CRDTs

Christopher Meiklejohn
Université catholique de Louvain, Belgium
Annette Bieniusa
University of Kaiserslautern, Germany

basho

bet365

ROVIO

SCALITY

RIOT GAMES

LEAGUE of LEGENDS

riak

TOMTOM

TRIFORK.
...think software

Tapjoy

SOUNDCLOUD

HELIUM

akka

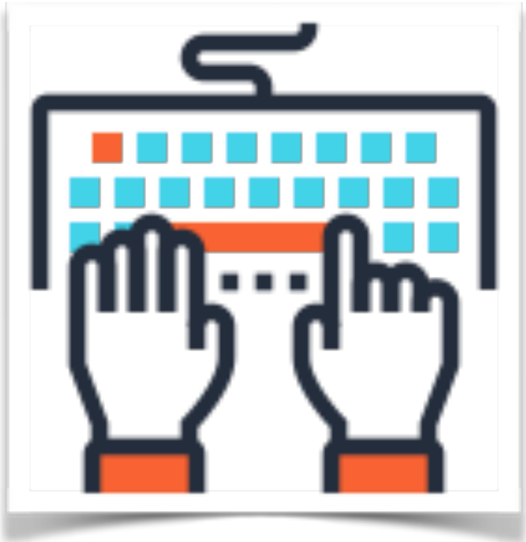
PROJECTFIFO

Outline

- What is the problem with concurrent modifications?
- Conflict-resolution strategies
- Taking data-type semantics into account
- OR-Sets, OR-Dictionaries and other CRDTs

Replicated data

- Share data ⇒ Replicate at many locations
  - ➡ Performance: local reads
  - ➡ Availability: immune from network failure
  - ➡ Fault-tolerance: replicate computation
  - ➡ Scalability: load balancing
- Updates are problematic
  - ➡ Push to all replicas
  - ➡ Conflicts: Consistency?
  - ➡ CAP impossibility

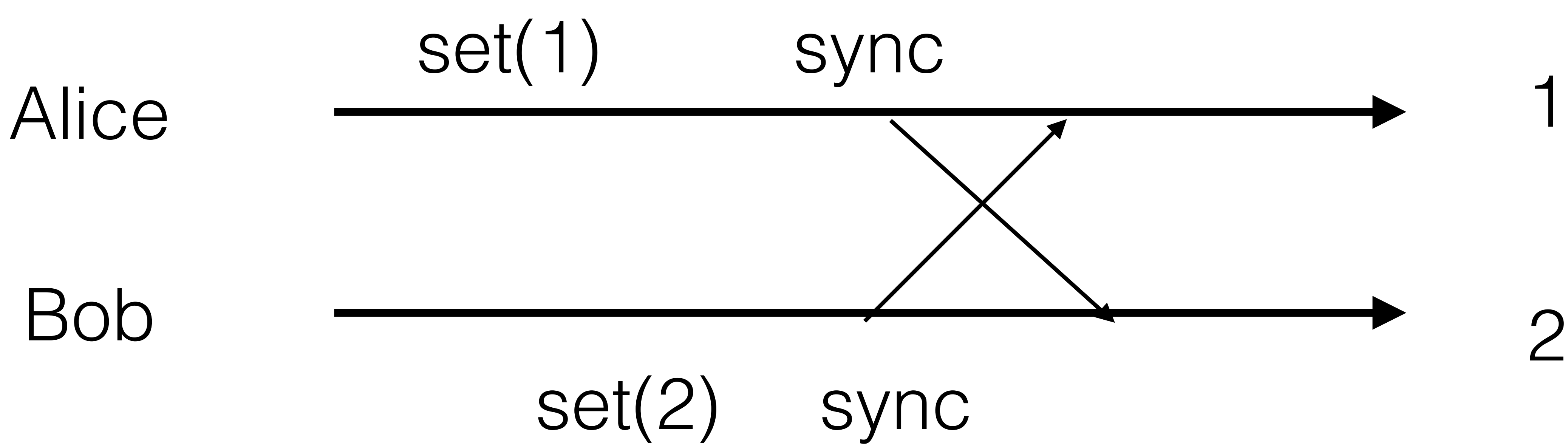


Hands-on

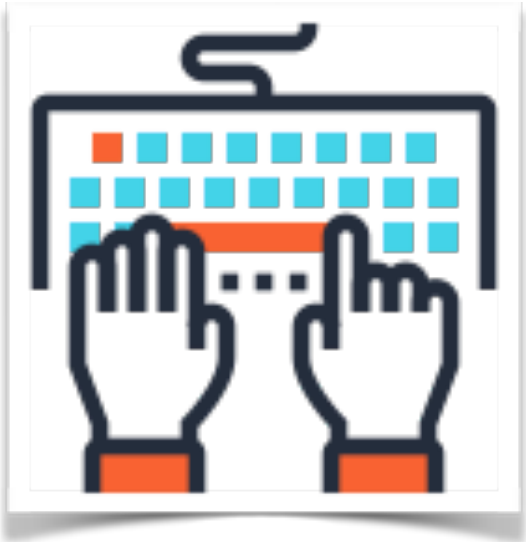
```
$ docker run -i -t cmeiklejohn/crdt-tutorial

alice@127.0.0.1> tutorial:connect()
```

Concurrency anomalies

- Two users update a shared integer (register), each user has a local copy which gets modified, then the changes get pushed to the other user



➡ Divergent replica state



Hands-on

```
Alice: tutorial:mutate(ivar, state_ivar, {set, 1}).
Bob:   tutorial:mutate(ivar, state_ivar, {set, 2}).

Alice: tutorial:sync()
Bob:   tutorial:sync()

Alice: tutorial:query(ivar, state_ivar).
Bob:   tutorial:query(ivar, state_ivar).
```

Failed convergence

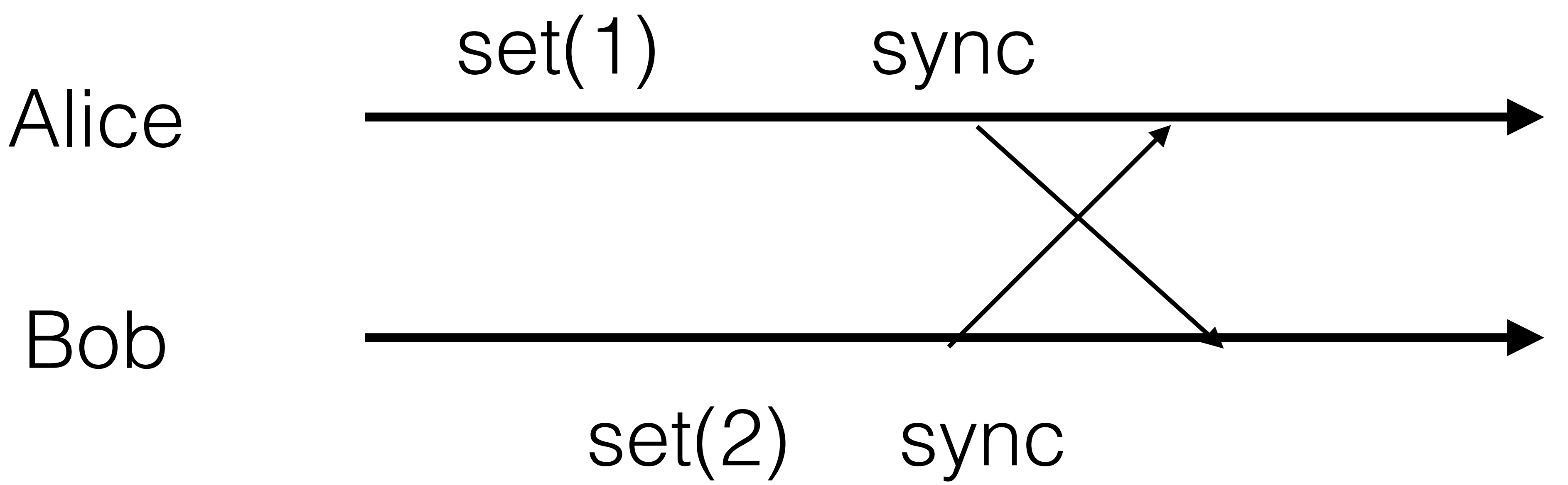
- They try to fix by taking a new register and deciding ahead of time about the value (out-of-band)
- Or just allow one writer

Why not strong consistency?

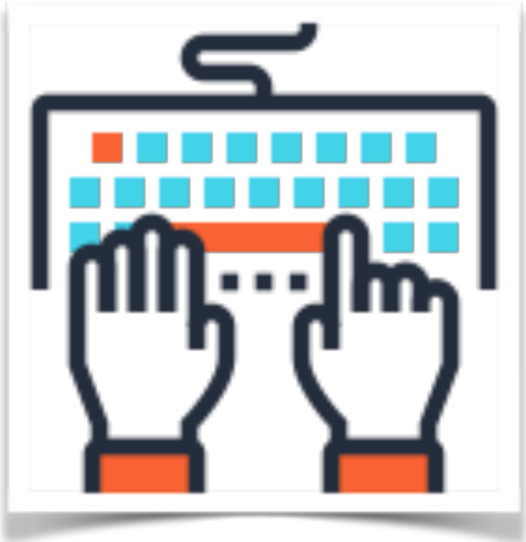
- *Idea*: Use a leader election algorithm to coordinate and order the operations
- Not feasible in highly-concurrent, large-scale replication scenario
    - Geo-replication
    - Mobile computing
- We will not trade availability!
- Fault-tolerance is essential

User-triggered conflict resolution

- Other option: Multi-Value-Register
- Flexible, but cumbersome



- No guarantee of convergence if users do not use the same conflict resolution policy



Hands-on

```
Alice: tutorial:mutate(mvregister, state_mvregister,
            {set, 1473063813940641, 1}).
Bob:   tutorial:mutate(mvregister, state_mvregister,
            {set, 1473063815940641, 2}).

Alice: tutorial:sync()
Bob:   tutorial:sync()

Alice: tutorial:query(mvregister, state_mvregister).
Bob:   tutorial:query(mvregister, state_mvregister).
```

- What happens if both users set the same value?
- How can we fix the inconsistency?

Systematic conflict resolution

- Last-writer-wins strategy:
  - Order all updates by some (logical/physical) time
  - Only the latest update will succeed

```
Alice: tutorial:mutate(lwwregister, state_lwwregister,
          {set, 1473063813940641, 2}).
Bob:   tutorial:mutate(lwwregister, state_lwwregister,
          {set, 1473063815940641, 2}).

Alice: tutorial:sync()
Bob:   tutorial:sync()

Alice: tutorial:query(lwwregister, state_lwwregister).
Bob:   tutorial:query(lwwregister, state_lwwregister).
```

Lost updates

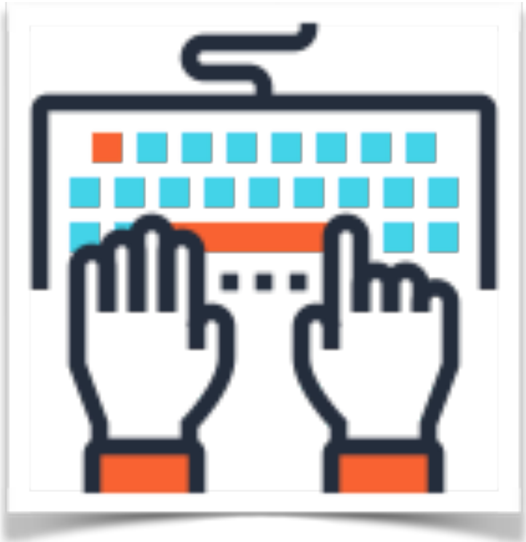
- Both users **increment** the shared integer by 1, starting from the same value

- Lost update, **but it is not observable** by the users as both think their update was successful

- *Idea*:
  - Use UIDs to distinguish operations
  - Replay all operations
  - Requirement: Commutativity

Data-type specific conflict resolution

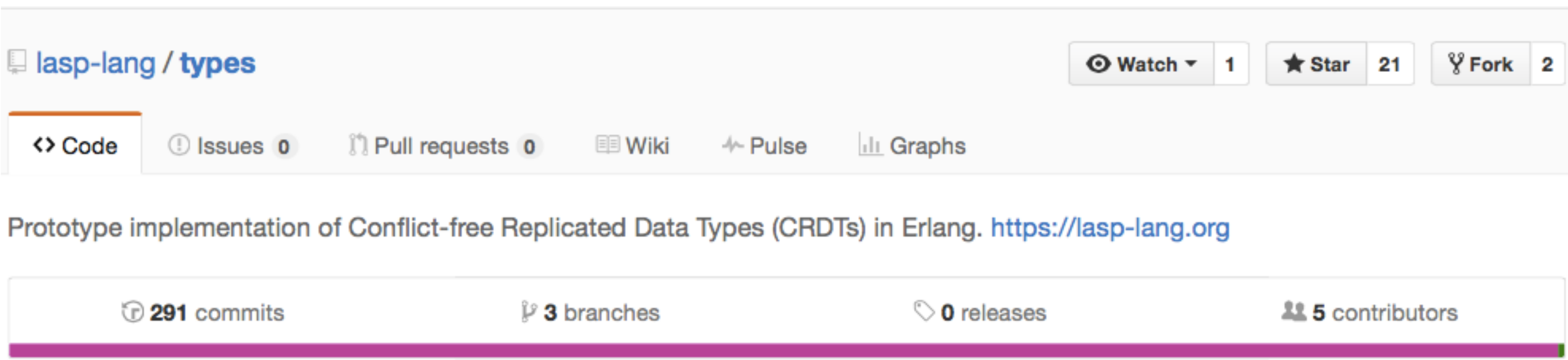
- Need an abstract data type definition
- Example: PNCounter
- Operations: increment, decrement
- Specification:
  - Initial value: 0
  - Increment the counter by 1
  - Decrement the counter by 1



Hands-on

```
Alice: tutorial:mutate(pncounter, state_pncounter, increment).
Bob:   tutorial:mutate(pncounter, state_pncounter, increment).

Alice: tutorial:sync()
Bob:   tutorial:sync()

Alice: tutorial:query(pncounter, state_pncounter).
Bob:   tutorial:query(pncounter, state_pncounter).
```

An Erlang library for CRDTs



https://github.com/lasp-lang/types

Overview: Counters

```
% Grow-only counter
tutorial:mutate(gcounter, state_gcounter, increment).

% Pos-Neg counter
tutorial:mutate(pncounter, state_pncounter, increment).
tutorial:mutate(pncounter, state_pncounter, decrement).
```

Overview: Registers

```
% Last-Writer-Wins register
tutorial:mutate(lwwregister, state_lwwregister,
     {set, Time, Value}).

% Multi-Value register
tutorial:mutate(mvregister, state_mvregister,
     {set, Time, Value}).
```

Correctness requirements

- What do we need to guarantee?
  - [Pure op-based] Causal delivery of updates
  - [State-based] Anti-entropy
  - [Causal-based] Causal per-object anti-entropy

Overview: CRDT Sets

```
%% Observed-Remove Set
tutorial:mutate(orset, state_orset, {add, Value}).
tutorial:mutate(orset, state_orset, {rmv, Value}).

%% Add-Wins Causal Set
tutorial:mutate(awset, state_awset, {add, Value}).
tutorial:mutate(awset, state_awset, {rmv, Value}).
```

CRDT design concept

Same API as sequential abstract data type with concurrency semantics

Commutativity => Concurrent = Sequential

Otherwise, requires arbitration

- Close to sequential version
- Don’t lose updates
- Result doesn't depend on order received
- Stable preconditions

Extending the Set seq. spec.

- Sequential specification of Set:
  - *{true}* add(*e*) *{e ∈ S}*
  - *{true}* rmv(*e*) *{e ∉ S}*
- Commutative (*e ≠ f)*:
  - *{true}* add(*e*) || add(*e*) *{e ∈ S}*
  - *{true}* rmv(*e*) || rmv(*e*) *{e ∉ S}*
  - *{true}* add(*e*) || add(*f*) *{e,f ∈ S}*
  - *{true}* rmv(*e*) || rmv(*f*) *{e, f ∉ S}*
  - *{true}* add(*e*) || rmv(*f*) *{e ∈ S, f ∉ S}*
- Ambiguous:
  - *{true}* add(*e*) || rmv(*e*) *{????}*

add(e) || rem(e)

- *{true}* add(e) || rmv(e) *{????}*
  - ~~linearisable?~~
  - last writer wins? $\{ \quad \text{add}(e) < \text{rmv}(e) \Rightarrow e \notin S$

    $\wedge \quad \text{rmv}(e) < \text{add}(e) \Rightarrow e \in S \}$
  - error state? $\{T_e \in S\}$
  - add wins? $\{e \in S\}$
  - remove wins? $\{e \notin S\}$

Other set designs

- Grow-only set + union merge
  - ➡ No *remove*
- 2P-Set: [Wuu & Bernstein PODC 1984]
  - ➡ Add + tombstones
  - ➡ Add/remove once
  - ➡ Violates sequential spec
- c-set: [Sovran et al., SOSP 2011]
  - ➡ Add/remove counter
  - ➡ Violates sequential spec

Strong eventual consistency ≠ Sequential Consistency

- Consider Set-like object S such that:
  - $\{true\}$ add(e) $\{e \in S\}$
  - $\{true\}$ remove(e) $\{e \notin S\}$
  - $\{true\}$ add(e) || remove(e) $\{e \in S\}$
- Satisfies SEC conditions

$$\{true\} \left|\begin{array}{c} \text{add}(e); \text{remove}(e') \\ \| \\ \text{add}(e'); \text{remove}(e) \end{array}\right| \{e, e' \in S\}$$

- Not sequentially consistent

Concurrent
≠ sequential permutation

- Multivalue Register (Dynamo):
  - *{true}* x ≔ 1 → x ≔ 2 ; *{x=2}*
  - *{true}* x ≔ 2 → x ≔ 1 ; *{x=1}*
  - *{true}* x ≔ 1 || x ≔ 2 *{x={1,2}}*
- Can't be explained sequentially
  - ~~linearisability~~
- Concurrent specification
  - Conflict-free
  - Deterministic

Concurrent
≠ sequential permutation

- Multivalue Register (Dynamo):
  - *{true}* x ≔ 1 → x ≔ 2 *{x=2}*
  - *{true}* x ≔ 2 → x ≔ 1 *{x=1}*
  - *{true}* x ≔ 1 || x ≔ 2 *{x={1,2}}*
- Can't be explained sequentially

Taxonomy of CRDTs

- Operation-based vs. state-based
- Delta CRDTs
- Bounded CRDTs
- Optimized CRDTs (Garbage collection)

CRDT types in Literature

Register
- Last-Writer Wins
- Multi-Value

Set
- Grow-Only
- 2P
- Observed-Remove

Map

Flags (boolean)

Pairs

Counter
- Unlimited
- Restricted ≥0

Graph
- Directed
- Monotonic DAG
- Edit graph

Sequence

And where is the catch?

- Meta-data overhead
  - Version vectors usually used to identify concurrent modifications
  - Grows as O(N), where N is the number of distinguishable modifying entities -> Churn!
- Monotonically growing state with state-based CRDTs (tombstones …) -> Garbage collection necessary
- LWW with physical clocks -> clock skew

Outlook: Composability

- Nested CRDTs
  - Maps, pairs, sequences of (keys to) CRDT objects
  - Recursive, structural merge
  - Examples: Riak DT Maps, JSON CRDT
- Transactional CRDT updates (-> Antidote)