Byzantine fault tolerance in eventually consistent systems

MARTIN KLEPPMANN

until Dec 2023
TU Munich

from Jan 2024
University of Cambridge

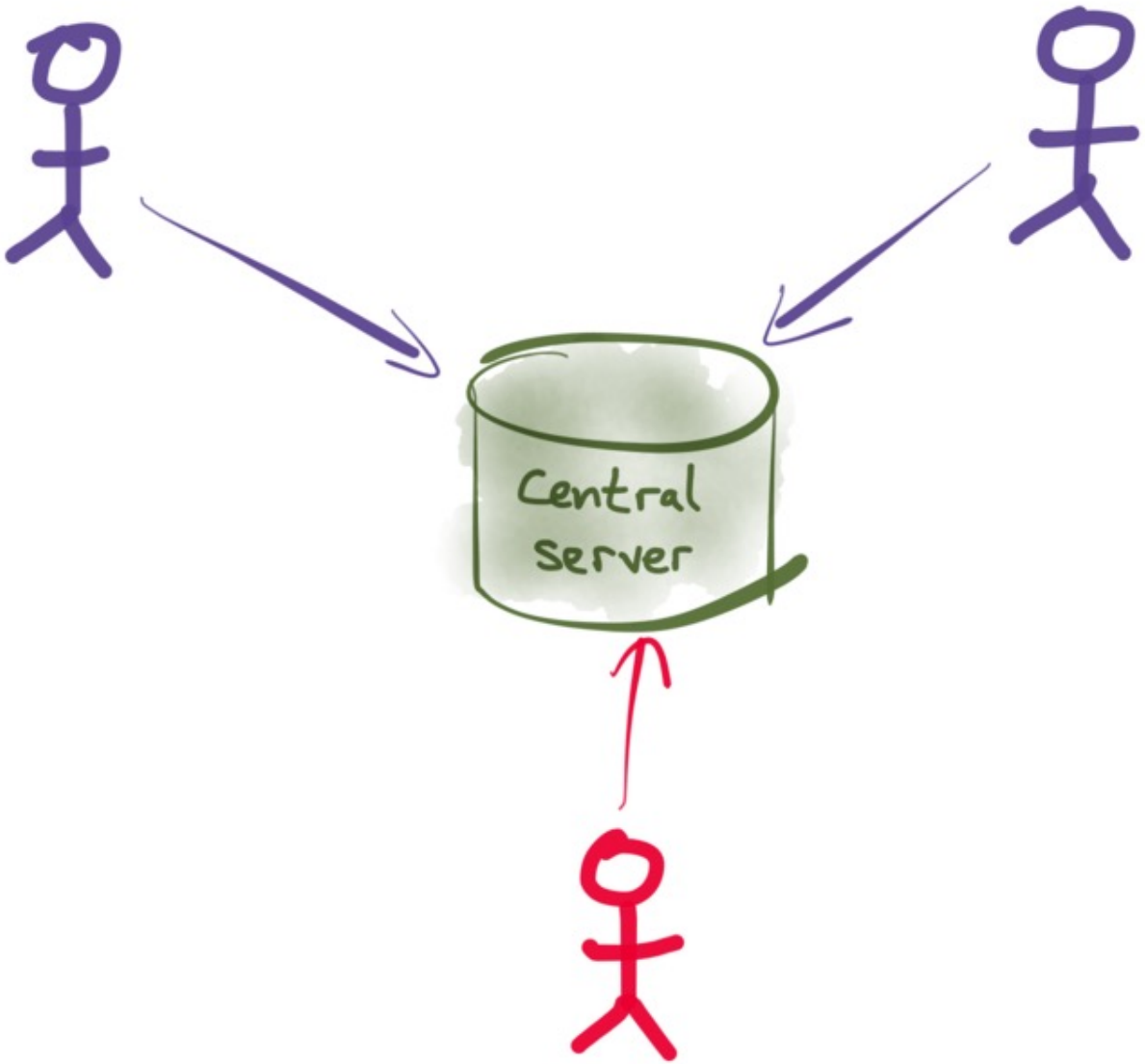
email martin@kleppmann.com
web https://martin.kleppmann.com
bluesky @martin.kleppmann.com
mastodon @martin@nondeterministic.computer

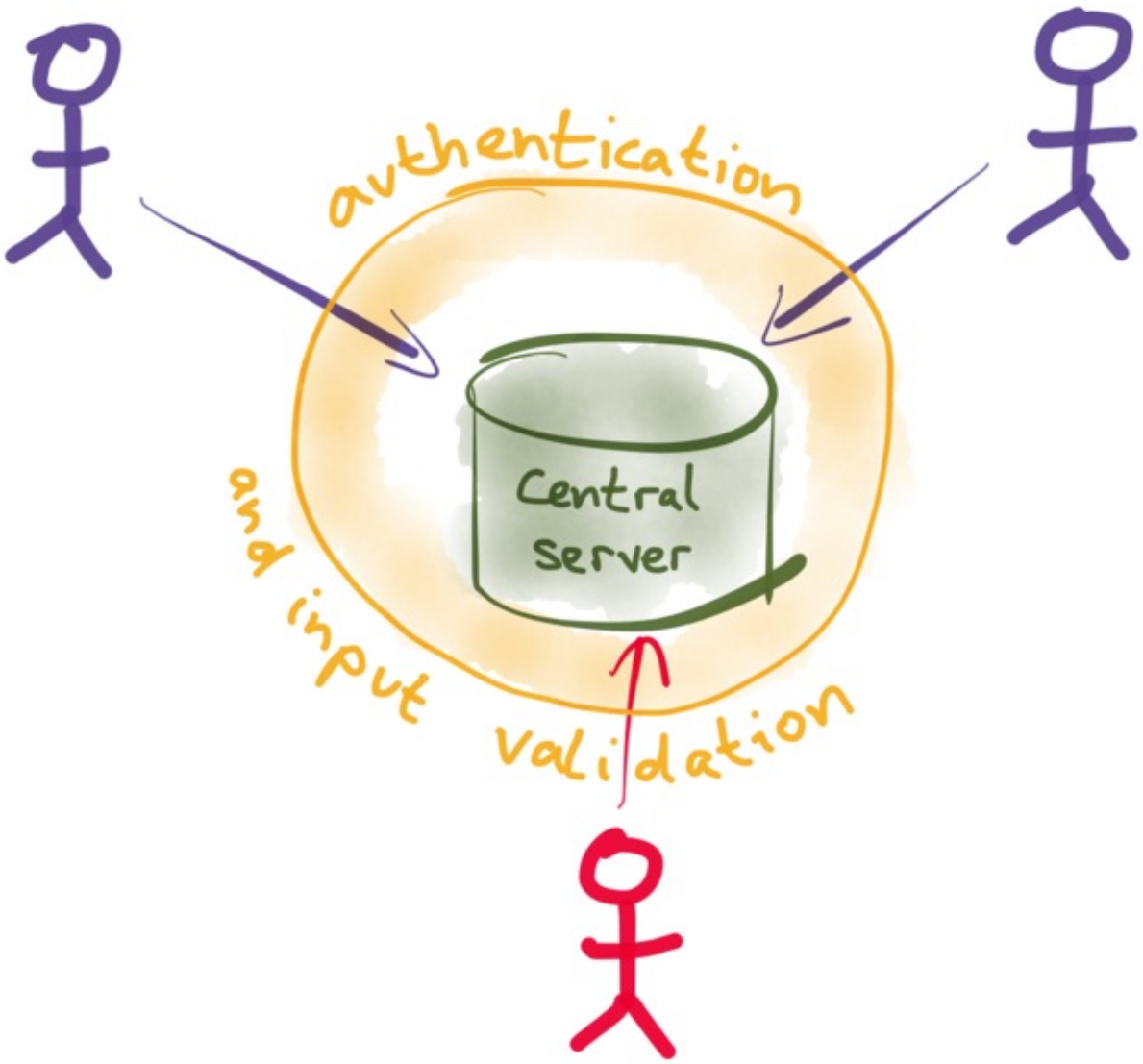
VolkswagenStiftung

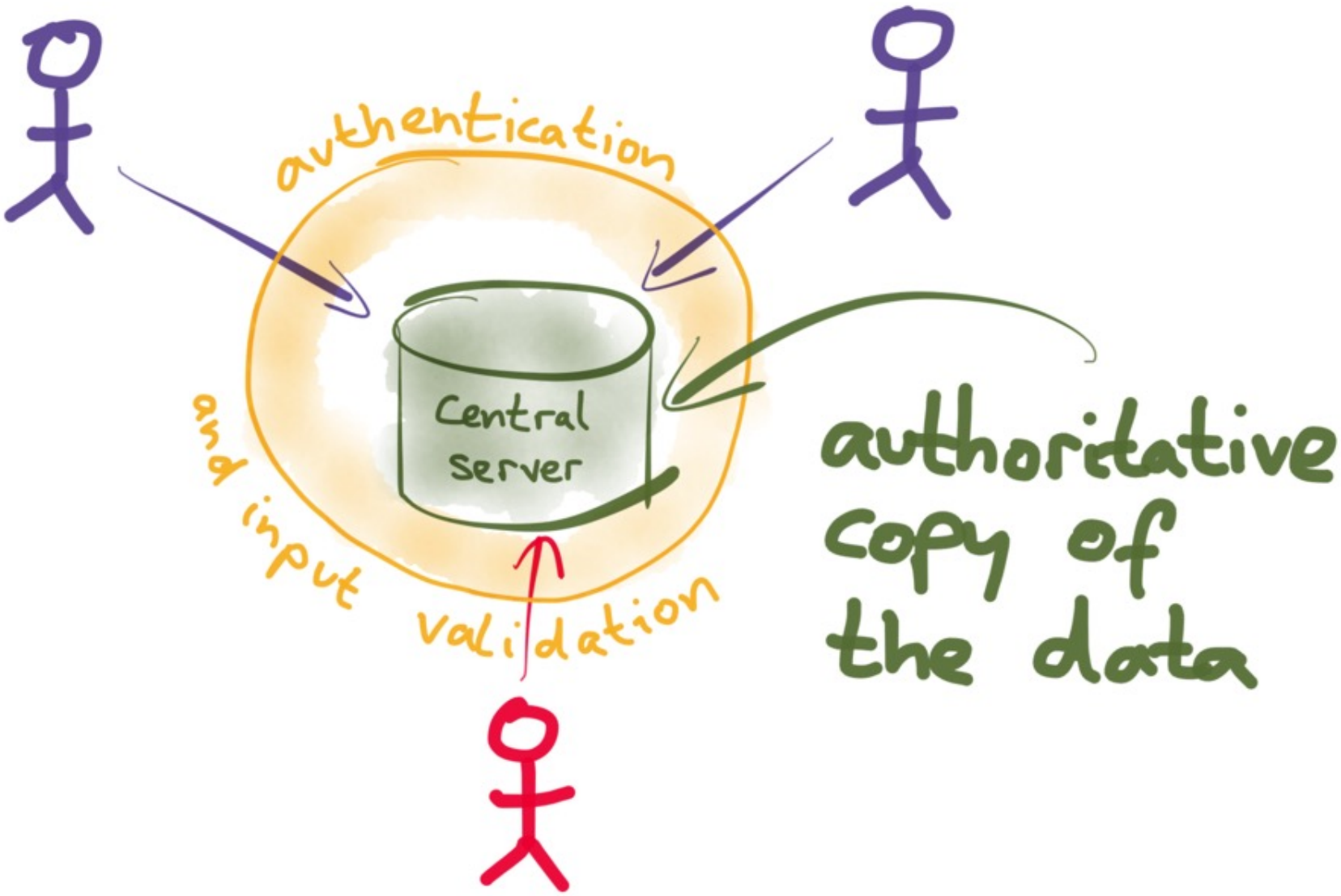
Ink & Switch

Byzantine fault tolerance:

System continues to provide its advertised guarantees, even if some nodes are **malicious** (do not correctly follow protocol).

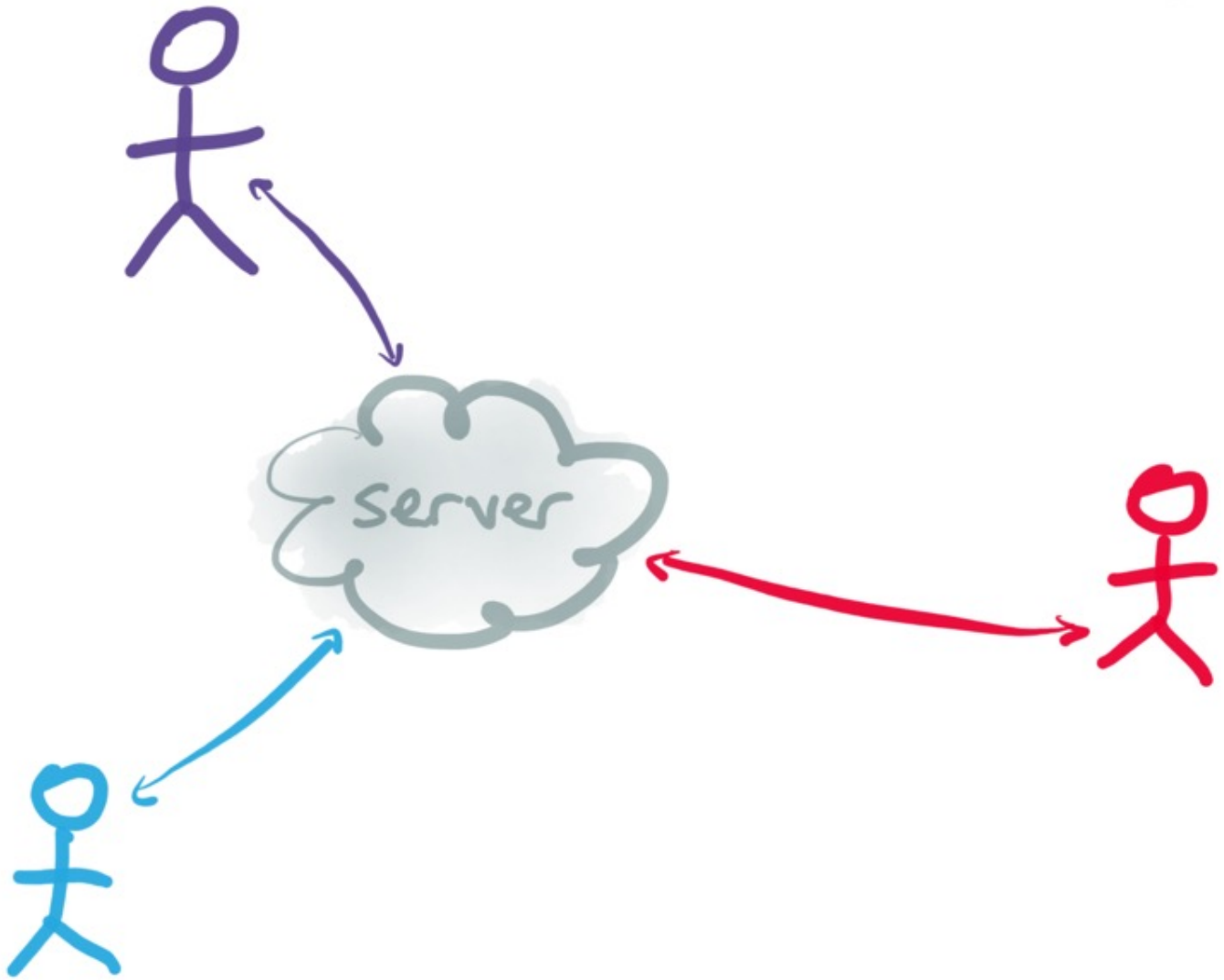

Central
Server


authentication
and input validation
Central Server


authentication
and input validation
Central Server
authoritative copy of the data

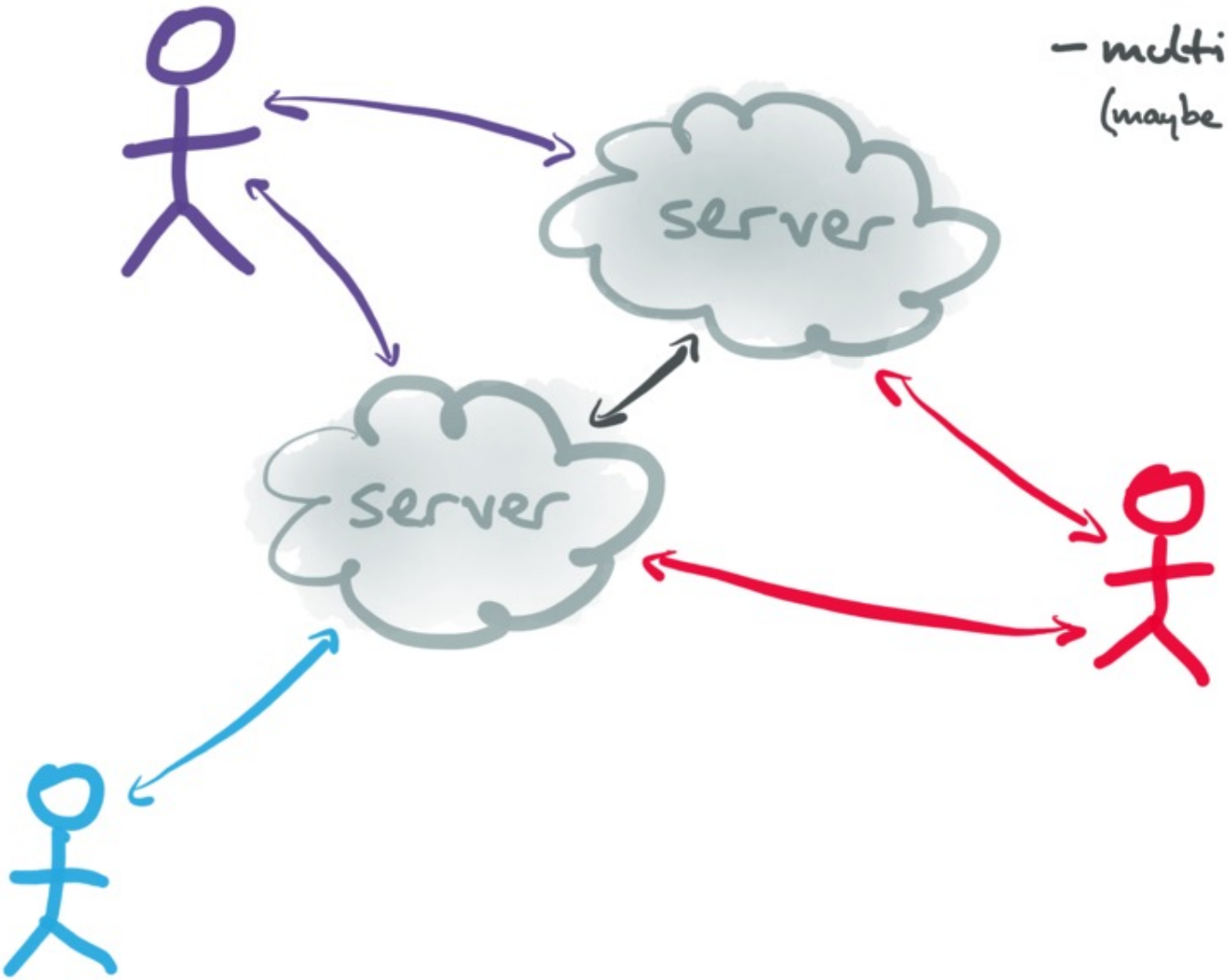
COLLABORATION IN ANY NETWORK TOPOLOGY

- one server

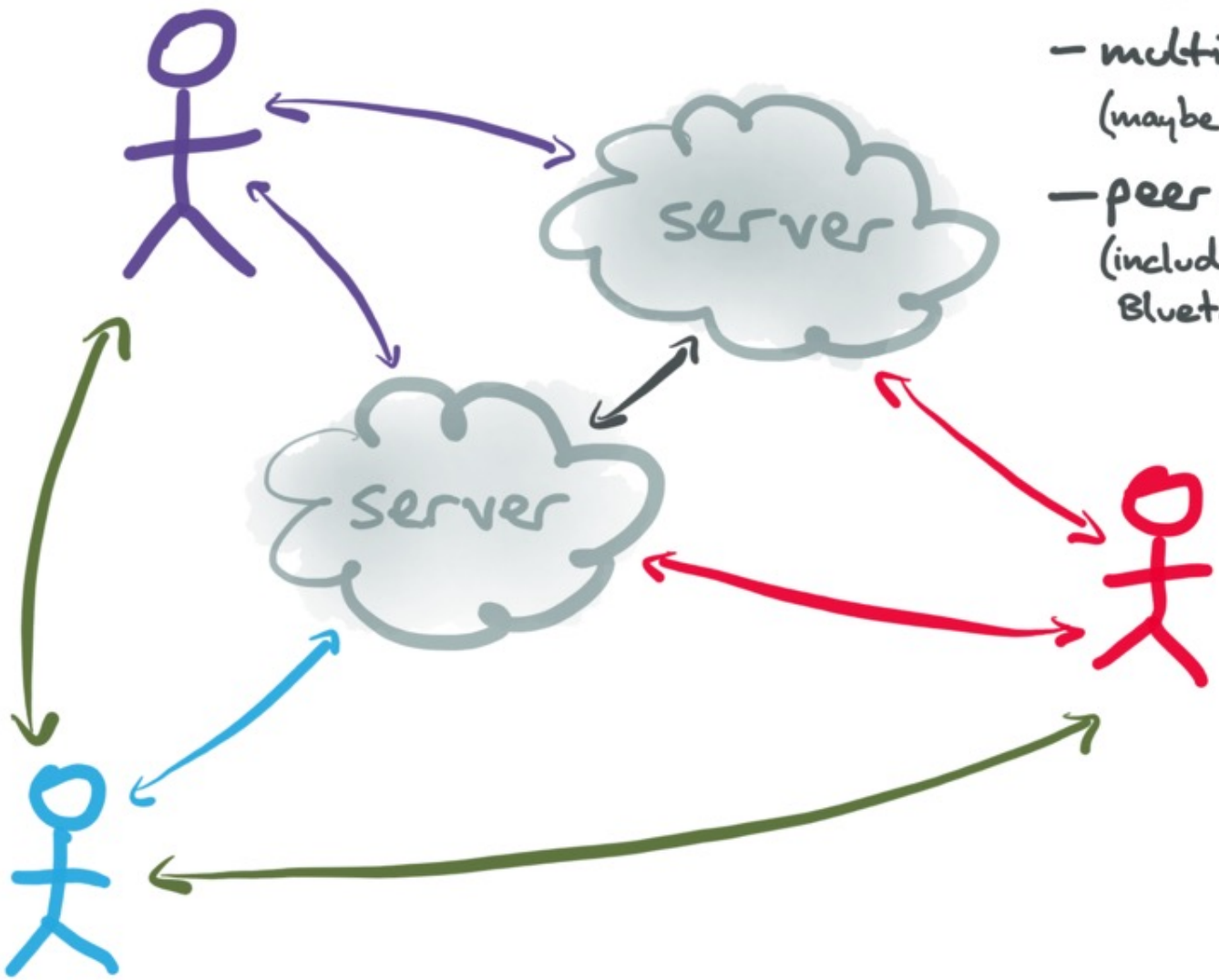


COLLABORATION IN ANY NETWORK TOPOLOGY

- one server
- multiple servers (maybe federated)



COLLABORATION IN ANY NETWORK TOPOLOGY



- one server
- multiple servers (maybe federated)
- peer-to-peer (including LAN, Bluetooth, etc.)

TODAY'S TOPICS.

Integrity in the presence of Byzantine nodes

1. Byzantine eventual consistency + invariants
2. Byzantine fault tolerant CRDTs
3. Authenticated snapshots

Confidentiality against Byzantine nodes

4. Decentralised access control list
5. End-to-end encryption

TODAY'S TOPICS.

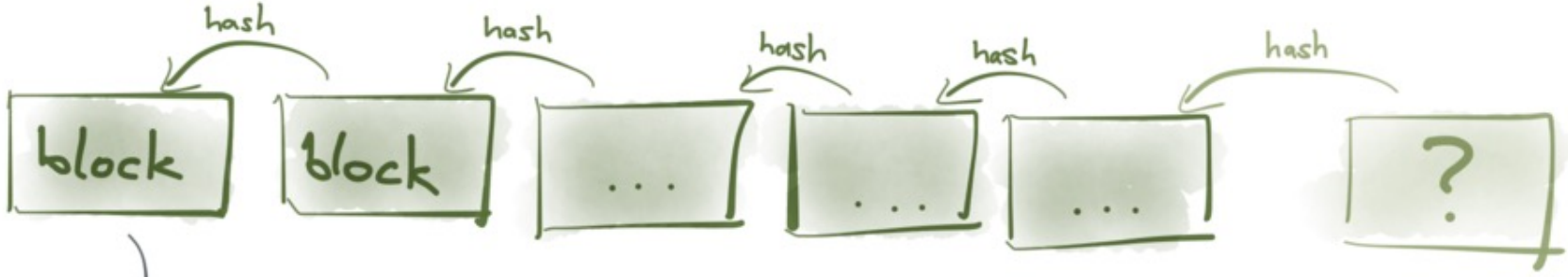
Integrity in the presence of Byzantine nodes

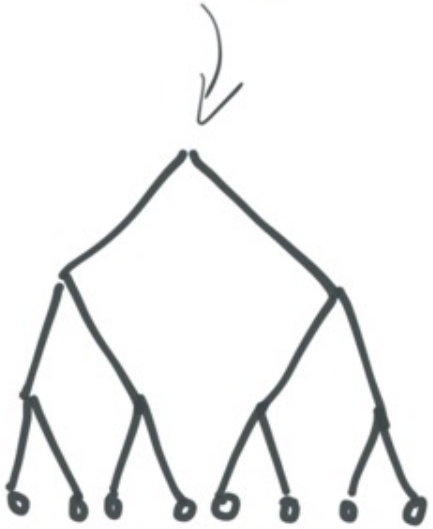
1. Byzantine eventual consistency + invariants
2. Byzantine fault tolerant CRDTs
3. Authenticated snapshots

Confidentiality against Byzantine nodes

4. Decentralised access control list
5. End-to-end encryption

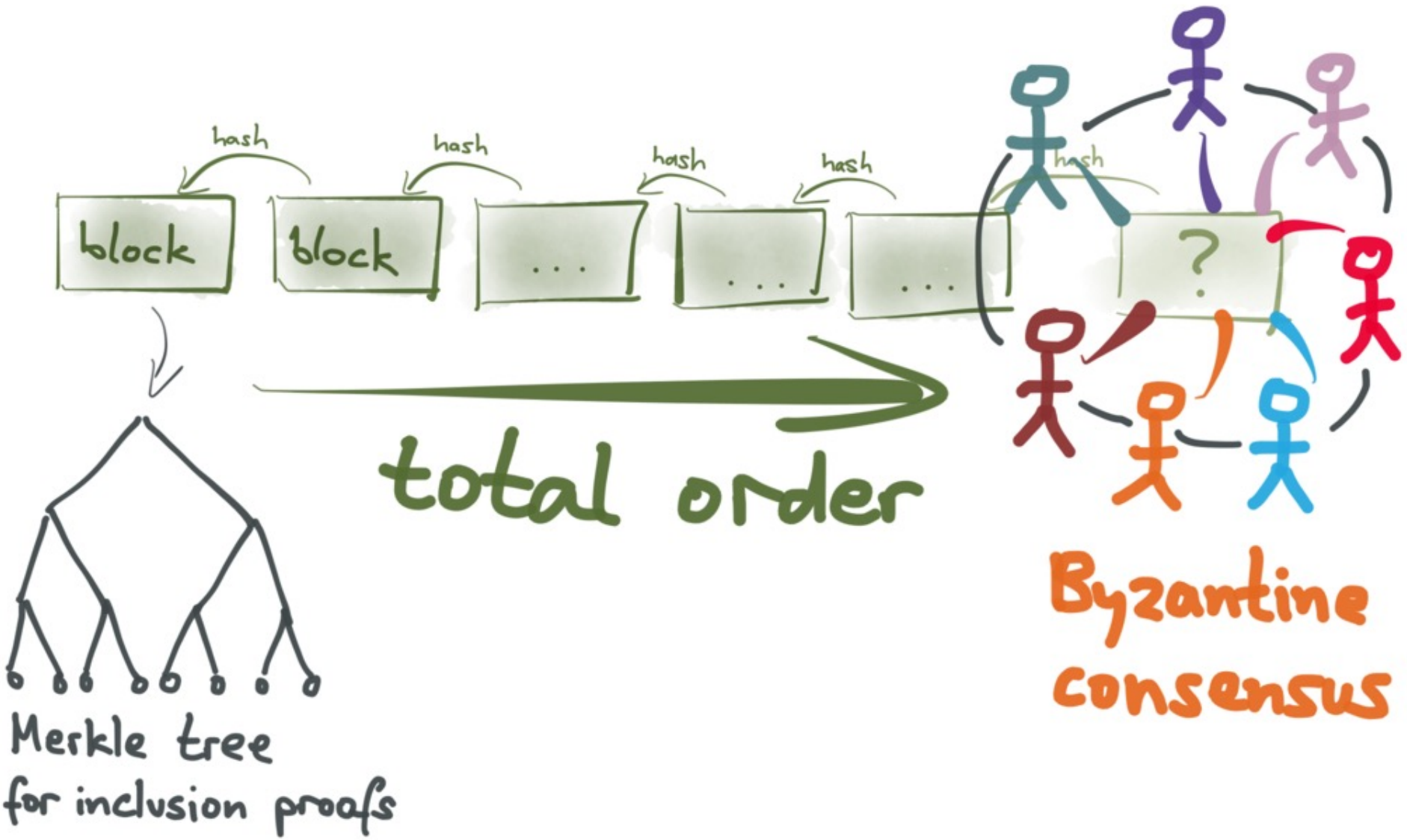
BLOCKCHAIN TO THE RESCUE?



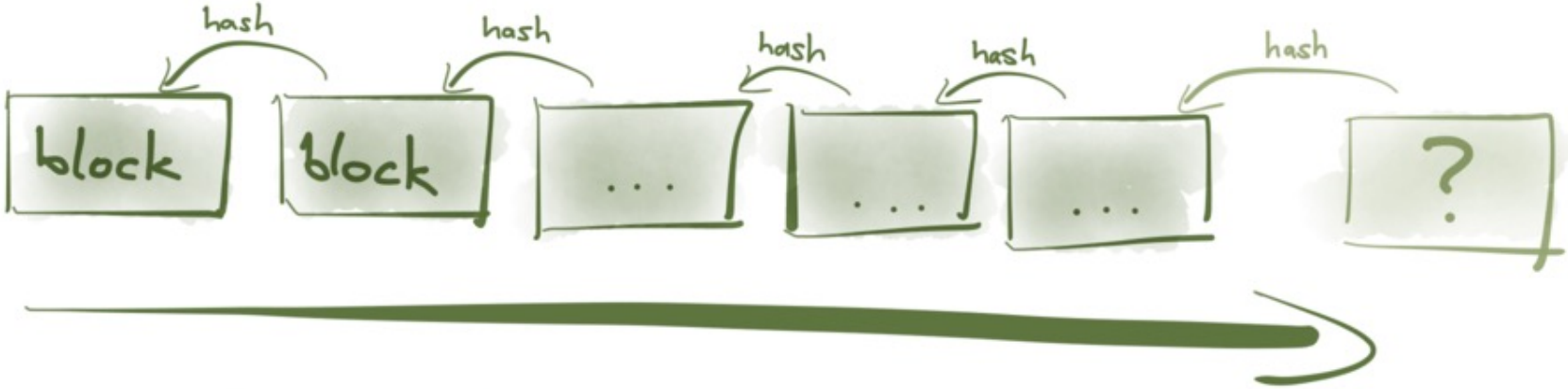


Merkle tree
for inclusion proofs

BLOCKCHAIN TO THE RESCUE?

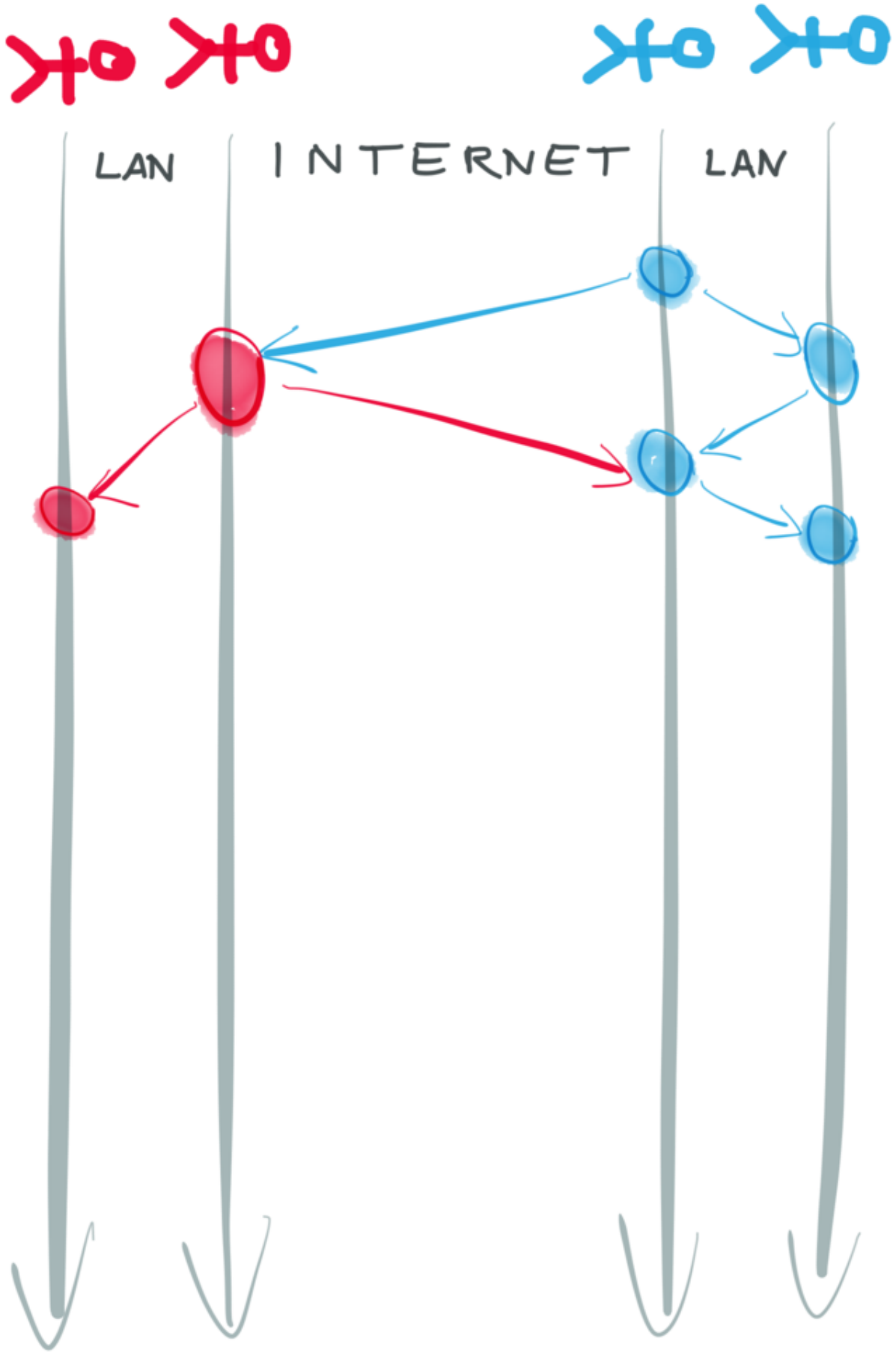


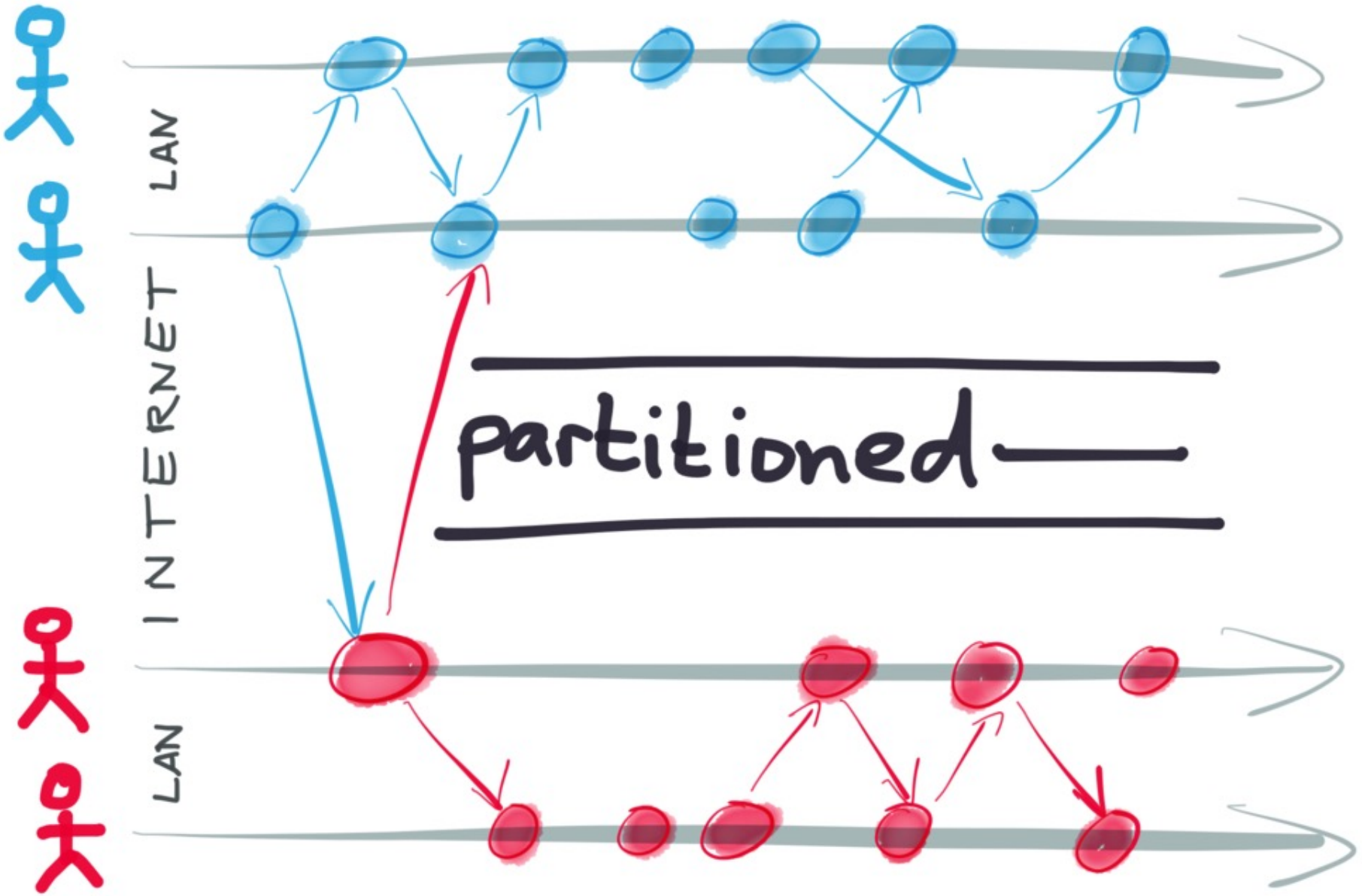
BLOCKCHAIN TO THE RESCUE?

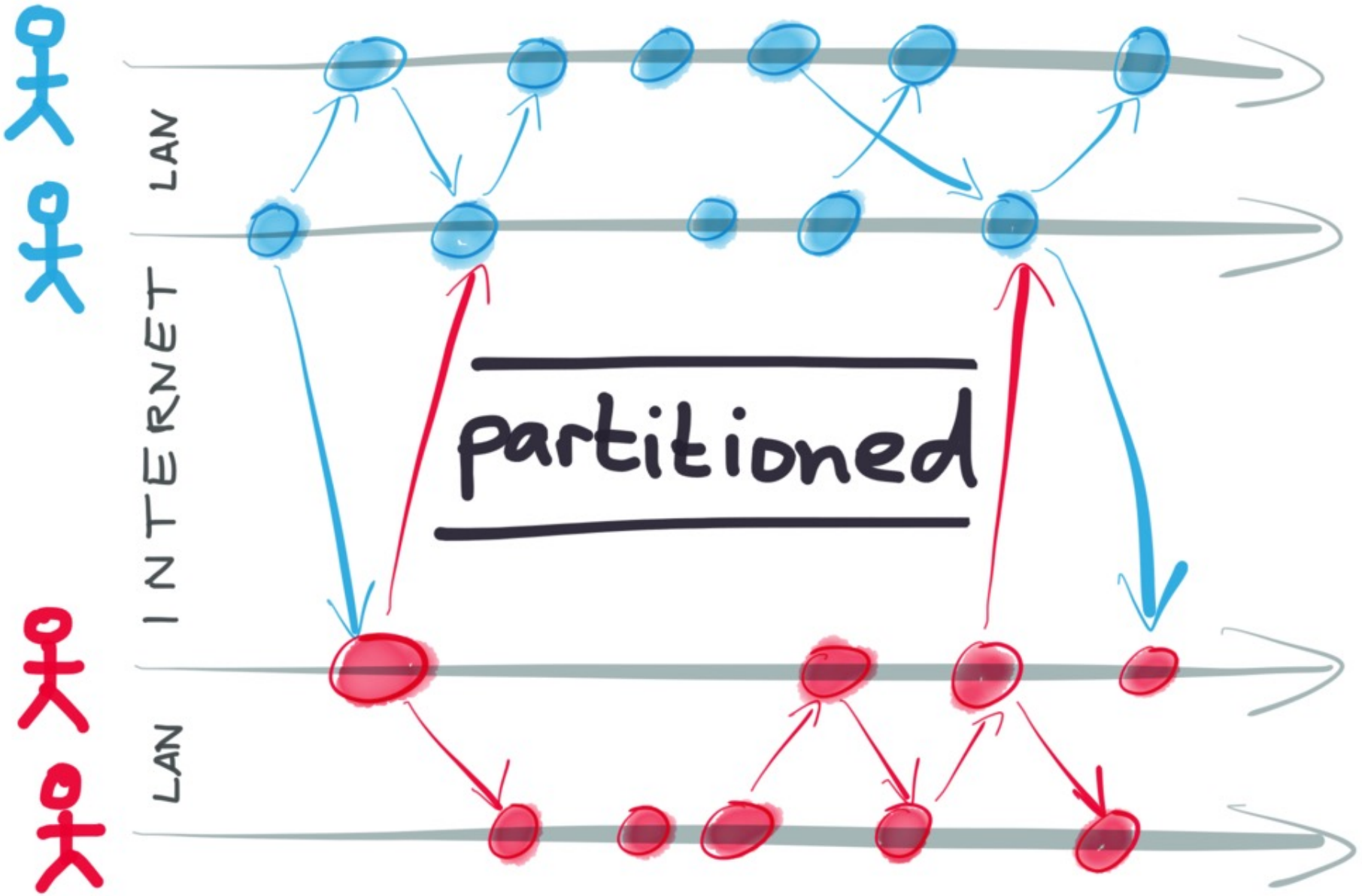


Total order required for cryptocurrencies (to prevent double-spending).

Overkill for replication!


LAN
INTERNET
LAN


LAN
INTERNET
partitioned
LAN


LAN
INTERNET
partitioned
LAN

https://arxiv.org/abs/2012.00472

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

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.

PROBLEM: Byzantine agreement requires $3f+1$ nodes to tolerate f malicious nodes

Need to prevent Sybil attacks

PROBLEM: Byzantine agreement requires $3f+1$ nodes to tolerate f malicious nodes

Need to prevent Sybil attacks

Permissioned blockchains: centralised control over who can vote

PROBLEM: Byzantine agreement requires $3f+1$ nodes to tolerate f malicious nodes

Need to prevent Sybil attacks

Permissioned blockchains: centralised control over who can vote

Votes allocated based on a finite resource

proof of work

More than f out of $3f+1$ nodes Byzantine-faulty?

ALL BETS ARE OFF.

More than f out of 3f+1 nodes Byzantine-faulty?

ALL BETS ARE OFF.

WHAT IF... some apps can handle ANY NUMBER of Byzantine-faulty nodes?

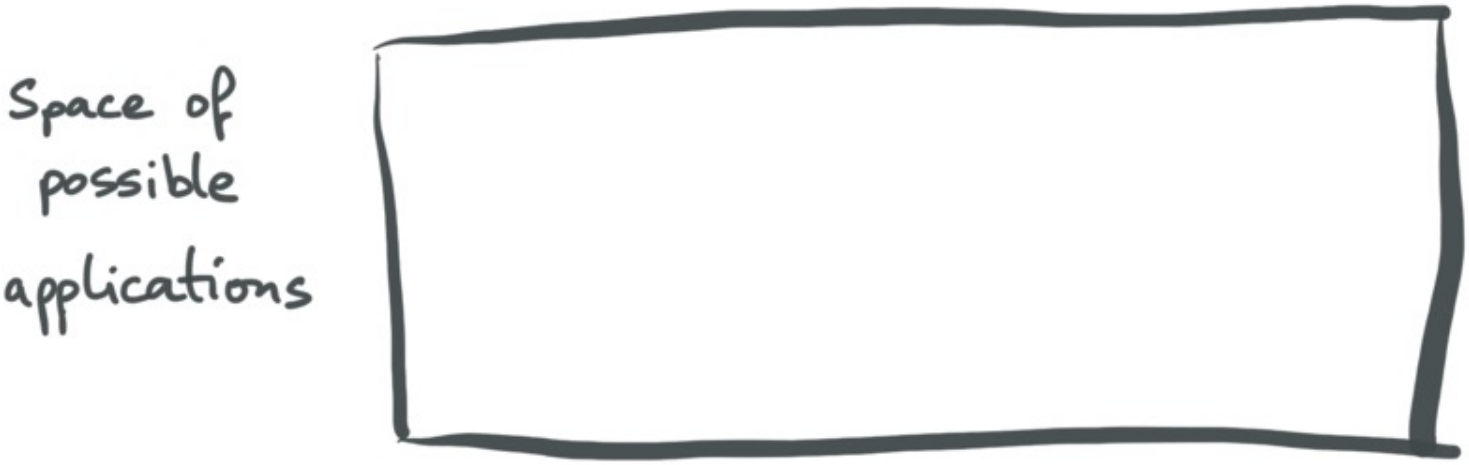
No voting... no Sybil attacks... no proof of work!

More than f out of 3f+1 nodes Byzantine-faulty?

ALL BETS ARE OFF.

WHAT IF... some apps can handle ANY NUMBER of Byzantine-faulty nodes?

No voting... no Sybil attacks... no proof of work!



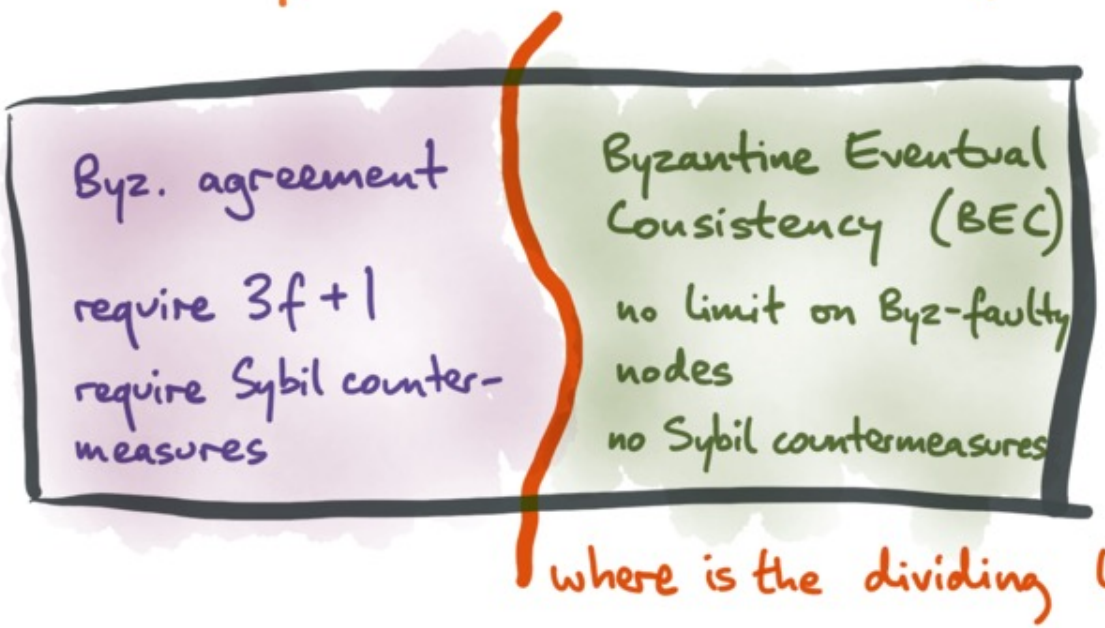
More than f out of 3f+1 nodes Byzantine-faulty?

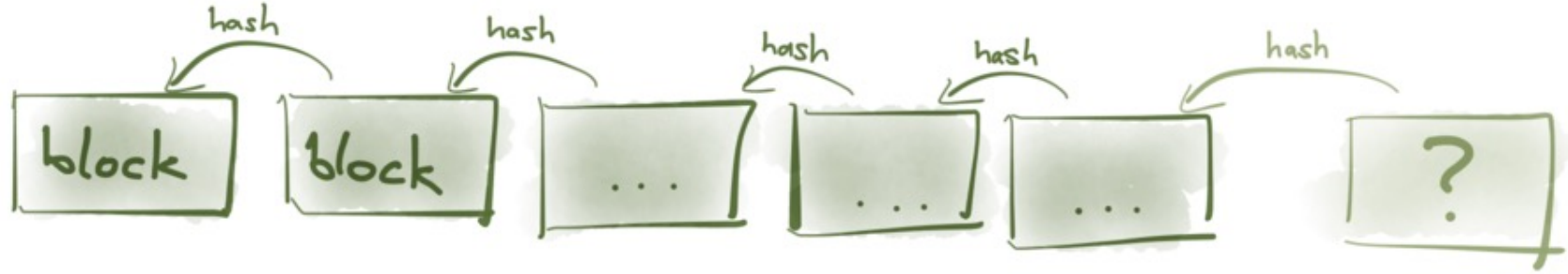
ALL BETS ARE OFF.

WHAT IF... some apps can handle ANY NUMBER of Byzantine-faulty nodes?

No voting... no Sybil attacks... no proof of work!

Space of possible applications





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

Consensus = pick one of several proposed values

Collaboration = keep all edits and merge them

Byzantine Eventual Consistency (BEC)

Eventual update:

One correct replica applies update u

$\Rightarrow$ 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: ⟵ use CRDTs

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: ⟵ 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)

Byzantine Eventual Consistency (BEC)

Eventual update:

One correct replica applies update u

$\Rightarrow$ all correct replicas eventually apply u

in the face of any number of Byz-faulty replicas

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)

Invariants?

state of a replica

A function $I(s) \rightarrow \{true, false\}$

Examples:

- Every account has a non-negative balance
- The same coin is not spent more than once
- Every action is performed by a user who is authorised to do so
- Every username is unique
- Every ID refers to an object that exists (i.e. foreign key integrity)

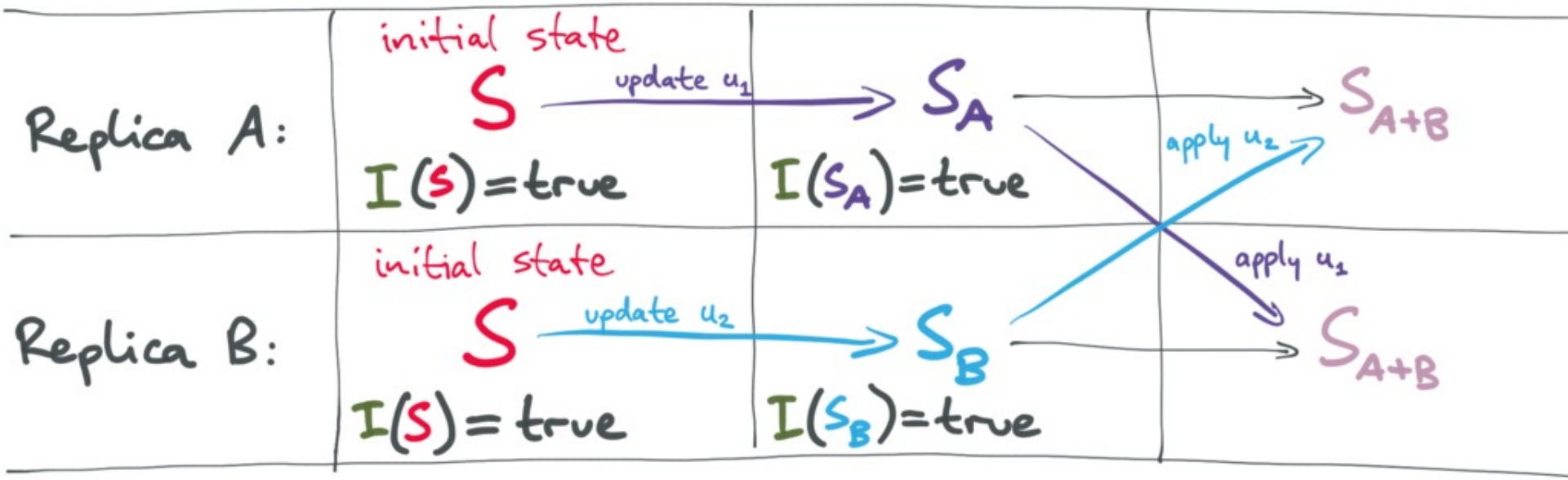
Invariant confluence

Replica A:	initial state S $I(S) = \text{true}$	
Replica B:	initial state S $I(S) = \text{true}$	

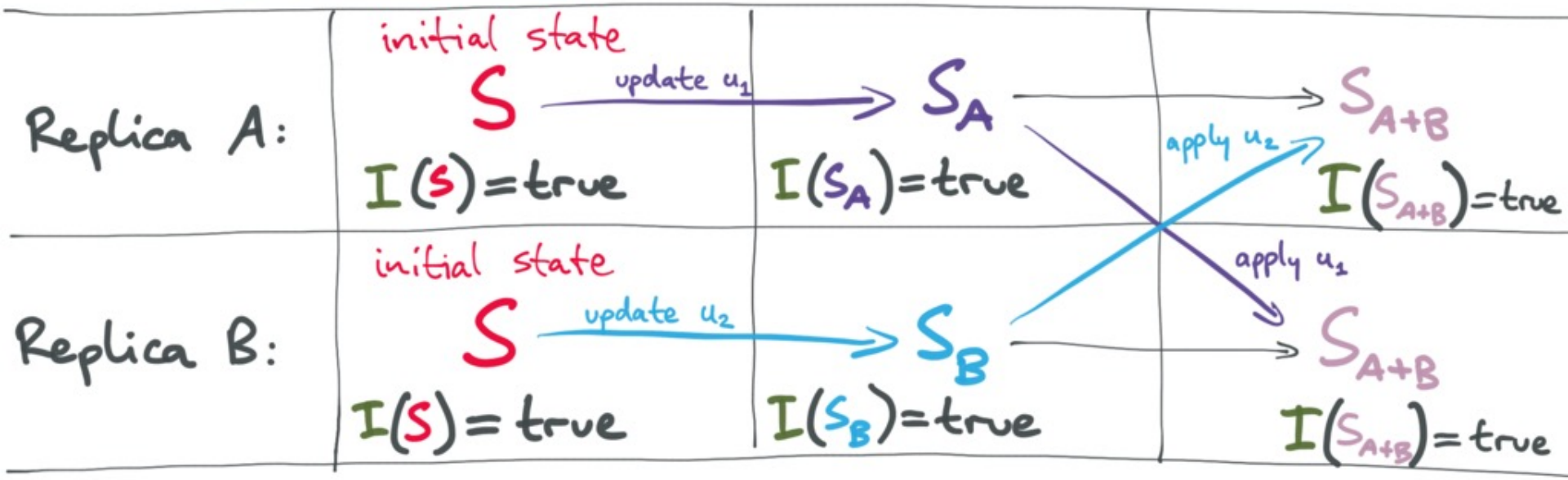
Invariant confluence

Replica A:	initial state S	update u_1 → S_A	
	$I(S)$ = true	$I(S_A)$ = true	
Replica B:	initial state S	update u_2 → S_B	
	$I(S)$ = true	$I(S_B)$ = true	

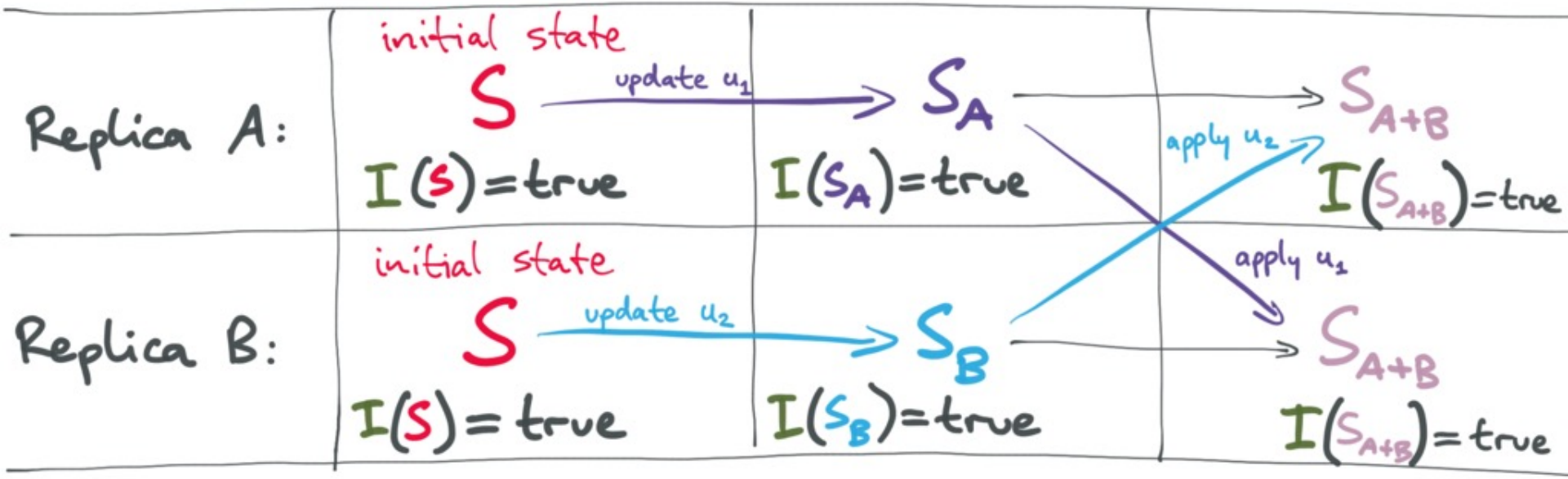
Invariant confluence



Invariant confluence



Invariant confluence

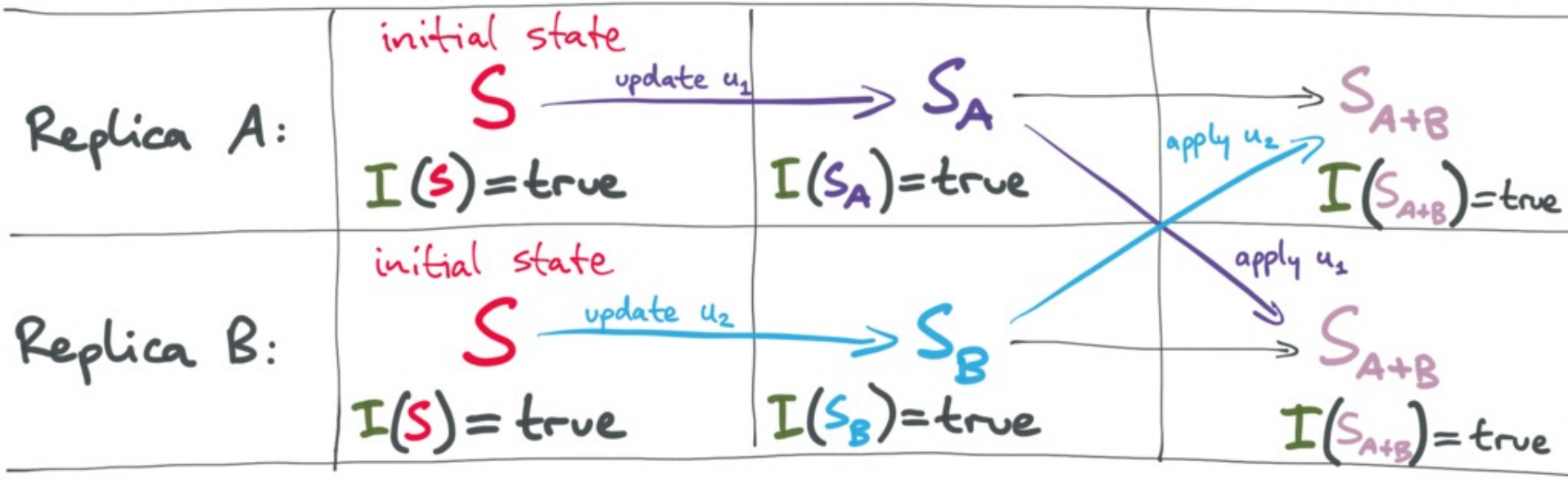


Examples:

- I = no negative balances, u_1 & u_2 decrease account balances

  $\Rightarrow$ $\{u_1, u_2\}$ are NOT invariant confluent with respect to I

Invariant confluence



Examples:

- I = no negative balances, u_1 & u_2 decrease account balances

  $\Rightarrow$ $\{u_1, u_2\}$ are NOT invariant confluent with respect to I

- I = no negative balances, u_1 & u_2 increase account balances

  $\Rightarrow$ $\{u_1, u_2\}$ ARE invariant confluent with respect to I

Theorem:

There exists an algorithm that ensures BEC
with any number of Byzantine-faulty replicas

if and only if

all updates are invariant confluent w.r.t. all invariants

($\Rightarrow$) impossibility: there is no such algorithm if not invariant confluent

($\Leftarrow$) existence: we have a BEC replication algorithm!

RESULT:

It is possible to guarantee Strong Eventual Consistency in a system with arbitrarily many Byzantine nodes, no trusted nodes, no proof-of-X

It is possible to retrofit BFT to existing CRDT algorithms, with modest changes

Assumption (unavoidable): every correct node can communicate with every other correct node (directly, or indirectly via other correct nodes)

TODAY'S TOPICS.

Integrity in the presence of Byzantine nodes

1. Byzantine eventual consistency + invariants
2. Byzantine fault tolerant CRDTs
3. Authenticated snapshots

Confidentiality against Byzantine nodes

4. Decentralised access control list
5. End-to-end encryption

Byzantine Eventual Consistency (BEC)

Eventual update:

One correct replica applies update u

$\Rightarrow$ all correct replicas eventually apply u

in the face of any number of Byz-faulty replicas

Convergence: ← use CRDTs

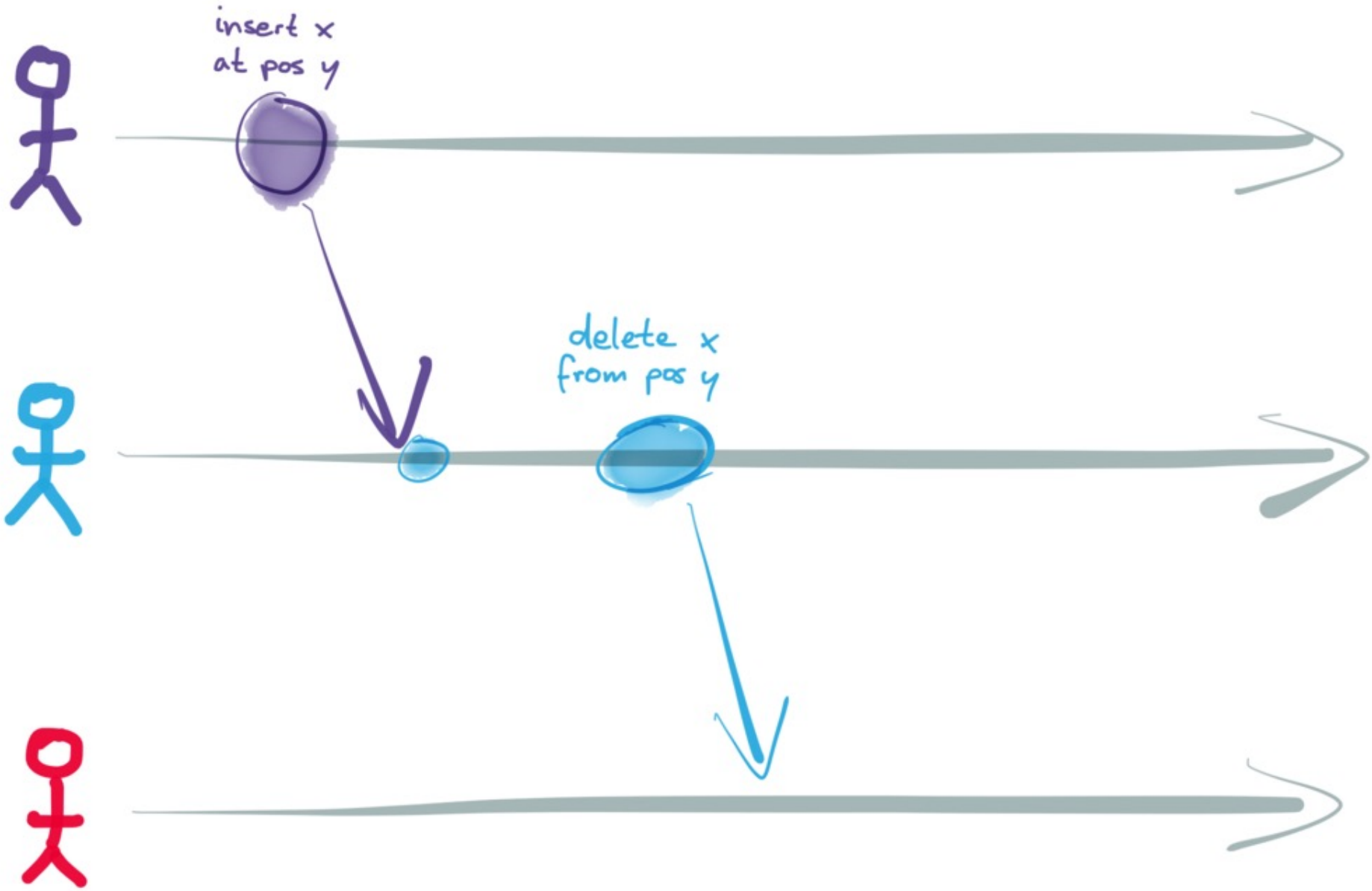
Two replicas have applied same set of updates

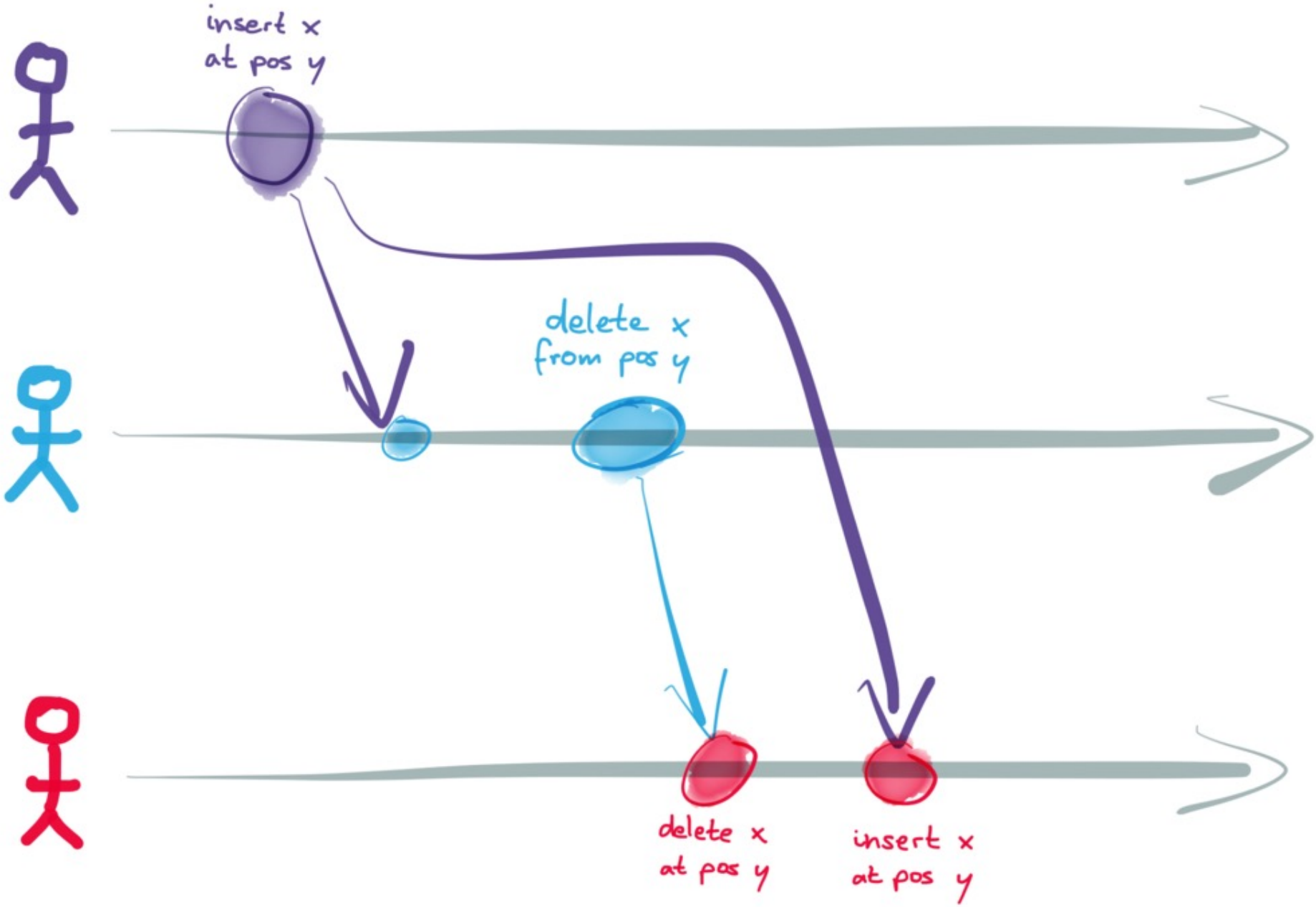
$\Rightarrow$ they are in the same state

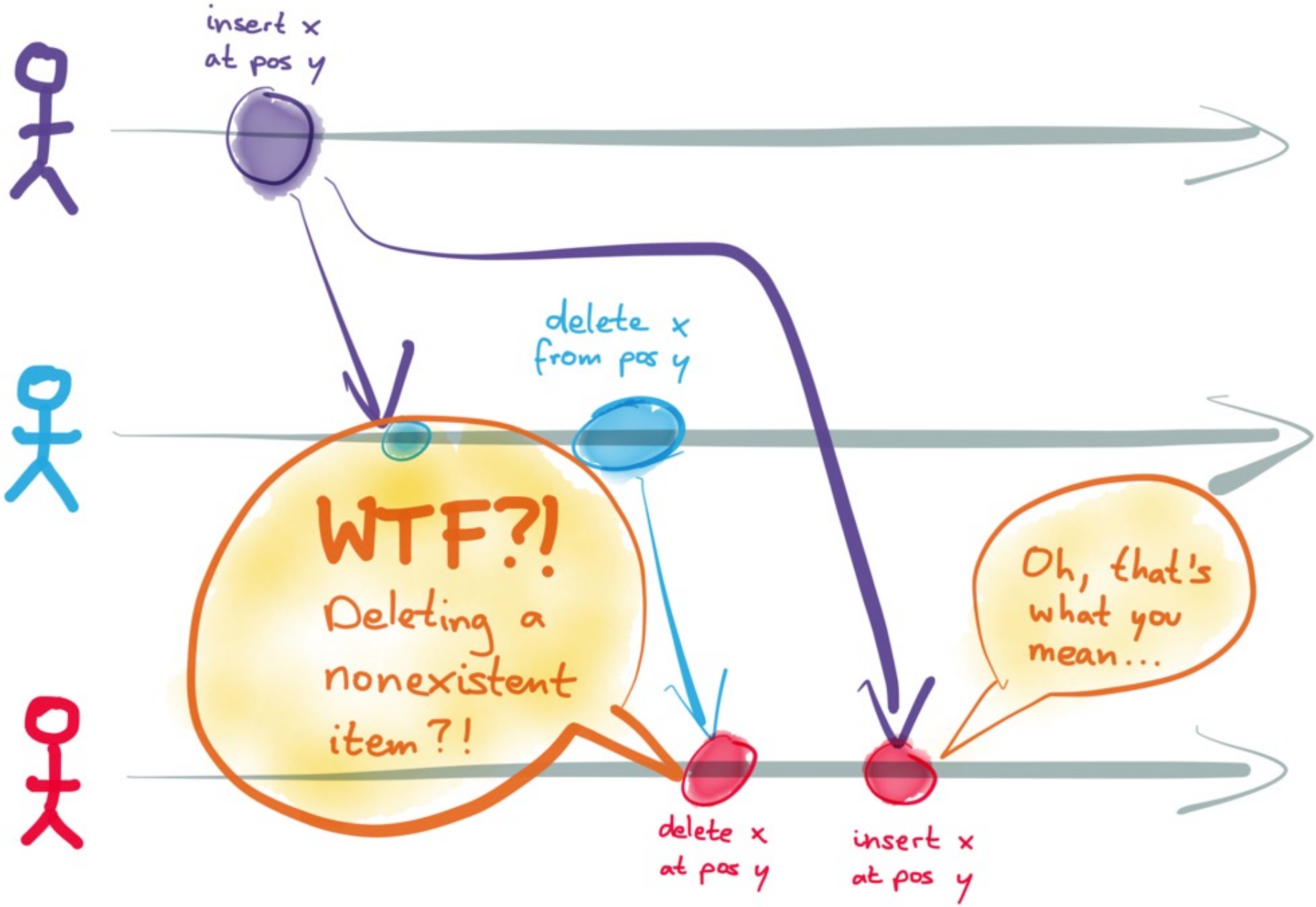
Invariant preservation:

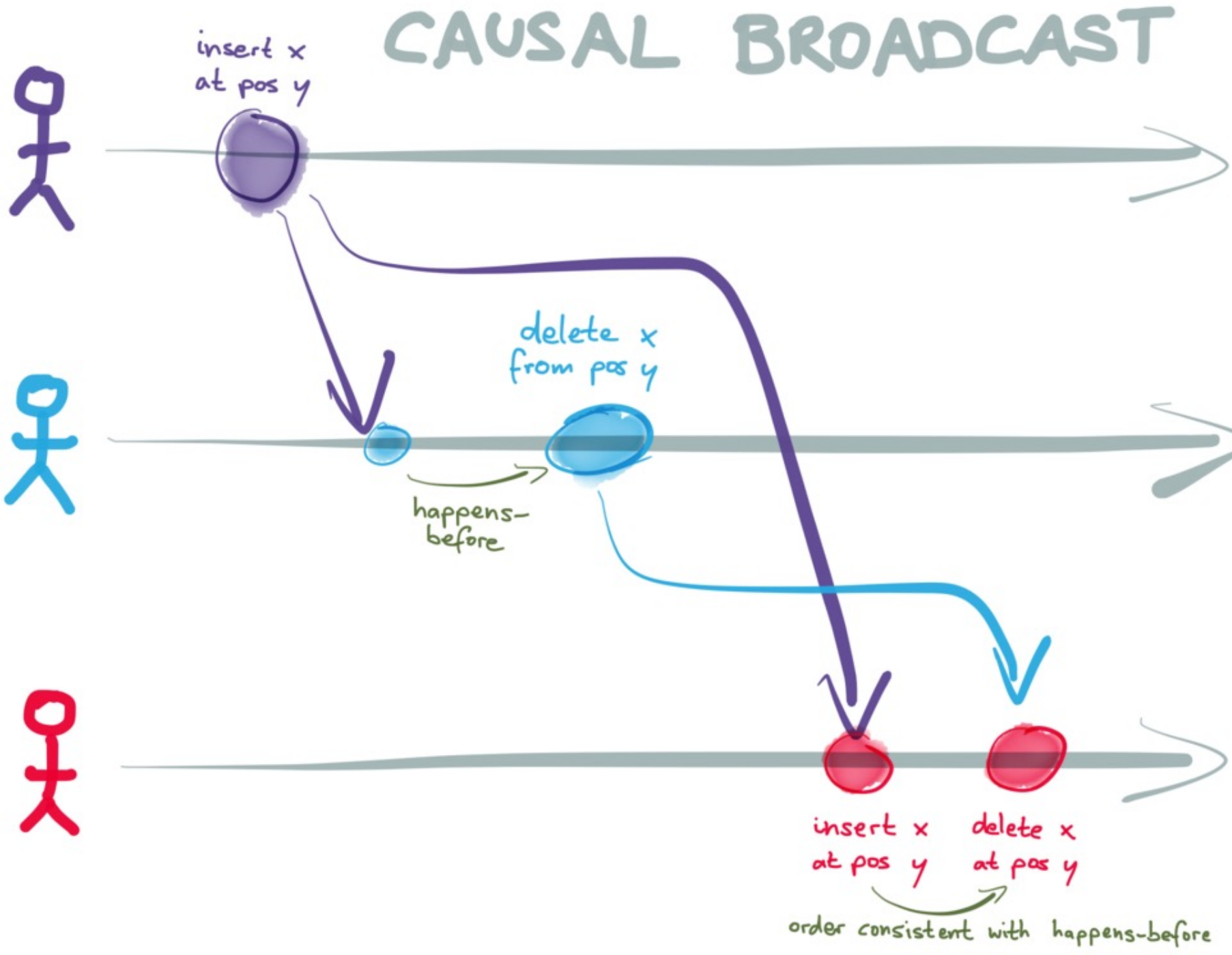
The state of a correct replica always satisfies all of the app's declared invariants

(and a few other, more technical properties)

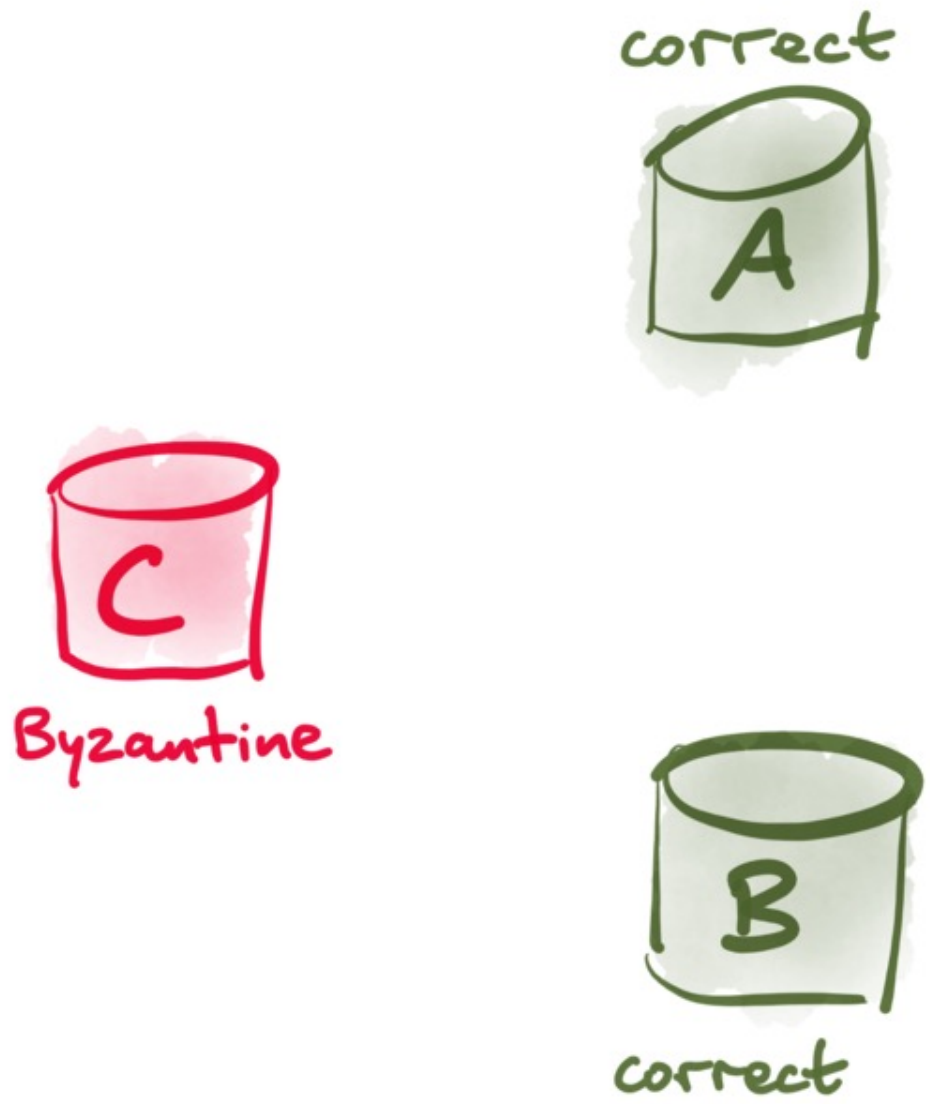

insert x
at pos y
delete x
from pos y


insert x
at pos y
delete x
from pos y
delete x
at pos y
insert x
at pos y


insert x
at pos y
delete x
from pos y
WTF?!
Deleting a
nonexistent
item?!
Oh, that's
what you
mean...
delete x
at pos y
insert x
at pos y

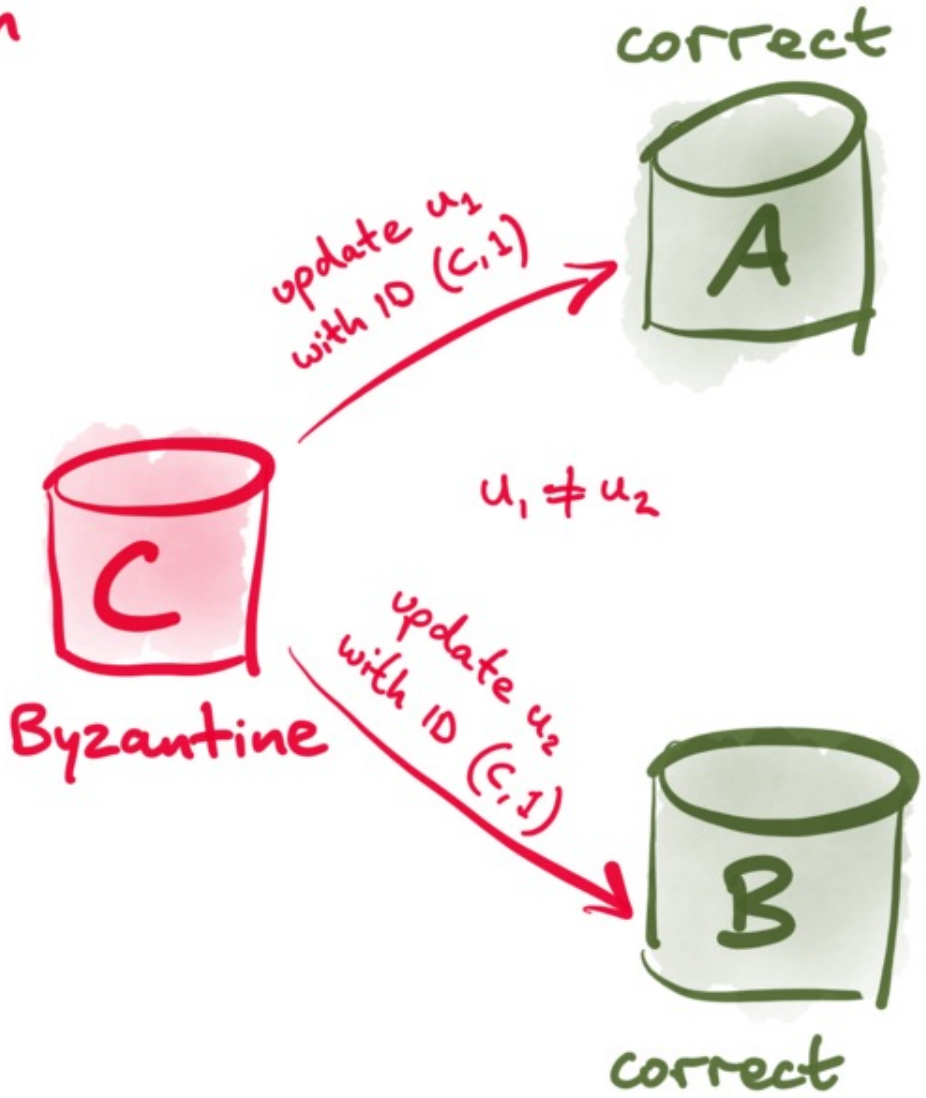

CAUSAL BROADCAST
insert x
at pos y
delete x
from pos y
happens-
before
insert x
at pos y
delete x
at pos y
order consistent with happens-before

VERSION VECTORS ARE NOT SAFE

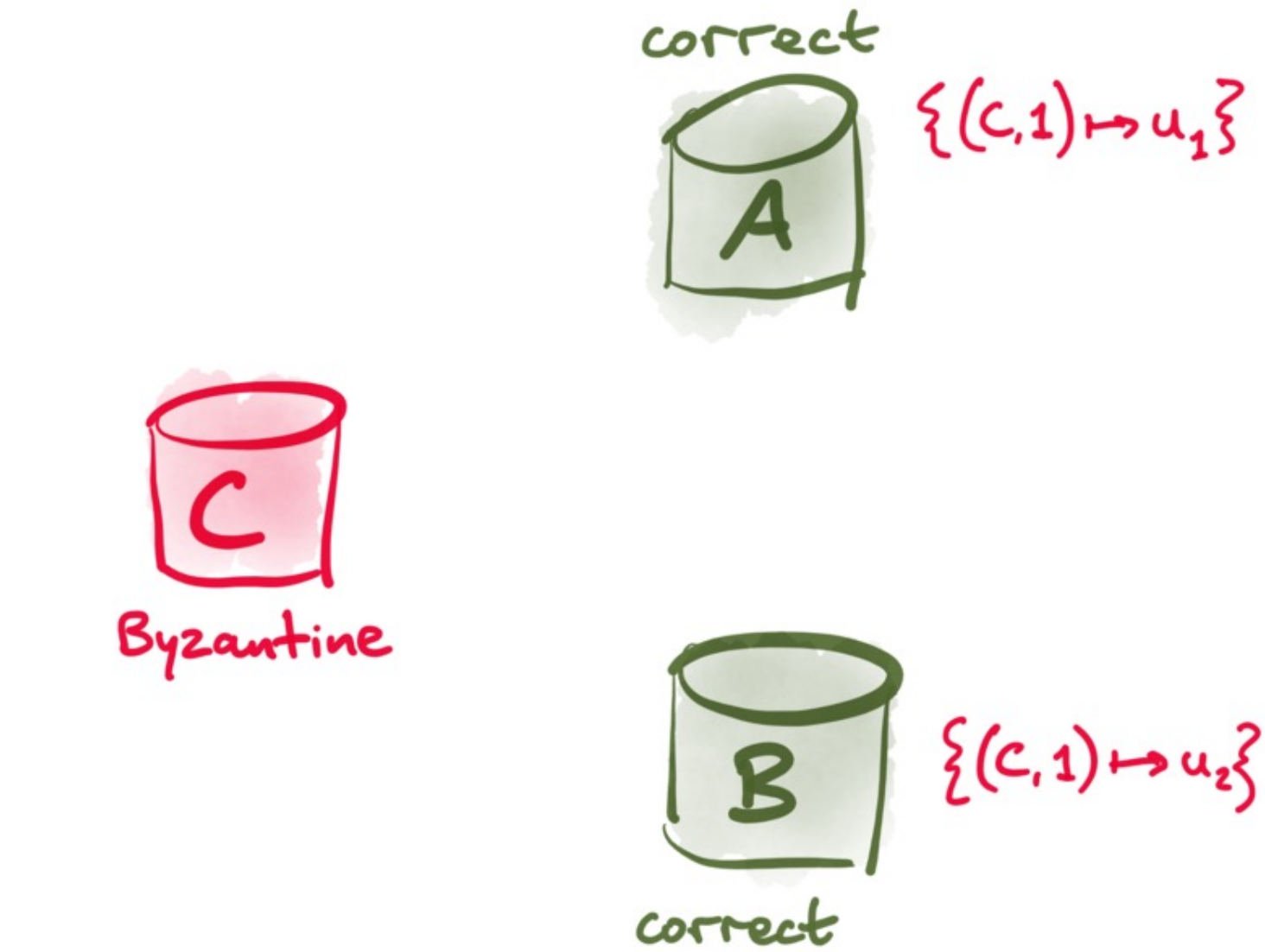


VERSION VECTORS ARE NOT SAFE

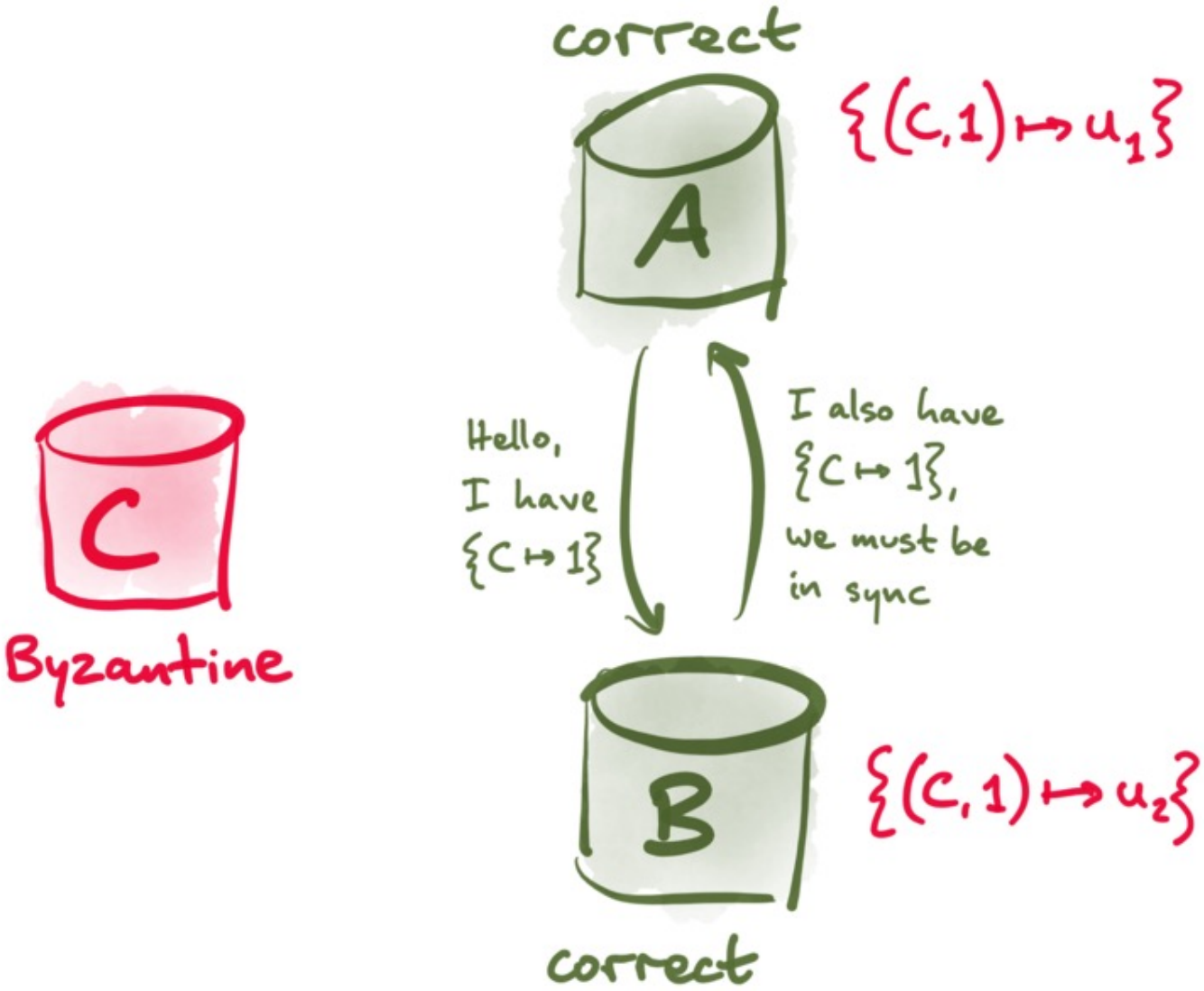
"equivocation"



VERSION VECTORS ARE NOT SAFE



VERSION VECTORS ARE NOT SAFE



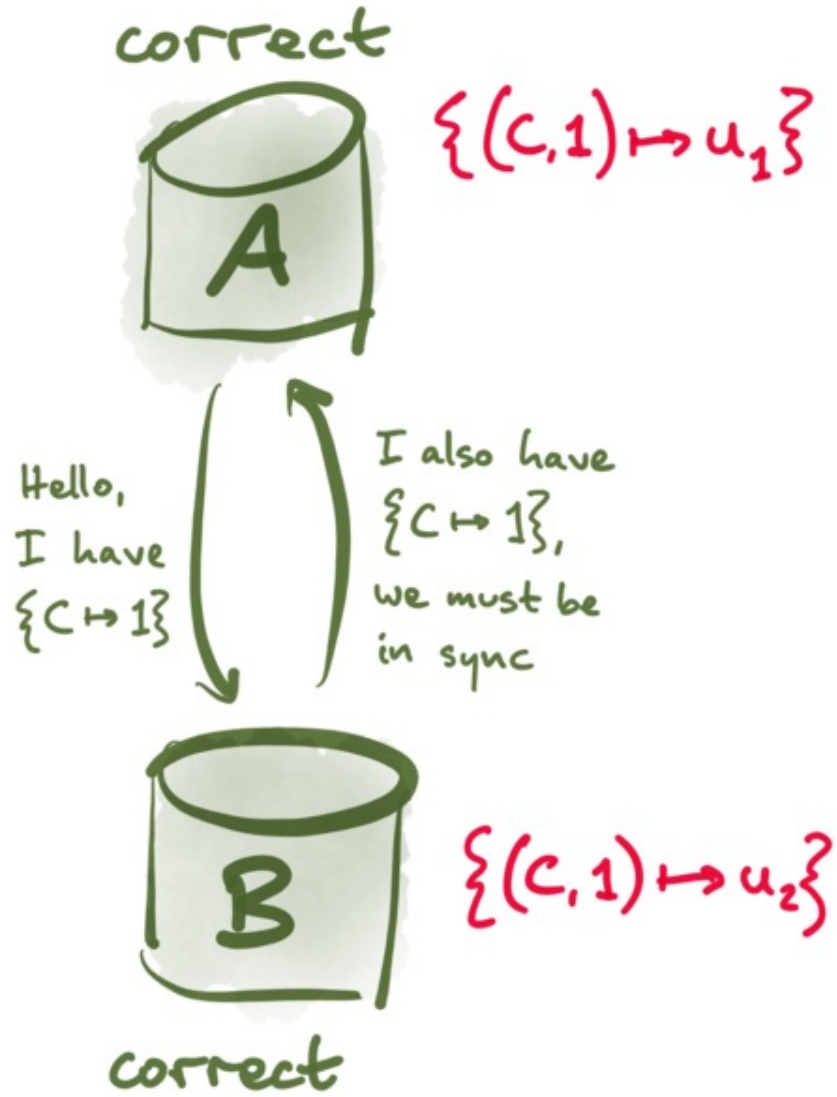
VERSION VECTORS ARE NOT SAFE

A never delivers u_2

B never delivers u_1

$\Rightarrow$ failure of eventual delivery

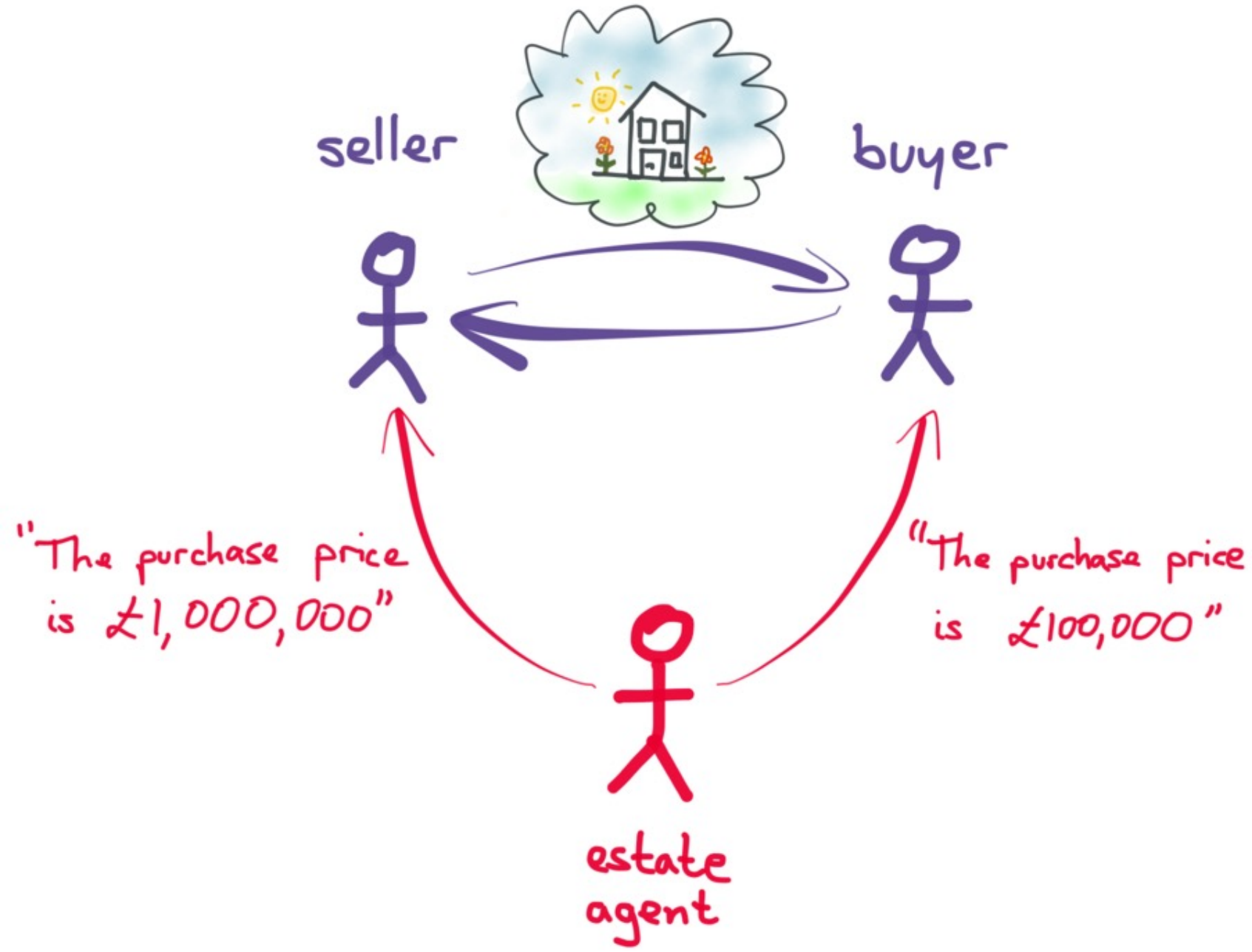
Byzantine

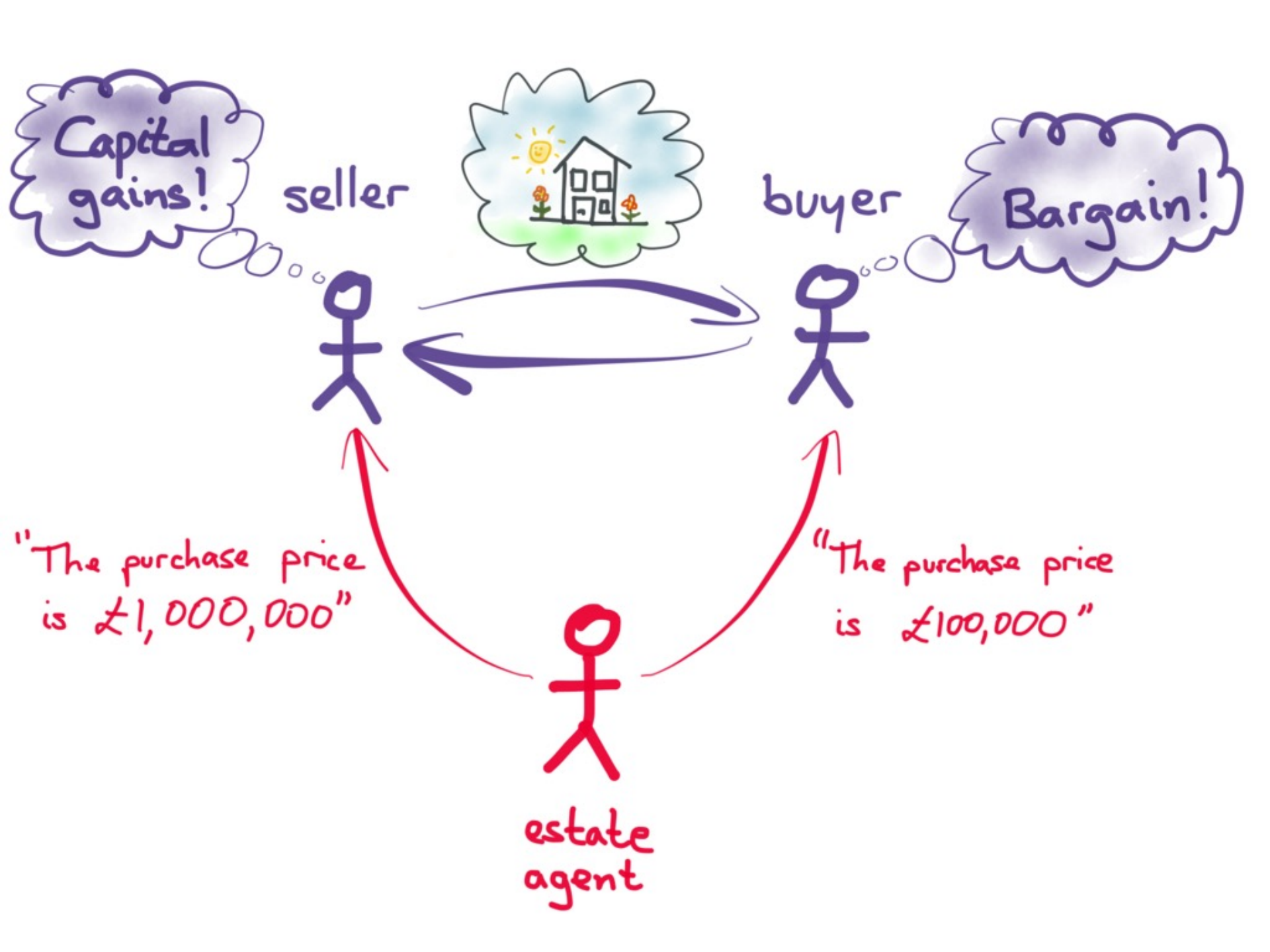


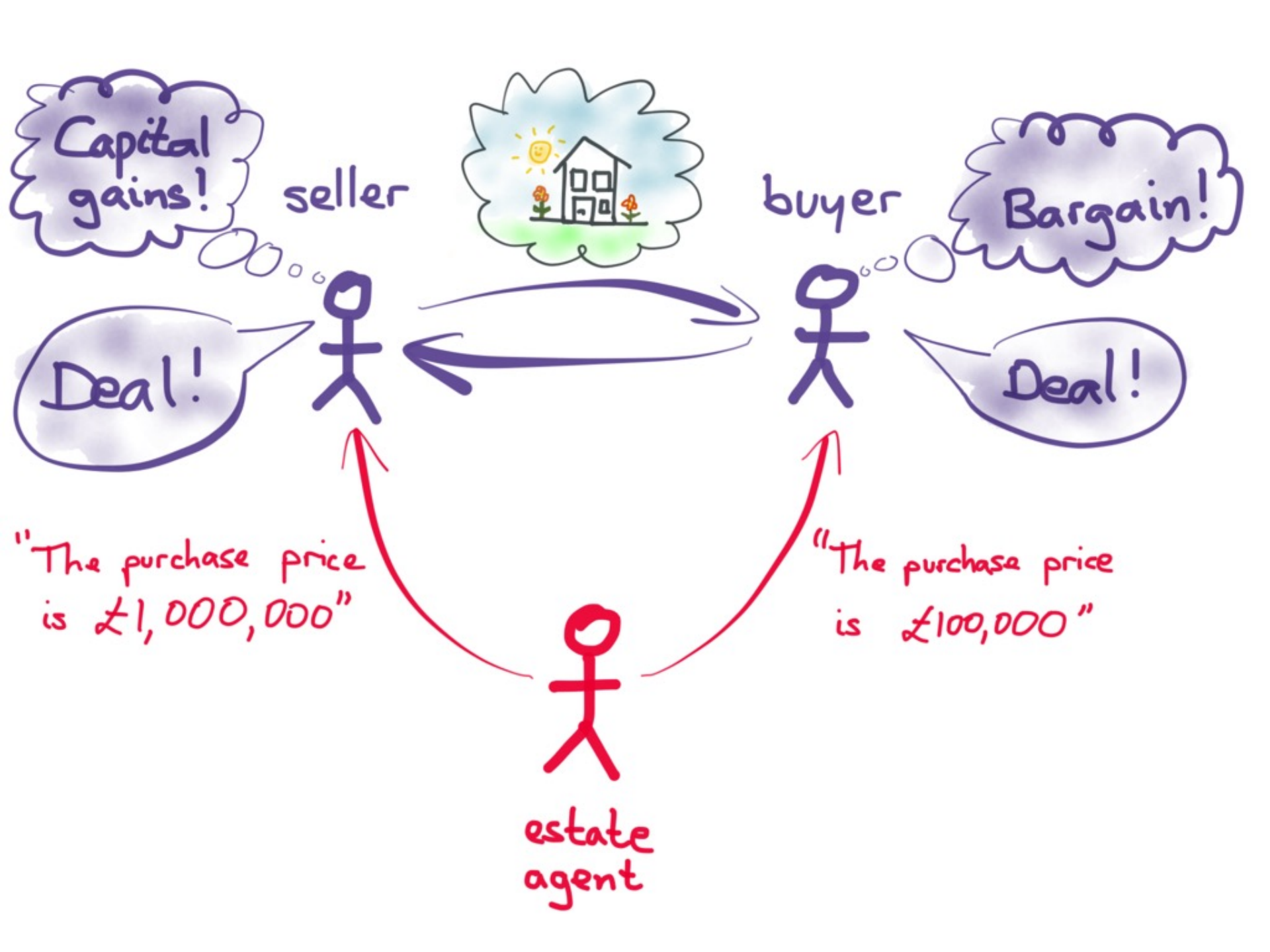

seller
buyer


seller
buyer


estate
agent


seller
buyer
"The purchase price is £1,000,000"
"The purchase price is £100,000"
estate agent


Capital gains!
seller
buyer
Bargain!
"The purchase price is £1,000,000"
"The purchase price is £100,000"
estate agent

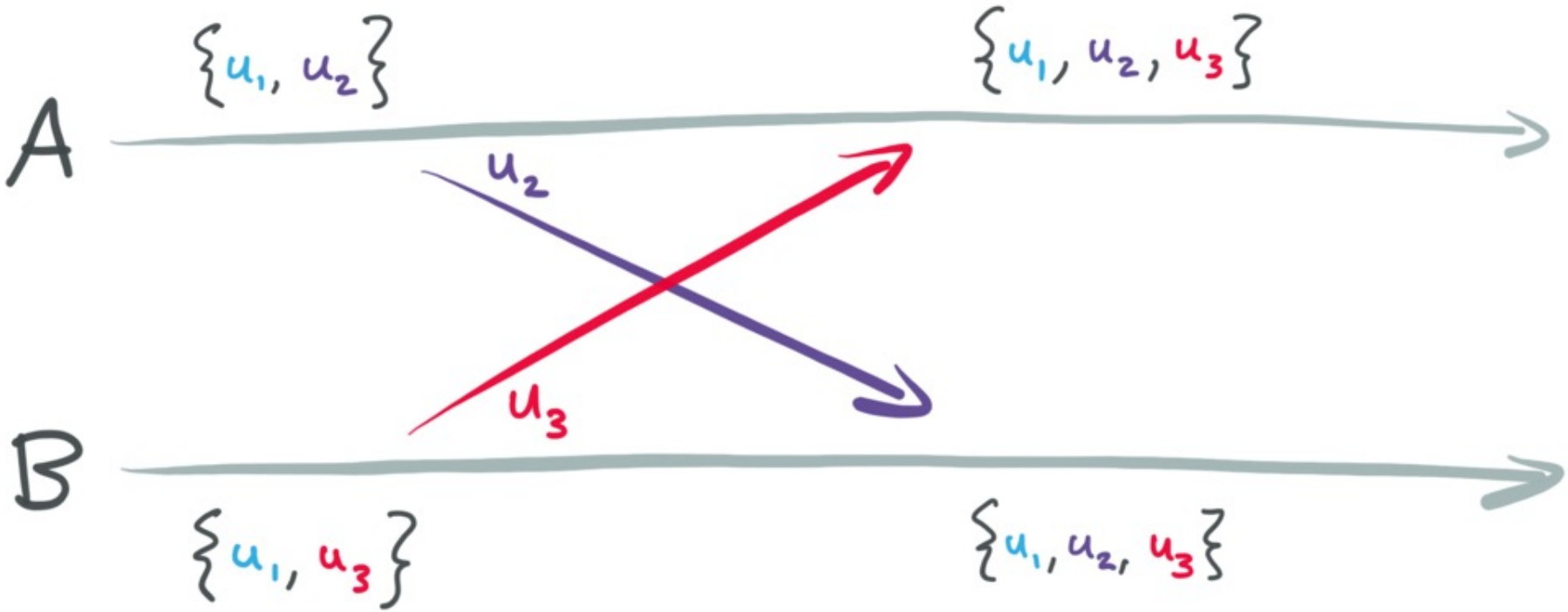

Capital gains!
seller
buyer
Bargain!
Deal!
Deal!
"The purchase price is £1,000,000"
"The purchase price is £100,000"
estate agent

ENSURING EVENTUAL DELIVERY

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

ENSURING EVENTUAL DELIVERY

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



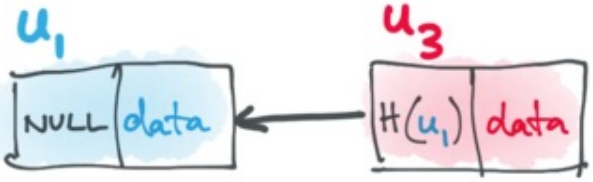
ENSURING EVENTUAL DELIVERY

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

Hash graph (like Git!):

A

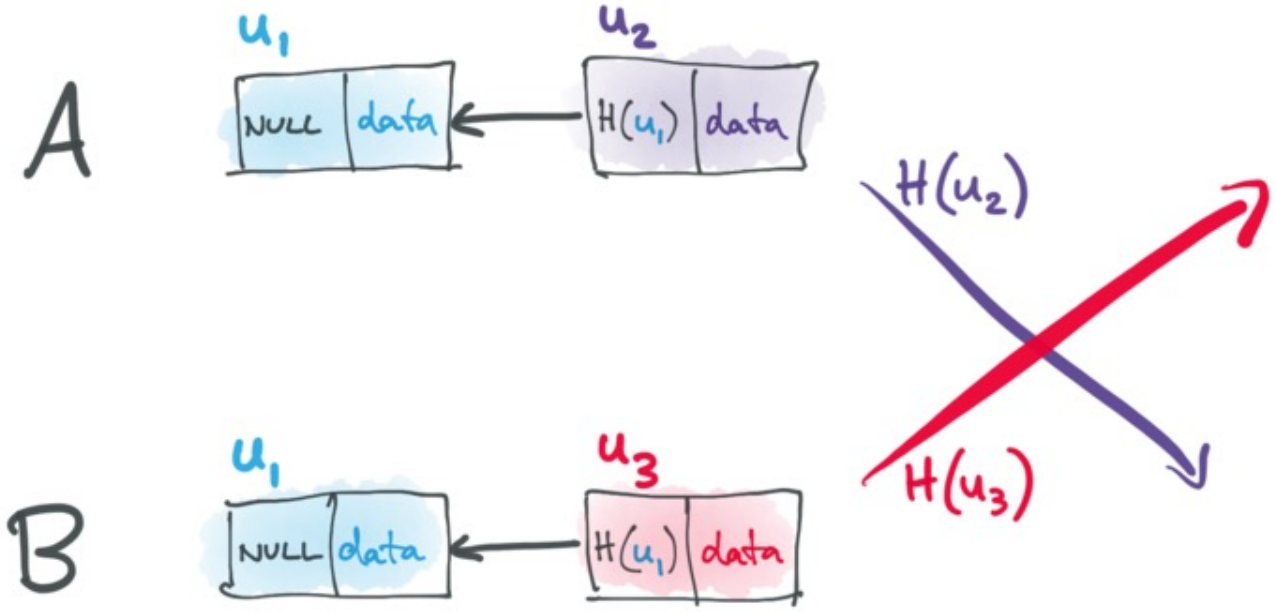
B



ENSURING EVENTUAL DELIVERY

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

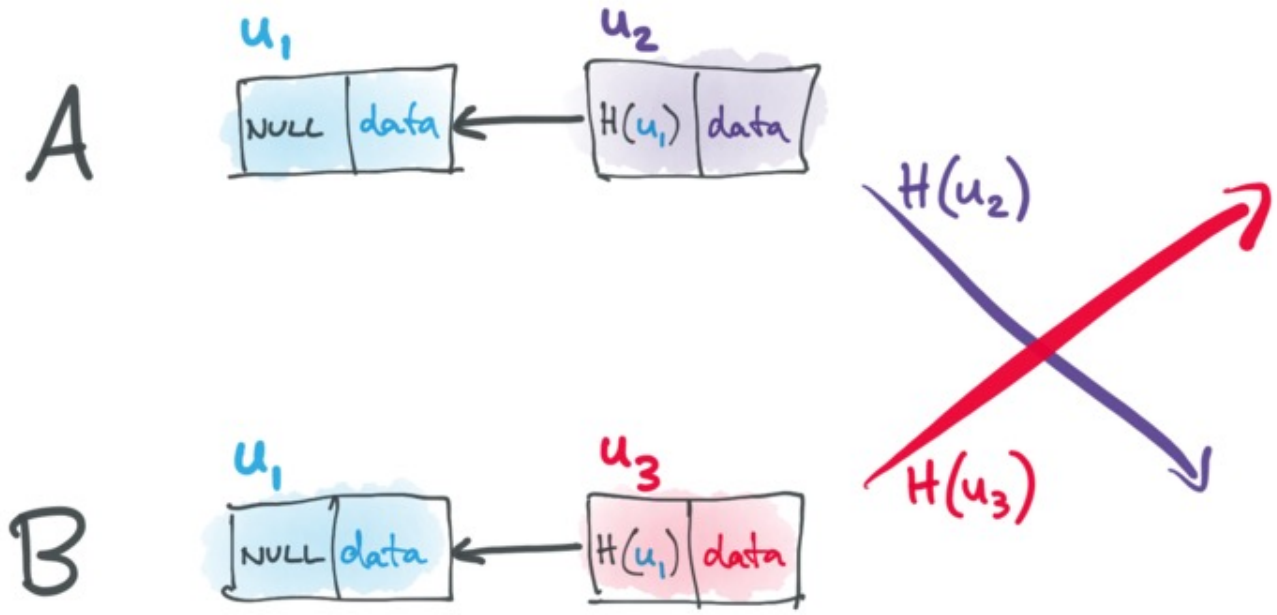
Hash graph (like Git!):



ENSURING EVENTUAL DELIVERY

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

Hash graph (like Git!):

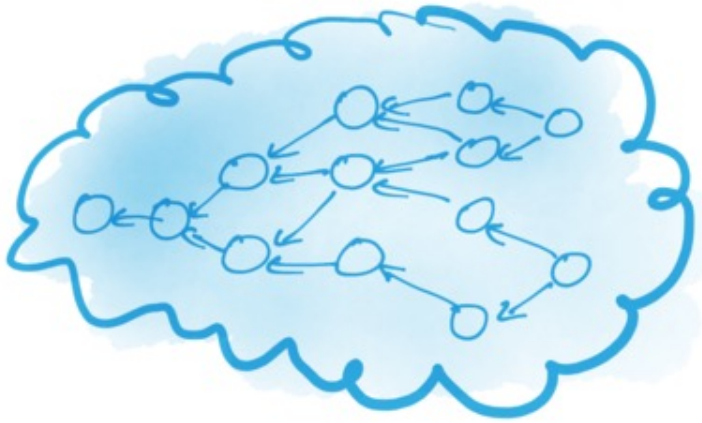


hash unknown?
work backwards
in hash DAG
until known hash
is found

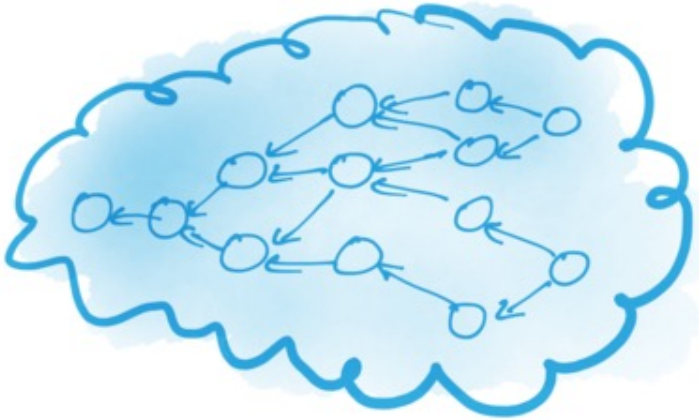
Assuming a collision-resistant hash function, a Byzantine node cannot cause two correct nodes to believe they are in sync when in fact they have diverged.

BEC replication

A



B

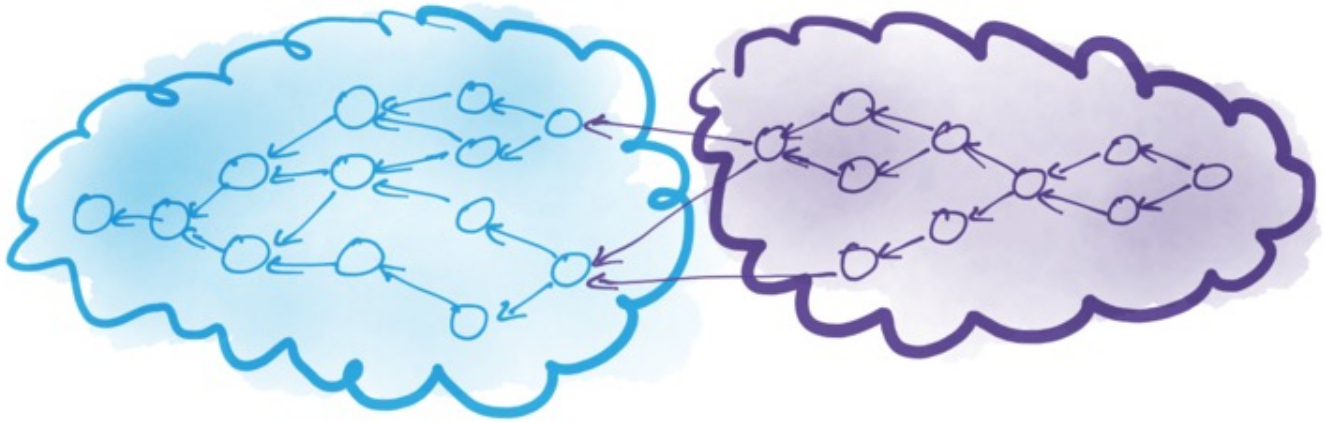


remember result of last sync between A and B

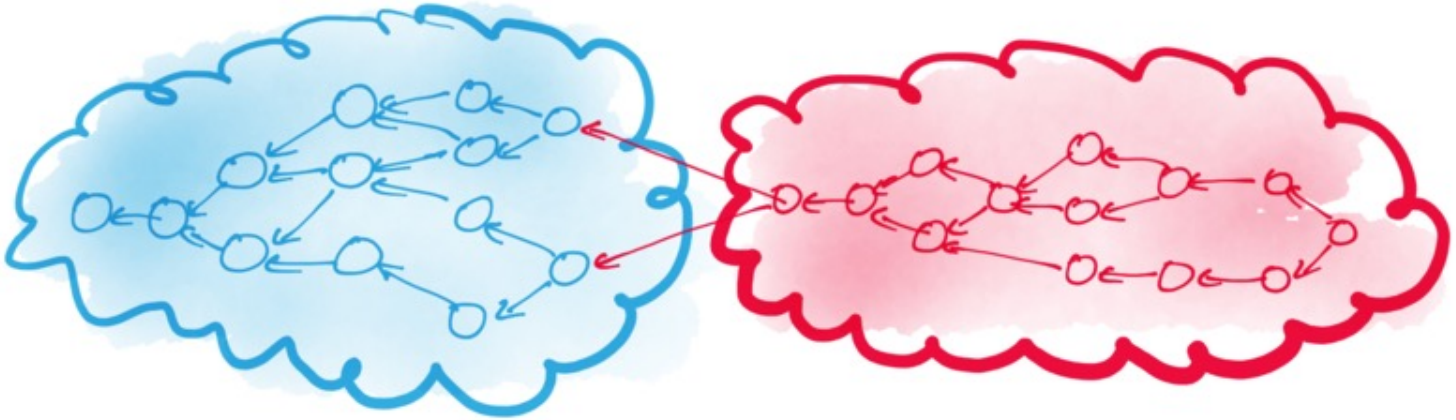
BEC replication

added by A since last sync

A



B



remember result of last sync between A and B

added by B since last sync

BEC replication

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

in the face of any number of Byz-faulty replicas

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)

Making CRDTs Byzantine Fault Tolerant

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

Abstract

It is often claimed that Conflict-free Replicated Data Types (CRDTs) ensure consistency of replicated data in peer-to-peer systems. However, peer-to-peer systems usually consist of untrusted nodes that may deviate from the specified protocol (i.e. exhibit Byzantine faults), and most existing CRDT algorithms cannot guarantee consistency in the presence of such faults. This paper shows how to adapt existing non-Byzantine CRDT algorithms and make them Byzantine fault-tolerant. The proposed scheme can tolerate any number of Byzantine nodes (making it immune to Sybil attacks), guarantees Strong Eventual Consistency, and requires only modest changes to existing CRDT algorithms.

CCS Concepts: • **Computer systems organization** → **Peer-to-peer architectures**; • **Security and privacy** → *Distributed systems security*.

Keywords: Byzantine fault tolerance, CRDTs, optimistic replication, eventual consistency

whether the deviation is by accident or by malice [20]. Byzantine behaviour is not always detectable by other nodes, since a Byzantine node may attempt to hide its protocol violations. One approach would be to try to identify Byzantine nodes and exclude them from the system, but this is unlikely to be effective if the banned node can simply rejoin the system under a different identity, and the system would still have to somehow repair the damage already done by the Byzantine node. A more robust approach is to *tolerate* Byzantine faults: that is, to ensure that the system can meet its advertised guarantees even when some of its nodes are Byzantine-faulty.

Conflict-free Replicated Data Types (CRDTs) [34] are often presented as an approach for providing consistency of replicated data in peer-to-peer systems [27, 36, 38], because they do not require a central server for concurrency control. However, despite ostensibly targeting P2P systems, the vast majority of CRDT algorithms actually do not tolerate Byzantine faults, since they assume that all participating nodes correctly follow the protocol. If such algorithms are deployed in a system with Byzantine nodes, the algorithm

ENSURING CONVERGENCE

Problem: Byzantine nodes may generate arbitrarily malformed updates. Must ensure convergence for any possible set of updates that might exist.

ENSURING CONVERGENCE

Problem: Byzantine nodes may generate arbitrarily malformed updates. Must ensure convergence for any possible set of updates that might exist.

Unique IDs

Non-Byzantine: ID is pair (counter, replica ID)

Byzantine node may generate multiple updates with the same ID

ENSURING CONVERGENCE

Problem: Byzantine nodes may generate arbitrarily malformed updates. Must ensure convergence for any possible set of updates that might exist.

Unique IDs

Non-Byzantine: ID is pair (counter, replica ID)

Byzantine node may generate multiple updates with the same ID

Byzantine: ID is the hash of the update that uses the ID!

First generate the update, then hash it, then apply it to the CRDT state

ENSURING CONVERGENCE

Checking whether an update is valid, ignore it if not.

All correct nodes must make same accept/reject decision!

ENSURING CONVERGENCE

Checking whether an update is valid, ignore it if not.

All correct nodes must make same accept/reject decision!

Validity is deterministic function of update alone? → easy.

valid(u) → {true, false}

ENSURING CONVERGENCE

Checking whether an update is valid, ignore it if not.

All correct nodes must make same accept/reject decision!

Validity is deterministic function of update alone? → easy.

$valid(u) \rightarrow \{true, false\}$

Validity depends on update and current state of replica?

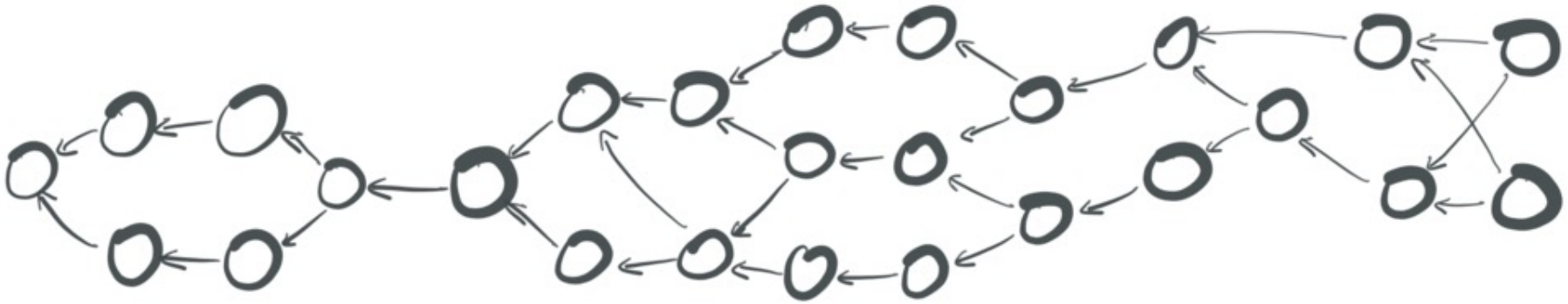
(e.g. you can only delete/update an object that exists, error if it does not exist)

$valid(u, state) \rightarrow \{true, false\}$

different replicas might be in different states when applying the same update

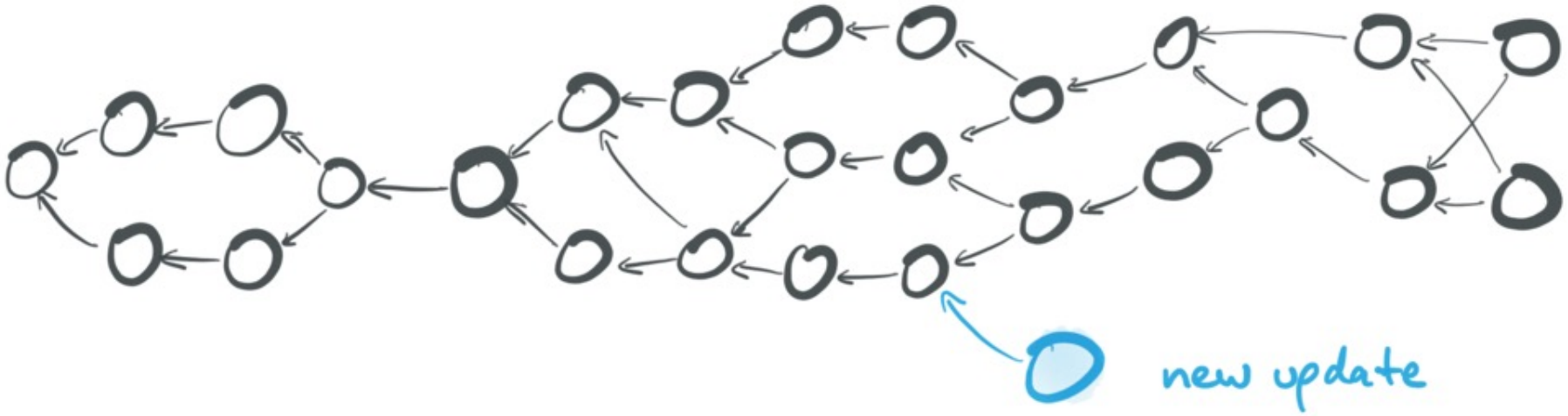
ENSURING CONVERGENCE

all of the updates previously delivered at this node



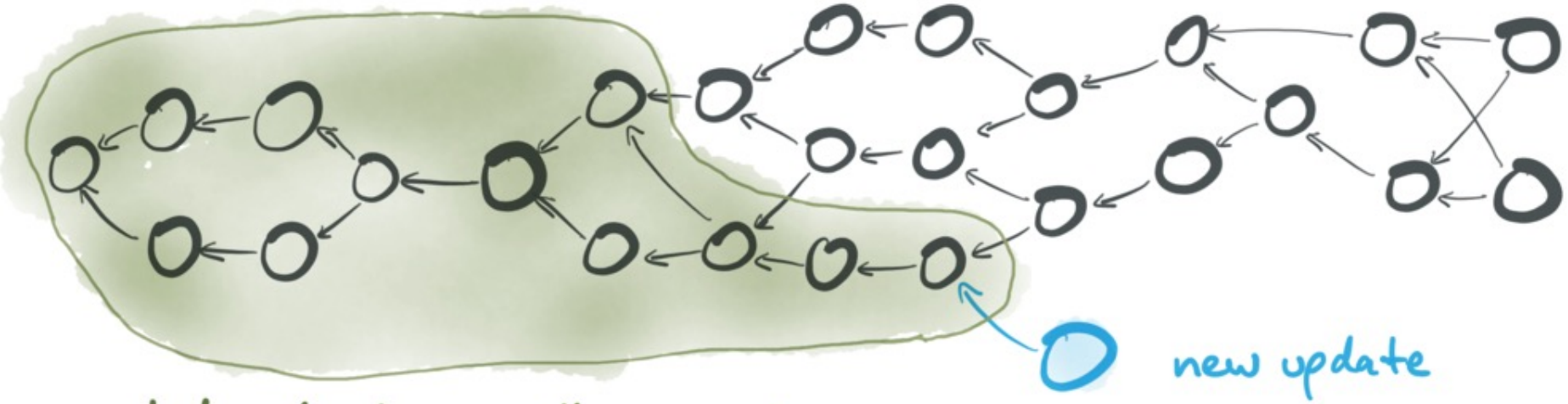
ENSURING CONVERGENCE

all of the updates previously delivered at this node



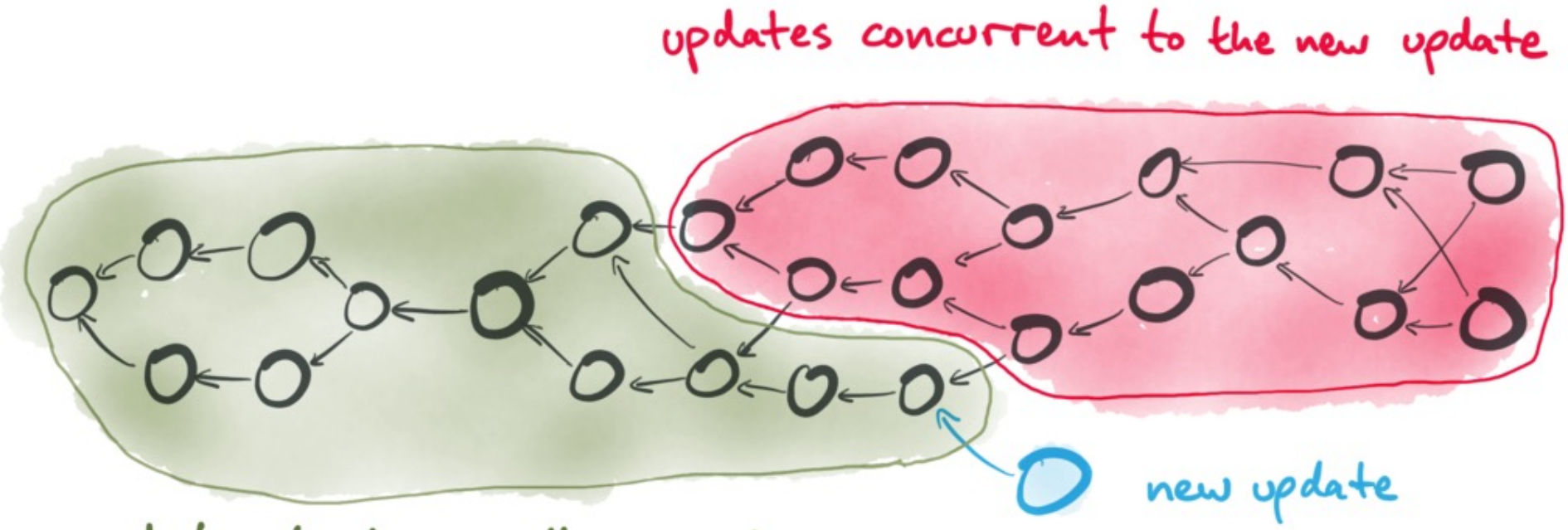
ENSURING CONVERGENCE

all of the updates previously delivered at this node

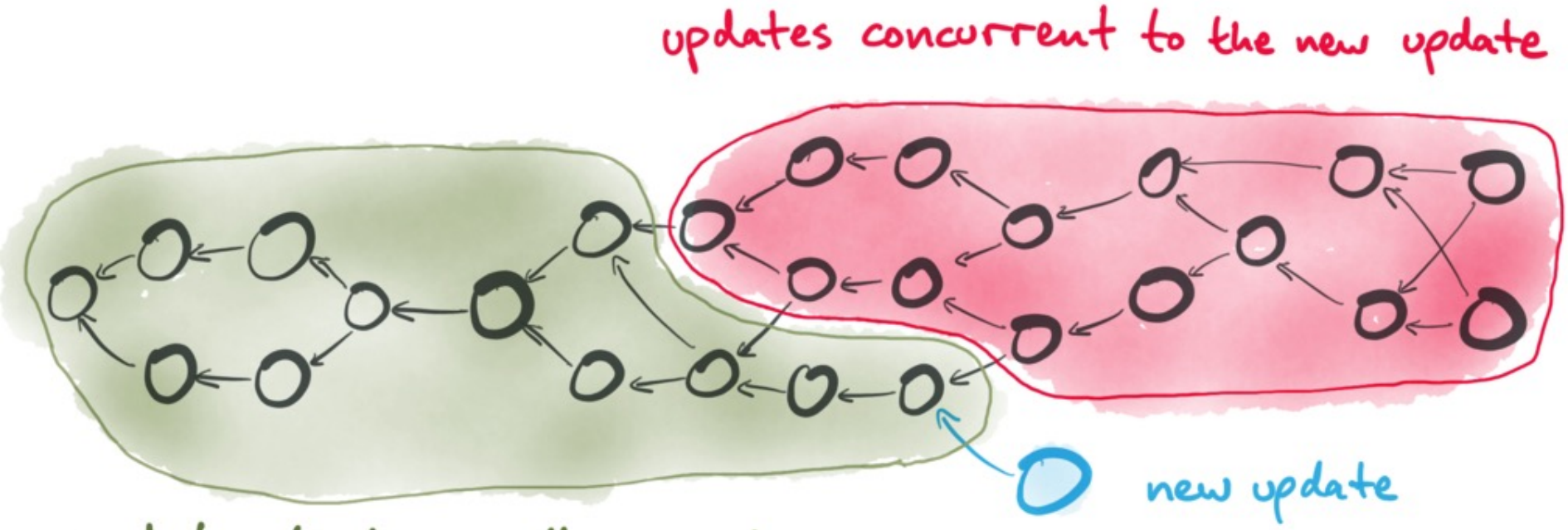


updates that causally precede the new update

ENSURING CONVERGENCE



ENSURING CONVERGENCE



The set of causally preceding updates will always be equal on any two correct nodes; the set of concurrent updates may differ.

ENSURING CONVERGENCE

Checking whether an update is valid, ignore it if not.

All correct nodes must make same accept/reject decision!

ENSURING CONVERGENCE

Checking whether an update is valid, ignore it if not.

All correct nodes must make same accept/reject decision!

valid (u, state) → {true, false}

Use the state resulting from applying only updates that causally precede u, ignoring updates concurrent to u!

e.g. u updates object x:

- u is valid if creation of x causally precedes u
- u is ignored if x does not exist or if creation of x is concurrent to u

STRONG EVENTUAL CONSISTENCY IN BYZANTINE SYSTEMS

1. Eventual delivery:

Arrange updates into hash graph, exchange hashes to sync between nodes

2. Convergence:

(a) Use hash of update as unique ID in CRDT

(b) Determine validity based on causally preceding updates

STRONG EVENTUAL CONSISTENCY IN BYZANTINE SYSTEMS

1. Eventual delivery:

Arrange updates into hash graph, exchange hashes to sync between nodes

no consensus
no quorums
no 3f+1
no proof-of-work

2. Convergence:

(a) Use hash of update as unique ID in CRDT

(b) Determine validity based on causally preceding updates

Challenges with this approach

- Hash-based unique IDs are big and hard to compress
- How do we make validity checks efficient?

Challenges with this approach

- Hash-based unique IDs are big and hard to compress
- How do we make validity checks efficient?
- Full editing history is big and exposes deleted data
  (e.g. accidentally pasted password or other sensitive data)

  ⇒ Solution: authenticated snapshots?

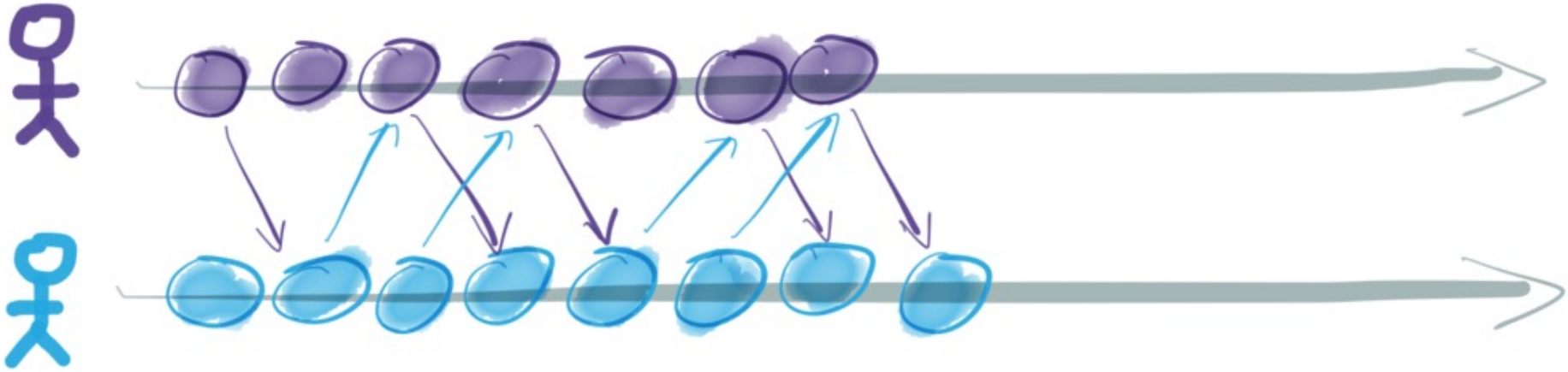
TODAY'S TOPICS.

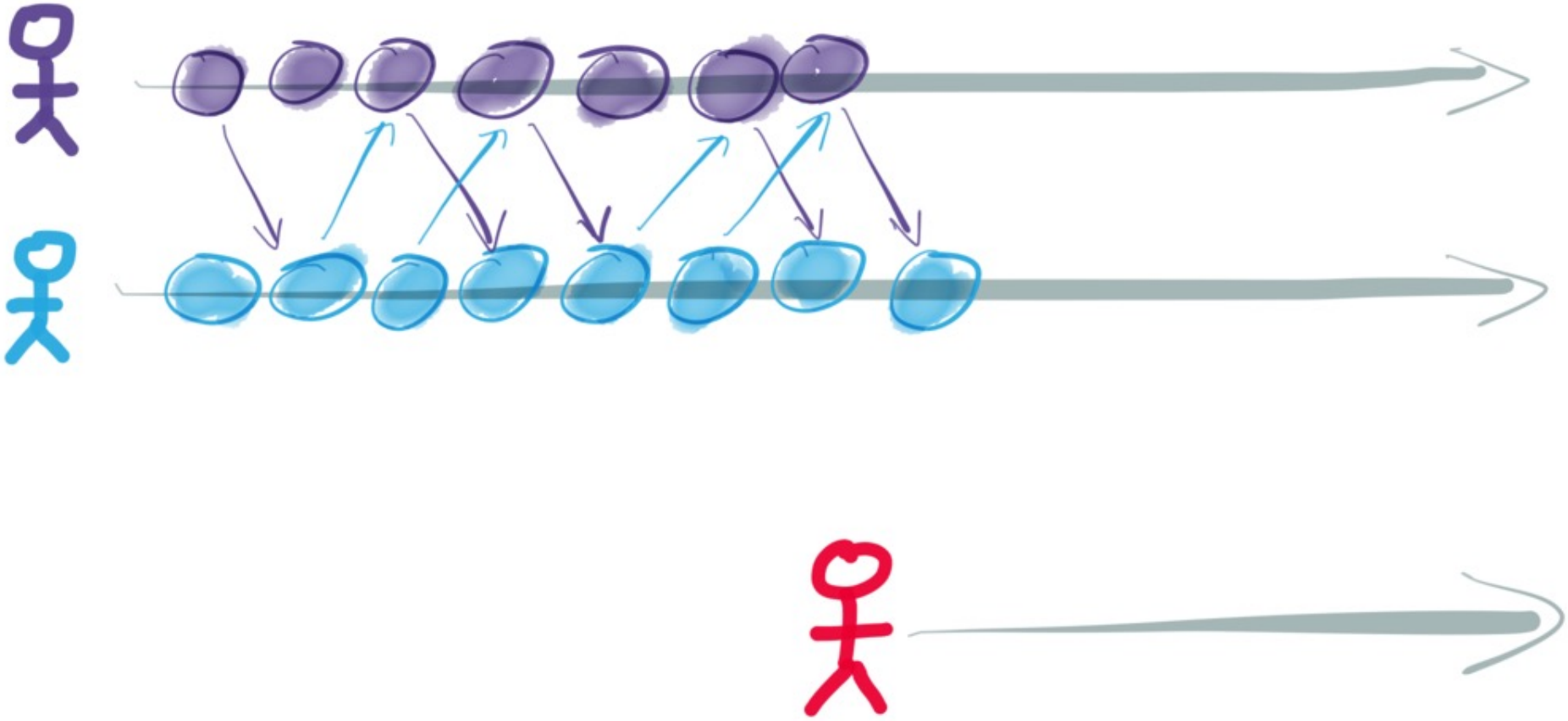
Integrity in the presence of Byzantine nodes

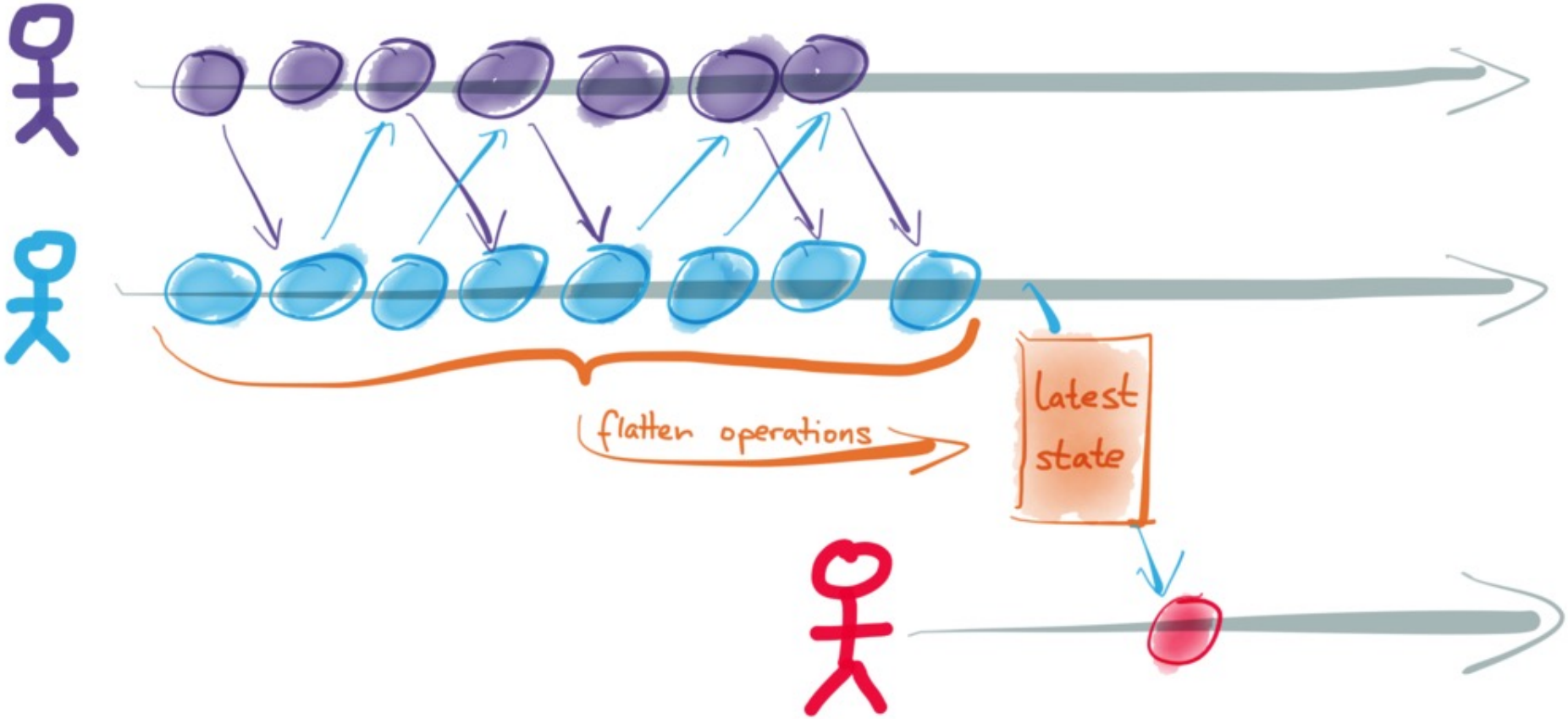
1. Byzantine eventual consistency + invariants
2. Byzantine fault tolerant CRDTs
3. Authenticated snapshots

Confidentiality against Byzantine nodes

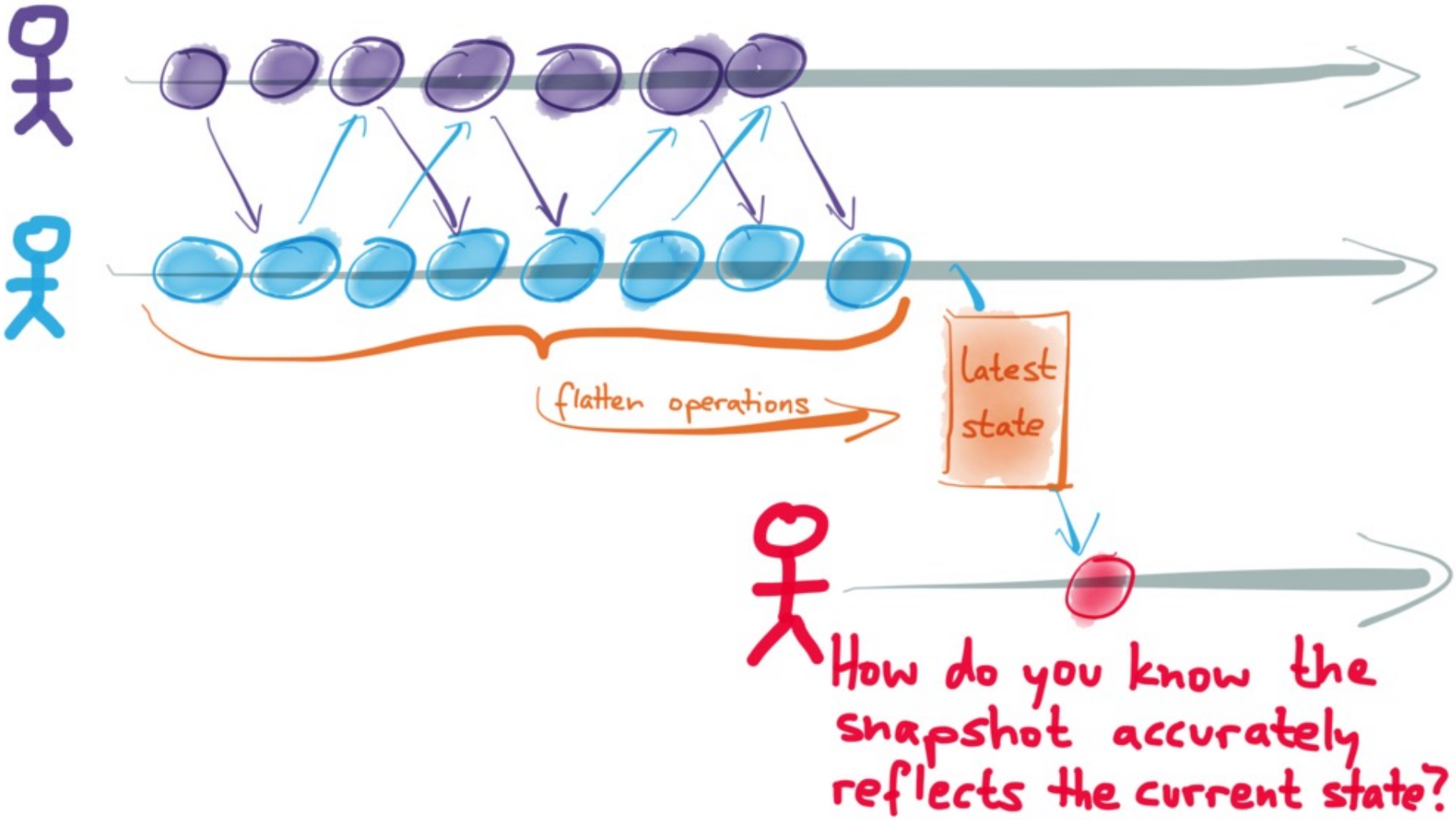
4. Decentralised access control list
5. End-to-end encryption





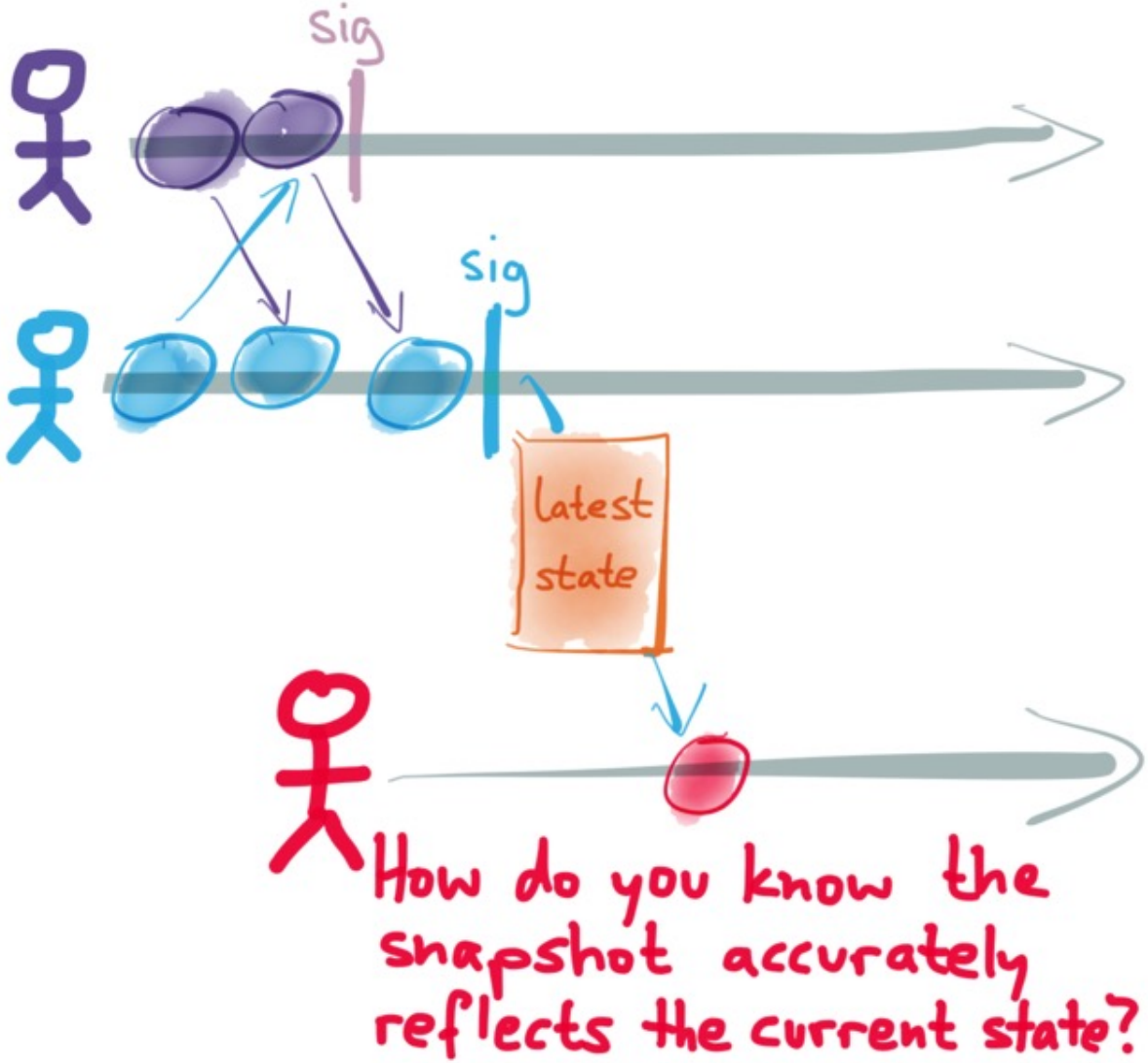

flatten operations
latest
state

AUTHENTICATED SNAPSHOTS



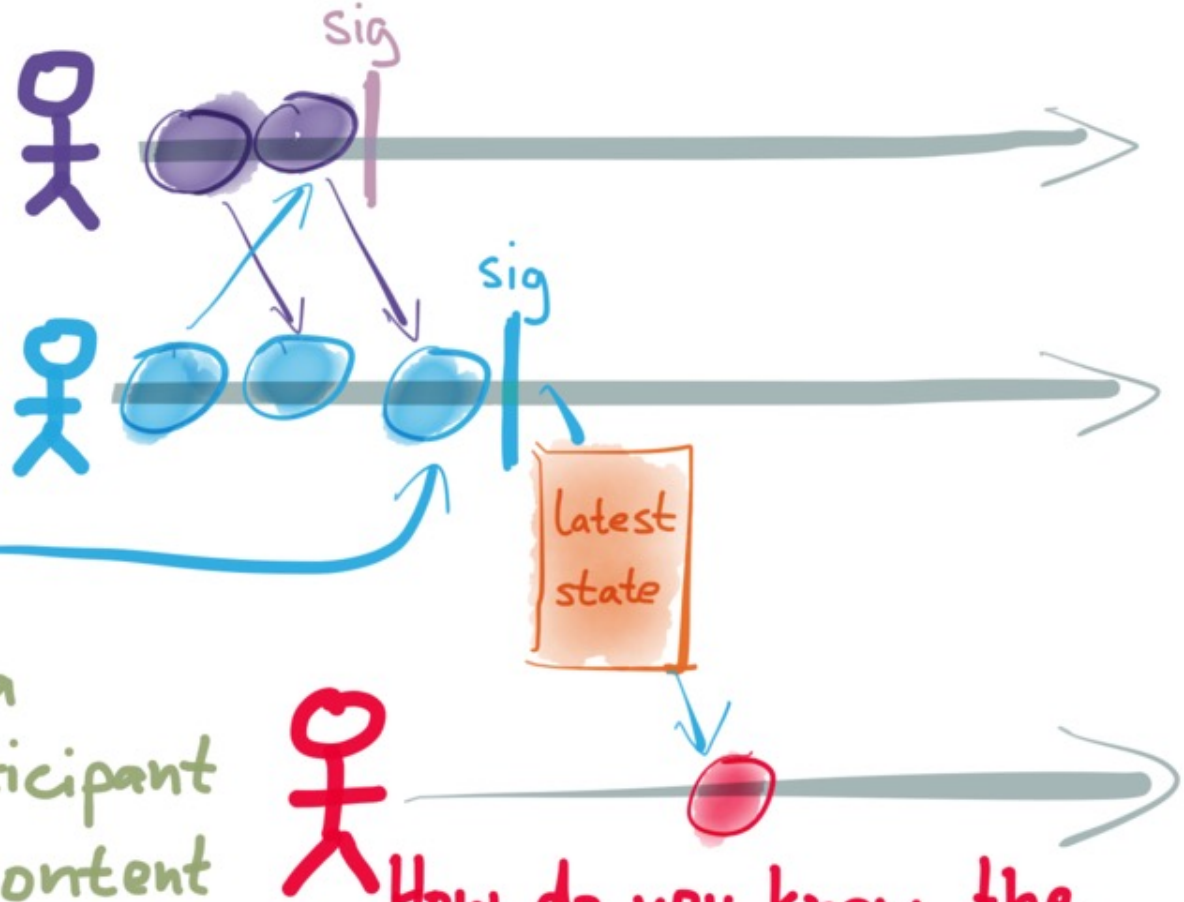
AUTHENTICATED SNAPSHOTS

Signatures?



AUTHENTICATED SNAPSHOTS

Signatures?



But if this is a deletion, new participant can see deleted content

How do you know the snapshot accurately reflects the current state?

sciendo

Proceedings on Privacy Enhancing Technologies 2019; 2019 (3):1–23

Stephan A. Kollmann*, Martin Kleppmann, and Alastair R. Beresford

Snapdoc: Authenticated snapshots with history privacy in peer-to-peer collaborative editing

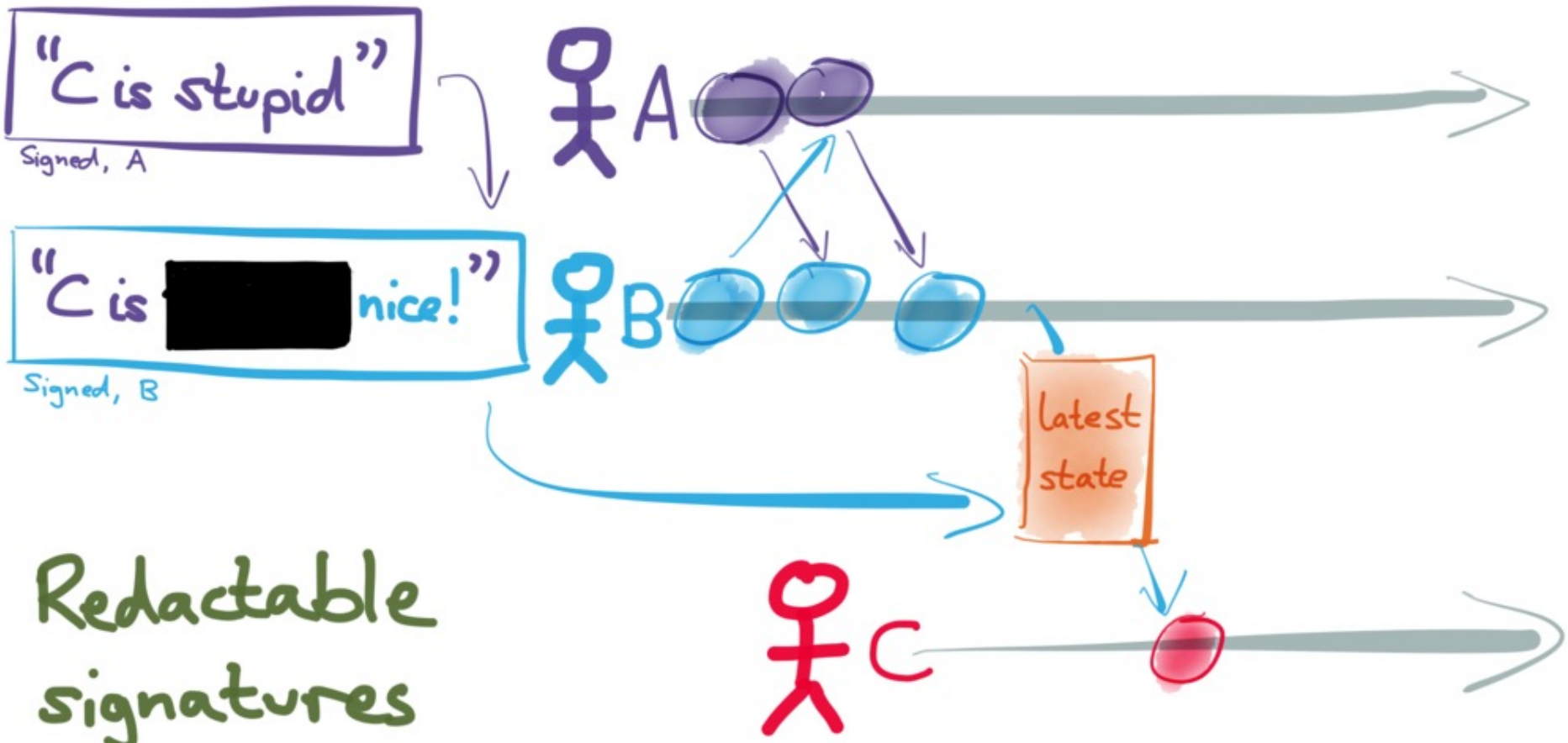
Abstract: Document collaboration applications, such as Google Docs or Microsoft Office Online, need to ensure that all collaborators have a consistent view of the shared document, and usually achieve this by relying on a trusted server. Other existing approaches that do not rely on a trusted third party assume that all collaborating devices are trusted. In particular, when inviting a new collaborator to a group, one needs to choose between a) keeping past edits private and sending only the latest state (a snapshot) of the document; or b) allowing the new collaborator to verify her view of the document is consistent with other honest devices by sending the full history of (signed) edits. We present a new protocol which allows an authenticated snapshot to be sent to new collaborators while both hiding the past editing history, and allowing them to verify consistency. We evaluate the costs of the protocol by emulating the editing history of 270 Wikipedia pages; 99% of insert operations were processed within 11.0 ms; 64.9 ms for delete operations. An additional benefit of authenticated snapshots is a median 84% reduction in the amount of data sent to a new collaborator compared to a basic protocol that transfers a full edit history.

copy of the document on every collaborating user's device. Whenever the user makes an edit on a device, these changes are first applied to the local copy of the document, and then an edit message containing the change is sent to a central server. When the server receives document edit messages, it modifies these messages to assist devices in resolving potential conflicting concurrent edits, and propagates these updated messages to the devices used by all collaborating users [8, 26].

While the communication between user devices and the server can be encrypted in such systems, e.g. using TLS, the server itself must be able to read and modify messages, and therefore it must be trusted to maintain the confidentiality and integrity of a cleartext copy of the document. For this reason, the current generation of collaborative document editing systems do not support end-to-end encryption between user devices.

Such systems require that users fully trust the service provider and its employees; this assumption is problematic in countries where service providers are obliged to hand over data to law enforcement and intelligence agencies without appropriate judicial oversight. In addition, a data breach at the service provider will reveal the

AUTHENTICATED SNAPSHOTS



Redactable signatures

RSA accumulator

$$acc(S) = x^{P(S)} \mod N \qquad (x, N \text{ public})$$

$$\text{where } P(S) = \prod_{a \in S} prime(a) \qquad (N = pq \text{ private})$$

RSA accumulator

$$acc(S) = x^{P(S)} \mod N \qquad (x, N \text{ public})$$

$$\text{where } P(S) = \prod_{a \in S} \text{prime}(a) \qquad (N = pq \text{ private})$$

Sending snapshot T to new collaborator:
prove that $T \subseteq S$ by sending $x^{P(S-T)} \mod N$
along with T and $acc(S)$.

New collaborator checks $\left(x^{P(S-T)}\right)^{P(T)} \stackrel{?}{=} acc(S)$.

TODAY'S TOPICS.

Integrity in the presence of Byzantine nodes

1. Byzantine eventual consistency + invariants
2. Byzantine fault tolerant CRDTs
3. Authenticated snapshots

Confidentiality against Byzantine nodes

4. Decentralised access control list
5. End-to-end encryption

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 Bieniusa and Herb Caudill

work in progress

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**

may sometimes fall into the hands of a malicious adversary. When this happens, the remaining users must be able to revoke the compromised credentials' permissions, so as to limit the damage that the adversary can do.

The problem is: once the secret keys of an authorised user are in the hands of an adversary, the adversary may perform arbitrary actions pretending to be that user. For example, if Bob's keys were compromised, and Alice (another authorised user) tries to revoke the permissions associated with Bob's key, the adversary may try to first revoke Alice's permissions and thus prevent her from removing the adversary's access. Alternatively, the adversary may use Bob's key to add several new devices that are also controlled by the adversary; thus, even if Bob's key is revoked, the adversary may be able to continue accessing the system through one of these other devices, until they are also removed.

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

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

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

public (identifies user) → pk_A ; private → sk_A

- 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(operationsReceived) at each device

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, ...

$op_1 = (\text{create}, pk_A)$

$\text{Sign}(op_1, sk_A)$

Set of members:

$\{pk_A\}$

Users/devices: Alice, Bob, Carol, Dave, ...

Public keys: pk_A, pk_B, pk_C, pk_D, ...

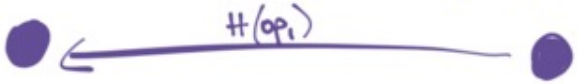
Private keys: sk_A, sk_B, sk_C, sk_D, ...

$op_1 = (\text{create}, pk_A)$

$\text{Sign}(op_1, sk_A)$

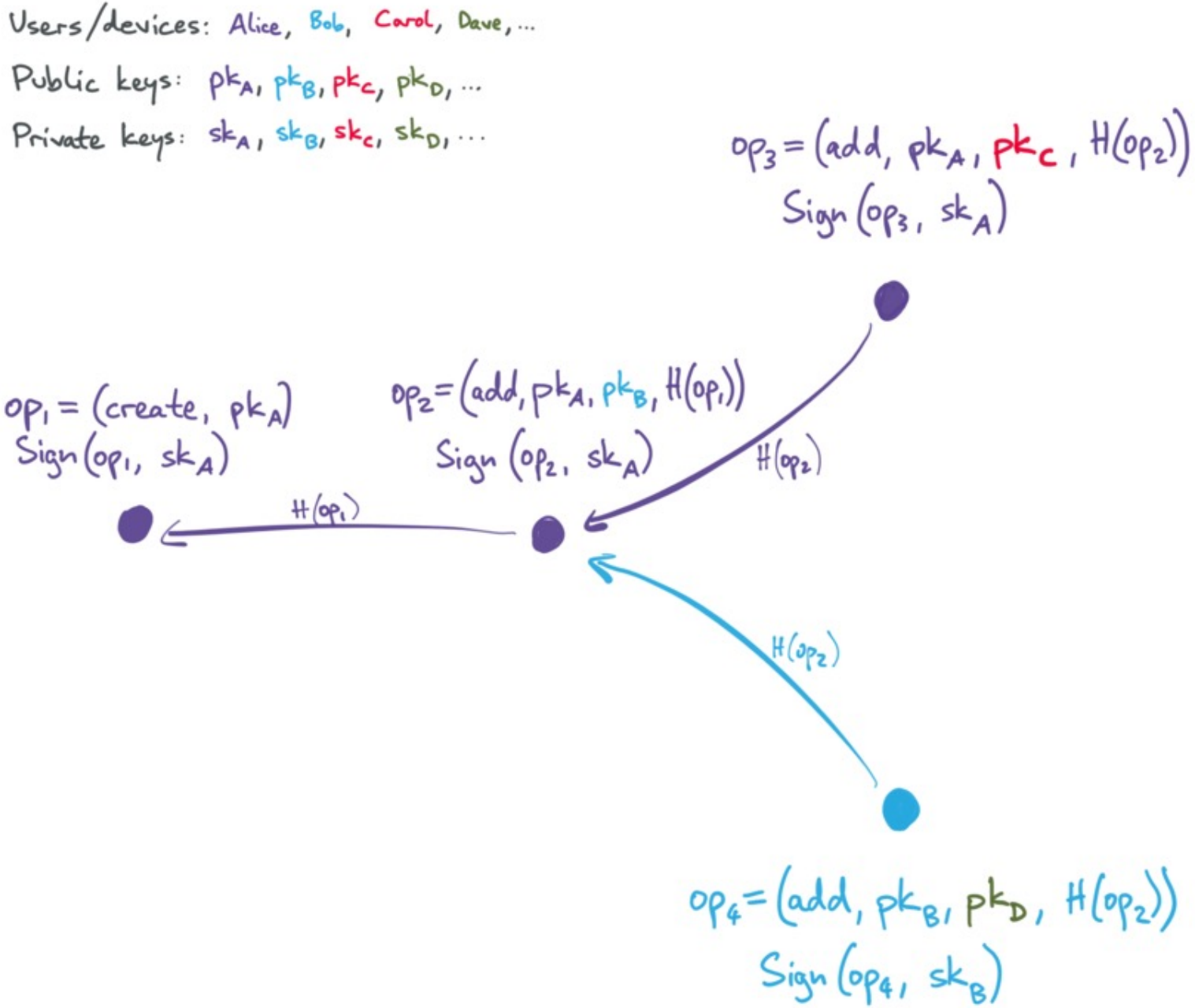
$op_2 = (\text{add}, pk_A, pk_B, H(op_1))$

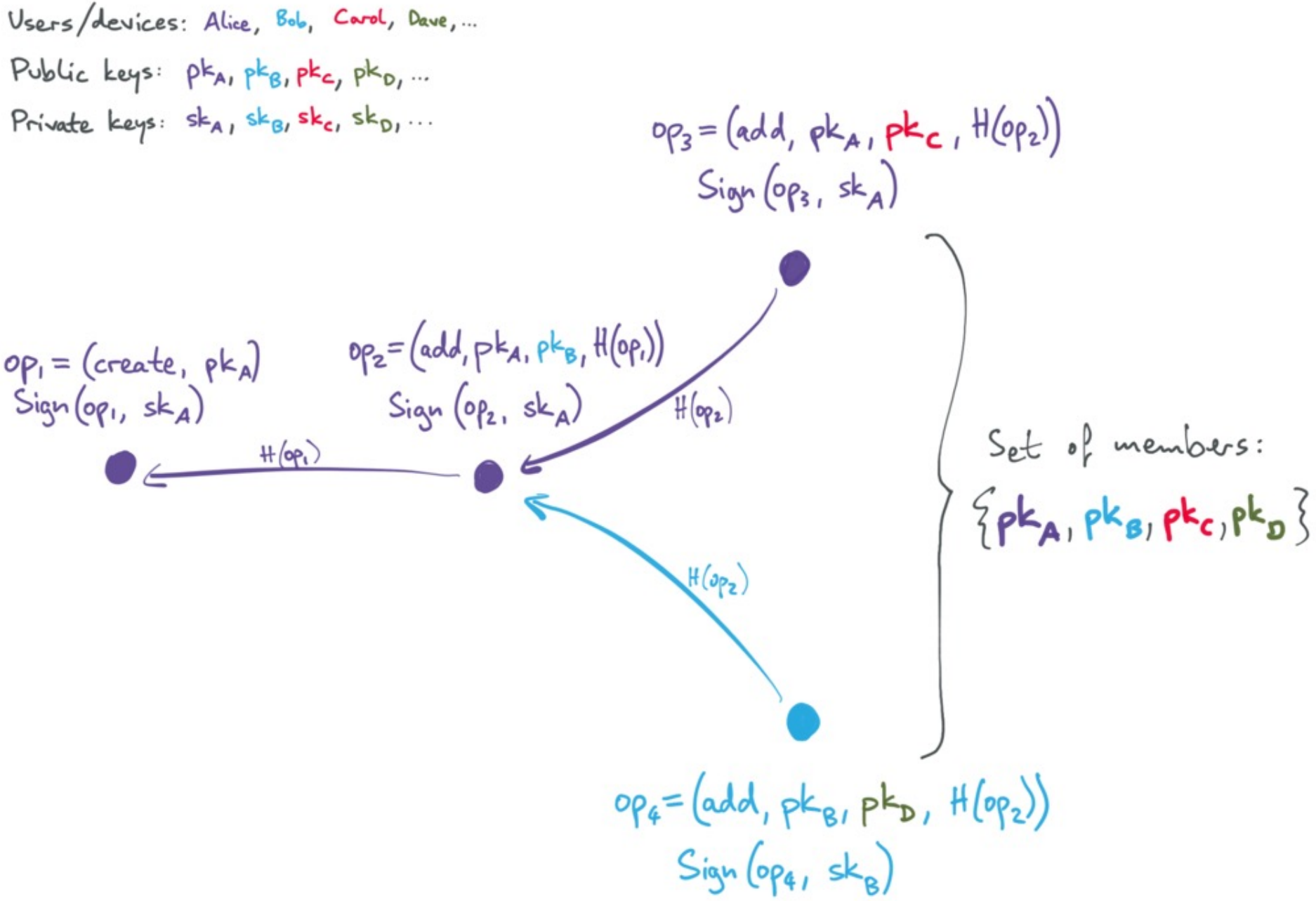
$\text{Sign}(op_2, sk_A)$

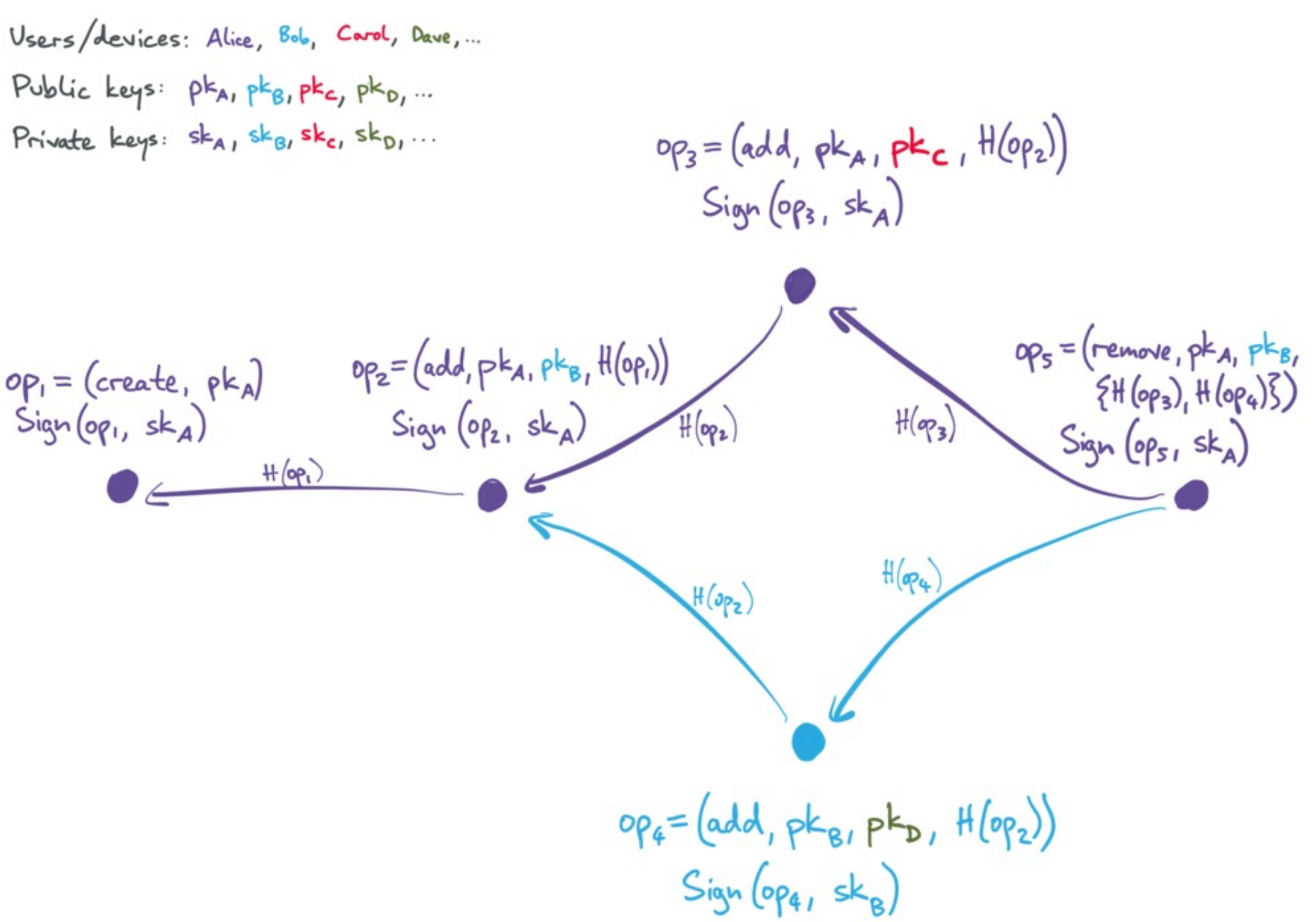


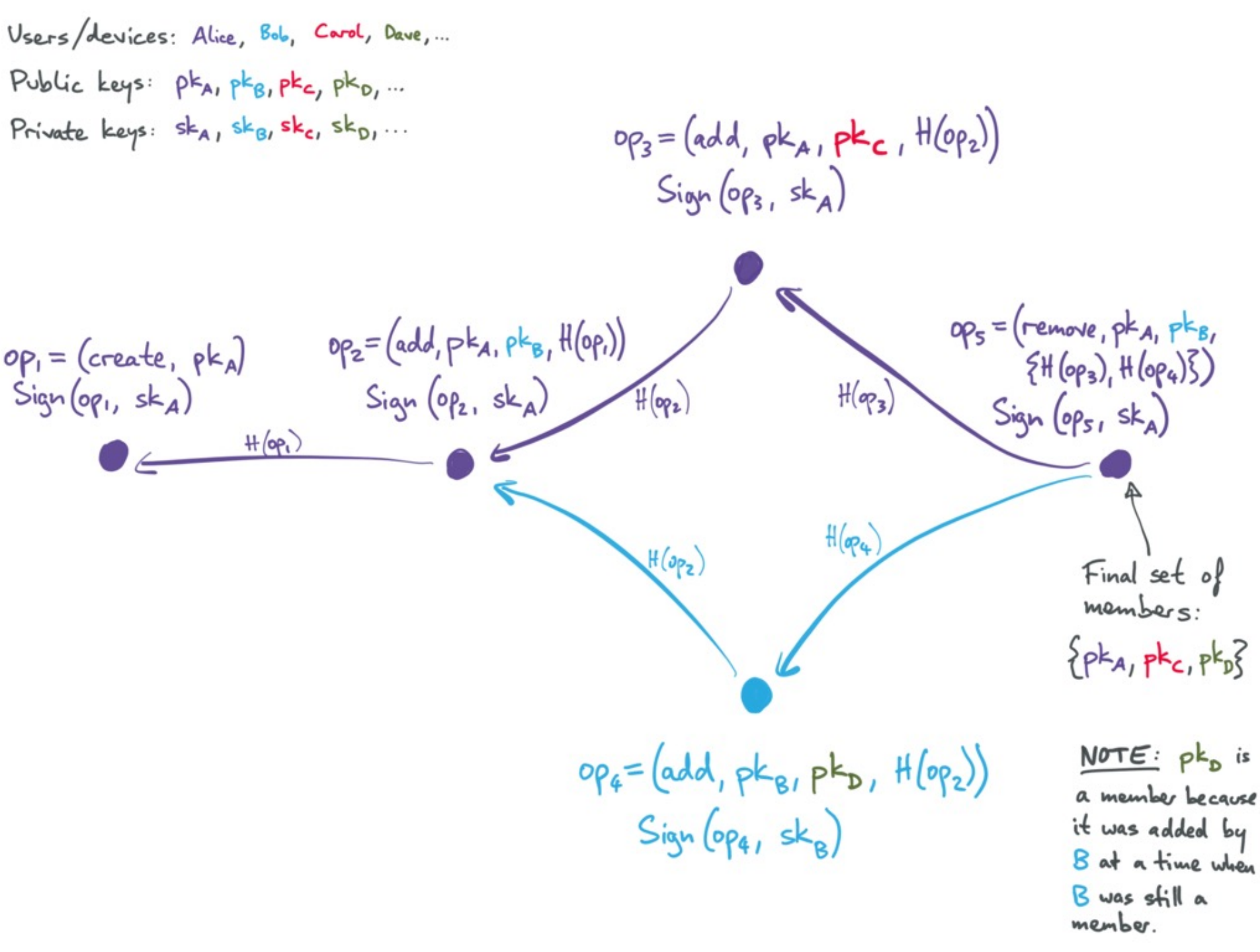
Set of members:

$\{pk_A, pk_B\}$


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, ...
$op_3 = (add, pk_A, pk_C, H(op_2))$
$Sign(op_3, sk_A)$
$op_1 = (create, pk_A)$
$Sign(op_1, sk_A)$
$op_2 = (add, pk_A, pk_B, H(op_1))$
$Sign(op_2, sk_A)$
$H(op_2)$
$H(op_1)$
$H(op_2)$
$op_4 = (add, pk_B, pk_D, H(op_2))$
$Sign(op_4, sk_B)$


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, ...
$op_3 = (add, pk_A, pk_C, H(op_2))$
$Sign(op_3, sk_A)$
$op_1 = (create, pk_A)$
$Sign(op_1, sk_A)$
$op_2 = (add, pk_A, pk_B, H(op_1))$
$Sign(op_2, sk_A)$
$H(op_1)$
$H(op_2)$
$H(op_2)$
Set of members:
$\{pk_A, pk_B, pk_C, pk_D\}$
$op_4 = (add, pk_B, pk_D, H(op_2))$
$Sign(op_4, sk_B)$


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, ...
$op_3 = (add, pk_A, pk_C, H(op_2))$
$Sign(op_3, sk_A)$
$op_1 = (create, pk_A)$
$Sign(op_1, sk_A)$
$op_2 = (add, pk_A, pk_B, H(op_1))$
$Sign(op_2, sk_A)$
$op_5 = (remove, pk_A, pk_B, \{H(op_3), H(op_4)\})$
$Sign(op_5, sk_A)$
$H(op_1)$
$H(op_2)$
$H(op_3)$
$H(op_2)$
$H(op_4)$
$op_4 = (add, pk_B, pk_D, H(op_2))$
$Sign(op_4, sk_B)$

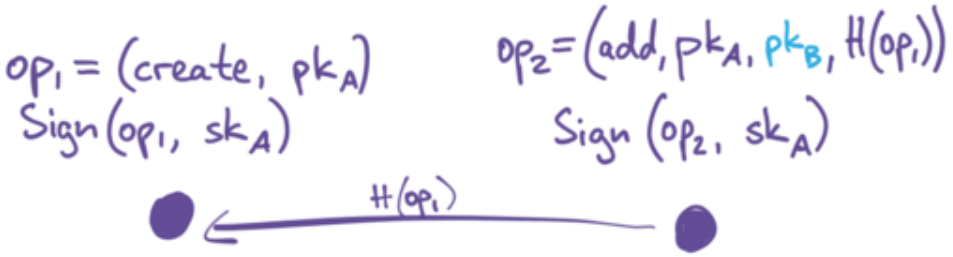

Users/devices: Alice, Bob, Carol, Dave, ...
Public keys: pk_A, pk_B, pk_C, pk_D, ...
Private keys: sk_A, sk_B, sk_C, sk_D, ...
$op_3 = (add, pk_A, pk_C, H(op_2))$
$Sign(op_3, sk_A)$
$op_1 = (create, pk_A)$
$Sign(op_1, sk_A)$
$op_2 = (add, pk_A, pk_B, H(op_1))$
$Sign(op_2, sk_A)$
$op_5 = (remove, pk_A, pk_B, \{H(op_3), H(op_4)\})$
$Sign(op_5, sk_A)$
$H(op_1)$
$H(op_2)$
$H(op_3)$
$H(op_2)$
$H(op_4)$
Final set of members:
$\{pk_A, pk_C, pk_D\}$
$op_4 = (add, pk_B, pk_D, H(op_2))$
$Sign(op_4, sk_B)$
NOTE: pk_D is a member because it was added by B at a time when B was still a member.

Access control / permissions for local-first software.

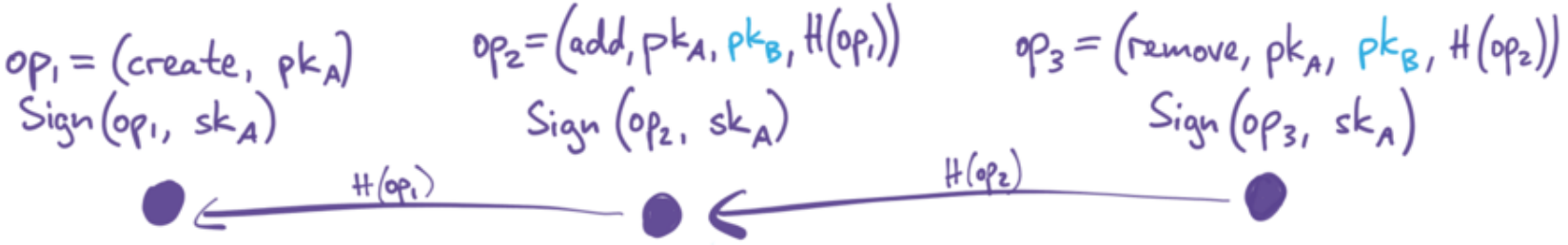
Problem: what if you want to remove the permissions from someone who doesn't want to be removed? (Byzantine behaviour)

e.g. adversary stole + unlocked a team member's phone

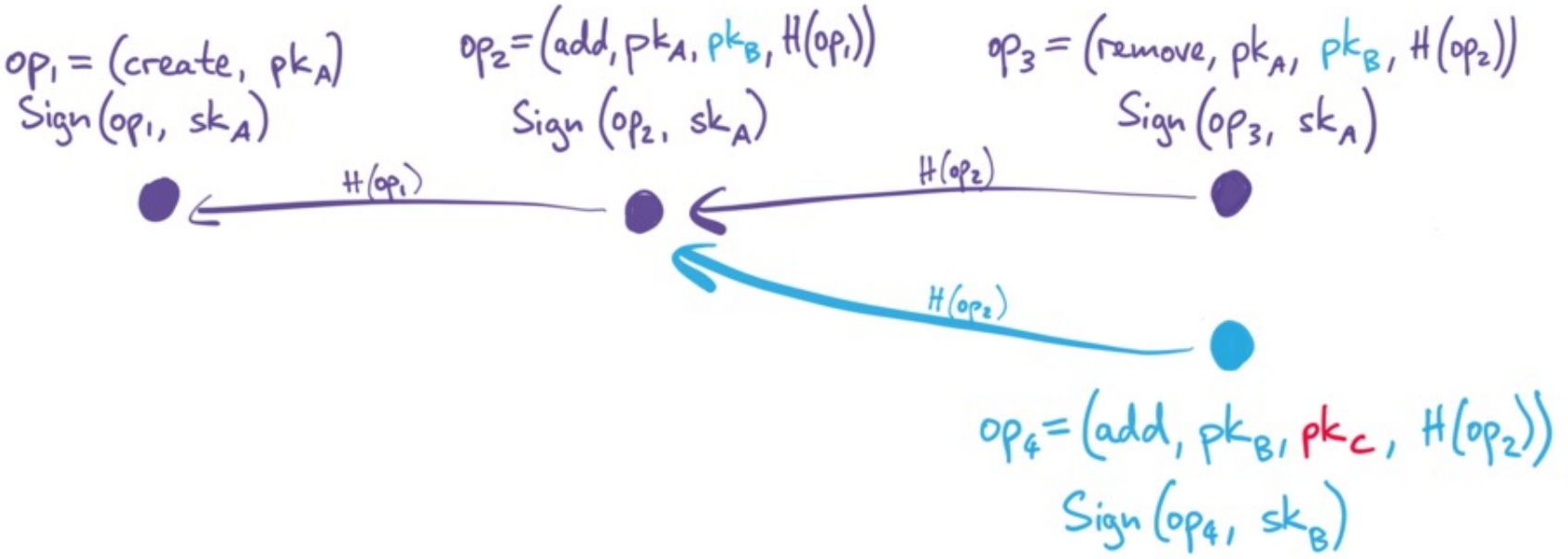
A removed user concurrently adds a new user...



A removed user concurrently adds a new user...

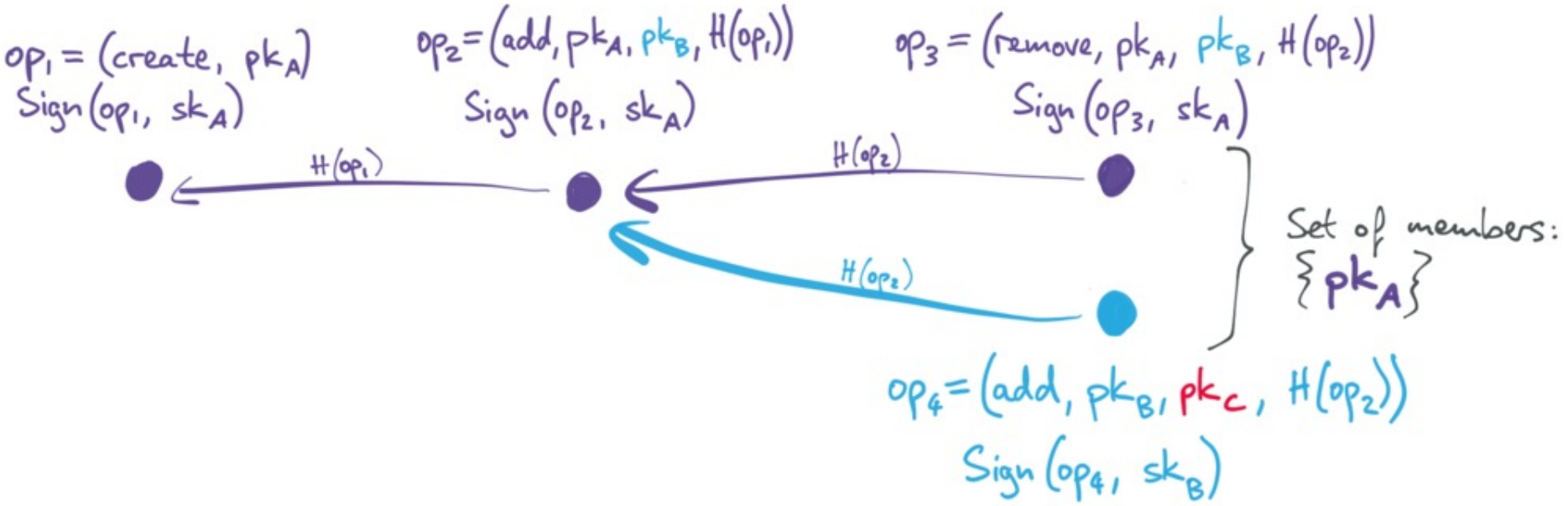


A removed user concurrently adds a new user...



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...



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

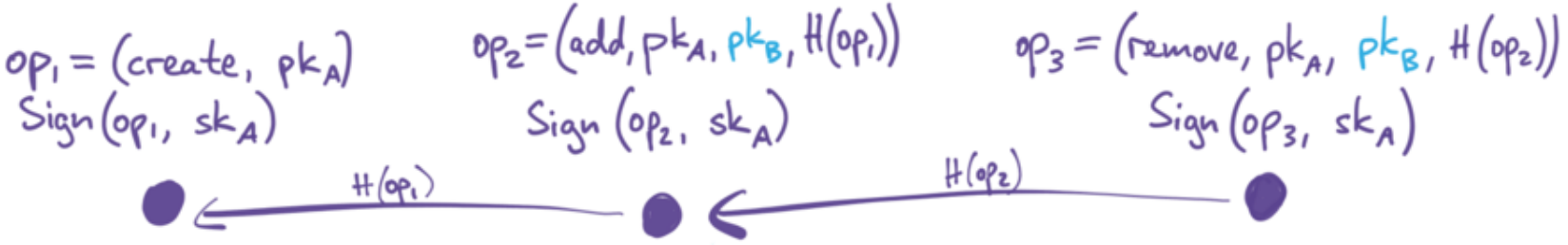
⇒ ignore all ops by B concurrent with removal of B

Two users concurrently remove each other...

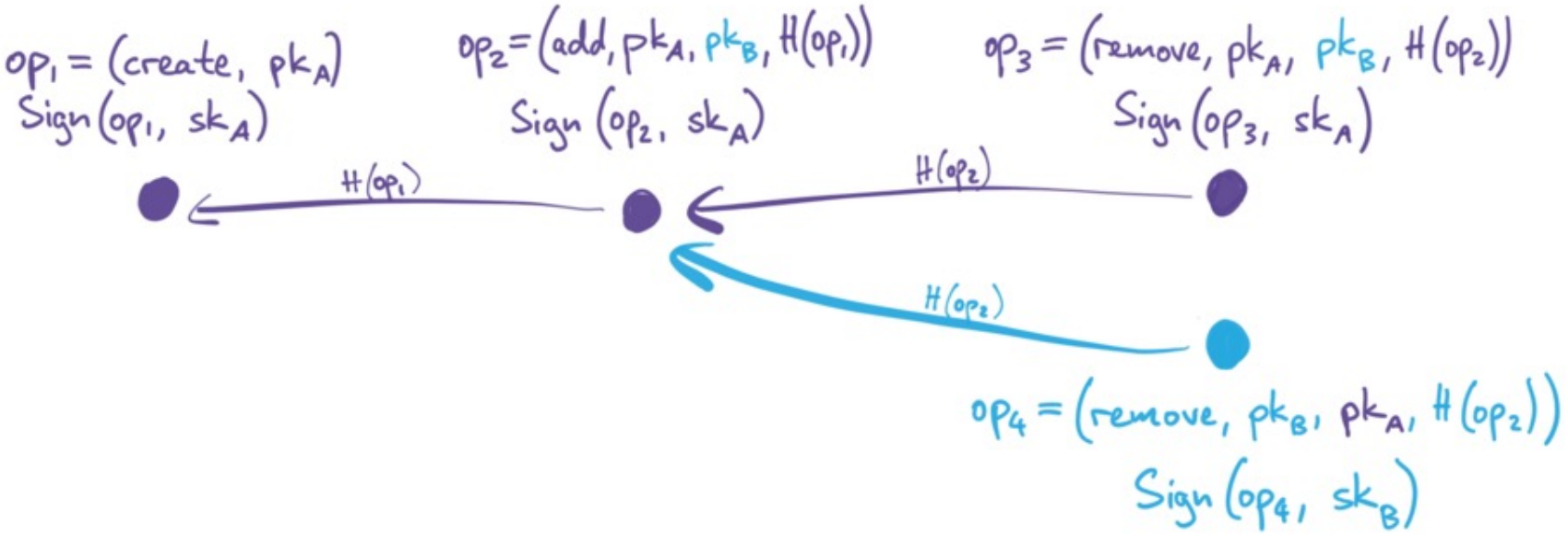
$op_1 = (\text{create}, pk_A)$
$\text{Sign}(op_1, sk_A)$

$op_2 = (\text{add}, pk_A, pk_B, H(op_1))$
$\text{Sign}(op_2, sk_A)$

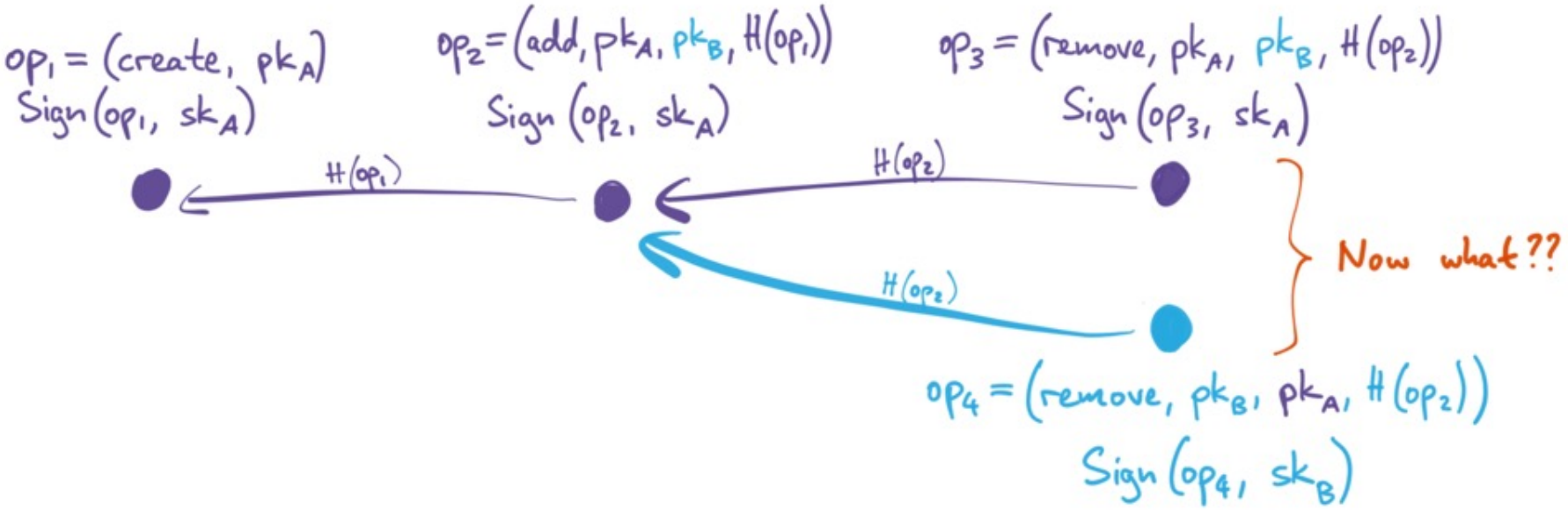
Two users concurrently remove each other...



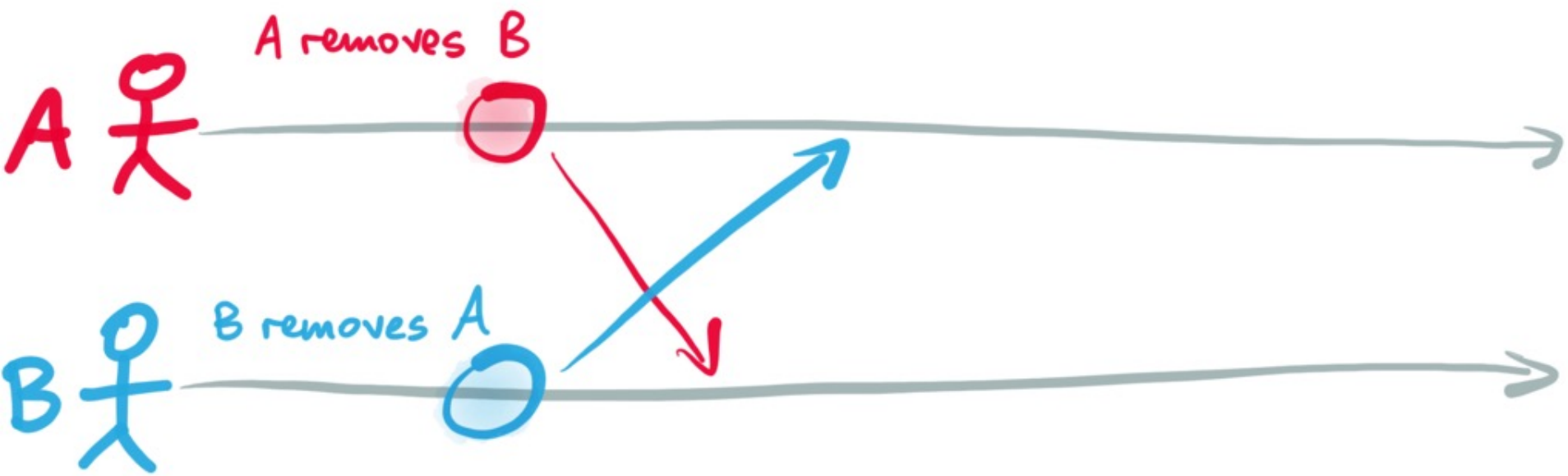
Two users concurrently remove each other...

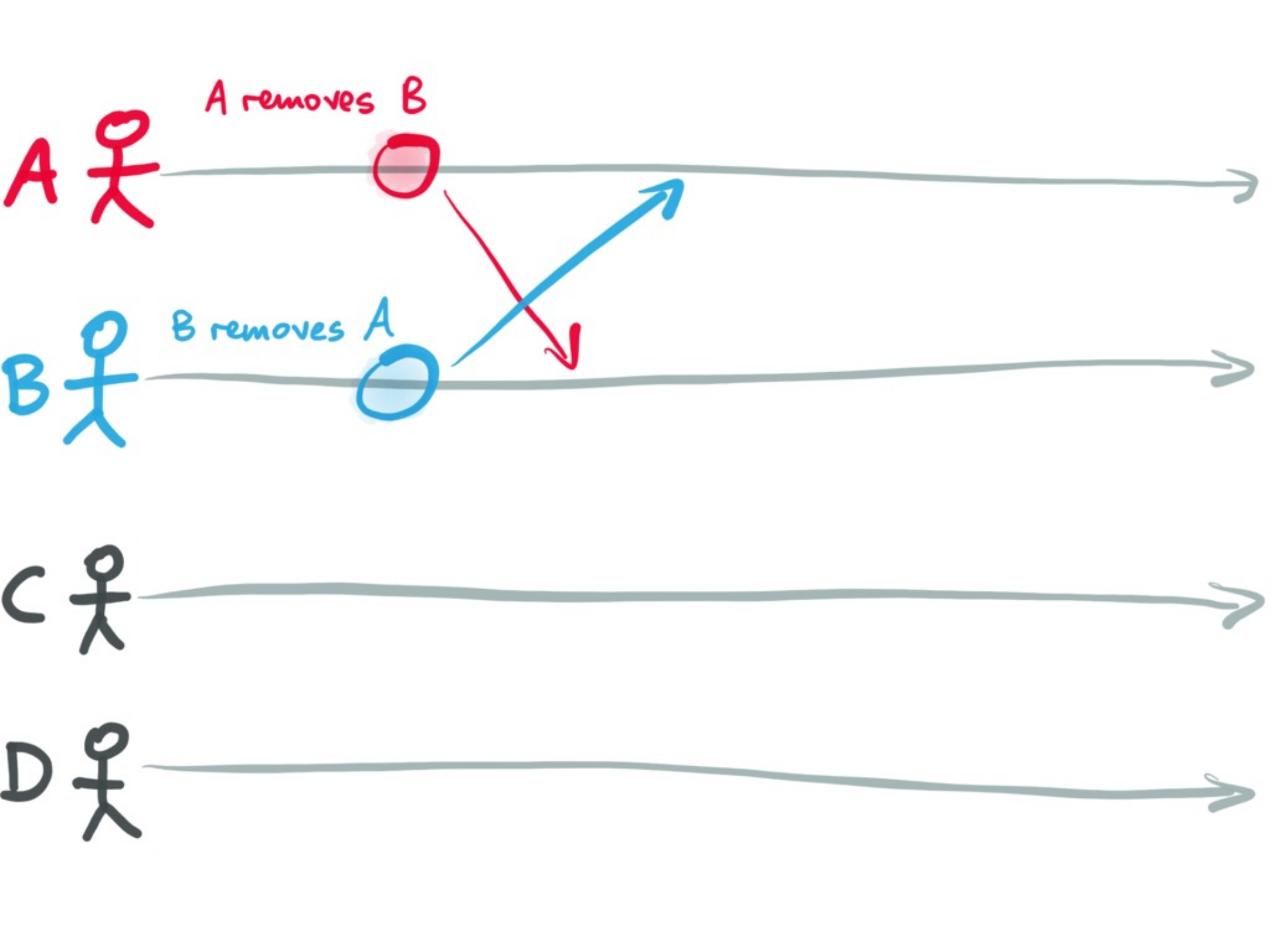


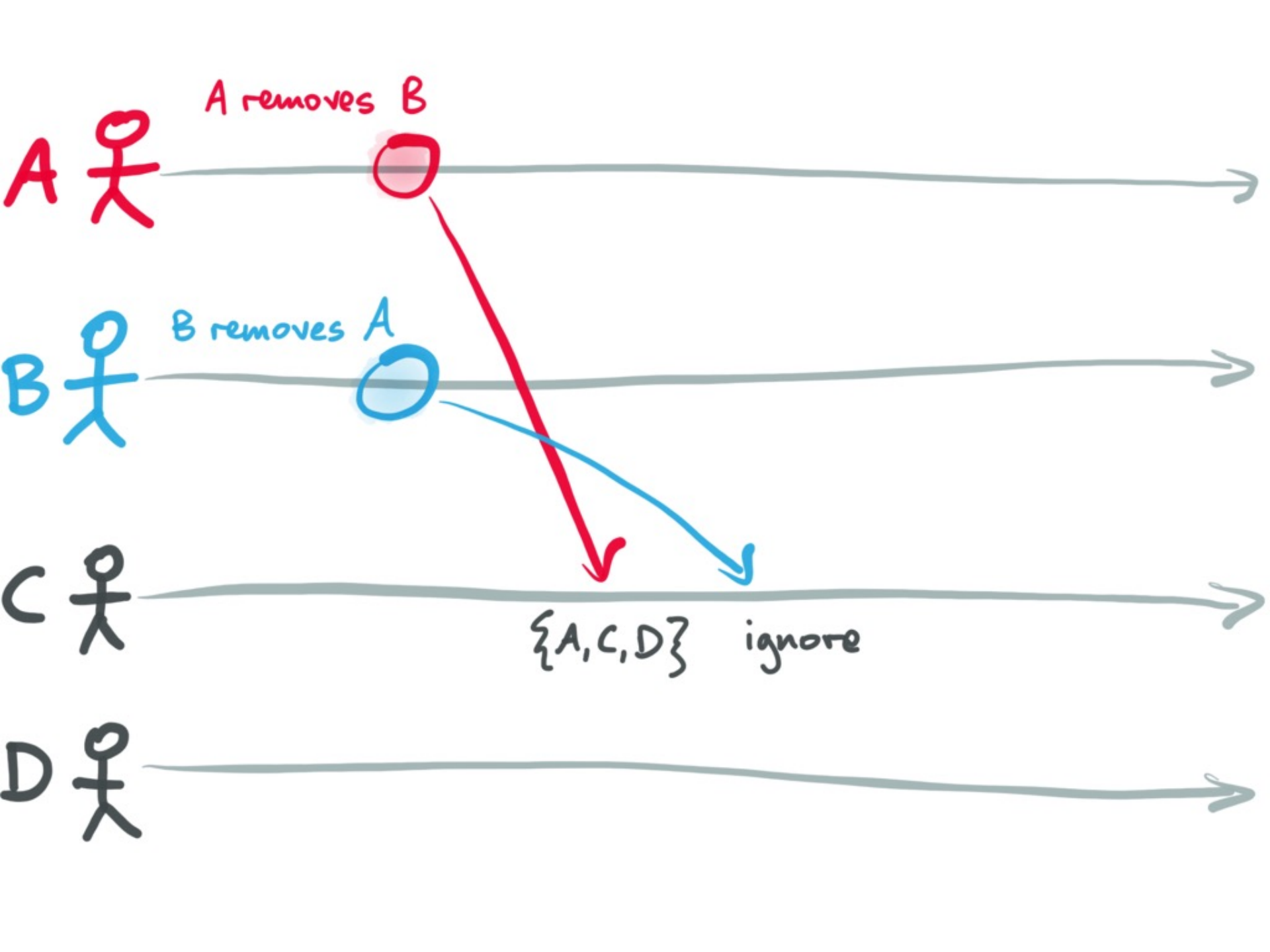
Two users concurrently remove each other...

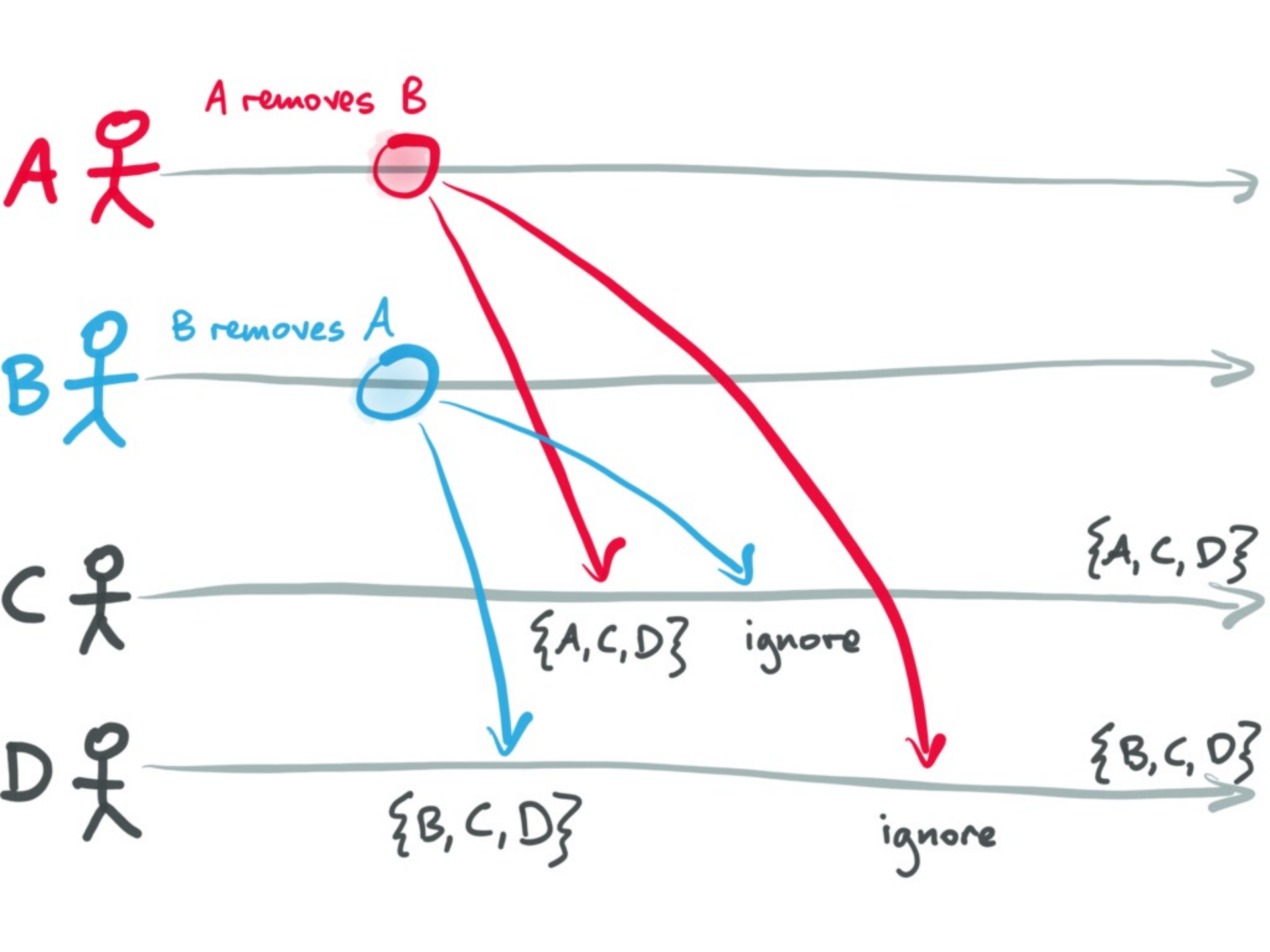



A
B


A removes B
A
B removes A
B


A removes B
A
B removes A
B
C
D


A removes B
A
B removes A
B
C
{A, C, D}
ignore
D


A removes B
A
B removes A
B
C
D
{A, C, D}
{A, C, D}
ignore
{B, C, D}
{B, C, D}
ignore

How to handle mutual revocation?

Operation timestamps?

How to handle mutual revocation?

~~Operation timestamps?~~ Adversarially chosen timestamps

Remove both?

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?

A

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?

A 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?

B 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

⟹ k out of n users can override seniority ranking

⟹ need scheme for redistributing secret shares after group membership changes

TODAY'S TOPICS.

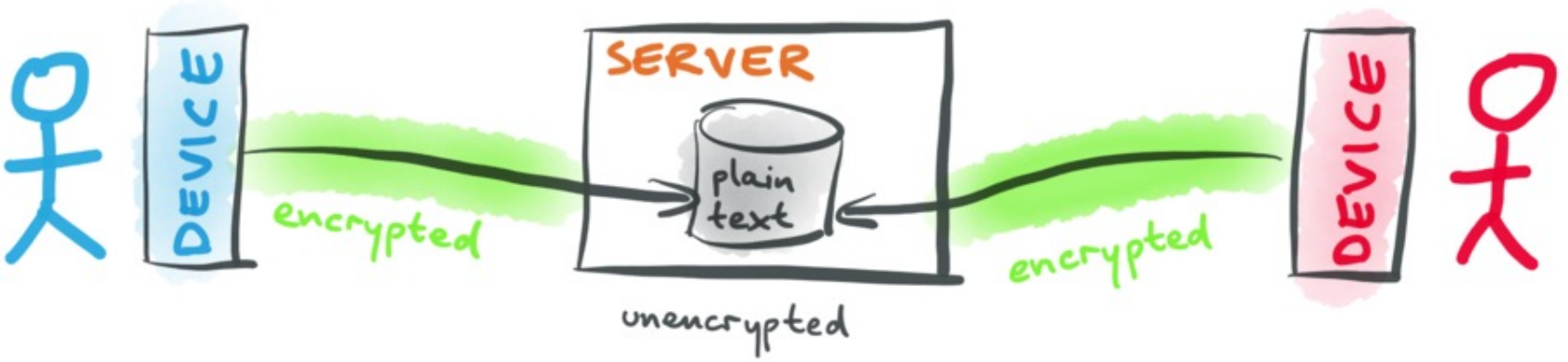
Integrity in the presence of Byzantine nodes

1. Byzantine eventual consistency + invariants
2. Byzantine fault tolerant CRDTs
3. Authenticated snapshots

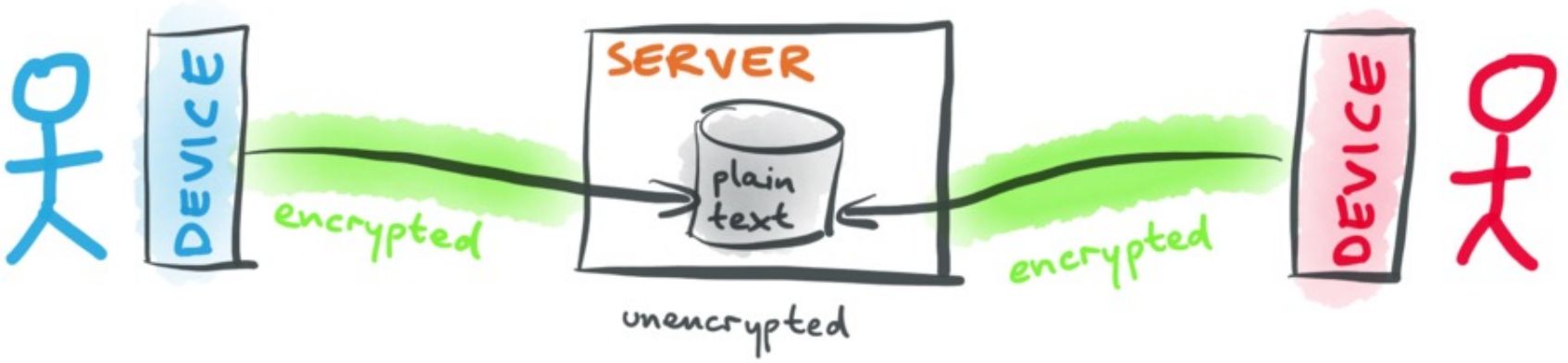
Confidentiality against Byzantine nodes

4. Decentralised access control list
5. End-to-end encryption

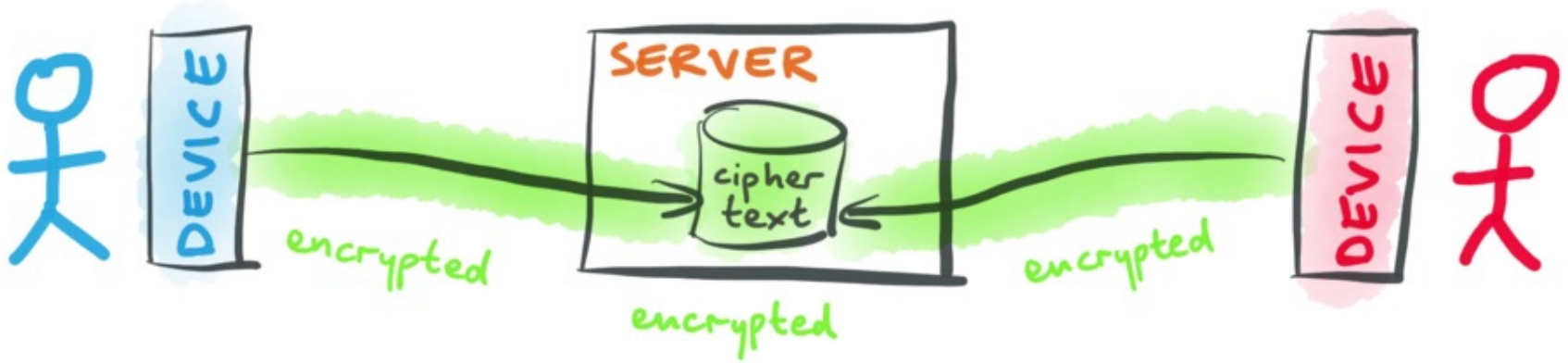
TLS/SSL


DEVICE
encrypted
SERVER
plain
text
unencrypted
encrypted
DEVICE

TLS/SSL



End-to-end encryption



We have end-to-end encryption for messaging:


WhatsApp


Signal


iMessage

+ various others

We have end-to-end encryption for messaging:


WhatsApp


Signal


iMessage

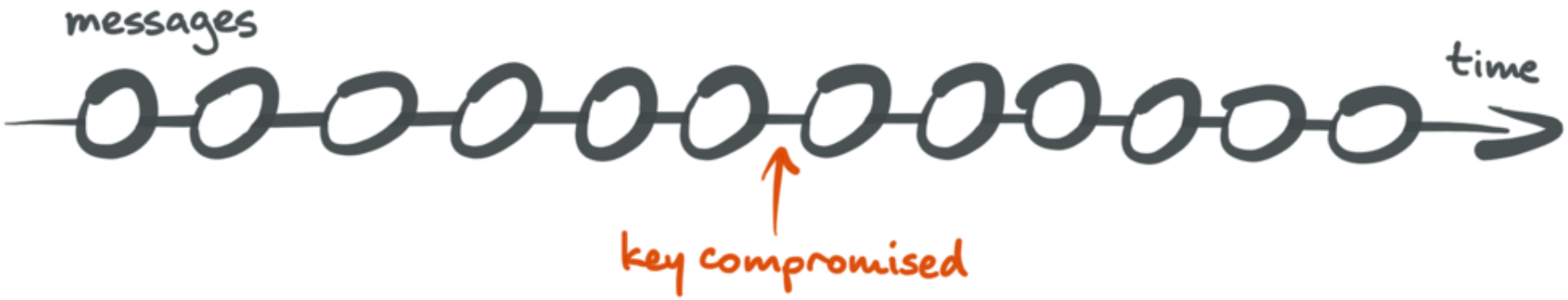
+ various others

Why not for other types of collaboration software?

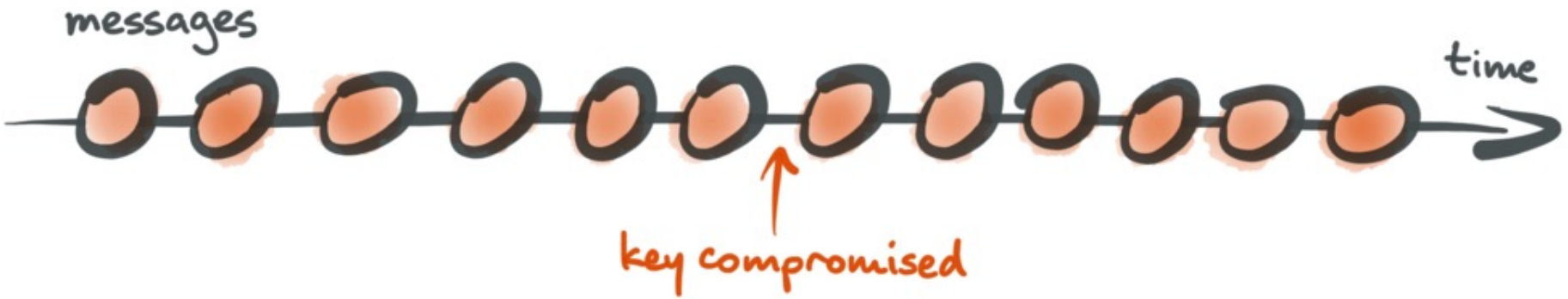

Google Docs

Also: wide range of domain-specific collaboration software, e.g. for investigative journalism, medical records, scientific data analysis, engineering/CAD, etc...

if all messages are encrypted with the same key...

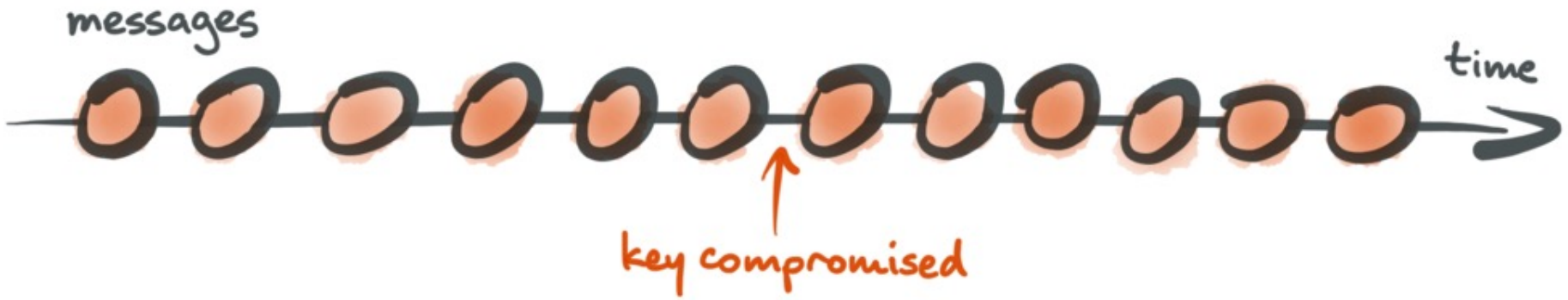


if all messages are encrypted with the same key...

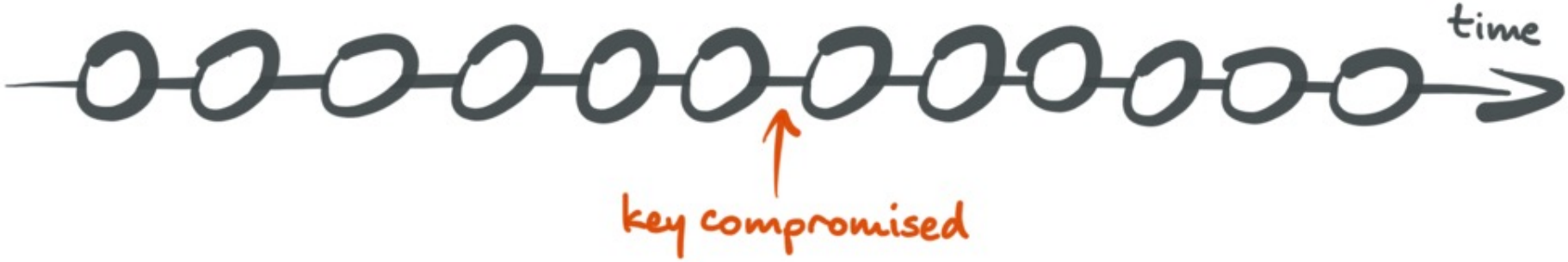


all messages vulnerable

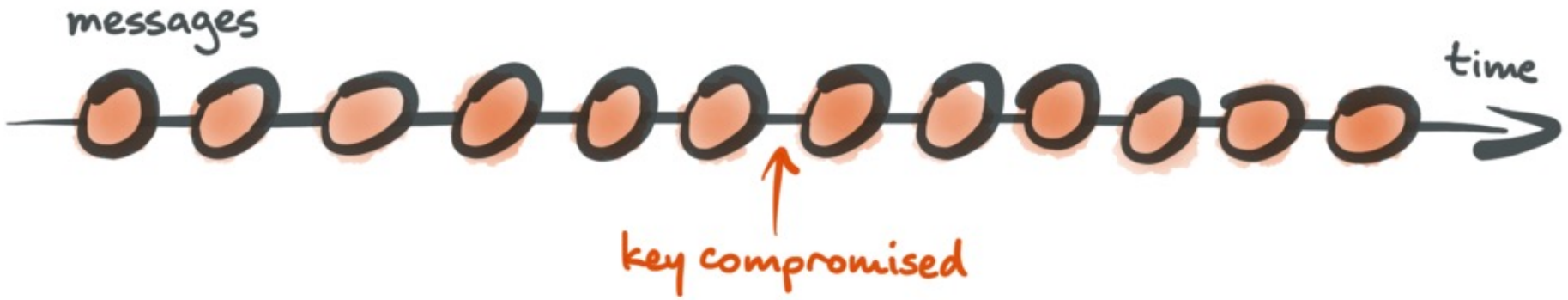
if all messages are encrypted with the same key...



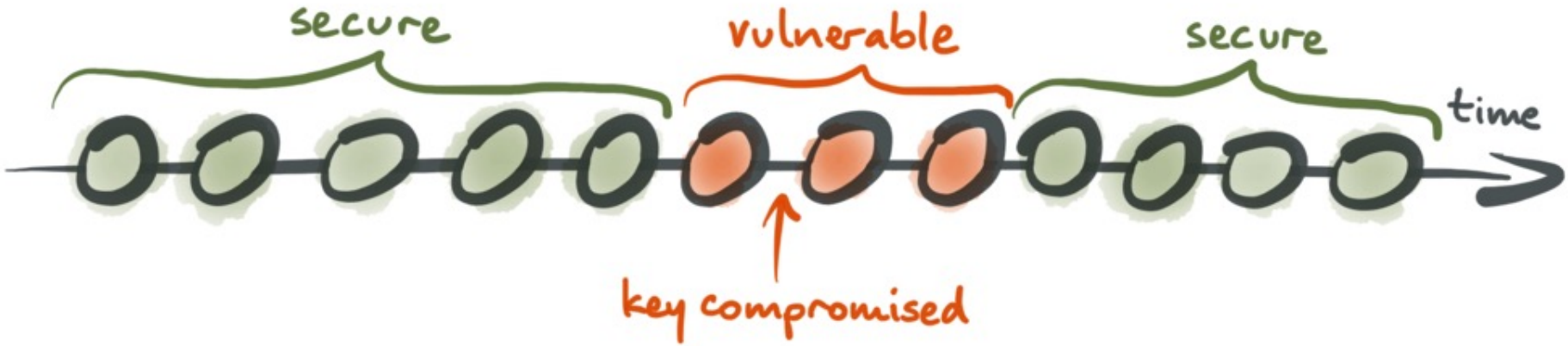
Forward secrecy and post-compromise security (PCS):



if all messages are encrypted with the same key...



Forward secrecy and post-compromise security (PCS):



TRICKY SET OF FEATURES REQUIRED

1. Growable + shrinkable group
2. Asynchronous (do not require users to be online)
3. Forward secrecy + PCS
4. Efficiency + scalability
5. Decentralised (compatible with mixnets/P2P)

Key Agreement for Decentralized Secure Group Messaging with Strong Security Guarantees

Matthew Weidner
maweidne@andrew.cmu.edu
Carnegie Mellon University
Pittsburgh, PA, USA

Martin Kleppmann
Daniel Hugenroth
Alastair R. Beresford
(mk428,dh623,arb33)@cst.cam.ac.uk
University of Cambridge
Cambridge, UK

ABSTRACT

Secure group messaging protocols, providing end-to-end encryption for group communication, need to handle mobile devices frequently being offline, group members being added or removed, and the possibility of device compromises during long-lived chat sessions. Existing work targets a centralized network model in which all messages are routed through a single server, which is trusted to provide a consistent total order on updates to the group state. In this paper we adapt secure group messaging for *decentralized* networks that have no central authority. Servers may still optionally be used, but they are trusted less. We define *decentralized continuous group key agreement* (DCGKA), a new cryptographic primitive encompassing the core of a decentralized secure group messaging protocol; we give a practical construction of a DCGKA protocol and prove its security; and we describe how to construct a full messaging protocol from DCGKA. In the face of device compromise our protocol achieves forward secrecy and post-compromise security. We evaluate the performance of a prototype implementation, and demonstrate that our protocol has practical efficiency.

CCS CONCEPTS

• **Security and privacy → Key management**; **Distributed systems security**.

1 INTRODUCTION

WhatsApp, Signal, and similar messaging apps have brought end-to-end encryption to billions of users globally, demonstrating that the benefits of such privacy-enhancing technologies can be enjoyed by users who are not technical experts. Modern secure messaging protocols used by these apps have several important characteristics:

Asynchronous: A user can send messages to other users regardless of whether the recipients are currently online. Offline recipients receive their messages when they are next online again (even if the sender is now offline). This property is important for mobile devices, which are frequently offline.

Resilient to device compromise: If a user's device is compromised, i.e., all of that device's secret key material is revealed to the adversary, the protocol nevertheless provides *forward secrecy* (FS): any messages received before the compromise cannot be decrypted by the adversary. Moreover, protocols can provide *post-compromise security* (PCS) [13]: users regularly update their keys so the adversary eventually loses the ability to decrypt further communication. As secure messaging sessions may last for years, these properties are important for limiting the impact of a compromise.

Dynamic: Group members can be added and removed at any time.

In the case when only two users are communicating, the Signal protocol [31] is widely used. However, generalizations of this two-

	Cost of group update (n = group size)	Forward secrecy + PCS	Comments
Signal groups			
WhatsApp (Sender keys)			
Matrix (Megolm)			
MLS (TreeKEM)			
Our protocol (DCGKA)			

	Cost of group update (n = group size)	Forward secrecy + PCS	Comments
Signal groups	O(n)	PCS update requires every group member to send a message	Individual message to each recipient (no multicast)
WhatsApp (Sender keys)			
Matrix (Megolm)			
MLS (TreeKEM)			
Our protocol (DCGKA)			

	Cost of group update (n = group size)	Forward secrecy + PCS	Comments
Signal groups	$O(n)$	PCS update requires every group member to send a message	Individual message to each recipient (no multicast)
WhatsApp (Sender keys)	$O(n^2)$	No PCS as currently implemented	—
Matrix (Megolm)			
MLS (TreeKEM)			
Our protocol (DCGKA)			

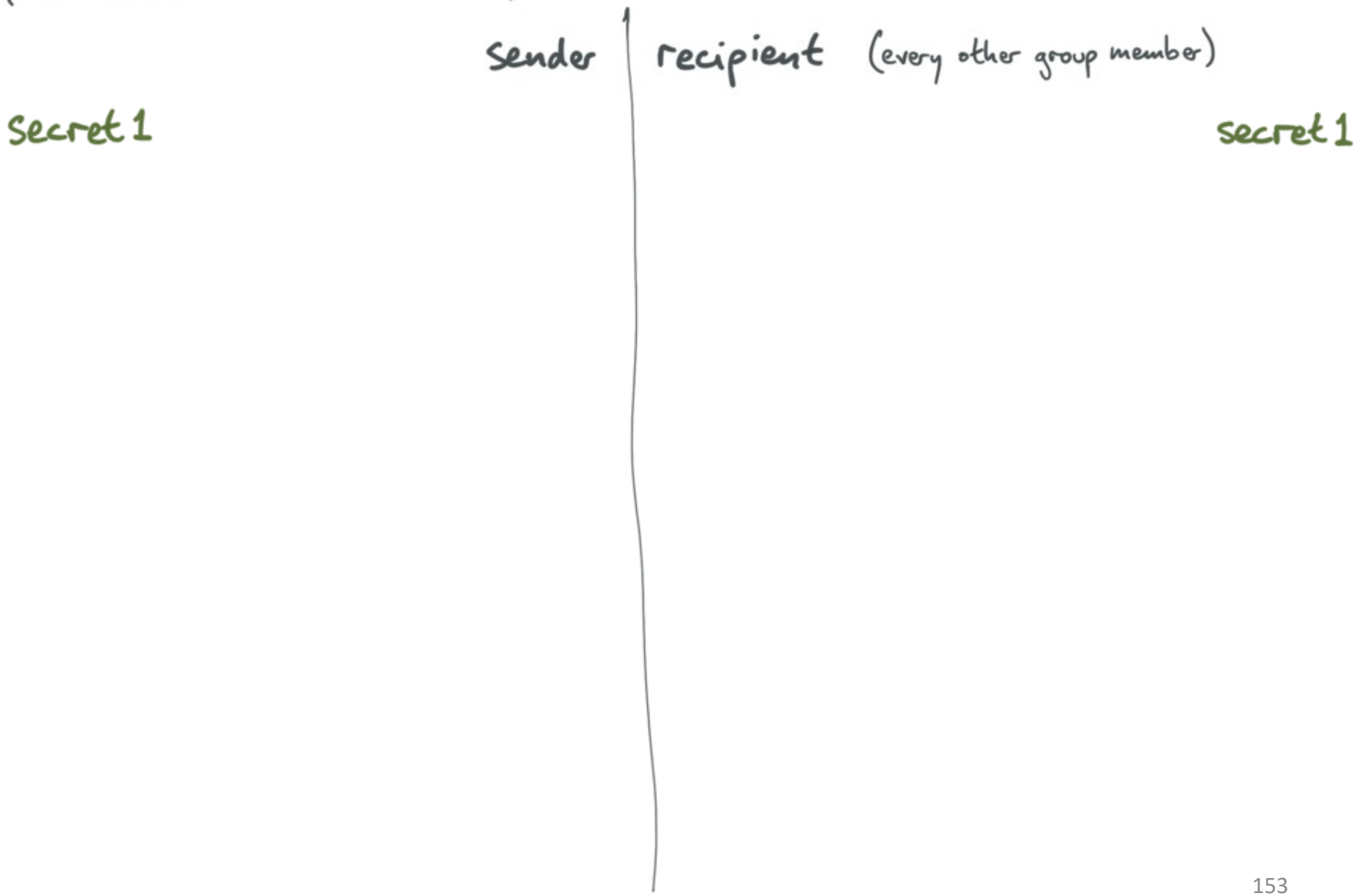
	Cost of group update (n = group size)	Forward secrecy + PCS	Comments
Signal groups	$O(n)$	PCS update requires every group member to send a message	Individual message to each recipient (no multicast)
WhatsApp (Sender keys)	$O(n^2)$	No PCS as currently implemented	—
Matrix (Megolm)	$O(n^2)$	No forward secrecy	—
MLS (TreeKEM)			
Our protocol (DCGKA)			

	Cost of group update (n = group size)	Forward secrecy + PCS	Comments
Signal groups	$O(n)$	PCS update requires every group member to send a message	Individual message to each recipient (no multicast)
WhatsApp (Sender keys)	$O(n^2)$	No PCS as currently implemented	—
Matrix (Megolm)	$O(n^2)$	No forward secrecy	—
MLS (TreeKEM)	$O(\log n)$	Forward secrecy incomplete	Requires central server — not mixnet compatible
Our protocol (DCGKA)			

	Cost of group update (n = group size)	Forward secrecy + PCS	Comments
Signal groups	$O(n)$	PCS update requires every group member to send a message	Individual message to each recipient (no multicast)
WhatsApp (Sender keys)	$O(n^2)$	No PCS as currently implemented	—
Matrix (Megolm)	$O(n^2)$	No forward secrecy	—
MLS (TreeKEM)	$O(\log n)$	Forward secrecy incomplete	Requires central server – not mixnet compatible
Our protocol (DCGKA)	$O(n)$	✓	—

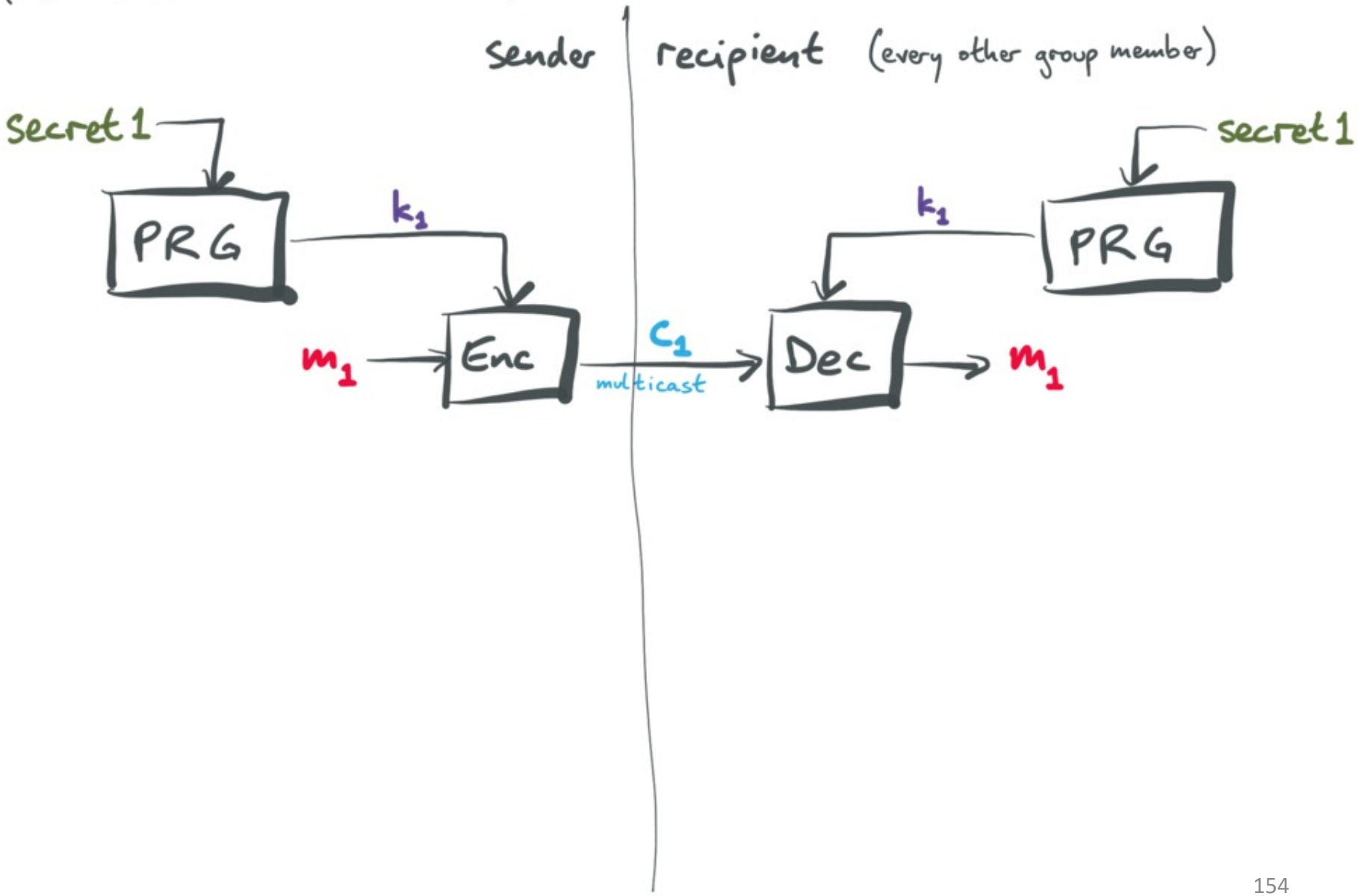
Ratchet for messages sent by a given group member

(each member has their own ratchet)



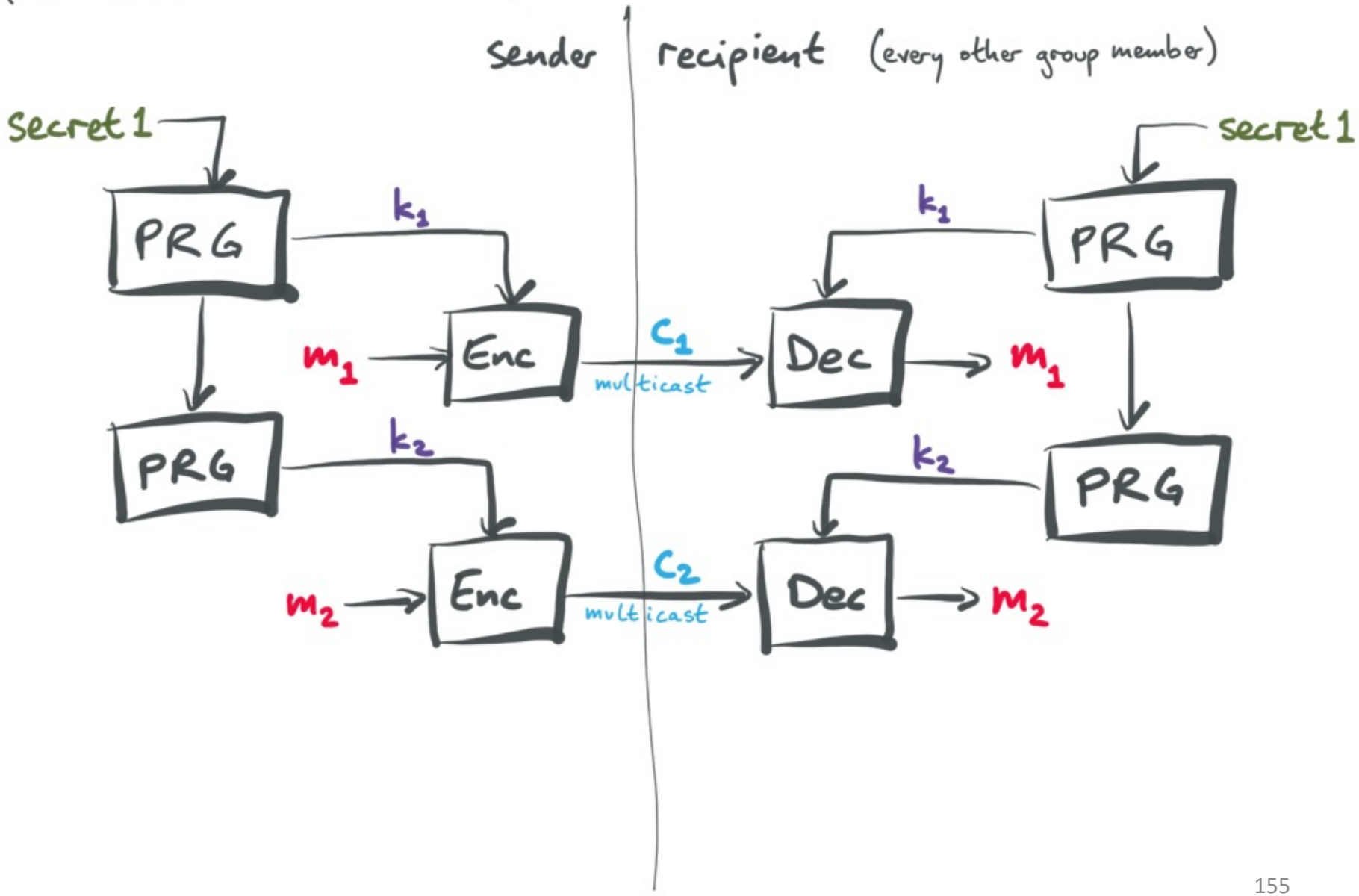
Ratchet for messages sent by a given group member

(each member has their own ratchet)



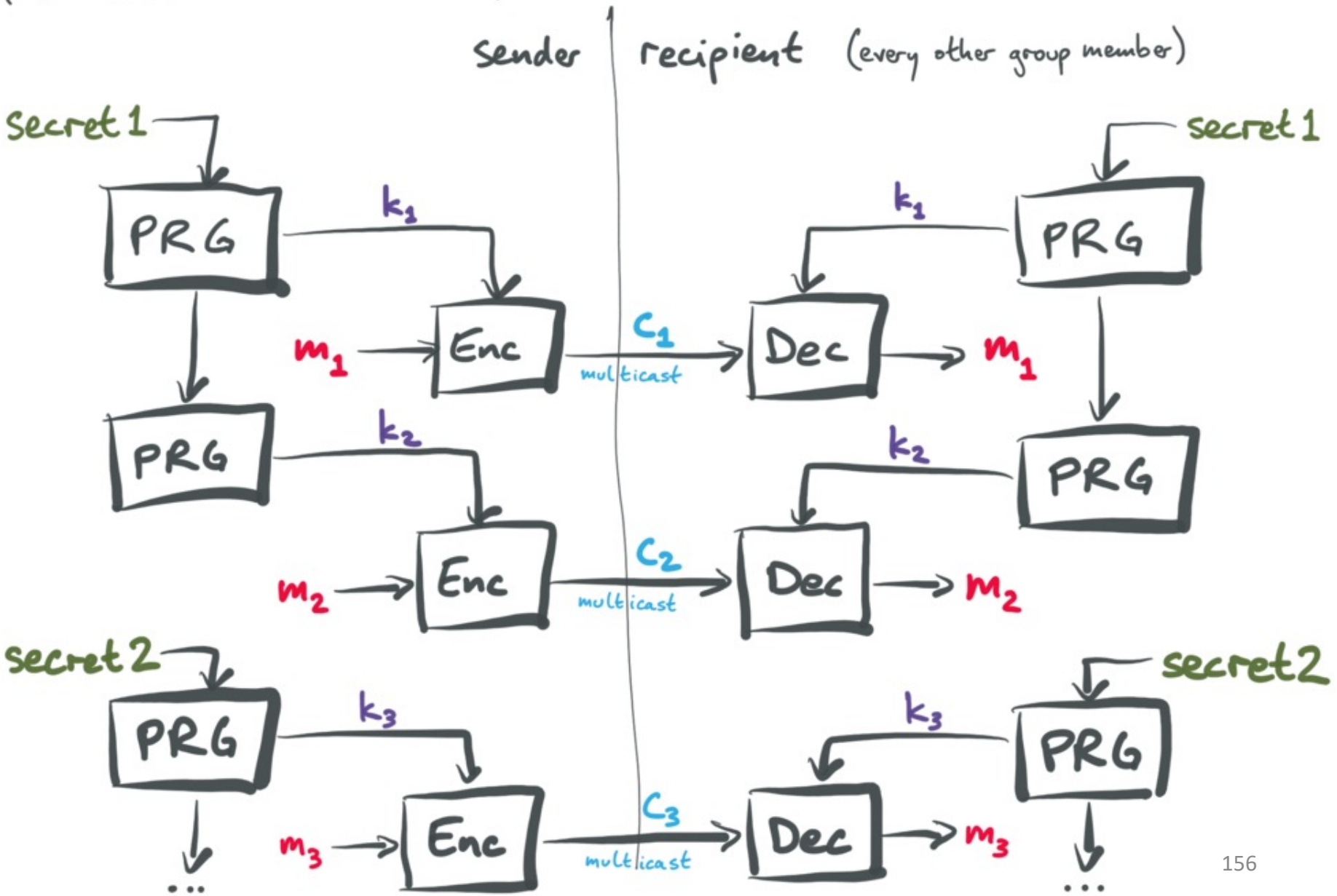
Ratchet for messages sent by a given group member

(each member has their own ratchet)



Ratchet for messages sent by a given group member

(each member has their own ratchet)



Group membership operations

Add member:

- Each existing member sends their ratchet state to the new member (via pairwise Signal protocol)

Group membership operations

Add member:

- Each existing member sends their ratchet state to the new member (via pairwise Signal protocol)

Remove member:

- Send fresh seed secret to everyone except removed user (again via pairwise Signal)

Group membership operations

Add member:

- Each existing member sends their ratchet state to the new member (via pairwise Signal protocol)

Remove member:

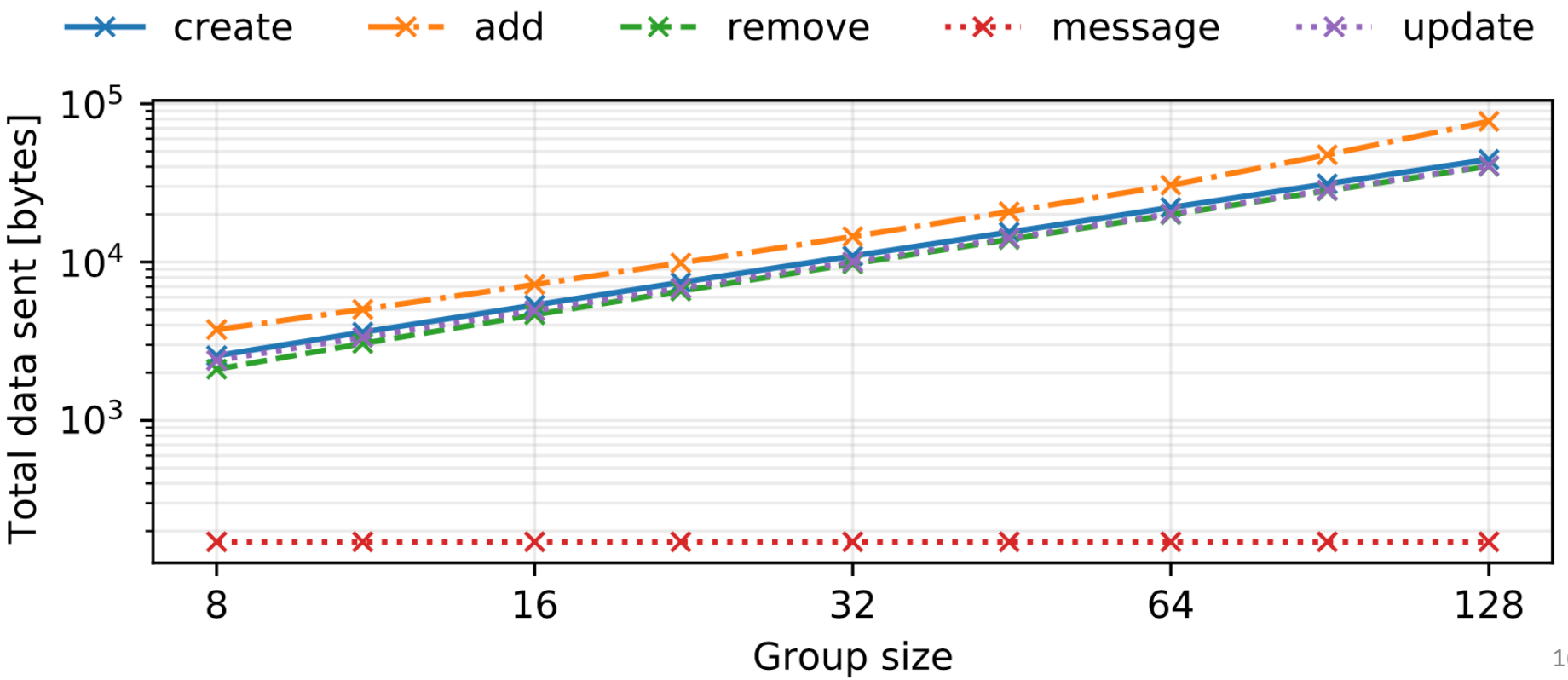
- Send fresh seed secret to everyone except removed user (again via pairwise Signal)

PCS update (key rotation):

- Send fresh seed secret to everyone (pairwise)
- On receipt, acknowledge by multicast to group (unencrypted)
- On receipt of secret or ack: update sender's ratchet

Edge cases

- Two users added concurrently
- New user added concurrently to PCS update
- Deleting secrets ASAP for forward secrecy



FUTURE WORK

- Group key agreement robust against Byzantine users + hiding user identity
- Decentralised authorisation / ACL
- Authenticating collaborators without trusted PKI
- Short, unique usernames (in mixnet context)
- Formal verification of our protocols
- Performance of BFT CRDTs
- CRDTs for more file types (rich text, spreadsheets, graphics, document annotation, CAD, ...)
- Engagement with users: journalists, activists, ...

References

- M. Kleppmann. Making CRDTs Byzantine Fault Tolerant. PaPoC 2022. doi:10.1145/3517209.3524042
- M. Weidner, M. Kleppmann, D. Hugenroth, A.R. Beresford. Key Agreement for Decentralized Secure Group Messaging with Strong Security Guarantees. ACM CCS 2021. doi:10.1145/3460120.3484542
- D. Hugenroth, M. Kleppmann, A.R. Beresford. Rollercoaster: An Efficient Group-Multicast Scheme for Mix Networks. USENIX Security 2021
- M. Kleppmann, H. Howard. Byzantine Eventual Consistency and the Fundamental Limits of Peer-to-Peer Databases. Preprint, 2020. https://arxiv.org/abs/2012.00472
- S.A. Kollmann, M. Kleppmann, A.R. Beresford. Snapdoc: Authenticated snapshots with history privacy in peer-to-peer collaborative editing. PETS 2019. doi:10.2478/popets-2019-0044
- M. Kleppmann, A. Wiggins, P. van Hardenberg, M. McGranaghan. Local-first software: You own your data, in spite of the cloud. Onward! 2019. doi:10.1145/3359591.3359737
- More at https://martin.kleppmann.com

Project suggestions

- Implement the hash graph reconciliation algorithm from https://arxiv.org/abs/2012.00472 – can you think of further optimisations for syncing large hash graphs?
- How does the hash graph reconciliation algorithm compare to Minisketch (https://arxiv.org/abs/1905.10518, https://github.com/sipa/minisketch) and Byzantine fault tolerant causal ordering (https://arxiv.org/abs/2112.11337)?
- Byzantine fault tolerant CRDTs seem like they are possible in principle, but challenging to make efficient – for example, using hashes for unique IDs would add a lot of overhead to text editing CRDTs (see Automerge's compression scheme in my other lecture). Can you sketch out a way of allowing Automerge-like metadata compression while keeping Byzantine fault tolerance?
- There have been some efforts to support the MLS encryption protocol in environments where there isn't one central server. What is the status of these projects?