ArangoDB

A RAFT based application store on ArangoDB’s Agency

Who?

- Myself
  - C++ autodidact > 20 yrs
  - Theoretical physics / compiler building PhD, MRI physics research
- ArangoDB
  - NoSQL multi-model database (documents, kv, graphs, ...)
    - Document: joins, transactions, schemaless, JSON, secondary indexes, compact storage
    - Graph: pattern matching, shortest path, distributed, nested properties, traversals, transactions
  - HA, Clustering, DC2DC
    - Multi master, horizontal scaling, resilient and self healing, query optimiser, sync replication
  - AQL covers all
  - Foxx microservice server
  - Cloud infrastructure support for k8s / dcos

Introduction

- Distributed systems have fundamental problems
  - Configure once
  - Consensus - Ground truth shared by all

- Solutions
  - PAXOS too complicated - hard to get right
  - RAFT github.io/RAFT

Introduction

- Distributed systems have fundamental problems
  - Configure once
  - Consensus - Ground truth shared by all

- Solutions
  - PAXOS too complicated - hard to get right
  - RAFT github.io/RAFT
  - Actually, …🤣… we tried to do one of our own.

Introduction

- Distributed systems have fundamental problems
  - Configure once
  - Consensus - Ground truth shared by all

- Solutions
  - PAXOS too complicated - hard to get right
  - RAFT github.io/RAFT

- What is the application store anyway?
  - Foxx becomes Consensus Foxx
    Microservice container for running consensus code

Nomenclature

Some nomenclature up front

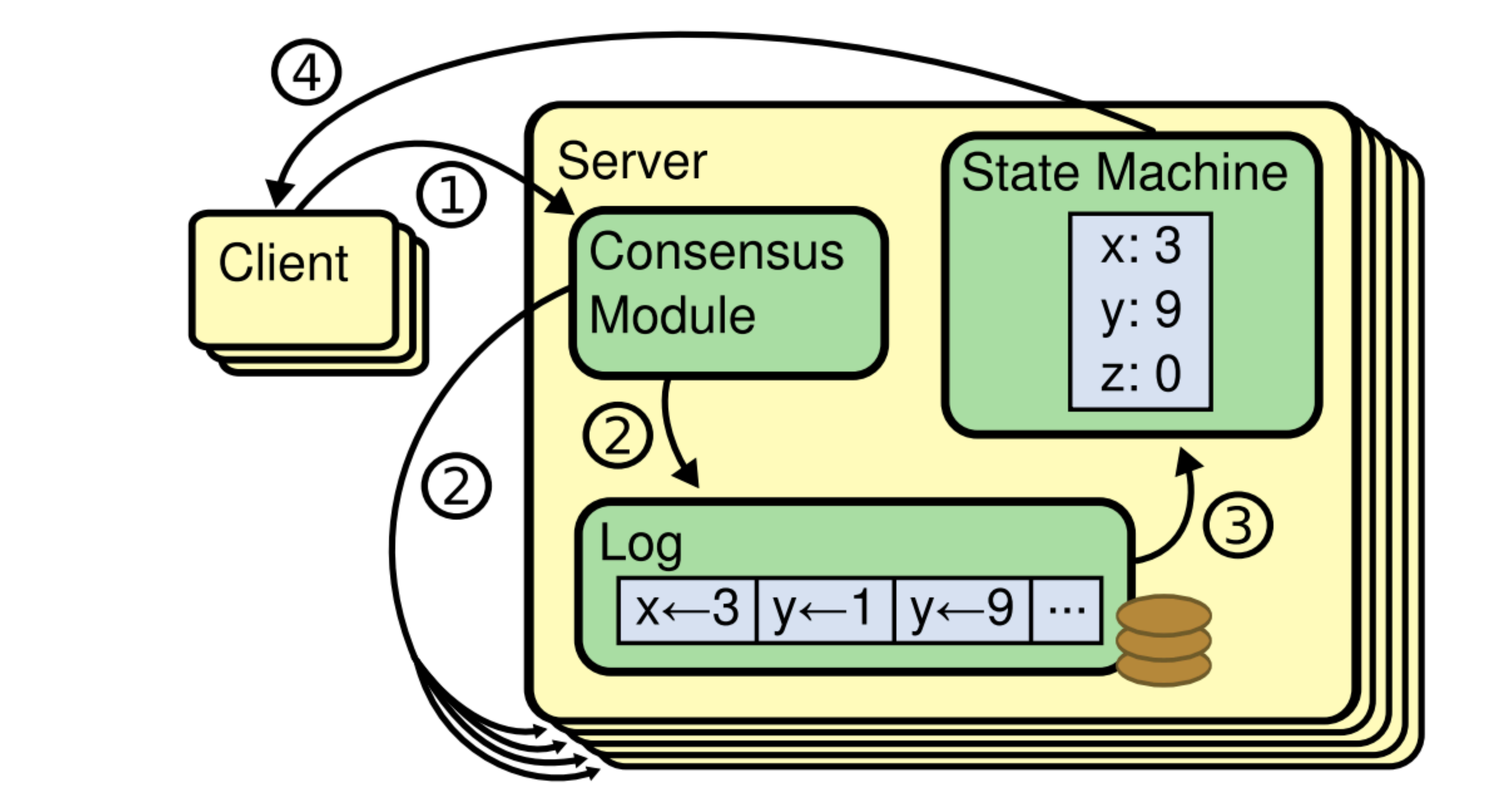
- Consensus

- Replicated log
  Ordered list of log entries $\mathtt{l}_{\mathtt{i}}$
  `a:12 ,a:12 ,a++ ,a++,...`

- State machine
  Application of $\mathtt{l}_{\mathtt{i}}$
  `{a:12},{a:12},{a:13},{a:14},...`

> nihil novi nisi commune consensu
> nothing new unless by the common consensus
> – law of the polish-lithuanian common-wealth, 1505

RAFT


4
1
Client
Server
Consensus Module
State Machine
x: 3
y: 9
z: 0
2
2
Log
x←3
y←1
y←9
...
3

State machine

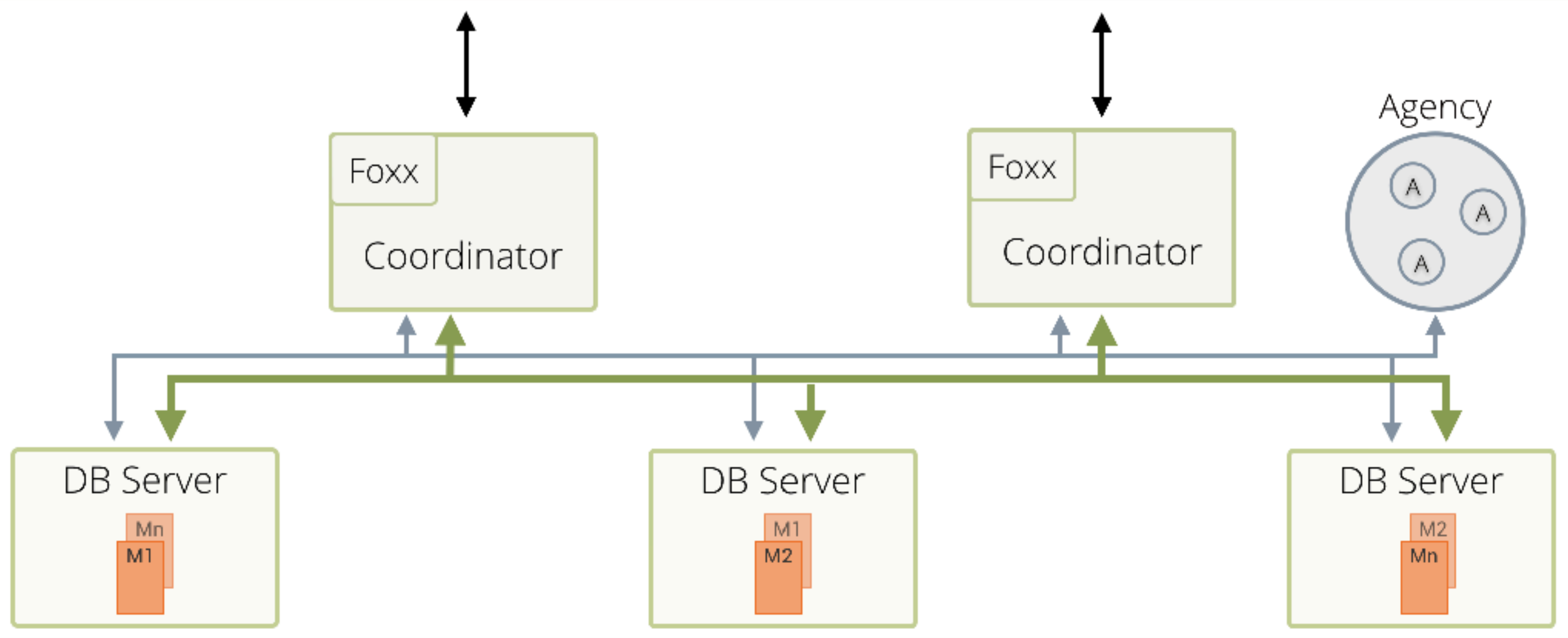
- Internal state exposed through …
- External interaction (public API)
- Problem:
  - Easy: local and unreliable
    - Configuration file
    - Local database
    - …
  - Surprisingly hard: global and reliable
    - Replicated log
    - Block chain
    - …
  - Handles non-Byzantine errors

Consensus

- Distributed system
  - One time initialisation
  - Configuration management

- Fault tolerance - Resilience
  - Network partitions, split brain
  - Hardware failure

- Bottle neck!



Paxos

- Few people know Paxos anywhere near completeness. And completeness is KEY!
- Significant gaps between the description the needs of a real world system
- Inefficient: 2 rounds of messages to choose one value

RAFT

- While true
  - Leadership election

    Randomised waits. Time limits on vote validity.
  - Rebuild state machine

    Apply all compactions. Apply all logs.
  - Serve requests

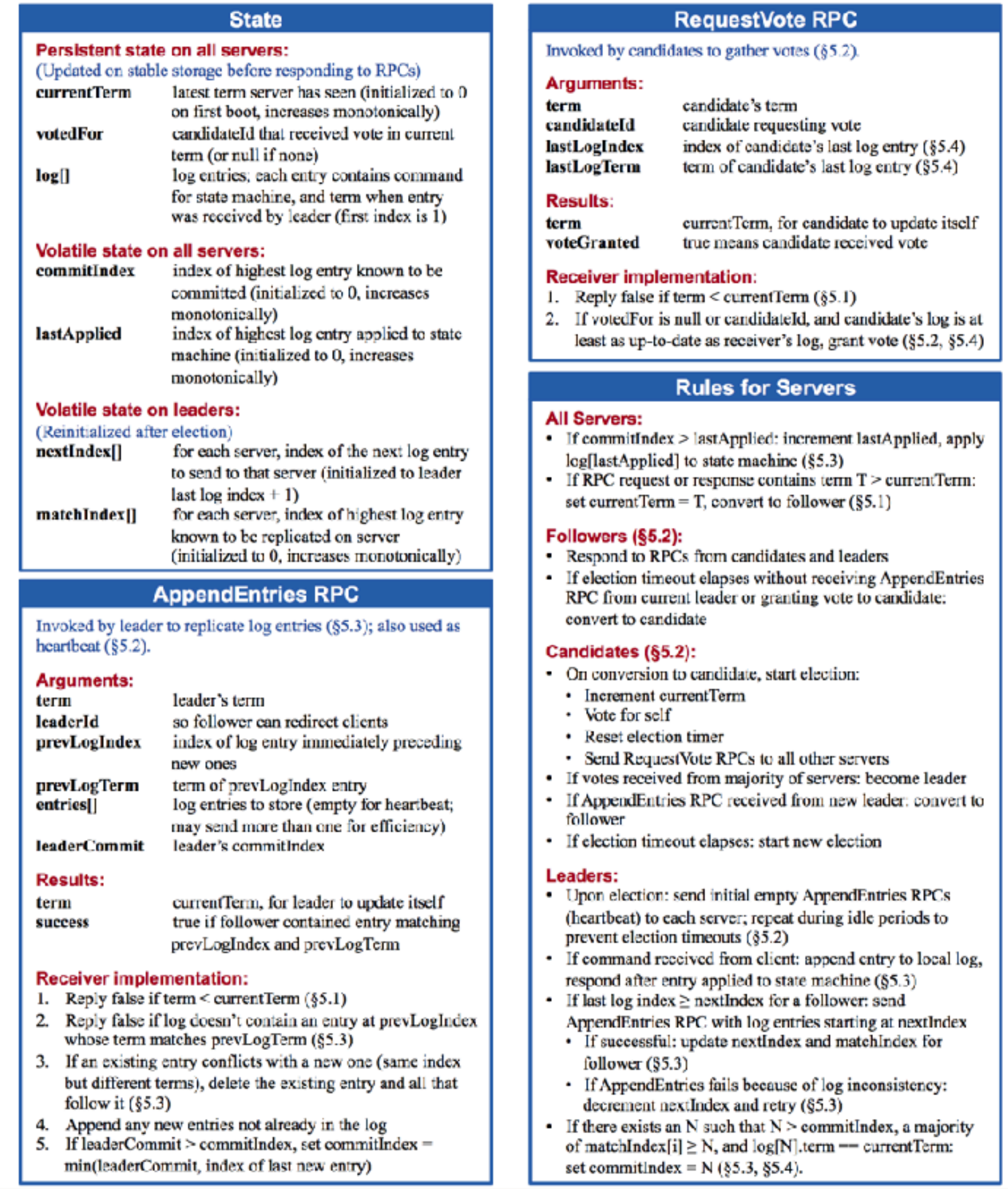
    Leader: Append new logs. Spearhead/Read.
    Wait for majority to commit. Respond to reads.
    Keep followers devout.
    Followers: Append new logs from leader only. Report to leader.
    Candidate for leadership, if gone too long without hearing from leader.

RAFT

- Formal proof
- 100s of implementations (mostly Rust, Go, some C++)
- Performance analysis
- User study of understandability

RAFT

- Cheat sheet:
  - RequestVote
  - sendAppendEntries
  - Rules for all roles
  - Does not cover
    - Compaction
    - Resizing



State

Persistent state on all servers:

(Updated on stable storage before responding to RPCs)

currentTerm	latest term server has seen (initialized to 0 on first boot, increases monotonically)
votedFor	candidateId that received vote in current term (or null if none)
log[]	log entries; each entry contains command for state machine, and term when entry was received by leader (first index is 1)

Volatile state on all servers:

commitIndex	index of highest log entry known to be committed (initialized to 0, increases monotonically)
lastApplied	index of highest log entry applied to state machine (initialized to 0, increases monotonically)

Volatile state on leaders:

(Reinitialized after election)

nextIndex[]	for each server, index of the next log entry to send to that server (initialized to leader last log index + 1)
matchIndex[]	for each server, index of highest log entry known to be replicated on server (initialized to 0, increases monotonically)

AppendEntries RPC

Invoked by leader to replicate log entries (§5.3); also used as heartbeat (§5.2).

Arguments:

term	leader's term
leaderId	so follower can redirect clients
prevLogIndex	index of log entry immediately preceding new ones
prevLogTerm	term of prevLogIndex entry
entries[]	log entries to store (empty for heartbeat; may send more than one for efficiency)
leaderCommit	leader's commitIndex

Results:

term	currentTerm, for leader to update itself
success	true if follower contained entry matching prevLogIndex and prevLogTerm

Receiver implementation:

1. Reply false if term < currentTerm (§5.1)
2. Reply false if log doesn't contain an entry at prevLogIndex whose term matches prevLogTerm (§5.3)
3. If an existing entry conflicts with a new one (same index but different terms), delete the existing entry and all that follow it (§5.3)
4. Append any new entries not already in the log
5. If leaderCommit > commitIndex, set commitIndex = min(leaderCommit, index of last new entry)

RequestVote RPC

Invoked by candidates to gather votes (§5.2).

Arguments:

term	candidate's term
candidateId	candidate requesting vote
lastLogIndex	index of candidate's last log entry (§5.4)
lastLogTerm	term of candidate's last log entry (§5.4)

Results:

term	currentTerm, for candidate to update itself
voteGranted	true means candidate received vote

Receiver implementation:

1. Reply false if term < currentTerm (§5.1)
2. If votedFor is null or candidateId, and candidate's log is at least as up-to-date as receiver's log, grant vote (§5.2, §5.4)

Rules for Servers

All Servers:

- If commitIndex > lastApplied: increment lastApplied, apply log[lastApplied] to state machine (§5.3)
- If RPC request or response contains term T > currentTerm: set currentTerm = T, convert to follower (§5.1)

Followers (§5.2):

- Respond to RPCs from candidates and leaders
- If election timeout elapses without receiving AppendEntries RPC from current leader or granting vote to candidate: convert to candidate

Candidates (§5.2):

- On conversion to candidate, start election:
  - Increment currentTerm
  - Vote for self
  - Reset election timer
  - Send RequestVote RPCs to all other servers
- If votes received from majority of servers: become leader
- If AppendEntries RPC received from new leader: convert to follower
- If election timeout elapses: start new election

Leaders:

- Upon election: send initial empty AppendEntries RPCs (heartbeat) to each server; repeat during idle periods to prevent election timeouts (§5.2)
- If command received from client: append entry to local log, respond after entry applied to state machine (§5.3)
- If last log index ≥ nextIndex for a follower: send AppendEntries RPC with log entries starting at nextIndex
  - If successful: update nextIndex and matchIndex for follower (§5.3)
  - If AppendEntries fails because of log inconsistency: decrement nextIndex and retry (§5.3)
- If there exists an N such that N > commitIndex, a majority of matchIndex[i] ≥ N, and log[N].term == currentTerm: set commitIndex = N (§5.3, §5.4).

RAFT

- Safety
  - Allow most one winner per term
  - Each server gives only one vote per term (persist on disk)
  - Majority required to win election
- Liveness
  - Election timeout random
  - One server usually times out before others. some candidate eventually wins
  - Works well if T >> round trip ArangoDB: .5s - 2.5s

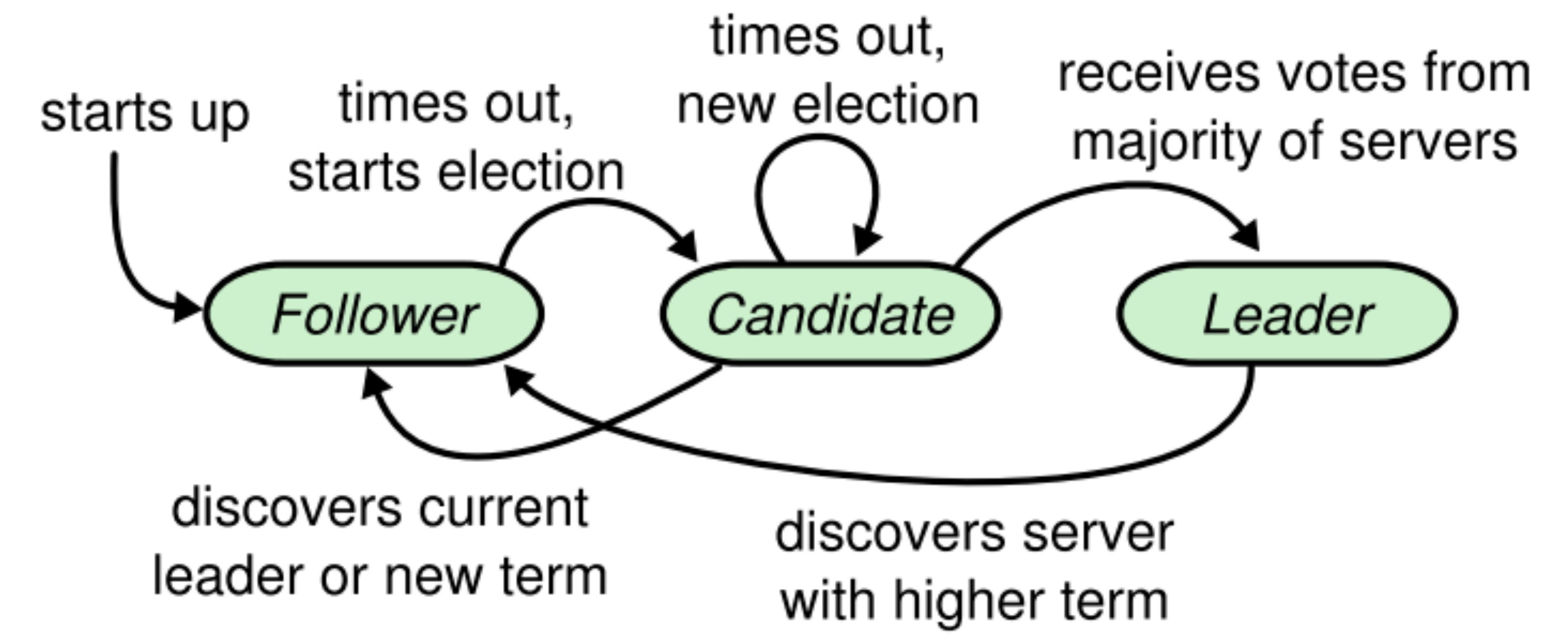
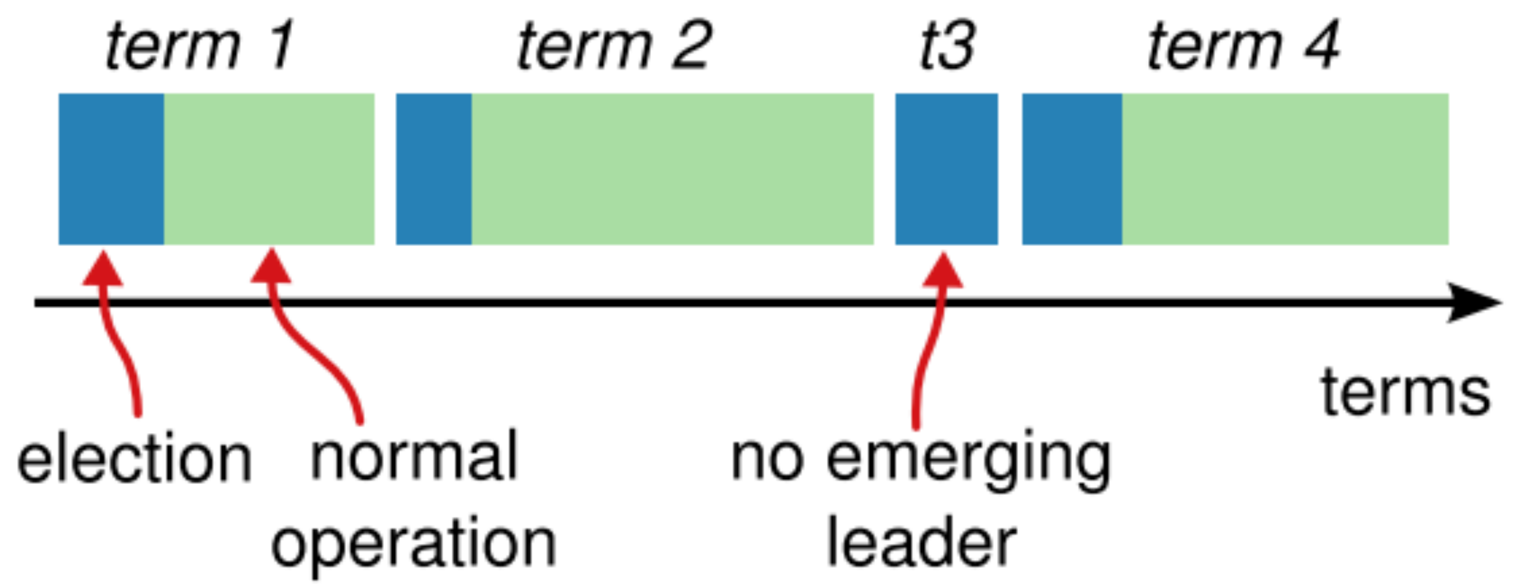
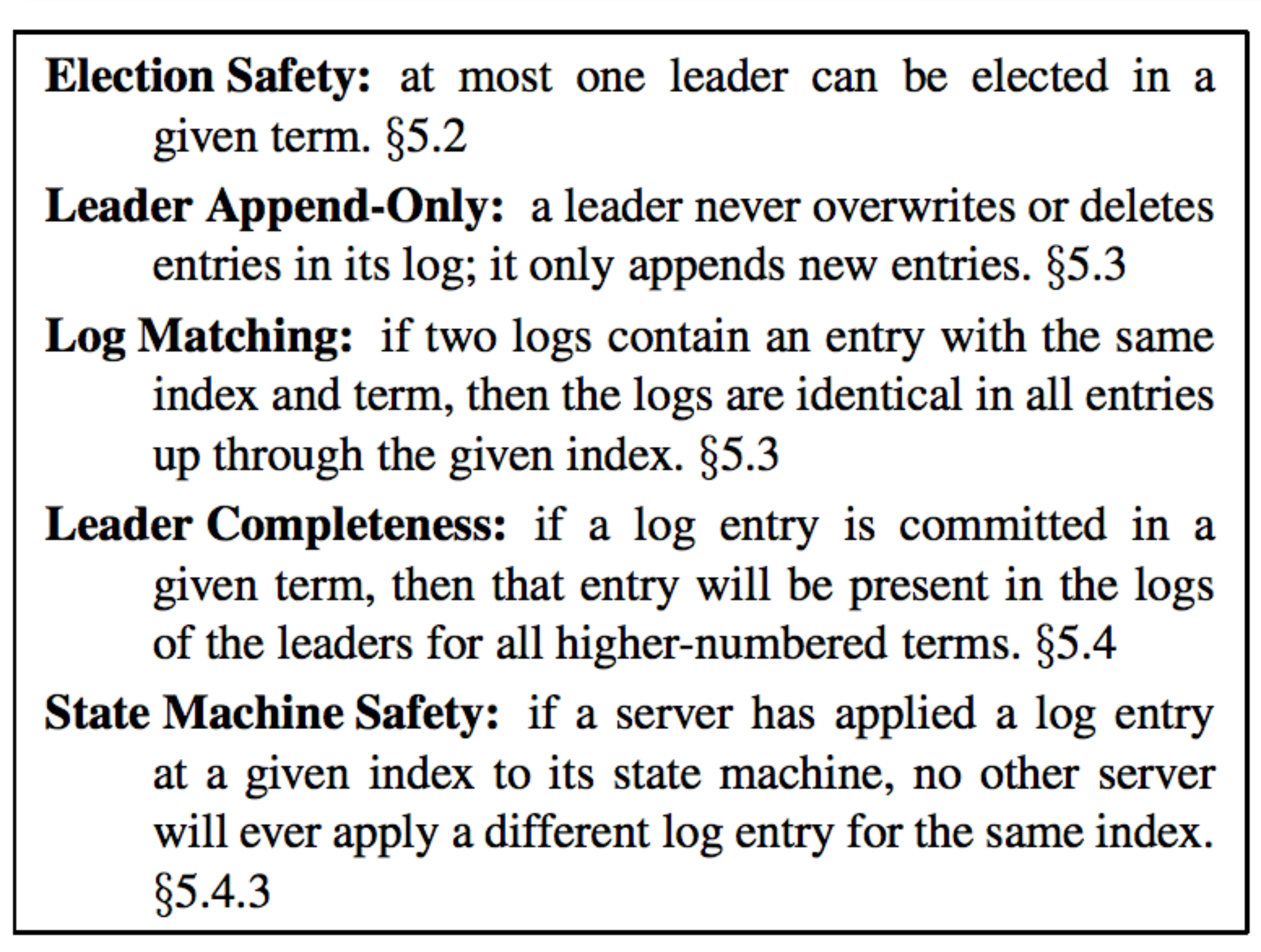

starts up
times out, starts election
times out, new election
receives votes from majority of servers
Follower
Candidate
Leader
discovers current leader or new term
discovers server with higher term

Figure 4: Server states. Followers only respond to requests from other servers. If a follower receives no communication, it becomes a candidate and initiates an election. A candidate that receives votes from a majority of the full cluster becomes the new leader. Leaders typically operate until they fail.



RAFT



Election Safety: at most one leader can be elected in a given term. §5.2

Leader Append-Only: a leader never overwrites or deletes entries in its log; it only appends new entries. §5.3

Log Matching: 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. §5.3

Leader Completeness: if a log entry is committed in a given term, then that entry will be present in the logs of the leaders for all higher-numbered terms. §5.4

State Machine 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. §5.4.3

RAFT

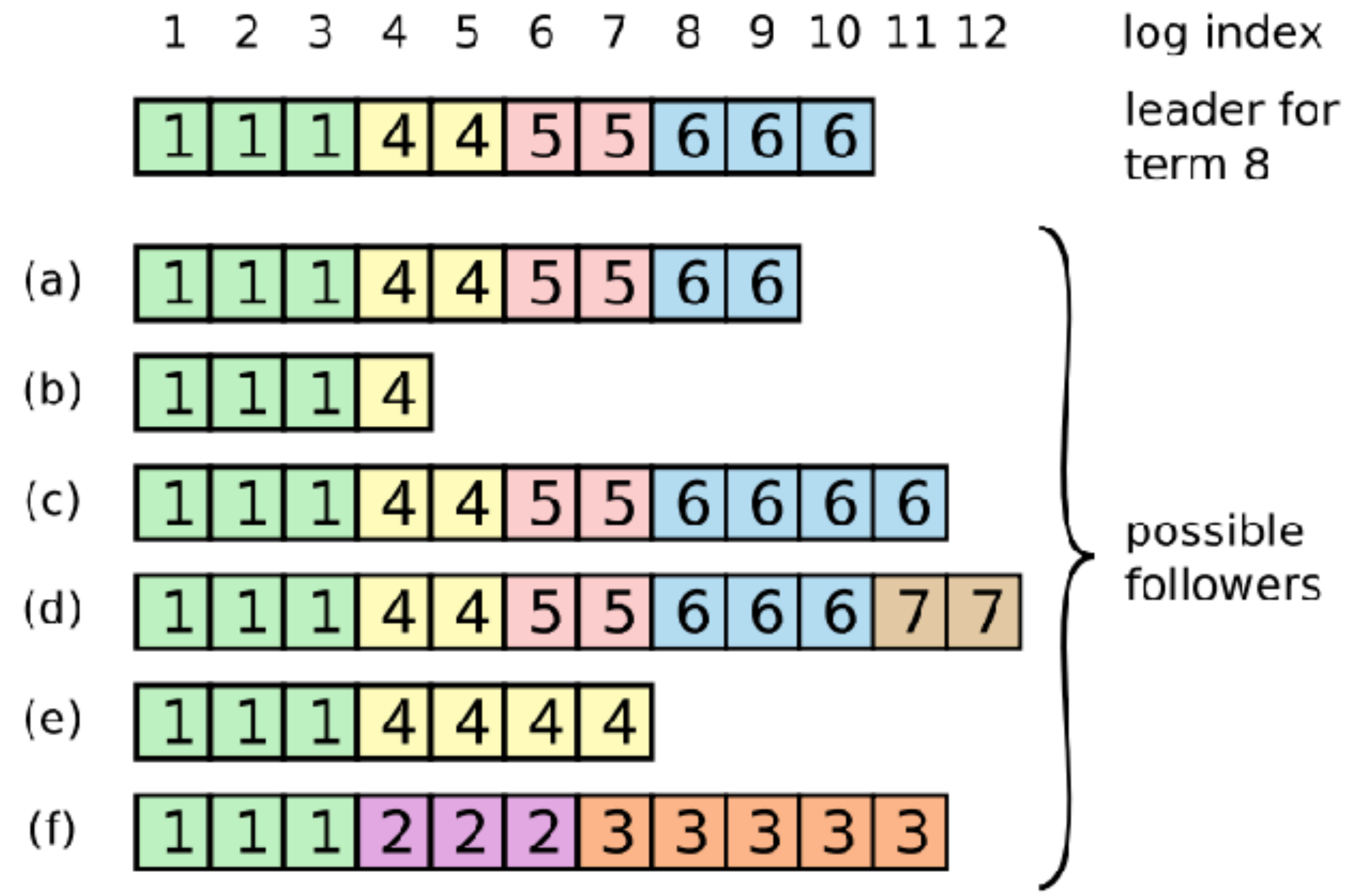


Figure 7: When the leader at the top comes to power, it is possible that any of scenarios (a–f) could occur in follower logs. Each box represents one log entry; the number in the box is its term. A follower may be missing entries (a–b), may have extra uncommitted entries (c–d), or both (e–f). For example, scenario (f) could occur if that server was the leader for term 2, added several entries to its log, then crashed before committing any of them; it restarted quickly, became leader for term 3, and added a few more entries to its log; before any of the entries in either term 2 or term 3 were committed, the server crashed again and remained down for several terms.

ArangoDB agency

- Let's read some code

  github.com/arangodb/arangodb/tree/3.3/arangod/Agency

- `arangodb::consensus`
  - `Inception` - Gossip protocol for establishing the agency
  - **`Agent`** - Main thread does all the RAFT skeleton work
    `Agent::sendAppendEntries`, **`Agent::recvAppendEntries`**
  - `Constituent` - Election mechanism
  - **`State`** - Replicated log, Compaction
  - **`Store`** - State machine (Spearhead / Committed)
  - `Supervision, Job, ...` Goodies :)

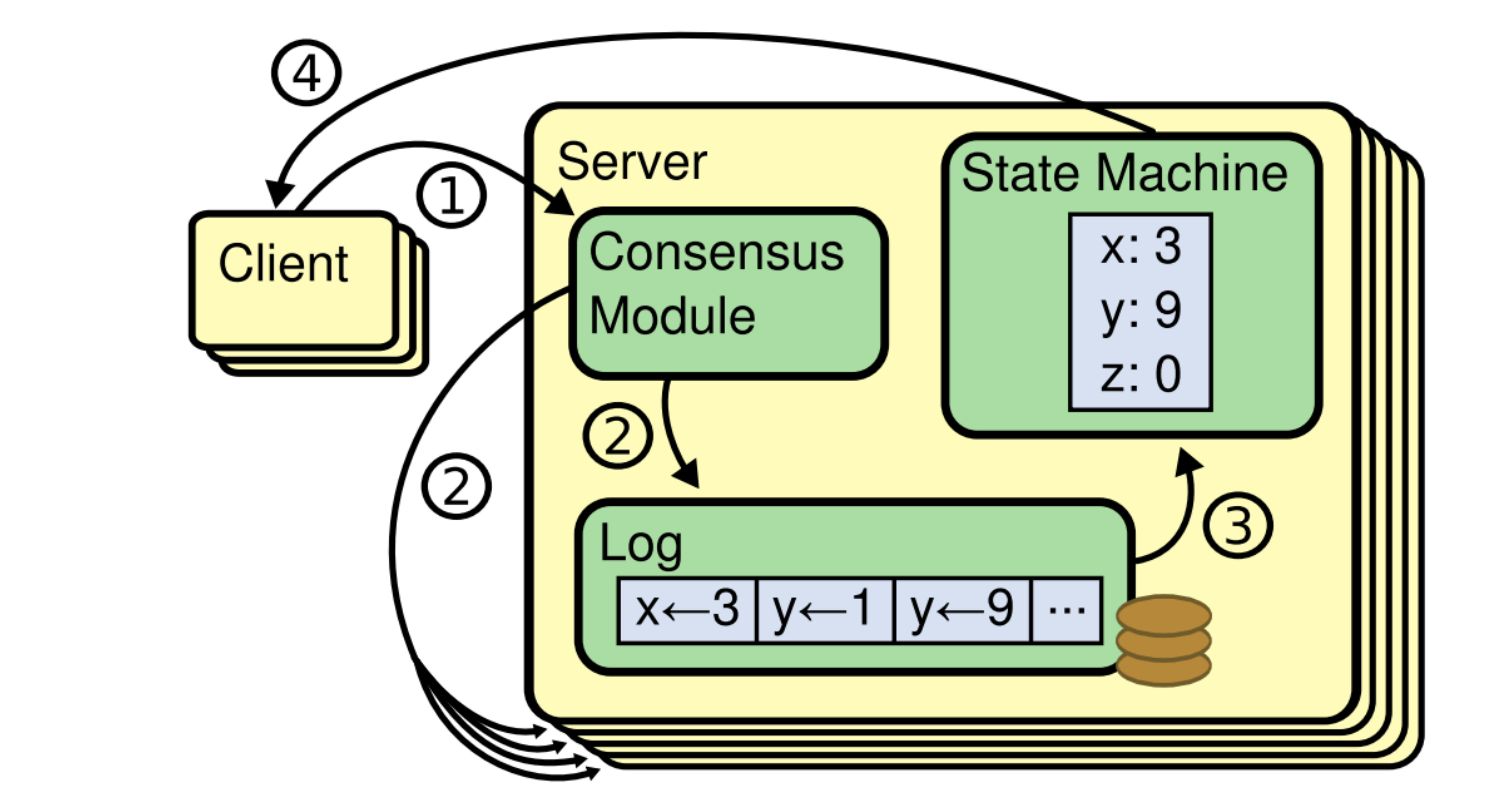
ArangoDB agency

- Nice, so what ..?
  - Try to implement a realtime protocol on real computers
    - Gotta stick to the rules 100% or you are …
  - What does time mean anyway?
    - `std::chrono::steady_clock` wherever durations calculated
    - `std::chrono::system_clock` whereever user input
  - HOW THE F**K DO YOU DEBUG S**T like this?
    - Deterministic debugging (gotta love **`rr`**)

ArangoDB agency

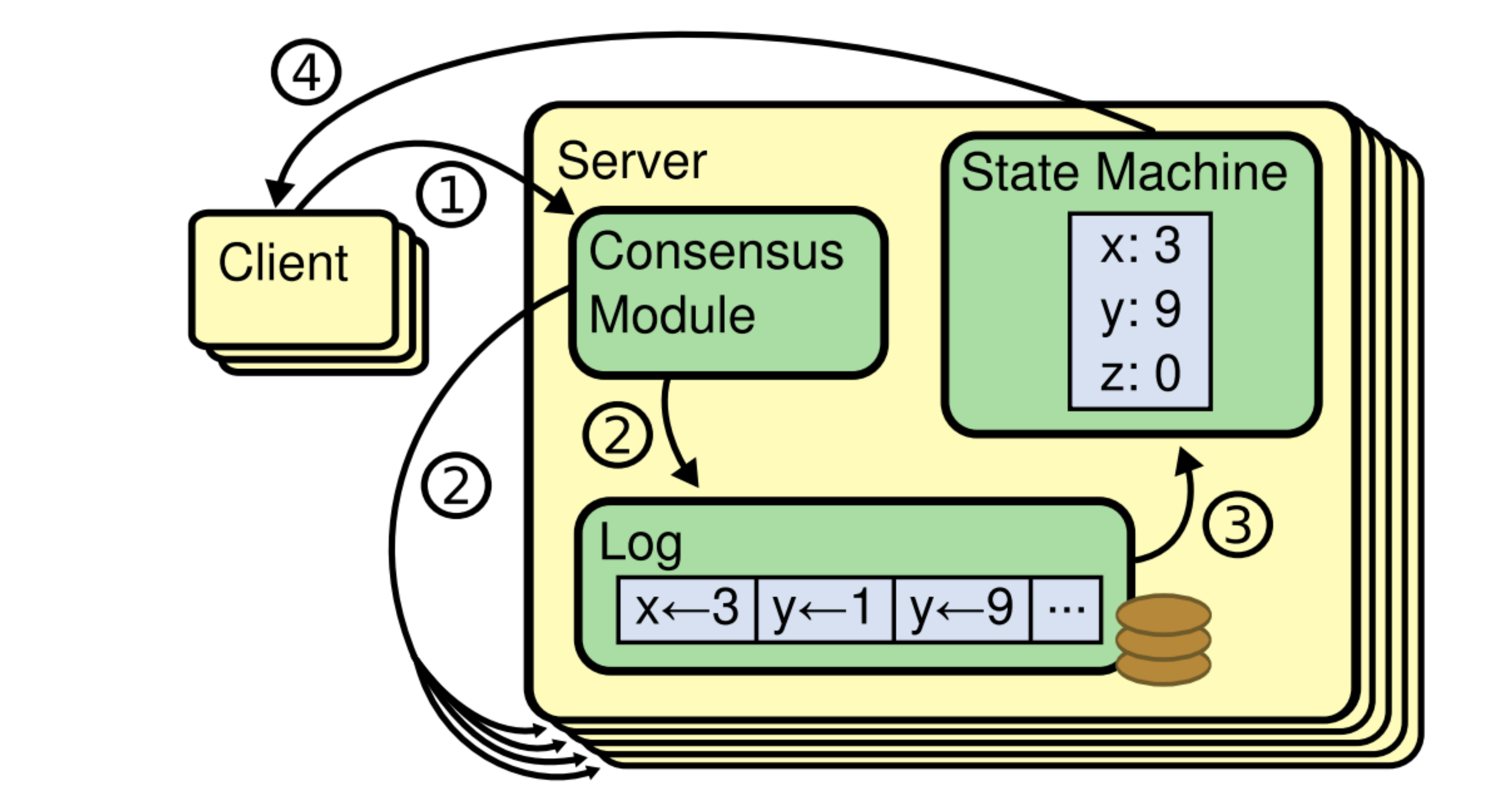
- We went all the way:
  - Compaction
  - Replicated program store

Something missing



Microservice container with RAFT

- Cluster supervision
  - Detecting server failures + evasive action
  - Cleaning out servers for maintenance / shutdown
  - Moving shards around the cluster
  - …



+ Microservice Container

Thanks for listening

- Example Code in go

  https://github.com/neunhoef/AgencyUsage

- Slides will be uploaded to

  https://www.arangodb.com/speakers/kaveh-vahedipour

- Reach me

  Mail kaveh@arangodb.com

  Twitter @kv4all