2024

TEE + Raft → Confidential Replicated State Machines

Dzmitry Huba

Introduction

Confidential services landscape

Scenarios

Lock service
Policy bookkeeping and enforcement

Key value store
Transactional data store for private data

Key provisioning service
Key management and distribution for trusted workloads

Private data processing
Analytics or learning over private data

Properties

Stateful
Uses private state throughout the lifetime

Long-running
Lifetime measures in days

Strongly consistent
Necessary to guarantee privacy

Multiple releases
Partial results are released multiple times throughout the lifetime

Challenges

Scalability
Ranges from small to massive scale

Failure resilience
Failures on the days long timeframe are likely to happen

Protection against attacks
Attack surface grows with complexity

Understandability
Critical for review process and overall narrative

Privacy mechanisms options and desirable properties

Specialized hardware

Privacy is guaranteed by a specialized hardware (**may not be feasible** for widespread adoption due unacceptable price or provisioning, **may not be attestable**, **reproducible only if publicly available**)

Reproducible
Mechanism must be verifiable by a third party for review and adoption reasons

Scalable
Mechanism must have a reasonable scalability story beyond toy examples

Feasible
Hardware capacity, failure and acquisition rate must be acceptable, price is reasonable

Attestable
Mechanism must minimize behavior that affects privacy and is not externally verifiable

Commodity hardware + distributed systems

Privacy is guaranteed by a distributed algorithm (**reproducible**) executed on non-colluding or non-compromised majority running on off-the-shelf TEE hardware (**feasible, attestable**)

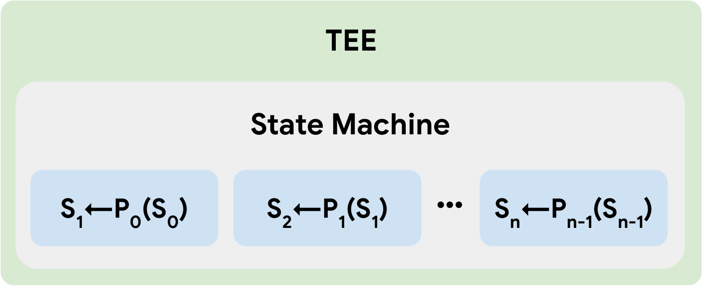
Replicated state machines is a general method for implementing a fault-tolerant service

Critical step is choosing an order for the inputs to be processed, can be done with consensus protocol (Raft!)

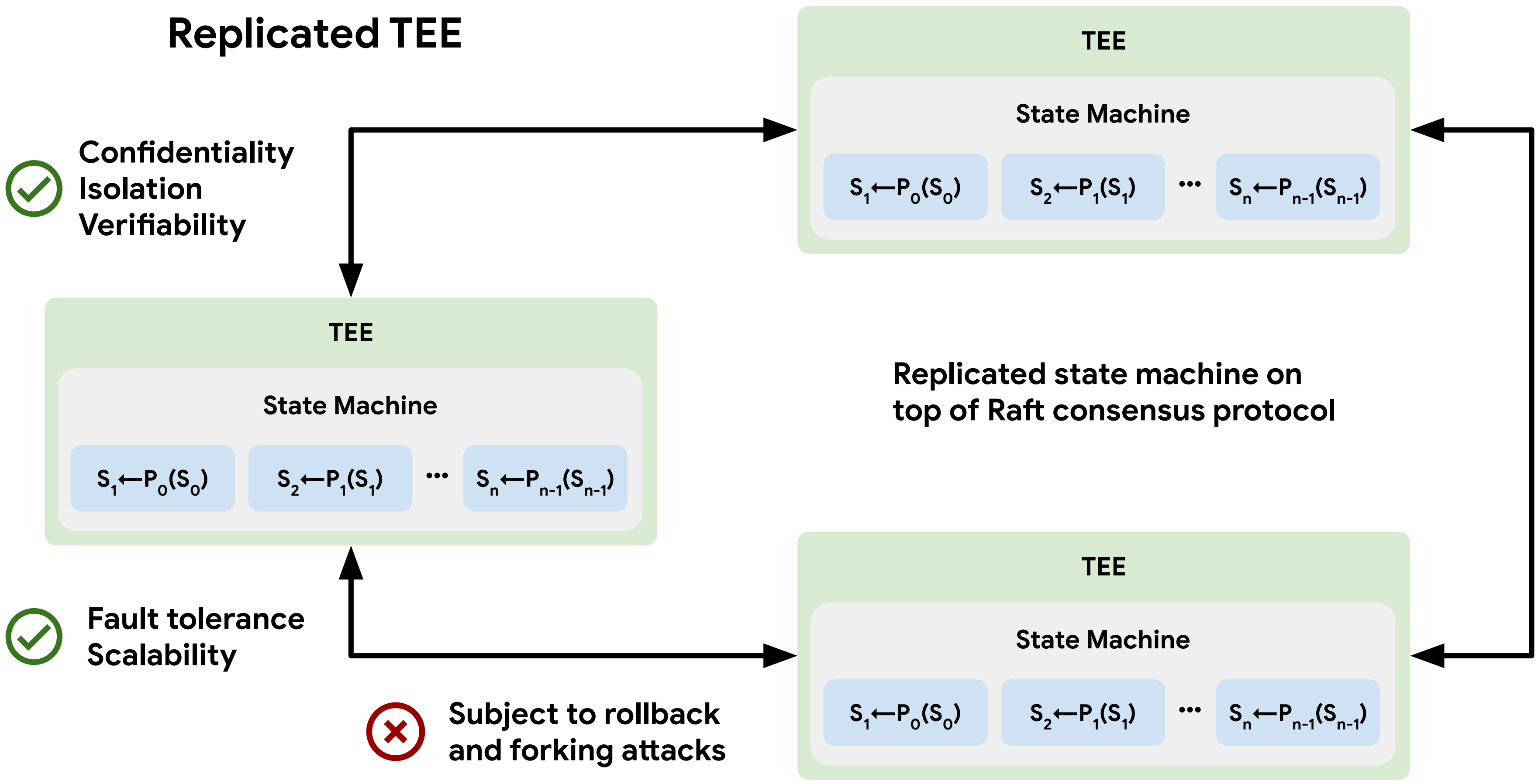
TEE + Raft enables replicated state machines with rollback protection and externally verifiable behavior

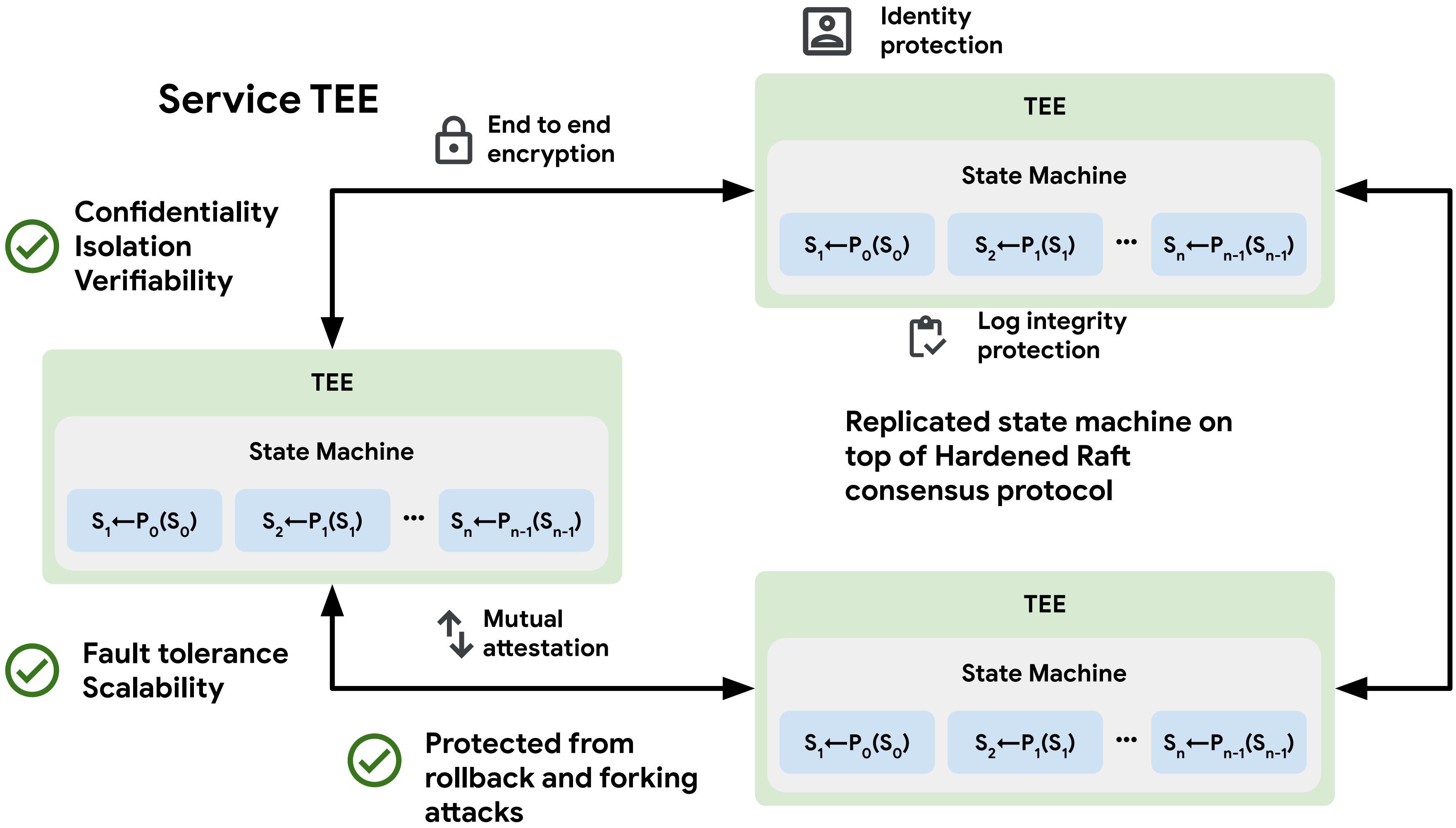
Physical TEE

Confidentiality
Isolation
Verifiability



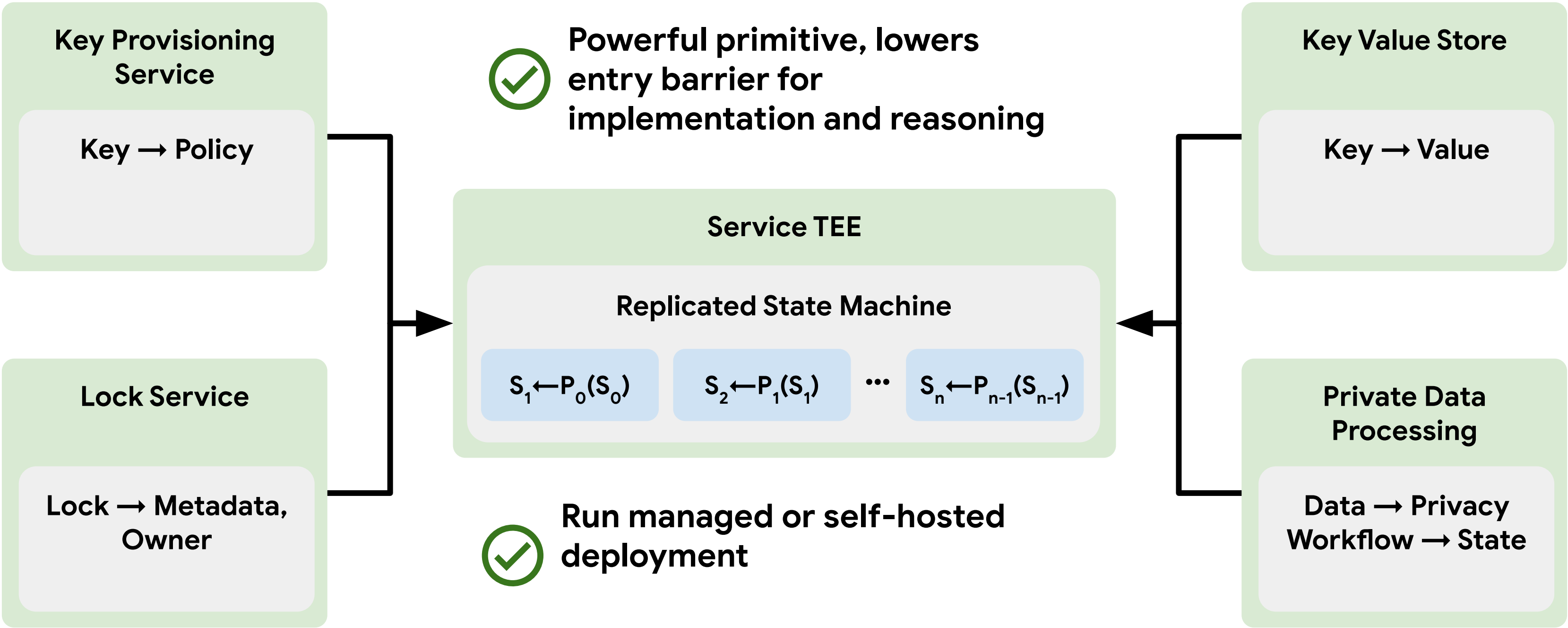
Fault tolerance
Scalability


Replicated TEE
Confidentiality
Isolation
Verifiability
TEE
State Machine
$S_1 \leftarrow P_0(S_0)$
$S_2 \leftarrow P_1(S_1)$
...
$S_n \leftarrow P_{n-1}(S_{n-1})$
TEE
State Machine
$S_1 \leftarrow P_0(S_0)$
$S_2 \leftarrow P_1(S_1)$
...
$S_n \leftarrow P_{n-1}(S_{n-1})$
Replicated state machine on
top of Raft consensus protocol
Fault tolerance
Scalability
Subject to rollback
and forking attacks
TEE
State Machine
$S_1 \leftarrow P_0(S_0)$
$S_2 \leftarrow P_1(S_1)$
...
$S_n \leftarrow P_{n-1}(S_{n-1})$


Service TEE
Confidentiality
Isolation
Verifiability
End to end
encryption
Identity
protection
TEE
State Machine
$S_1 \leftarrow P_0(S_0)$
$S_2 \leftarrow P_1(S_1)$
...
$S_n \leftarrow P_{n-1}(S_{n-1})$
Log integrity
protection
Replicated state machine on top of Hardened Raft consensus protocol
TEE
State Machine
$S_1 \leftarrow P_0(S_0)$
$S_2 \leftarrow P_1(S_1)$
...
$S_n \leftarrow P_{n-1}(S_{n-1})$
Mutual
attestation
Fault tolerance
Scalability
TEE
State Machine
$S_1 \leftarrow P_0(S_0)$
$S_2 \leftarrow P_1(S_1)$
...
$S_n \leftarrow P_{n-1}(S_{n-1})$
Protected from rollback and forking attacks

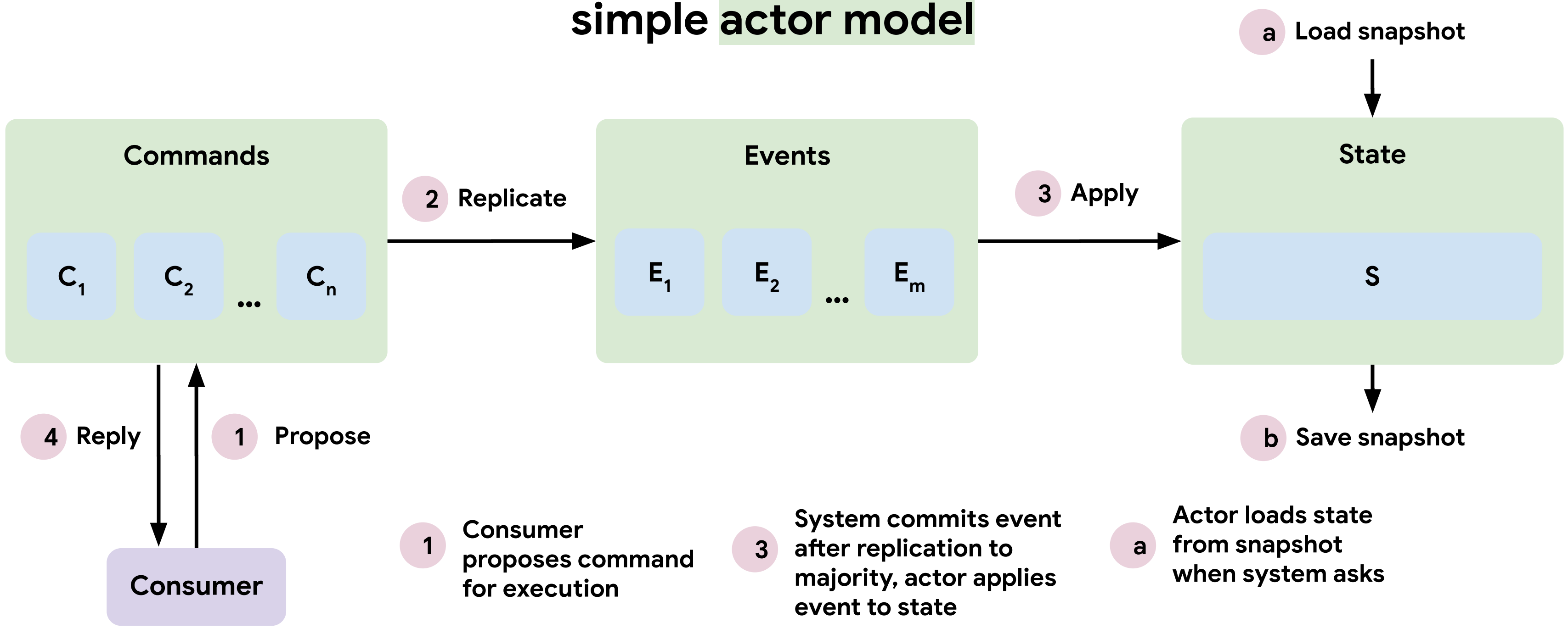
Lift TEE guarantees
from process to service level

Building confidential services



Programming model

Confidential computations are represented through a simple actor model



1 Consumer proposes command for execution

2 Actor generates event, system replicates event into event log

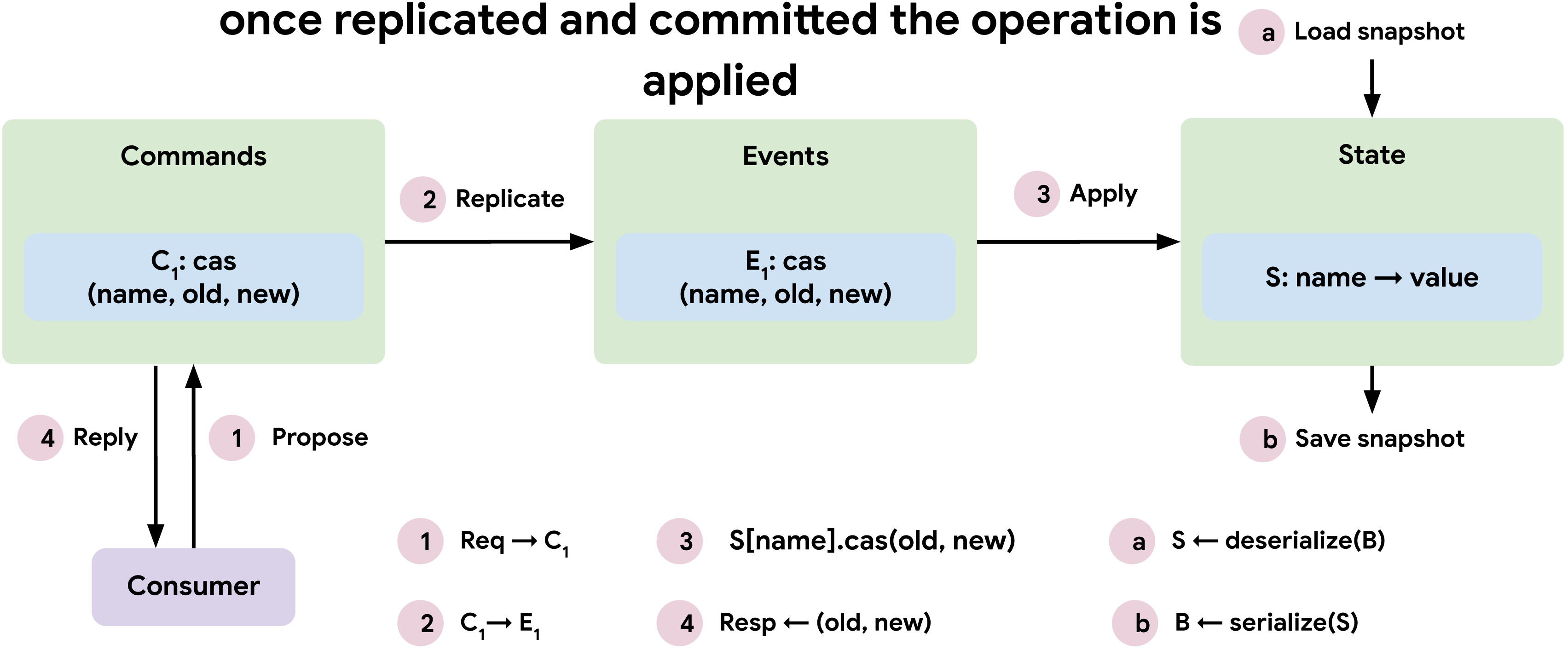
3 System commits event after replication to majority, actor applies event to state

4 Actor replies to consumer

a Actor loads state from snapshot when system asks

b Actor saves state to snapshot when system asks

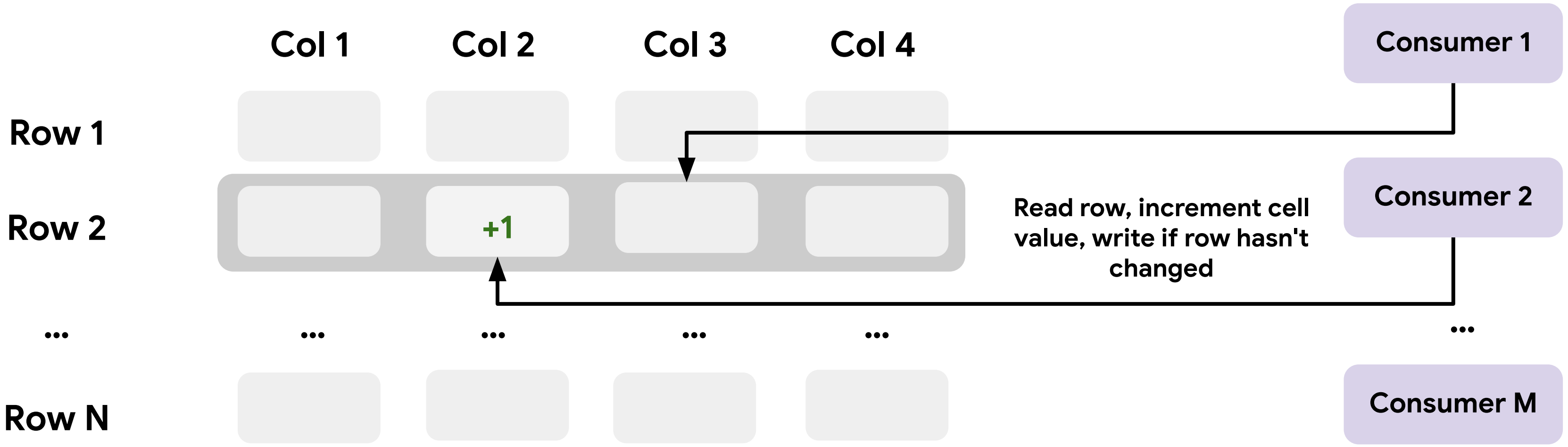
Consumers propose compare and swap operation, once replicated and committed the operation is applied



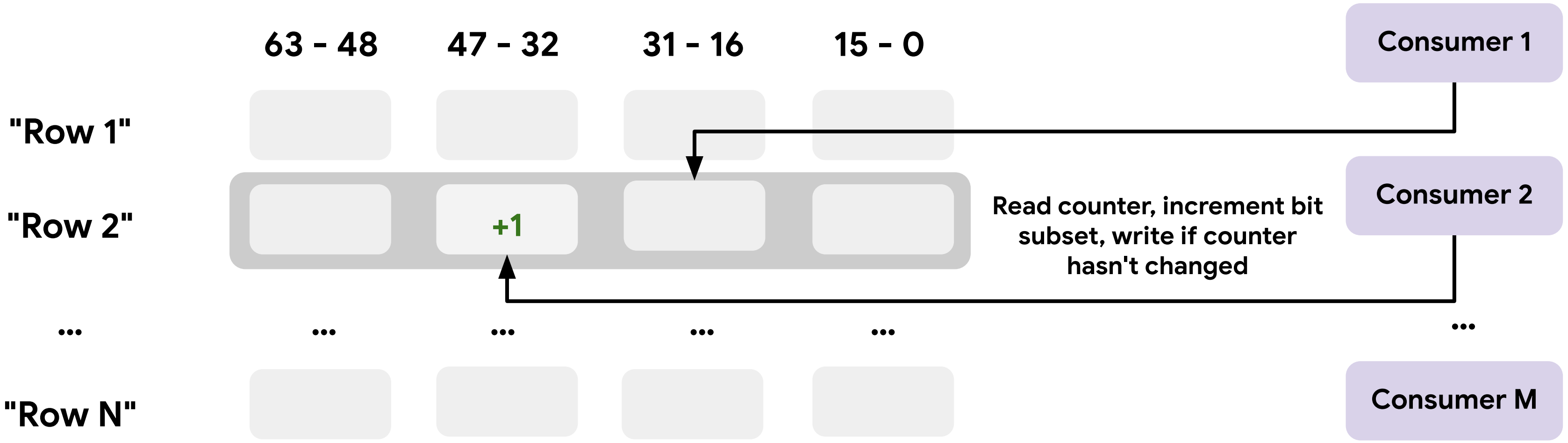
Actor implementation

Demo

Consumers update assigned table cell with optimistic concurrency



Consumers update 16 bits of a named 64 bits counter with a compare and swap operation



• • •

Raft

Best source is
the paper and the visualization

Raft in a nutshell

Focus

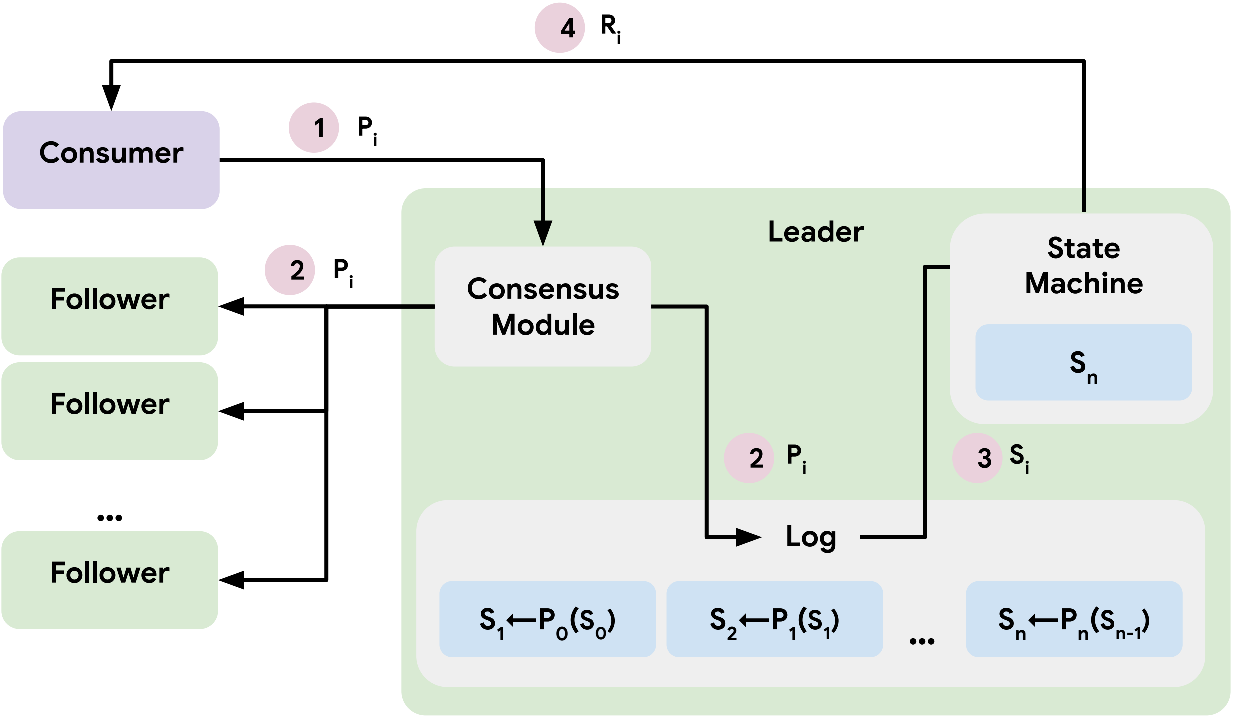
Makes design decisions based on **understandability**. Uses **problem decomposition** and **state space minimization** (handle multiple problems with single mechanism, eliminate special cases, minimize non-determinism, etc.).

Decomposition

Consists of **leader election** (select one server to act as leader, detect crashes, choose new leader), **log replication** (leader accepts commands from clients, appends to its log, replicates its log to other servers and overwrites inconsistencies), **safety** (keep logs consistent, only servers with up-to-date logs can become leader).

Slide is borrowed from author's presentation, restyled

Replicated state machine



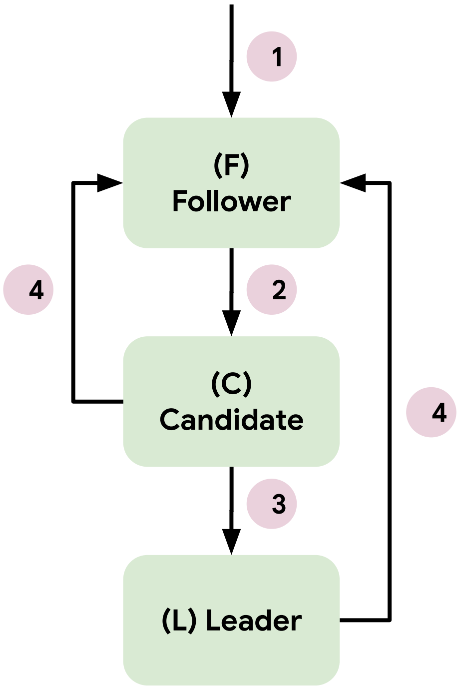
1. Consumer issues proposals to the leader Raft node
2. Consensus module appends proposal entry to the log and replicates log to the follower Raft nodes
3. The committed log entry (replicated to the majority) is applied to the state machine
4. The result is sent to the client

Node states

F Follower is passive, expects regular heartbeats

C Candidate requests others to vote to get elected

L Leader sends heartbeats and replicates entries



1 Every node starts as a follower

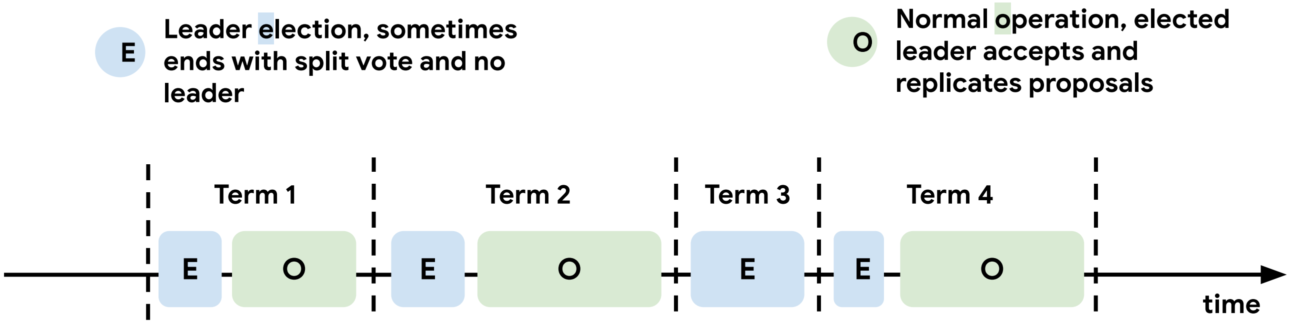
2 Follower switches to candidate role with no heartbeats from leader

3 Candidate switches to leader role with majority of votes secured

4 Leader and candidate switch to follower after discovering higher term

Slide is borrowed from [author's presentation](), restyled

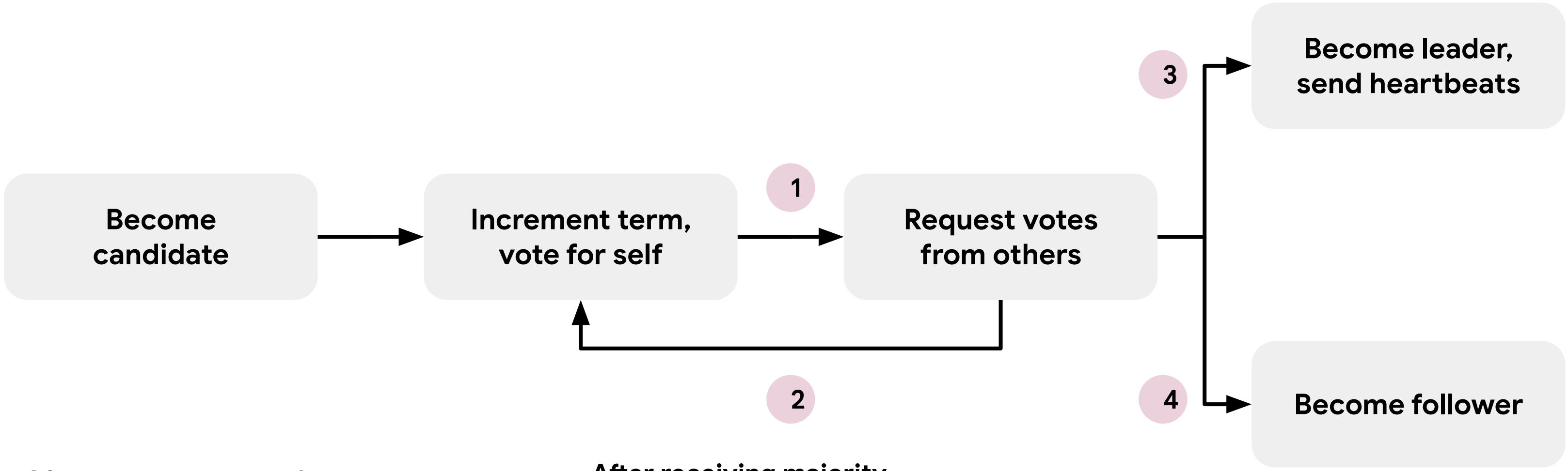
Terms



1. At most one leader per term, some terms have no leader due to failed election
2. Each node maintains current term value, exchanged with peers, used to identify obsolete information

Slide is borrowed from [author's presentation](), restyled

Leader election



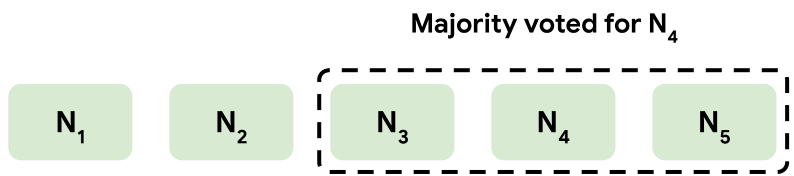
1. After current term and vote for self persisted
2. After timeout if cannot receive majority votes
3. After receiving majority votes
4. After receiving rpc from leader or with higher term

Slide is borrowed from [author's presentation](), restyled

Election correctness

S **Safety**: allow at most one winner per term

L **Liveness**: some candidate must eventually win



1. Each node gives only one vote per term (persist on disk), majority required to win election

2. Choose election timeouts randomly in [T, 2T] (e.g. 150-300 ms), one node usually times out and wins election before others time out, works well if T >> broadcast time

Slide is borrowed from [author's presentation](), restyled

Log structure

	1	2	3	4	5	6	7	8	9	10
(L, 3) N_1	1 | A	1 | B	1 | C	2 | D	2 | E	3 | F	3 | G	3 | H	3 | I	3 | J
(F) N_2	1 | A	1 | B	1 | C	2 | D	2 | E	3 | F	3 | G			
(F) N_3	1 | A	1 | B	1 | C	2 | D	2 | E	3 | F	3 | G	3 | H	3 | I	3 | J
(F) N_4	1 | A	1 | B								
(F) N_5	1 | A	1 | B	1 | C	2 | D	2 | E	3 | F	3 | G	3 | H		

Entry committed if safe to execute in state machines: replicated to majority by the leader of the current term

Index

(Leader, Term) Node

Slide is borrowed from author's presentation, restyled

Term | Proposal

Log inconsistencies

	1	2	3	4	5	6	7	8	9	10
(L, 4) N_1	1 | A	1 | B	1 | C	2 | D	2 | E	3 | F	3 | G	3 | H		
(F) N_2	1 | A	1 | B	1 | C	2 | D	2 | E	3 | F	3 | G			
(F) N_3	1 | A	1 | B	1 | C	2 | D	2 | E	3 | F	3 | G	3 | H	3 | I	
(F) N_4	1 | A	1 | B								
(F) N_5	1 | A	1 | B	1 | C	2 | D	2 | E	2 | P	2 | Q	2 | R	2 | T	

Raft minimizes special code for repairing inconsistencies: leader assumes its log is correct, Normal operation will repair all inconsistencies

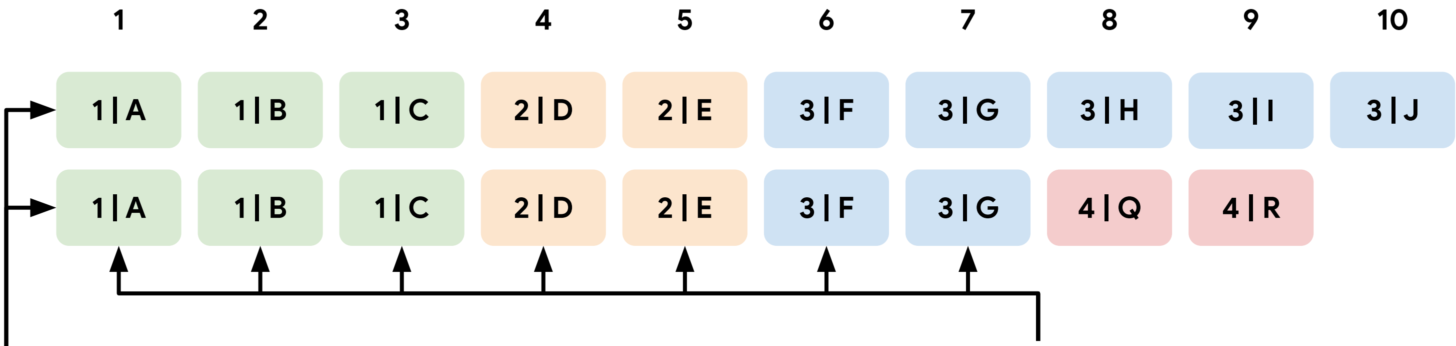
(Leader, Term) Node

Slide is borrowed from [author's presentation](), restyled

Index

Term | Proposal

Log matching property



1 If log entries on different nodes have same index and term: they store the same command and the logs are identical in all preceding entries

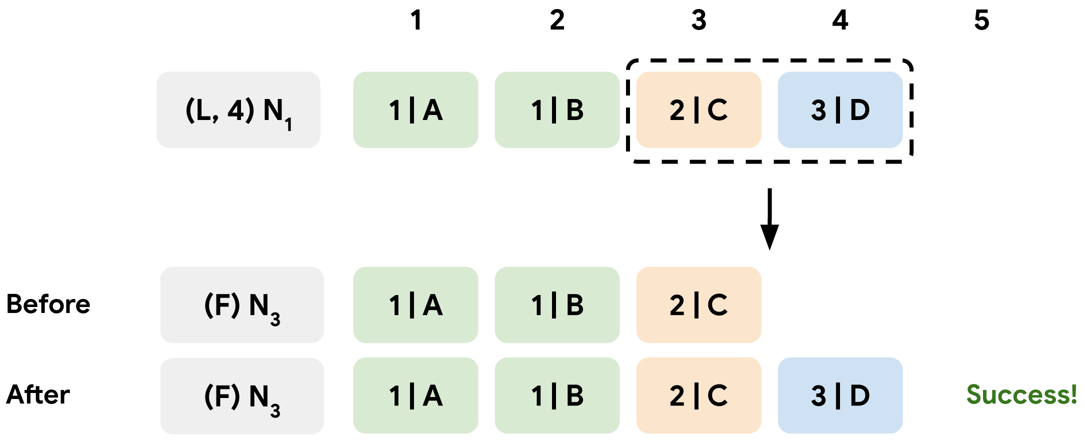
2 If a given entry is committed, all preceding entries are also committed

Slide is borrowed from [author's presentation](), restyled

Index

Term | Proposal

Replication consistency check



Each append from leader includes <index, term> of entry preceding new one(s), follower must contain matching entry; otherwise it rejects request leader retries with lower log index

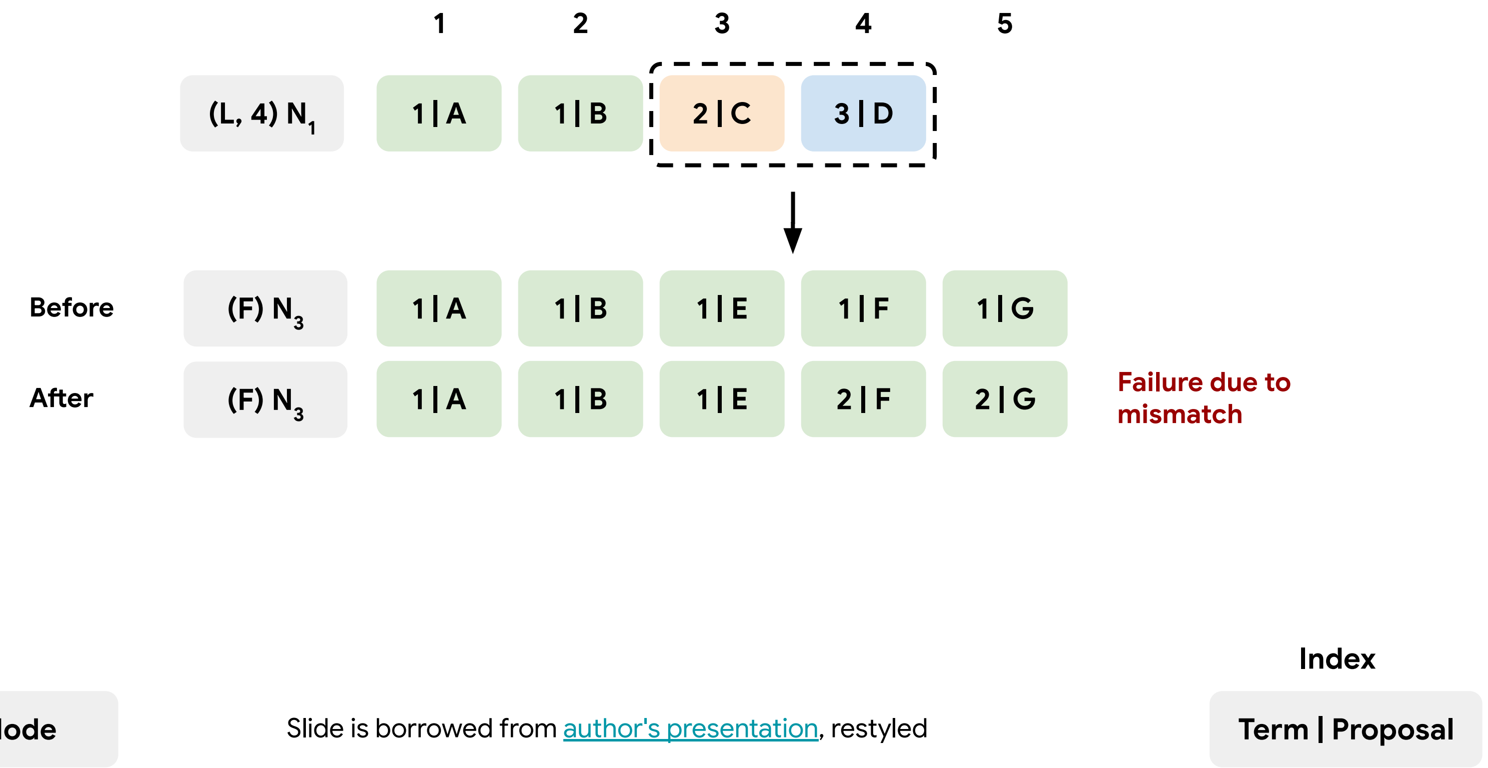
(Leader, Term) Node

Slide is borrowed from [author's presentation](), restyled

Index

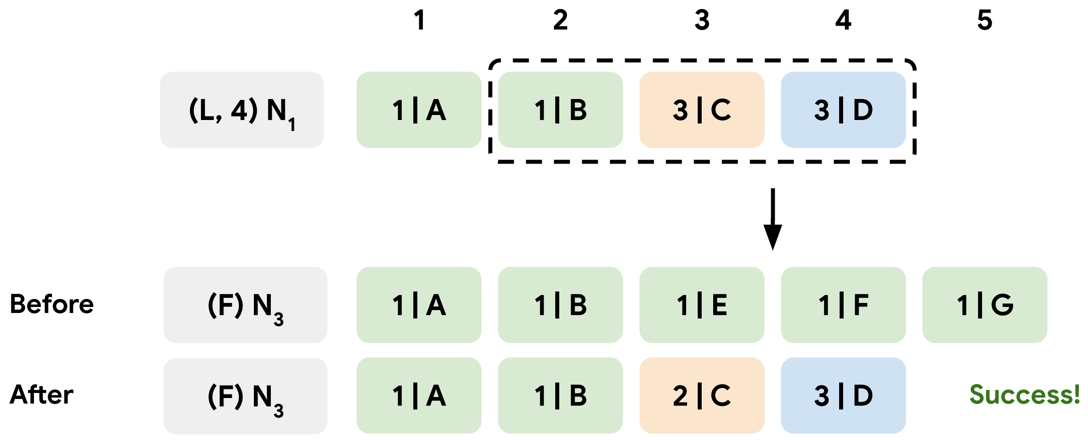
Term | Proposal

Replication consistency check



Slide is borrowed from author's presentation, restyled

Replication consistency check



(Leader, Term) Node

Slide is borrowed from author's presentation, restyled

Index

Term | Proposal

Leader completeness

<table>
<tr><th></th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th></tr>
<tr><td>(F) N_1</td><td>1 | A</td><td>1 | B</td><td>1 | C</td><td>2 | D</td><td>2 | E</td><td>3 | F</td><td>3 | G</td><td>3 | H</td><td></td></tr>
<tr><td>(F) N_2</td><td>1 | A</td><td>1 | B</td><td>1 | C</td><td>2 | D</td><td>2 | E</td><td>3 | F</td><td>3 | G</td><td></td><td></td></tr>
<tr><td>(F) N_5</td><td>1 | A</td><td>1 | B</td><td>1 | C</td><td>2 | D</td><td>2 | E</td><td>2 | P</td><td>2 | Q</td><td>2 | R</td><td></td></tr>
</table>

Electing leader for term 4

1. Once log entry committed, all future leaders must store that entry

2. Nodes with incomplete logs must not get elected: candidates include index and term of last log entry during vote. Voting node denies vote if its log is more up-to-date.
Logs ranked by <lastTerm, lastIndex>

(Leader, Term) Node

Slide is borrowed from author's presentation, restyled

Index

Term | Proposal

Hardened Raft

Applications add app specific guarantees (e.g. user level DP for analytics)

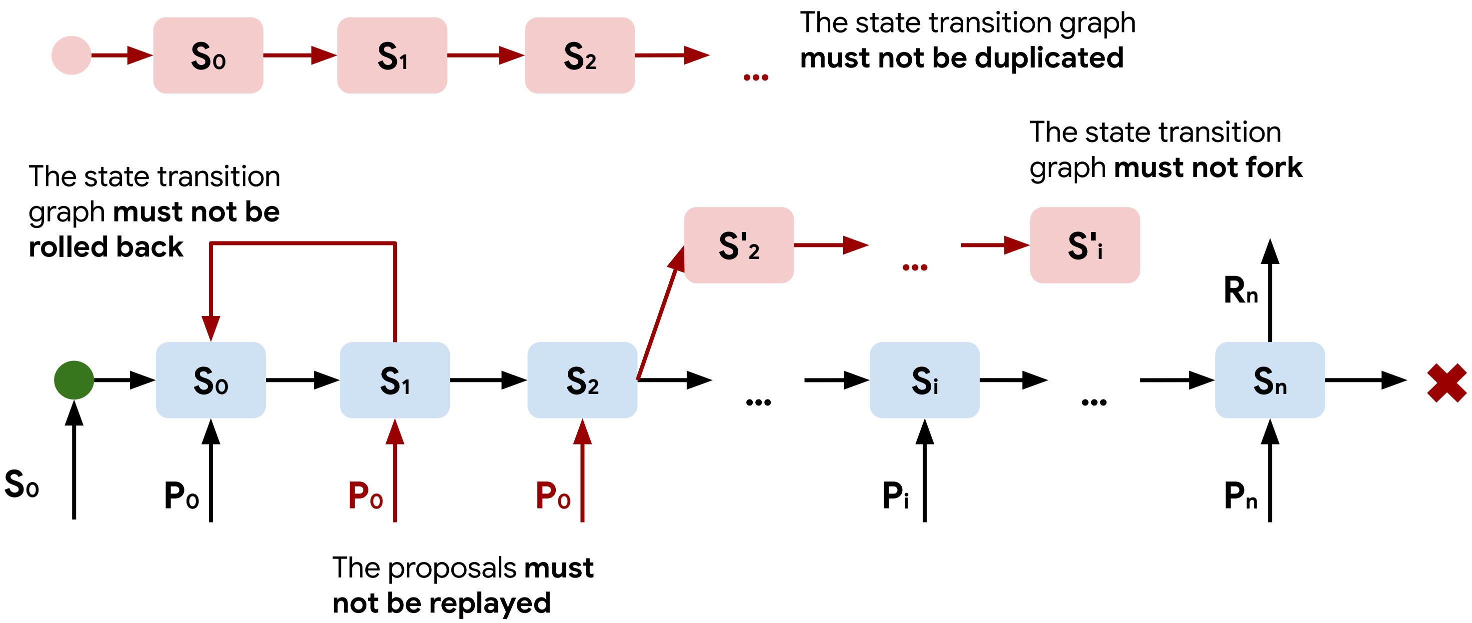
Protect *replicated state machines from rollback, fork or replay.

Must not forget state transitions

Must not allow identical instances to run concurrently

Must not allow inputs replay

Attackers may control network and TEE instance execution, substitute persisted state



Vectors of attacks

State persistence

Raft saves its state to the persistent storage for crash recovery, including the current term, the most recent vote and log.

Communication

Raft communicates with other nodes to maintain the consensus of the distrusted system, including leader election, replication and snapshot distribution.

Vectors of attacks

Node identity

Raft implementation (full details can be found in this repo) identifies nodes participating in its cluster by 64 bit integers, where node identity is used for the purposes of communication and membership. Raft relies on the fact that no two different nodes can share the same identity.

Log integrity

Raft performs log matching during replication using log entry index and term to identify a common log prefix. Raft doesn't perform log entry contents check.

Membership changes

Raft supports two mechanisms to change cluster membership, namely simple and joint. Both mechanisms rely on the replication mechanism to achieve consensus on the current cluster configuration.

TEE guarantees and properties

Tampering

TEE preserves its integrity and confidentiality during its computation, by leveraging techniques such as memory isolation, encryption and remote attestation.

Cloning

TEE instantiates with unique encryption and signing key pairs.
It is impossible to clone and run TEE instance with its encryption and signing keys.

*Machines running TEE instances are assumed to be diskless

Raft guarantees and properties

Safety

At most one leader can be elected in a given term (**election safety**), if a server has applied a log entry at a given index to its state machine, no other server will ever apply a different log entry for the same index (**state machine safety**).

Communication

Raft does not require ordered or reliable message delivery for correctness (**communication safety**), allowing attacker only affect computation availability through message delay or drop.

Log

A leader never overwrites or deletes entries in its log (**log continuity**), if two logs contain an entry with the same index and term, then the logs are identical in all entries up through the given index (**log matching**), committed entry present in logs of all leaders from higher terms (**log completeness**).

Attacks mitigations

Lifetime

Trusted Computation is **alive as long as the majority of the Raft cluster is alive**. If the majority ceases to exist, the Trusted Computation terminates (there are no restarts or recovery, the state is lost).

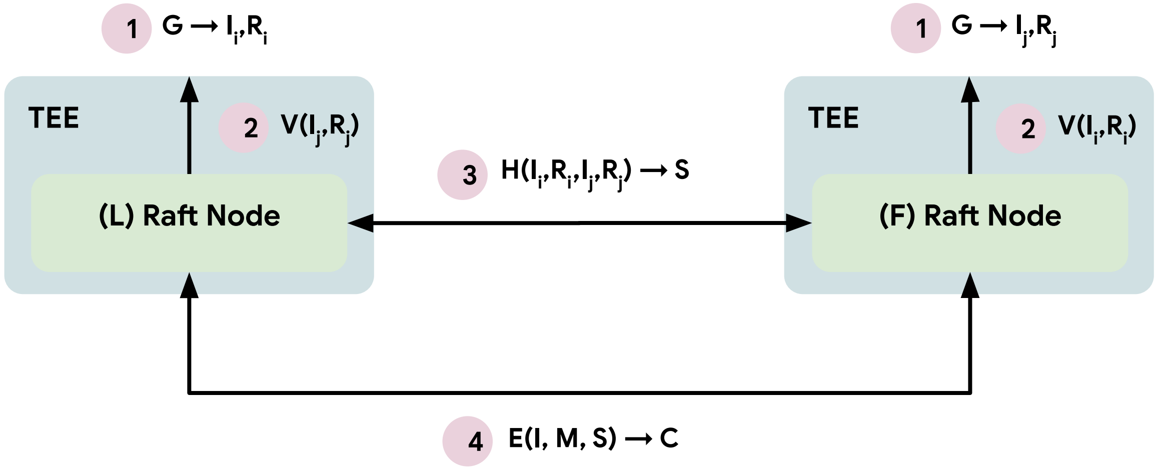
State persistence

The machines hosting TEEs are diskless, storing in distributed persistent storage will increase cost and complexity, decrease performance during normal operation. **Raft state is not stored in the persistent storage outside of the TEE. Raft state only resides inside of the encrypted TEE memory.**

Node Identity

Raft node identity must mean the same thing to all participants and must be resistant to impersonation. **Raft node identity is derived (hashed) from signing key pair (Sp / Sk). Raft node uses attestation report to prove it owns Sp / Sk and hence can be addressed during communication.**

End to end communication encryption to prevent tampering and snooping



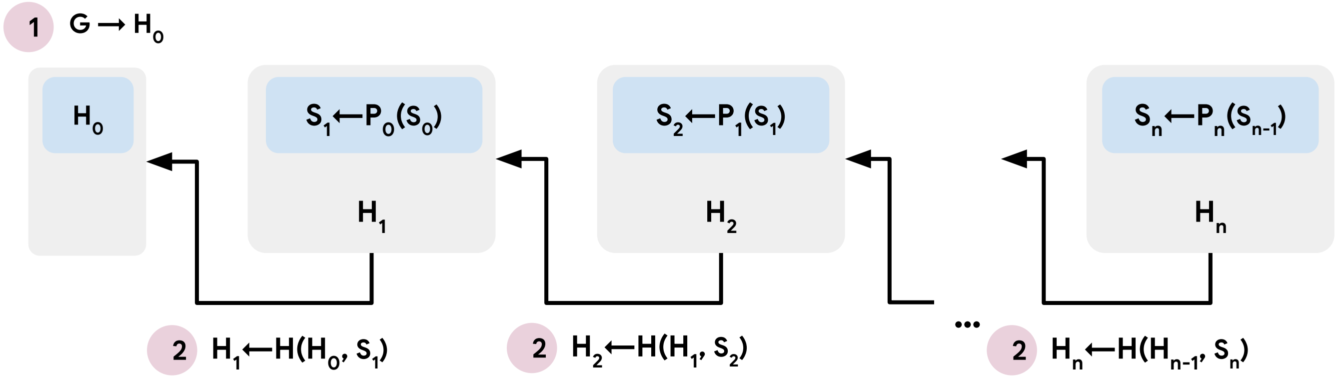
1 Generate node identity and attestation

2 Mutually verify behavior and identity

3 Perform hybrid public key encryption handshake to agree on secret

4 Encrypt messages with agreed secret

Replicated log integrity checks to prevent log substitution attacks



1 Generate replicated log identity during bootstrap

3 Maintain log integrity through chain of hashes

Platform

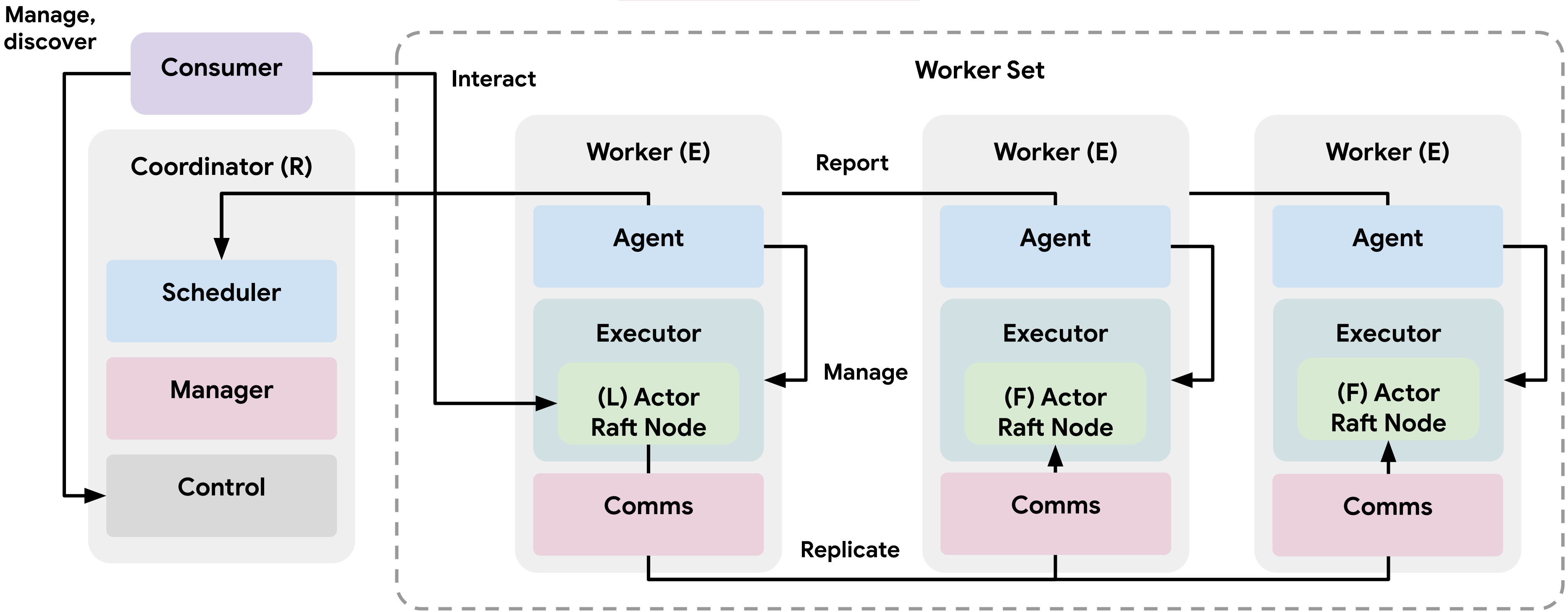
Platform is suitable for customer self-hosting with straightforward dependencies

Platform enables multi-tenant setup where customer owned workers are managed by team provided coordinator

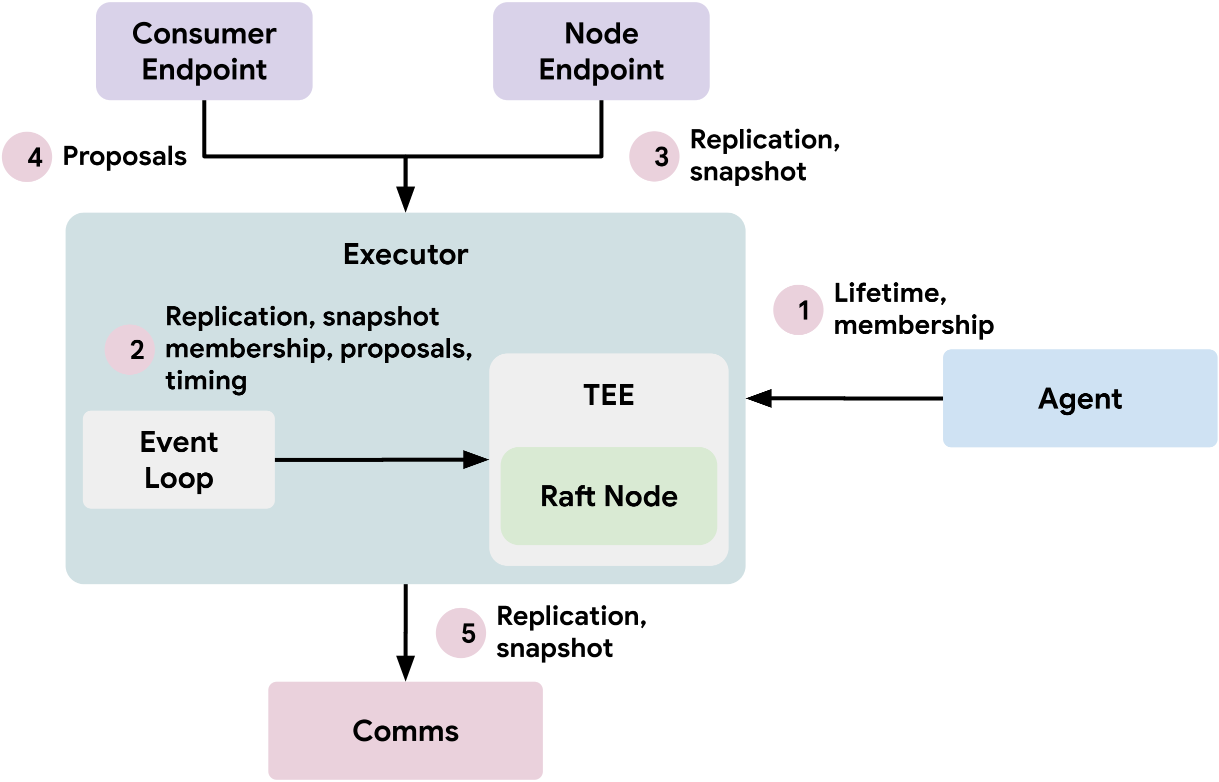
Enable self-hosted and managed scenarios, minimal implementation surface.

Platform only requires customer to implement application actor and worker (assembled from provided components)

Platform provides Raft nodes execution, placement and membership management, discovery and communication

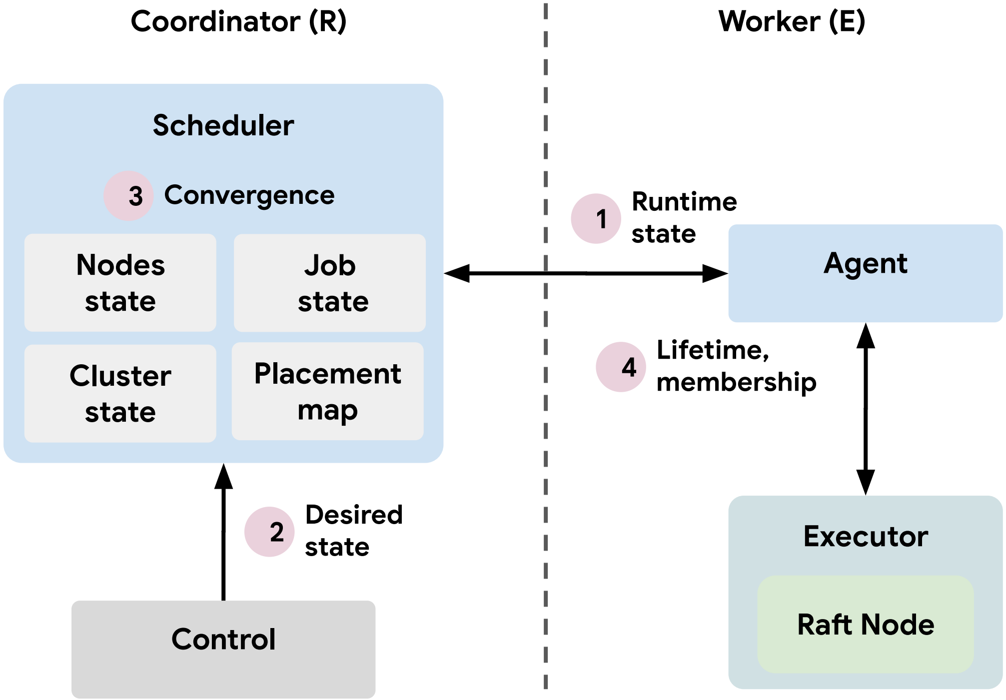


Zoom in on execution



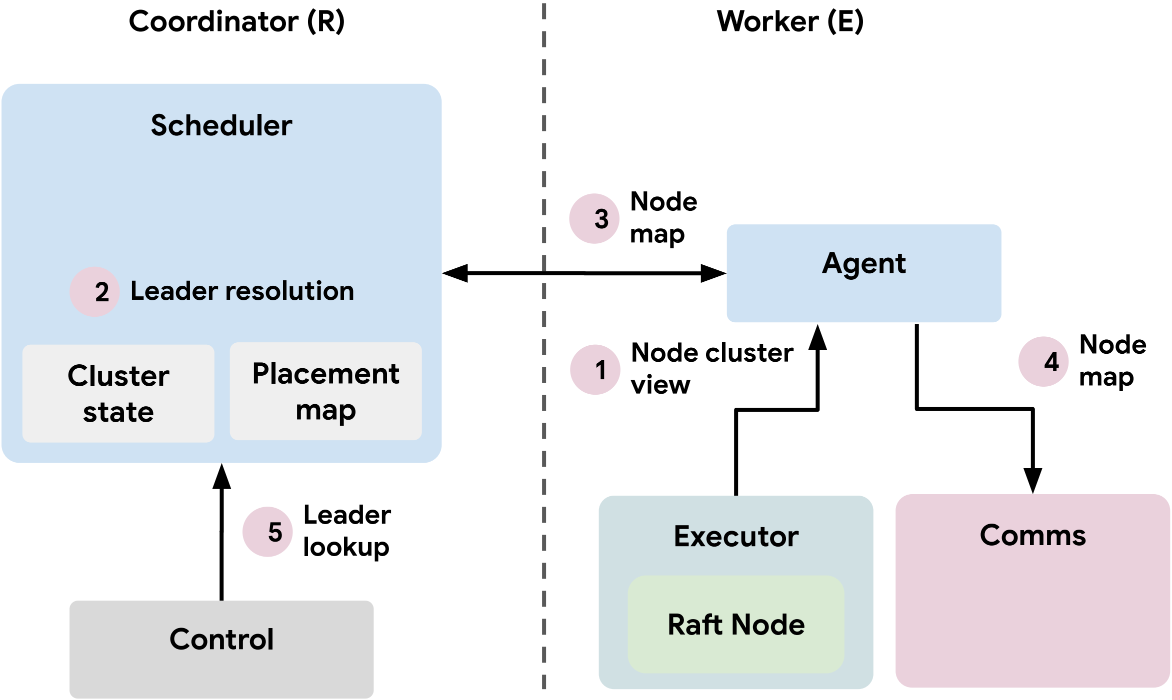
1. Instructs to create and destroy nodes, maintain Raft cluster
2. Runs event loop driving execution in TEE, events are replication, snapshot, membership, proposals, timing
3. Receives replication and snapshots from leader node
4. Receives consumer proposals for execution
5. Sends replication and snapshot to follower nodes

Zoom in on membership management



1. Report nodes state and cluster membership
2. Load desired job state
3. Converge runtime and desired state by creating and destroying nodes, changing their membership in Raft cluster
4. Issue node lifetime and membership commands

Zoom in on discovery and communication



1. Executor reports node's view of the Raft cluster
2. Scheduler resolves which view is most up to date based on Raft term
3. Scheduler builds and distributes node map that shows where nodes are placed, Raft cluster status
4. Comms uses node map to enable cross node communication, clients to make proposals
5. Lookup leader identity and location to issue proposals

Key Value Store

Core principles

State and computation offload

State persistence must be offloaded to a persistent blob storage where the data is encrypted at rest (trading expensive memory for cheap persistent blob storage). State processing must be offloaded to stateless elastically scalable service (trading strict serial execution for relaxed parallel execution).

Coarse reads and writes

The coarse reads and writes are crucial for the cost amortization. Therefore data is organized into batches that are processed as an atomic unit, thus reducing overhead associated with the number of stored blobs and the privacy state maintained by the replicated state machine.

Core principles

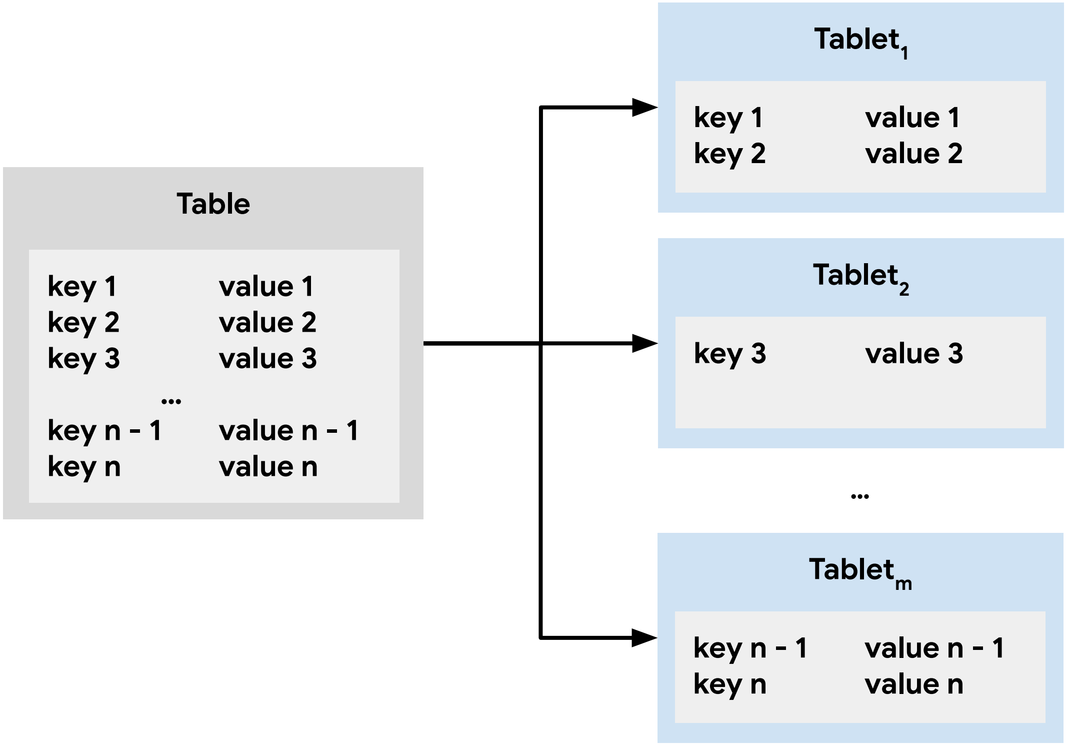
Batch requests with write through cache

Requests batching serves as a bridge between fine and coarse interface while write through cache further amortizes the costs. Note that individual request latency is traded (increased) for the overall throughput (increased). Coarse operations lessen the load on the replicated state machine that are prohibitively expensive for the fine operations.

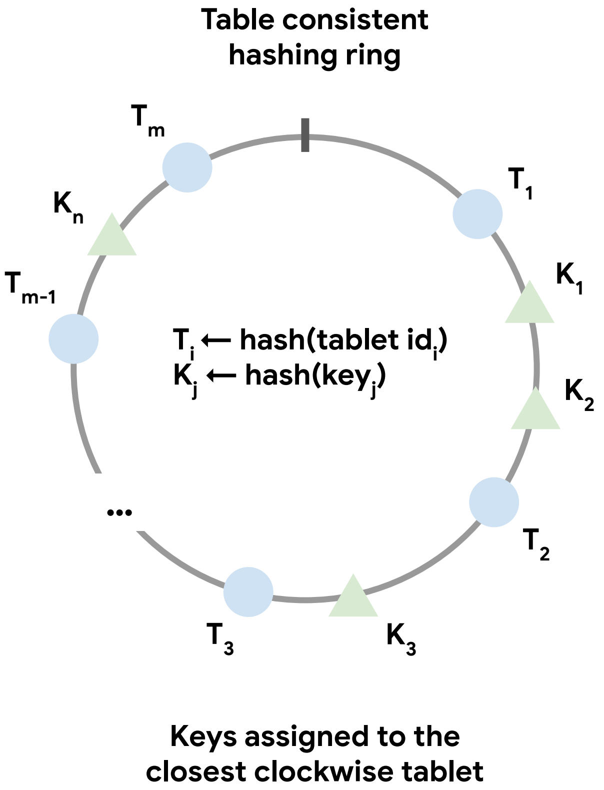
Replicated rollback protected metadata store

Replicated state machine running inside of the TEEs provides a rollback protected replicated store that maintains metadata for the data batches. Specifically data batch is considered updated only after metadata update is committed to the metadata store.

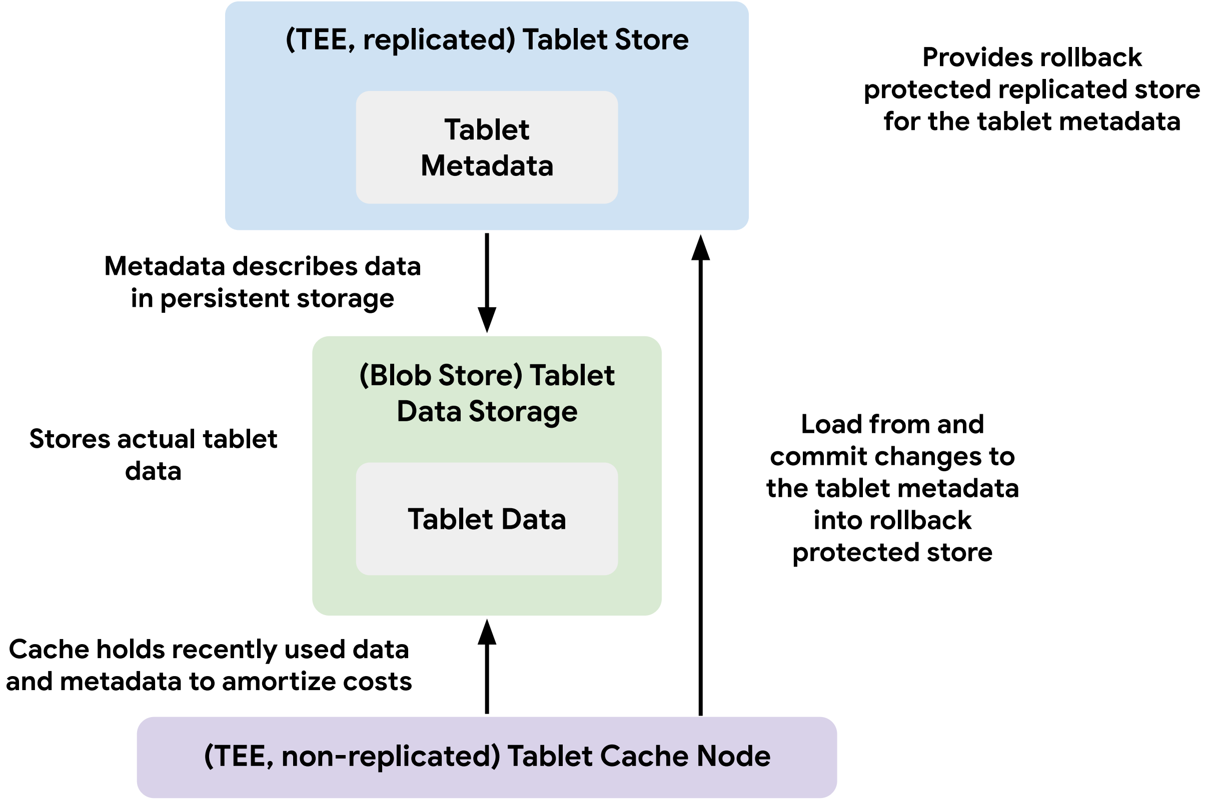
Split tables into smaller atomic units called tablets



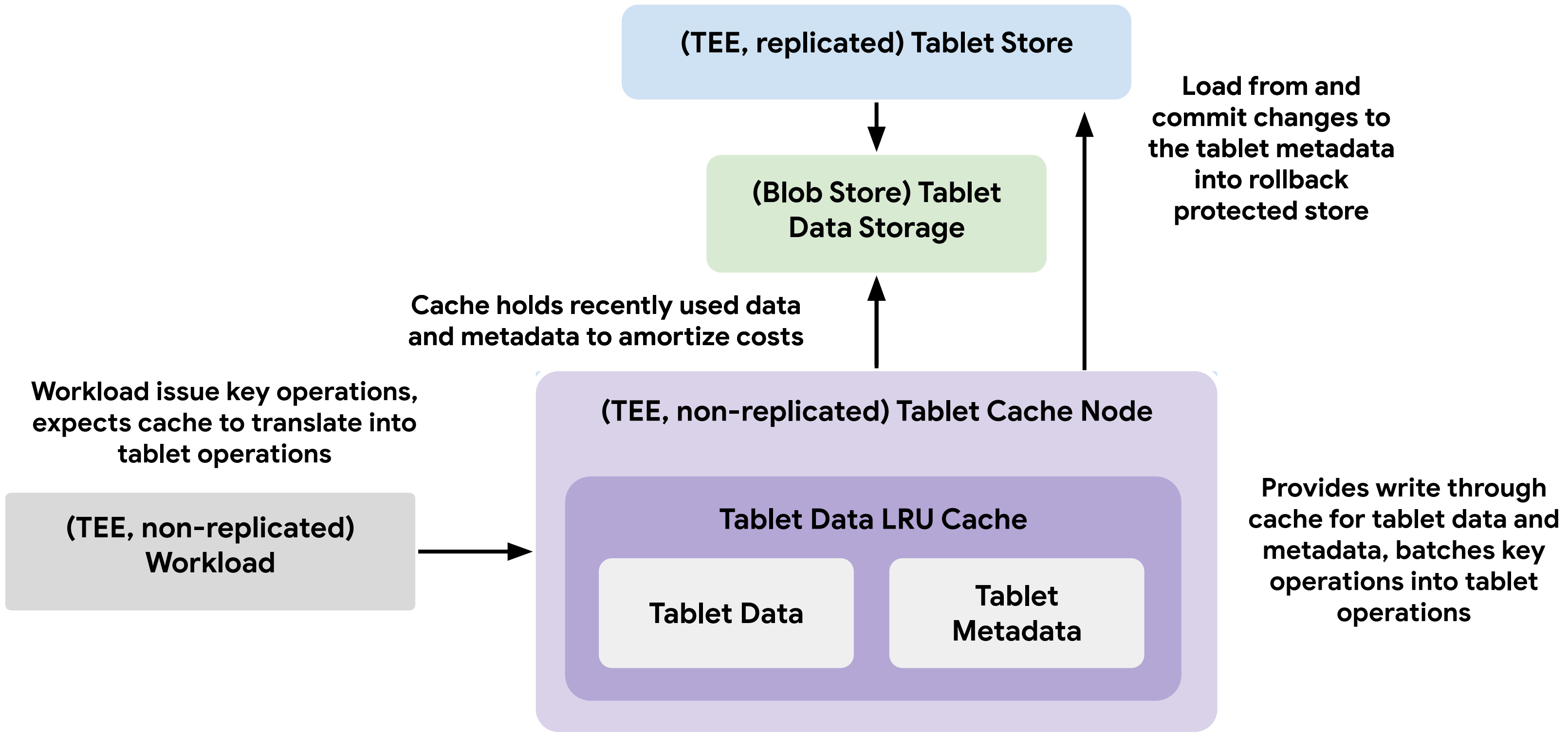
Assign keys to tablets using consistent hashing



Tablet Store manages metadata, Tablet Data Storage manages data



Tablet Cache Node translates key ops into tablet ops, provides write through cache



Thank you