Irmin: Immutable, Fast, Distributed Storage

KC Sivaramakrishnan (kcsrk.info)

Winter School on Decentralised Trust and Blockchains 2025
19th December 2025

Irmin

- A distributed database built on the same principles as **Git**
  - Irmin is an **OCaml** library for building ***mergeable, branchable distributed data stores.***

Built-In Snapshotting

Backup and restore your data at any point in time.

Storage Agnostic

You can use Irmin on top of your own storage layer.

Custom Datatypes

Automatic (de)serialisation for custom data types.

Highly Portable

Runs anywhere from Linux to web browsers and Xen unikernels.

Git Compatibility

Bidirectional compatibility with the Git on-disk format. Irmin state can be inspected and modified using the Git command-line tool.

Dynamic Behaviour

Allows users to define custom merge functions and create event-driven workflows using a notification mechanism.

📜 Language

- Functional-first but multi-paradigm (imperative, OO)
- Static-type system with Hindley-Milner type inference
- Advanced features — powerful module system, GADTs, Polymorphic variants
- Multicore support and ***effect handlers***

⚙️ Platform

- Fast, native code— x86, ARM, RISC-V, etc.
- JavaScript and WebAssembly (using ***WasmGC***) compilation
- **Platform tools** — editor (LSP), build system (dune), package manager (opam), docs generator (odoc), etc.

🌐 Ecosystem

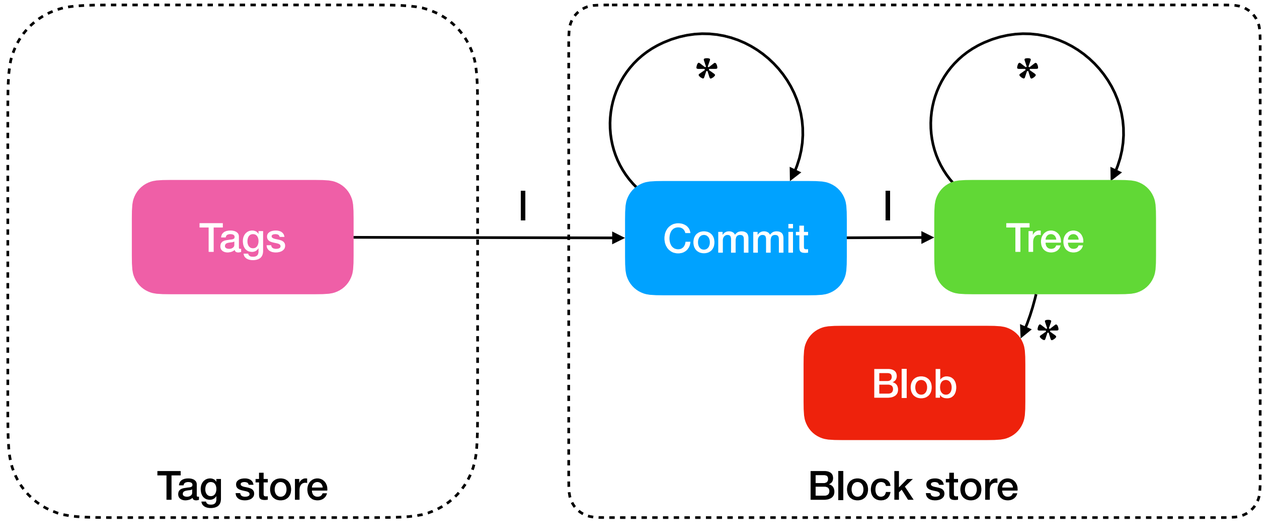
- Opam repository — small but mature package ecosystem
- Notable Industrial users — Jane Street, Meta, Microsoft, Ahrefs, Citrix, ***Tezos***, Bloomberg, Docker

Tezos

- A blockchain founded in 2018, $232M in ICO
- Innovations
  - The first proof-of-stake blockchain
  - On-chain governance
    - Protocol upgrades are built into the system
    - Soft upgrades without hard forks
- **Michelson** is the smart contract language
- **Protocol safety and formal verification** through strong typing, OCaml, and Rocq semantics
- **Irmin** is the distributed database and storage layer
  - Peers gossip blocks and state info, receive blocks, update state

- A distributed version control system
  - …. also a great model for a local-first, asynchronous, distributed system
- Do some work locally and then …
  - ***Branch*** to create temporary copies efficiently
  - ***Pull*** remote branches to get remote updates
  - ***Push*** to remotes to get your local changes to upstream
- How does this work efficiently?

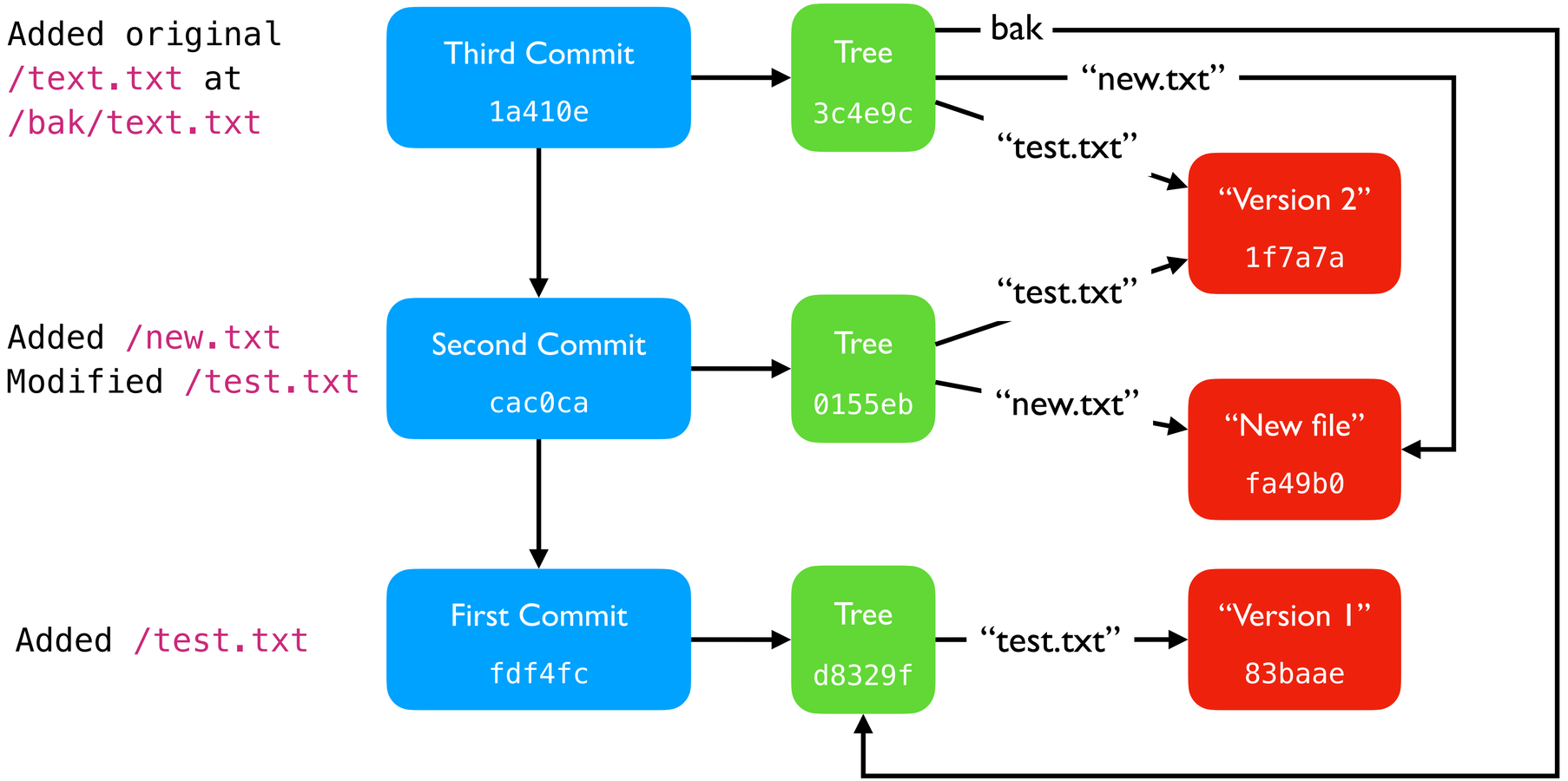
Git store data model



- Branches / tags
- Mutable

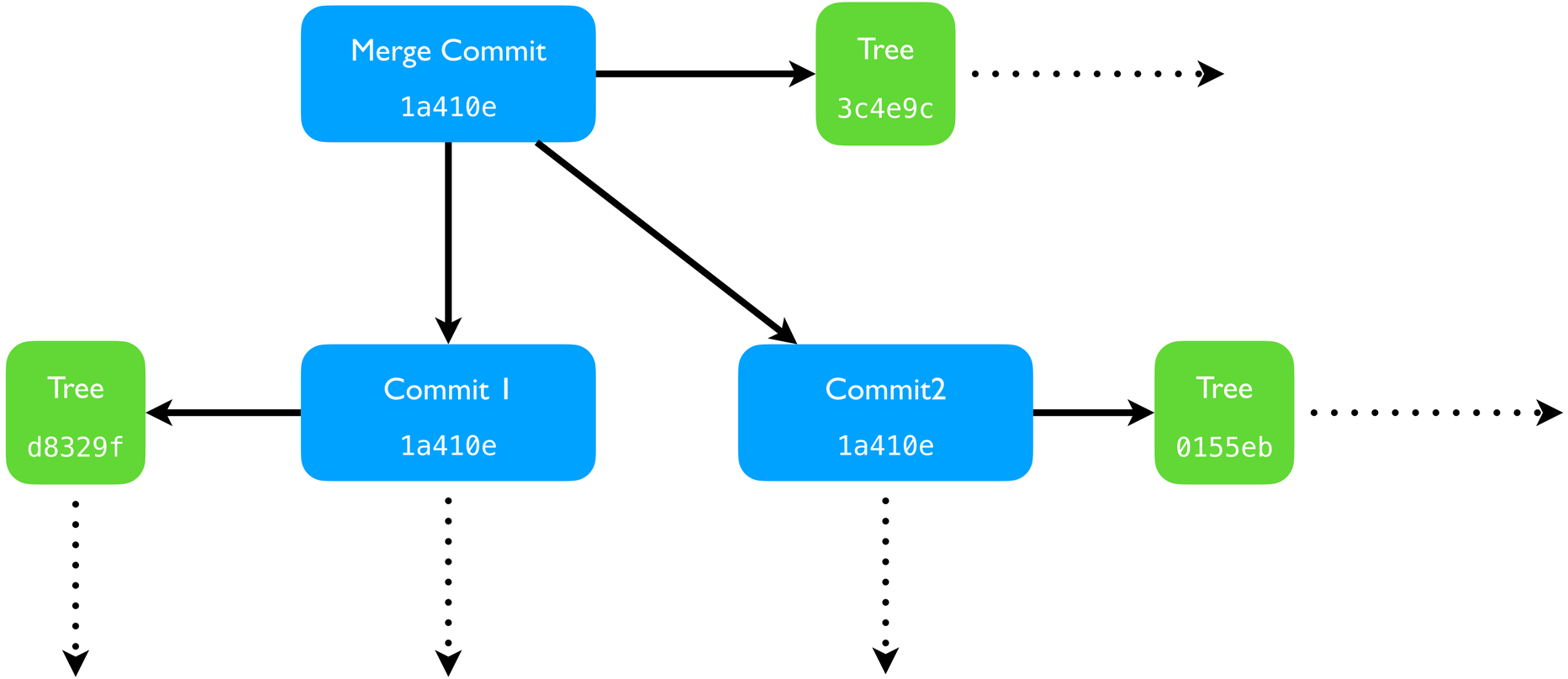
- Stores the files under version control
- Immutable, append-only & content addressed
  - hash → object

Block store — Persistence and Merkle DAG

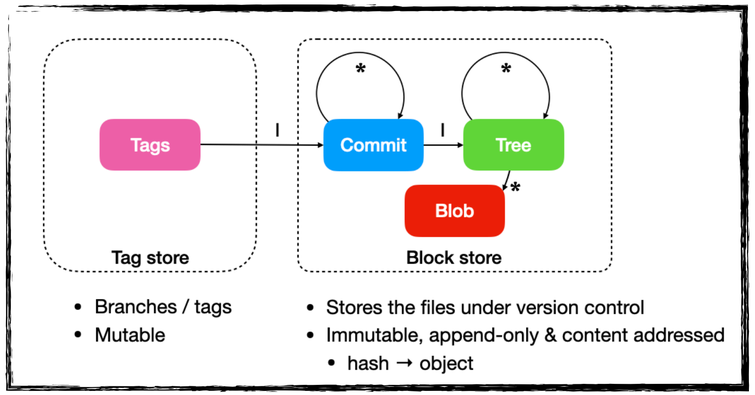


Example from Pro Git book: https://git-scm.com/book/en/v2/Git-Internals-Git-Objects

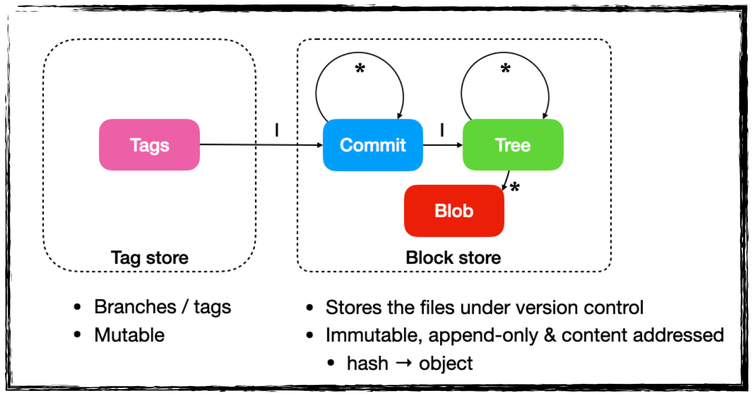
Merge commits



Excellent distributed model



Replica 1

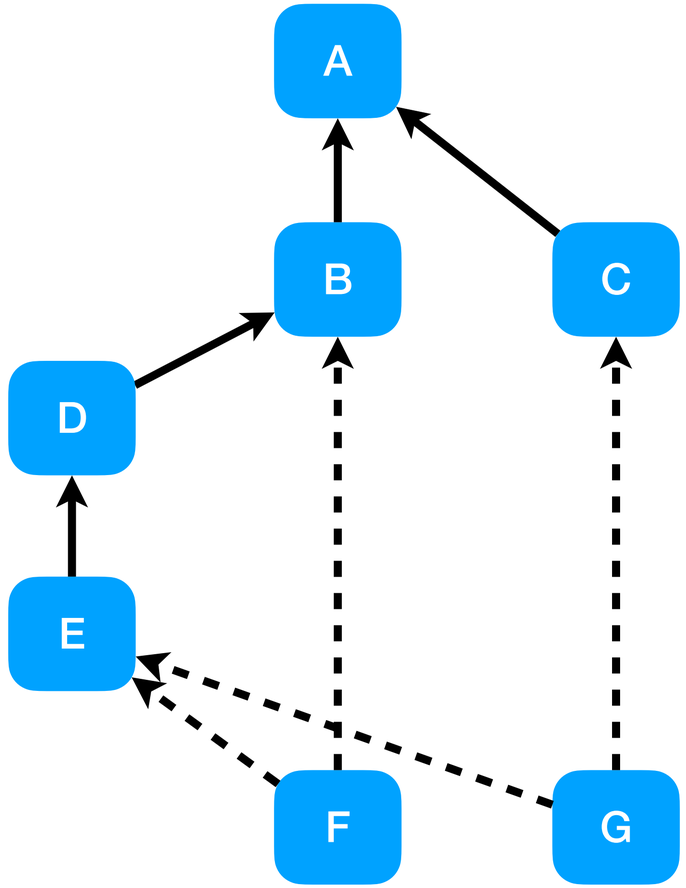


Replica 2

- Distributed communication
  - Pull changes from remote — block store has no conflicts!
  - Merge is local uses merge commit
- Strong integrity guarantees
  - Content-addressed storage (hashes), Merkle DAG of commits
  - History is immutable once referenced
  - This ensures that you can detect tampering

Commit history forms a DAG

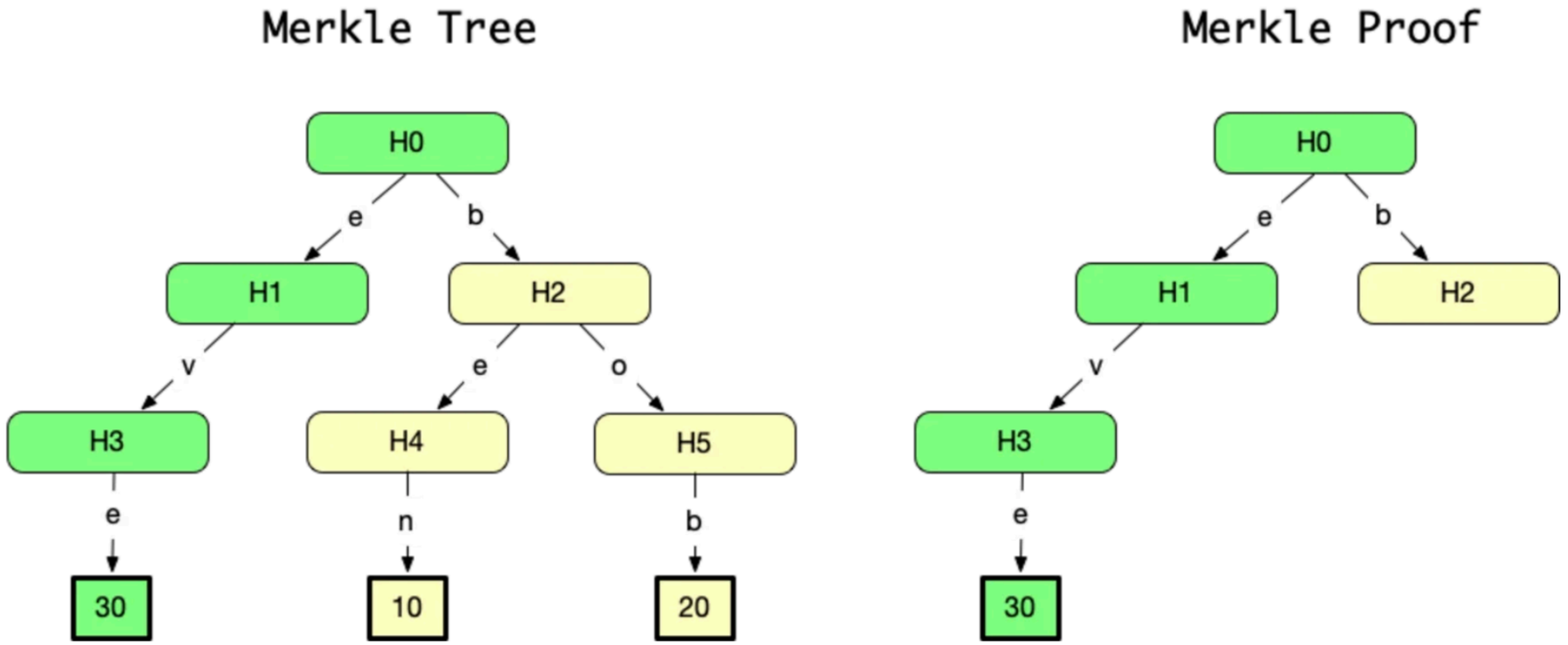
- Captures the causal history of the distribution
- ***Lowest common ancestor (LCA)*** commit always exists
  - Assuming there was an origin commit
- LCA represents the point in the history where the two commits/branches diverged
- What is the LCA of
  - E & B?
  - E & C?



Merkle Proofs

- The Tezos network builds trust between its nodes by using two components
  - **a tamper-proof database** (**Irmin**) that can generate cryptographic hashes, which uniquely and compactly represent the state of its contents; and
  - a consensus algorithm to share these cryptographic hashes across the network of (potentially adversarial) nodes.
- **Merkle proofs** let Tezos prove facts about the blockchain state without sharing the entire state.
  - Essential for scalability, decentralisation, and trust minimisation.

Merkle Proofs

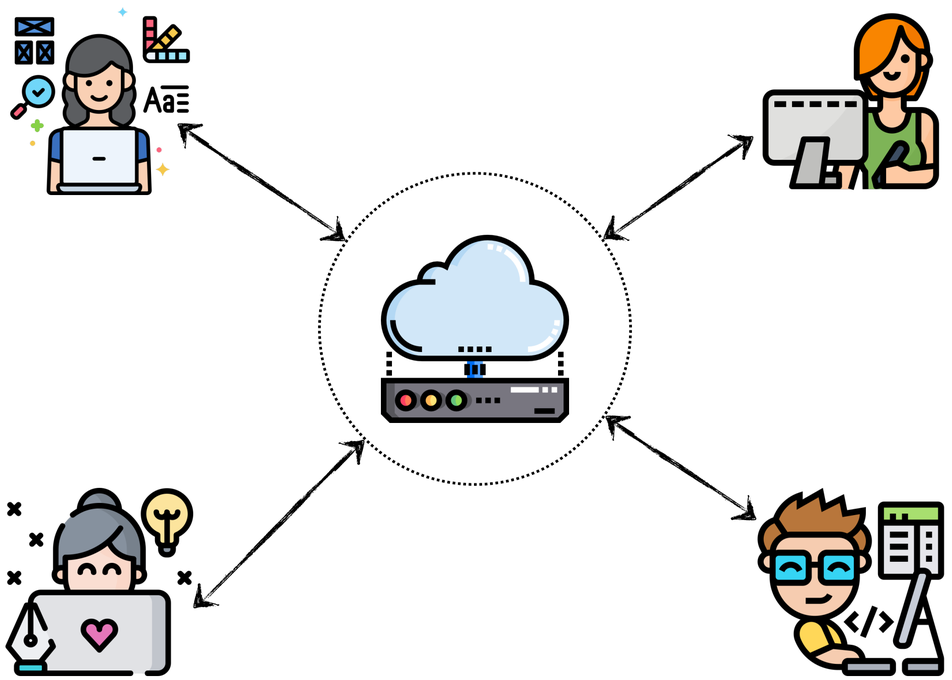


- Merkle proof is a compact representation of the Merkel tree
  - 100 ops, Merkle Proof = 46kB, Merkle Tree = 3.4 GB

Local-first software

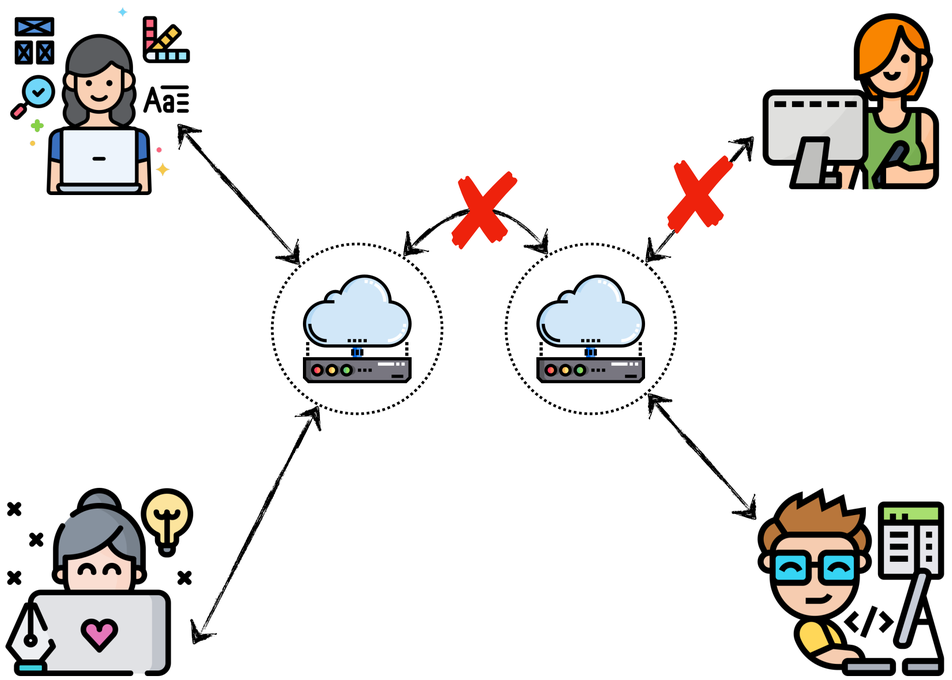
Collaborative Applications

Airtable Google Docs Figma Notion Overleaf



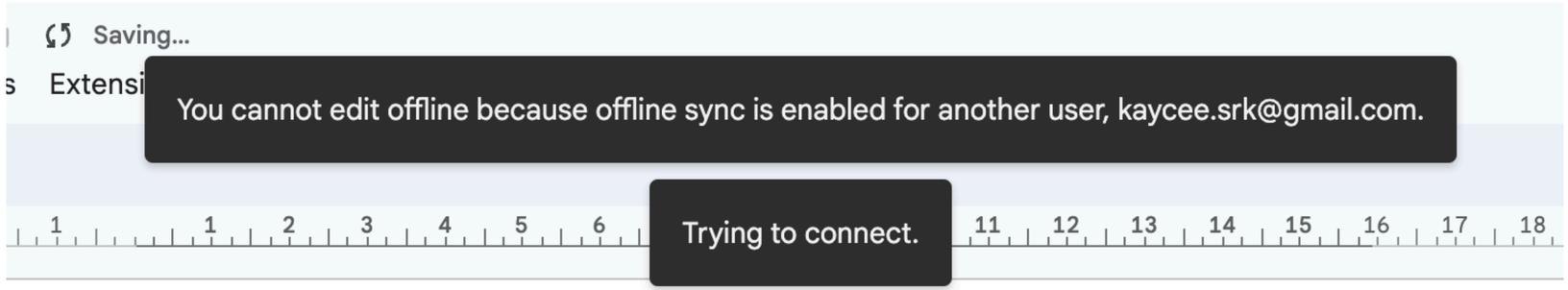
Collaborative Applications

Airtable Google Docs Figma Notion Overleaf



Network Partitions

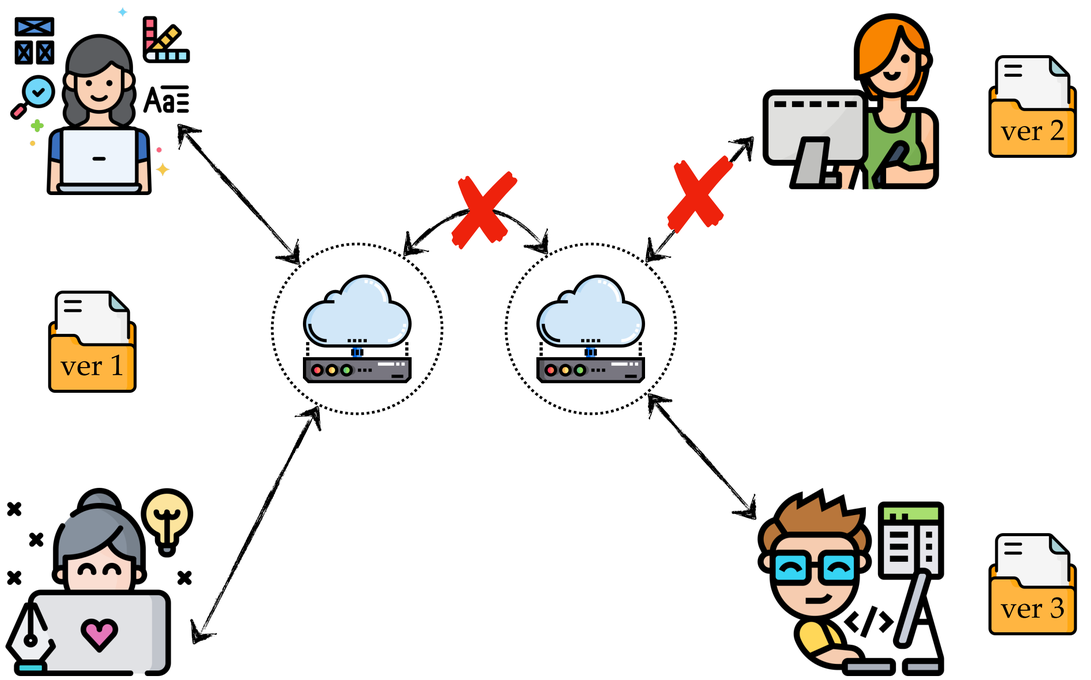
- Centralised Apps provide limited support for offline editing



Enabling offline sync for one account prevents other accounts from working offline

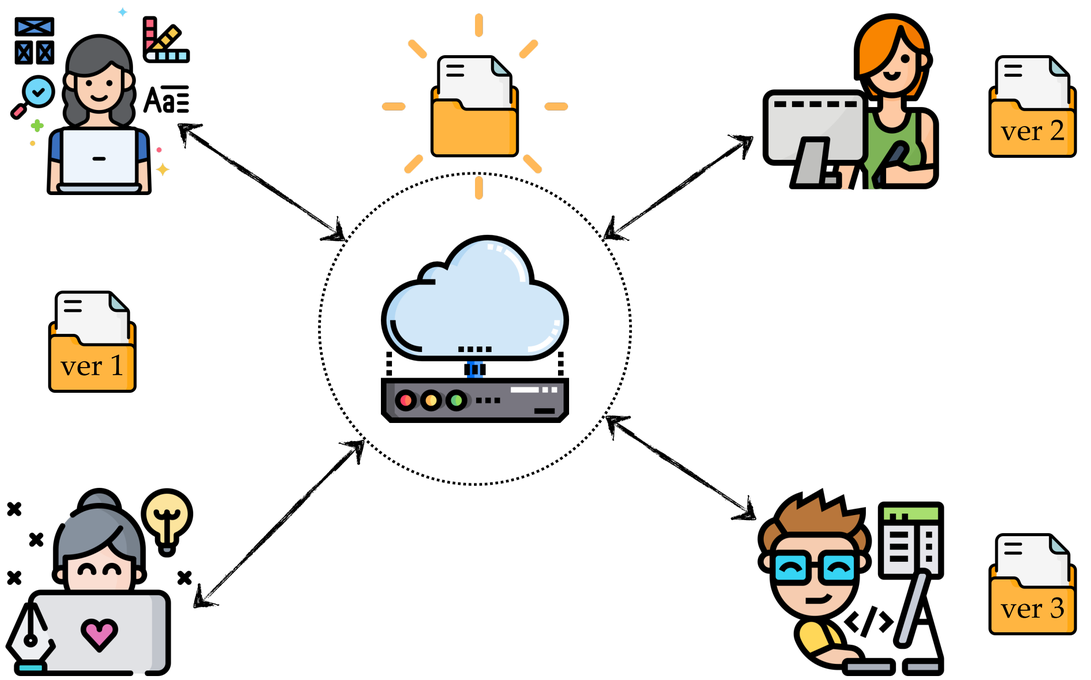
Local-first software

Airtable Google Docs Figma Notion Overleaf



Local-first software

Airtable Google Docs Figma Notion Overleaf



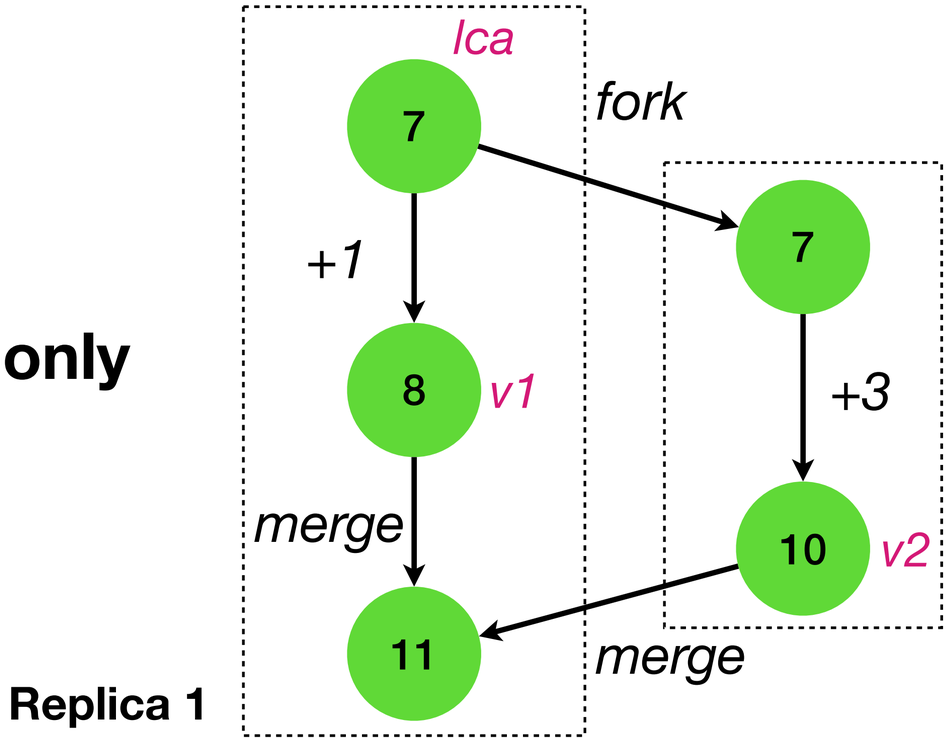
How do we build such applications?

Embed the notion of replication into the data types

Mergeable Replicated Data Types (MRDTs)

- MRDTs = Sequential data types + 3-way merge function à la Git

Increment-only counter

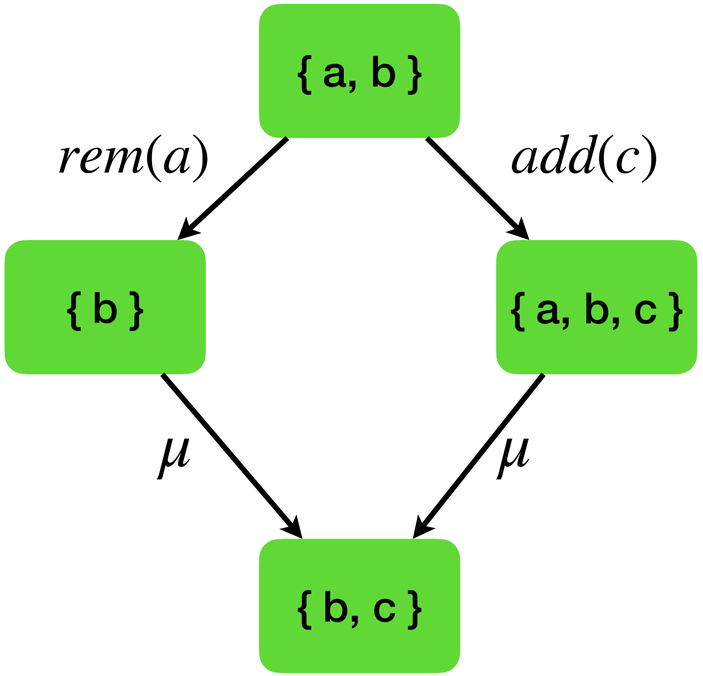


Thanks to Irmin!

```
let merge lca v1 v2 =
  lca + (v1 - lca) + (v2 - lca)
```

MRDT Set

- A replicated set with ***add***, ***remove*** and ***read*** operations

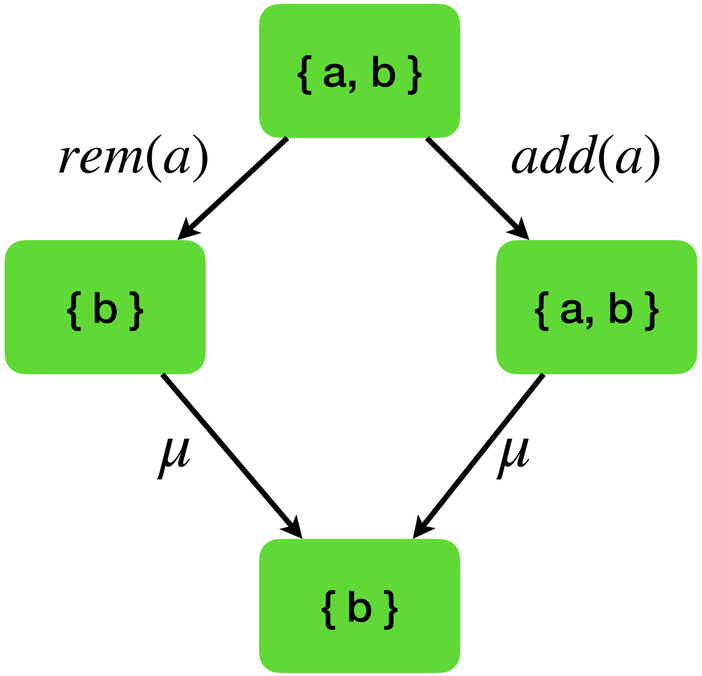



```
let merge lca v1 v2 =
  (lca ∩ v1 ∩ v2) (* unchanged or removed elements *)
  ∪ (v1 \ lca) (* added elements on left *)
  ∪ (v2 \ lca) (* added elements on right *)
```

- Do you foresee any issues with the implementation?
  - *Conflicts between concurrent add and remove of same elements!*

MRDT Set

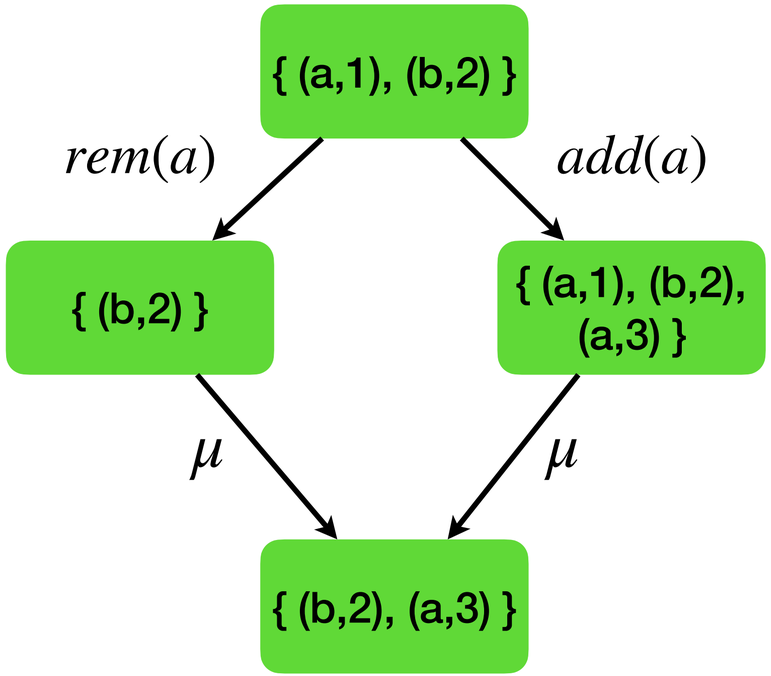
- By default, “remove wins”




```
let merge lca v1 v2 =
  (lca ∩ v1 ∩ v2) (* unchanged or removed elements *)
  ∪ (v1 \ lca) (* added elements on left *)
  ∪ (v2 \ lca) (* added elements on right *)
```

MRDT Set

- How to do “add wins”?



- Add associates a ***fresh id*** with the element
- Remove removes all matching elements with ***any id***
- Read returns the set removing ids
- Merge remains unchanged

```
let merge lca v1 v2 =
  (lca ∩ v1 ∩ v2) (* unchanged or removed elements *)
  ∪ (v1 \ lca) (* added elements on left *)
  ∪ (v2 \ lca) (* added elements on right *)
```

Why is this correct?

How do we automatically verify it?

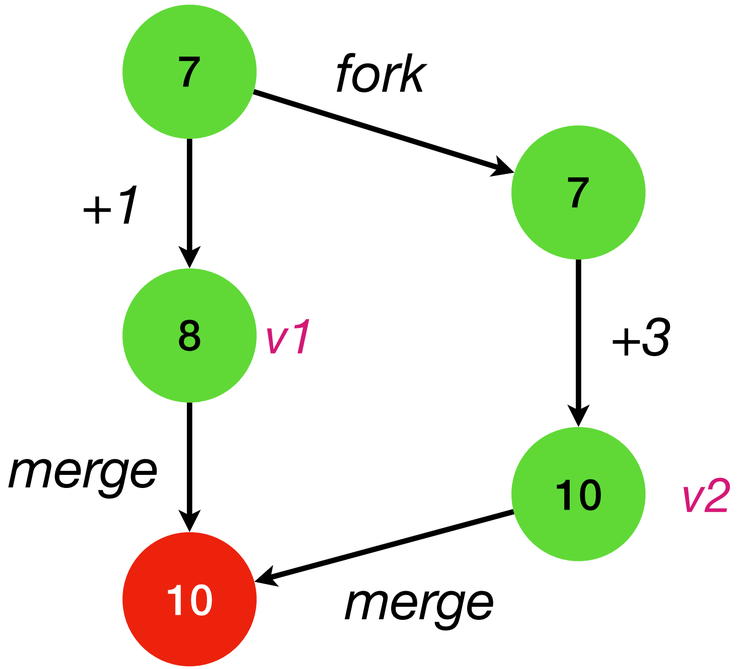
Algebraic properties are insufficient

$$\mu(a, b) = \mu(b, c) \quad \text{Commutativity}$$
$$\mu(a, a) = a \quad \text{Idempotence}$$
$$\mu(\mu(a, b), c) = \mu(a, \mu(b, c)) \quad \text{Associativity}$$

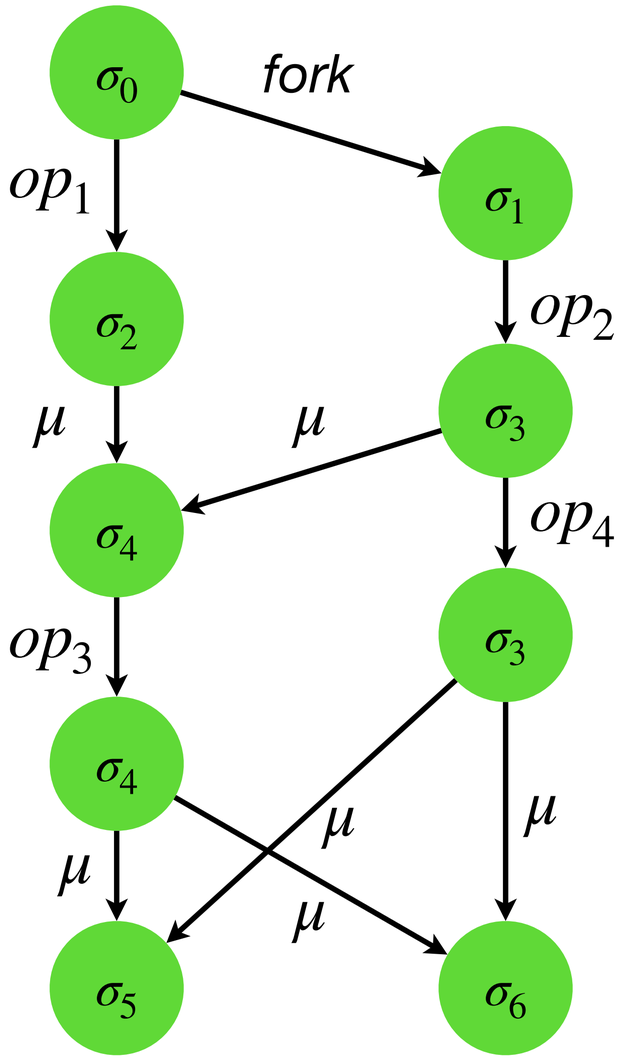
Satisfies algebraic properties

```
let merge v1 v2 = max v1 v2
```

Intent is not captured



Is there a more natural spec?



$\sigma_5 = \sigma_6 = \text{linearization}(\{op_1, op_2, op_3, op_4\})\ \sigma_0$

Replication-aware Linearizability

RESEARCH-ARTICLE

Replication-aware linearizability

Authors: Chao Wang, Constantin Enea, Suha Orhun Mutluergil, Gustavo Petri | Authors Info & Clai

PLDI 2019: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation
https://doi.org/10.1145/3314221.3314617

- Replica states should be a ***linearisation*** of observed ***update*** operations
  - Linearisation total order lo compatible with partially-ordered visibility relation vis
  - No real-time ordering requirement unlike traditional linearizability
- Payoff
  - If a replicated object is RA-linearizable, reason about it using sequential semantics

Using RA-linearizability for verification

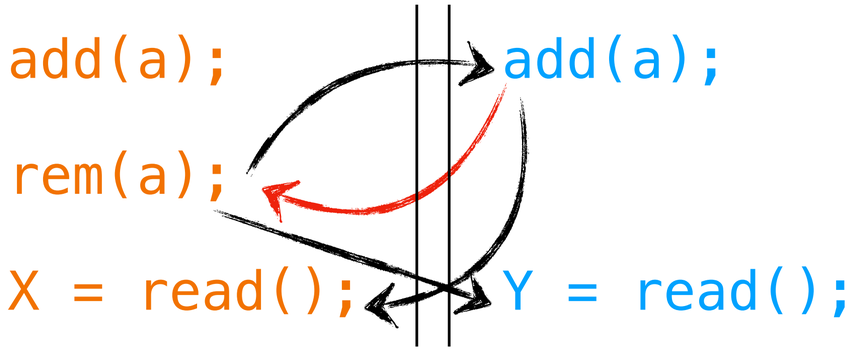
```
add(a);   ||  add(a);
rem(a);   ||
X = read(); || Y = read();
```

$$a \in X \implies a \in Y$$

- Since Add-wins set is RA-linearizable, you can use *totally ordered trace* and the *sequential spec* to reason about correctness

```
add(a);     rem(a);     add(a);     X = read();     Y = read()
{a}         {}          {a}         X = {a}         Y = {a}
```

Using RA-linearizability for verification



$$a \in X \implies a \in Y$$

- Let's try to make the statement false
  - Make $a \in X$ true and $a \in Y$ false

Replication-aware Linearizability

RESEARCH-ARTICLE

Replication-aware linearizability

Authors: Chao Wang, Constantin Enea, Suha Orhun Mutluergil, Gustavo Petri | Authors Info & Clai

PLDI 2019: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation
https://doi.org/10.1145/3314221.3314617

- Presented a proof methodology to show that an RDT is linearisable
- Not automated or mechanised

Neem — Automatic verification of RDTs

- **What's in the box?**
  - Definition of RA-linearizability for MRDTs
  - A novel induction scheme for MRDTs and state-based CRDTs to ***automatically*** verify RA-linearizability
  - Implemented in F*

RESEARCH-ARTICLE | OPEN ACCESS |

Automatically Verifying Replication-Aware Linearizability

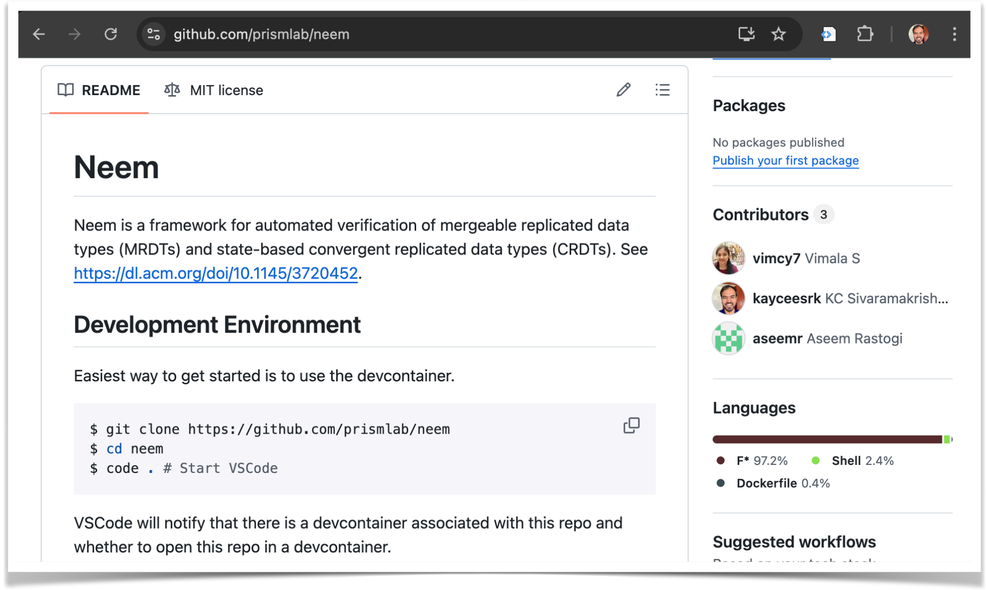
Authors: Vimala Soundarapandian, Kartik Nagar, Aseem Rastogi, KC Sivaramakrishnan | Authors Info & Claims

Proceedings of the ACM on Programming Languages, Volume 9, Issue OOPSLA1 • Article No.: 111, Pages 871 - 897
https://doi.org/10.1145/3720452

Published: 09 April 2025 Publication History | Check for updates

Related Artifact: Automatically Verifying Replication-aware Linearizability - artifact • April 2025 • software •
https://doi.org/10.5281/zenodo.14591614

0 77 | PDF | eReader



github.com/prismlab/neem

README | MIT license

Neem

Neem is a framework for automated verification of mergeable replicated data types (MRDTs) and state-based convergent replicated data types (CRDTs). See https://dl.acm.org/doi/10.1145/3720452.

Development Environment

Easiest way to get started is to use the devcontainer.

```
$ git clone https://github.com/prismlab/neem
$ cd neem
$ code . # Start VSCode
```

VSCode will notify that there is a devcontainer associated with this repo and whether to open this repo in a devcontainer.

Packages

No packages published
Publish your first package

Contributors 3

vimcy7 Vimala S
kayceesrk KC Sivaramakrish...
aseemr Aseem Rastogi

Languages

F* 97.2% Shell 2.4%
Dockerfile 0.4%

Suggested workflows

Verified MRDTs

MRDT	rc Policy	#LOC	Verification Time (s)
Increment-only counter [12]	none	6	0.72
PN counter [23]	none	10	1.64
Enable-wins flag*	disable $\xrightarrow{rc}$ enable	30	29.80
Disable-wins flag*	enable $\xrightarrow{rc}$ disable	30	37.91
Grows-only set [12]	none	6	0.45
Grows-only map [23]	none	11	4.65
OR-set [23]	$rem_a \xrightarrow{rc} add_a$	20	4.53
OR-set (efficient)*	$rem_a \xrightarrow{rc} add_a$	34	660.00
Remove-wins set*	$add_a \xrightarrow{rc} rem_a$	22	9.60
Set-wins map*	$del_k \xrightarrow{rc} set_k$	20	5.06
Replicated Growable Array [1]	none	13	1.51
Optional register*	unset $\xrightarrow{rc}$ set	35	200.00
Multi-valued Register*	none	7	0.65
JSON-style MRDT*	Fig. 13	26	148.84

Neem also supports verification of RA-linearizability of state-based CRDTs

https://github.com/prismlab/neem

Conclusion

- **Irmin** is an excellent foundation for efficient and principled distributed applications
  - Rich data model using MRDTs
  - Ability to verify the correctness of the application layer automatically through Neem
  - Can be compiled to various storage-efficient backends transparently
- Questions
  - Can we use replication-aware linearizability to **reconcile the DAG view to linear blockchain**?
    - Hedera Hashgraph and other DAGs are interesting alternatives to blockchain
  - Could Irmin act as a **state substrate** for L2s, rollups, or light clients?
  - Can we reason about **Byzantine behaviour** when merges are application-defined?
    - Should RA-linearizability be a prerequisite for user-defined merges?