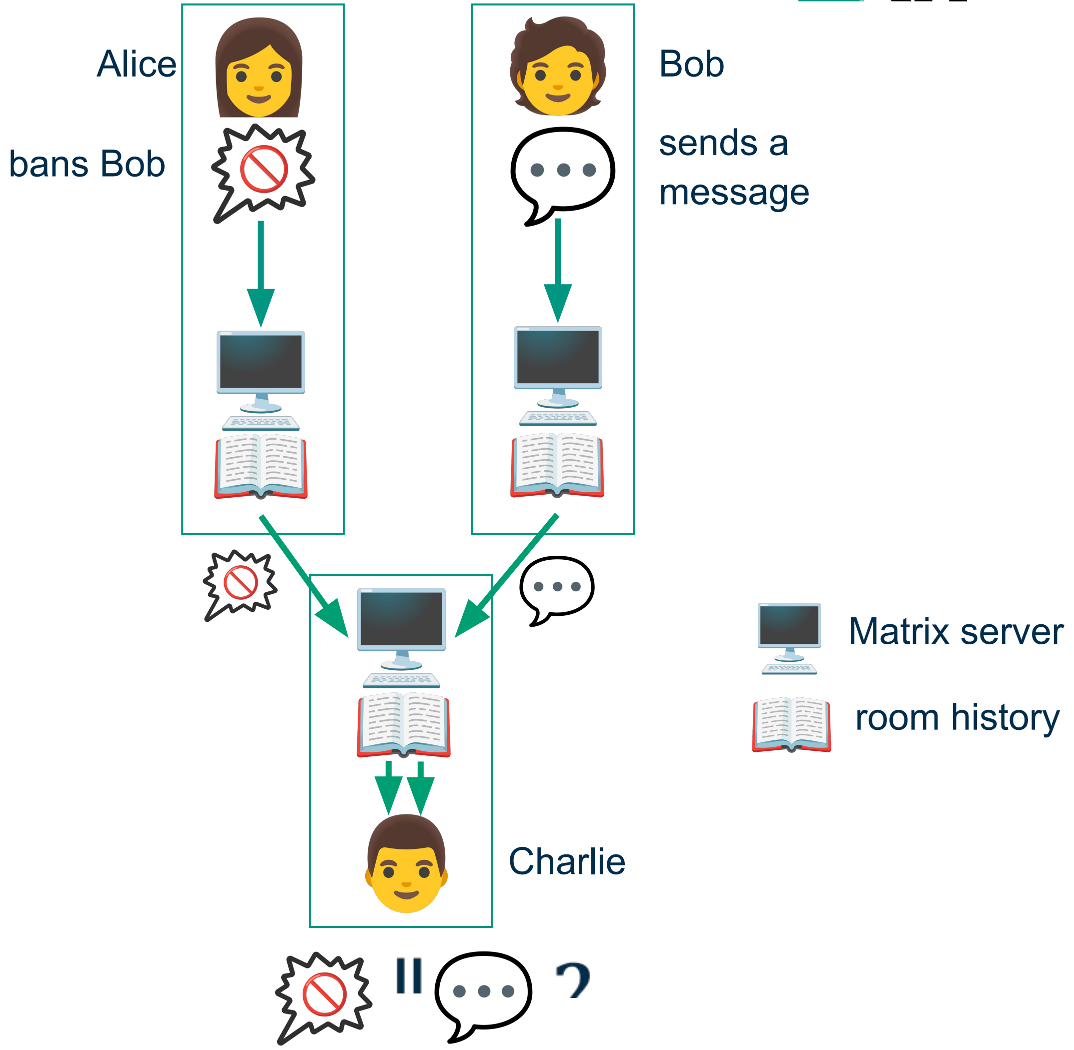
Eventually Consistent Access Control: Practical Insights on Matrix from Decentralized Systems Theory

Florian Jacob, Hannes Hartenstein
Karlsruhe Institute of Technology

@ Matrix Conference, Strasbourg, October 17, 2025

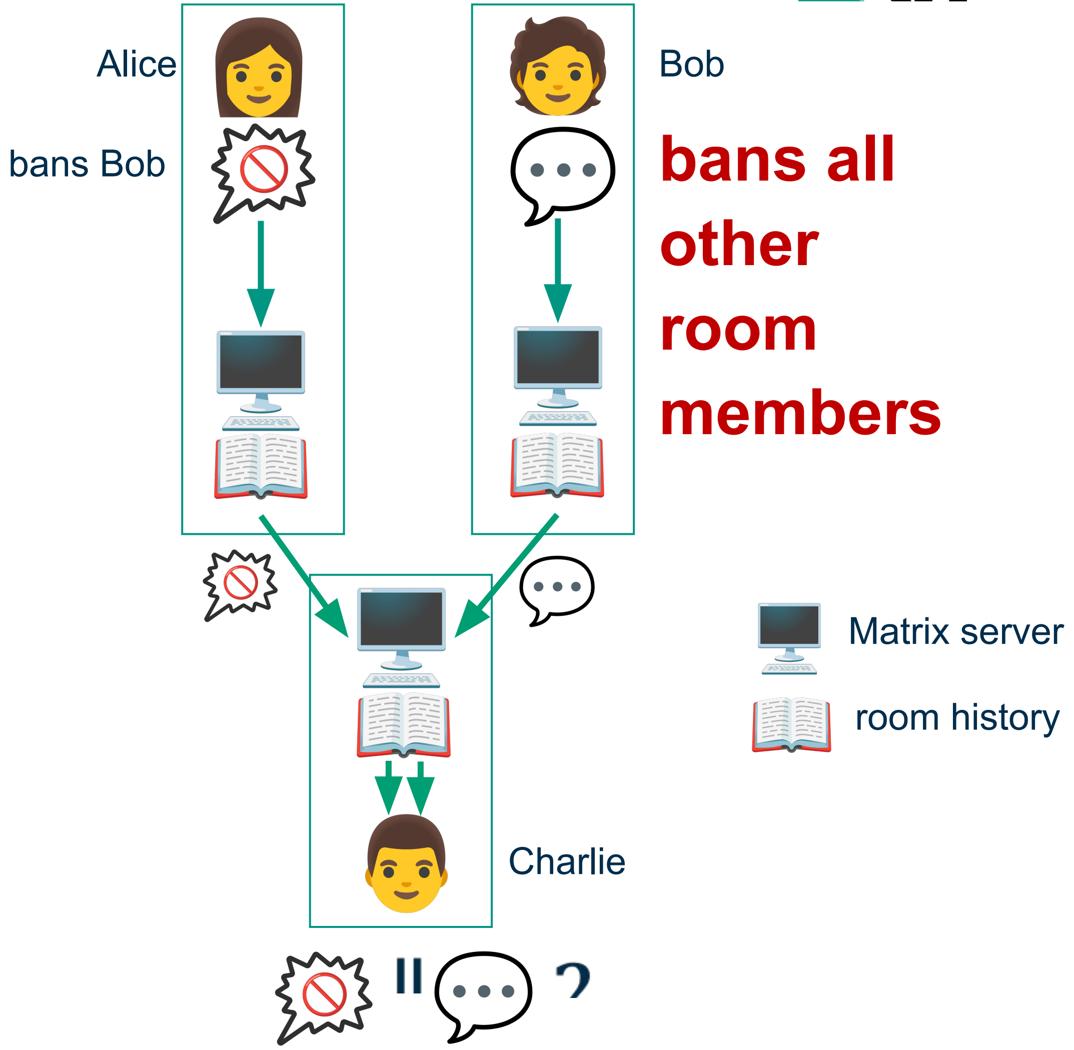


What to do when revocation and usage are concurrent?

Eventually Consistent Access Control: Practical Insights on Matrix from Decentralized Systems Theory

Florian Jacob, Hannes Hartenstein
Karlsruhe Institute of Technology

@ Matrix Conference, Strasbourg, October 17, 2025



What to do when revocation and usage are concurrent?

This Talk: The Essence after 6 Years of Research on Matrix

- **To the Best of Knowledge and Belief: On Eventually Consistent Access Control.** Fifteenth ACM Conference on Data and Application Security and Privacy (CODASPY 2025).
- **Proof-Carrying CRDTs allow Succinct Non-Interactive Byzantine Update Validation.** 12th Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC 2025).
- **Logical Clocks and Monotonicity for Byzantine-Tolerant Replicated Data Types.** 11th Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC 2024).
- **On Extend-Only Directed Posets and Derived Byzantine-Tolerant Replicated Data Types.** 10th Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC 2023).
- **On CRDTs in Byzantine Environments.** Proceedings of GI SICHERHEIT 2022.
- **Analysis of the Matrix Event Graph Replicated Data Type.** IEEE Access 2021.
- **Matrix Decomposition: Analysis of an Access Control Approach on Transaction-based DAGs without Finality.** 25th ACM Symposium on Access Control Models and Technologies (SACMAT 2020).

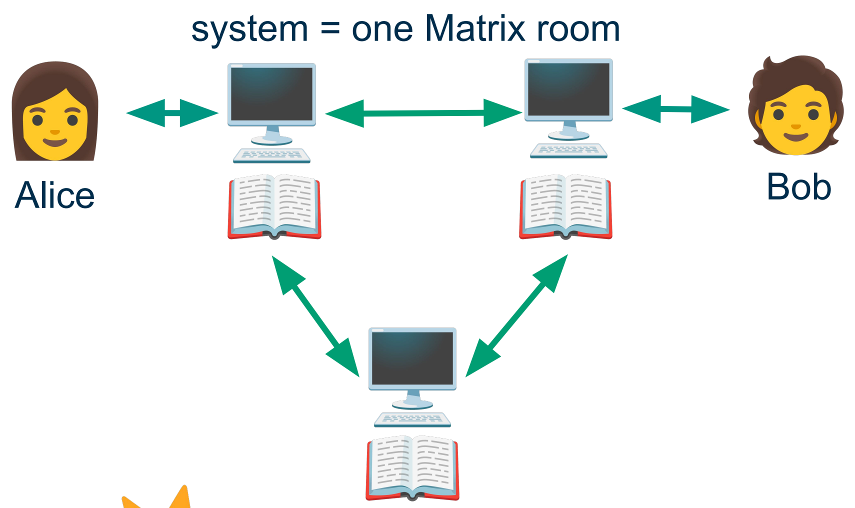
All papers are open access:
https://dsn.kastel.kit.edu/staff_jacob.php

Abstract

1. ## What we want: Resilient, Decentralized Messaging
   - Independent of arbitrary faults in the network or other servers.
   - Then, “eventual consistency” is among the best we can get.
2. ## How-To: Design Patterns from Decentralized Systems Theory
   - Matrix Event Graph
   - Conflict-Free Replicated Data Types
   - Consistency as Logical Monotonicity
3. ## What we get: Security to the Best of Knowledge and Belief
   - Authorization under the conviction of their correctness, but

Local-First Systems for Resilience

Local-first software: you own your data, in spite of the cloud. Martin Kleppmann et al., ACM Onward! 2019. https://doi.org/10.1145/3359591.3359737



Benefits of Local-First Resilience

- Availability
- Latency
- Scalability

Local-first systems are the ultimate way to maximize resilience.

Challenges to Local-First Consistency

- Concurrency
- Relativity of Time
- Arbitrary Faults of Everything Else (“Byzantine” faults)

Managing a set of autonomous entities to form a consistent system is hard.

Ordered Event Sets for Decentralized Messaging

Core Idea: Derive **system state** (= room state) from **ordered set of state changes** (= Matrix events)

- Replicate event set among all entities for consistency
- Resolve system state by executing events in order

$$\begin{tikzcd}
a_3 \arrow[d] \\
b_2 \arrow[d] \\
a_1 \arrow[d] \\
a_\perp
\end{tikzcd}$$

Logbook

$$\begin{tikzcd}
b_2 \arrow[d] & b_2' \arrow[dl] \arrow[d] & \\
a_1 \arrow[dr] & b_1 \arrow[d] & a_1' \arrow[dl] \\
& a_\perp &
\end{tikzcd}$$

Chronicle = Matrix Event Graph, Room DAG

What we would want: Collaborative Logbooks

- Append-only, totally-ordered set of events
- New events appear after all previous logbook events.
- New events cannot invalidate known events
- ***Entities must append events in coordination***

What we get in Matrix: Collaborative Chronicles

- Append-only, partially-ordered set of events
- New events appear after a subset of chronicle events
- New events may invalidate known, concurrent events
- ***Entities may append events local-first***

Why do we get chronicles in Matrix?

The Order of Events is Relative to the Observer

- Order in which events become visible for an entity differs between entities.

- Chronological order is an append-only partial order
  - Equal for all entities
  - Events whose effects were visible during creation of a new event chronologically precede the created event.
  - If the effects of two events are invisible for each other, they are concurrent.

$$\begin{tikzcd}
a_3 \arrow[d] \\
b_2 \arrow[d] \\
a_1 \arrow[d] \\
a_\perp
\end{tikzcd}$$

Logbook

$$\begin{tikzcd}
b_2 \arrow[d] & b'_2 \arrow[dl] \arrow[d] & \\
a_1 \arrow[dr] & b_1 \arrow[d] & a'_1 \arrow[dl] \\
& a_\perp &
\end{tikzcd}$$

Chronicle

Why does Matrix avoid coordination?

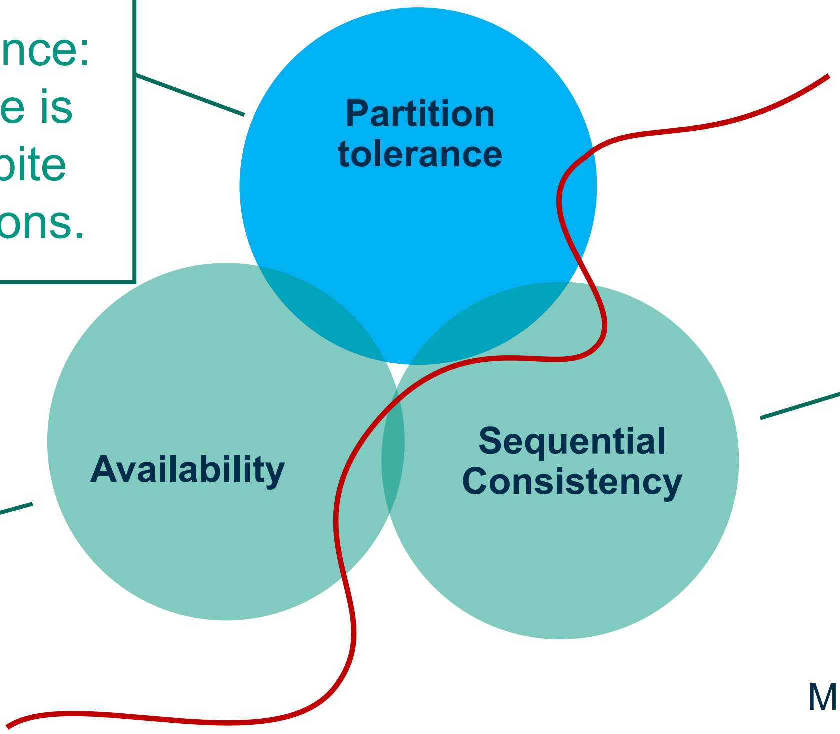
Detecting causal relationships in distributed computations: in search of the holy grail, Schwarz & Mattern, 1994. https://doi.org/10.1007/BF02277859

In Decentralized Systems, Coordination impacts Resilience

- Coordination among entities impacts latency, scalability, and fault tolerance.
  - Coordination implies waiting for an answer that may never comes.
  - Coordination is required for strong consistency, like with the logbook
- CAP theorem: Strong consistency contradicts availability under partition.
- Matrix wants availability under partition ⇒ Matrix has chronicles, not logbooks

Partition tolerance: system service is provided despite network partitions.

Availability: system service is provided despite entity faults.



Sequential Consistency: system service behaves as-if offered by central entity.

A Critique of the CAP Theorem,
Martin Kleppmann, 2015. http://arxiv.org/abs/1509.05393

Availability under Partition leads to Eventual Consistency

Eventual Consistency: what we get in Matrix

- Concurrency is accepted □ Chronicles
- Autonomy and resilience by avoiding coordination

Properties of Eventual Consistency:

- Local Visibility: Any new event is immediately visible to its author entity.
- Monotonic Visibility: Any visible events remain visible.
- Eventual Visibility: Any event visible to one correct entity is eventually visible to all correct entities.
- Convergence: If two correct entities see the same event set, they derive the same state.

A framework for consistency models in distributed systems,
Paulo Sérgio Almeida, 2024. https://arxiv.org/abs/2411.16355

Abstract

1. ## What we want: Resilient Decentralized Messaging
   - Independent of connection to and behaviour of other servers.
   - Then, “eventual consistency” is the best achievable consistency.
2. ## How-To: Design Patterns from Decentralized Systems Theory
   - Matrix Event Graph
   - Conflict-Free Replicated Data Types
   - Consistency as Logical Monotonicity
3. ## What we get: Security to the Best of Knowledge and Belief
   - [illegible]

Overview

Replicated Room Chronicle:

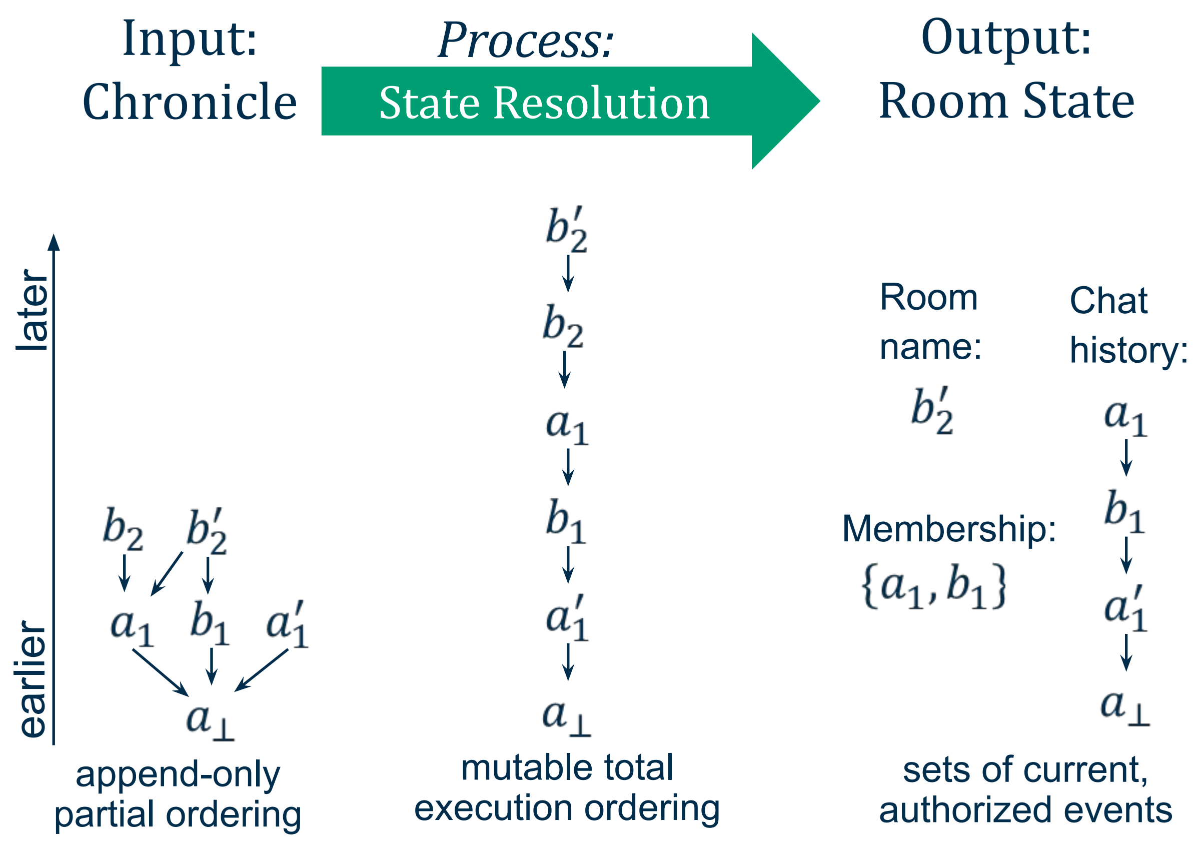
- Append-only set of events (system state changes)
- New events
  - either succeed or are concurrent to old events
  - cannot appear before old events

State Resolution:

- Derives total order for event execution by topological sorting
  - Concurrent events are ordered via execution priority
  - New events may appear anywhere after their chronological predecessors
- Resolve state changes by executing events in order

Room State:

- Current room name, set of members, chat history, …

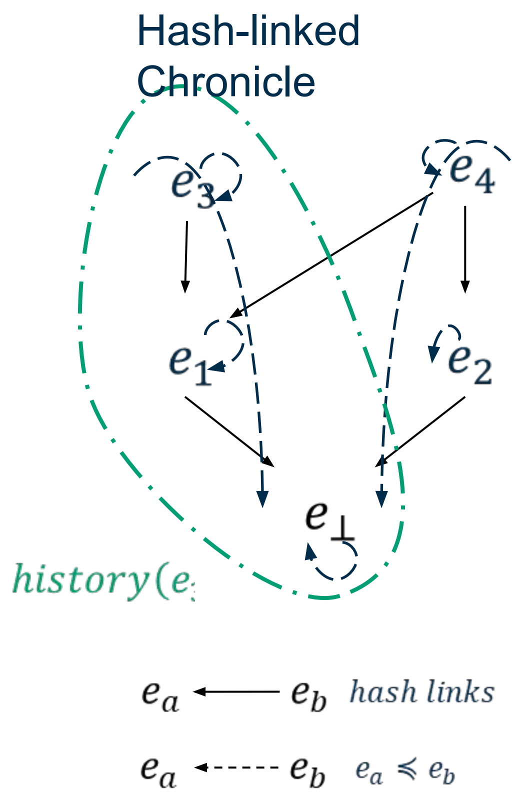

Input:
Chronicle
Process:
State Resolution
Output:
Room State
later
earlier
b_2 b'_2 a_1 b_1 a'_1 $a_\perp$
append-only partial ordering
b'_2 b_2 a_1 b_1 a'_1 $a_\perp$
mutable total execution ordering
Room name: b'_2
Membership: $\{a_1, b_1\}$
Chat history: a_1 b_1 a'_1 $a_\perp$
sets of current, authorized events

Matrix Event Graphs are Hash-Linked Chronicles

Hash-linked Chronicles

- Hash-link to set of newest chronological predecessors
  - Predecessor hashes recursive link event to its history, down to create event $e_\perp$
- Common design found in theory and practice
  - Matrix: Event Graph / Room DAG (Directed, Acyclic Graph)
  - Automerge (framework for local-first apps): collaborative document history

Hash-linked Chronicles provide eventual consistency


Hash-linked Chronicle
e_3
e_4
e_1
e_2
$e_\perp$
$history(e$
$e_a \leftarrow e_b$ $hash\ links$
$e_a \dashleftarrow e_b$ $e_a \preccurlyeq e_b$

Design Pattern: Conflict-Free Replicated Data Types (CRDTs)

CRDTs are *the* decentralized systems design pattern for eventual consistency.

- Collaborative, replicated data types that guarantee automatic conflict resolution
  - Arbitrary concurrency without mutual exclusion
  - Available under partition
- *“Like git, but for anything, and without needing manual conflict resolution.”*

Conflict-Free Replicated Data Types

1. Entities periodically broadcast their local CRDT state.
2. On update operations, entities broadcast a delta state.
3. On receiving a remote CRDT state / delta, entities merge it with their local CRDT state.
   - Merging must be the “least upper bound” of local and remote CRDT state, then eventual consistency is guaranteed by mathematics.

Approaches to Conflict-free Replicated Data Types.
Paulo Sérgio Almeida, 2024. https://doi.org/10.1145/3695249

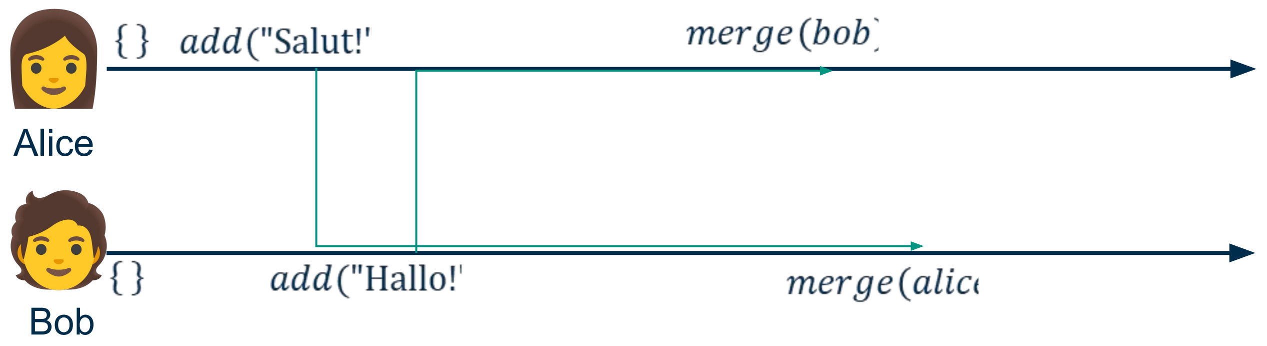
Example: Add-only Set CRDT

1. Periodically broadcast local set.
2. Update operation `add(item)`: add item to local set, broadcast delta state with that item.
3. On receiving remote set, merge with local set via set union.
   - Set union is the least upper bound of local and remote set.

- This simple snippet is a CRDT already
  - Eventually consistent, available under partition
  - Very resilient through state broadcasting
  - Trade-off: state growth

```rust
use std::collections::HashSet;

struct AddOnlySet(HashSet<String>);

impl AddOnlySet {
 fn value(&self) -> HashSet<String> {
   self.0
 }

 fn periodic(&self) {
   broadcast(self.0);
 }

 fn add(&mut self, item: String) {
   self.0.insert(item);
   broadcast(&HashSet::from([item]));
 }

 fn merge(&mut self, remote: &HashSet<String>) {
   self.0.extend(remote);
 }
}
```

Eventual Consistency of Add-only Set CRDT



Does not seem too useful yet?

- What if items encode hash-linked Matrix events as JSON strings?
- Add-only set CRDTs are the core of a chronicle implementation
- Hash-linked chronicles are also CRDTs, extending add-only set CRDTs with hash linking and partial ordering

```rust
use std::collections::HashSet;

struct AddOnlySet(HashSet<String>);

impl AddOnlySet {
 fn value(&self) -> HashSet<String> {
   self.0
 }

 fn periodic(&self) {
   broadcast(self.0);
 }

 fn add(&mut self, item: String) {
   self.0.insert(item);
   broadcast(&HashSet::from([item]));
 }

 fn merge(&mut self, remote: &HashSet<String>) {
   self.0.extend(remote);
 }
}
```

Chronicles are sets of system state changes. How to resolve system state?

What we have now:

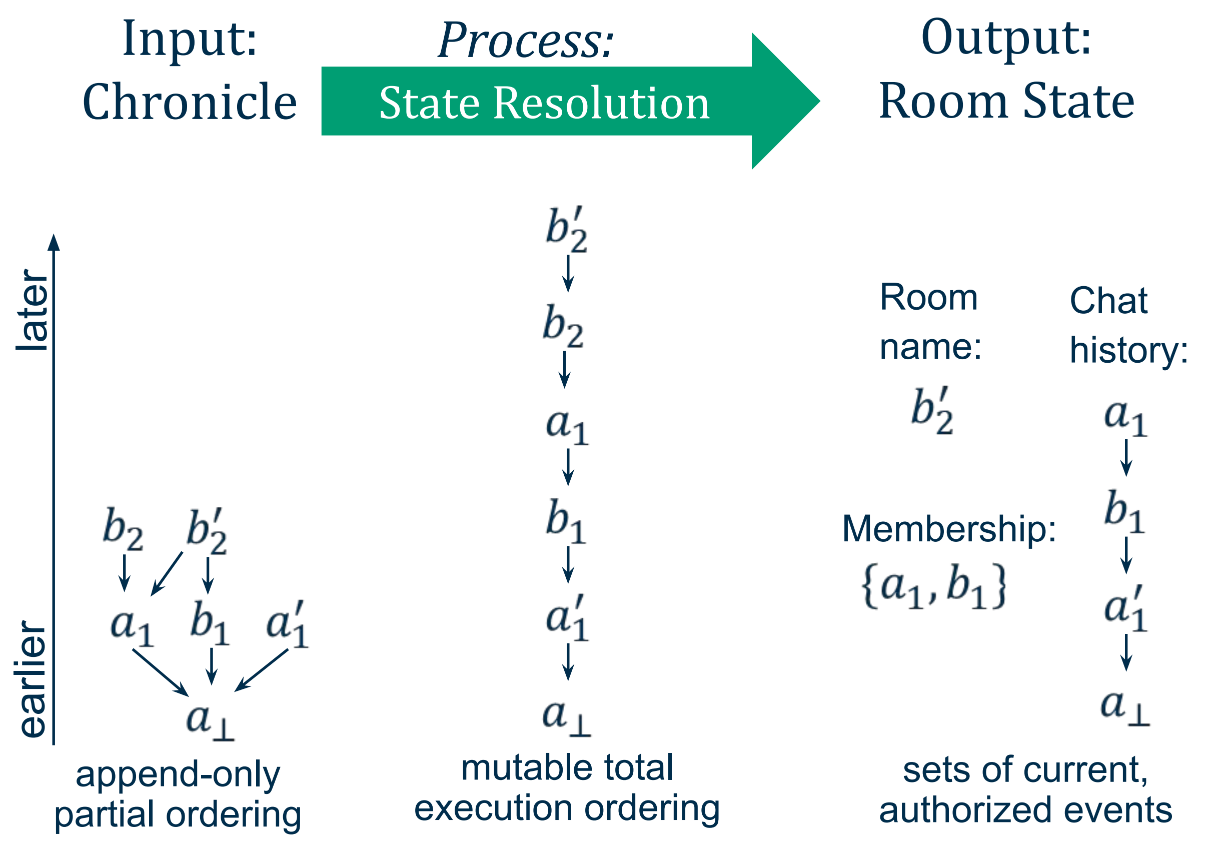
- Chronicles: partially-ordered sets of events
- Events: describe system state changes

What users actually care about:

- Properties of rooms and room members
  - Name
  - Topic
  - Authorizations, …
- A totally-ordered chat history

How to get there:

- We use chronicles as foundation and compose specific CRDTs on top.
- The composed CRDTs' states make up the room state.


Input: Chronicle
Process: State Resolution
Output: Room State
later
earlier
b_2 b_2' a_1 b_1 a_1' $a_\perp$
append-only partial ordering
$b_2' \to b_2 \to a_1 \to b_1 \to a_1' \to a_\perp$
mutable total execution ordering
Room name: b_2'
Membership: $\{a_1, b_1\}$
Chat history: $a_1 \to b_1 \to a_1' \to a_\perp$
sets of current, authorized events

Example: Composed CRDT for a String in Room State

- Compose a "register" for strings on top of chronicle
- Assumptions on Chronicle CRDT
  - Events are prefiltered to concern only our register
  - Events have chronological `history` set
  - Events have a UNIX `timestamp`

Topological sorting:

2. Sort events by comparing history sets (set inclusion)
3. Then, sort events by comparing UNIX timestamp
   - Issue with Byzantine entities / for authorization…?
4. Take last event ("last writer wins")

- This is a simple form of state resolution already
- In a similar way, we can build lists, sets, maps, …

```
impl Chronicle {
 fn events(&self) -> HashSet<Event> …;
 fn newest(&self) -> HashSet<Event> …;
 // appends `e` after `prev` events
 fn append(&mut self, e: Event, prev: HashSet<Event>)
…;
 …
}

struct LastWriterWinsRegister(Chronicle);

impl LastWriterWinsRegister {
 fn value(&self) -> Event {
  let events = Vec::from(self.0.value().events());
  events.sort_by(|e1, e2| e1.history.cmp(e2.history)
                    .then(e1.timestamp.cmp(e2.timestamp)));
  events.last()
 }

 fn assign(&mut self, value: String) {
   self.0.append(Event::new(value), self.0.newest());
 }
}
```

Monotonicity, State Resets, and the CALM Theorem

- If `value()` changes, the new event is either:
  - a chronological successor to the old event, or
  - chronologically concurrent to the old event,
    but has a later timestamp than the old event.
  - ► *`value()` events get "newer" monotonically*
- If current `value()` would get invalidated and rolled back to an older event, that is a dreaded "state reset".

CALM Theorem: algorithms are consistent without coordination if and only if they are monotonic.

- "Logical monotonicity": previous knowledge (□ state) derived from old facts (□ events) is not invalidated (□ state reset) when learning new facts.
  - Note: immutability = best kind of monotonicity
- "Consistency": any local state is a lower bound on global state, independent of event reception ordering.

Keeping CALM: when distributed consistency is easy. Hellerstein et al., Communications of the ACM 2020. https://doi.org/10.1145/3369736

Hash-Linked Chronicles: Defense by Monotonicity

Main attack vectors of Byzantine entities:

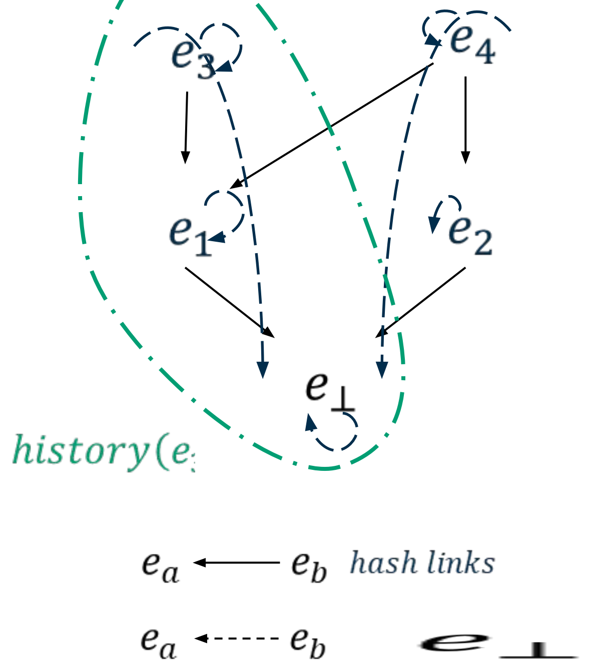
- Omission: Attacker omit sending events to all / receiving events they dislike.
  - We have to expect lost events anyway…
- Equivocation: Attacker sends concurrent but different events to different entities.
- Backdating: Attacker retroactively performs equivocation later on.

Recursive hash linking guarantees:

- Chronologic ordering: link targets must have existed before hash links.
- Integrity: Attempts to retroactively change events would result in a new event.
- Monotonicity: equivocations are distinguishable by hash, merging includes both equivocations. Equivocations cannot harm consistency equivocation tolerance.

- Equivocation and backdating mainly attack reception ordering, but monotonic algorithms give consistent results independent of reception ordering
  - ▶ Monotonicity is also a strong defense against Byzantine entities

Hash-linked Chronicle



Design Pattern: “Maximal” Monotonicity

- General idea: Let‘s make everything monotonic!
- **But**: Issue: occasionally, we need to ban users in Matrix rooms
  □ **permission revocation is non-monotonic** ☹

- CALM gives us two options:

1. Coordinate and cease availability under partition. ☹
2. Make revocations force deliberate state resets □ Matrix follows this approach

Decentralized Systems Design Pattern: “Maximal” Monotonicity

- Monotonic system parts need no coordination (fast, resilient, secure, …)
- Maximize monotonic parts of your system
- Minimize non-monotonic parts **and thoroughly examine them**.
  - Analyze consequences of order dependence on consistency and security.
  - Unacceptable? Bear with coordination / centralization. But in no other case.

Logic and lattices for distributed programming.
Conway et al, ACM SoCC 2019.
https://doi.org/10.1145/2391229.2391230

Abstract

1. What we want: Resilient Decentralized Messaging

- Independent of connection to and behaviour of other servers.
- Then, “eventual consistency” is the best achievable consistency.

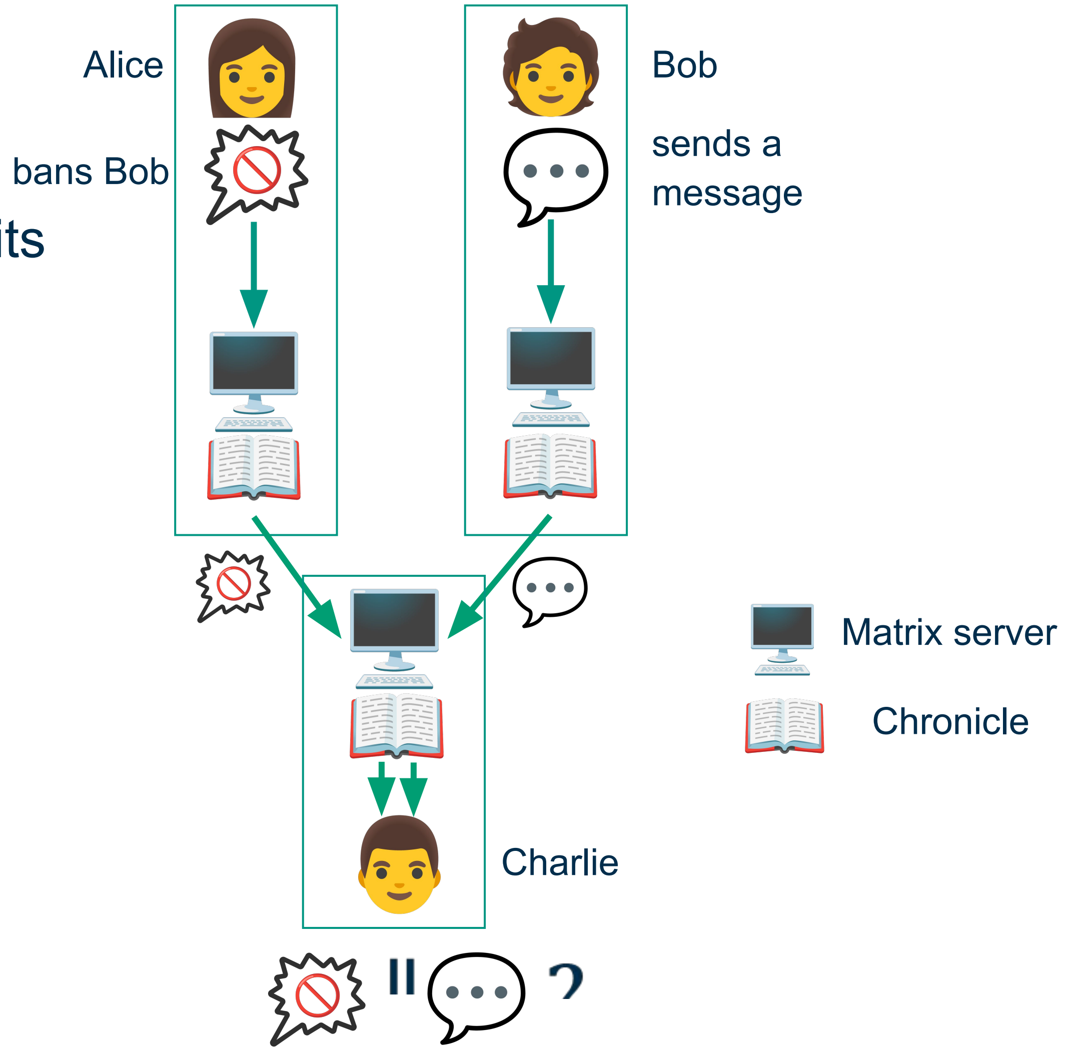
2. How-To: Design Patterns from Decentralized Systems Theory

- Matrix Event Graph
- Conflict-Free Replicated Data Types
- Consistency as Logical Monotonicity

3. What we get: Security to the Best of Knowledge and Belief

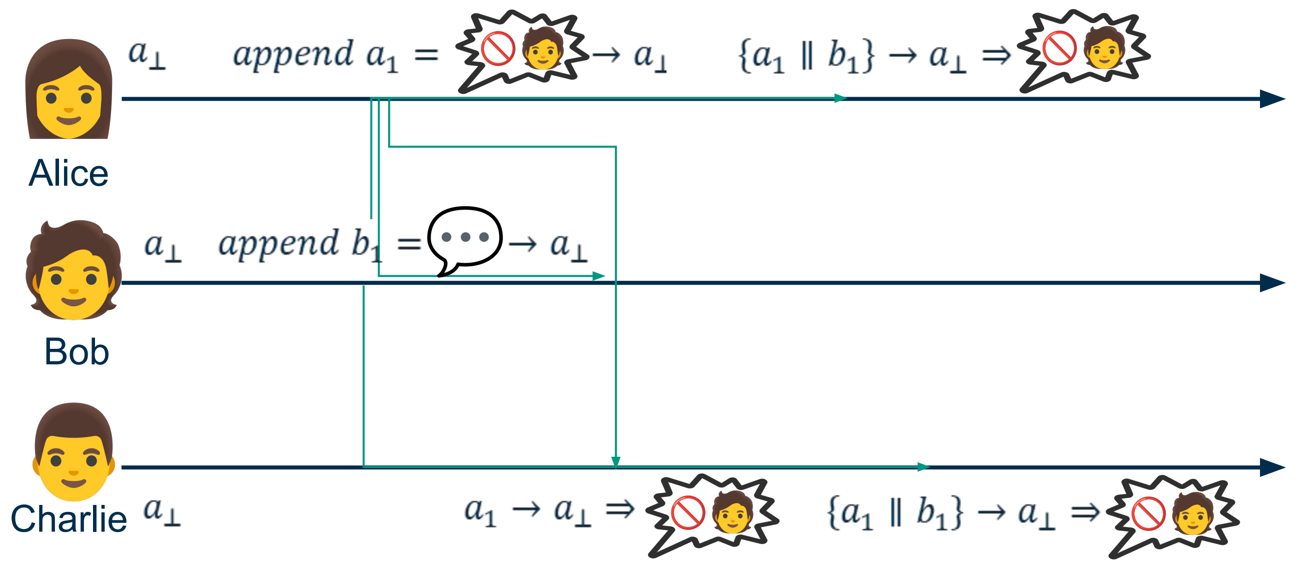
Concurrency: Core Problem of Event Authorization

- Differentiate between:
  - Chronicle authorization: event must be authorized by its chronological predecessor to appear in the chronicle.
  - State authorization: event must be authorized by its execution predecessors to appear in the state.



What to do when revocation and usage are concurrent?

Practical State Resolution: Ban arrives before Message



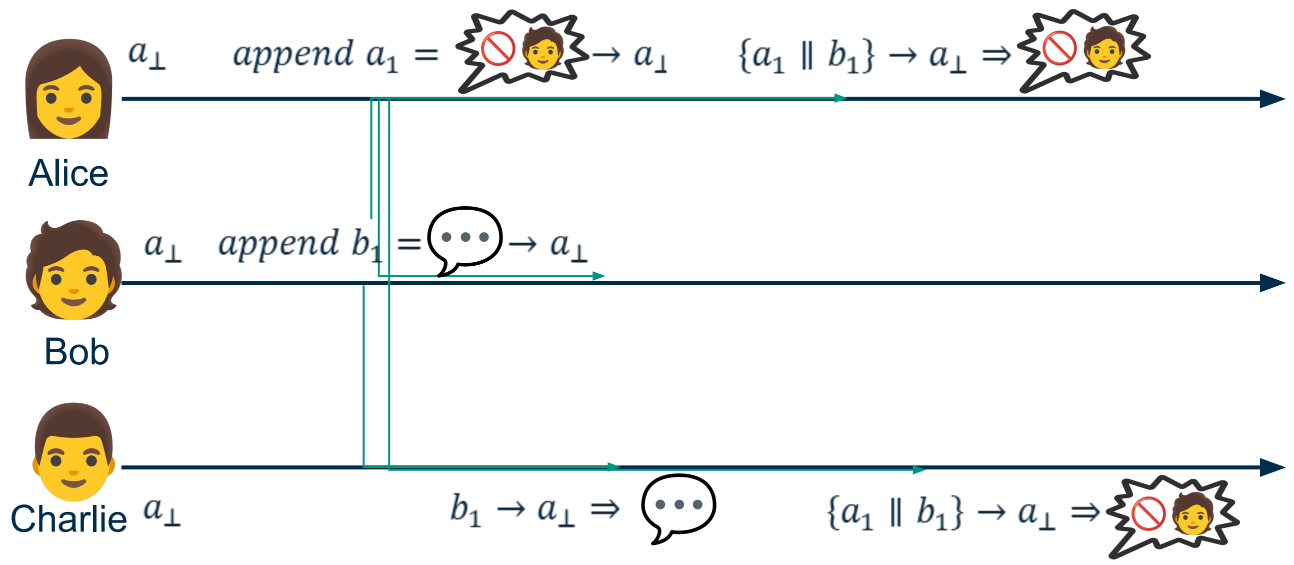
Alice
Bob
Charlie
$a_\perp$
$append\ a_1 =$
$\rightarrow a_\perp$
$\{a_1 \parallel b_1\} \rightarrow a_\perp \Rightarrow$
$a_\perp$
$append\ b_1 =$
$\rightarrow a_\perp$
$a_\perp$
$a_1 \rightarrow a_\perp \Rightarrow$
$\{a_1 \parallel b_1\} \rightarrow a_\perp \Rightarrow$

Execution priority for concurrent events

1. Revocations before anything else
2. Events from higher permission level users before events from lower permission level users
3. Events with earlier UNIX timestamp before events with later UNIX timestamps
4. Events with smaller hash value before events with larger hash value

1. Alice appends event a_1 to ban Bob, while Bob concurrently appends message b_1.
2. Charlie sees Alice's ban a_1 $\Rightarrow$ Bob is banned
   - Charlie knows that a_1 has chronicle authorization.
   - Charlie currently believes that a_1 has state authorization.
3. Charlie sees Bob's message b_1 $\Rightarrow$ Bob is banned, b_1 not in chat history
   - Charlie knows that b_1 has chronicle authorization
   - Charlie currently believes that b_1 has no state authorization, because a_1 executes first.

Practical State Resolution: Message arrives before Ban



Alice
$a_\perp$
$append\ a_1 =$
$\to a_\perp$
$\{a_1 \parallel b_1\} \to a_\perp \Rightarrow$
Bob
$a_\perp$
$append\ b_1 =$
$\to a_\perp$
Charlie
$a_\perp$
$b_1 \to a_\perp \Rightarrow$
$\{a_1 \parallel b_1\} \to a_\perp \Rightarrow$

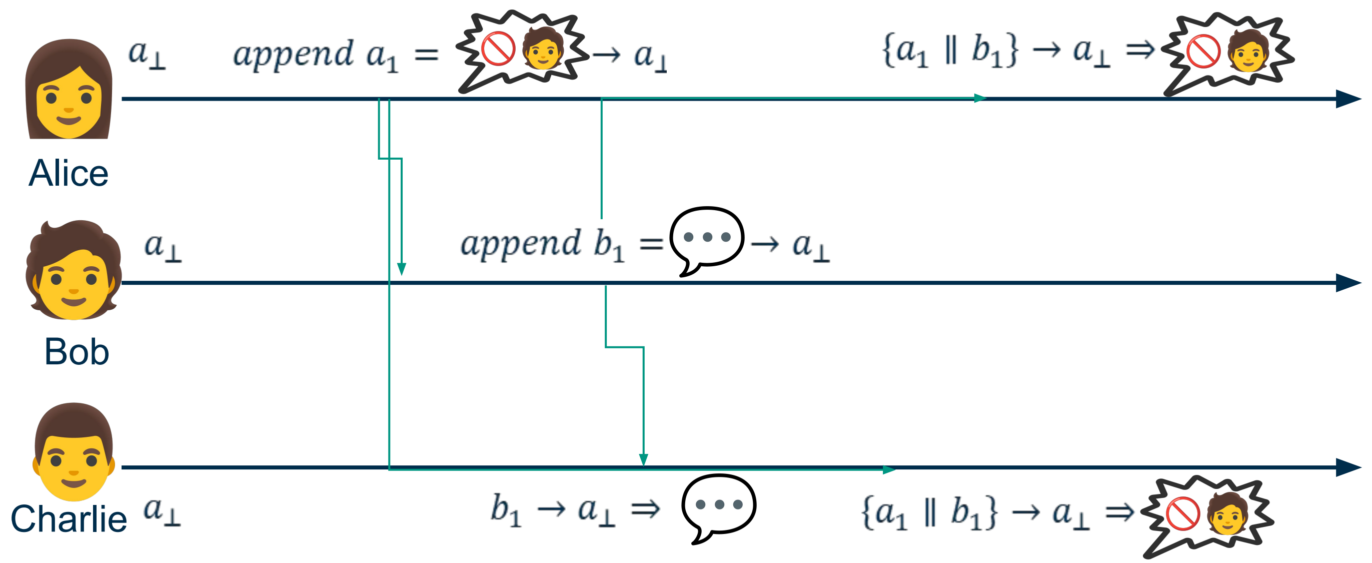
Execution priority for concurrent events

1. Revocations before anything else
2. Events from higher permission level users before events from lower permission level users
3. Events with earlier timestamp before events with later timestamps
4. Events with smaller hash value before events with larger hash value

1. Alice appends event a_1 to ban Bob, while Bob concurrently appends message b_1.
2. Charlie sees Bob's message b_1 $\Rightarrow$ b_1 in chat history
   - Charlie knows that b_1 has chronicle authorization.
   - Charlie currently believes that b_1 has state authorization.
3. Charlie sees Alice's ban a_1 $\Rightarrow$ Bob is banned, b_1 not in chat history
   - Charlie knows that a_1 has chronicle authorization.
   - Charlie changes beliefs: a_1 has state authorization, and revokes state authorization of b_1.

Revocations are non-monotonic / state resets: Depending on reception order, b_1 is in Charlie's chat history

Practical State Resolution: Bob backdates Message



$a_\perp$
$append\ a_1 =$
$\rightarrow a_\perp$
$\{a_1 \parallel b_1\} \rightarrow a_\perp \Rightarrow$
Alice
$a_\perp$
$append\ b_1 =$
$\rightarrow a_\perp$
Bob
Charlie
$a_\perp$
$b_1 \rightarrow a_\perp \Rightarrow$
$\{a_1 \parallel b_1\} \rightarrow a_\perp \Rightarrow$

Execution priority for concurrent events

1. Revocations before anything else
2. Events from higher permission level users before events from lower permission level users
3. Events with earlier timestamp before events with later timestamps
4. Events with smaller hash value before events with larger hash value

1. Alice appends event a_1 to ban Bob.
2. Bob acts as if not having received a_1, and backdates message b_1 as concurrent to a_1.
3. Charlie sees Bob's message b_1 $\Rightarrow b_1$ in chat history
   - Charlie knows that b_1 has chronicle authorization.
   - Charlie currently believes that b_1 has state authorization.
4. Charlie sees Alice's ban a_1 $\Rightarrow$ Bob is banned, b_1 not in chat history
   - Charlie knows that a_1 has chronicle authorization.

Backdating events is indistinguishable from high network latency.

Event Authorization to the Best of Knowledge and Belief

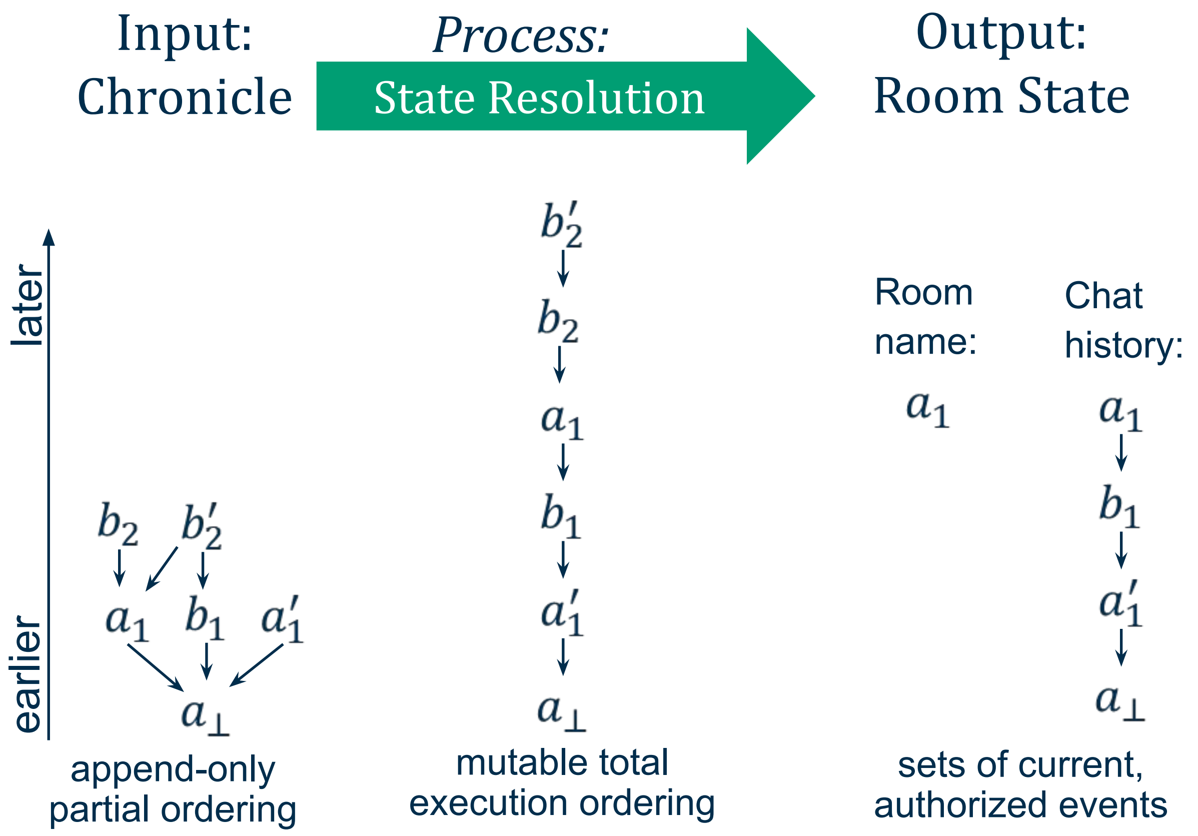
Chronicle Authorization

- Events must be authorized by chronological predecessors.
- Set of chronological predecessors is immutable
- Chronicle authorization is monotonic
  - Authorization decisions under knowledge of correctness.

State Authorization

- Events must be authorized by execution predecessors.
- Set of execution predecessors is non-monotonic
- State authorization is non-monotonic
  - Revocations are on-purpose state resets.
  - Authorization decisions under conviction of belief correctness, but fallibility due to incomplete knowledge.

Learning new, concurrent events
may invalidate my beliefs on state authorization of events,
but not my knowledge on chronicle authorization of events.



Summary: Practical Insights on Matrix from Academia

- Local-first systems
- Monotonicity
- Conflict-free replicated data types

- Matrix is a local-first system for resilient, decentralized messaging
- Matrix event graph: a conflict-free replicated data type (CRDT) for chronicles
  - Recursive hash linking ⇒ monotonicity ⇒ security
  - Chronicle authorization: monotonic knowledge
- Matrix state resolution: a composition of CRDTs on top of chronicles
  - Event execution in topological order resolve state changes to state
  - Challenge: state authorization: non-monotonic belief, because we want revocations

Eventually consistent access control is
to the best of knowledge and belief.
And: the non-monotonic part needs further investigation.

[m] @bh7953:kit.edu

florian.jacob@kit.edu