock out the adversary, regardless of how the adversary uses secret keys from the compromised devices. Our algorithm requires no trusted authority and no central control, and can therefore be used in decentralised settings such as mesh or mix networks.

Index Terms—access control, authorisation, group messaging, group membership, decentralisation, key compromise, CRDT

In some systems, it is possible to use a centralised arbiter, such as a trusted server, to resolve such conflicting permission changes. However, the problem becomes harder in decentralised systems that have no such central point of control: for example, mesh networks have been used by protesters to communicate without using the Internet [2],
PERMISSIONS / ACCESS CONTROL

WARNING: WORK IN PROGRESS
PERMISSIONS / ACCESS CONTROL

WARNING: WORK IN PROGRESS

Who may read some data?

- Centralised: authenticate to server with password
PERMISSIONS / ACCESS CONTROL

WARNING: WORK IN PROGRESS

- Who may read some data?
  - Centralised: authenticate to server with password
  - Basic P2P: you can have the data if you know URL
Who may read some data?

- Centralised: authenticate to server with password
- Basic P2P: you can have the data if you know URL
- Better: end-to-end encryption + decentralised access control lists
PERMISSIONS / ACCESS CONTROL

WARNING: WORK IN PROGRESS

- Who may read some data?
  - Centralised: authenticate to server with password
  - Basic P2P: you can have the data if you know URL
  - Better: end-to-end encryption + decentralised access control lists

- Who may write some data?
  - Centralised: server rejects unauthorised changes
PERMISSIONS / ACCESS CONTROL

WARNING: WORK IN PROGRESS

- Who may read some data?
  - Centralised: authenticate to server with password
  - Basic P2P: you can have the data if you know URL
  - Better: end-to-end encryption + decentralised access control lists

- Who may write some data?
  - Centralised: server rejects unauthorised changes
  - Decentralised: every peer maintains ACL, ignores changes from peers who don't have permission
Want a "decentralised access control list" protocol:

- Group creator is an admin
- Any admin can add/remove other admins

[Distinction between admins and non-admin group members elided for now]
Want a "decentralised access control list" protocol:

- Group creator is an admin
- Any admin can add/remove other admins

[Distinction between admins and non-admin group members elided for now]

Requirements

- No server, no trusted authority
- No infrastructure besides P2P networking (no blockchain)
- Must tolerate users being offline
- Non-admins cannot affect group state
- Everyone agrees who the admins are (eventually, after messages delivered)
Approach

- Every user/device A has a keypair \((pk_A, sk_A)\)
- Operations: create group, add member, remove member
- Each operation is signed by its creator
- Signed operations are broadcast (e.g. by gossip protocol) to all group members
- currentMembers = \(f(\text{operationsReceived})\) at each device
Users/devices: Alice, Bob, Carol, Dave, ...

Public keys: \( pk_A, pk_B, pk_C, pk_D, \ldots \)

Private keys: \( sk_A, sk_B, sk_C, sk_D, \ldots \)

\[ \text{op}_1 = (\text{create, } pk_A) \]

\[ \text{Sign(op}_1, sk_A) \]

\[ \{ \text{Set of members: } pk_A \} \]
Users/devices: Alice, Bob, Carol, Dave, ...
Public keys: \( pk_A, pk_B, pk_C, pk_D, ... \)
Private keys: \( sk_A, sk_B, sk_C, sk_D, ... \)

\[ \text{op}_1 = (\text{create}, pk_A) \]
\[ \text{Sign} (\text{op}_1, sk_A) \]

\[ \text{Set of members: } \{ pk_A, pk_B \} \]

\[ \text{op}_2 = (\text{add}, pk_A, pk_B, H(\text{op}_1)) \]
\[ \text{Sign} (\text{op}_2, sk_A) \]
Users/devices: Alice, Bob, Carol, Dave, ...
Public keys: \( pk_A, pk_B, pk_C, pk_D, \ldots \)
Private keys: \( sk_A, sk_B, sk_C, sk_D, \ldots \)

\[ \text{op}_1 = (\text{create}, pk_A) \]
\[ \text{Sign}(\text{op}_1, sk_A) \]

\[ \text{op}_2 = (\text{add}, pk_A, pk_B, H(\text{op}_1)) \]
\[ \text{Sign}(\text{op}_2, sk_A) \]
\[ H(\text{op}_2) \]

\[ \text{op}_3 = (\text{add}, pk_A, pk_C, H(\text{op}_2)) \]
\[ \text{Sign}(\text{op}_3, sk_A) \]

\[ \text{op}_4 = (\text{add}, pk_B, pk_D, H(\text{op}_2)) \]
\[ \text{Sign}(\text{op}_4, sk_B) \]
Users/devices: Alice, Bob, Carol, Dave, ...
Public keys: \(pk_A, pk_B, pk_C, pk_D, \ldots\)
Private keys: \(sk_A, sk_B, sk_C, sk_D, \ldots\)

\[\text{op}_1 = (\text{create}, pk_A)\]
\[\text{Sign}(\text{op}_1, sk_A)\]

\[h(\text{op}_1)\]

\[\text{op}_2 = (\text{add}, pk_A, pk_B, H(\text{op}_1))\]
\[\text{Sign}(\text{op}_2, sk_A)\]
\[h(\text{op}_2)\]

\[\text{Set of members: } \{pk_A, pk_B, pk_C, pk_D\}\]

\[\text{op}_3 = (\text{add}, pk_A, pk_C, H(\text{op}_2))\]
\[\text{Sign}(\text{op}_3, sk_A)\]

\[\text{op}_4 = (\text{add}, pk_B, pk_D, H(\text{op}_2))\]
\[\text{Sign}(\text{op}_4, sk_B)\]
Users/devices: Alice, Bob, Carol, Dave, ...

Public keys: \( pk_A, pk_B, pk_C, pk_D, \ldots \)

Private keys: \( sk_A, sk_B, sk_C, sk_D, \ldots \)

\[ \begin{align*}
    op_1 &= (create, pk_A) \\
    \text{Sign}(op_1, sk_A)
\end{align*} \]

\[ \begin{align*}
    op_2 &= (add, pk_A, pk_B, H(op_1)) \\
    \text{Sign}(op_2, sk_A)
\end{align*} \]

\[ \begin{align*}
    op_3 &= (add, pk_A, pk_C, H(op_2)) \\
    \text{Sign}(op_3, sk_A)
\end{align*} \]

\[ \begin{align*}
    op_4 &= (add, pk_B, pk_D, H(op_2)) \\
    \text{Sign}(op_4, sk_B)
\end{align*} \]

\[ \begin{align*}
    op_5 &= (remove, pk_A, pk_B, \{H(op_3), H(op_4)\}) \\
    \text{Sign}(op_5, sk_A)
\end{align*} \]
Users/devices: Alice, Bob, Carol, Dave, ...
Public keys: \( pk_A, pk_B, pk_C, pk_D, \ldots \)
Private keys: \( sk_A, sk_B, sk_C, sk_D, \ldots \)

\[
\begin{align*}
op_1 &= (create, pk_A) \\
\text{Sign}(\text{op}_1, sk_A)
\end{align*}
\]

\[
\begin{align*}
op_2 &= (add, pk_A, pk_B, H(\text{op}_1)) \\
\text{Sign}(\text{op}_2, sk_A) \\
H(\text{op}_2)
\end{align*}
\]

\[
\begin{align*}
op_3 &= (add, pk_A, pk_C, H(\text{op}_2)) \\
\text{Sign}(\text{op}_3, sk_A)
\end{align*}
\]

\[
\begin{align*}
op_4 &= (add, pk_B, pk_D, H(\text{op}_2)) \\
\text{Sign}(\text{op}_4, sk_B)
\end{align*}
\]

\[
\begin{align*}
op_5 &= (remove, pk_A, pk_B, \{H(\text{op}_3), H(\text{op}_4)\}) \\
\text{Sign}(\text{op}_5, sk_A)
\end{align*}
\]

Final set of members: \( \{pk_A, pk_C, pk_D\} \)

\[
\begin{align*}
\text{NOTE: } pk_D \text{ is a member because it was added by B at a time when B was still a member.}
\end{align*}
\]
Problem: what if you want to remove the permissions from someone who doesn't want to be removed? (Byzantine behaviour)

E.g. adversary stole and unlocked a team member's phone.
A removed user concurrently adds a new user...

\[ \text{op}_1 = (\text{create}, \text{pk}_A) \]
\[ \text{Sign} (\text{op}_1, \text{sk}_A) \]

\[ \text{op}_2 = (\text{add}, \text{pk}_A, \text{pk}_B, H(\text{op}_1)) \]
\[ \text{Sign} (\text{op}_2, \text{sk}_A) \]
A removed user concurrently adds a new user:

\[ \text{op}_1 = (\text{create, } pk_A) \]
\[ \text{Sign}(\text{op}_1, \text{sk}_A) \]

\[ \text{op}_2 = (\text{add, } pk_A, pk_B, H(\text{op}_1)) \]
\[ \text{Sign}(\text{op}_2, \text{sk}_A) \]

\[ \text{op}_3 = (\text{remove, } pk_A, pk_B, H(\text{op}_2)) \]
\[ \text{Sign}(\text{op}_3, \text{sk}_A) \]
A removed user concurrently adds a new user...

\[ \text{op}_1 = (\text{create, pk}_A) \]
\[ \text{Sign}(\text{op}_1, \text{sk}_A) \]

\[ \text{op}_2 = (\text{add, pk}_A, \text{pk}_B, H(\text{op}_1)) \]
\[ \text{Sign}(\text{op}_2, \text{sk}_A) \]

\[ \text{op}_3 = (\text{remove, pk}_A, \text{pk}_B, H(\text{op}_2)) \]
\[ \text{Sign}(\text{op}_3, \text{sk}_A) \]

\[ \text{op}_4 = (\text{add, pk}_B, \text{pk}_C, H(\text{op}_2)) \]
\[ \text{Sign}(\text{op}_4, \text{sk}_B) \]

B's operation to add C can be back-dated to appear concurrent with A's removal of B.
A removed user concurrently adds a new user...

\[ \text{op}_1 = (\text{create}, \text{pk}_A) \]
\[ \text{Sign}(\text{op}_1, \text{sk}_A) \]

\[ \text{op}_2 = (\text{add}, \text{pk}_A, \text{pk}_B, \text{H}(\text{op}_1)) \]
\[ \text{Sign}(\text{op}_2, \text{sk}_A) \]

\[ \text{op}_3 = (\text{remove}, \text{pk}_A, \text{pk}_B, \text{H}(\text{op}_2)) \]
\[ \text{Sign}(\text{op}_3, \text{sk}_A) \]

\[ \text{op}_4 = (\text{add}, \text{pk}_B, \text{pk}_C, \text{H}(\text{op}_2)) \]
\[ \text{Sign}(\text{op}_4, \text{sk}_B) \]

\[ \begin{align*}
\text{Set of members:} \\
\{ \text{pk}_A \}
\end{align*} \]

B's operation to add C can be back-dated to appear concurrent with A's removal of B.

\[ \Rightarrow \text{ignore all ops by B concurrent with removal of } B \]
Two users concurrently remove each other...

\[ \text{op}_1 = (\text{create}, \text{pk}_A) \]
\[ \text{Sign}(\text{op}_1, \text{sk}_A) \]

\[ \text{op}_2 = (\text{add}, \text{pk}_A, \text{pk}_B, \text{H}(\text{op}_1)) \]
\[ \text{Sign}(\text{op}_2, \text{sk}_A) \]

\[ \text{H}(\text{op}_1) \]
Two users concurrently remove each other...

\[
\begin{align*}
\text{op}_1 &= (\text{create}, \text{pk}_A) \\
\text{Sign} (\text{op}_1, \text{sk}_A) \\
\text{op}_2 &= (\text{add}, \text{pk}_A, \text{pk}_B, H(\text{op}_1)) \\
\text{Sign} (\text{op}_2, \text{sk}_A) \\
\text{op}_3 &= (\text{remove}, \text{pk}_A, \text{pk}_B, H(\text{op}_2)) \\
\text{Sign} (\text{op}_3, \text{sk}_A)
\end{align*}
\]
Two users concurrently remove each other...

\[ \text{op}_1 = (\text{create, pk}_A) \]
\[ \text{Sign} (\text{op}_1, \text{sk}_A) \]

\[ \text{op}_2 = (\text{add, pk}_A, \text{pk}_B, H(\text{op}_1)) \]
\[ \text{Sign} (\text{op}_2, \text{sk}_A) \]

\[ \text{op}_3 = (\text{remove, pk}_A, \text{pk}_B, H(\text{op}_2)) \]
\[ \text{Sign} (\text{op}_3, \text{sk}_A) \]

\[ \text{op}_4 = (\text{remove, pk}_B, \text{pk}_A, H(\text{op}_2)) \]
\[ \text{Sign} (\text{op}_4, \text{sk}_B) \]
Two users concurrently remove each other...

\[ \text{op}_1 = \text{(create, pk}_A) \]
\[ \text{Sign}(\text{op}_1, \text{sk}_A) \]
\[ \text{H}(\text{op}_1) \]

\[ \text{op}_2 = \text{(add, pk}_A, \text{pk}_B, \text{H}(\text{op}_1)) \]
\[ \text{Sign}(\text{op}_2, \text{sk}_A) \]
\[ \text{H}(\text{op}_2) \]

\[ \text{op}_3 = \text{(remove, pk}_A, \text{pk}_B, \text{H}(\text{op}_2)) \]
\[ \text{Sign}(\text{op}_3, \text{sk}_A) \]
\[ \text{H}(\text{op}_2) \]

\[ \text{op}_4 = \text{(remove, pk}_B, \text{pk}_A, \text{H}(\text{op}_2)) \]
\[ \text{Sign}(\text{op}_4, \text{sk}_B) \]

\{ \text{Now what??} \}
A removes B
B removes A
A removes B

B removes A

∃A, C, D \ ignore
A removes B

B removes A

\{A, C, D\} ignore

\{B, C, D\} ignore
How to handle mutual revocation?

Operation timestamps?
How to handle mutual revocation?

Remove both?

Operation timestamps?  Adversarially chosen timestamps
How to handle mutual revocation?

Operation timestamps?  Adversarially chosen timestamps

Remove both?  DoS: might remove all admins

Remove neither?
How to handle mutual revocation?

Operation timestamps? Adversarially chosen timestamps

Remove both? DoS: might remove all admins

Remove neither? User can cancel their removal

Trusted server as arbiter?
How to handle mutual revocation?

- Operation timestamps? Adversarially chosen timestamps
- Remove both? DoS: might remove all admins
- Remove neither? User can cancel their removal
- Trusted server as arbiter? Not decentralised

Blockchain smart contract?
How to handle mutual revocation?

Operation timestamps? Adversarially chosen timestamps

Remove both? DoS: might remove all admins

Remove neither? User can cancel their removal

Trusted server as arbiter? Not decentralised

Blockchain smart contract? How do you ensure control over smart contract is consistent with the ACL?

What do you do while waiting for blockchain decision?
How to handle mutual revocation?

Seniority ranking of users

e.g. group creator has rank 1, user added by rank-i user has rank i+1,
break ties by lexicographic order on hashes of operations that added the users

Problem: how do you remove the most senior user?
How to handle mutual revocation?

Seniority ranking of users

- e.g. group creator has rank 1, user added by rank-i user has rank i + 1,
- break ties by lexicographic order on hashes of operations that added the users

Problem: how do you remove the most senior user?

Users vote on who is right

Problems:
- who gets a vote? Sybil attack prevention needed
- how does a user know the correct answer?
- risk of social engineering attacks
- what happens while waiting for vote to complete?
How to handle mutual revocation?

Seniority ranking of users

Problem: how do you remove the most senior user?

Solution:

Most senior public key is not for a single user/device, but rather a public key for a threshold signature scheme where the group members hold secret shares

$\Rightarrow k \text{ out of } n \text{ users can override seniority ranking}$

$\Rightarrow \text{need scheme for redistributing secret shares after group membership changes}$
Byzantine Eventual Consistency (BEC)

Eventual update:
One correct replica applies update \( u \)
\[ \Rightarrow \] all correct replicas eventually apply \( u \)

Convergence:
- use CRDTs
Two replicas have applied same set of updates
\[ \Rightarrow \] they are in the same state

Invariant preservation:
The state of a correct replica always satisfies all of the app’s declared invariants
(and a few other, more technical properties)
References


• More at https://martin.kleppmann.com