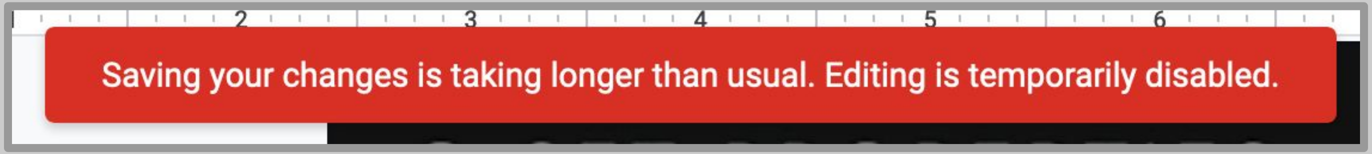
Real-time Collaboration with CRDTs

Brown Bag — Aug 21, 2020


The document I have to edit
This document is locked by user@company.com
Unlock?
Edit


Saving your changes is taking longer than usual. Editing is temporarily disabled.

Outline

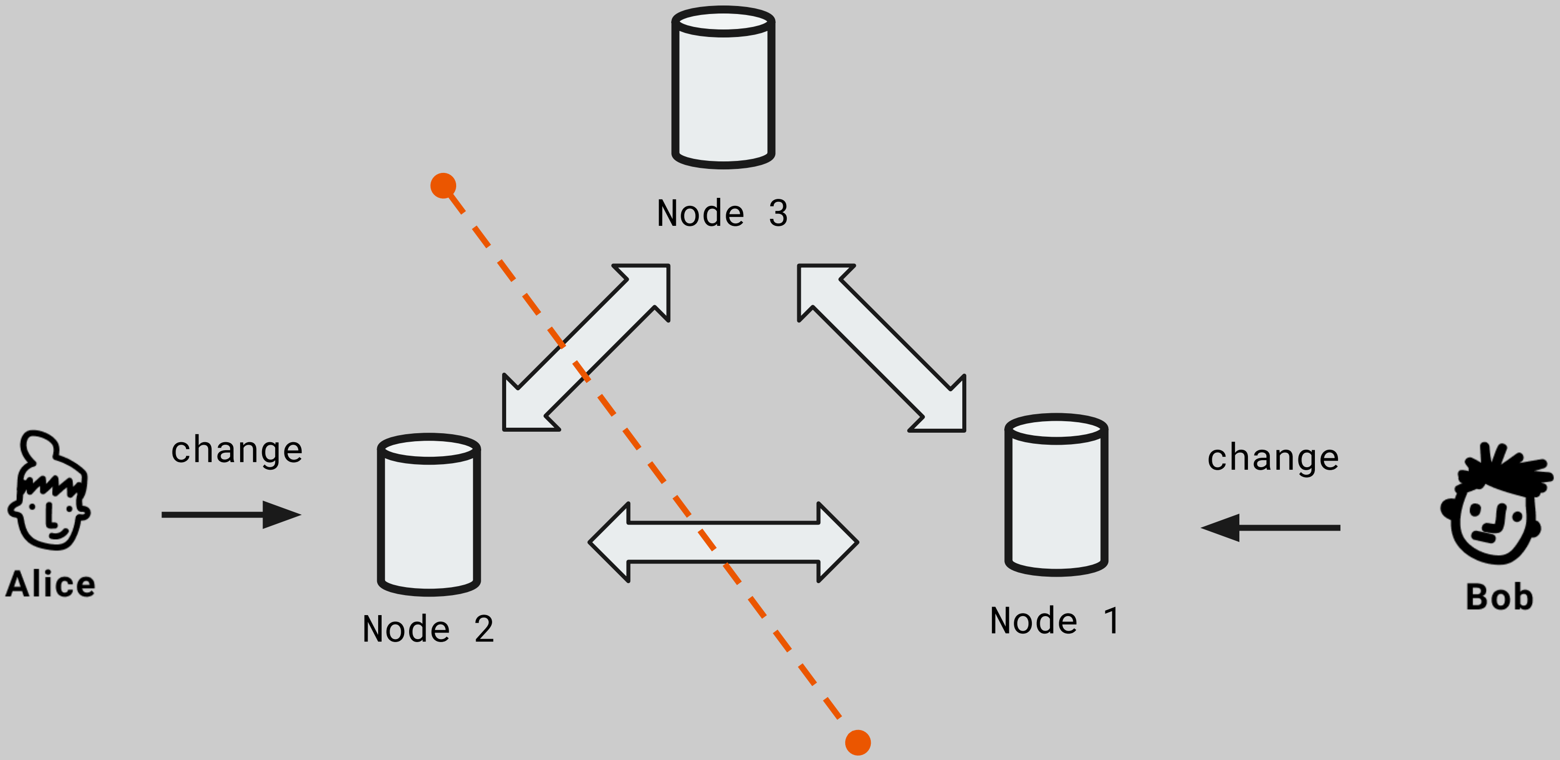
Why CRDTs?

Code Example*

A brief look at Automerge

* We are building our own G-Counter!

Why CRDTs?

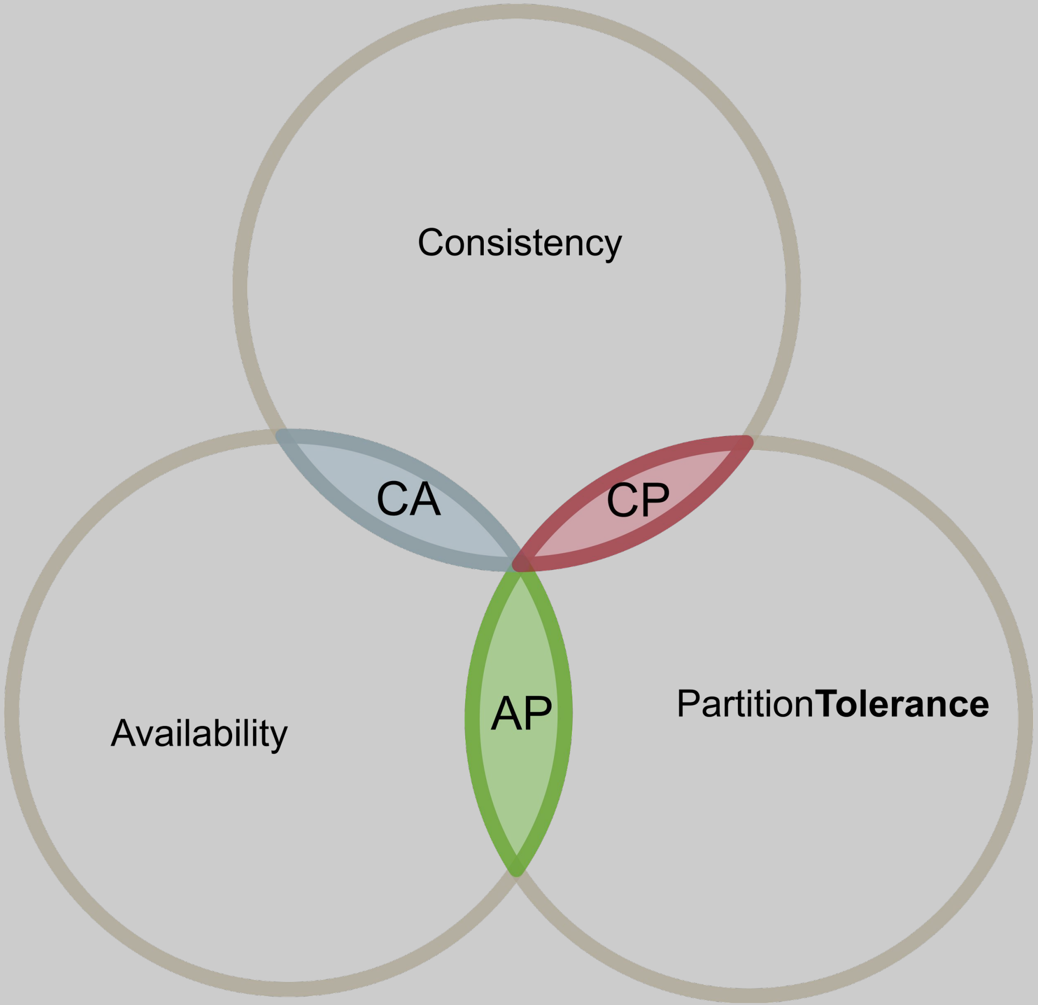


Replication

Wishlist

1. Multiple people can edit the same document concurrently (A)
2. App works offline / State is available locally (P)
3. Consistent state across all devices (C)

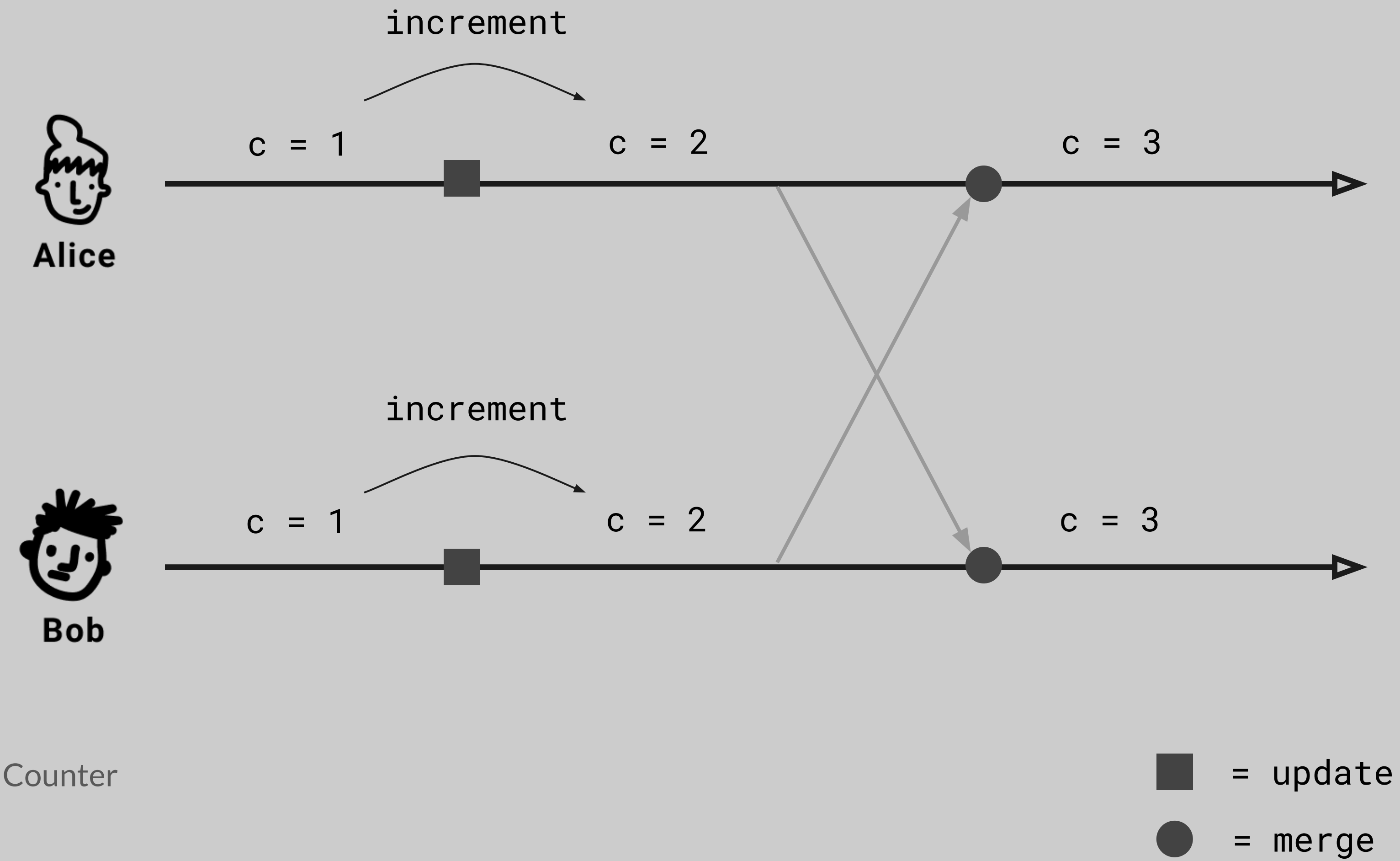
CAP Theorem



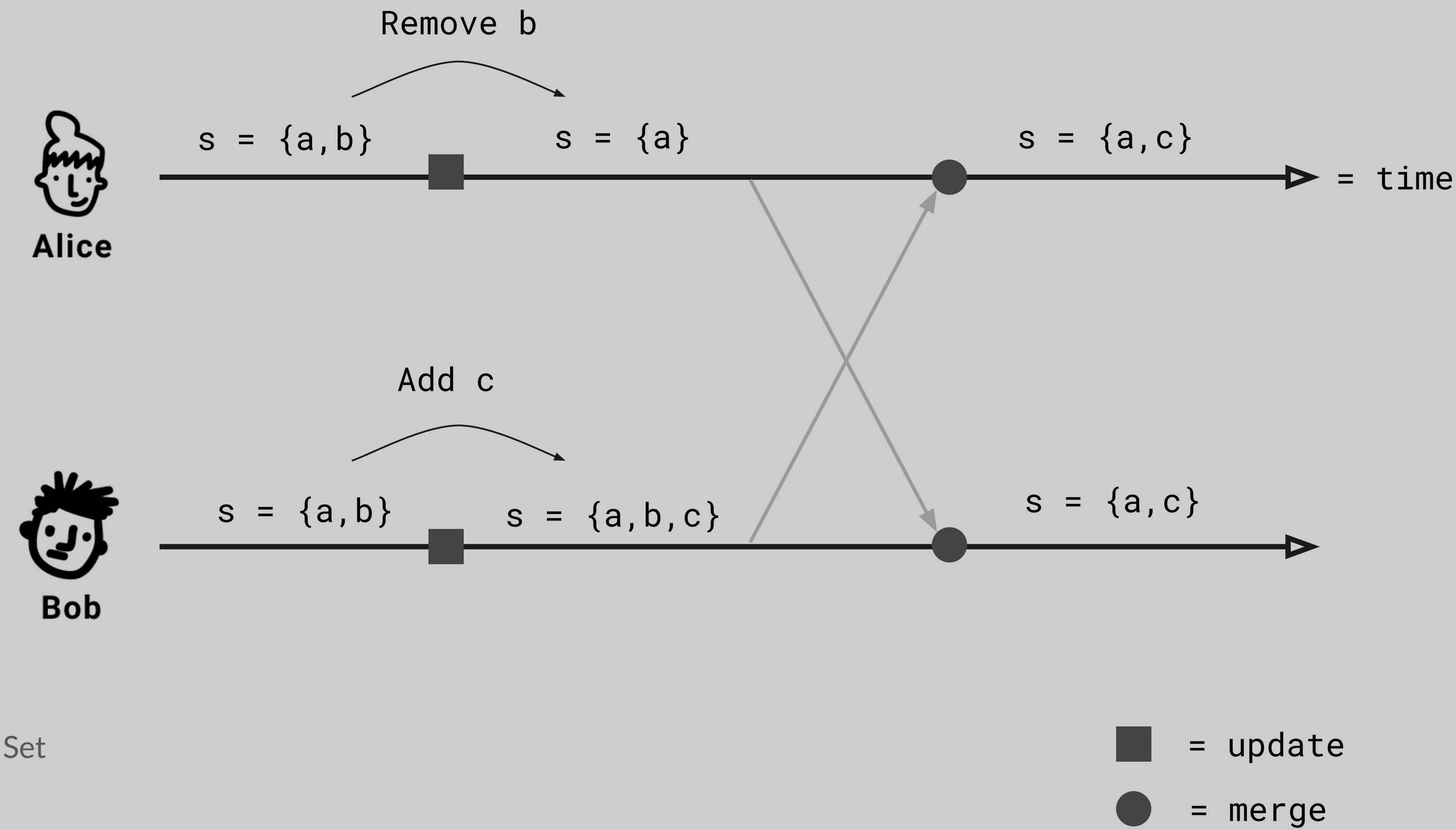
Conflict free Replicated Data Types

- Distributed data type
- Strong **Eventual** Consistency
- Well defined interface
- Mathematical sound (as opposed to Operational Transformations)*
- "Holy grail"

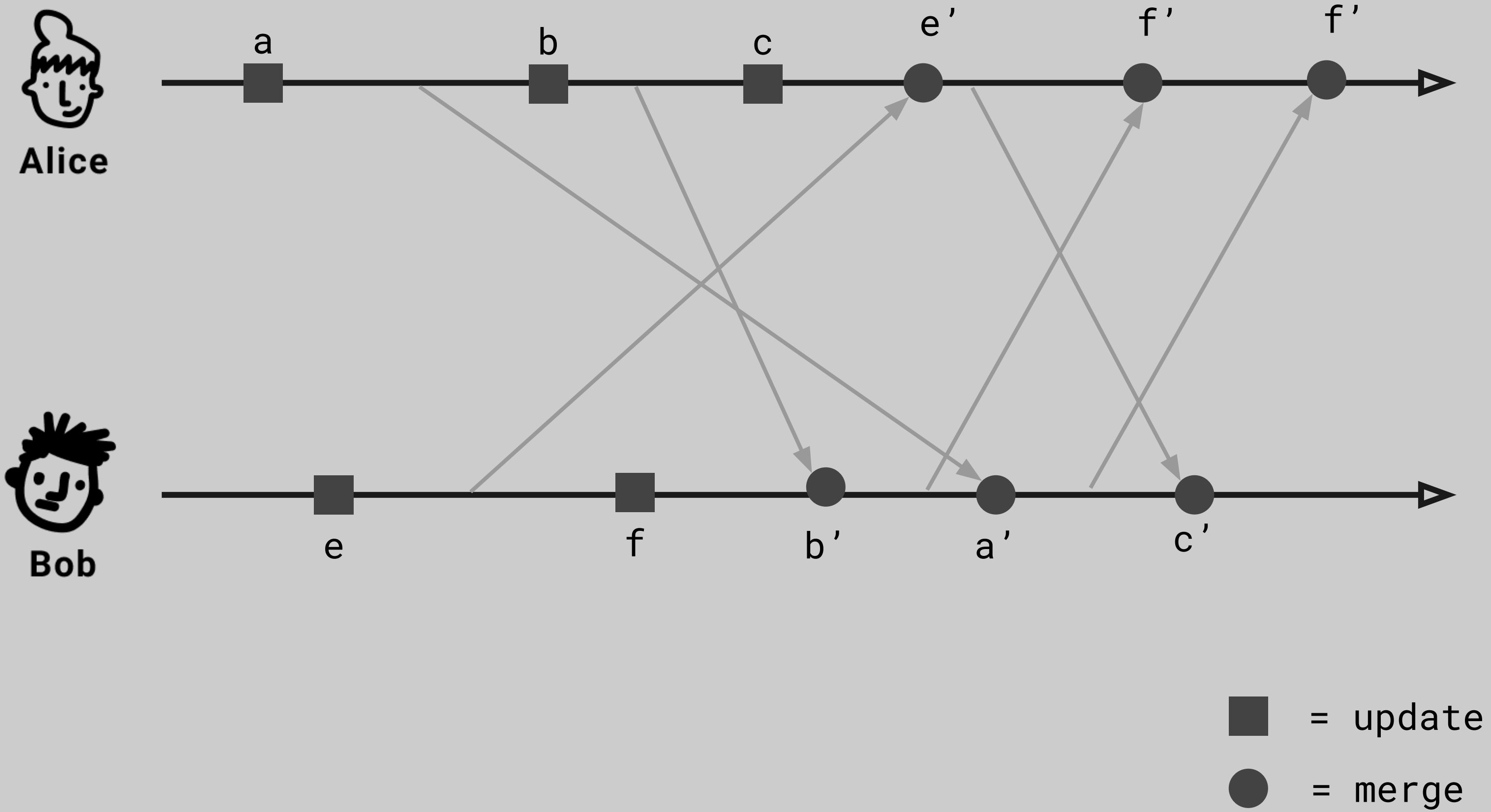
*but must be 100% error free!



Counter


Remove b
Alice
s = {a,b}
s = {a}
s = {a,c}
= time
Add c
Bob
s = {a,b}
s = {a,b,c}
s = {a,c}
= update
= merge

Set


Alice
Bob
a
b
c
e’
f’
f’
e
f
b’
a’
c’
= update
= merge

CRDT Merge

...or how to keep things in sync

1. **Commutative:**

   $x \bullet y = y \bullet x$

2. **Associative**

   $(x \bullet y) \bullet z = x \bullet (y \bullet z)$

3. **Idempotent:**

   $x \bullet x = x$

Conflict free Replicated Data Types

1. Any replica can be modified without coordinating with another replica
2. When two replicas have seen the same set of updates, they reach the same state

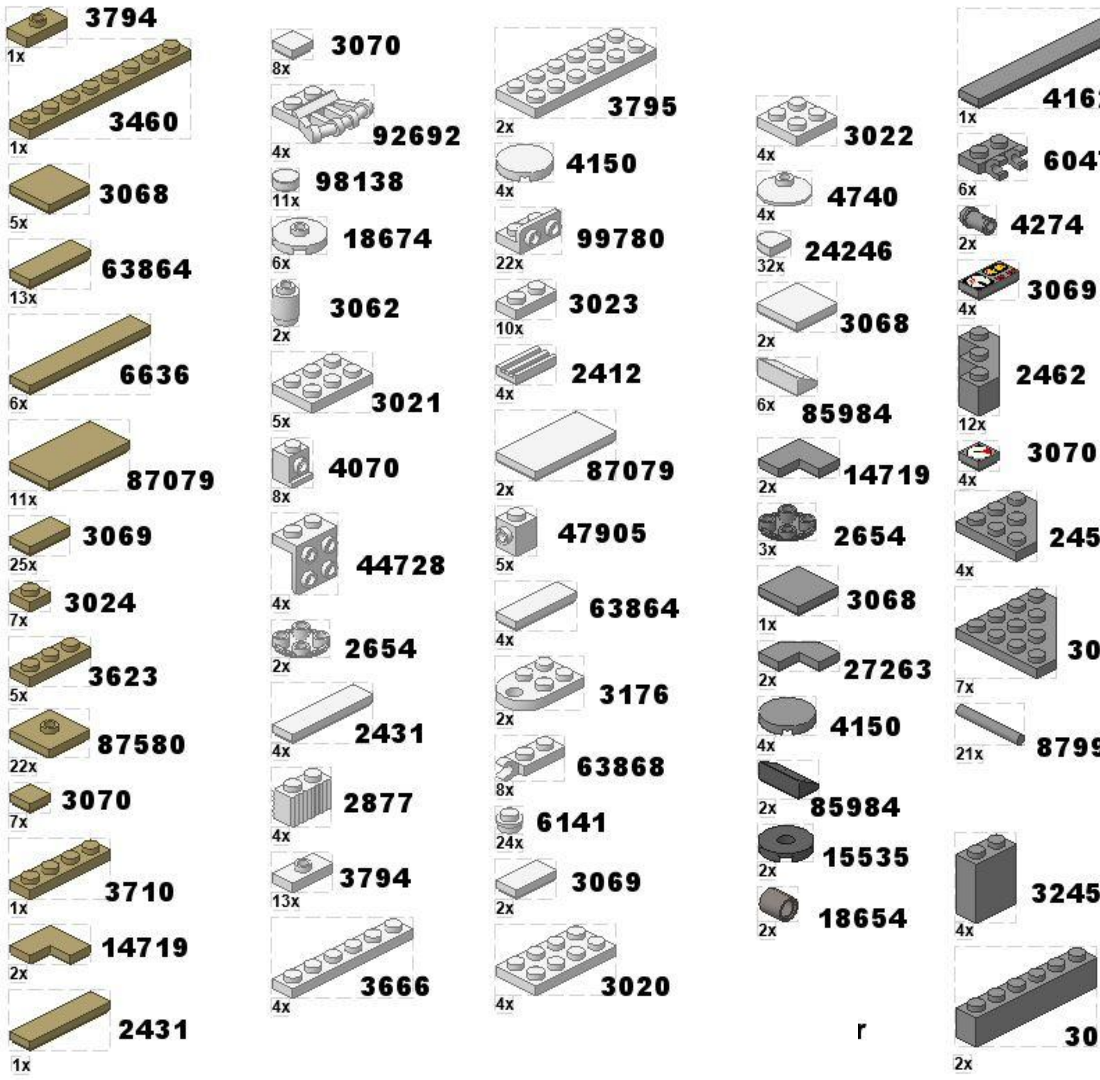
CRDT Examples

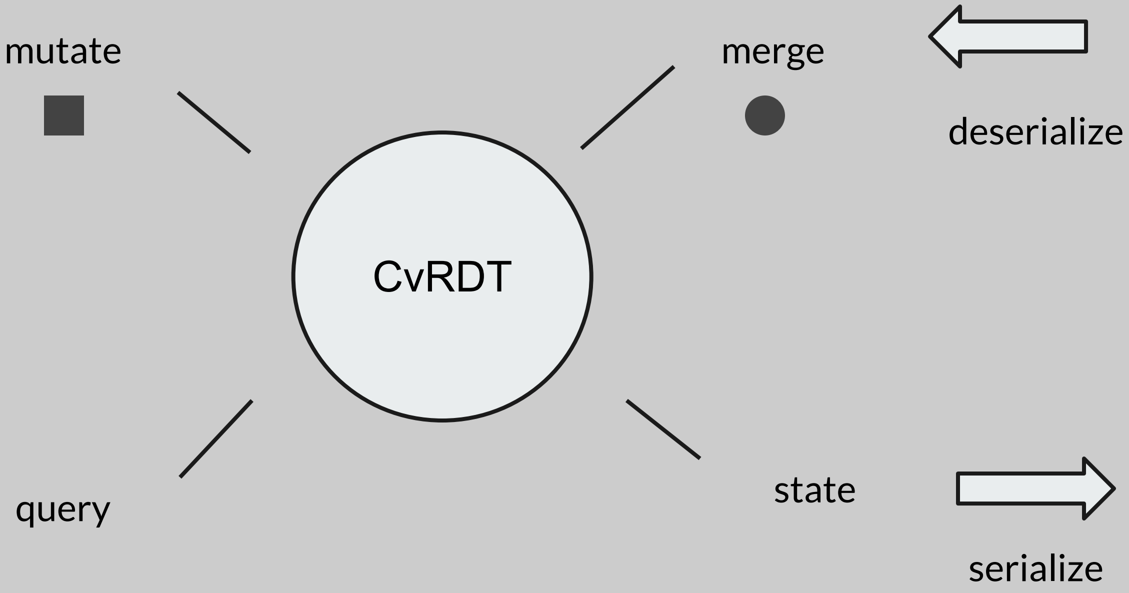
1. Figma
2. Atom Teletype
3. Apple Notes
4. TomTom

Google Docs → Operational Transformations (OT)

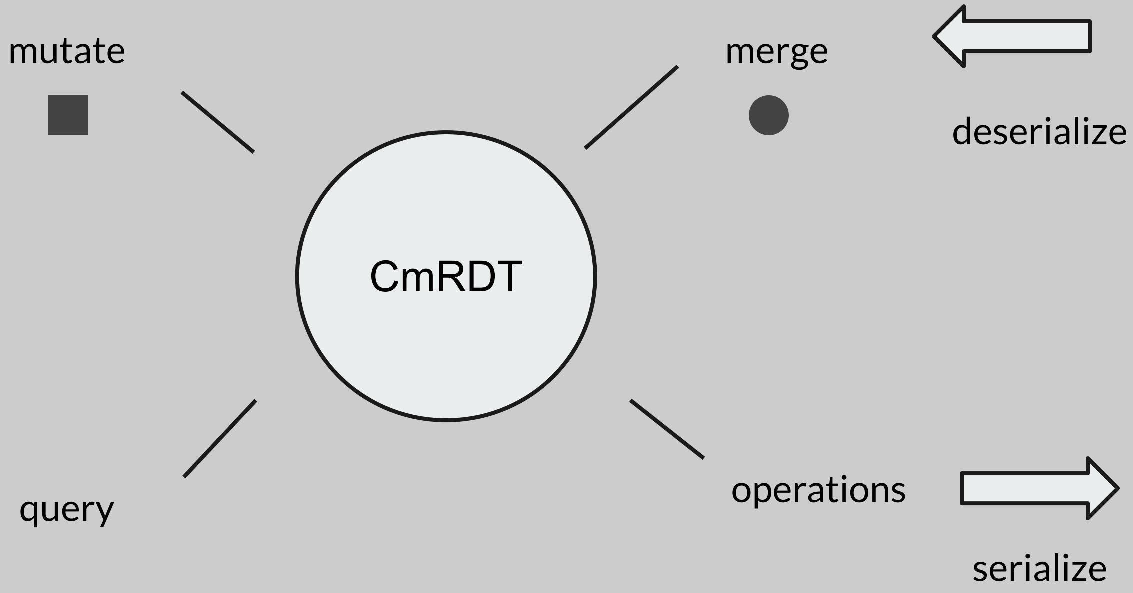
“Known CRDTs”

G-Counter
PN-Counter
G-Set
2P-Set
OR-Set
LWW-Set
AWOR-Set
LWW-Register
Multi-Value Register
PN-Set
DW-Flag
Replicated Growable Array (RGA)
...

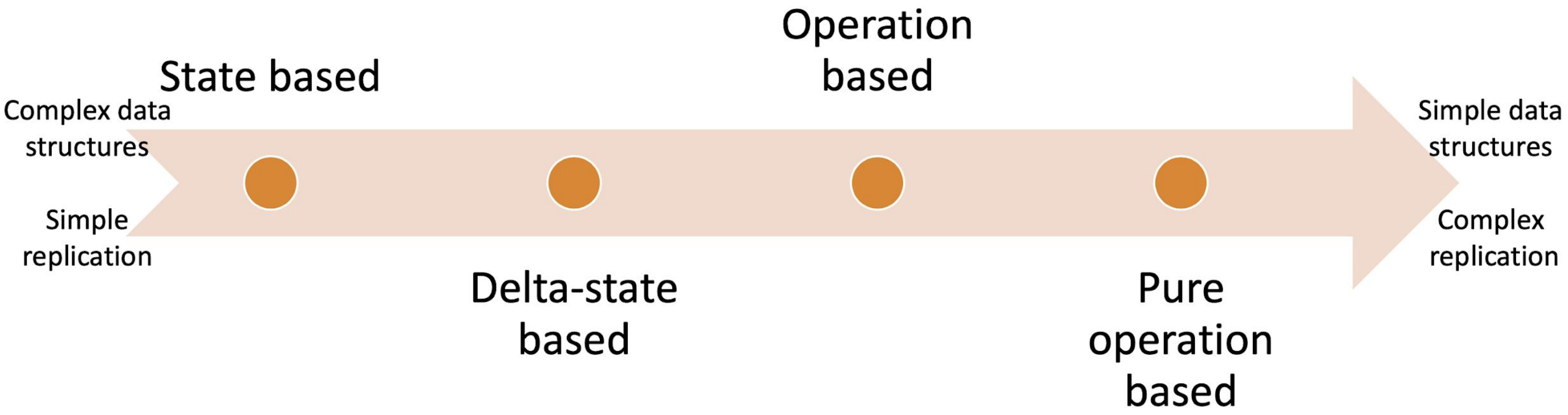




State-based CRDT

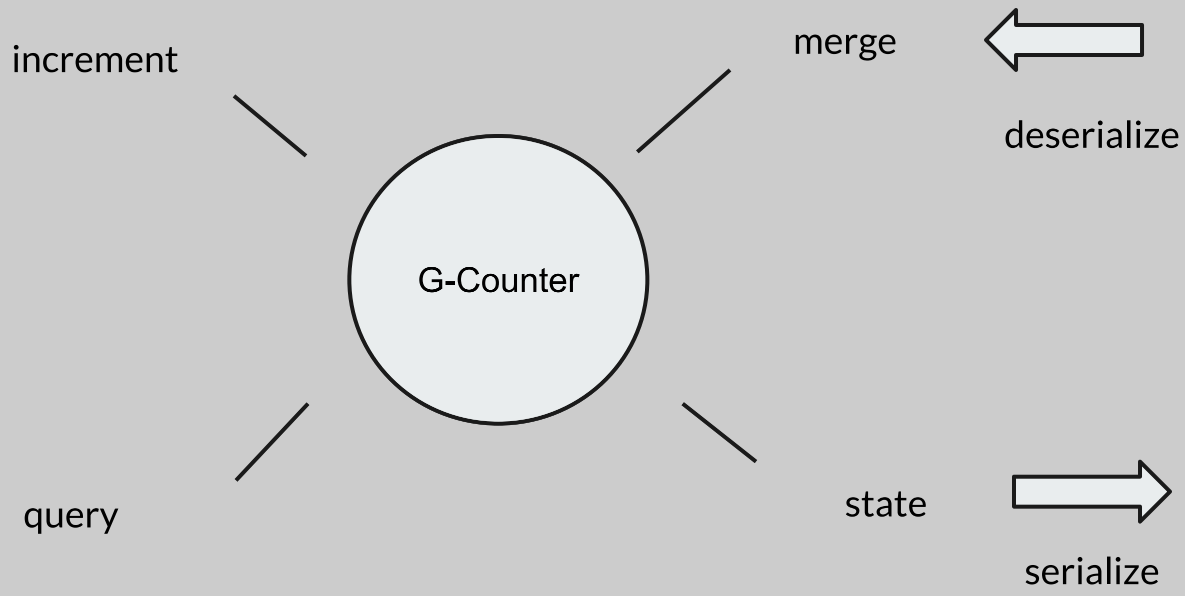


Operations-based CRDT



State based
Operation based
Complex data structures
Simple replication
Simple data structures
Complex replication
Delta-state based
Pure operation based

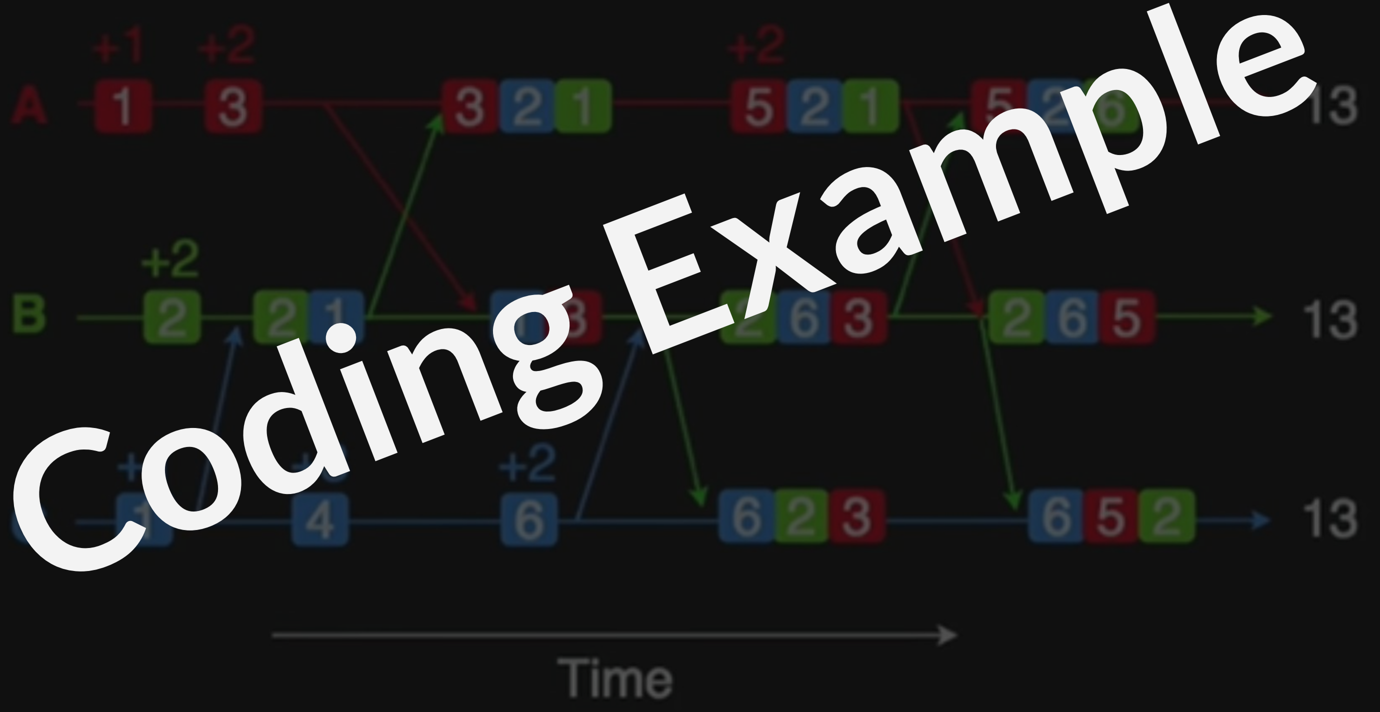
The G-Counter


increment
merge
deserialize
G-Counter
query
state
serialize

G-Counter

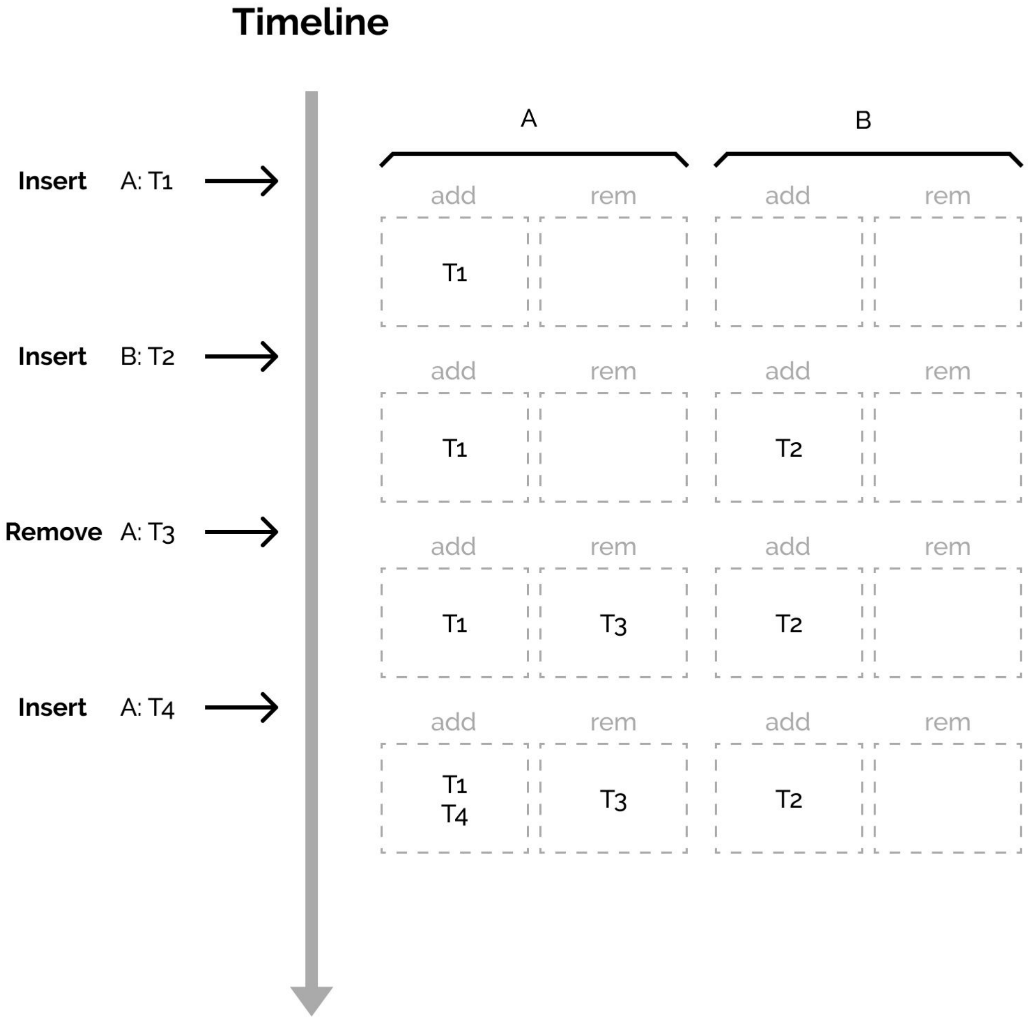
GROW-ONLY COUNTER

G-COUNTER

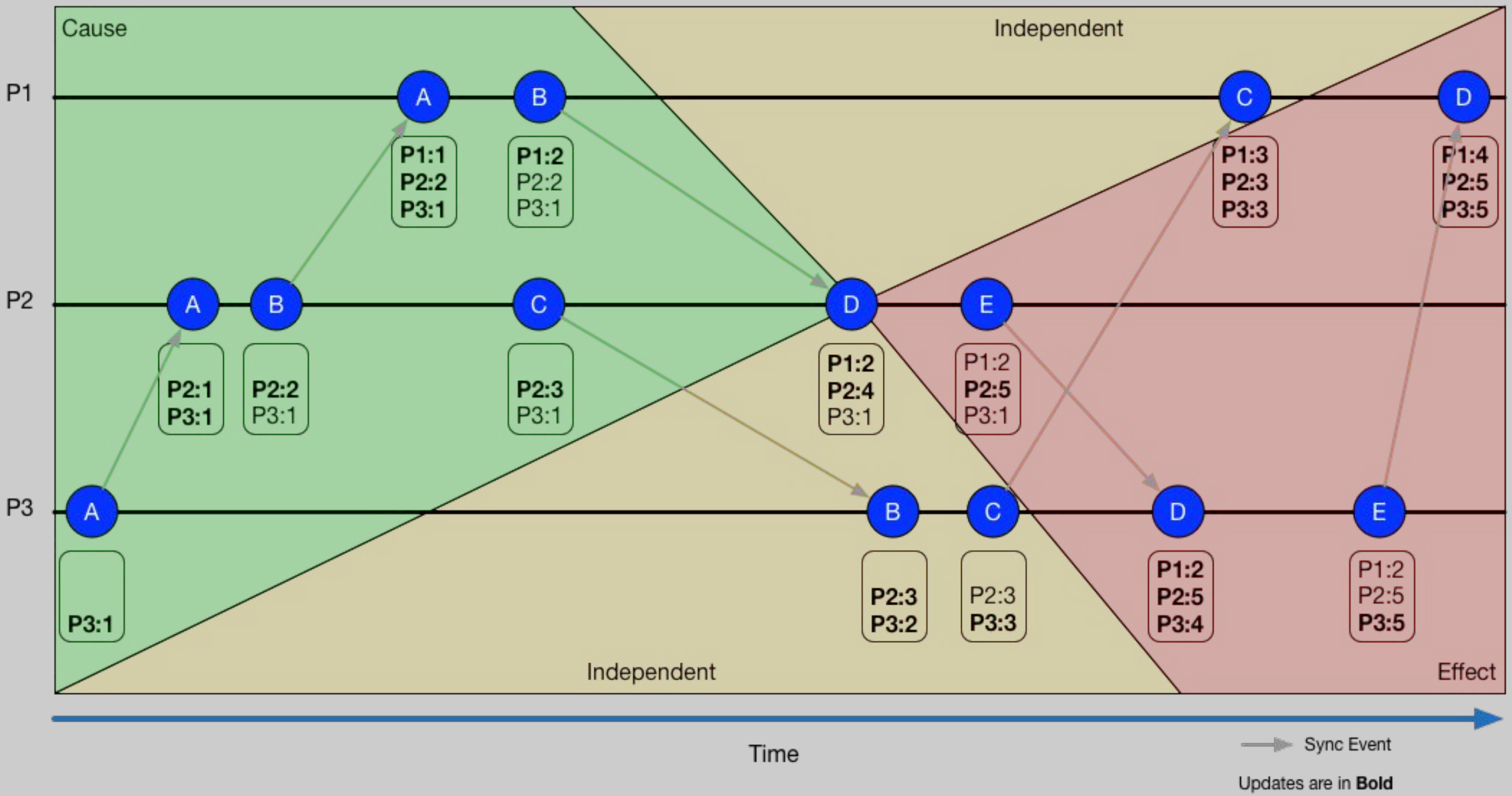


OR-Set

Add-Wins-Observed-Remove-Set



Logical Clock

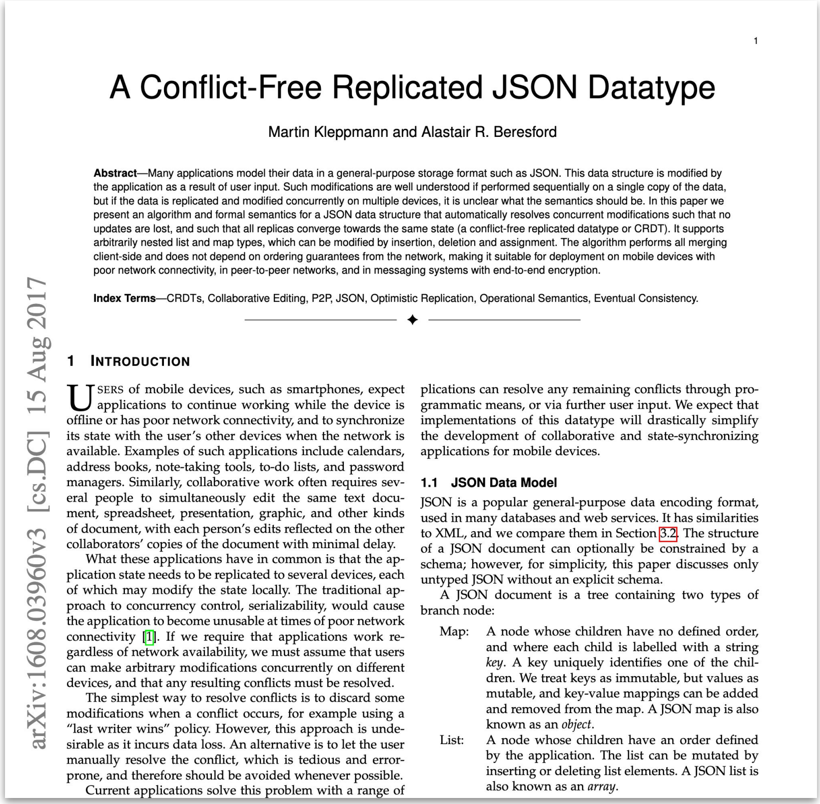


Vector Clock

CRDTs Implementations in JavaScript

- Automerge
- Y.js
- delta-crdts

https://arxiv.org/abs/1608.03960



A Conflict-Free Replicated JSON Datatype

Martin Kleppmann and Alastair R. Beresford

Abstract—Many applications model their data in a general-purpose storage format such as JSON. This data structure is modified by the application as a result of user input. Such modifications are well understood if performed sequentially on a single copy of the data, but if the data is replicated and modified concurrently on multiple devices, it is unclear what the semantics should be. In this paper we present an algorithm and formal semantics for a JSON data structure that automatically resolves concurrent modifications such that no updates are lost, and such that all replicas converge towards the same state (a conflict-free replicated datatype or CRDT). It supports arbitrarily nested list and map types, which can be modified by insertion, deletion and assignment. The algorithm performs all merging client-side and does not depend on ordering guarantees from the network, making it suitable for deployment on mobile devices with poor network connectivity, in peer-to-peer networks, and in messaging systems with end-to-end encryption.

Index Terms—CRDTs, Collaborative Editing, P2P, JSON, Optimistic Replication, Operational Semantics, Eventual Consistency.

1 Introduction

Users of mobile devices, such as smartphones, expect applications to continue working while the device is offline or has poor network connectivity, and to synchronize its state with the user's other devices when the network is available. Examples of such applications include calendars, address books, note-taking tools, to-do lists, and password managers. Similarly, collaborative work often requires several people to simultaneously edit the same text document, spreadsheet, presentation, graphic, and other kinds of document, with each person's edits reflected on the other collaborators' copies of the document with minimal delay.

What these applications have in common is that the application state needs to be replicated to several devices, each of which may modify the state locally. The traditional approach to concurrency control, serializability, would cause the application to become unusable at times of poor network connectivity [1]. If we require that applications work regardless of network availability, we must assume that users can make arbitrary modifications concurrently on different devices, and that any resulting conflicts must be resolved.

The simplest way to resolve conflicts is to discard some modifications when a conflict occurs, for example using a "last writer wins" policy. However, this approach is undesirable as it incurs data loss. An alternative is to let the user manually resolve the conflict, which is tedious and error-prone, and therefore should be avoided whenever possible.

Current applications solve this problem with a range of applications can resolve any remaining conflicts through programmatic means, or via further user input. We expect that implementations of this datatype will drastically simplify the development of collaborative and state-synchronizing applications for mobile devices.

1.1 JSON Data Model

JSON is a popular general-purpose data encoding format, used in many databases and web services. It has similarities to XML, and we compare them in Section 3.2. The structure of a JSON document can optionally be constrained by a schema; however, for simplicity, this paper discusses only untyped JSON without an explicit schema.

A JSON document is a tree containing two types of branch node:

Map: A node whose children have no defined order, and where each child is labelled with a string *key*. A key uniquely identifies one of the children. We treat keys as immutable, but values as mutable, and key-value mappings can be added and removed from the map. A JSON map is also known as an *object*.

List: A node whose children have an order defined by the application. The list can be mutated by inserting or deleting list elements. A JSON list is also known as an *array*.

$$\textsc{Descend}\quad ctx,\ \mathsf{op}(id, deps, \mathsf{cursor}(\langle k_1, k_2, \dots, k_{n-1}\rangle, k_n), mut) \implies ctx'[\,k_1 \mapsto child'\,]$$

$$\textsc{Child-Get}\ \frac{k \in \mathrm{dom}(ctx)}{ctx,\ k \implies ctx(k)} \qquad \textsc{Child-Map}\ \frac{\mathsf{mapT}(k) \notin \mathrm{dom}(ctx)}{ctx,\ \mathsf{mapT}(k) \implies \{\}} \qquad \textsc{Child-List}\ \frac{\mathsf{listT}(k) \notin \mathrm{dom}(ctx)}{ctx,\ \mathsf{listT}(k) \implies \{\,\mathsf{next}(\mathsf{head}) \mapsto \mathsf{tail}\,\}}$$

$$\textsc{Child-Reg}\ \frac{\mathsf{regT}(k) \notin \mathrm{dom}(ctx)}{ctx,\ \mathsf{regT}(k) \implies \{\}} \qquad \textsc{Presence}_1\ \frac{\mathsf{pres}(k) \in \mathrm{dom}(ctx)}{ctx,\ \mathsf{pres}(k) \implies ctx(\mathsf{pres}(k))} \qquad \textsc{Presence}_2\ \frac{\mathsf{pres}(k) \notin \mathrm{dom}(ctx)}{ctx,\ \mathsf{pres}(k) \implies \{\}}$$

$$\textsc{Add-ID}_1\ \frac{mut \neq \mathsf{delete} \qquad k_{tag} \in \{\mathsf{mapT}(k), \mathsf{listT}(k), \mathsf{regT}(k)\} \qquad ctx,\ \mathsf{pres}(k) \implies pres}{ctx,\ \mathsf{addId}(k_{tag}, id, mut) \implies ctx[\,\mathsf{pres}(k) \mapsto pres \cup \{id\}\,]} \qquad \textsc{Add-ID}_2\ \frac{mut = \mathsf{delete}}{ctx,\ \mathsf{addId}(k_{tag}, id, mut) \implies ctx}$$

$$\textsc{Assign}\ \frac{val \neq \texttt{[]} \wedge val \neq \texttt{\{\}} \qquad ctx,\ \mathsf{clear}(deps, \mathsf{regT}(k)) \implies ctx',\ pres \qquad ctx',\ \mathsf{addId}(\mathsf{regT}(k), id, \mathsf{assign}(val)) \implies ctx'' \qquad ctx'',\ \mathsf{regT}(k) \implies child}{ctx,\ \mathsf{op}(id, deps, \mathsf{cursor}(\langle\rangle, k), \mathsf{assign}(val)) \implies ctx''[\,\mathsf{regT}(k) \mapsto child[\,id \mapsto val\,]\,]}$$

$$\textsc{Empty-Map}\ \frac{val = \texttt{\{\}} \qquad ctx,\ \mathsf{clearElem}(deps, k) \implies ctx',\ pres \qquad ctx',\ \mathsf{addId}(\mathsf{mapT}(k), id, \mathsf{assign}(val)) \implies ctx'' \qquad ctx'',\ \mathsf{mapT}(k) \implies child}{ctx,\ \mathsf{op}(id, deps, \mathsf{cursor}(\langle\rangle, k), \mathsf{assign}(val)) \implies ctx''[\,\mathsf{mapT}(k) \mapsto child\,]}$$

$$\textsc{Empty-List}\ \frac{val = \texttt{[]} \qquad ctx,\ \mathsf{clearElem}(deps, k) \implies ctx',\ pres \qquad ctx',\ \mathsf{addId}(\mathsf{listT}(k), id, \mathsf{assign}(val)) \implies ctx'' \qquad ctx'',\ \mathsf{listT}(k) \implies child}{ctx,\ \mathsf{op}(id, deps, \mathsf{cursor}(\langle\rangle, k), \mathsf{assign}(val)) \implies ctx''[\,\mathsf{listT}(k) \mapsto child\,]}$$

$$\textsc{Insert}_1\ \frac{ctx(\mathsf{next}(prev)) = next \qquad next < id \vee next = \mathsf{tail} \qquad ctx,\ \mathsf{op}(id, deps, \mathsf{cursor}(\langle\rangle, id), \mathsf{assign}(val)) \implies ctx'}{ctx,\ \mathsf{op}(id, deps, \mathsf{cursor}(\langle\rangle, prev), \mathsf{insert}(val)) \implies ctx'[\,\mathsf{next}(prev) \mapsto id,\ \mathsf{next}(id) \mapsto next\,]}$$

$$\textsc{Insert}_2\ \frac{ctx(\mathsf{next}(prev)) = next \qquad id < next \qquad ctx,\ \mathsf{op}(id, deps, \mathsf{cursor}(\langle\rangle, next), \mathsf{insert}(val)) \implies ctx'}{ctx,\ \mathsf{op}(id, deps, \mathsf{cursor}(\langle\rangle, prev), \mathsf{insert}(val)) \implies ctx'}$$

$$\textsc{Delete}\ \frac{ctx,\ \mathsf{clearElem}(deps, k) \implies ctx',\ pres}{ctx,\ \mathsf{op}(id, deps, \mathsf{cursor}(\langle\rangle, k), \mathsf{delete}) \implies ctx'}$$

$$\textsc{Clear-Elem}\ \frac{ctx,\ \mathsf{clearAny}(deps, k) \implies ctx', pres_1 \qquad ctx',\ \mathsf{pres}(k) \implies pres_2 \qquad pres_3 = pres_1 \cup pres_2 \setminus deps}{ctx,\ \mathsf{clearElem}(deps, k) \implies ctx'[\,\mathsf{pres}(k) \mapsto pres_3\,],\ pres_3}$$

$$\textsc{Clear-Any}\ \frac{\begin{array}{c}ctx,\ \mathsf{clear}(deps, \mathsf{mapT}(k))\\ \implies ctx_1,\ pres_1\end{array} \qquad \begin{array}{c}ctx_1,\ \mathsf{clear}(deps, \mathsf{listT}(k))\\ \implies ctx_2,\ pres_2\end{array} \qquad \begin{array}{c}ctx_2,\ \mathsf{clear}(deps, \mathsf{regT}(k))\\ \implies ctx_3,\ pres_3\end{array}}{ctx,\ \mathsf{clearAny}(deps, k) \implies ctx_3,\ pres_1 \cup pres_2 \cup pres_3}$$

$$\textsc{Clear-None}\ \frac{k \notin \mathrm{dom}(ctx)}{ctx,\ \mathsf{clear}(deps, k) \implies ctx,\ \{\}}$$

$$\textsc{Clear-Reg}\ \frac{\mathsf{regT}(k) \in \mathrm{dom}(ctx) \qquad concurrent = \{id \mapsto v \mid (id \mapsto v) \in ctx(\mathsf{regT}(k)) \wedge id \notin deps\}}{ctx,\ \mathsf{clear}(deps, \mathsf{regT}(k)) \implies ctx[\,\mathsf{regT}(k) \mapsto concurrent\,],\ \mathrm{dom}(concurrent)}$$

$$\textsc{Clear-Map}_1\ \frac{\mathsf{mapT}(k) \in \mathrm{dom}(ctx) \qquad ctx(\mathsf{mapT}(k)),\ \mathsf{clearMap}(deps, \{\}) \implies cleared,\ pres}{ctx,\ \mathsf{clear}(deps, \mathsf{mapT}(k)) \implies ctx[\,\mathsf{mapT}(k) \mapsto cleared\,],\ pres}$$

$$\textsc{Clear-Map}_2\ \frac{\begin{array}{c}k \in \mathrm{keys}(ctx)\\ \wedge\, k \notin done\end{array} \qquad \begin{array}{c}ctx,\ \mathsf{clearElem}(deps, k)\\ \implies ctx', pres_1\end{array} \qquad \begin{array}{c}ctx',\ \mathsf{clearMap}(deps, done \cup \{k\})\\ \implies ctx'', pres_2\end{array}}{ctx,\ \mathsf{clearMap}(deps, done) \implies ctx'',\ pres_1 \cup pres_2}$$

$$\textsc{Clear-Map}_3\ \frac{done = \mathrm{keys}(ctx)}{ctx,\ \mathsf{clearMap}(deps, done) \implies ctx,\ \{\}}$$

$$\textsc{Clear-List}_1\ \frac{\mathsf{listT}(k) \in \mathrm{dom}(ctx) \qquad ctx(\mathsf{listT}(k)),\ \mathsf{clearList}(deps, \mathsf{head}) \implies cleared,\ pres}{ctx,\ \mathsf{clear}(deps, \mathsf{listT}(k)) \implies ctx[\,\mathsf{listT}(k) \mapsto cleared\,],\ pres}$$

$$\textsc{Clear-List}_2\ \frac{\begin{array}{c}k \neq \mathsf{tail}\,\wedge\\ ctx(\mathsf{next}(k)) = next\end{array} \qquad \begin{array}{c}ctx,\ \mathsf{clearElem}(deps, k)\\ \implies ctx', pres_1\end{array} \qquad \begin{array}{c}ctx',\ \mathsf{clearList}(deps, next)\\ \implies ctx'', pres_2\end{array}}{ctx,\ \mathsf{clearList}(deps, k) \implies ctx'',\ pres_1 \cup pres_2}$$

$$\textsc{Clear-List}_3\ \frac{k = \mathsf{tail}}{ctx,\ \mathsf{clearList}(deps, k) \implies ctx,\ \{\}}$$

$$\textsc{Make-Assign}\ \frac{A_p,\ expr \implies cur \qquad val : \mathrm{VAL} \qquad A_p,\ \mathsf{makeOp}(cur, \mathsf{assign}(val)) \implies A'_p}{A_p,\ expr\ := val \implies A'_p}$$

$$\textsc{Make-Insert}\ \frac{A_p,\ expr \implies cur \qquad val : \mathrm{VAL} \qquad A_p,\ \mathsf{makeOp}(cur, \mathsf{insert}(val)) \implies A'_p}{A_p,\ expr.\mathsf{insertAfter}(val) \implies A'_p}$$

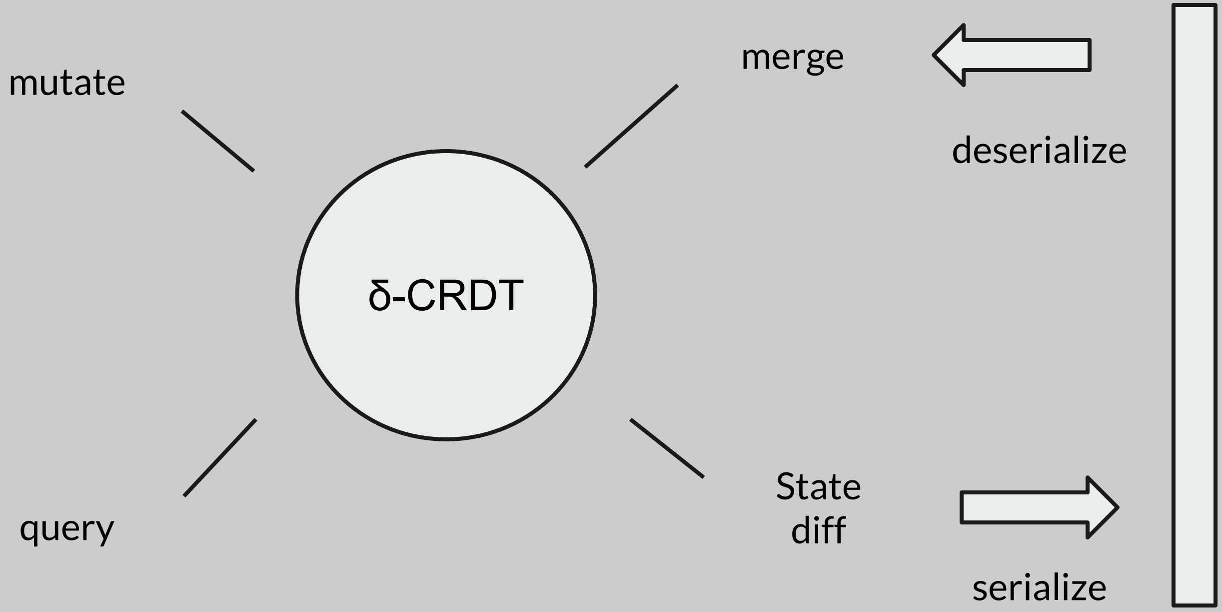
$$\textsc{Make-Delete}\ \frac{A_p,\ expr \implies cur \qquad A_p,\ \mathsf{makeOp}(cur, \mathsf{delete}) \implies A'_p}{A_p,\ expr.\mathsf{delete} \implies A'_p}$$

$$\textsc{Make-Op}\ \frac{ctr = \max(\{0\} \cup \{c_i \mid (c_i, p_i) \in A_p(\mathsf{ops})\} \qquad A_p,\ \mathsf{apply}(\mathsf{op}((ctr+1, p), A_p(\mathsf{ops}), cur, mut)) \implies A'_p}{A_p,\ \mathsf{makeOp}(cur, mut) \implies A'_p}$$

$$\textsc{Apply-Local}\ \frac{A_p,\ op \implies A'_p}{A_p,\ \mathsf{apply}(op) \implies A'_p[\,\mathsf{queue} \mapsto A'_p(\mathsf{queue}) \cup \{op\},\ \mathsf{ops} \mapsto A'_p(\mathsf{ops}) \cup \{op.id\}\,]}$$

Automerge

1. Implements a CRDT for a JSON-like data structure
2. State based CRDT, but supports deltas
3. Undo/Redo for free!
4. Does not implement a network protocol



Delta State-based CRDT

CLI Demo Automerge
Video Demo Automerge

state is immutable
(Automerge.change() returns
new object)

"Commit message"
(optional)

```
state = Automerge.change(state, "Add todo item",
    (doc) => {
        doc.todos.push({
            title: "Buy milk",
            done: false
        })
    })
```

doc is mutable
only within
this block
(Proxy object)

Plain old JS
objects and methods

```
doc.todos.push({
  title : "Buy milk",
  done : false
})
```

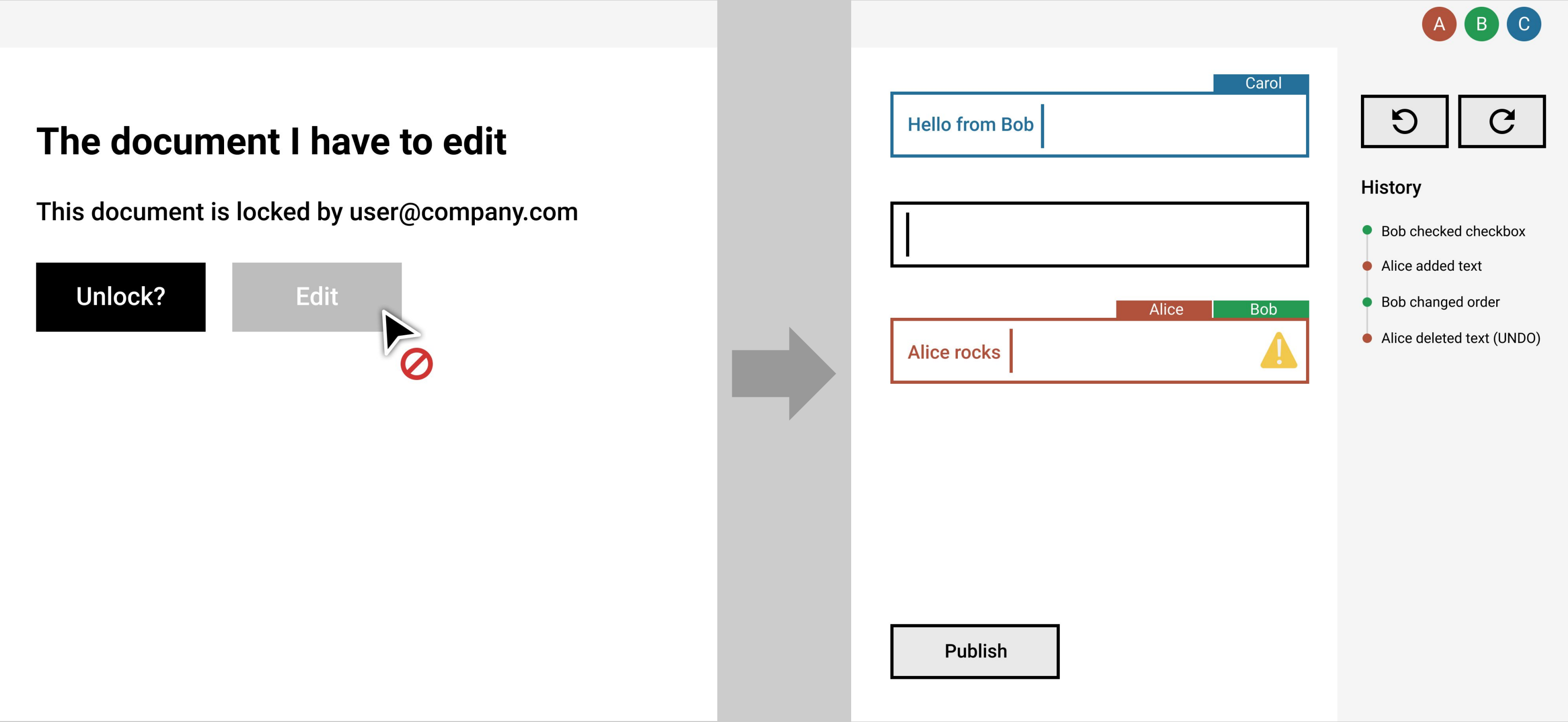
operation log

```
{action : "makeMap", obj: id1}
{action : "set", obj: id1, key : "title", value : "Buy milk"}
{action : "set", obj: id1, key : "done", value : false}
{action : "ins", obj : todosID, key : prevID, elem : 15}
{action : "link", obj : todosID, key : elem15, value : id1}
```

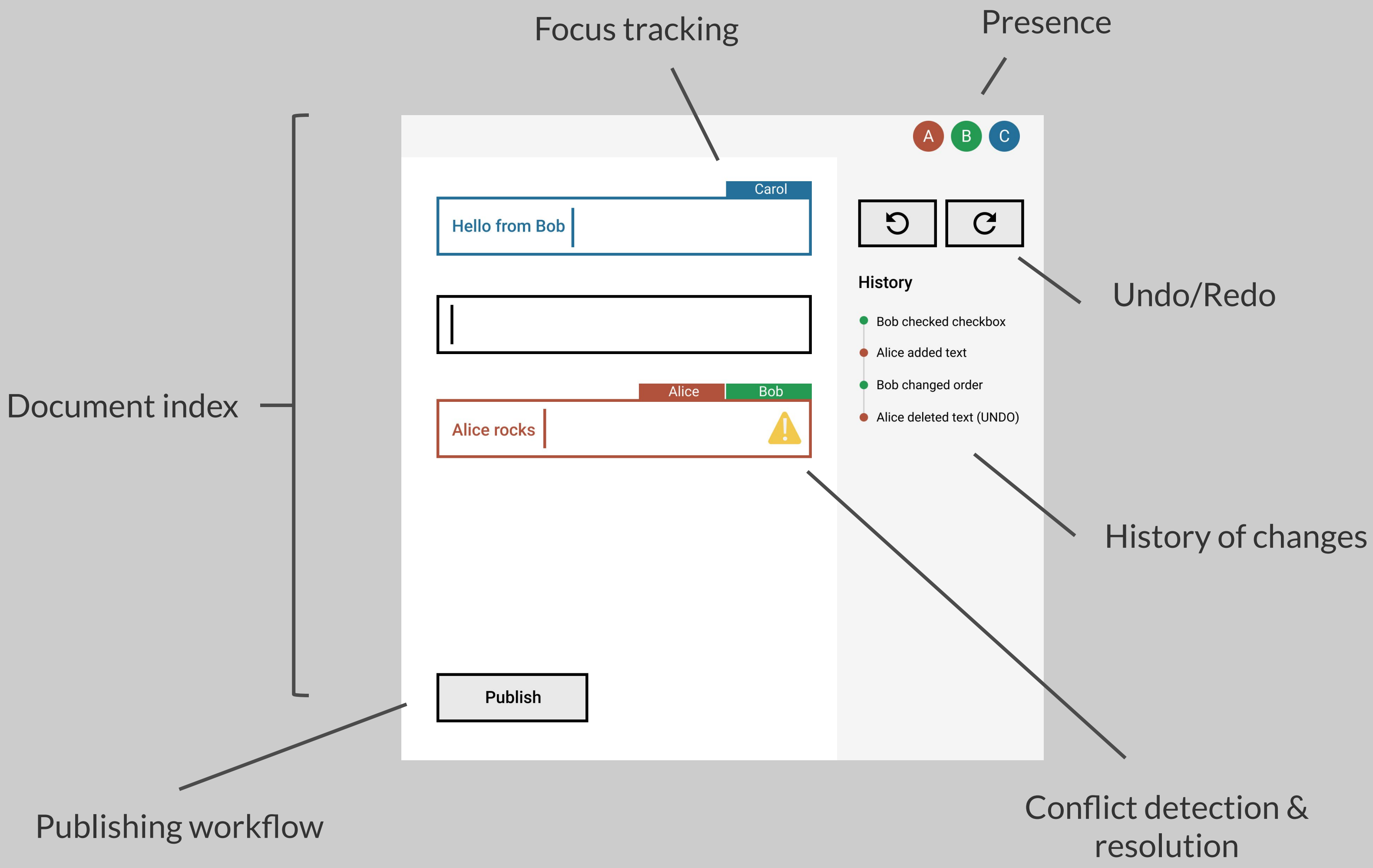
Caveats

- No communication layer → WebSockets
- Significant overhead → Garbage Collection
- No moving inside lists → Fractional Indexing
- Don’t forget

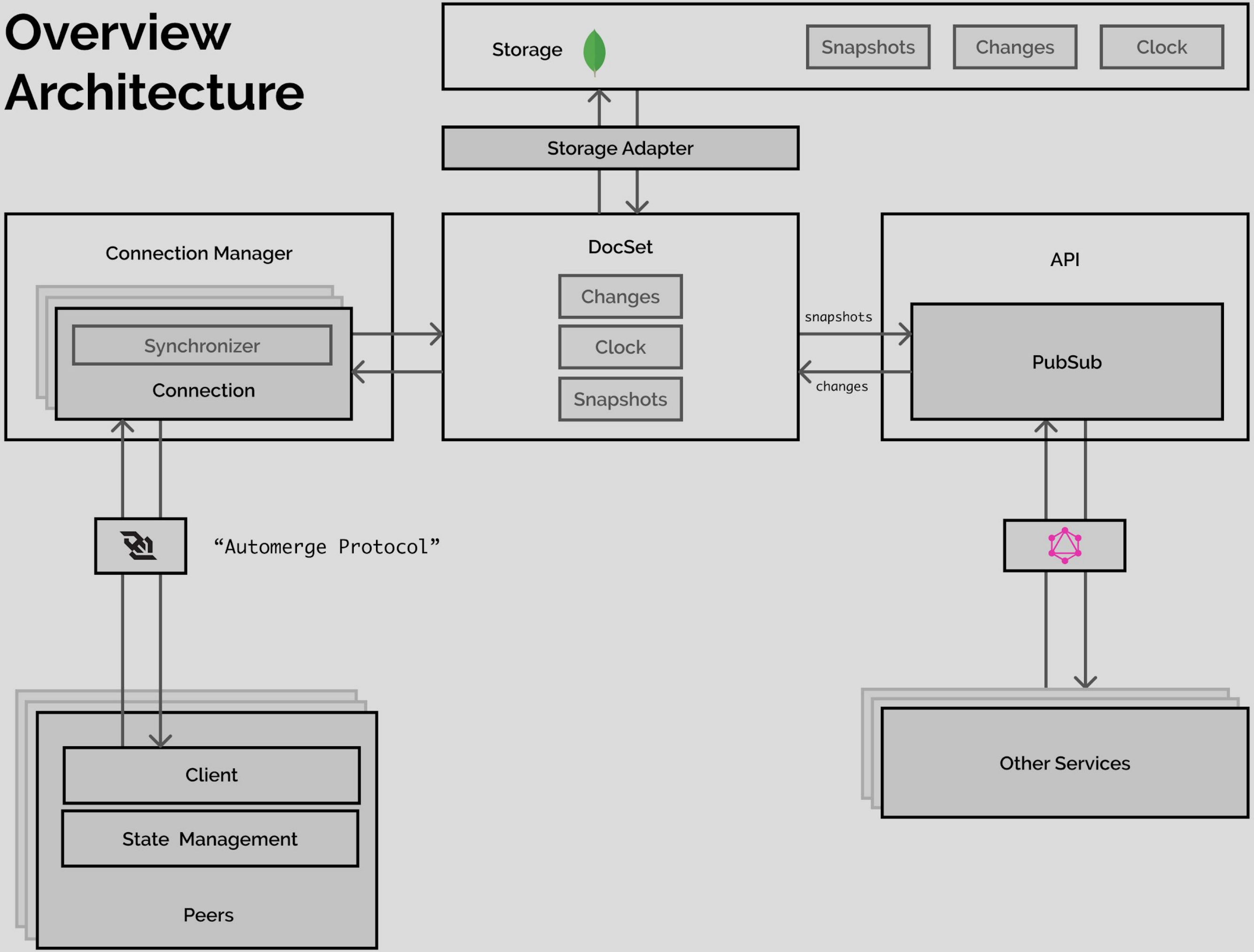
Outlook



Curation Tool 2.0


Focus tracking
Presence
A
B
C
Carol
Hello from Bob
History
Bob checked checkbox
Alice added text
Bob changed order
Alice deleted text (UNDO)
Undo/Redo
Alice
Bob
Alice rocks
Document index
History of changes
Publish
Publishing workflow
Conflict detection & resolution

Overview Architecture



Messages:	Description:
<> automerge:clock	Sync clocks
<> automerge:data	Sync data
> subscribe	Subscribe to document / presence
> subscribe:index	Subscribe to index of docs
< subscribed	Ack for subscribe
> unsubscribe	Unsubscribe from document
> focus	Focus on field
> unfocus	Unfocus field
> publish	Publish page
< merge:failed	Tried to merge after publish
< publish:failed	Tried to publish while merged

Thank you!
Q/A

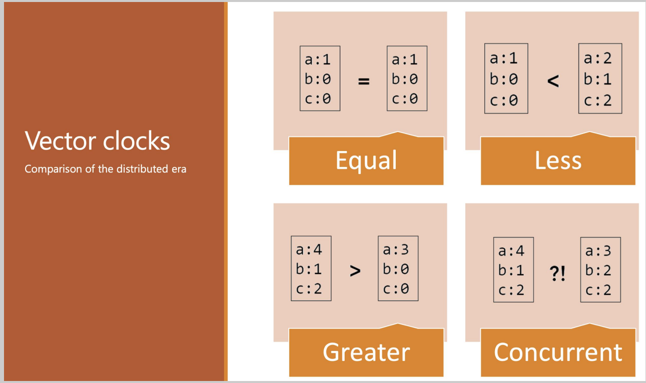
Lamport Clock

- Normal clock won't work
- Similar to G-Counter
- Gives us information which changes a client has seen

Automerge: op(timestamp, deps, cursor, mutation, value)

(c1,p1) < (c2,p2) iff
c1 < c2 or (c1 == c2 and p1 < p2)

(c1,p2) where c is counter and p is ReplicaID



heliosRX

Firebase Realtime Database + Vue + ORM = 🔥

Get started →

🗃 Firebase ORM

Object Relation Management for Firebase Realtime Database.

♻️ One codebase

Generate Frontend API and Backend API from one codebase.

⚡ Faster development

Significantly reduced development time of complex realtime applications.

heliosRX is a front-end Object-Relational Mapping layer for reactive real-time web applications using Firebase Realtime Database.

1. Define a Schema

```js
const taskModelDefinition = {
  schema: {
    fields: {
      title:     { type: 'String', required: true },
      createdAt: { type: 'ServerTimestamp' },
```