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

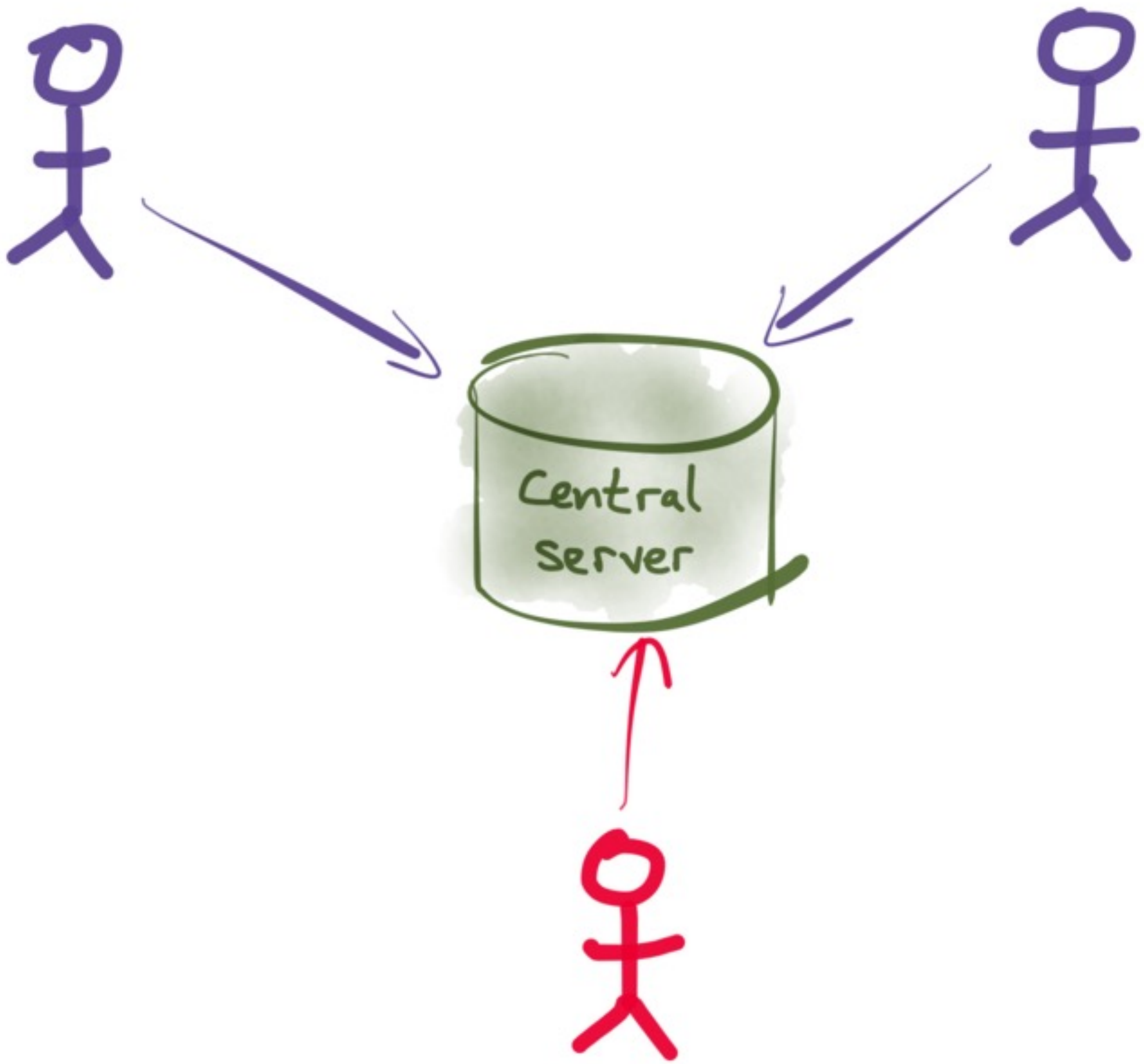
Trello

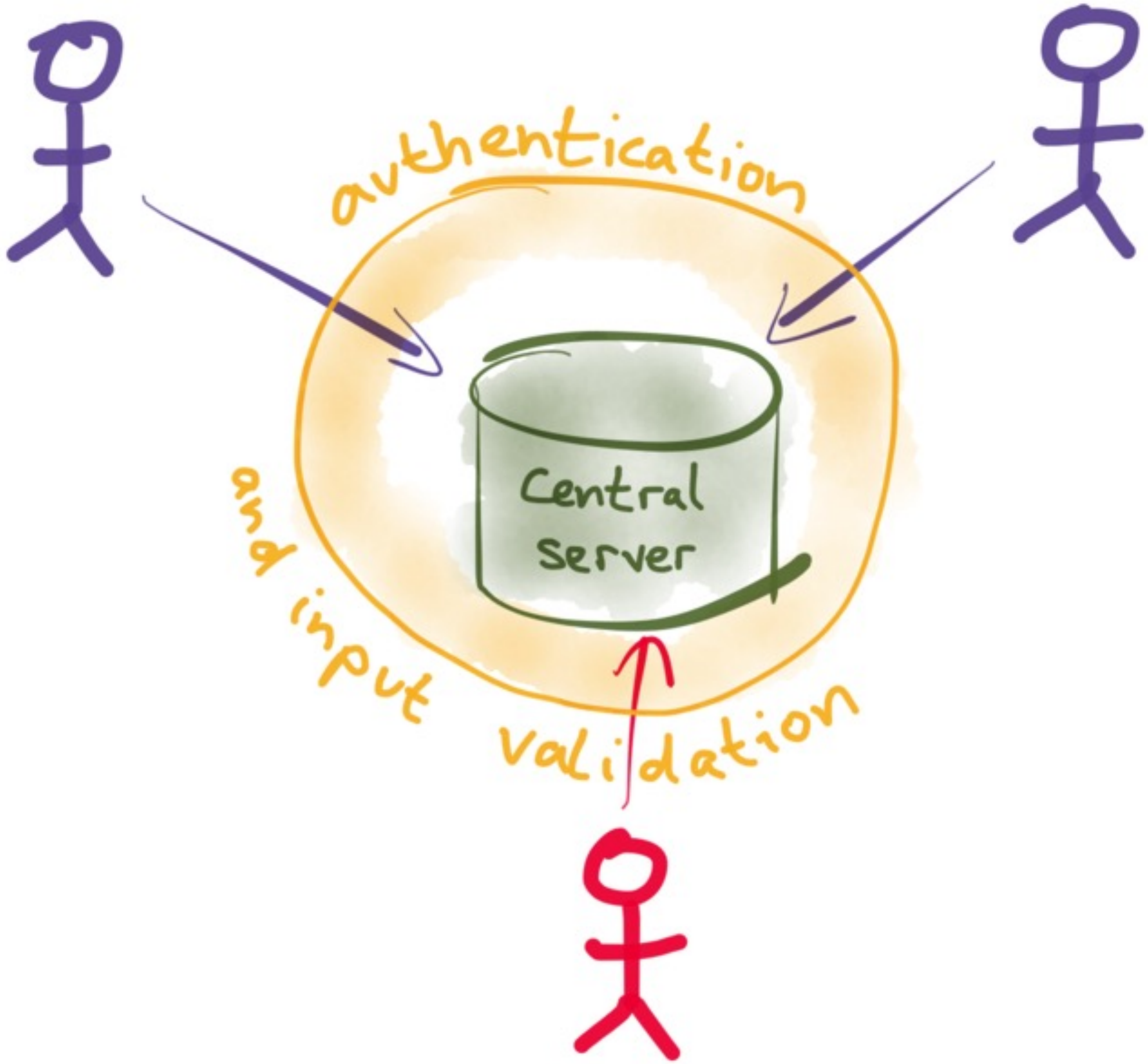
Figma

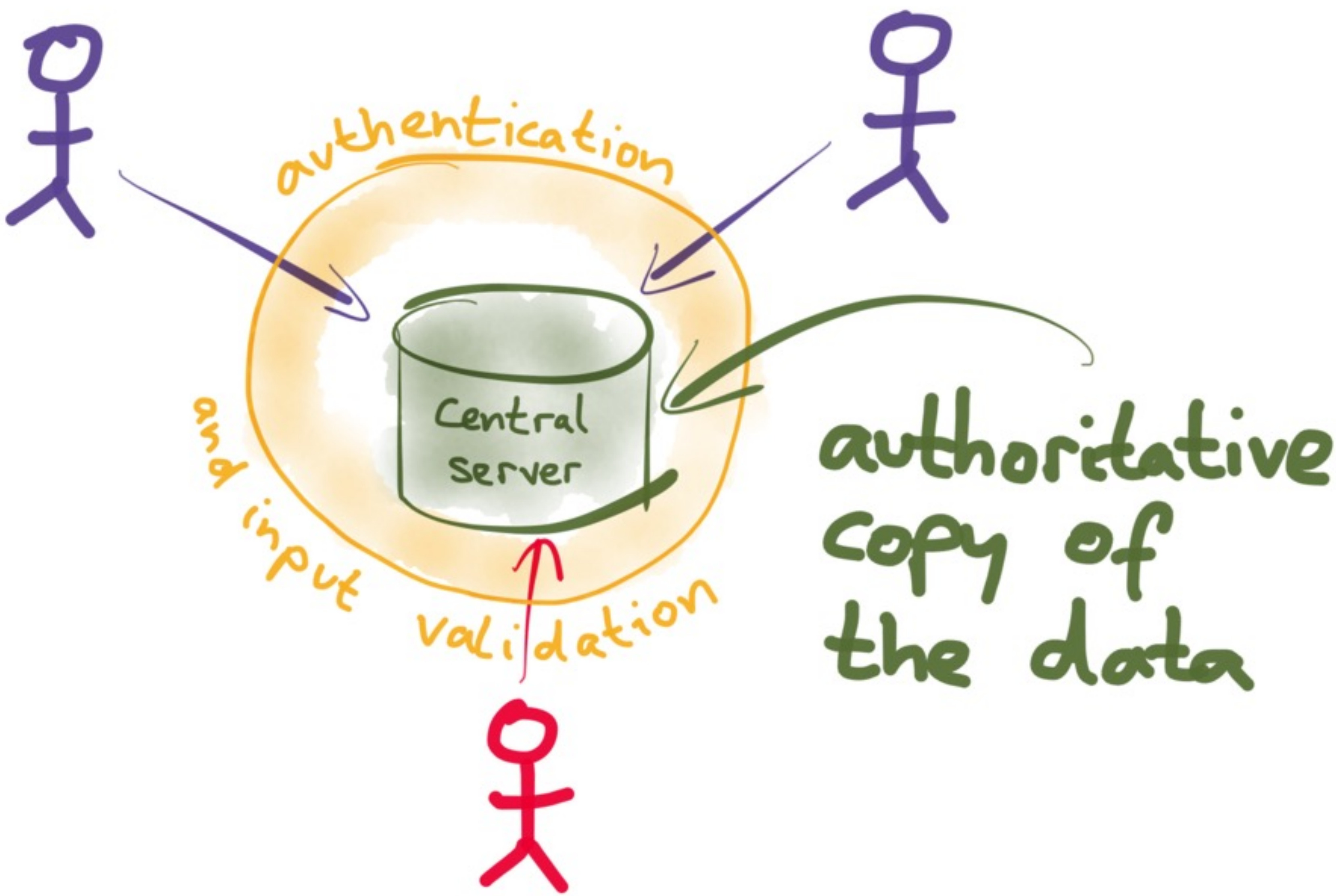
Wide range of domain-specific collaboration software, e.g. for investigative journalism, medical records, data analysis, engineering/CAD, ...

Byzantine fault tolerance:

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

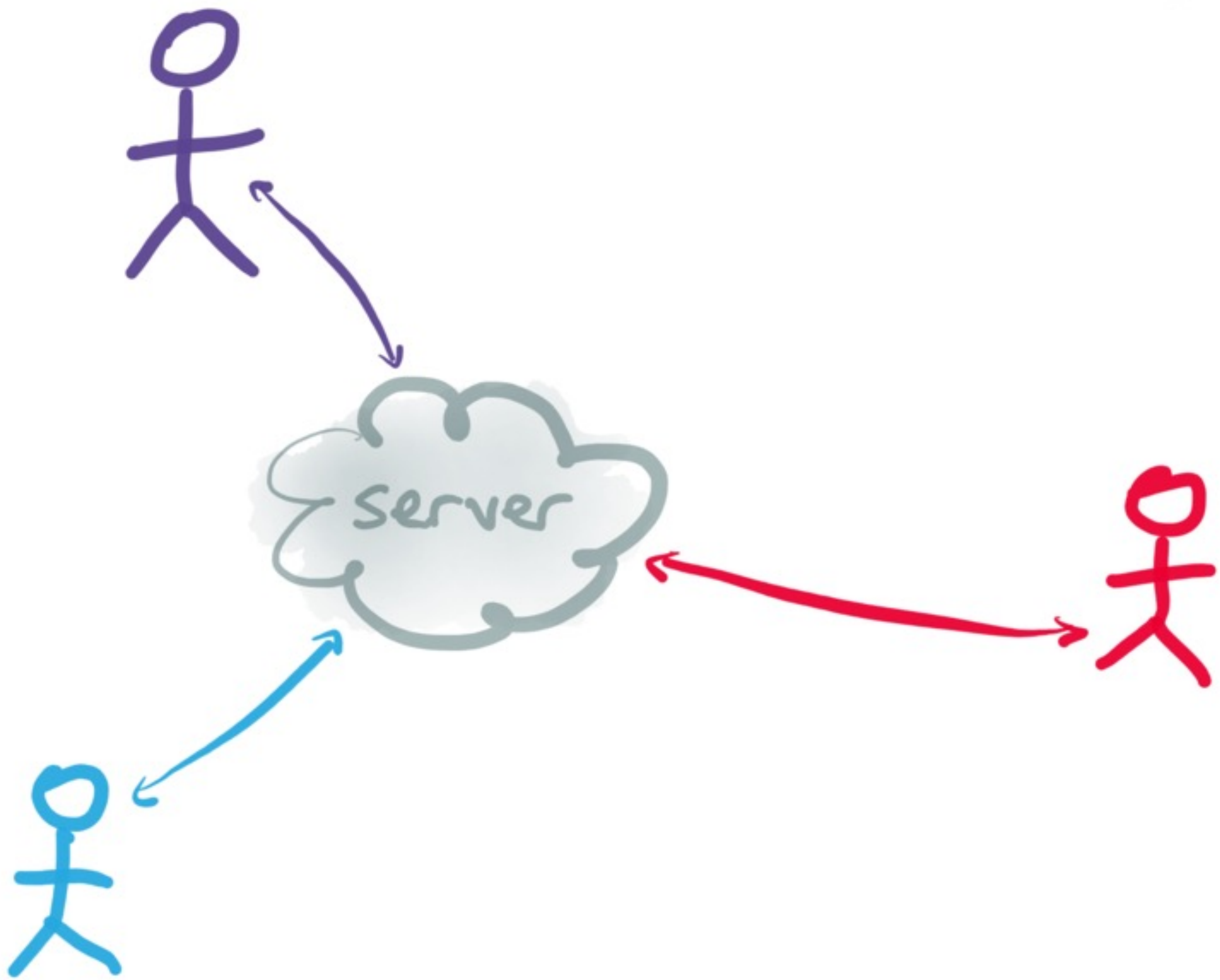

Central server


authentication
and input
validation
Central
server


authentication
and input validation
Central server
authoritative copy of the data

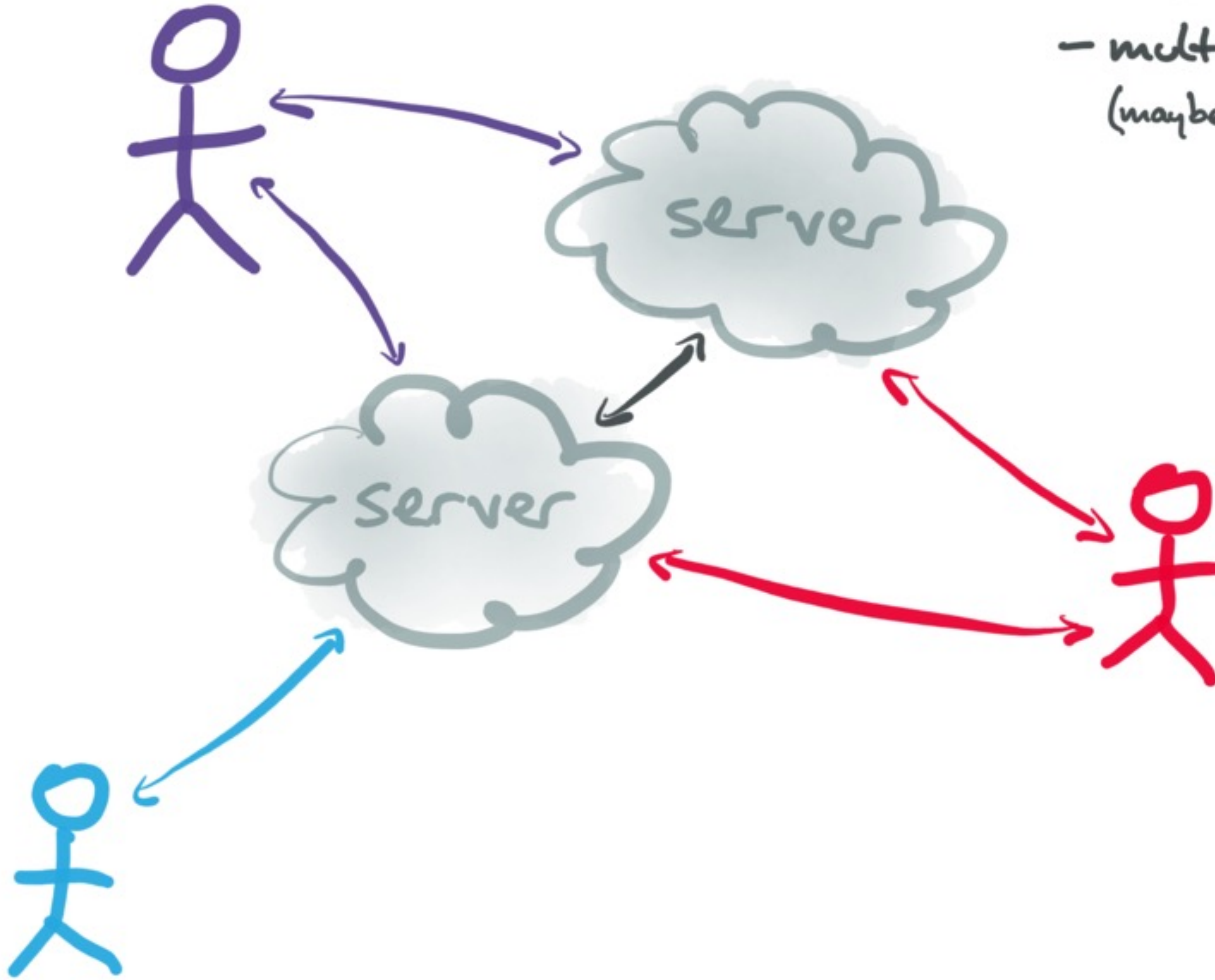
COLLABORATION IN ANY NETWORK TOPOLOGY

- one server

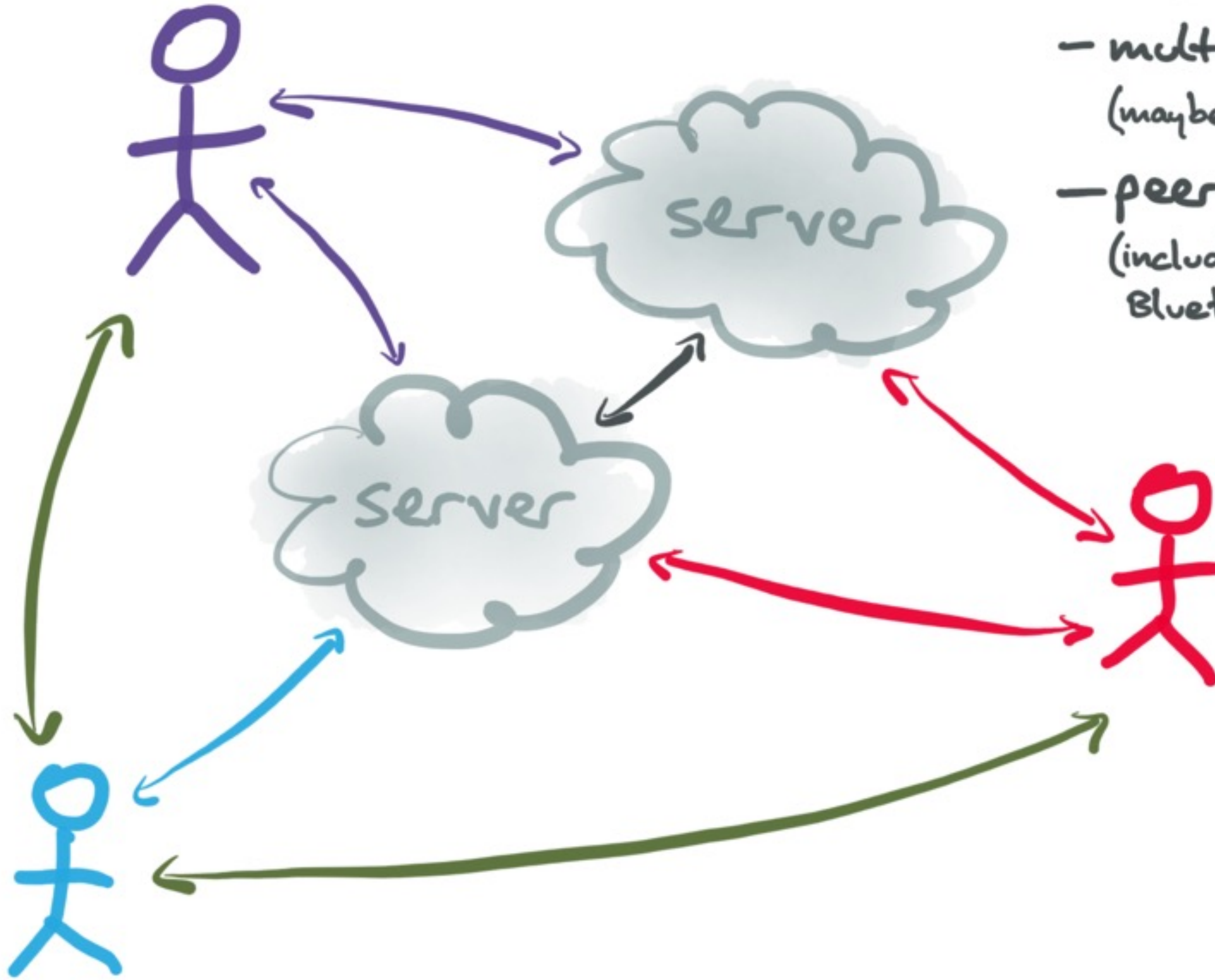


COLLABORATION IN ANY NETWORK TOPOLOGY

- one server
- multiple servers (maybe federated)



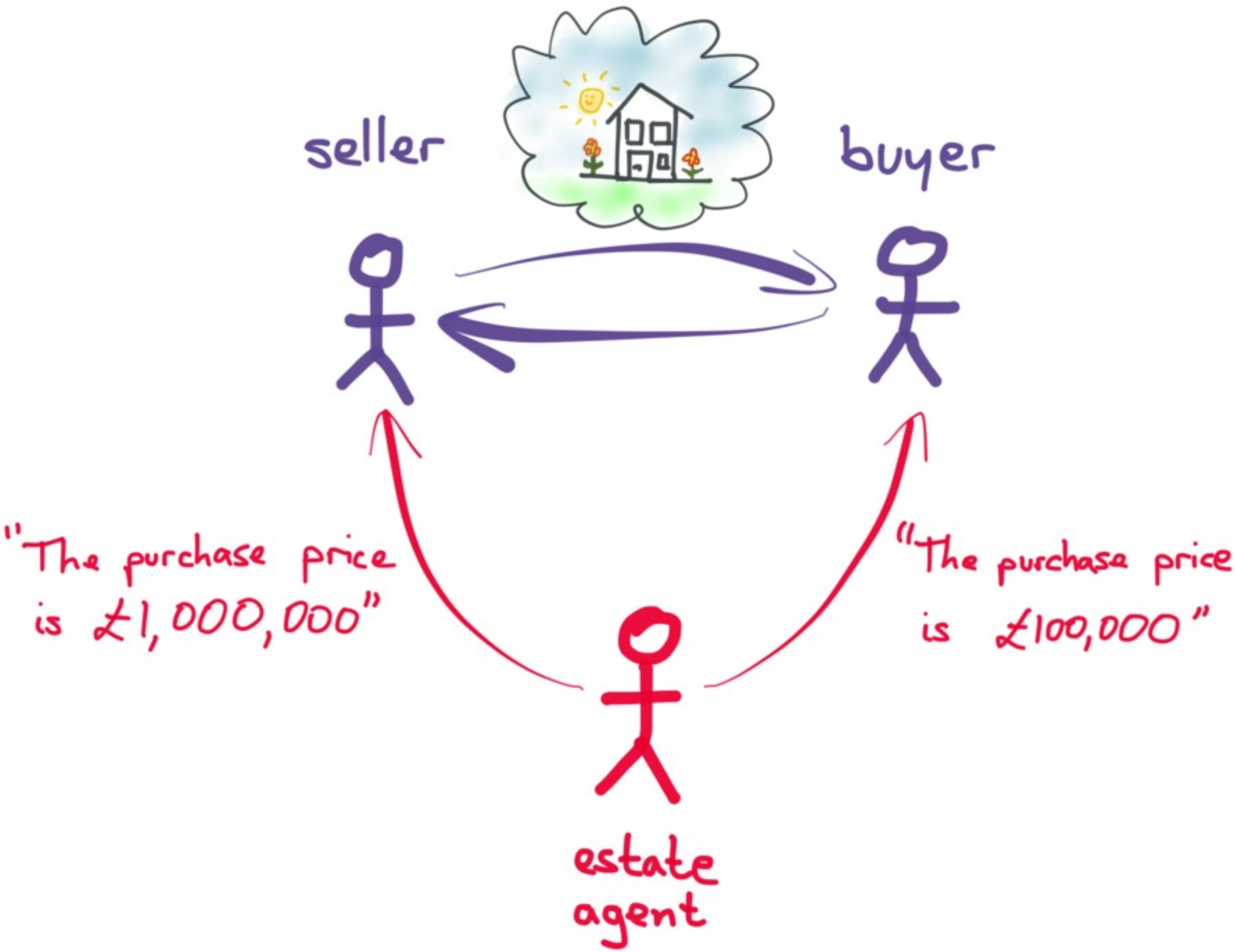
COLLABORATION IN ANY NETWORK TOPOLOGY

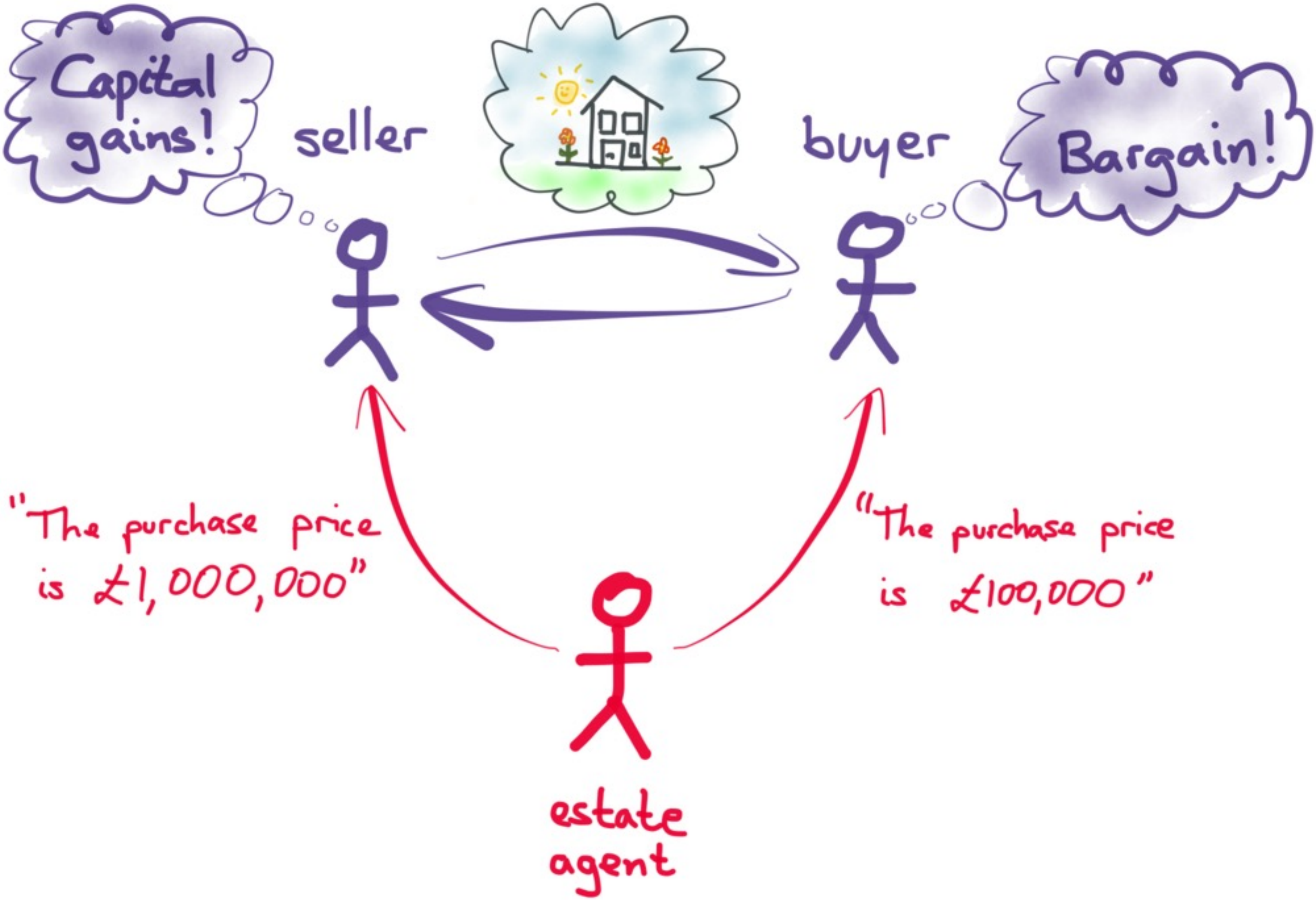


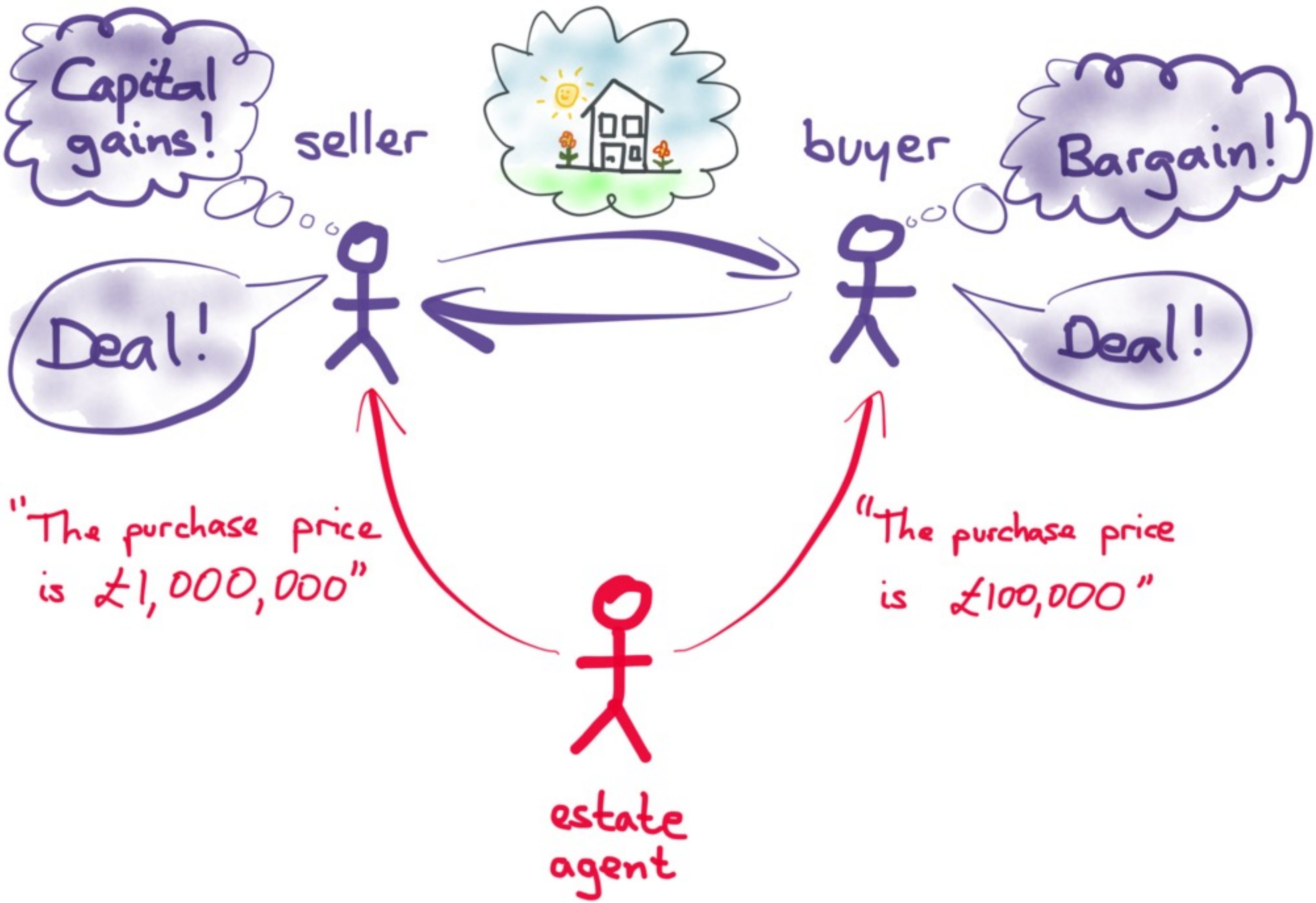
- one server
- multiple servers (maybe federated)
- peer-to-peer (including LAN, Bluetooth, etc.)


seller
buyer

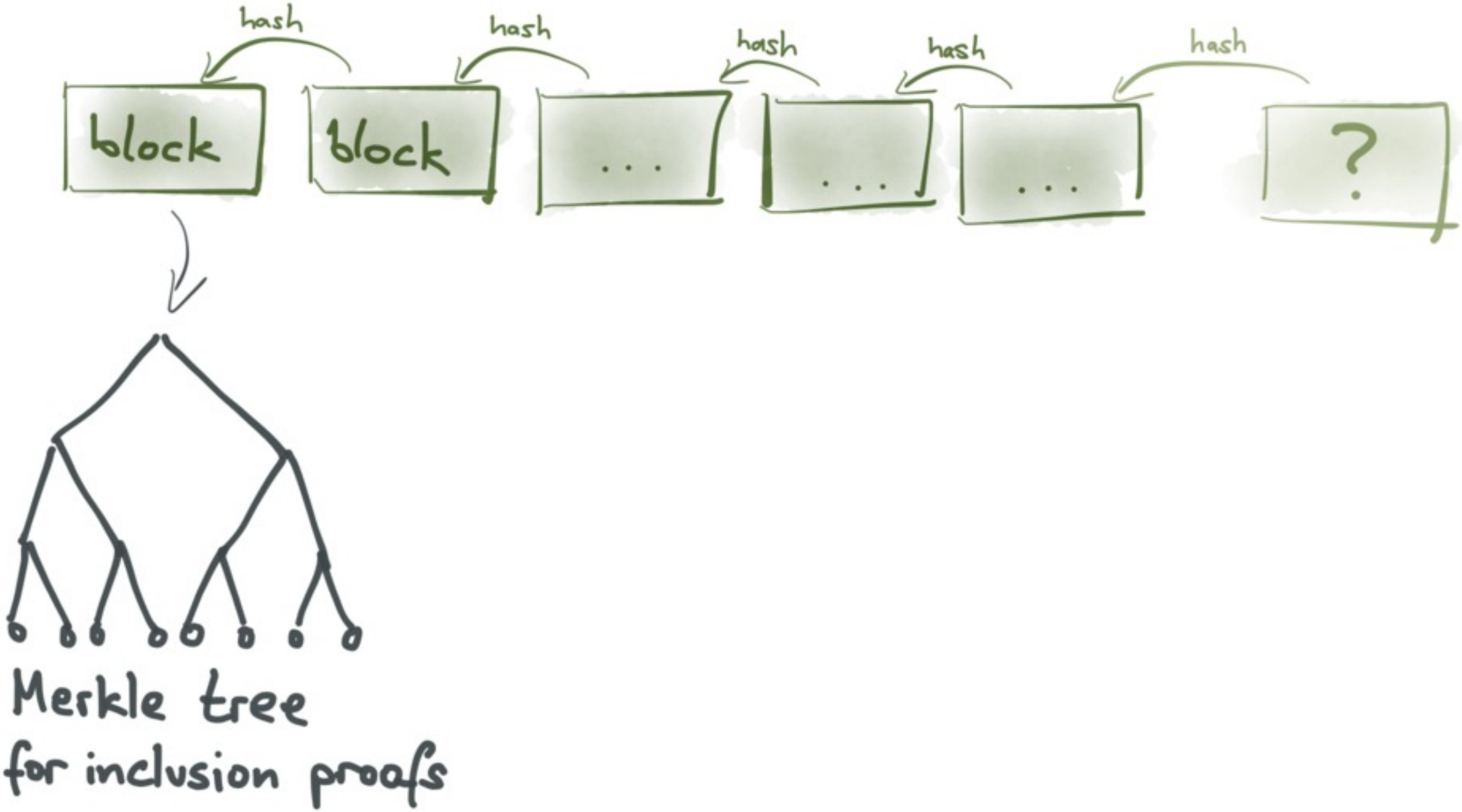

seller
buyer
estate
agent


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

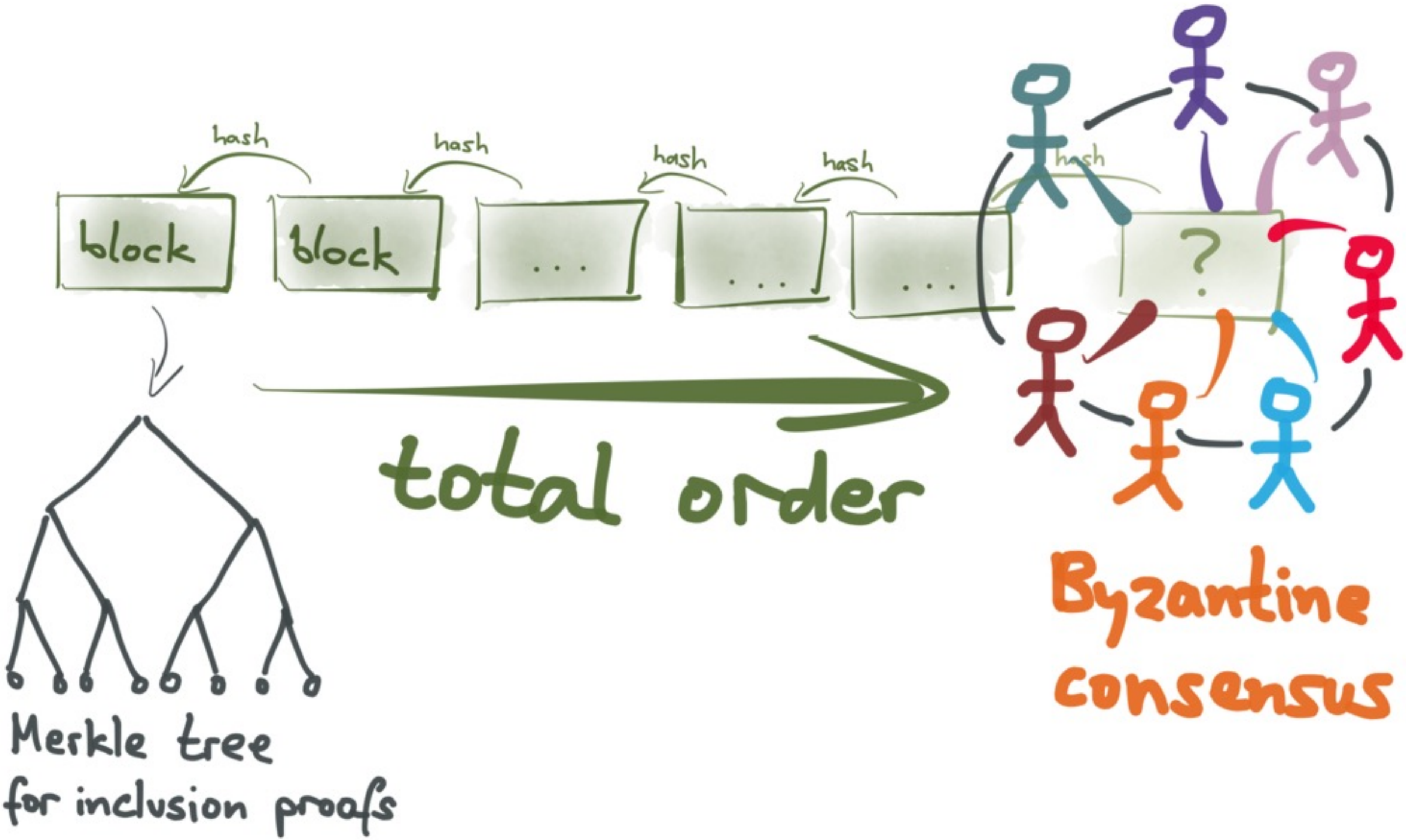

Capital gains!
seller
buyer
Bargain!
"The purchase price is £1,000,000"
"The purchase price is £100,000"
estate agent


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

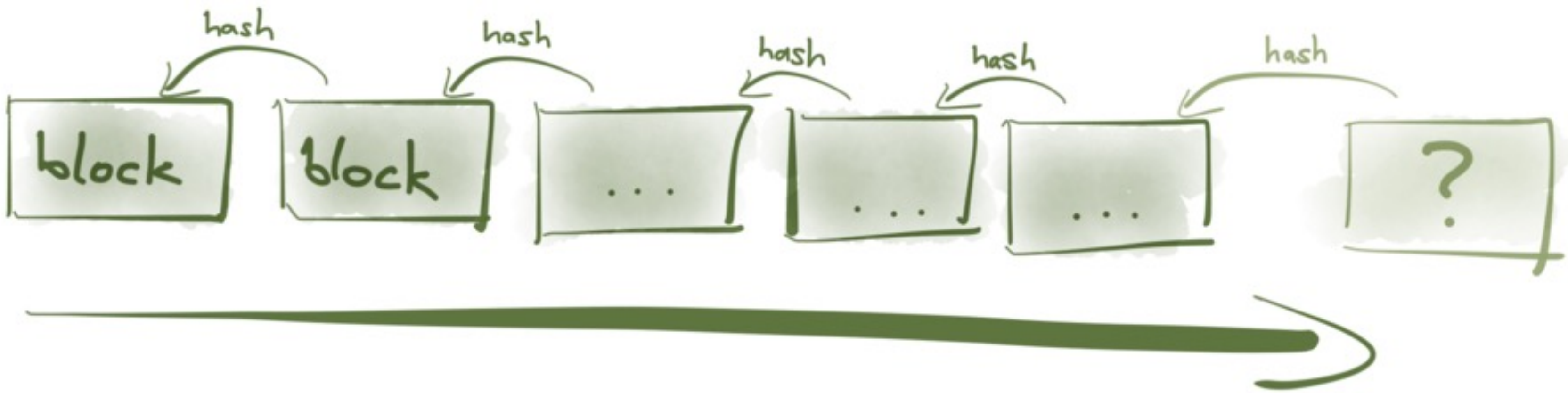
BLOCKCHAIN TO THE RESCUE?



BLOCKCHAIN TO THE RESCUE?

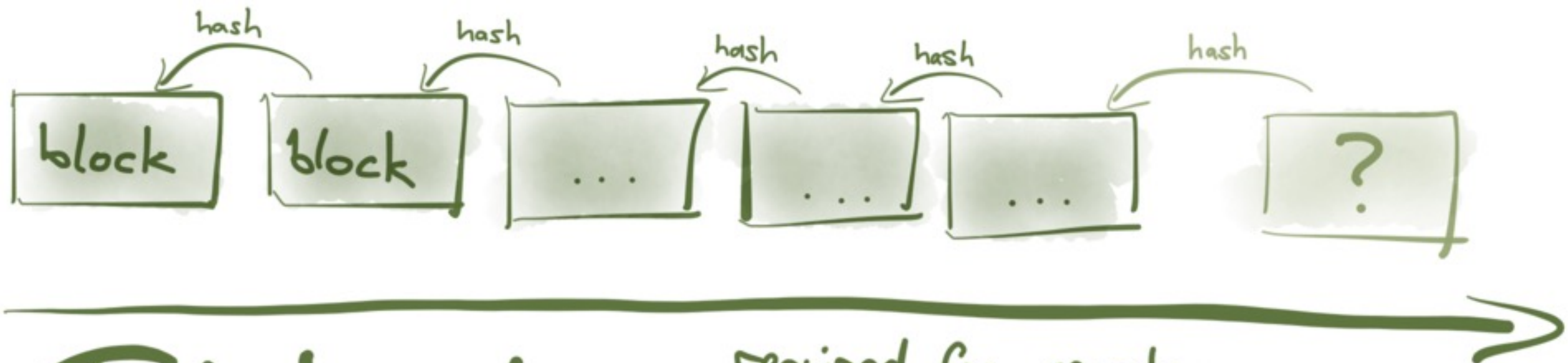


BLOCKCHAIN TO THE RESCUE?



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

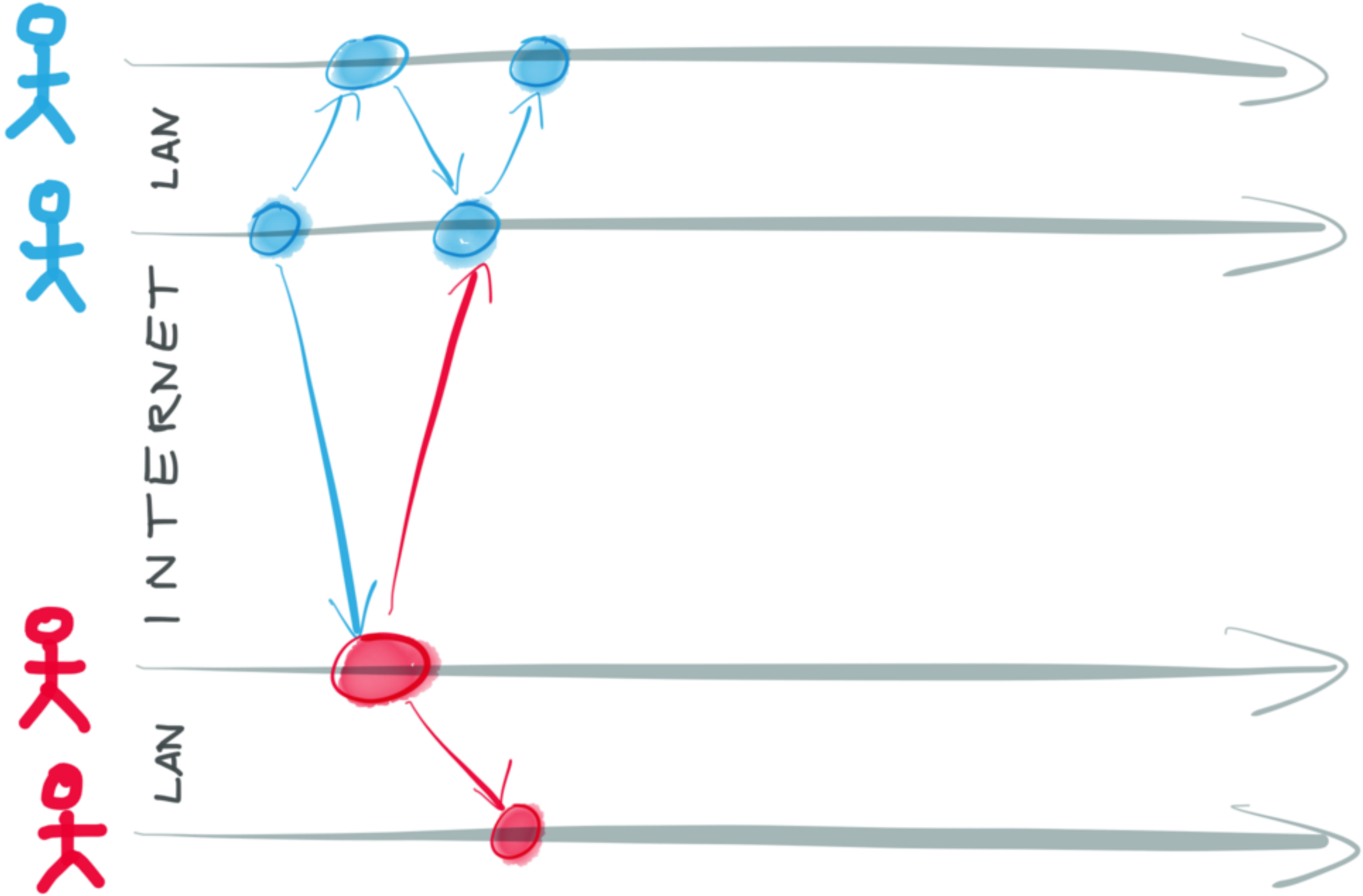
The wrong model for collaboration!

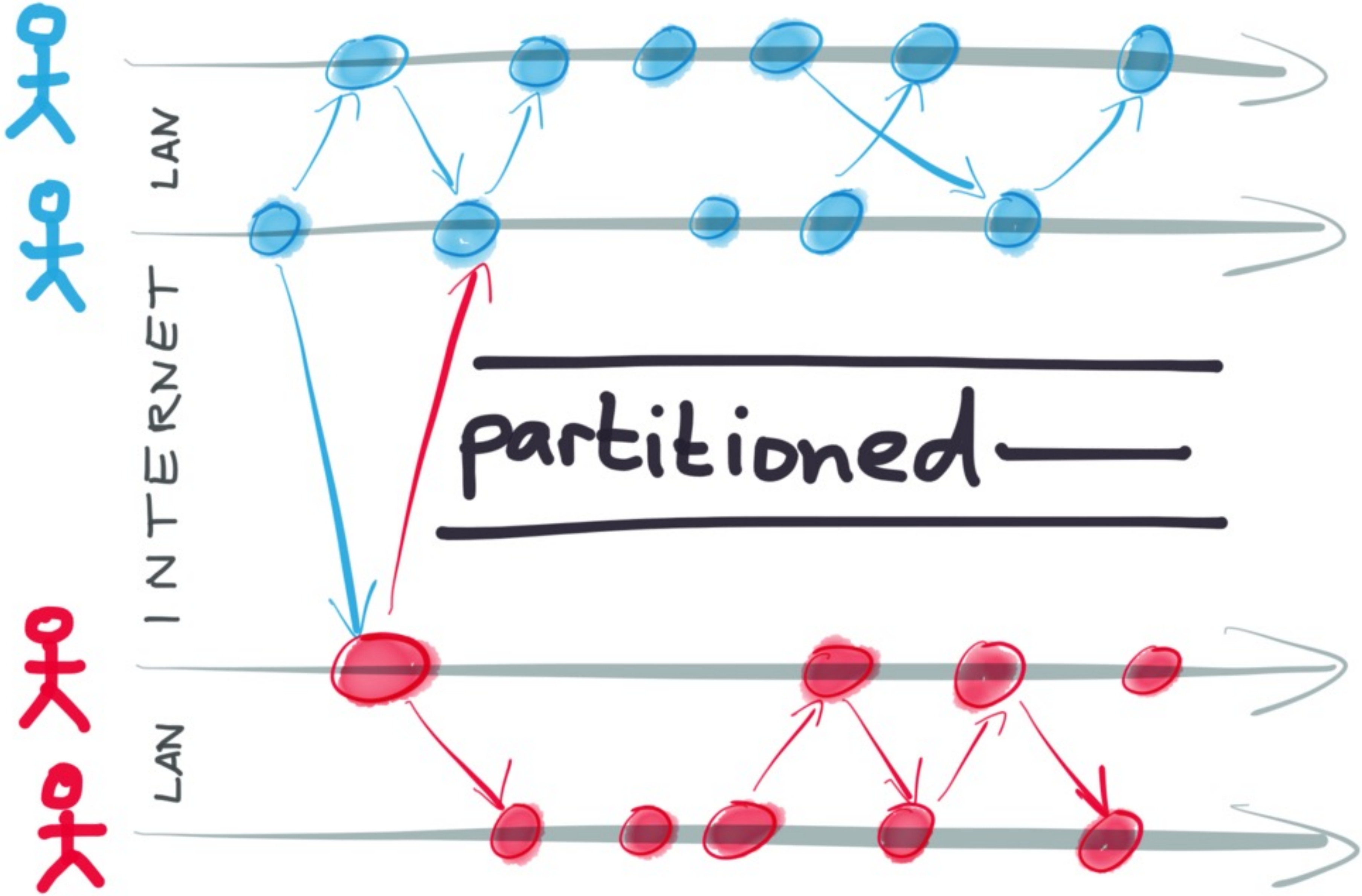


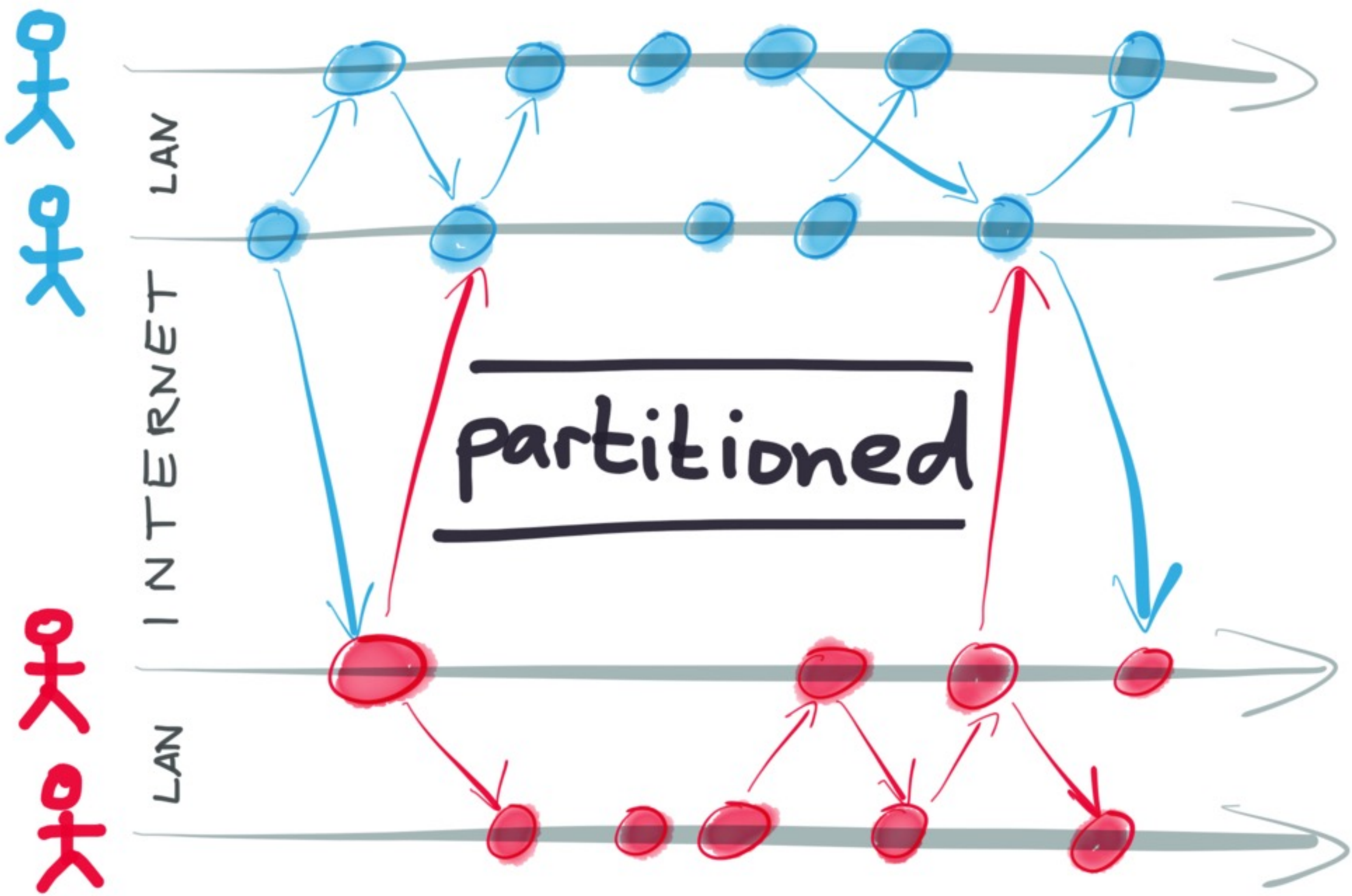
Total order required for cryptocurrencies (to prevent double-spending)

Consensus = pick one of several proposed values

Collaboration = keep all edits and merge them


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.

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

⟹ all correct replicas eventually apply u

Convergence:

Two replicas have applied same set of updates

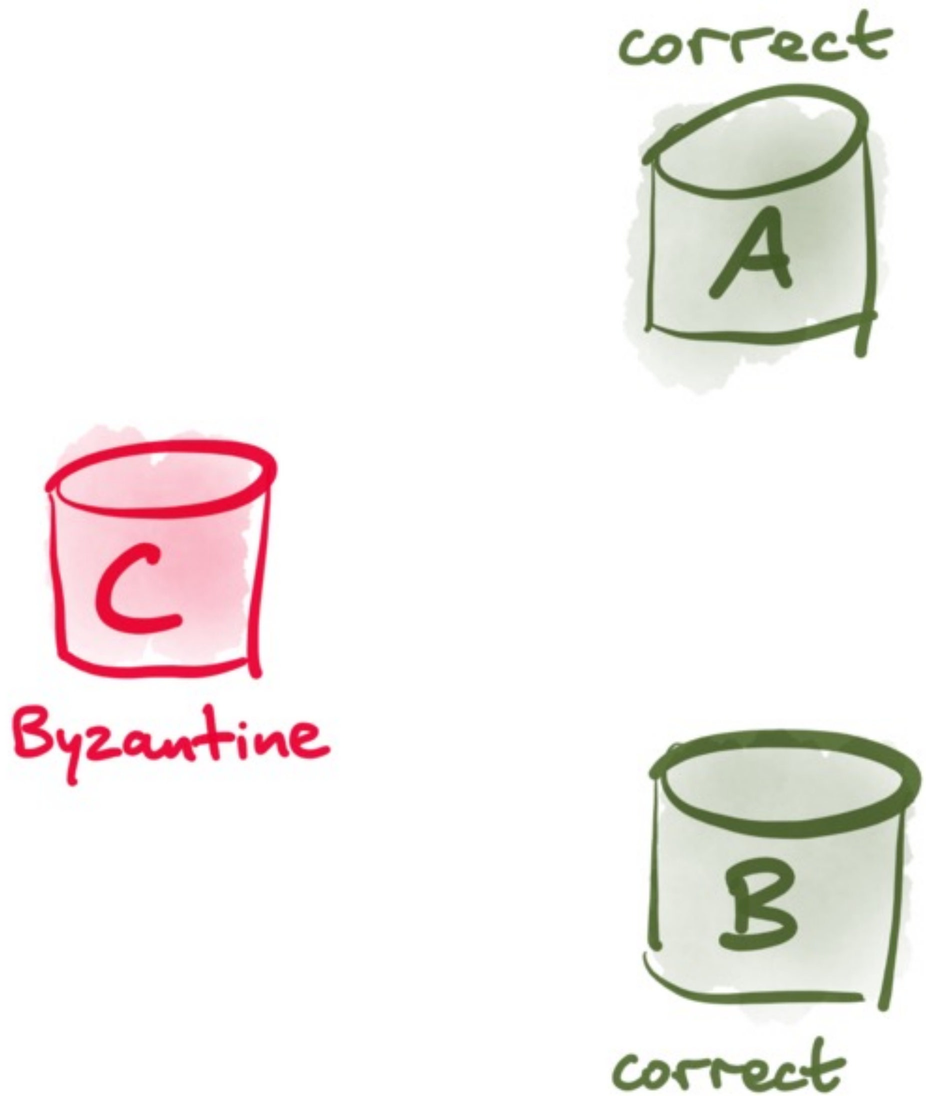
⟹ they are in the same state

Invariant preservation:

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

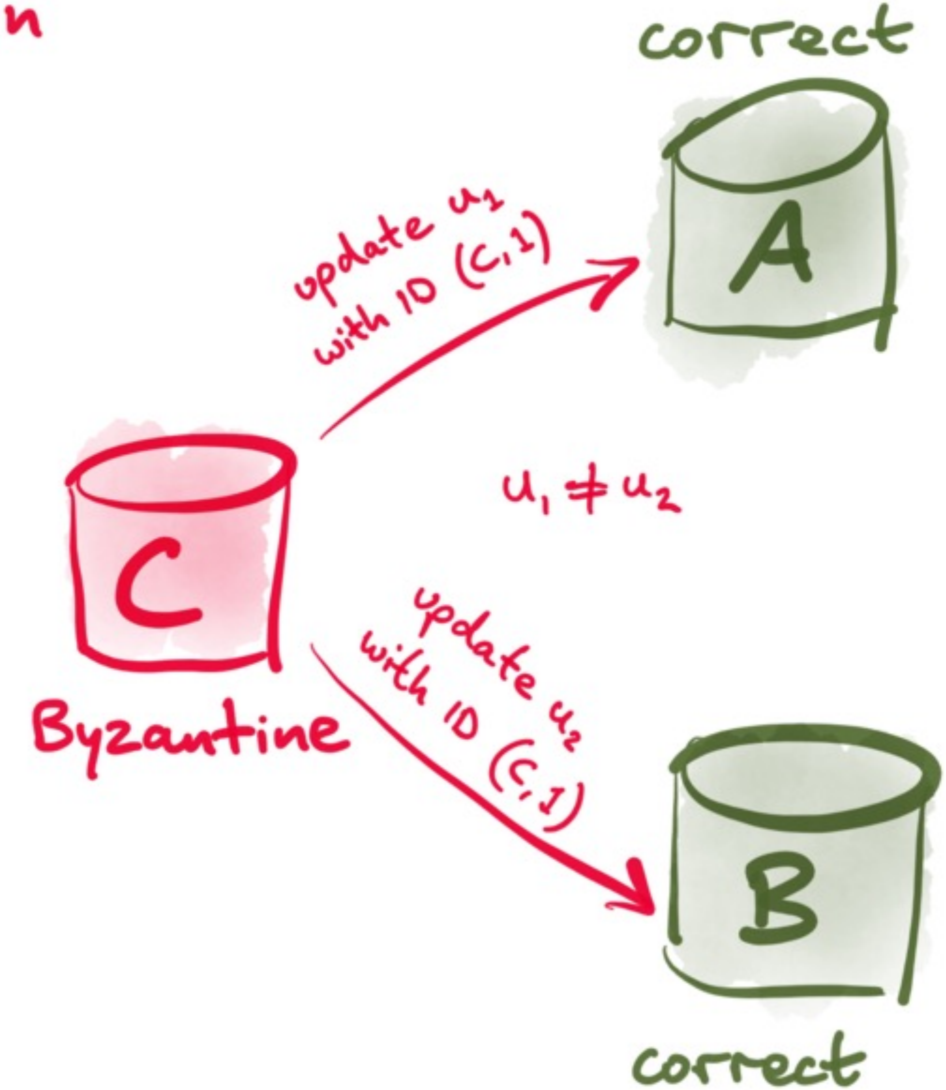
(and a few other, more technical properties)

VERSION VECTORS ARE NOT SAFE

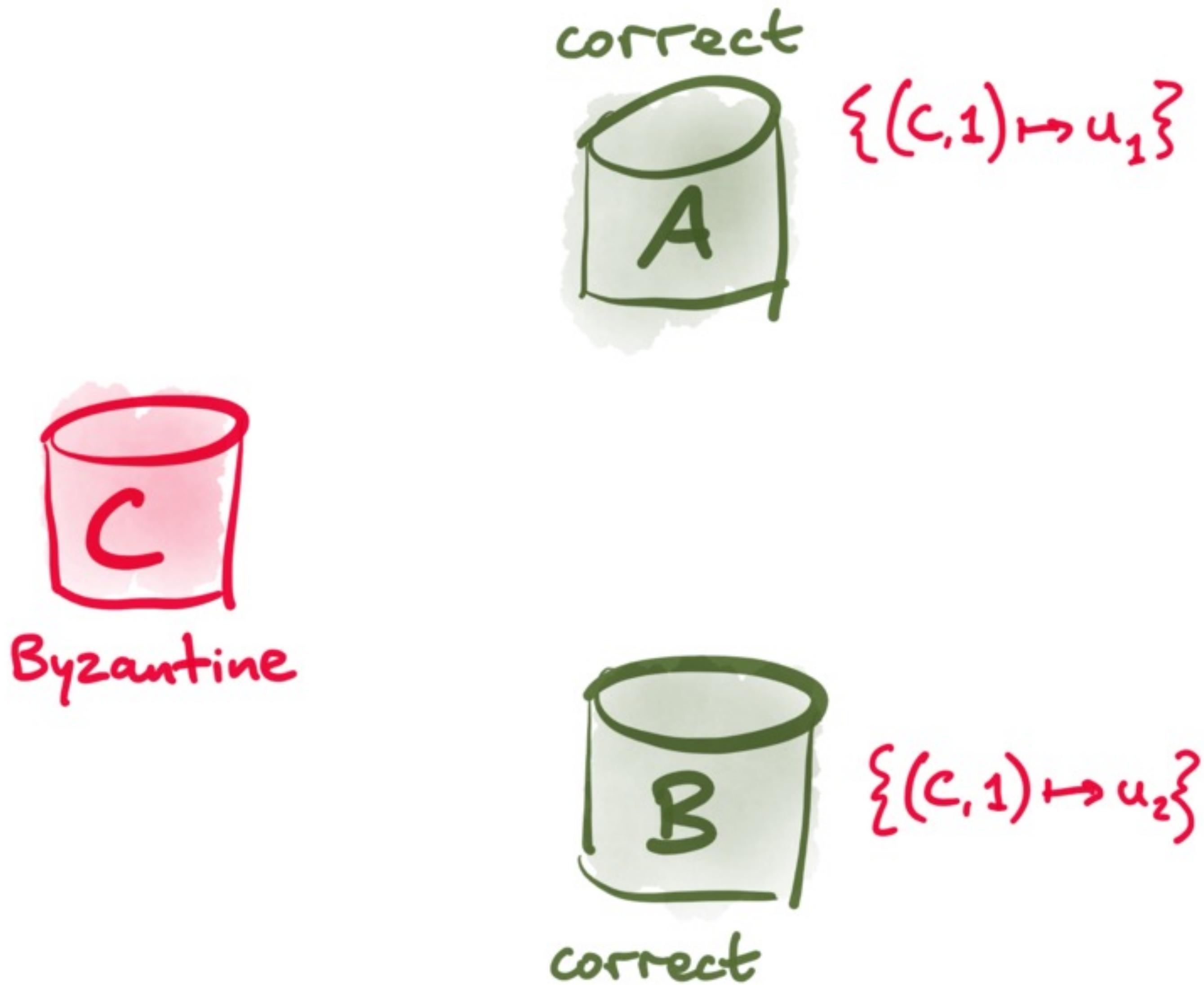


VERSION VECTORS ARE NOT SAFE

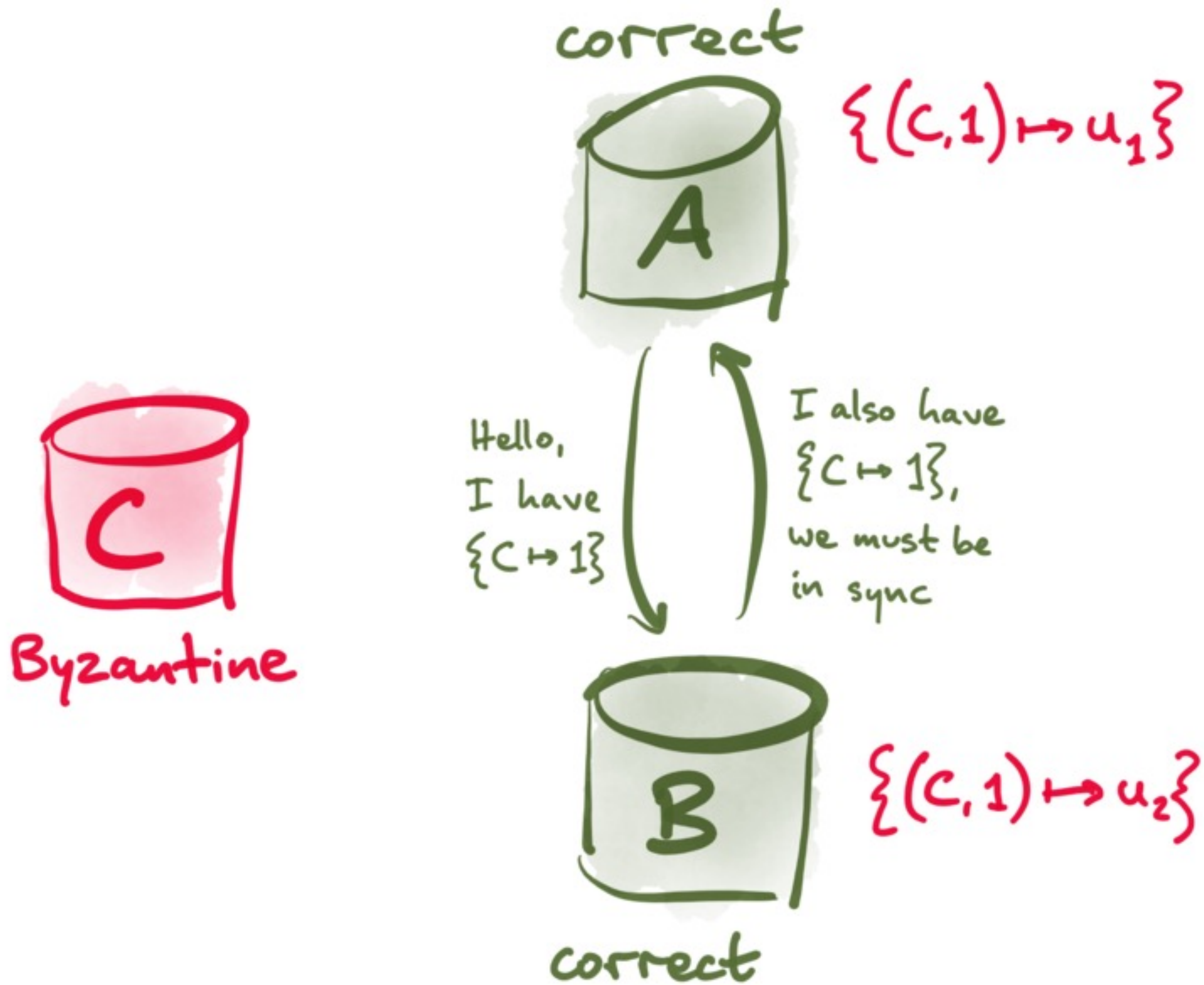
"equivocation"



VERSION VECTORS ARE NOT SAFE

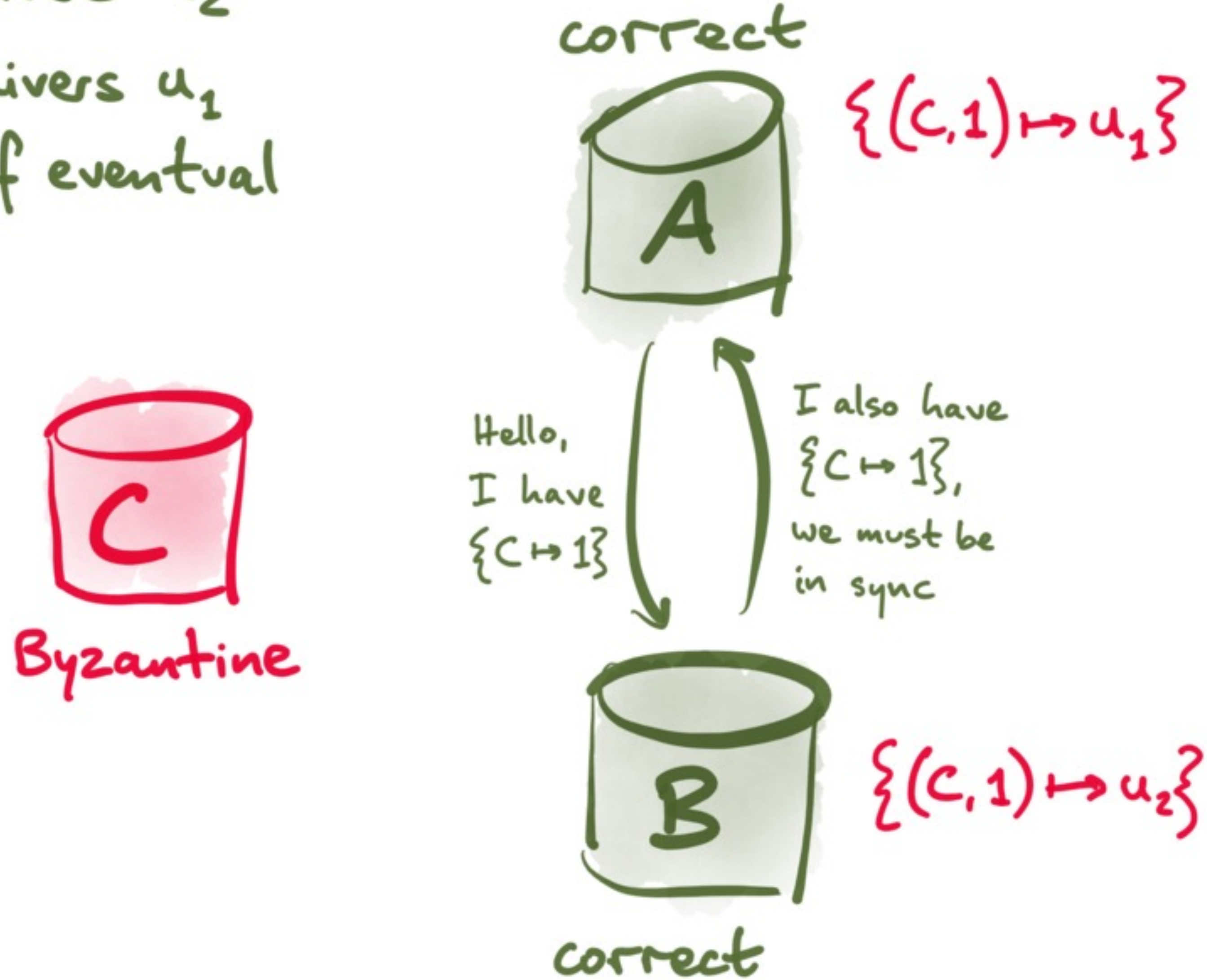


VERSION VECTORS ARE NOT SAFE



VERSION VECTORS ARE NOT SAFE

A never delivers u_2
B never delivers u_1
⇒ failure of eventual delivery

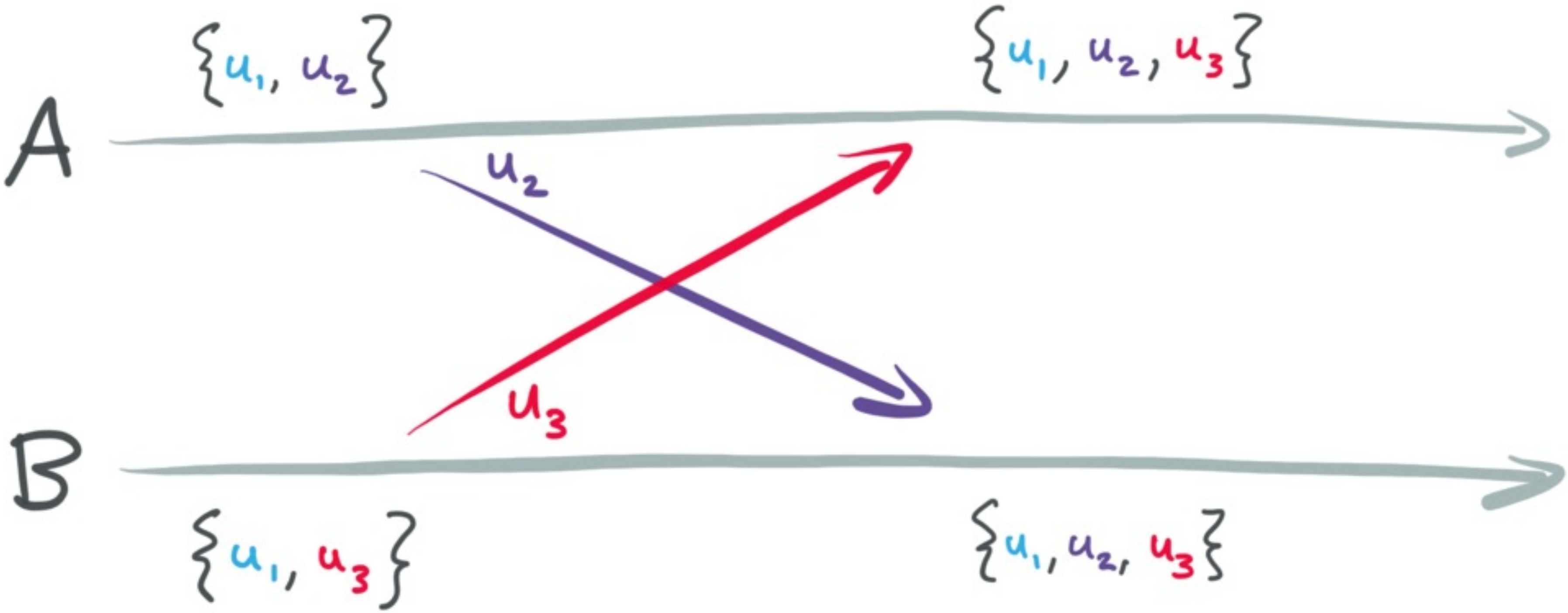


ENSURING EVENTUAL DELIVERY

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

ENSURING EVENTUAL DELIVERY

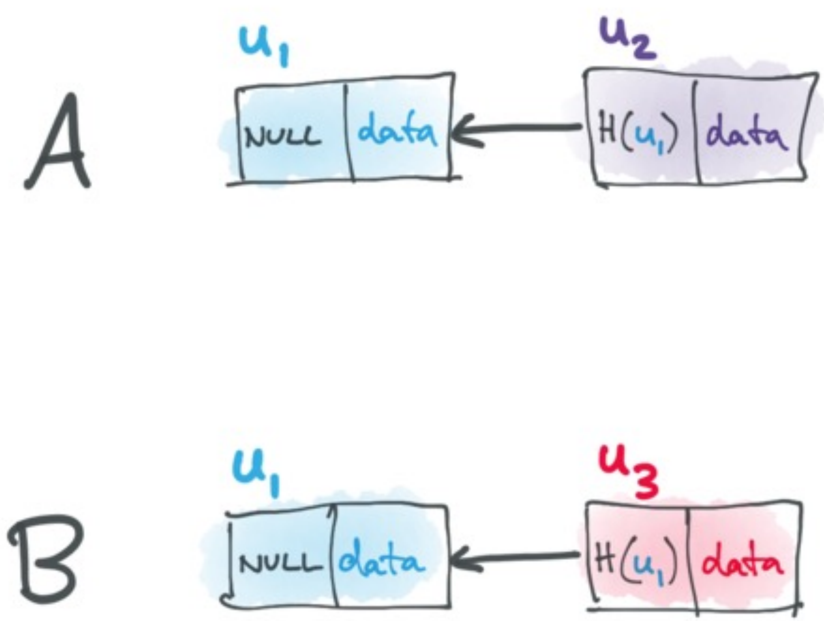
Nodes connect pairwise, send each other updates that the other doesn't have



ENSURING EVENTUAL DELIVERY

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

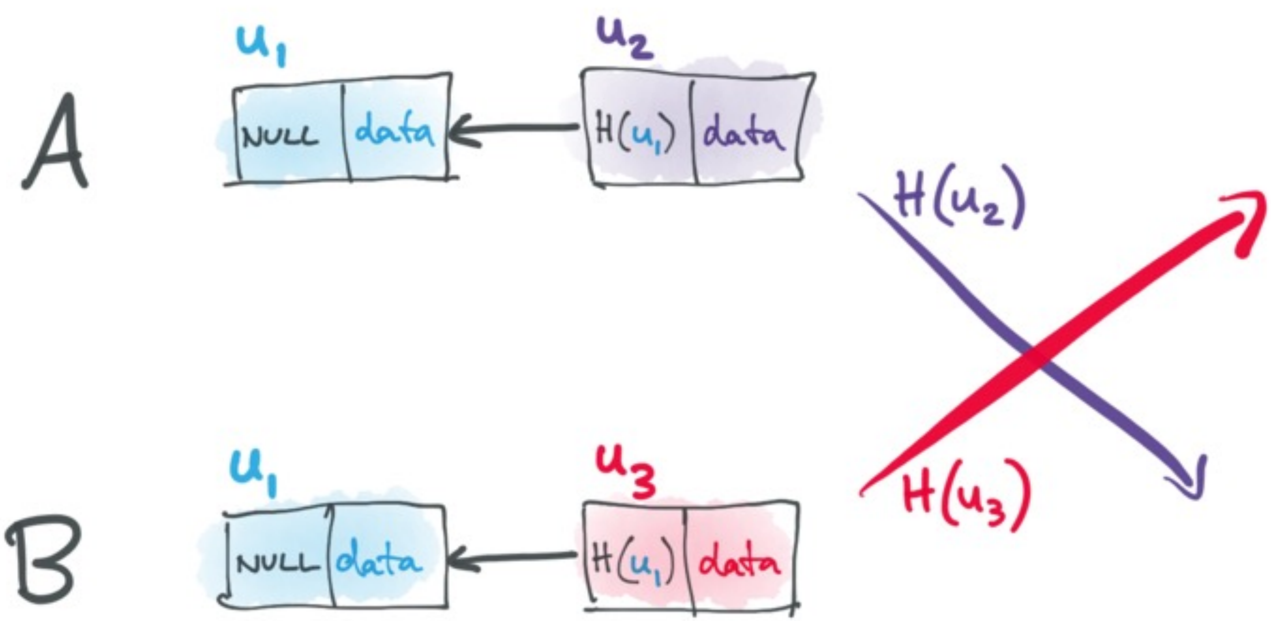
Hash graph (like Git!):



ENSURING EVENTUAL DELIVERY

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

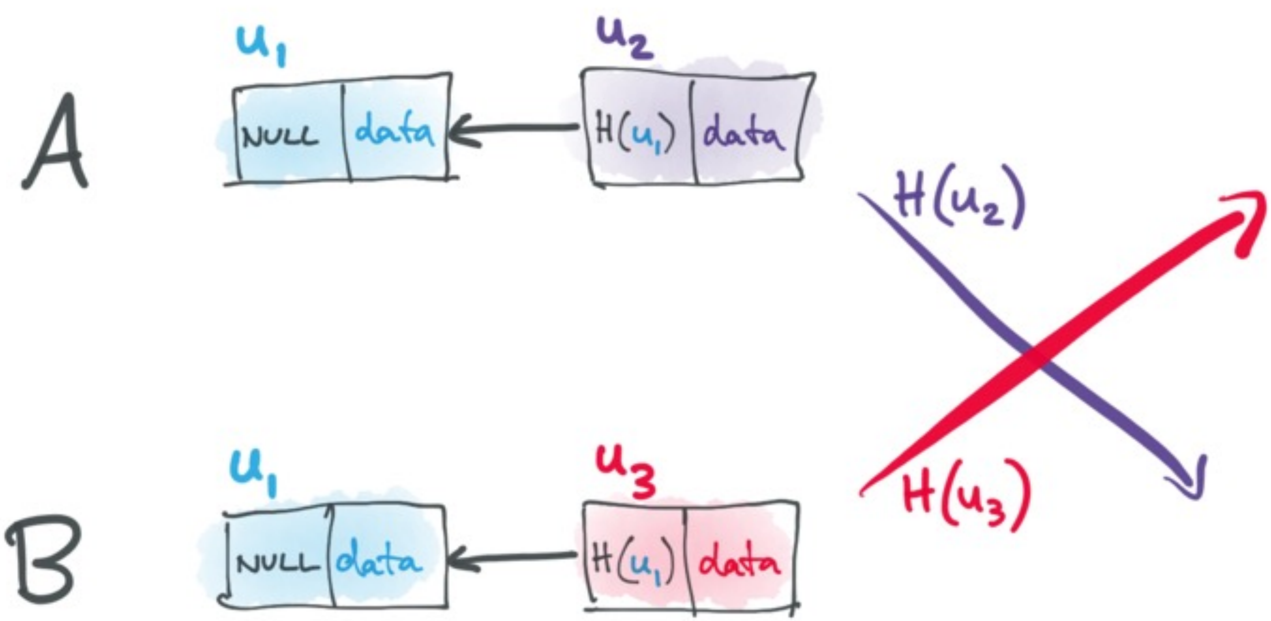
Hash graph (like Git!):



ENSURING EVENTUAL DELIVERY

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

Hash graph (like Git!):

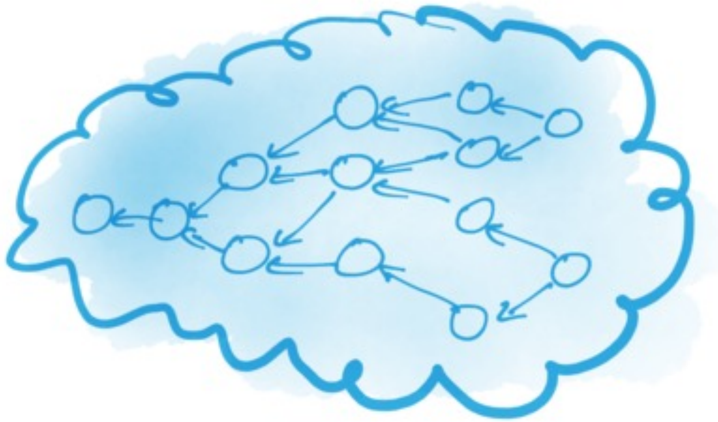


hash unknown? work backwards in hash DAG until known hash is found

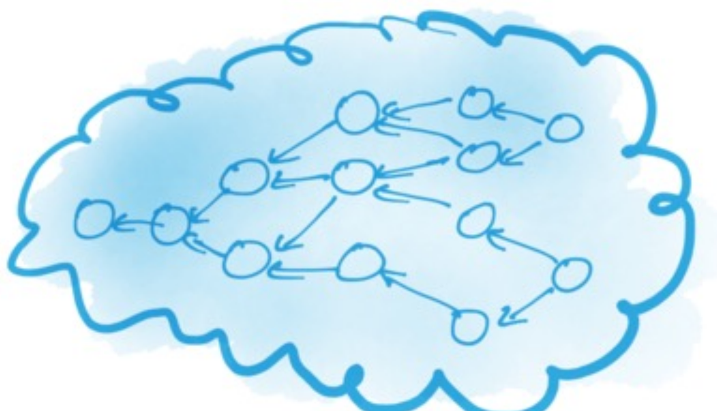
Assuming a collision-resistant hash function, a Byzantine node cannot cause two correct nodes to believe they are in sync when in fact they have diverged.

BEC replication

A

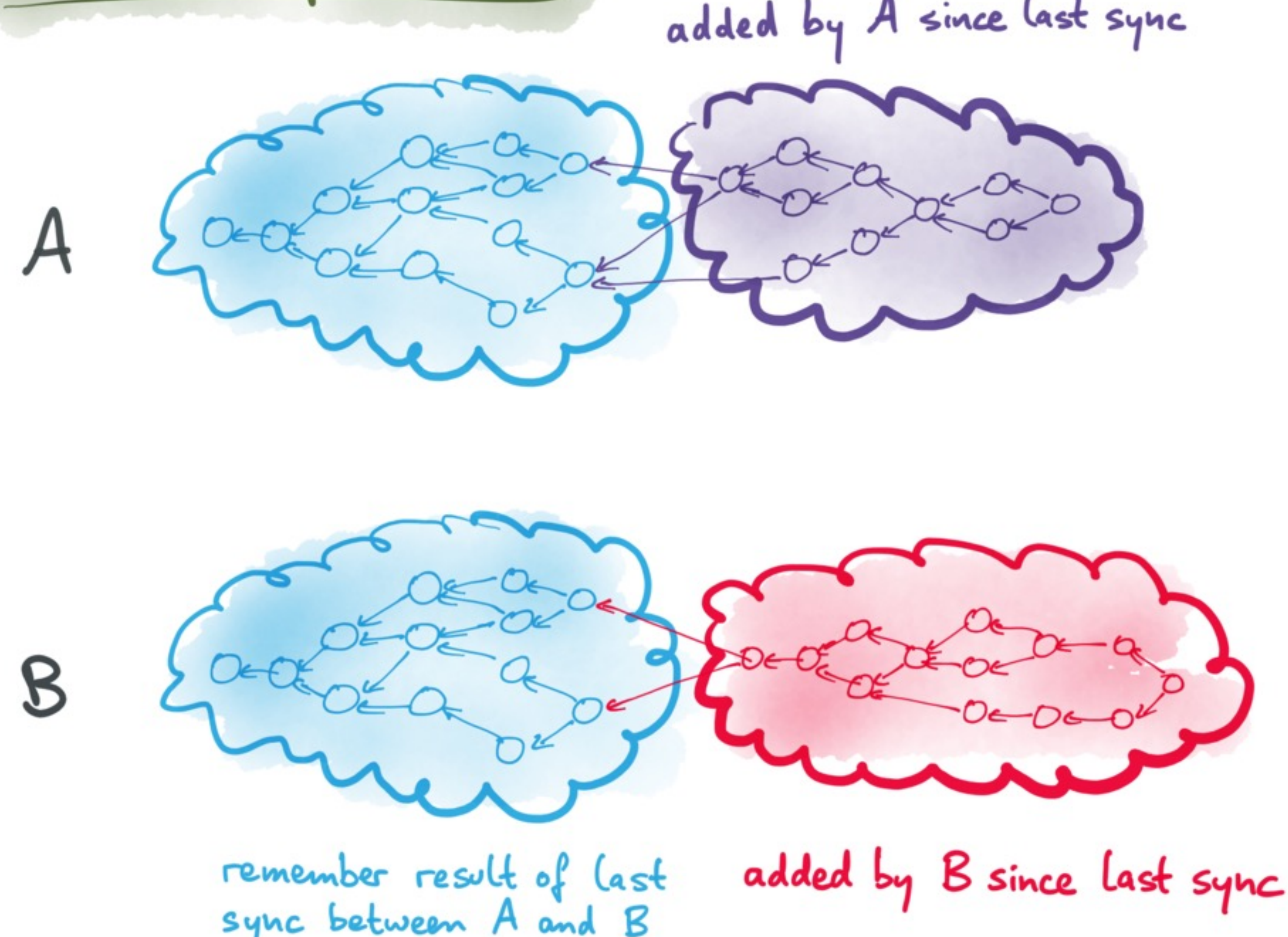


B

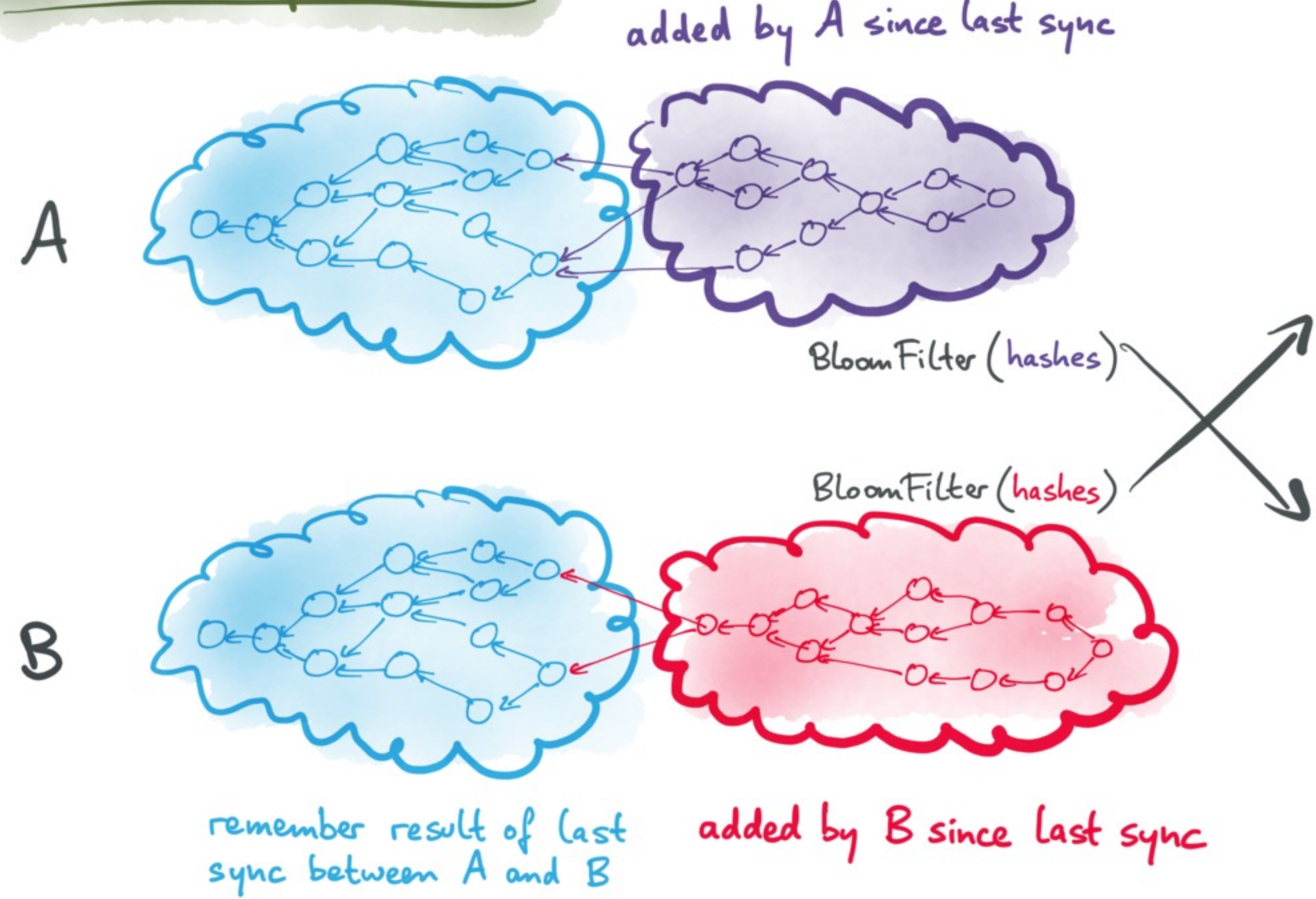


remember result of last sync between A and B

BEC replication



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

⇒ all correct replicas eventually apply u

Convergence: ← use CRDTs

Two replicas have applied same set of updates

⇒ 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 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, ...

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

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

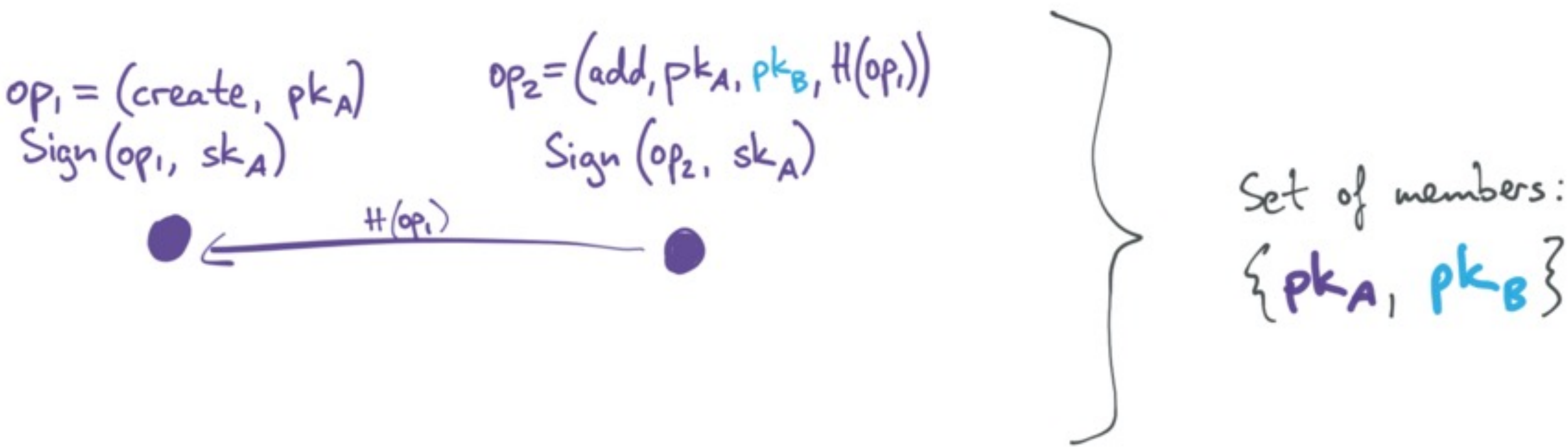
Set of members:

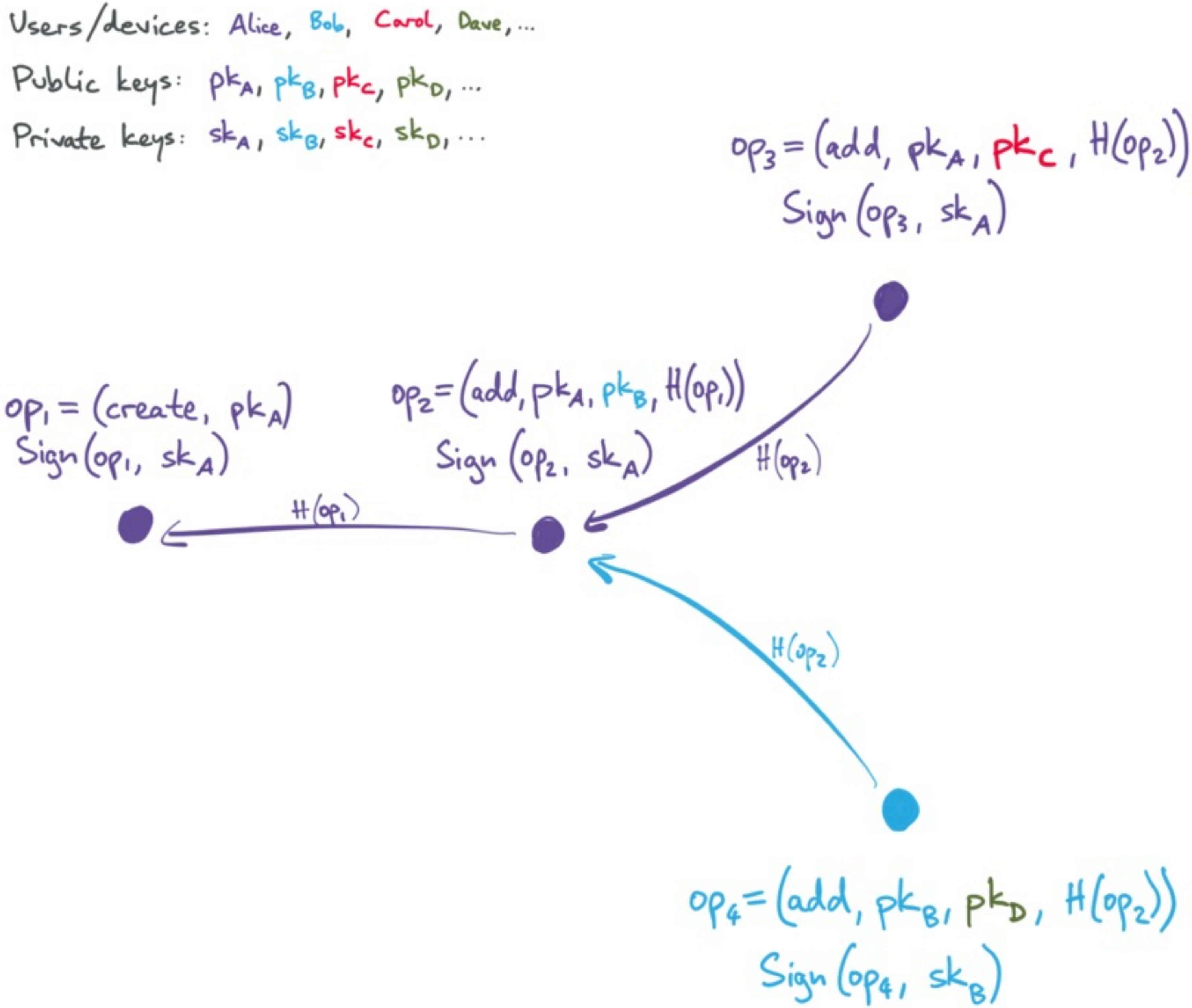
$\{pk_A\}$

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, …

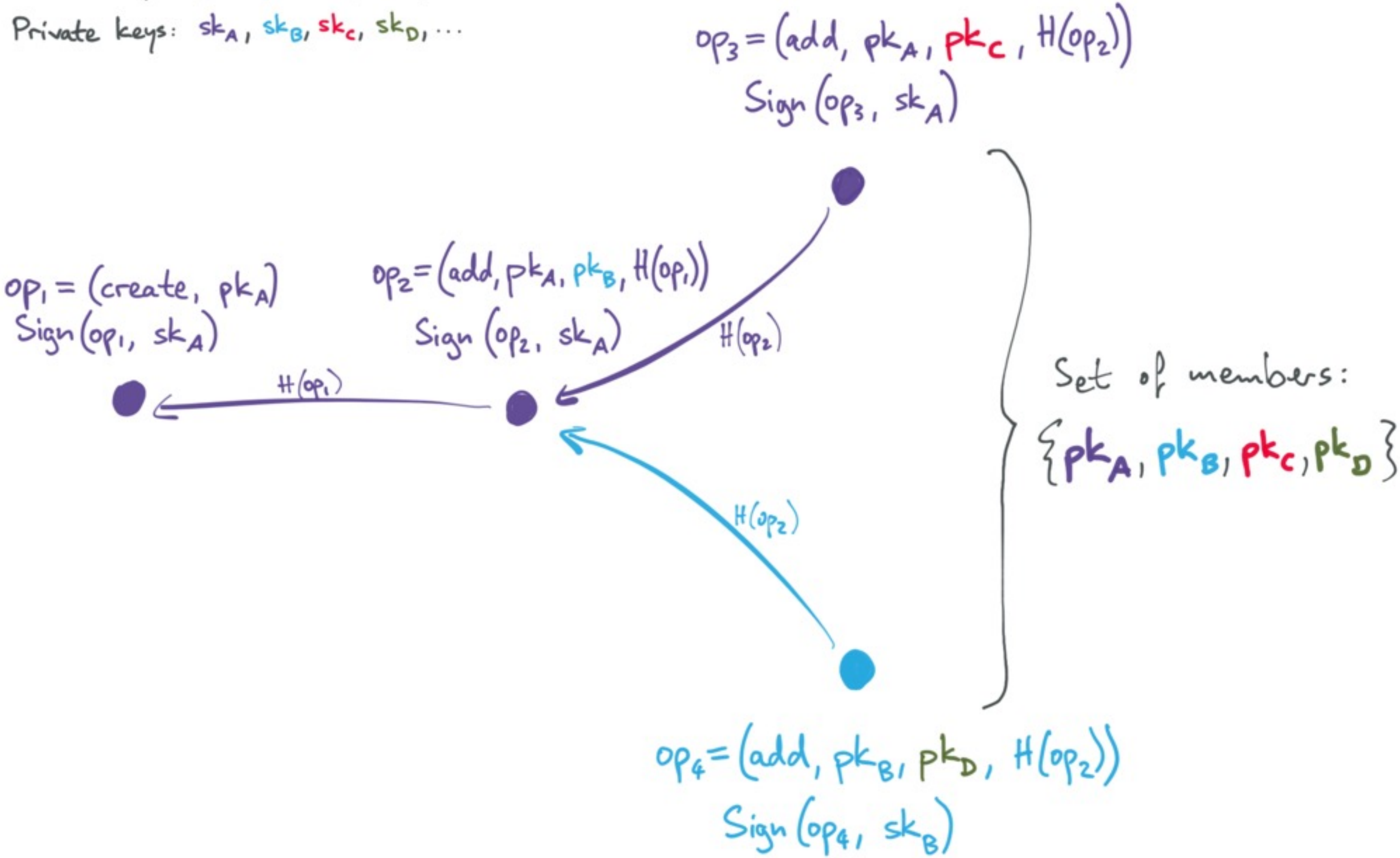


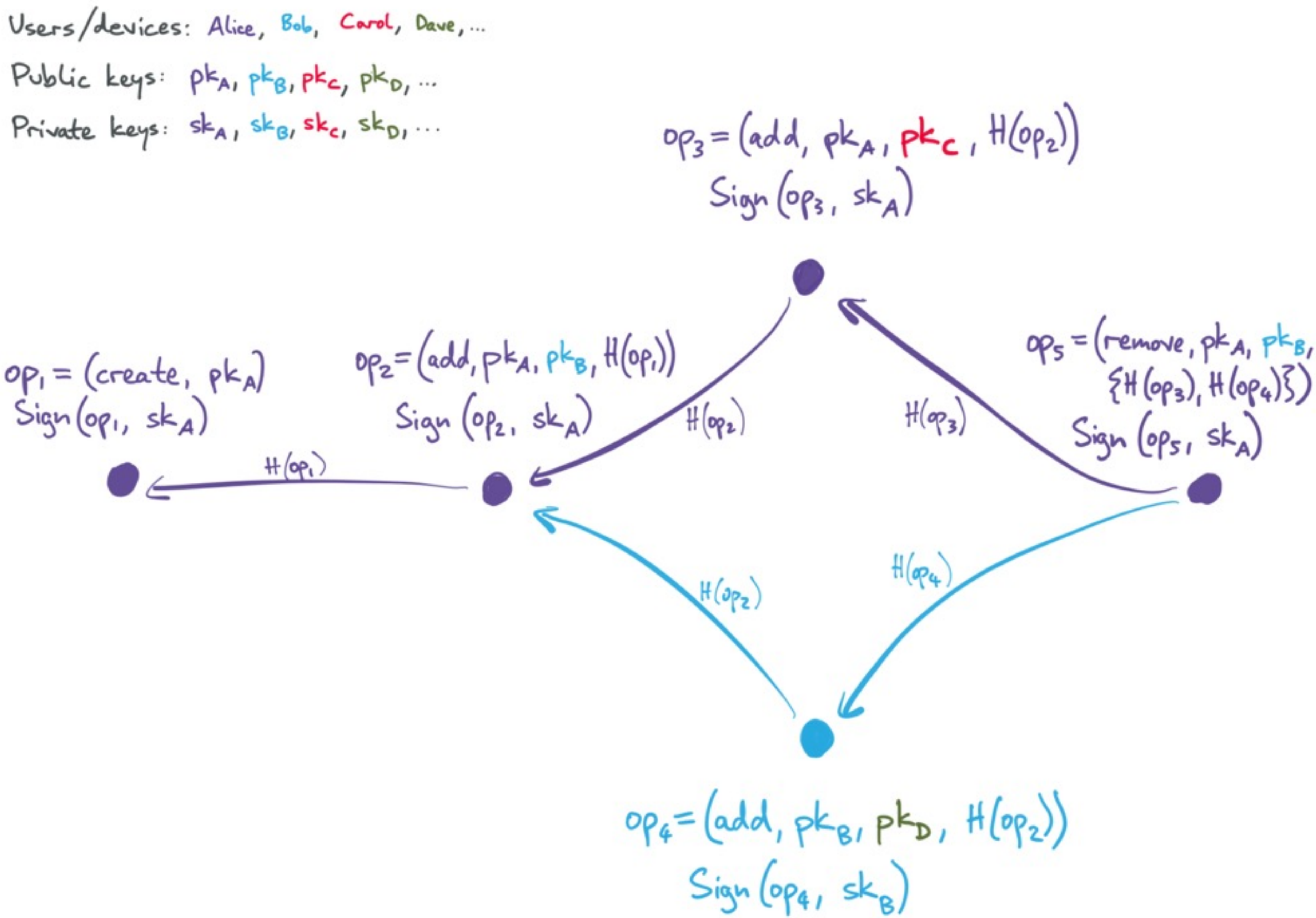

Users/devices: Alice, Bob, Carol, Dave, ...
Public keys: pk_A, pk_B, pk_C, pk_D, ...
Private keys: sk_A, sk_B, sk_C, sk_D, ...
op_3 = (add, pk_A, pk_C, H(op_2))
Sign(op_3, sk_A)
op_1 = (create, pk_A)
Sign(op_1, sk_A)
op_2 = (add, pk_A, pk_B, H(op_1))
Sign(op_2, sk_A)
H(op_2)
H(op_1)
H(op_2)
op_4 = (add, pk_B, pk_D, H(op_2))
Sign(op_4, sk_B)

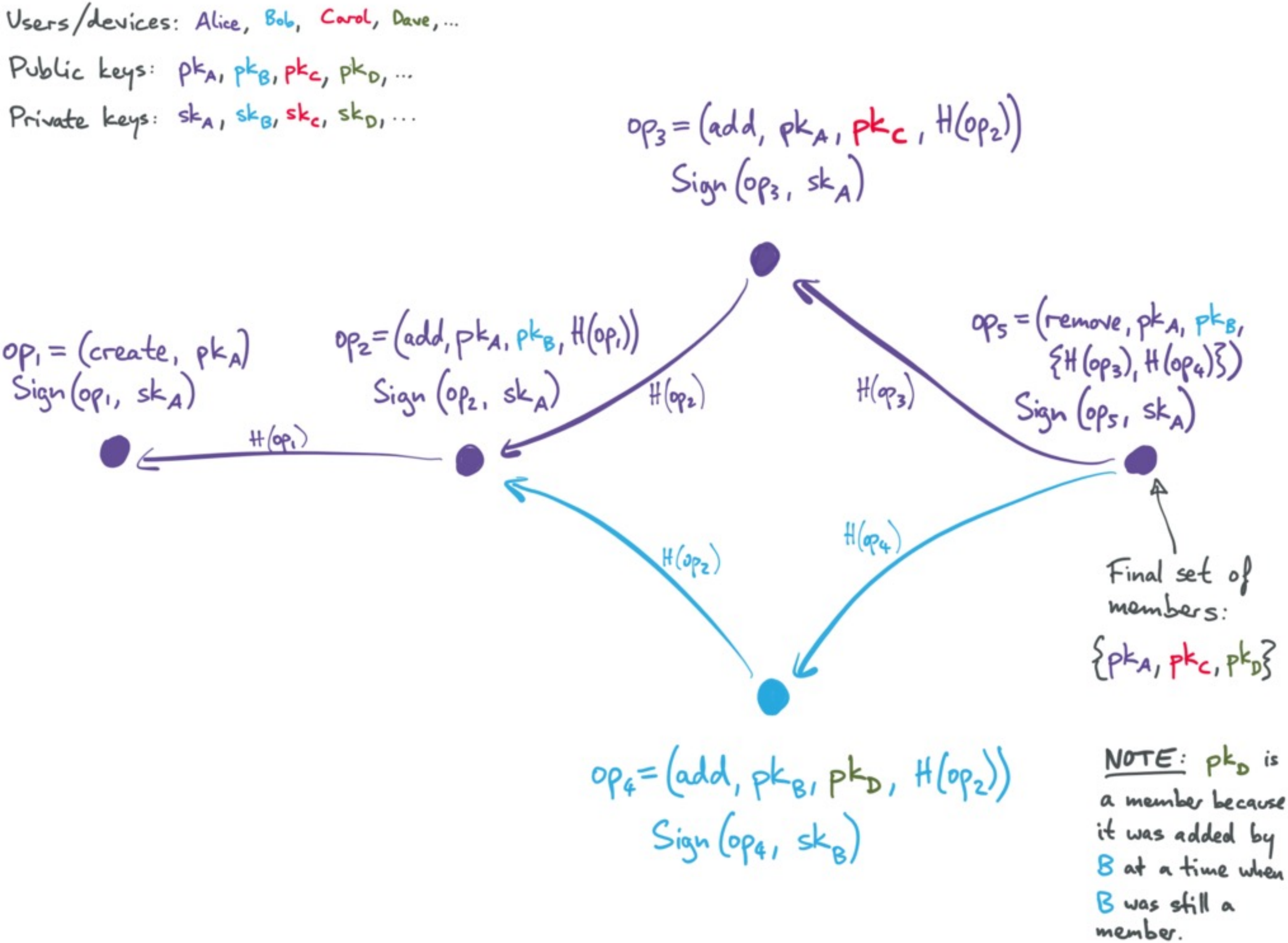
Users/devices: Alice, Bob, Carol, Dave, …

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

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




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


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.

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

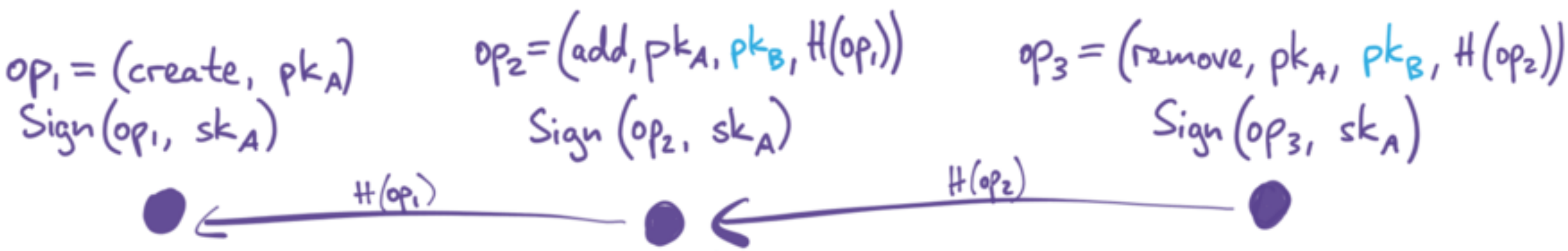
e.g. adversary stole + unlocked a team member's phone

A removed user concurrently adds a new user...

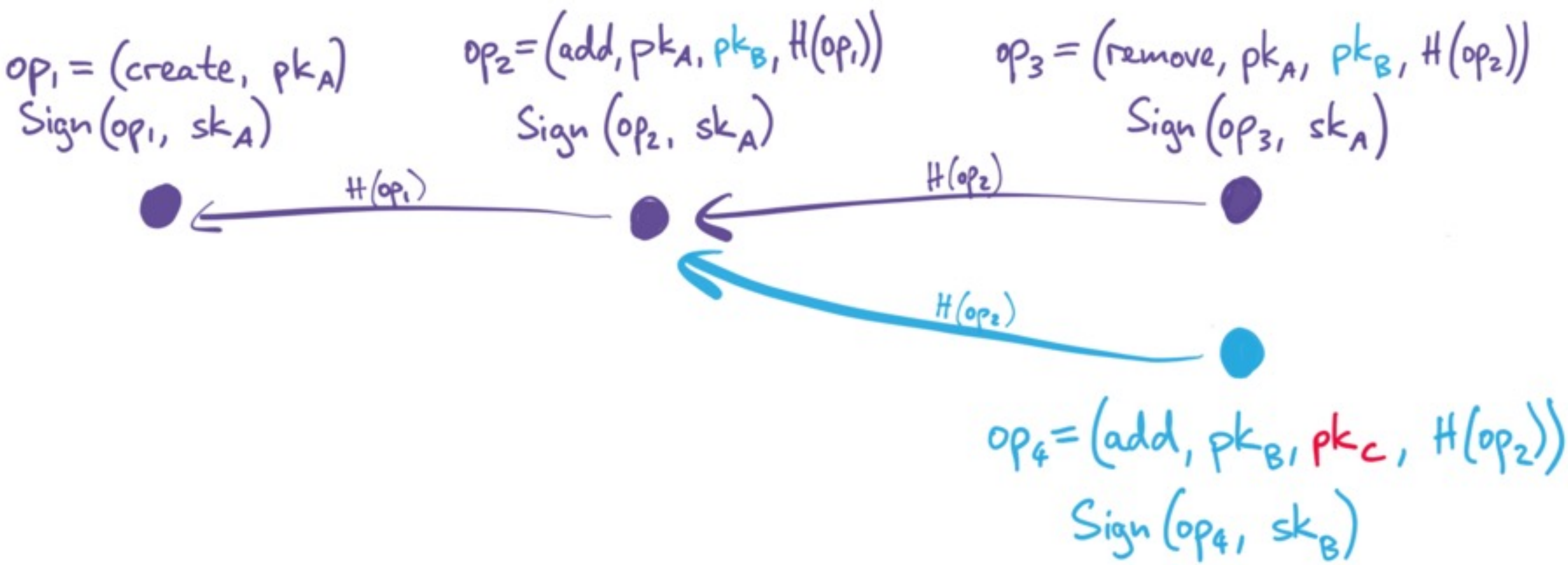
$op_1 = (\text{create}, pk_A)$
$\text{Sign}(op_1, sk_A)$

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

A removed user concurrently adds a new user...

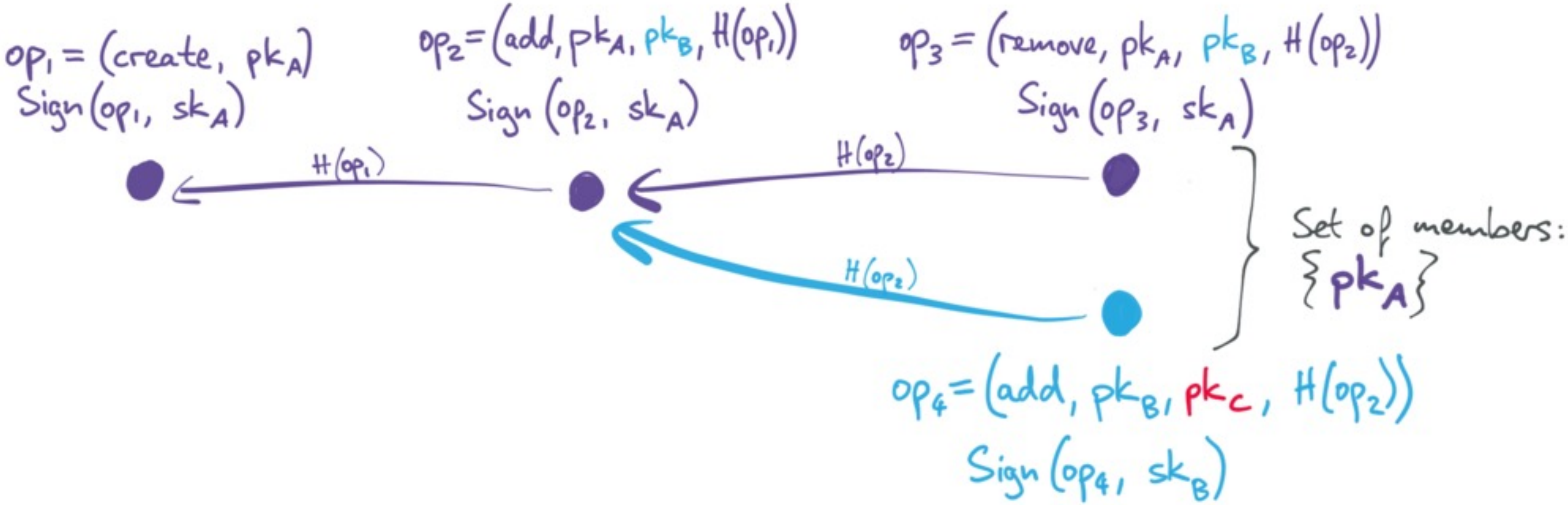


A removed user concurrently adds a new user...



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...

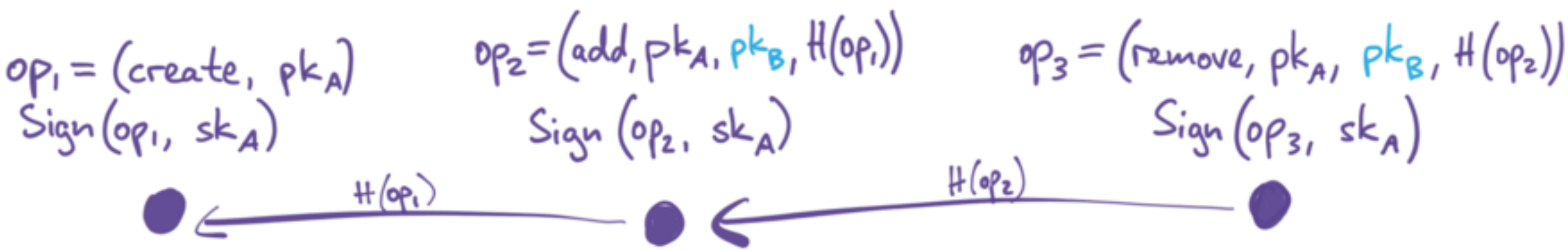
$op_1 = (\text{create}, pk_A)$

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

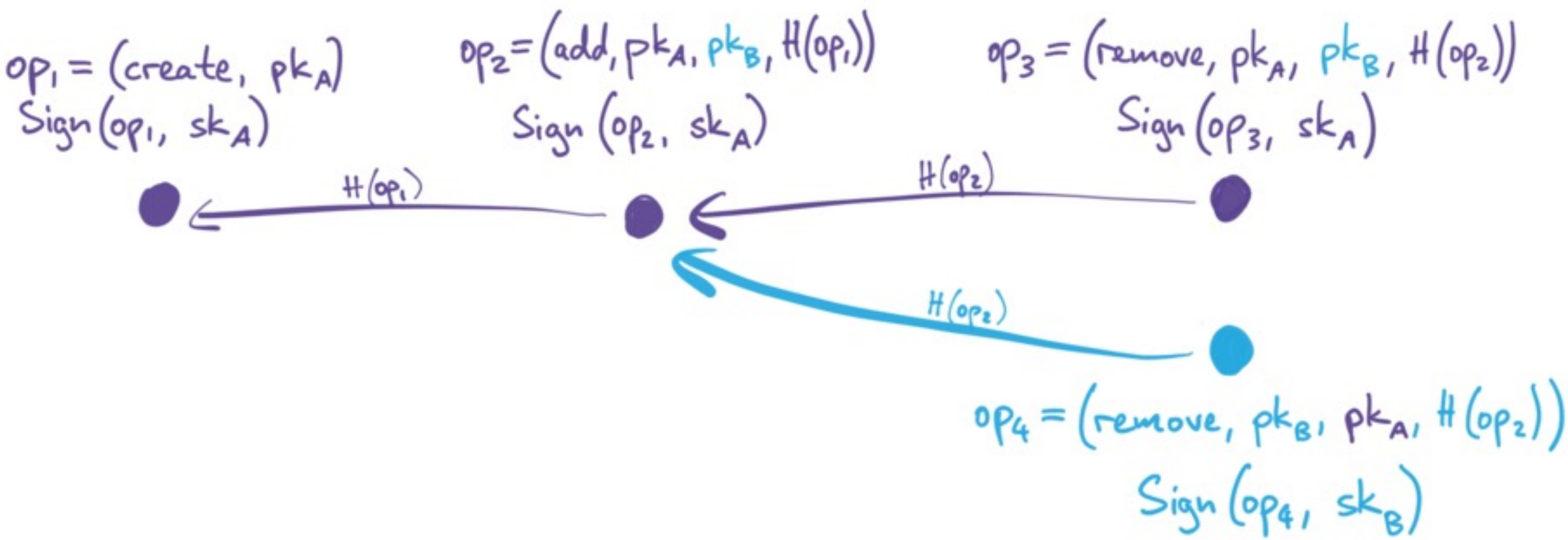
$op_2 = (\text{add}, pk_A, pk_B, H(op_1))$

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

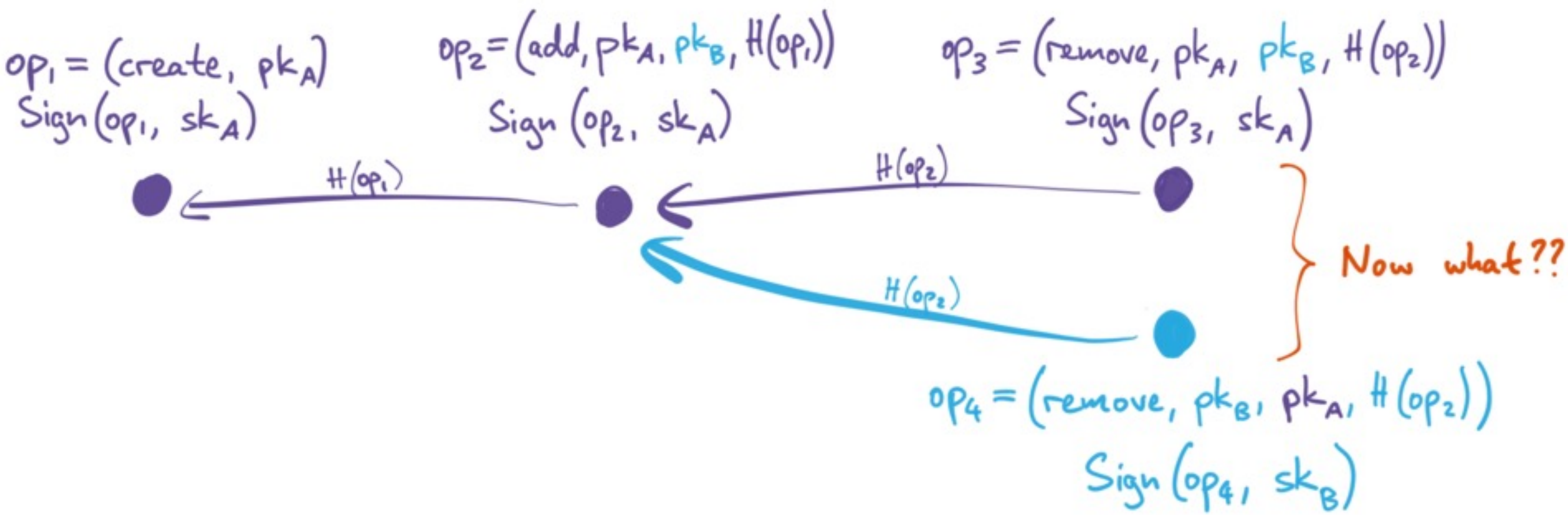
Two users concurrently remove each other…



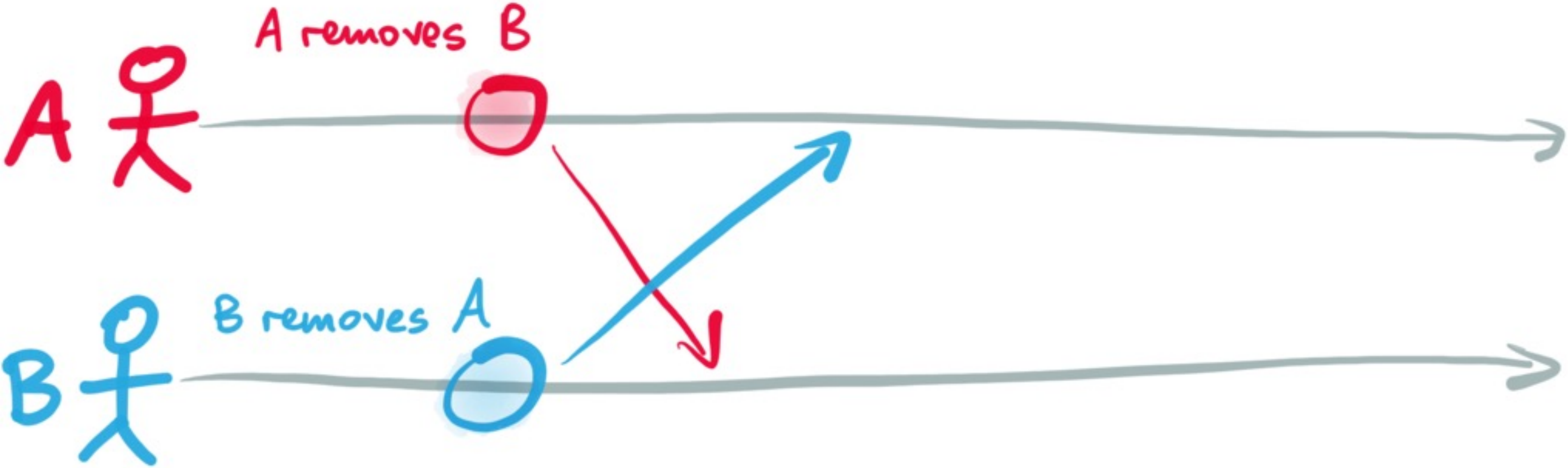
Two users concurrently remove each other...



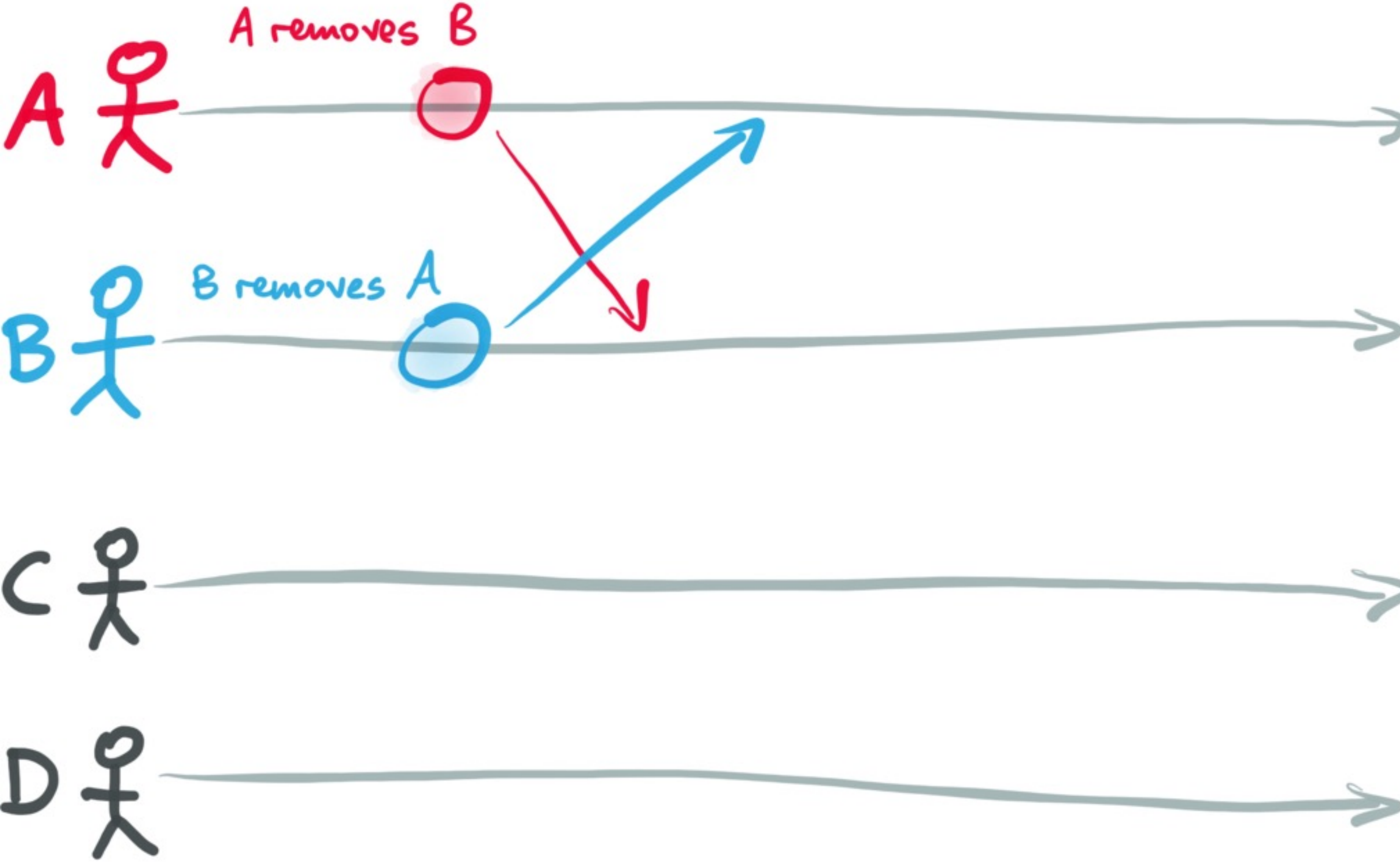
Two users concurrently remove each other...

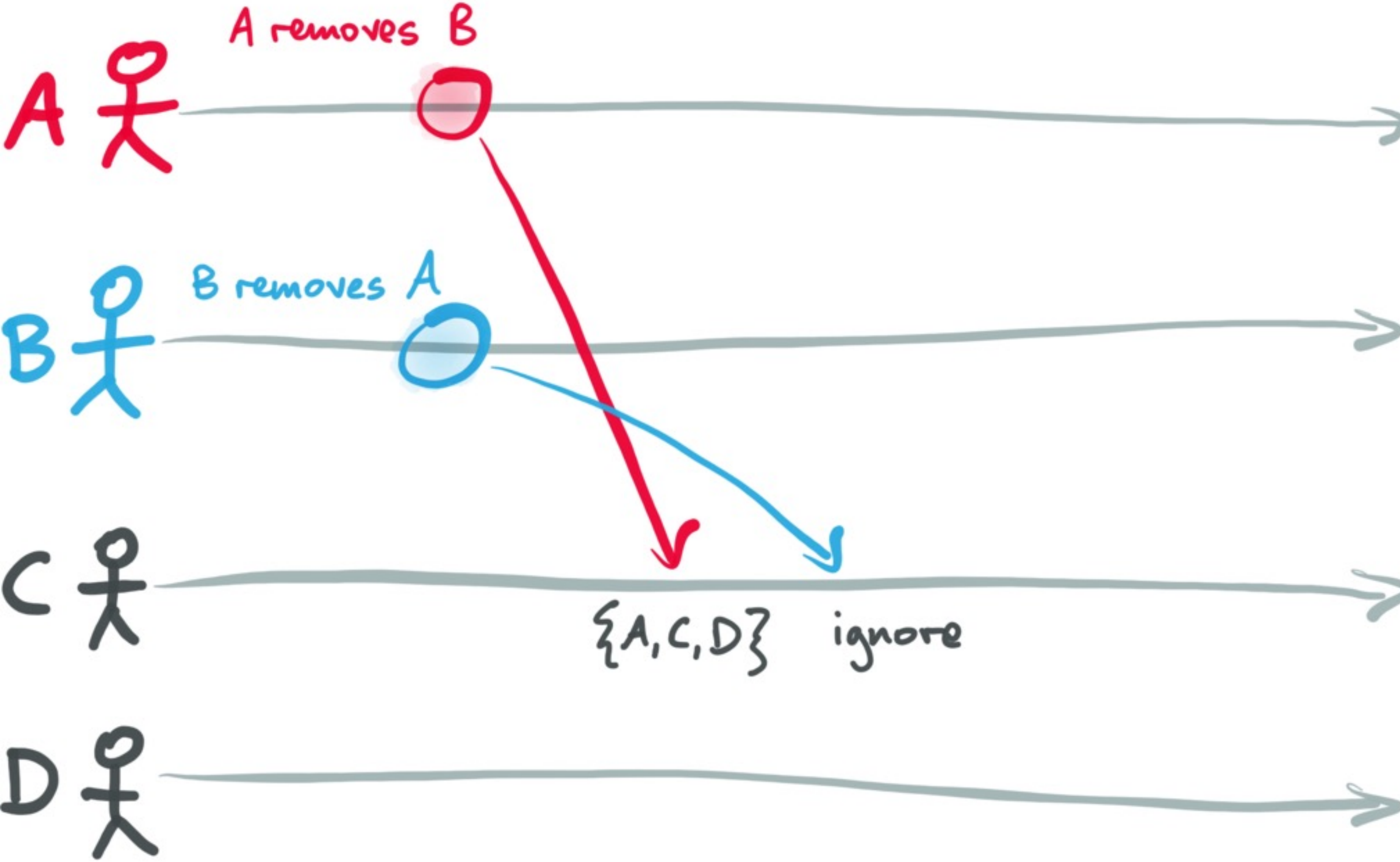


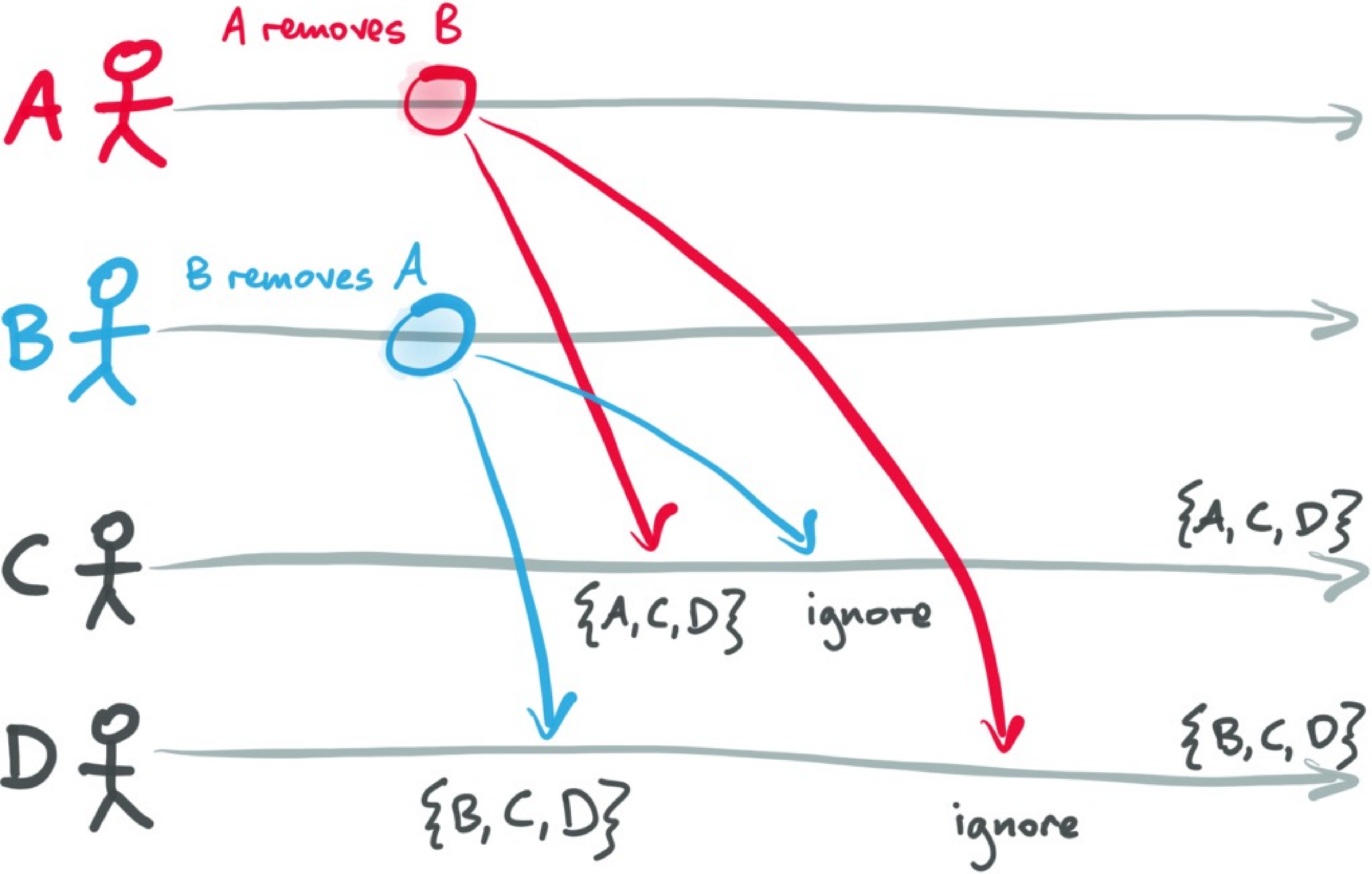

A
B



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

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?

A 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

Byzantine Eventual Consistency (BEC)

Eventual update:

One correct replica applies update u

⇒ all correct replicas eventually apply u

Convergence: ← use CRDTs

Two replicas have applied same set of updates

⇒ 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

- M. Kleppmann. Making CRDTs Byzantine Fault Tolerant. PaPoC 2022. doi:10.1145/3517209.3524042
- M. Kleppmann, H. Howard. Byzantine Eventual Consistency and the Fundamental Limits of Peer-to-Peer Databases. Preprint, 2020. https://arxiv.org/abs/2012.00472
- 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
- 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