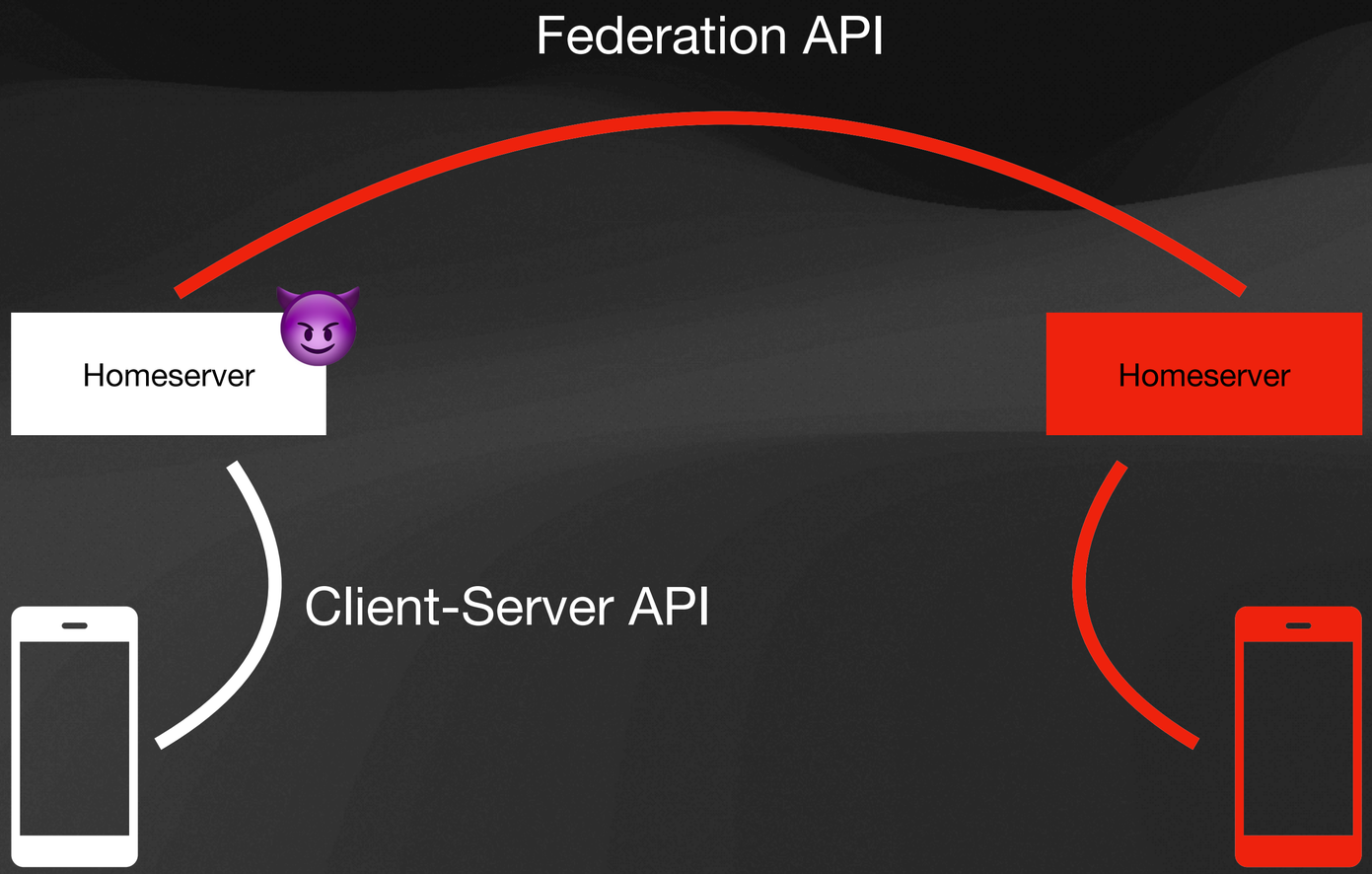
Improving the security of the federation protocol

Kegan Dougal, Staff Engineer @ Element

About me

- 👋 I’m Kegan, Staff Software Engineer @ Element
- 🔧 One of the core engineers who created Matrix
- 🌐 Working on federation protocol improvements for the past year
- 🐍 Hydra / Room version 12

Matrix Architecture

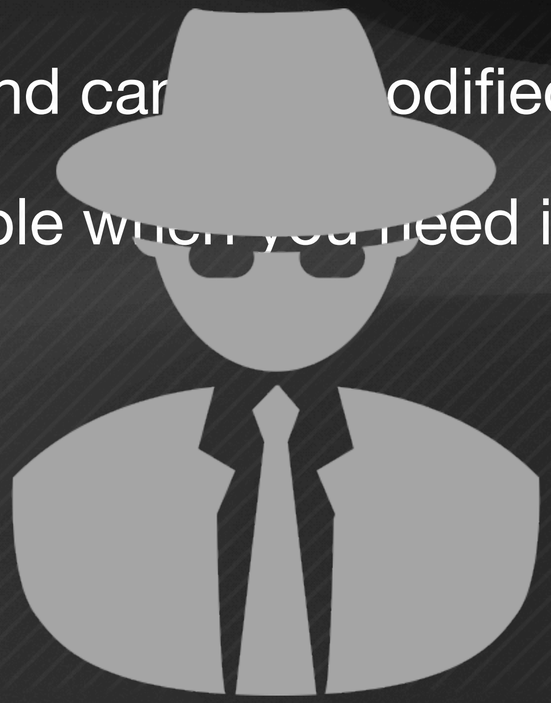


Aim of the protocol

- **Synchronise** a key-value map of room state with all other servers.
- With **no central server**.
- **Without trusting other servers** to do the right thing.

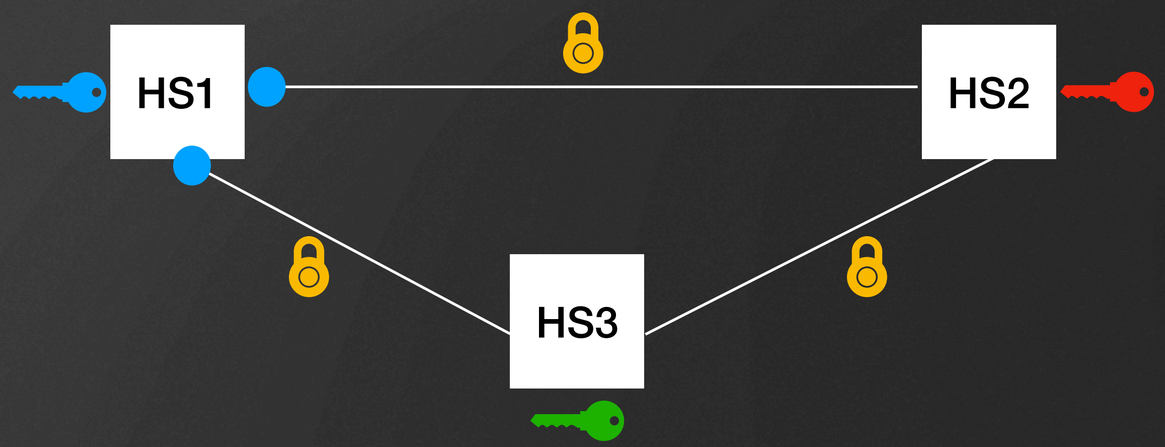
CIA

- **Confidentiality**: data only disclosed to authorised individuals.
- **Integrity**: data is accurate and can only be modified in an unauthorised way.
- **Availability**: data is accessible when you need it.



Basic Protection

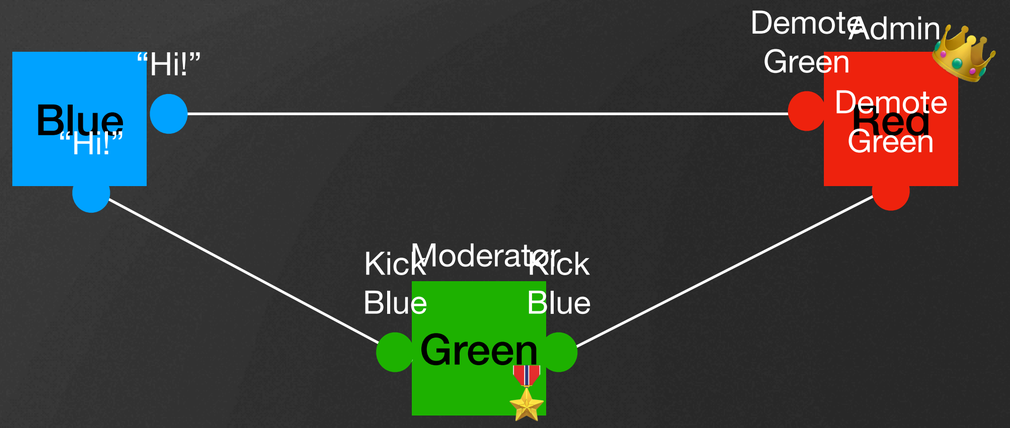
- [Confidentiality]: Encrypt communication between servers (HTTPS).
- [Integrity]: Servers have a signing key and sign everything
- [Availability]: Any server can write to the room if authorised. ???



Availability : The Hard Parts

- Opposing forces: **<u>any server</u>** can write to the room **<u>if authorised</u>**.
- What if the server loses its authorisation *whilst writing*?

Room state has diverged!

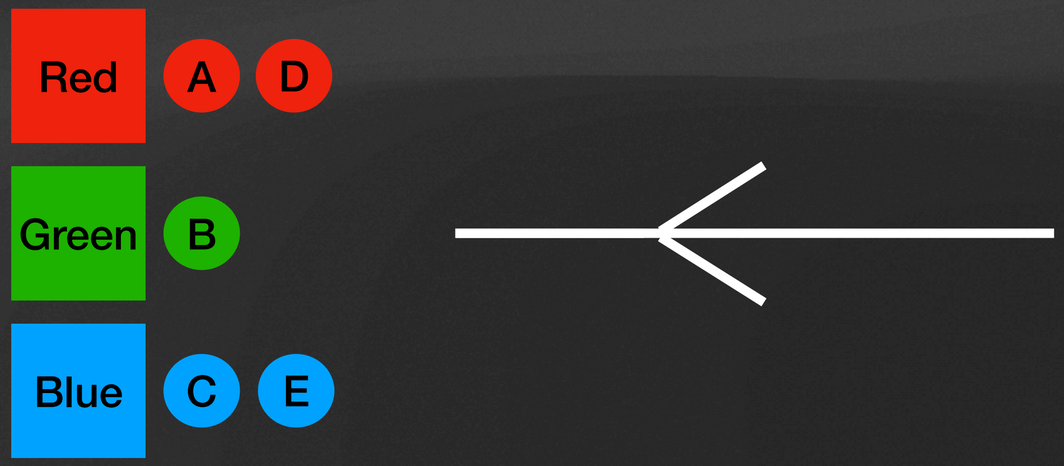


	Blue	Red	Green
1	"Hi"	Demote Green	Kick Blue
2	Demote Green	"Hi"	✗
3	Kick ✗ Blue	Kick ✗ Blue	Demote Green

Availability : The Hard Parts

It’s all about the ordering

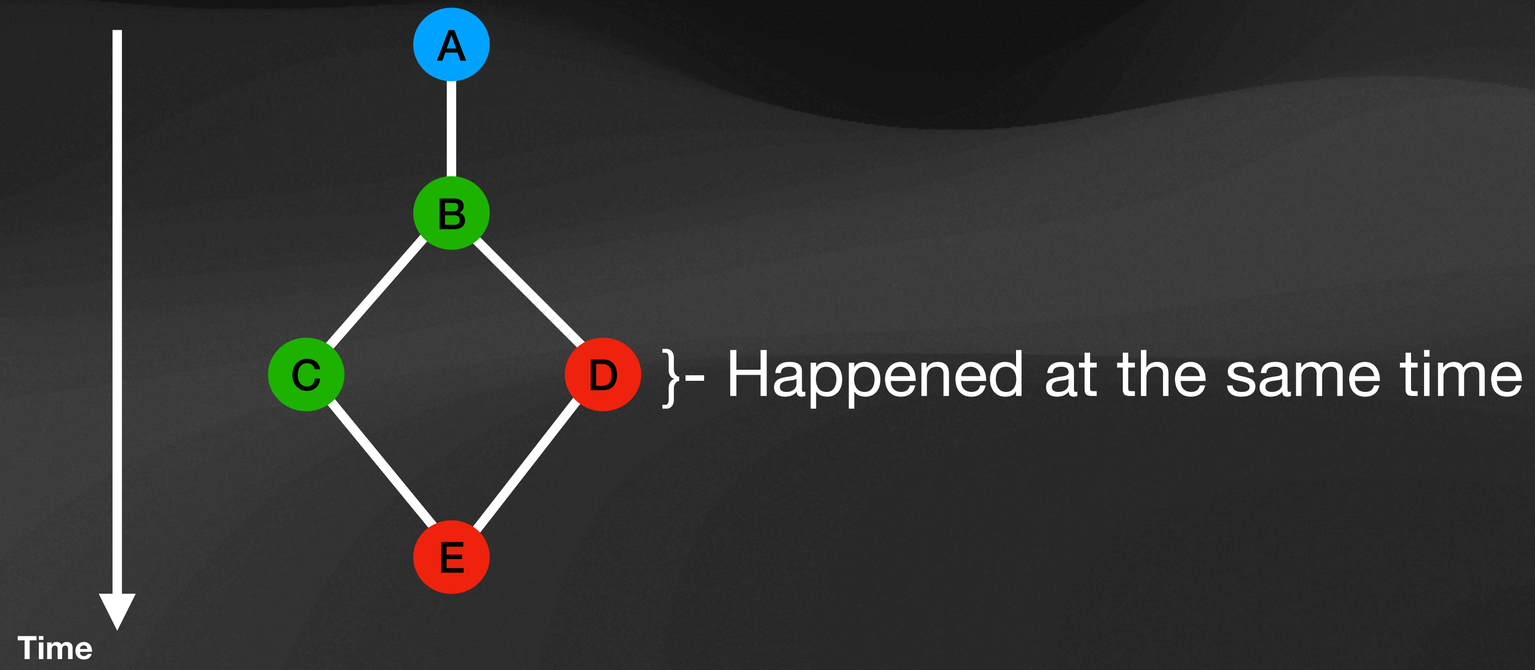
- Whether servers are authorised depends on the order they see operations.
- To guarantee we synchronise room state, we need to ensure we always see things in the same order, even if they arrive at different times.



It’s a DAG!

Availability : The Hard Parts

It’s all about the ordering



The algorithm determines the order of concurrent events

Consensus Algorithms!

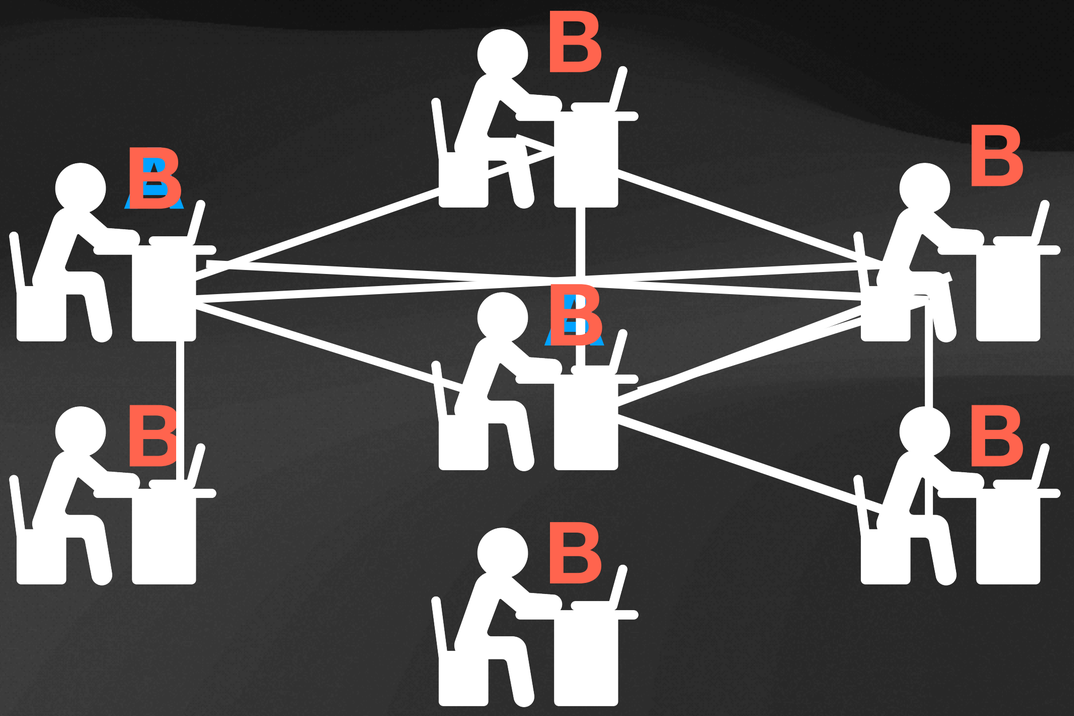

These aren't the droids you're looking for

Why not consensus?

- Consensus algorithms e.g Raft, PBFT:
  - Sacrifices Liveness for Safety
  - With **f** faulty servers, requires **3f+1** total nodes else Safety is lost.
- Convergence algorithms e.g CRDTs, Matrix:
  - Sacrifices Safety for Liveness
  - Tolerates unlimited numbers of faulty servers.

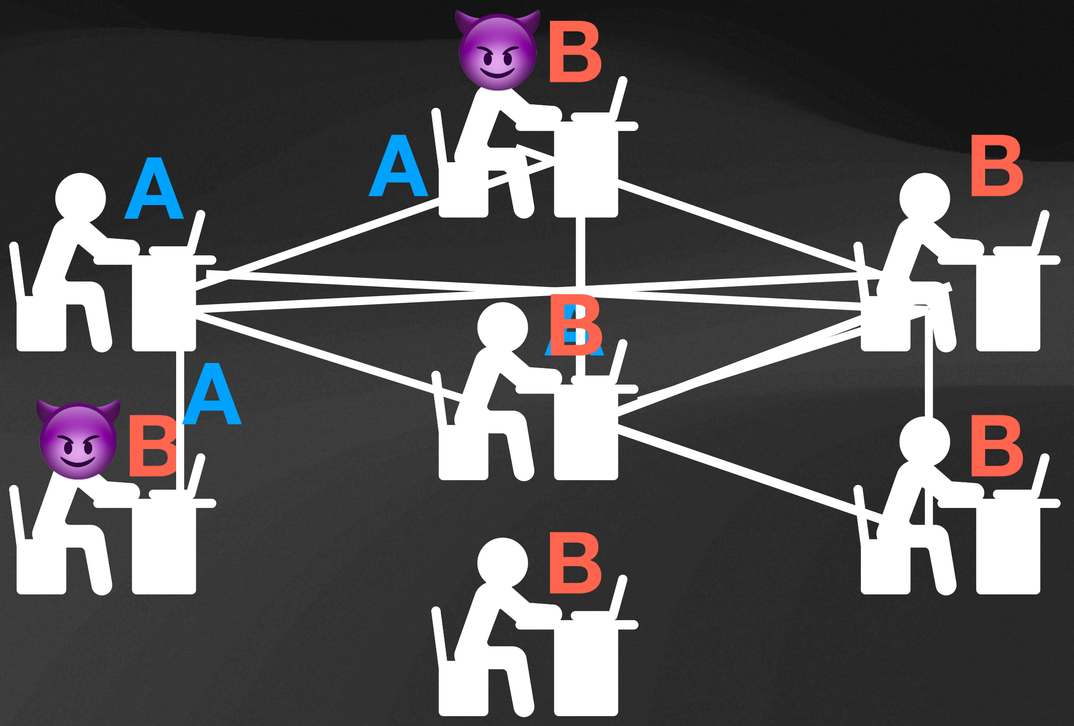
Consensus

Wisdom of the crowds



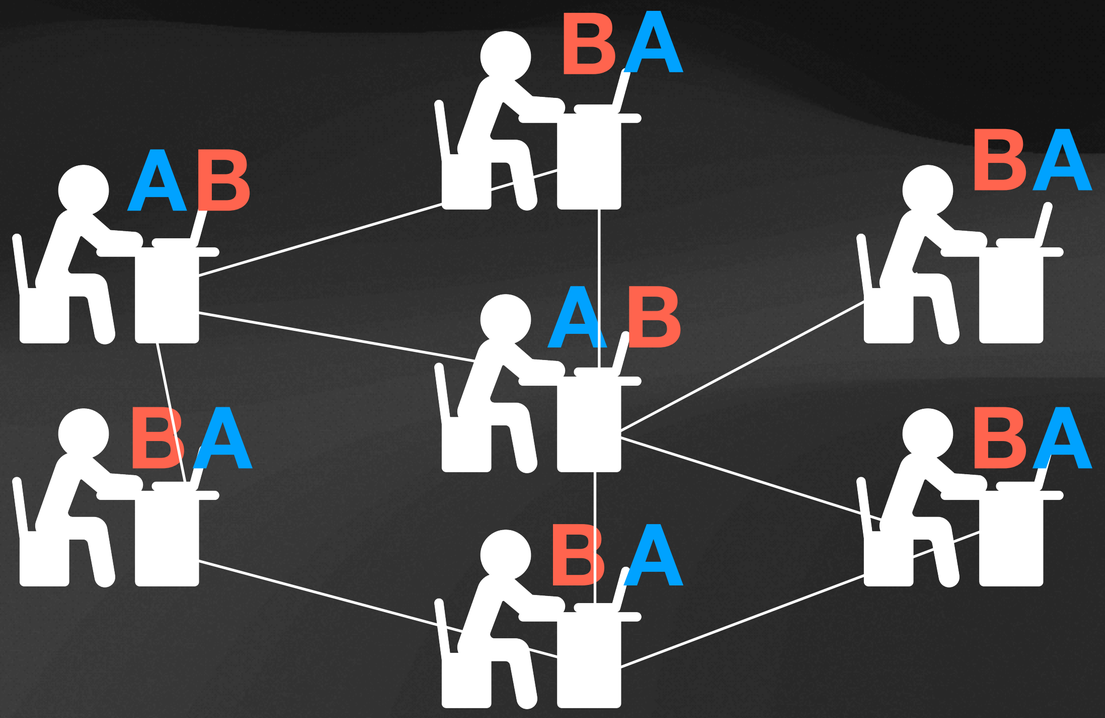
Consensus

Wisdom of the crowds



Convergence

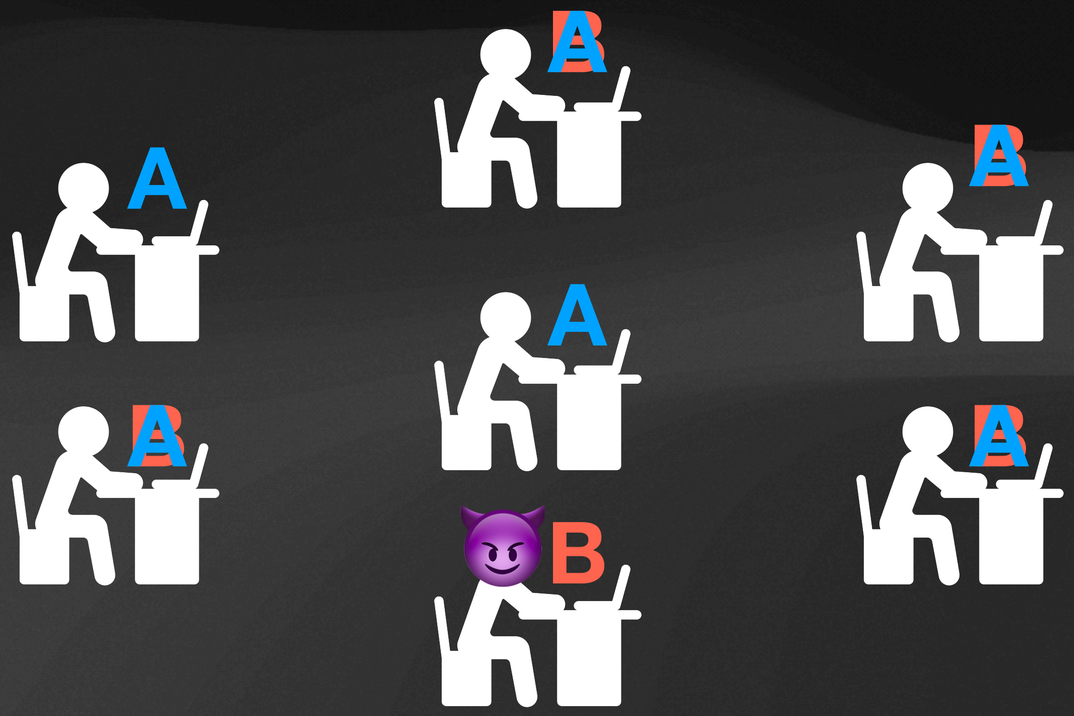
Follow the algorithm



Always choose the lowest alphabetical answer

Convergence

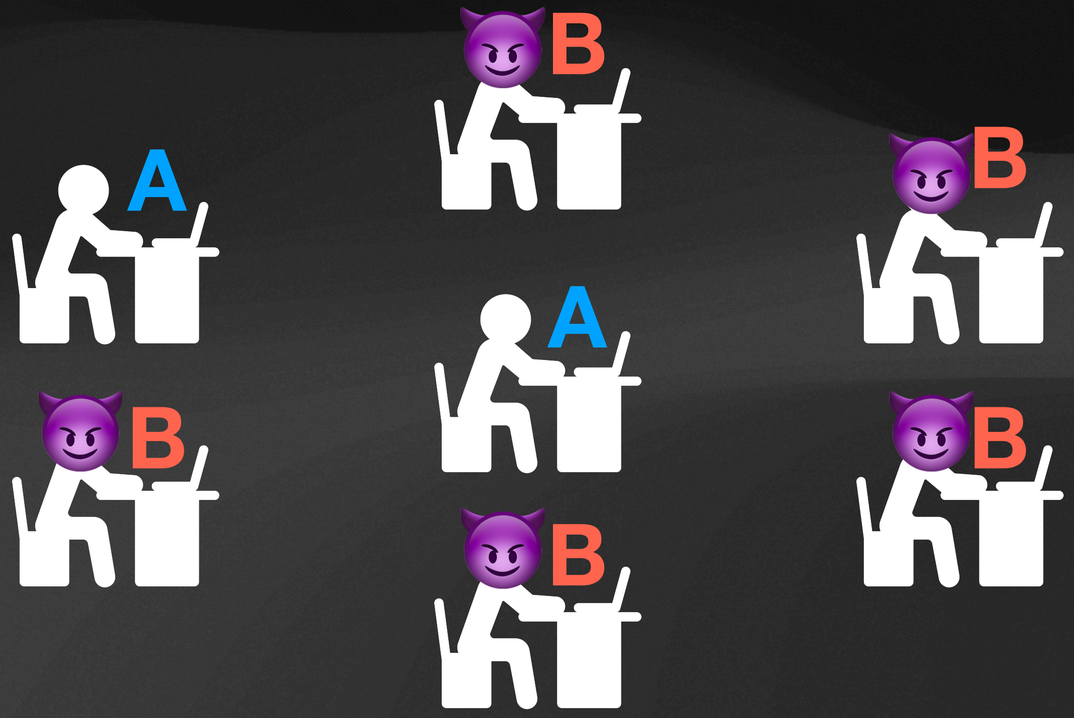
Follow the algorithm



Always choose the lowest alphabetical answer

Convergence

Follow the algorithm



Always choose the lowest alphabetical answer

How does Matrix authorise events?

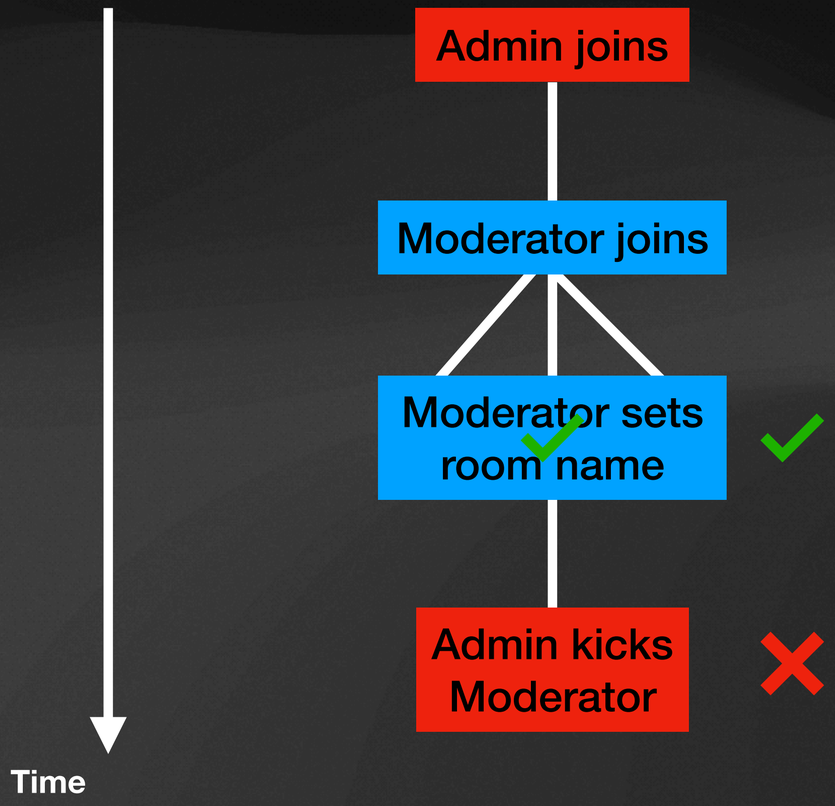
- Check the data is **valid**
- Check it's **signed**
- Check the **hashes**, redact if fails.
- Passes auth checks at **`auth_events`**
- Passes auth checks at **`prev_events`**
- Passes auth checks at the **current state**

How does Matrix authorise events?

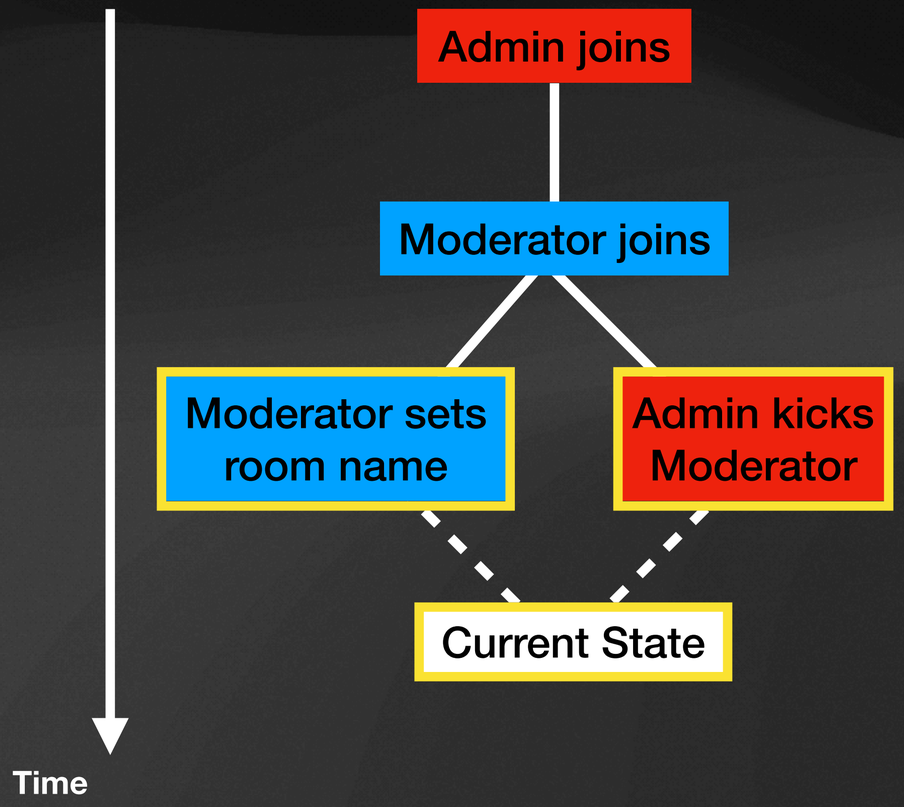
- Passes auth checks at **`auth_events`**
  - Provides positive authorisation. (“I am authorised because…”)
  - Quick to perform.
  - Catches obvious malice like random users trying to join invite-only rooms.
- Passes auth checks at **`prev_events`**
  - Checks the event was authorised *at that point in time.*
  - *Slow.* Need to calculate the room state before the event, which may do state resolution.
  - Catches omissions from `auth_events`.

How does Matrix authorise events?

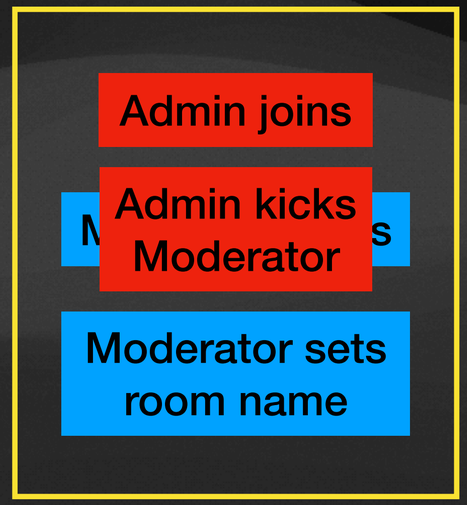
..at that point in time



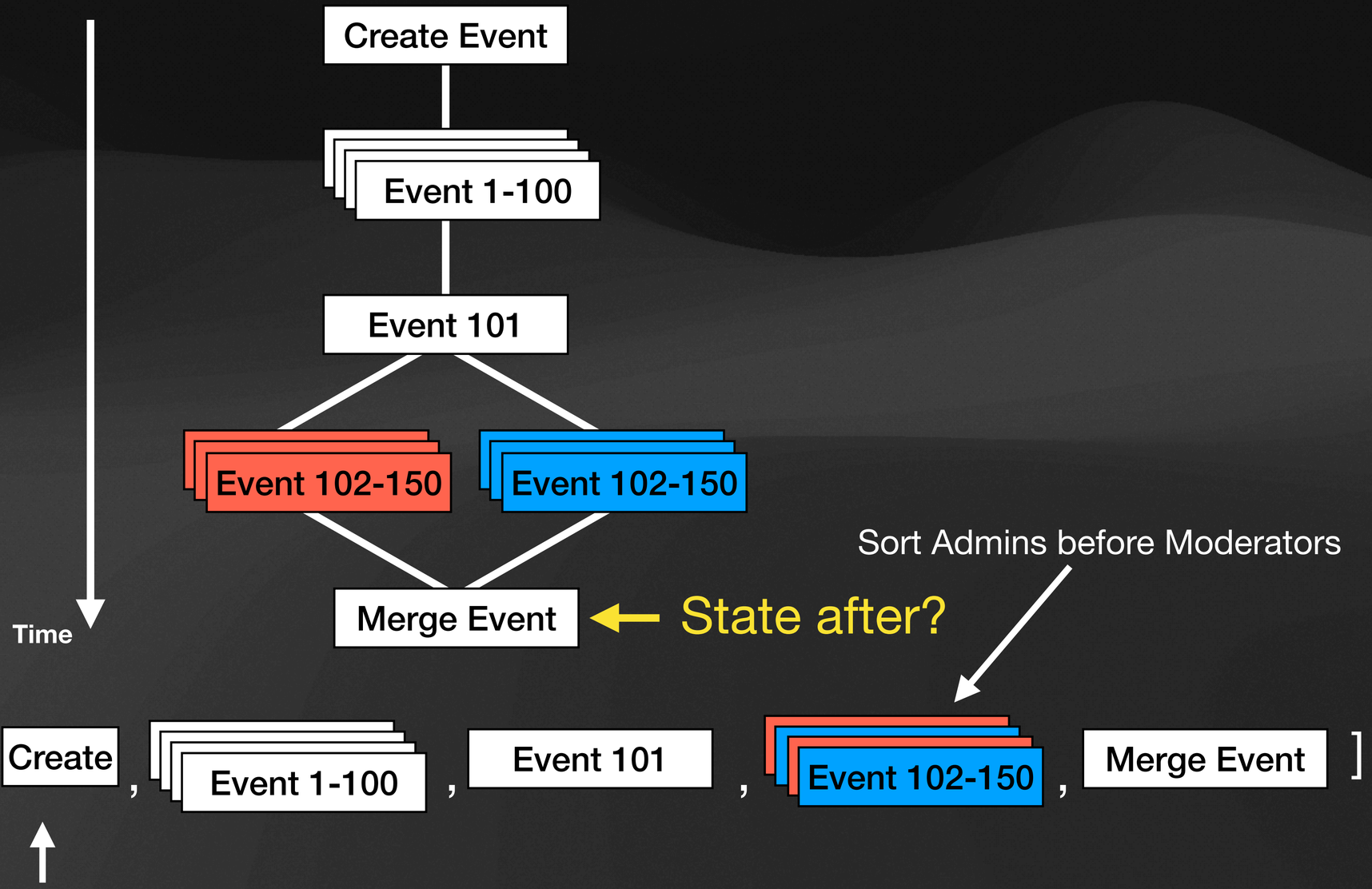
What is the current state?



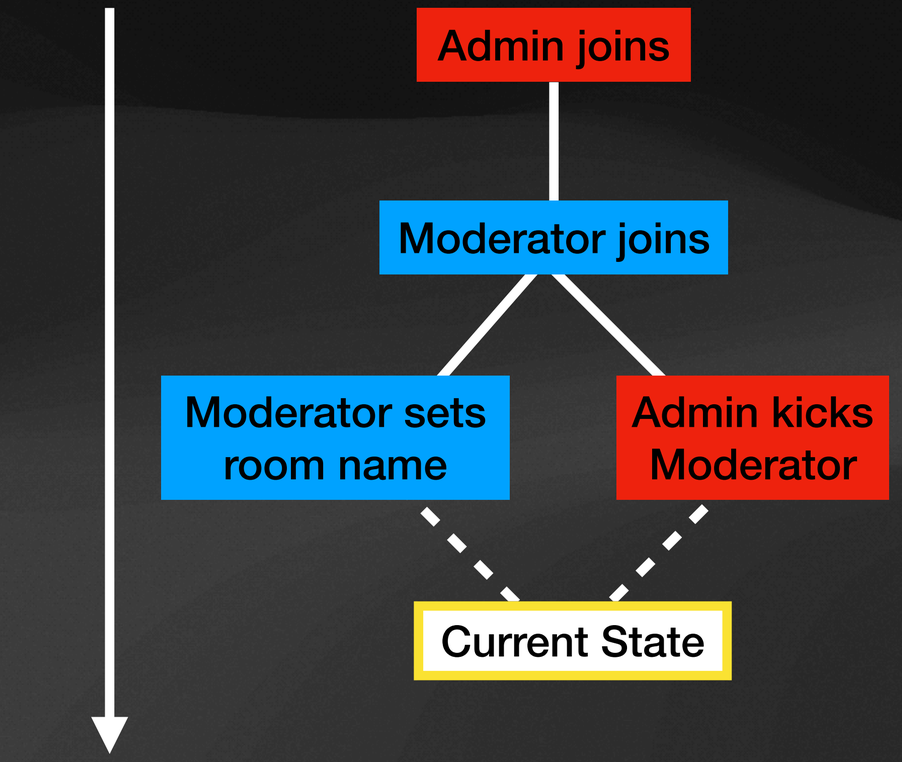
Room State After



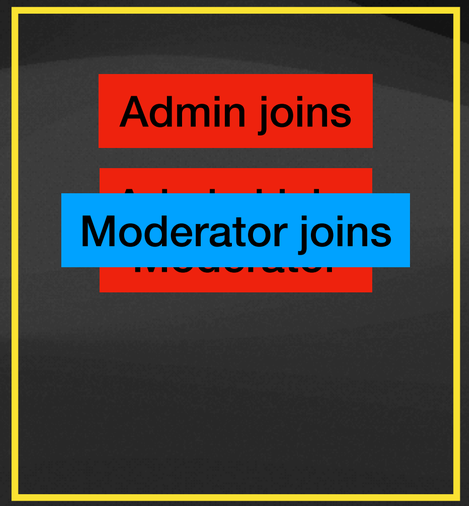
How to merge forks



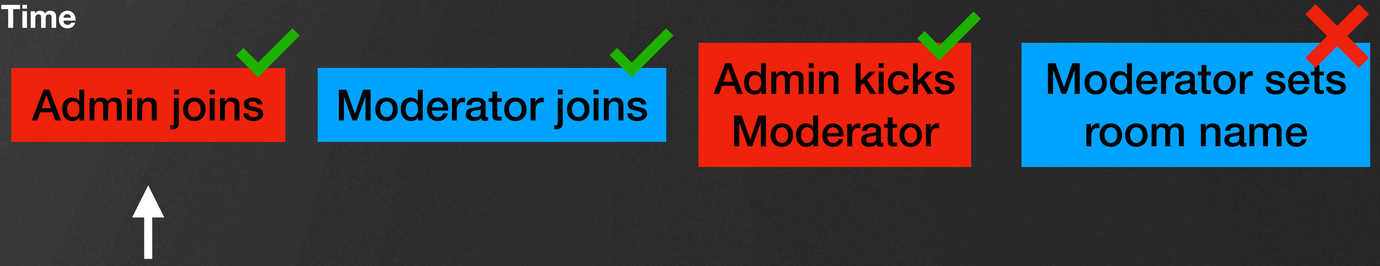
How to merge forks



Current Room State



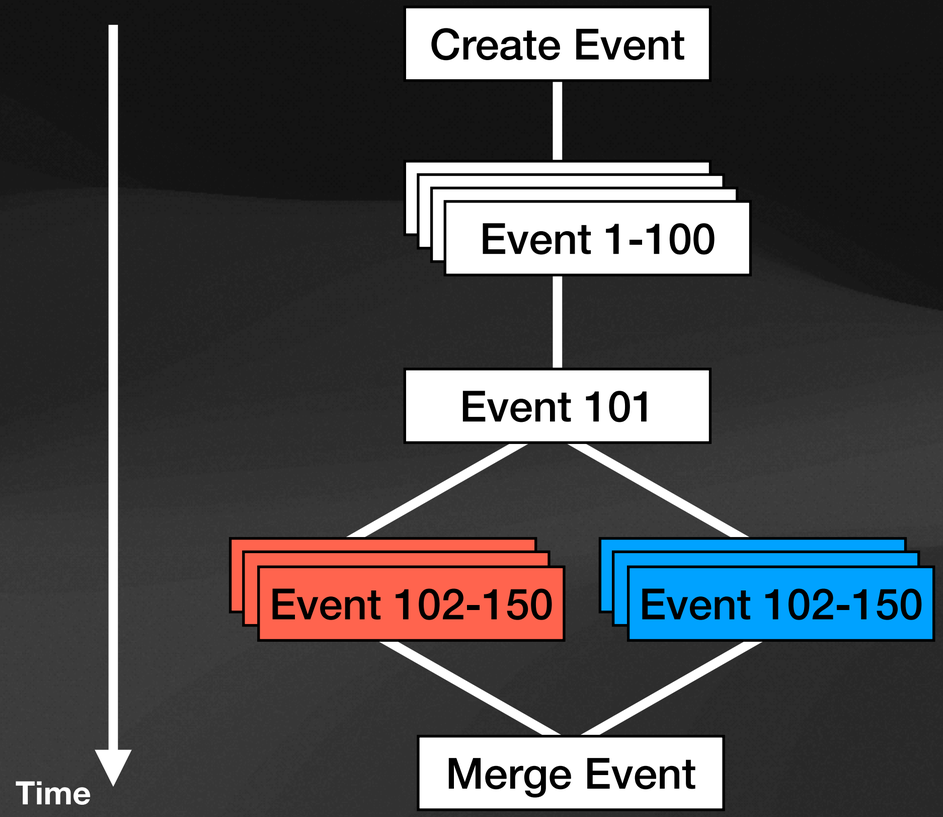
Execution Order:



Simple but slow

- Execution time increases linearly with the number of events in the room.
- Lots of redundant work.
- Need every event in the graph to perform the replay.
- Can we do better?

How to merge forks: redux

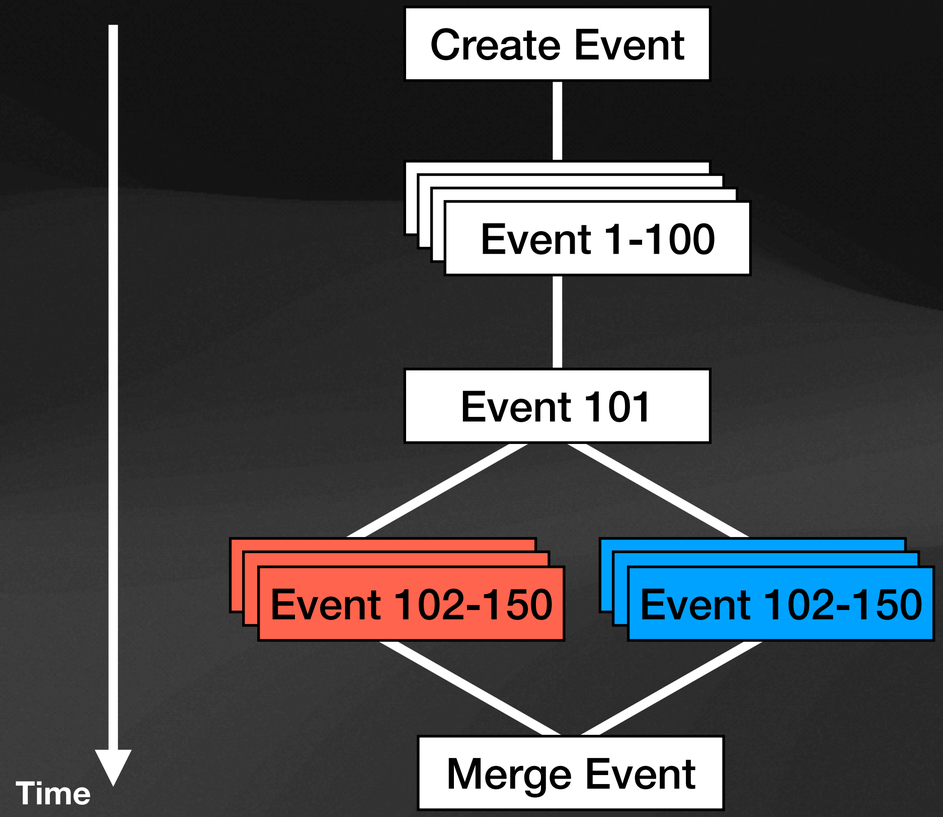


Execution Order: [Create , Event 1-100 , Event 101 , Event 102-150 , Merge Event]

Events we agree on

Events we don’t agree on

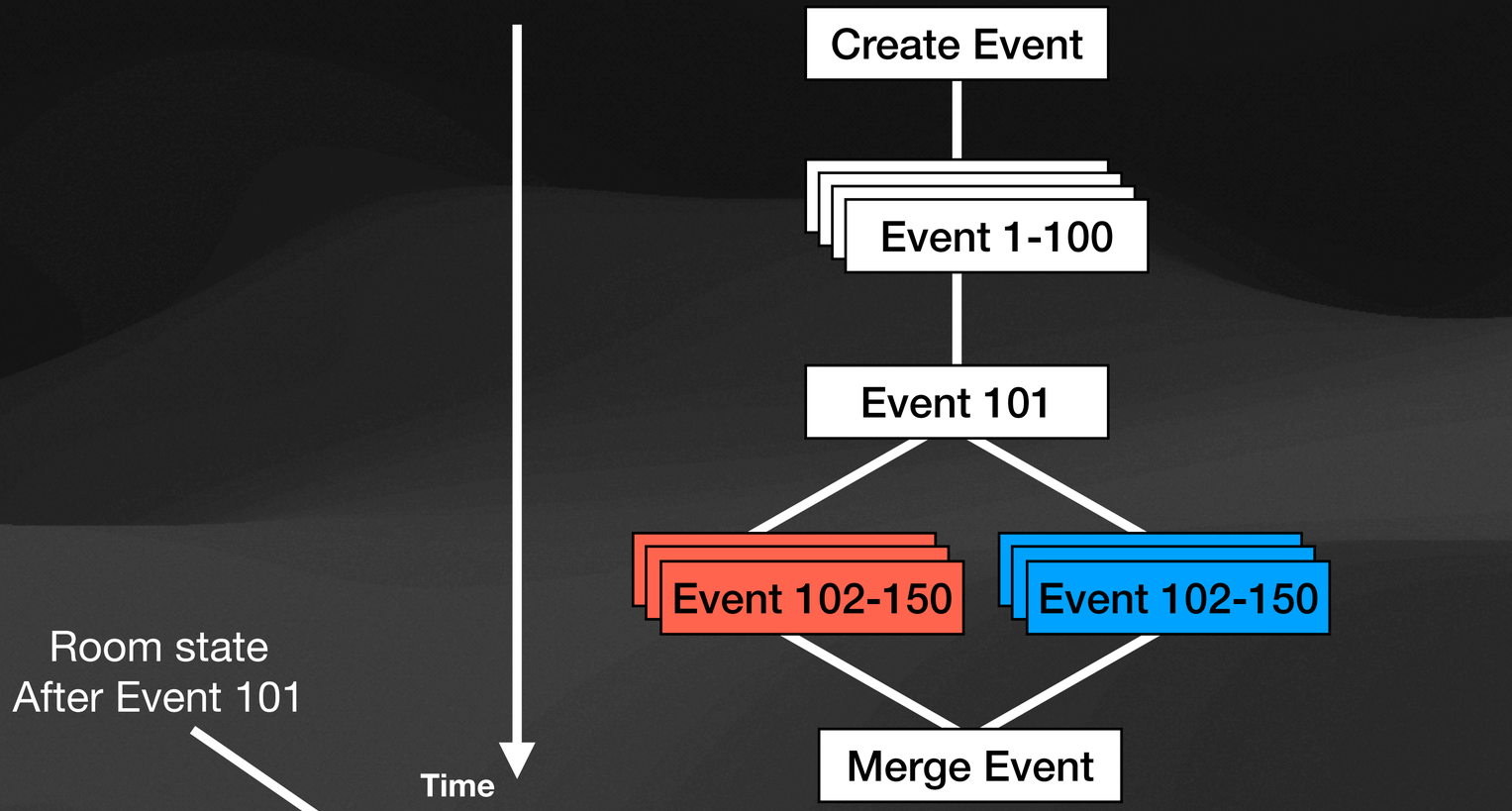
How to merge forks: redux



Execution Order: [Event 101 Snapshot , Event 102-150 , Merge Event]

Events we don’t agree on

How to merge forks: redux

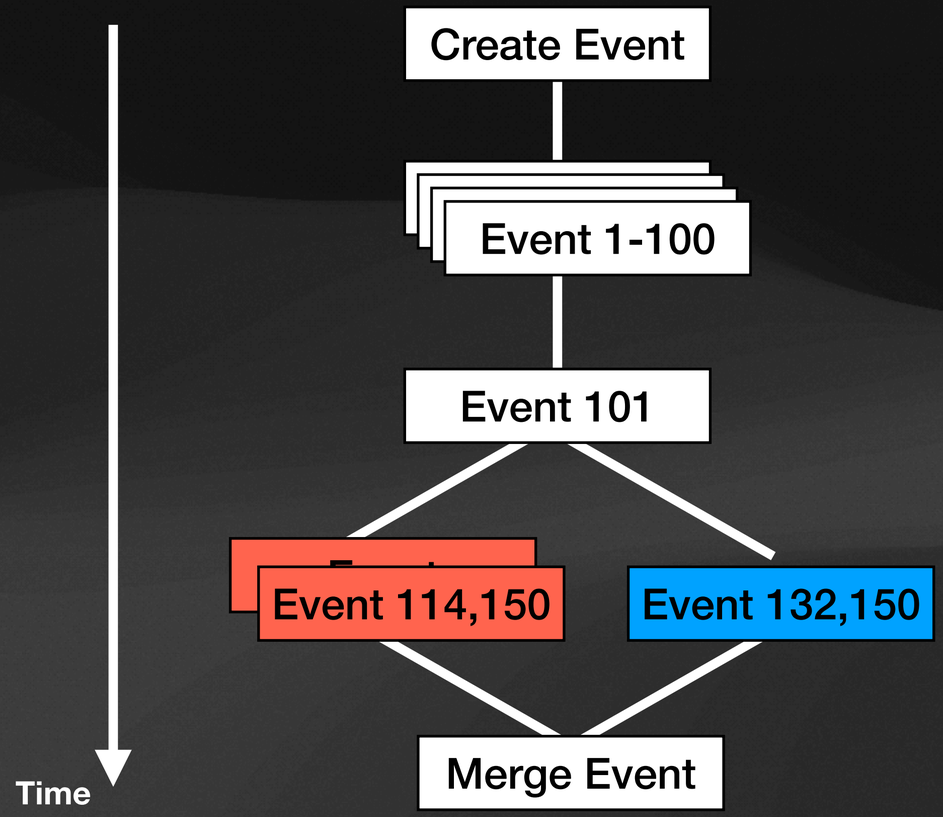


Execution Order: [Event 101 Snapshot , Event 102-150 , Merge Event]

`snapshot([]Event)-> Map<K, Event>`

Events we don’t agree on

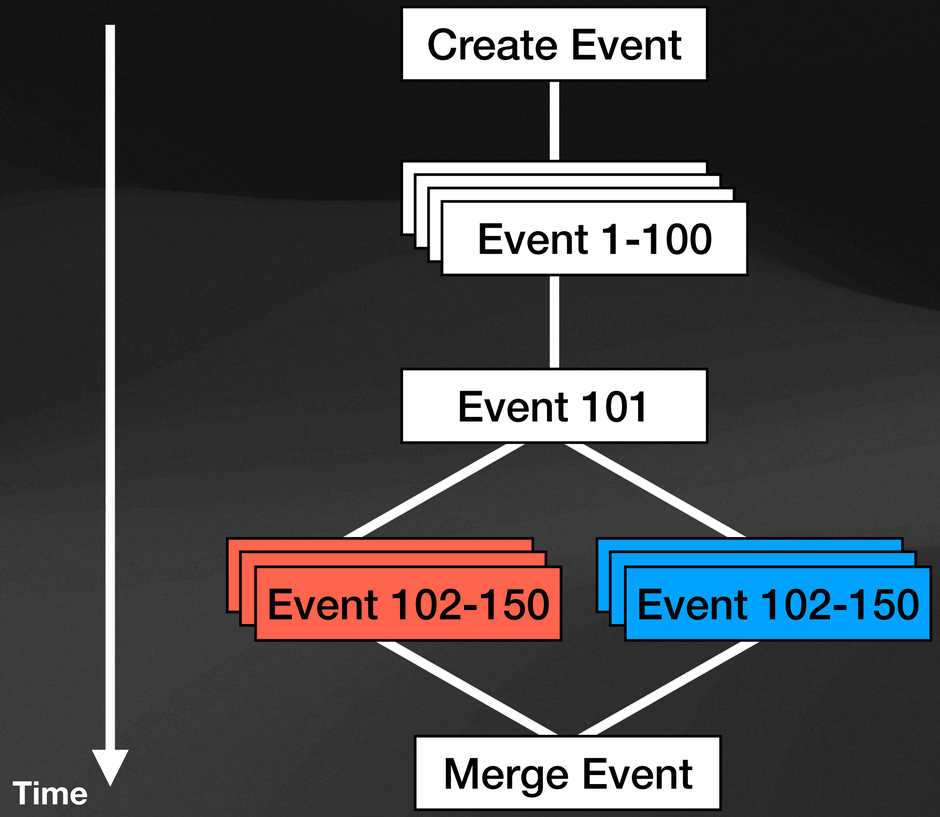
How to merge forks: redux



Execution Order: [Event 101 Snapshot , Event 102-150 , Merge Event]

Events we don’t agree on

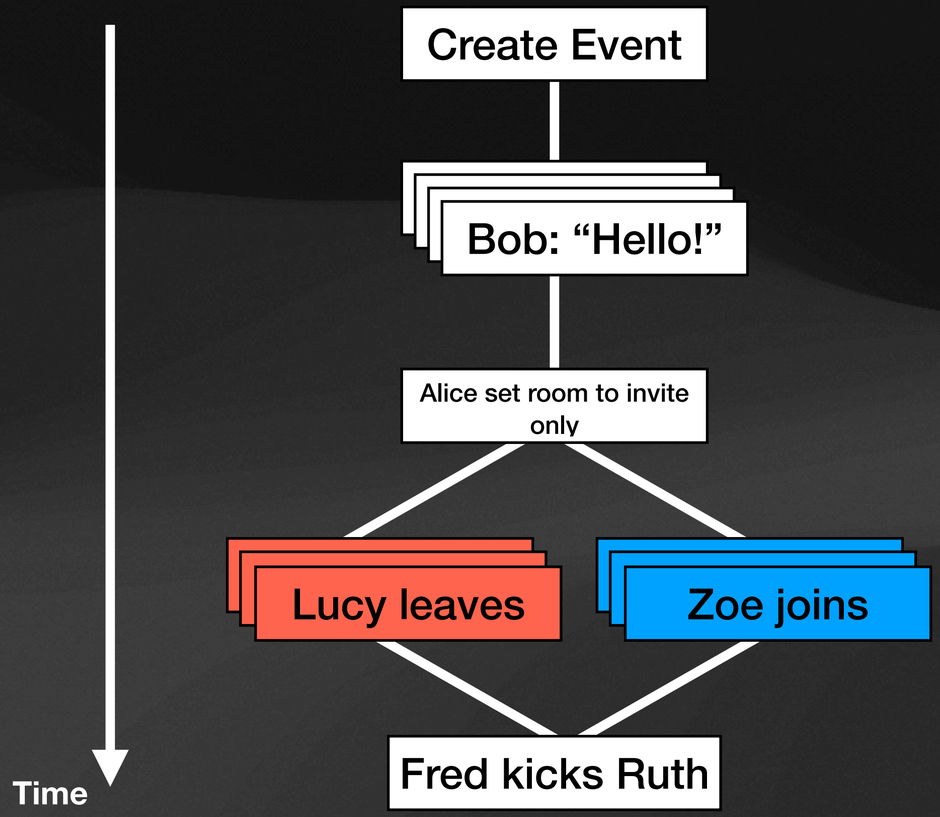
How to merge forks: snapshots



Execution Order: [ƒ({ Event 150 Snapshot , Event 150 Snapshot }), Merge Event]

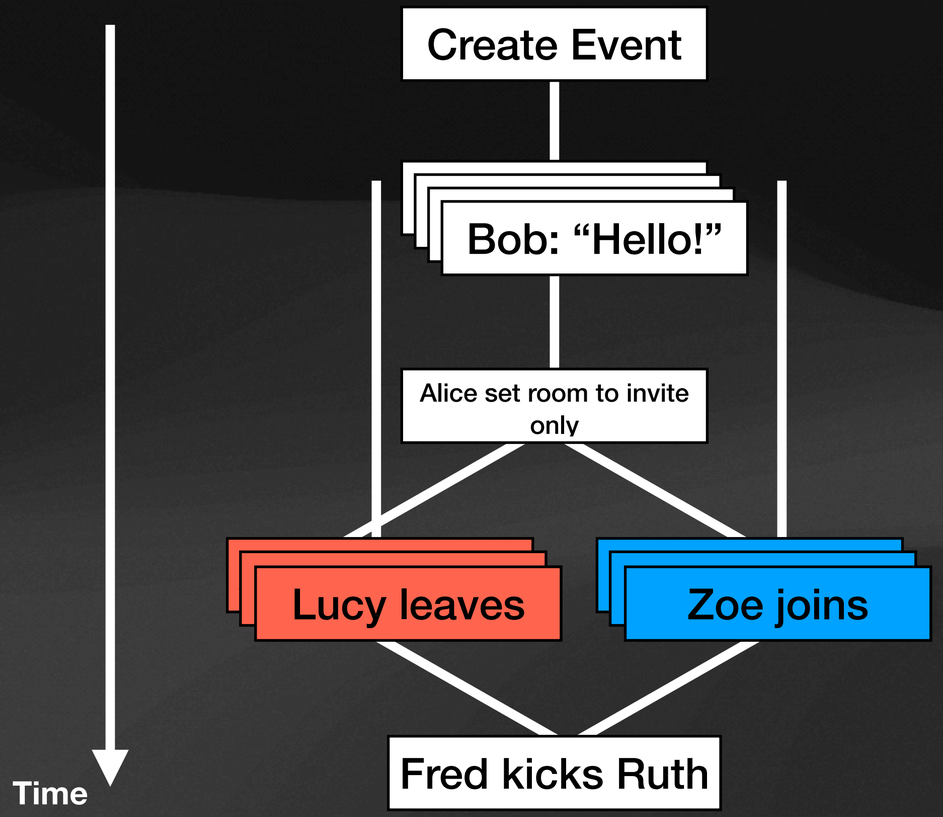
```
merge_states(state_1, state_2, … state_n)
```

How to merge room snapshots



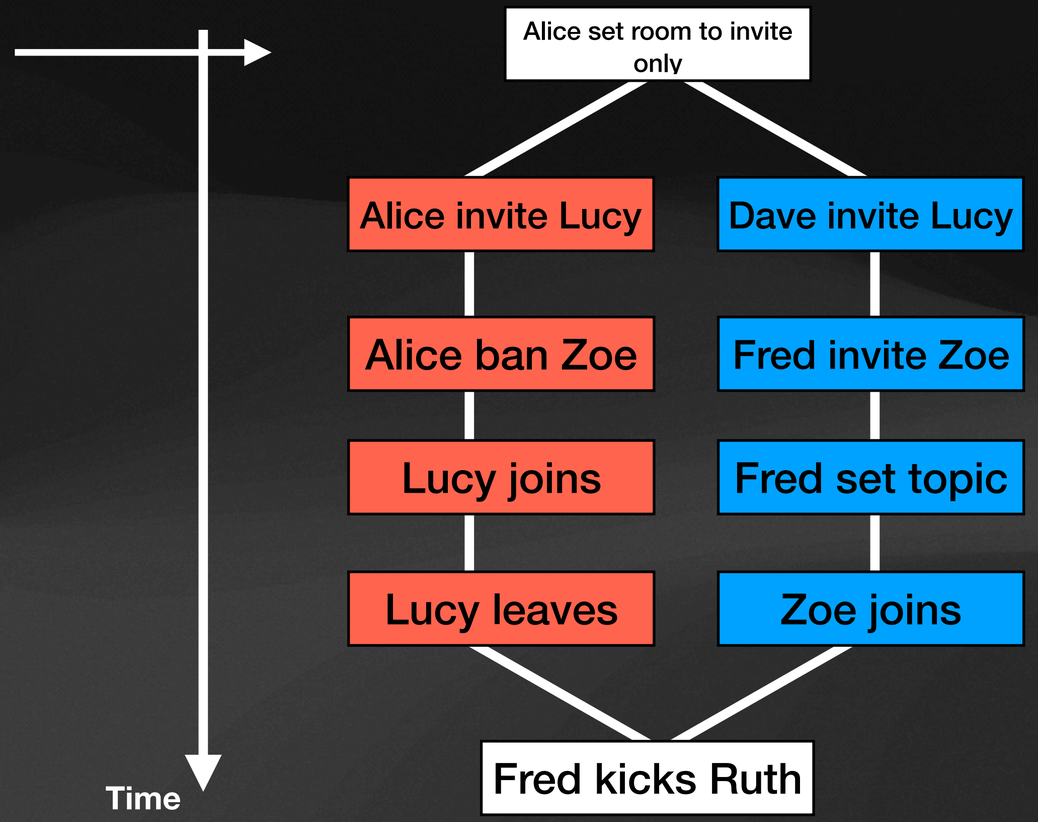
Execution Order: [ƒ({ Lucy leaves Snapshot , Zoe joins Snapshot }), Fred kicks Ruth]

How to merge room snapshots



Execution Order: [ƒ({ Lucy leaves Snapshot , Zoe joins Snapshot }), Fred kicks Ruth]

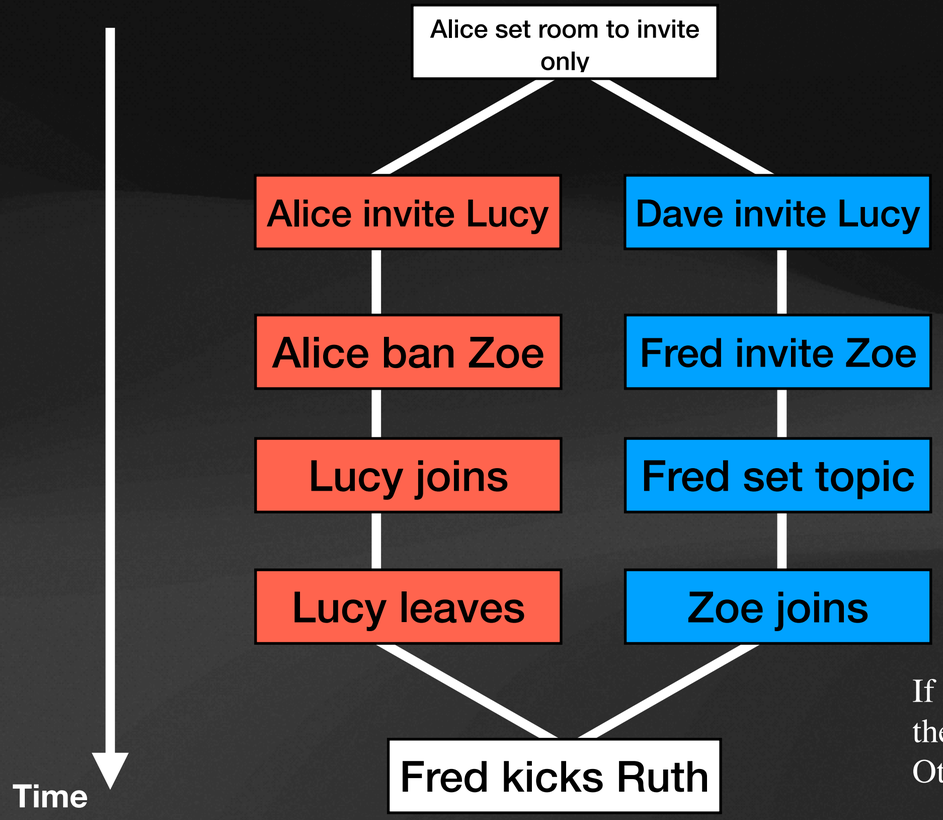
How to merge room snapshots



Join Rule	Lucy	Zoe
Alice set room to invite only	Lucy leaves	Alice ban Zoe

Events are keys in a map

Conflicted and Unconflicted Events



If a given key K is present in every S_i with the same value V in each state map, then the pair (K, V) belongs to the *unconflicted state map*.
Otherwise, V belongs to the *conflicted state set*.

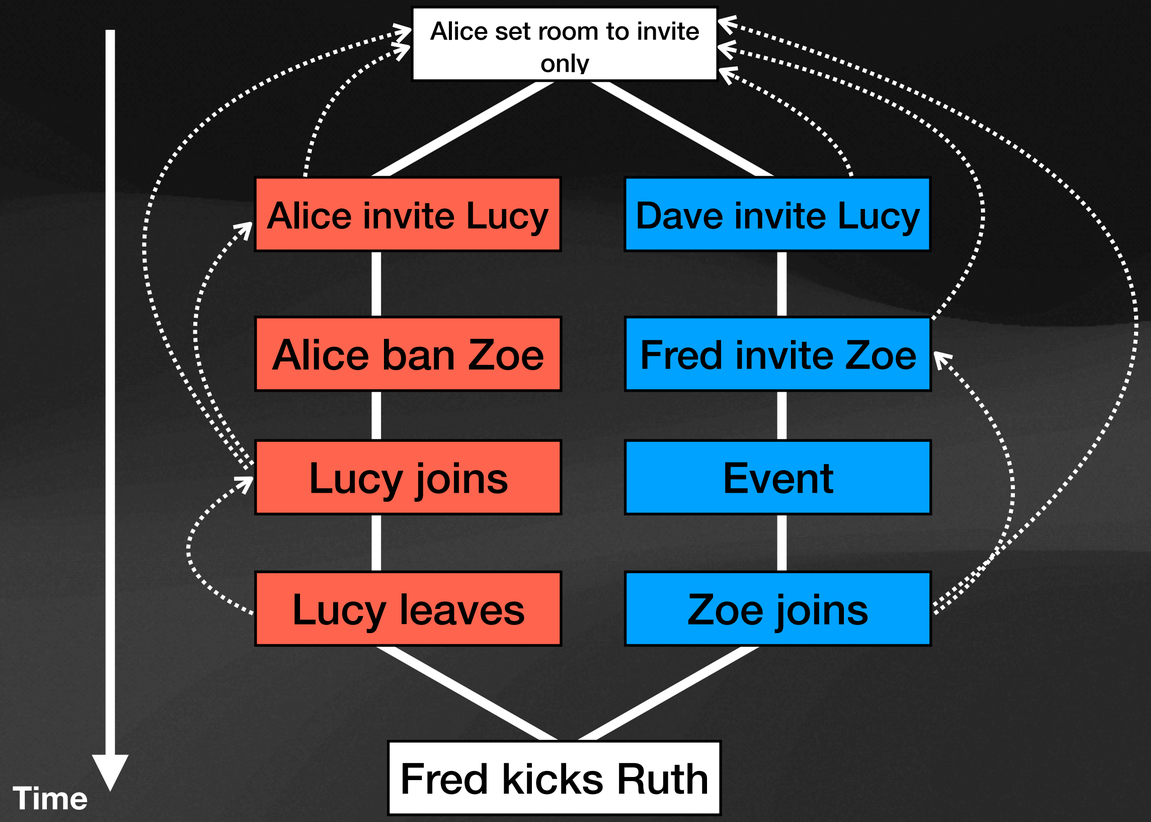
State After		Join Rule ✓	Lucy ✗	Zoe ✗	Topic ✗
Lucy leaves	=	Alice set room to invite only	Lucy leaves	Alice ban Zoe	
Zoe joins	=	Alice set room to invite only	Dave invite Lucy	Zoe joins	Fred set topic

Events depend on each other

- Whether you can join a room depends on if it’s invite-only and if you have an invite.
- Whether you can kick someone depends on if you’re a Moderator / Admin.
- Room snapshots are lossy, so we may not have this dependency information.
- If only we had something which represented these dependencies… oh wait:

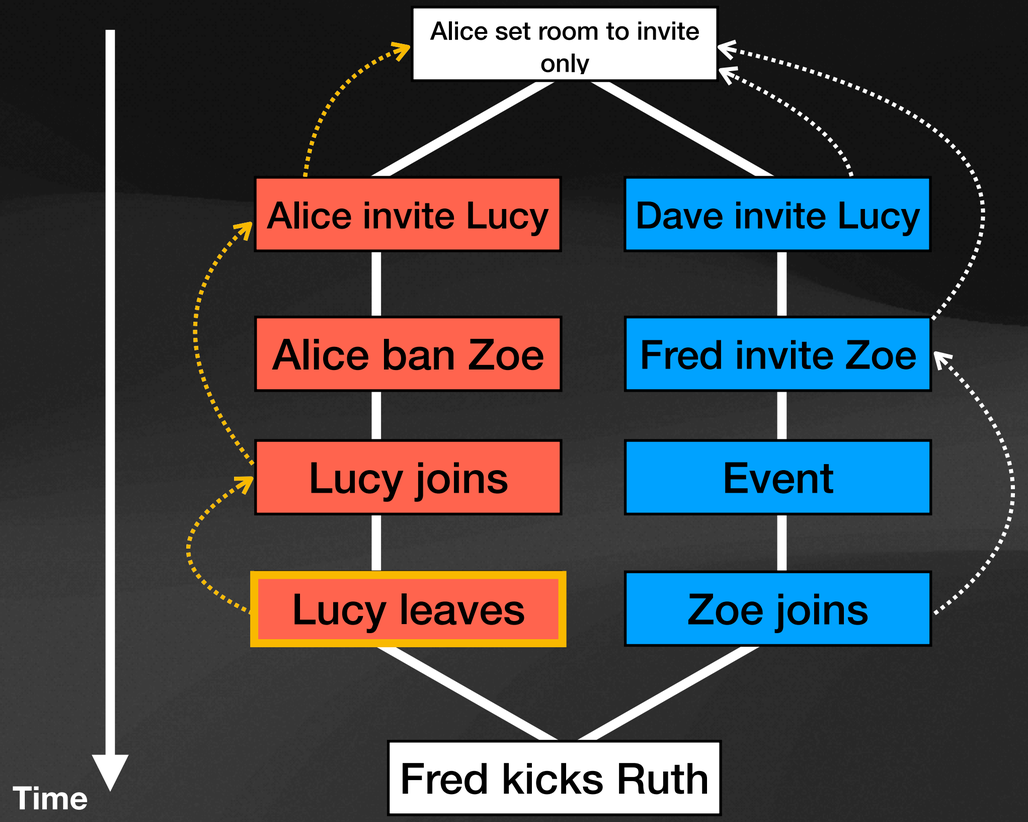
`auth_events!`

Calculating the auth difference



Auth chains from auth_events

Calculating the auth difference



The *auth difference* is calculated by first calculating the full auth chain for each state S_i, that is the union of the auth chains for each event in S_i,
and then taking every event that doesn't appear in every auth chain.

Lucy joins

Alice invite Lucy

Alice set room to invite only

Fred invite Zoe

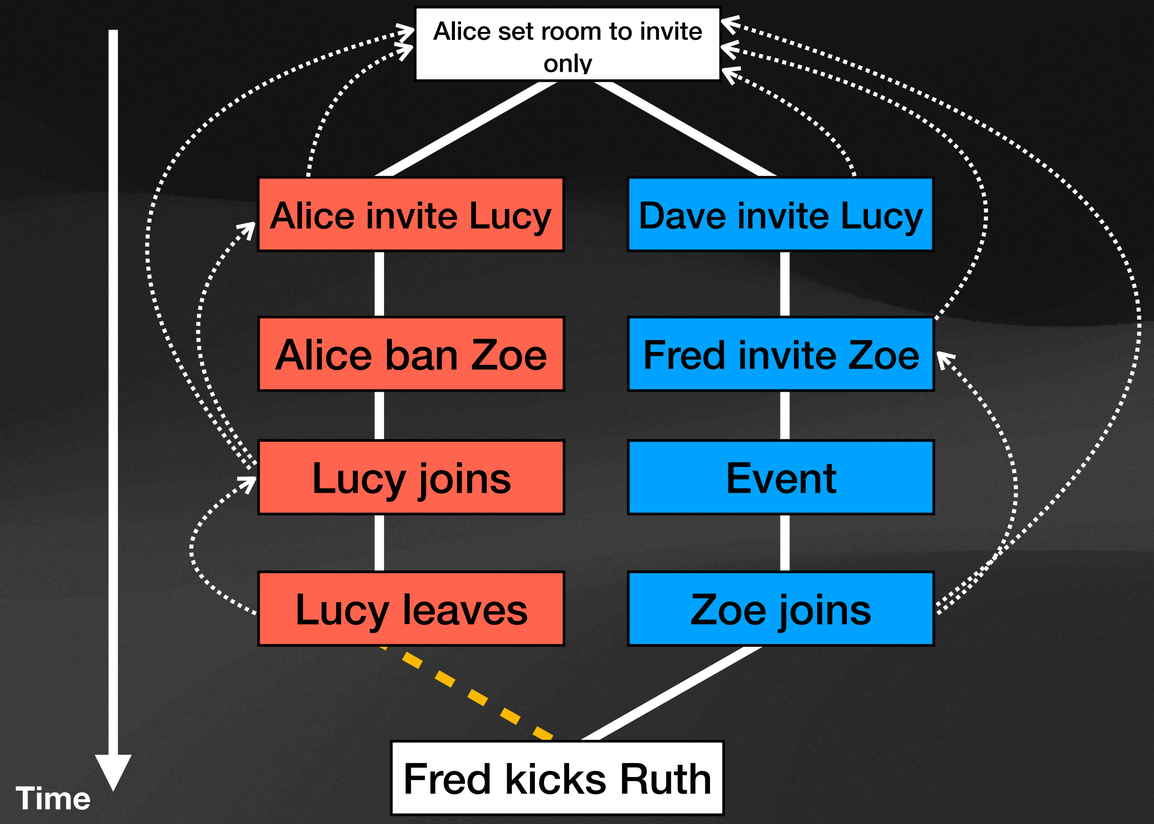
Alice set room to invite only

State After		Join Rule	Lucy	Zoe	Topic	Auth Difference
Lucy leaves	=	Alice set room to invite only	Lucy leaves	Alice ban Zoe		
Zoe joins	=	Alice set room to invite only	Dave invite Lucy	Zoe joins	Fred set topic	

What’s the point of all this?

- We don’t want to replay everything. These calculations provide all the necessary events we need to replay and that’s all.
- If we don’t replay everything, we can *partially synchronise* the DAG. This means we:
  - Sync all `auth_events`.
  - Only sync other events if they form part of the claimed room state on that fork.
- But there’s a cost to this…

What goes wrong with partial synchronisation



State After		Join Rule	Lucy	Zoe	Topic	Auth Difference	
Lucy leaves	=	Alice set room to invite only	Lucy joins !	Fred invite Zoe !		Alice invite Lucy	Lucy joins
Zoe joins	=	Alice set room to invite only	Dave invite Lucy	Zoe joins	Fred set topic		Fred invite Zoe

State Resolution v2.1

- The inability to verify state snapshots when performing partial synchronisation can lead to **state resets**: an *unintentional* rollback of room state.
- State Resolution v2.1 patches the algorithm to:
  - Start from the empty set, not the unconflicted set.
  - Include the conflicted state subgraph.
- TARDIS DEMO

State Resolution v2.1

- Ultimately, the only change in v2.1 is to *replay more events*. There's no changes to the way events are sorted or authorised.
- Is the optimisation worth the complexity cost?

What’s next?

- Statistical analysis of rooms on matrix.org show that it’s an over-optimisation to do partial synchronisation *for state events.* If we **fully synchronised all state events** this would pull in <100 additional events for 99.9% of rooms. This does not scale with the # users in the room, as these additional events are typically things like room names and topics.
- Some features which send lots of state events would be impacted performance wise, notably Spaces and MatrixRTC, the latter being helped via MSC4354: Sticky Events.
- => we’re seriously looking into “State DAGs” as a more secure model for synchronising room state over federation.

State DAGs

Where all state events are linked via `prev_state_events`

- **Obsoletes federation endpoints**: `/state, /state_ids, /event_auth`
- **Aligns with CRDT literature**: can use CRDT set reconciliation algorithms to synchronise DAGs.
- **Simplifies auth checks**: each server can calculate the **`auth_events`** so we only need to check once if an event is allowed, not twice.
- **Paves the way for simpler state resolution algorithms**: we have the entire graph so could replay events, or rely on (calculated) depth, or use CRDT-style commutative, associative merge functions.

Special Thanks

- Florian Jacob @ Karlsruher Institut für Technologie (KIT)
- Martin Kleppmann / Local-First community