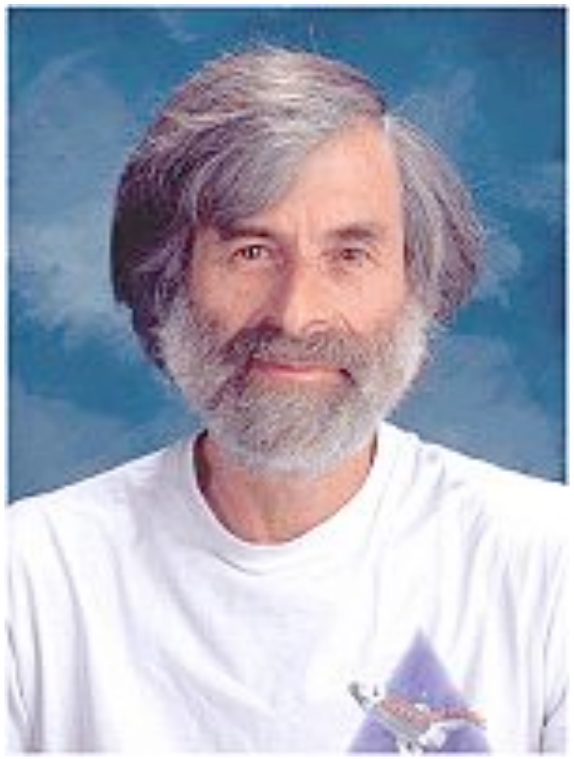
Failure, replication, replicated state machines (RSM), and consensus

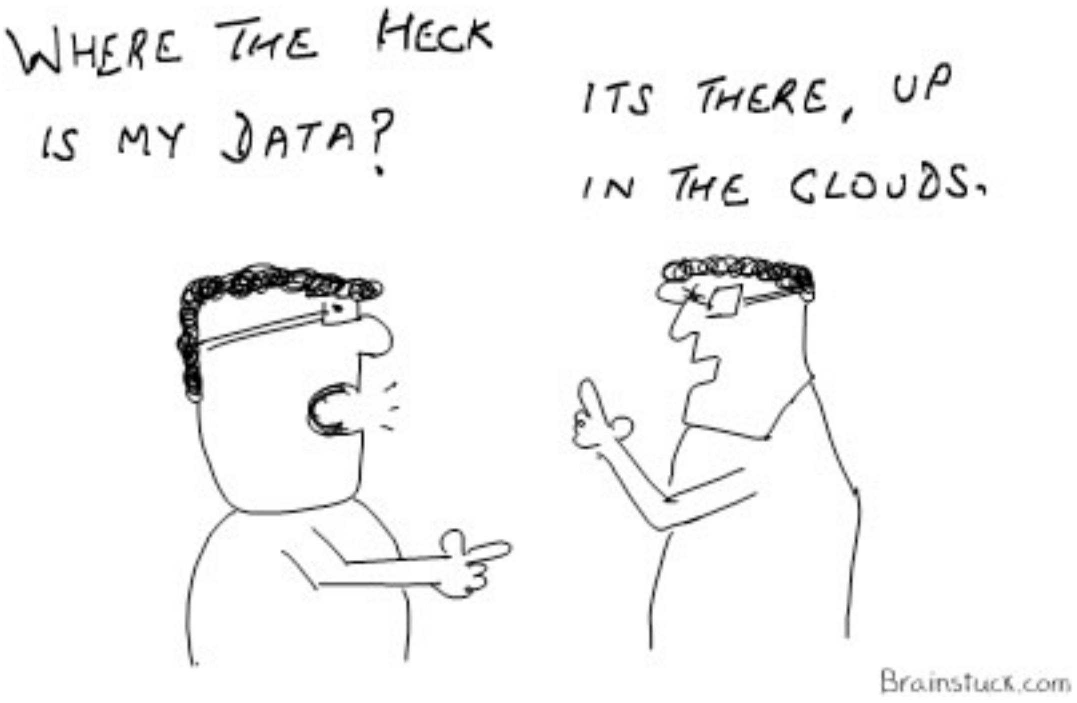
Jeff Chase

Duke University

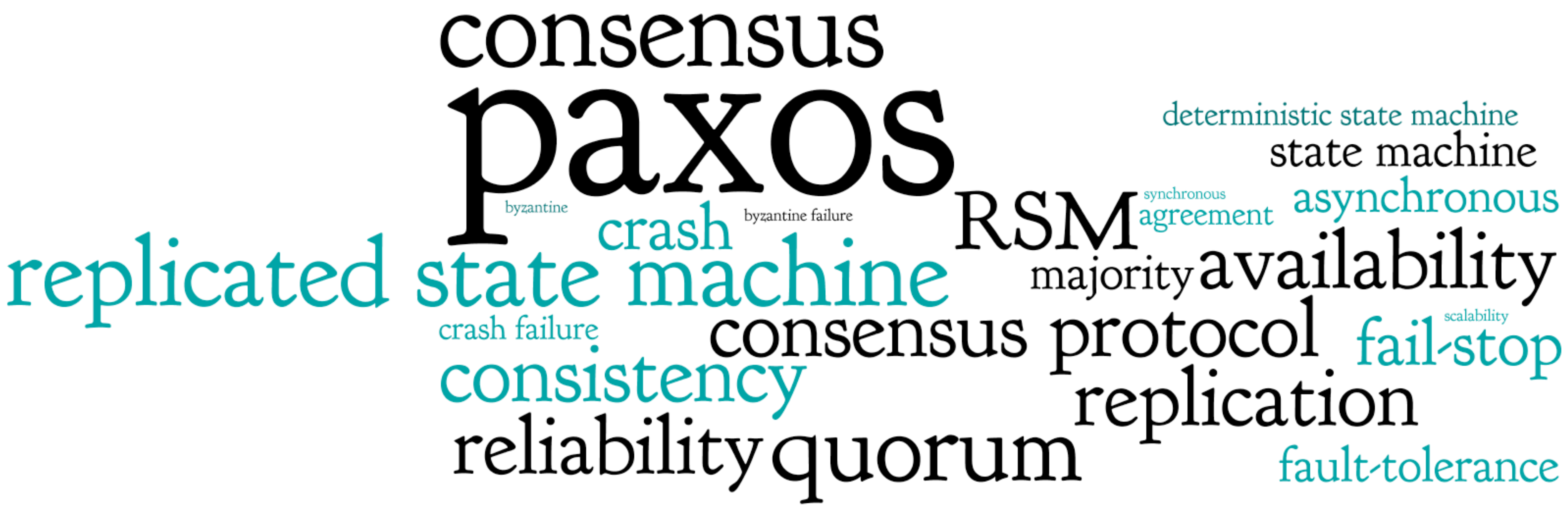
What is a distributed system?


Leslie Lamport

"A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable."   -- Leslie Lamport

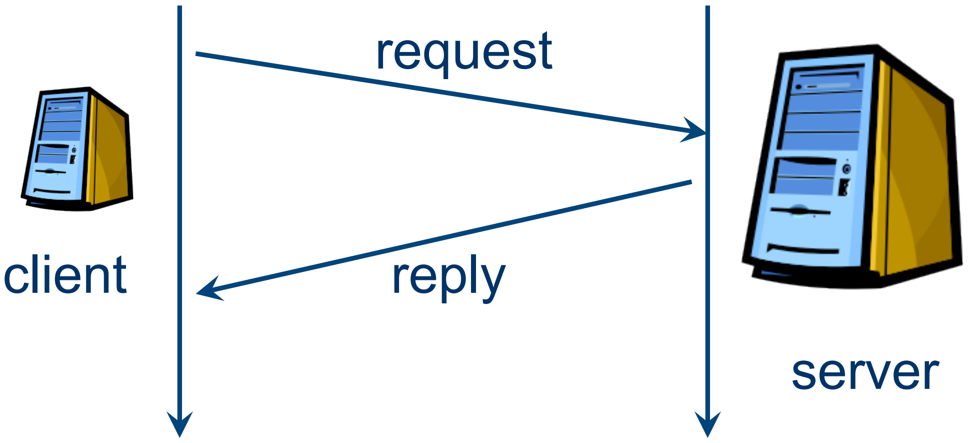


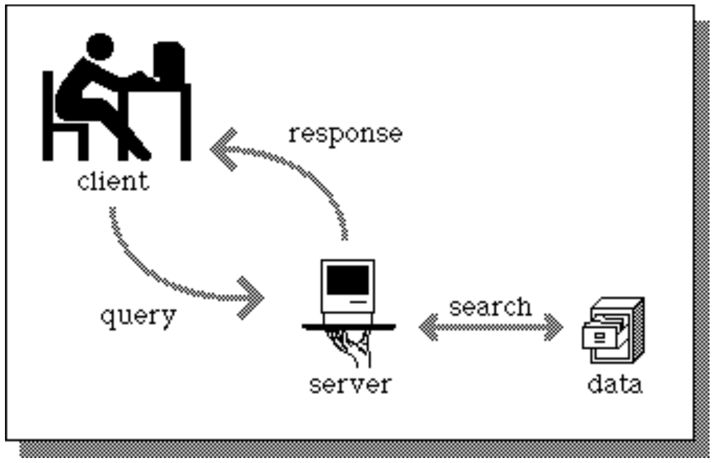
Just a peek, and a project (p3)

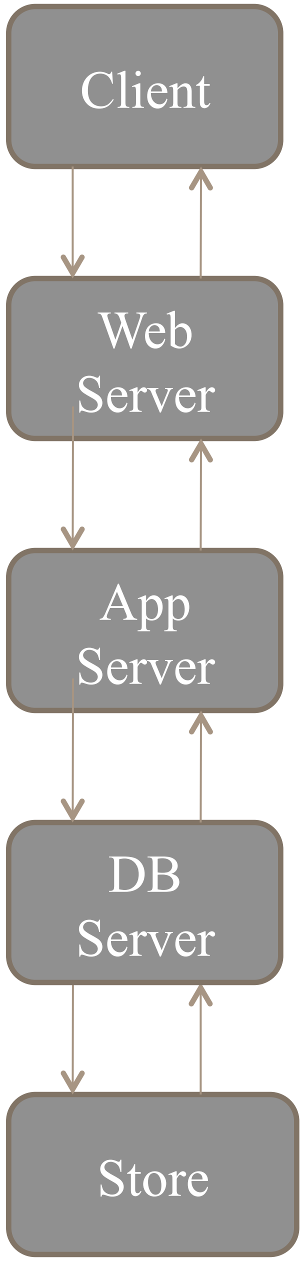


From http://**paxos.systems**

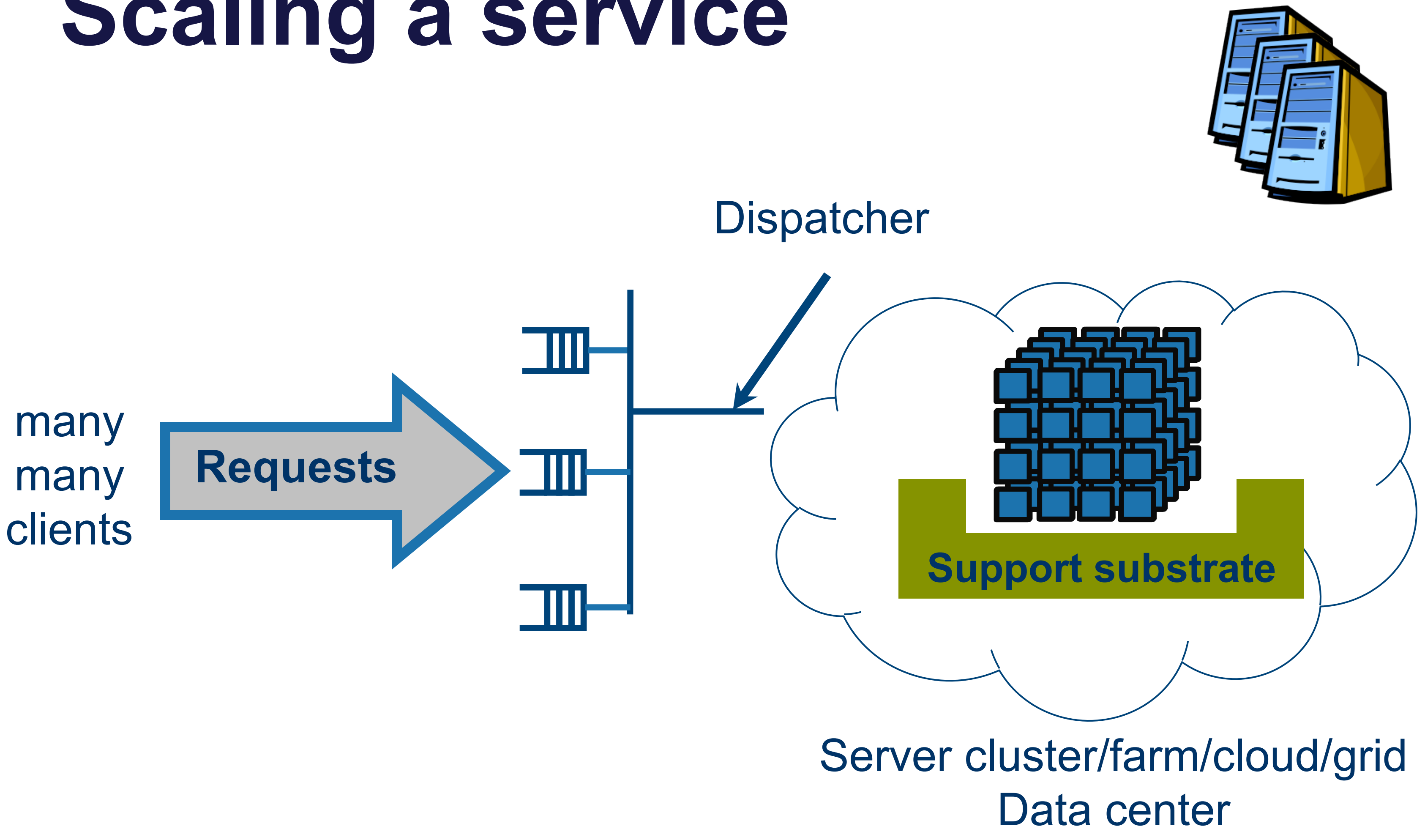
A service


request
client
reply
server


response
client
query
server
search
data


Client
Web Server
App Server
DB Server
Store

Scaling a service



Add interchangeable server “bricks” to **partition** (“shard”) and/or **replicate** service functionality for scale and robustness. Issues: state storage, server selection, request routing, etc.

What about failures?

- **Systems fail**. Here's a reasonable set of assumptions about failure properties for servers/bricks (or disks)
  - **Fail-stop** or **fail-fast** fault model
  - Nodes either function correctly or remain silent
  - A failed node may restart, or not
  - A restarted node loses its memory state, and recovers its secondary (disk) state
- If failures are random/independent, the probability of **some** failure is linear with the number of units.
  - Higher scale → less reliable!

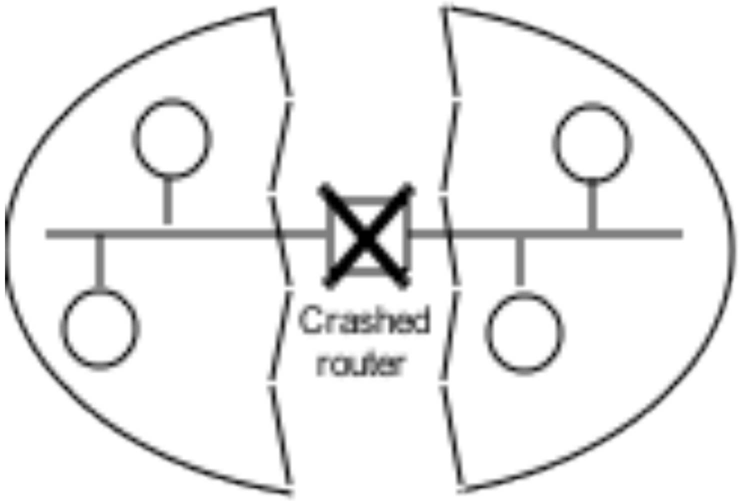
The image cannot be displayed. Your computer may not have en

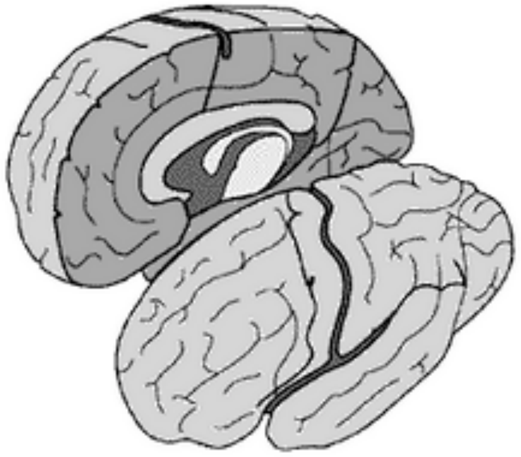
Nine

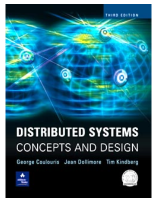
ave been corrupted. Restart your computer, and then open the file again. If the red x still appears, you may have to delete the image and then insert it again.

9. “Failures are independent.” - Chase

The problem of network partitions







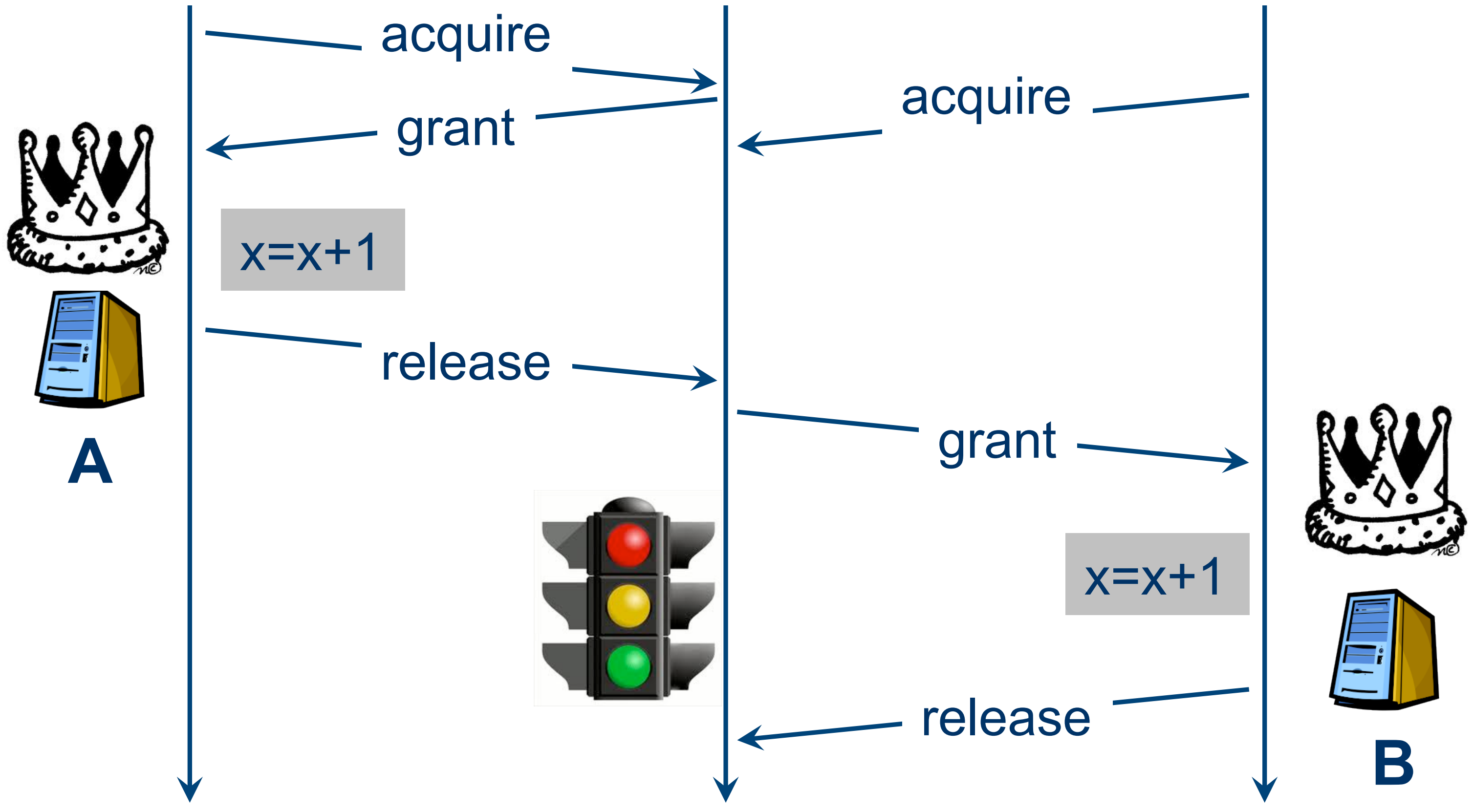
A network partition is any event that blocks all message traffic between some subsets of nodes.

Partitions cause "split brain syndrome": part of the system can't know what the other is doing.

Distributed mutual exclusion

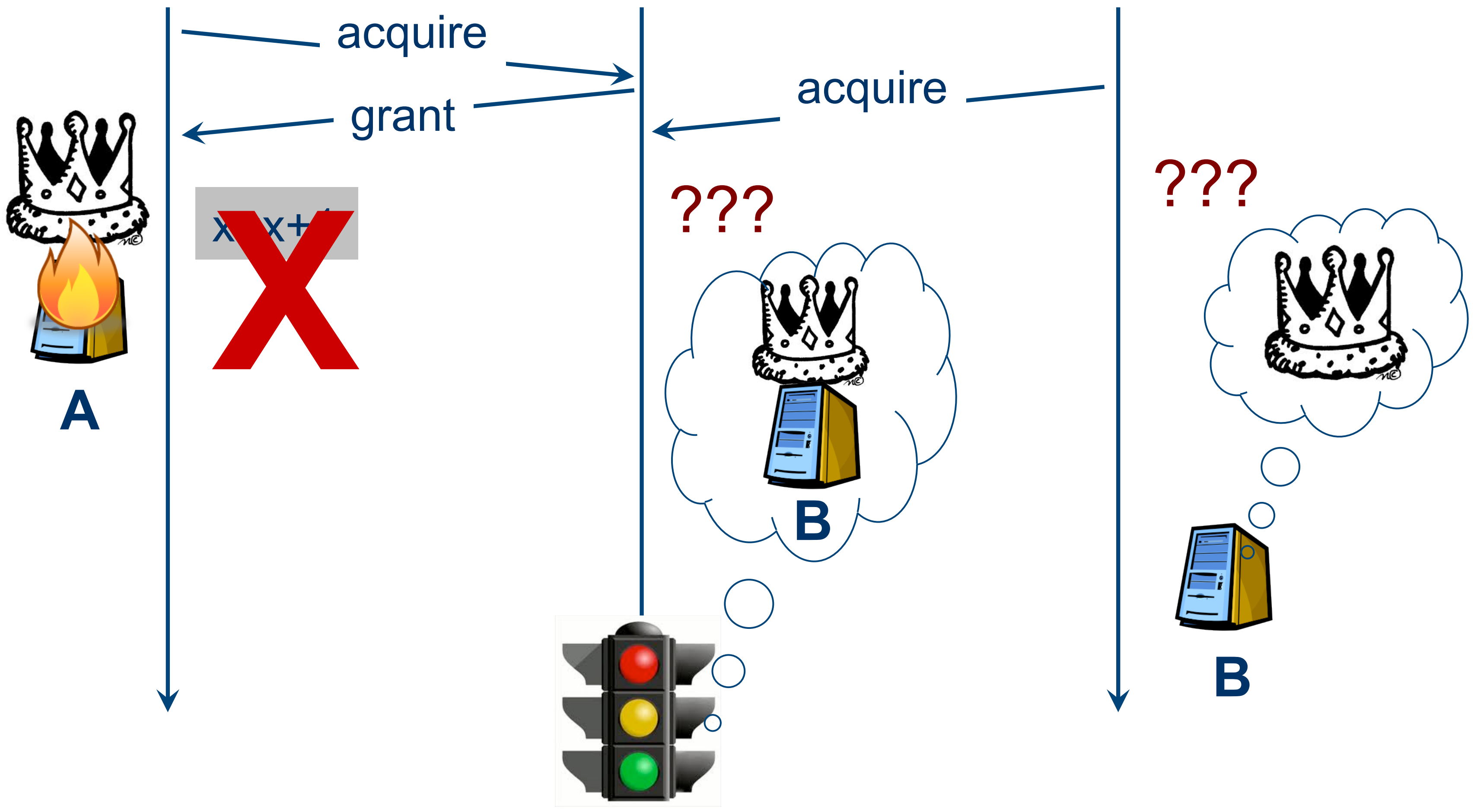
- It is often necessary to grant some node/process the "right" to "own" some given data or function.
- Ownership rights often must be mutually exclusive.
  - At most one owner at any given time.
- How to coordinate ownership?

One solution: lock service



lock service

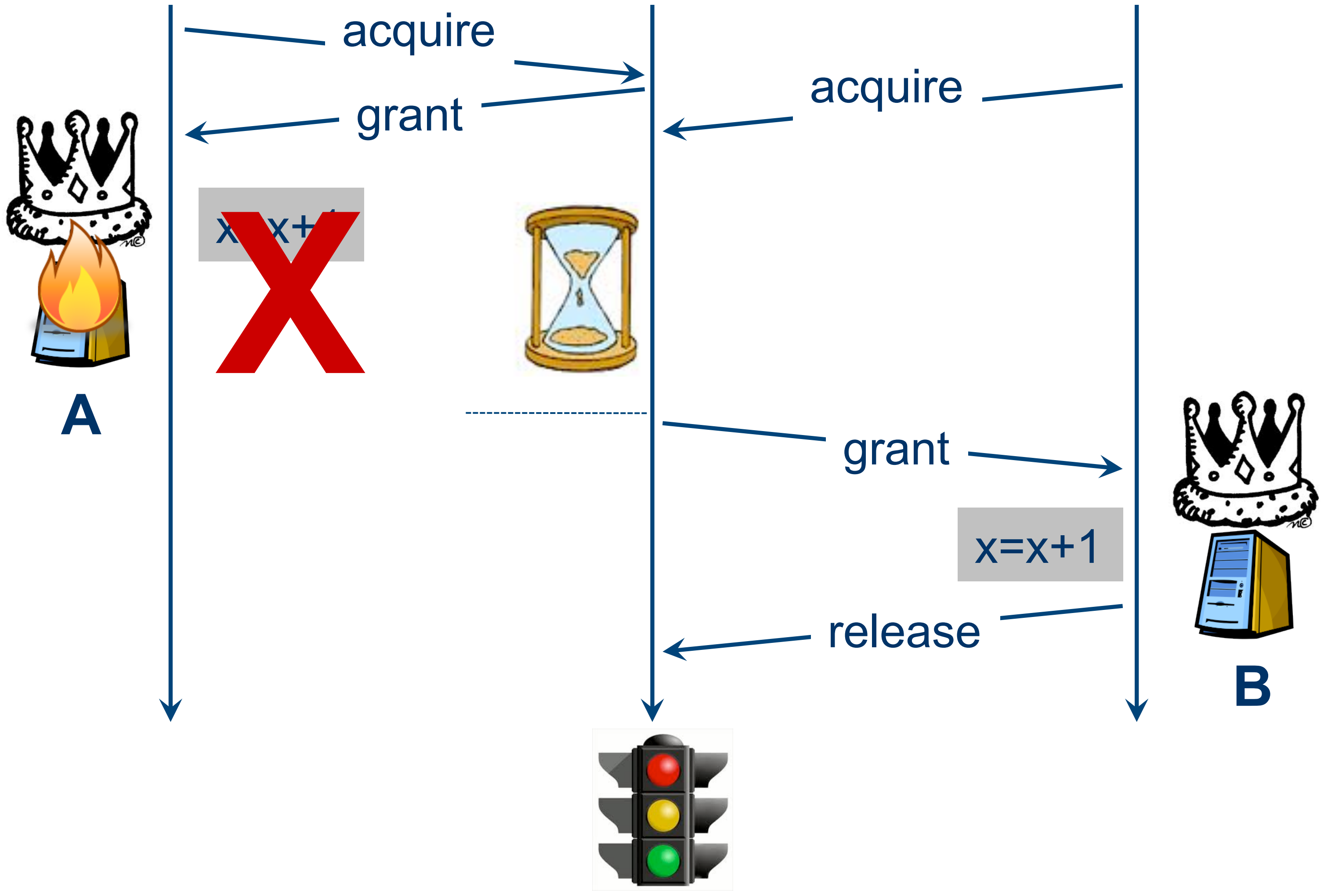
A lock service in the real world



Solution: leases (leased locks)

- A lease is a grant of ownership or control for a limited time.
- The owner/holder can renew or extend the lease.
- If the owner fails, the lease expires and is free again.
- The lease might end early.
  - lock service may recall or evict
  - holder may release or relinquish

A lease service in the real world

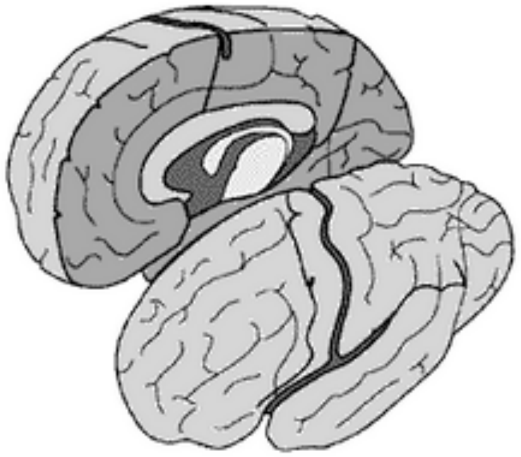


Leases and time

- The lease holder and lease service must agree when a lease has expired.
  - i.e., that its expiration time is in the past
  - Even if they can't communicate!
- We all have our clocks, but do they agree?
  - synchronized clocks
- For leases, it is sufficient for the clocks to have a known bound on clock drift.
  - $|T(C_i) - T(C_j)| < \varepsilon$
  - Build in slack time $> \varepsilon$ into the lease protocols as a safety margin.

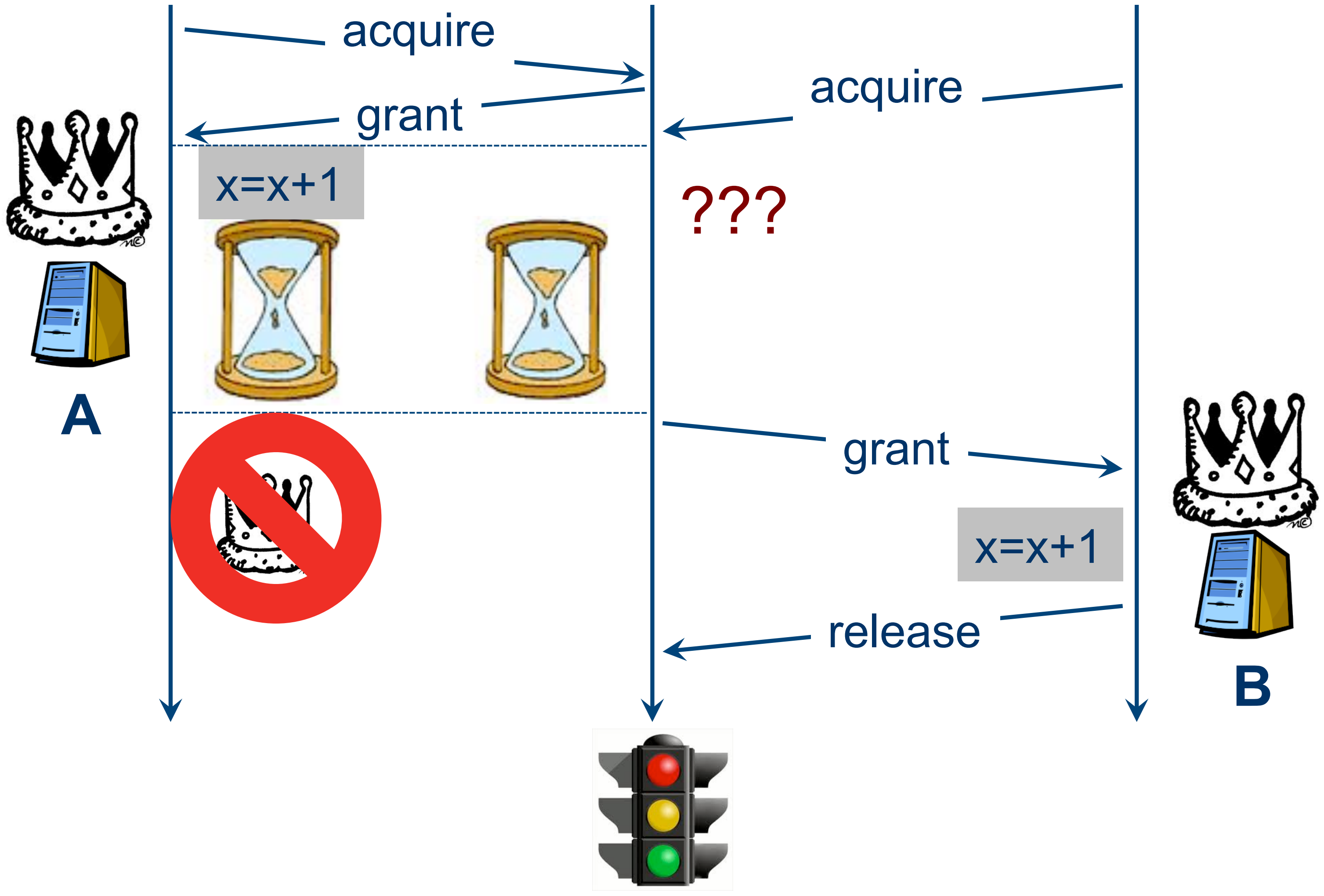
OK, fine, but…

- **What if the A does not fail, but is instead isolated by a network partition?**



This condition is often called a “split brain” problem: literally, one part of the system cannot know what the other part is doing, or even if it’s up.

Never two kings at once



OK, fine, but…

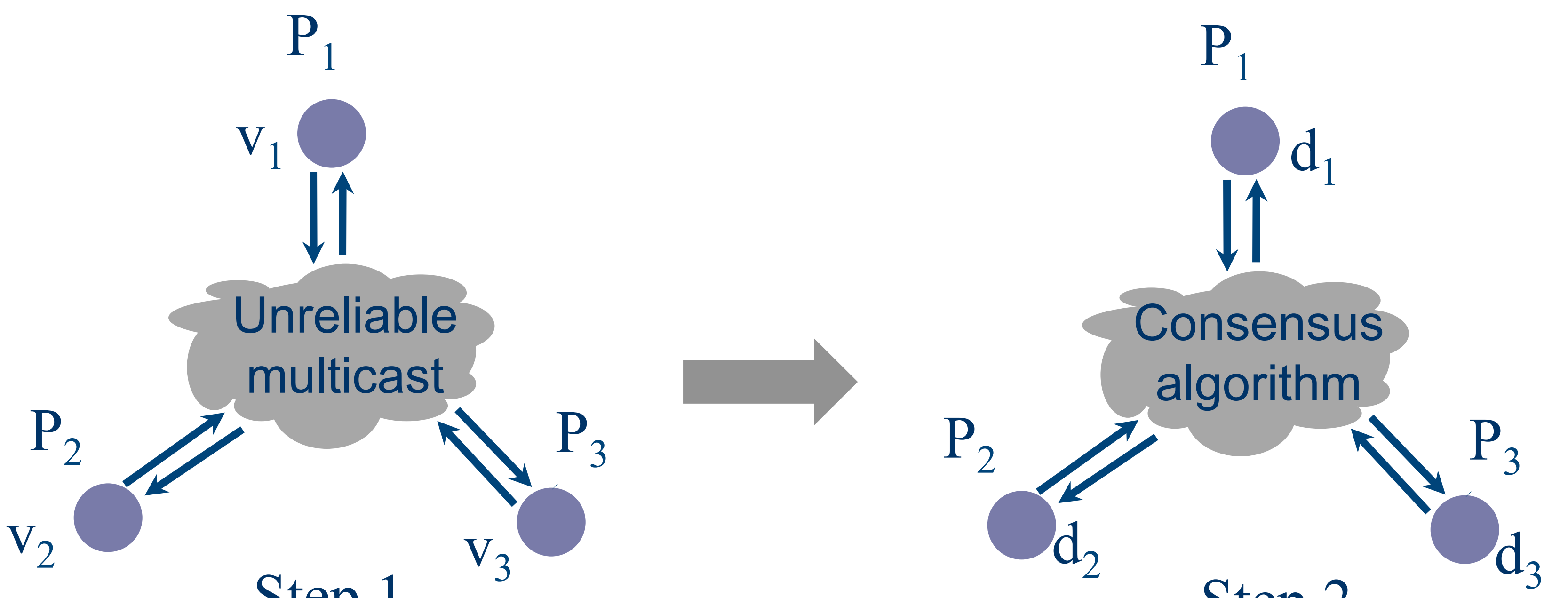
- **What if the manager/master itself fails?**

We can replace it, but the nodes must agree on who the new master is: requires **consensus**.

The Answer

- Replicate the functions of the manager/master.
  - Or other coordination service…
- Designate one of the replicas as a *primary*.
  - Or *master*
- The other replicas are backup servers.
  - Or standby or secondary
- If the primary fails, use a high-powered consensus algorithm to designate and initialize a new primary.

Consensus: abstraction



<u>Step 1</u>

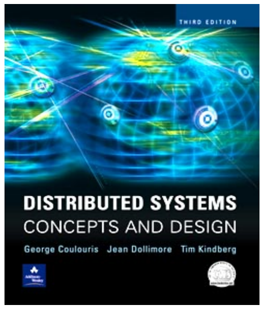
Propose.

Each P proposes a value to the others.

<u>Step 2</u>

Decide.

All nonfaulty P agree on a value in a bounded time.



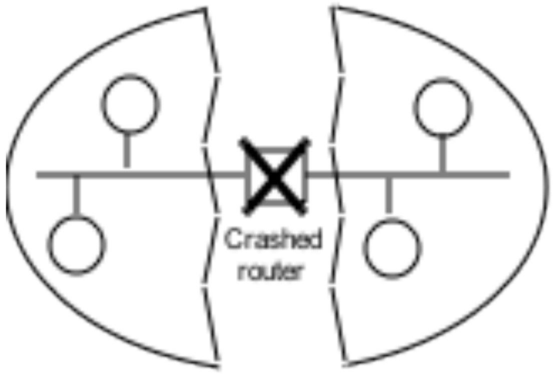
Coulouris and Dollimore

Coordination and Consensus

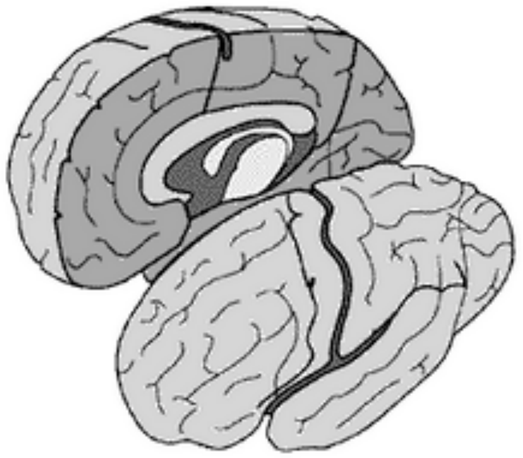
- **The key to availability and scalability is to decentralize and replicate functions and data.**
- **But how to coordinate the nodes?**
  - data consistency
  - update propagation
  - mutual exclusion
  - consistent global states
  - failure notification
  - group membership (views)
  - group communication
  - event delivery and ordering
- **All of these are consensus problems.**

Fischer-Lynch-Patterson (1985)

- No consensus can be guaranteed in an asynchronous system in the presence of failures.
- Intuition: a "failed" process may just be slow, and can rise from the dead at exactly the wrong time.
- Consensus may occur recognizably, rarely or often.



Network partition



Split brain

acmqueue The Network is Reliable

An informal survey of real-world communications failures

Peter Bailis, UC Berkeley
Kyle Kingsbury, Jepsen Networks

The celebrated FLP impossibility result demonstrates the inability to guarantee consensus in an asynchronous network (i.e., one facing indefinite communication partitions between processes) with one faulty process. This means that, in the presence of unreliable (untimely) message delivery, basic operations such as modifying the set of machines in a cluster (i.e., maintaining group membership, as systems such as Zookeeper are tasked with today) are not guaranteed to complete in the event of both network asynchrony and individual server failures.

…

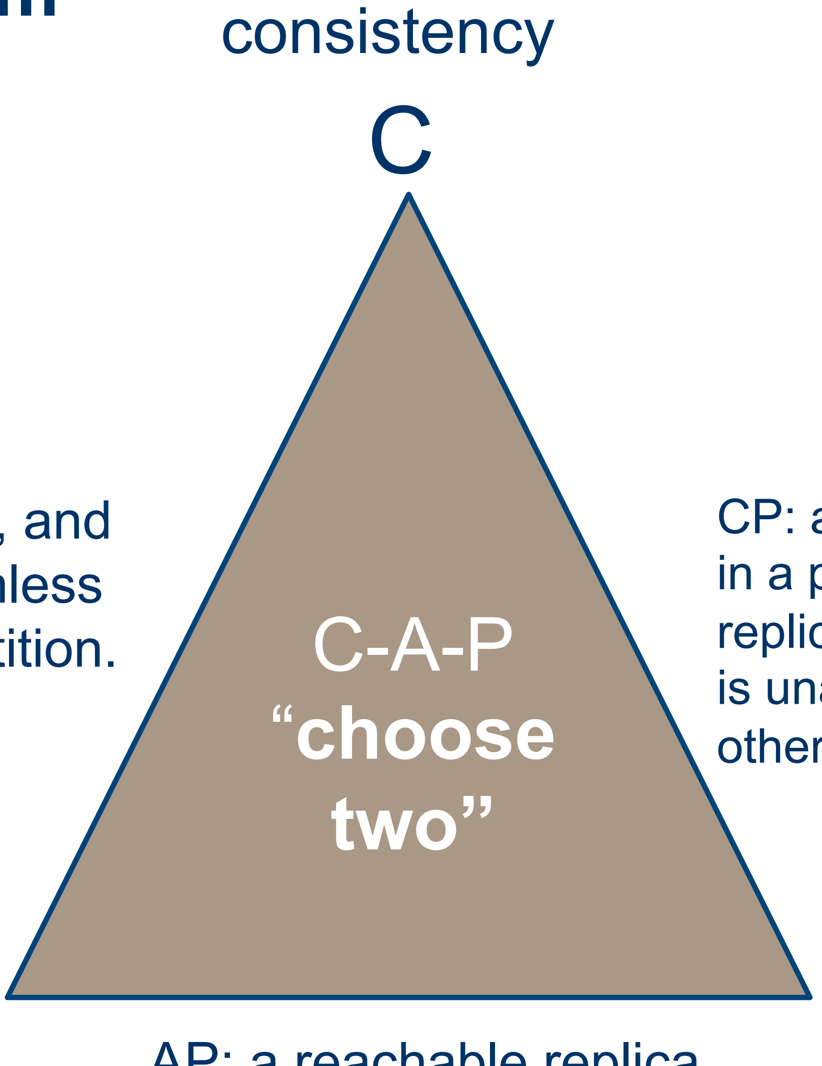
Therefore, the degree of reliability in deployment environments is critical in robust systems design and directly determines the kinds of operations that systems can reliably perform without waiting. Unfortunately, the degree to which networks are actually reliable in the real world is the subject of considerable and evolving debate.

…

CONCLUSIONS: WHERE DO WE GO FROM HERE?

This article is meant as a reference point—to illustrate that, according to a wide range of (often informal) accounts, communication failures occur in many real-world environments. Processes, servers, NICs, switches, and local and wide area networks can all fail, with real economic consequences. Network outages can suddenly occur in systems that have been stable for months at a time, during routine upgrades, or as a result of emergency maintenance. The consequences of these outages range from increased latency and temporary unavailability to inconsistency, corruption, and data loss. Split-brain is not an academic concern: it happens to all kinds of systems—sometimes for days on end. Partitions deserve serious consideration.

"CAP theorem"



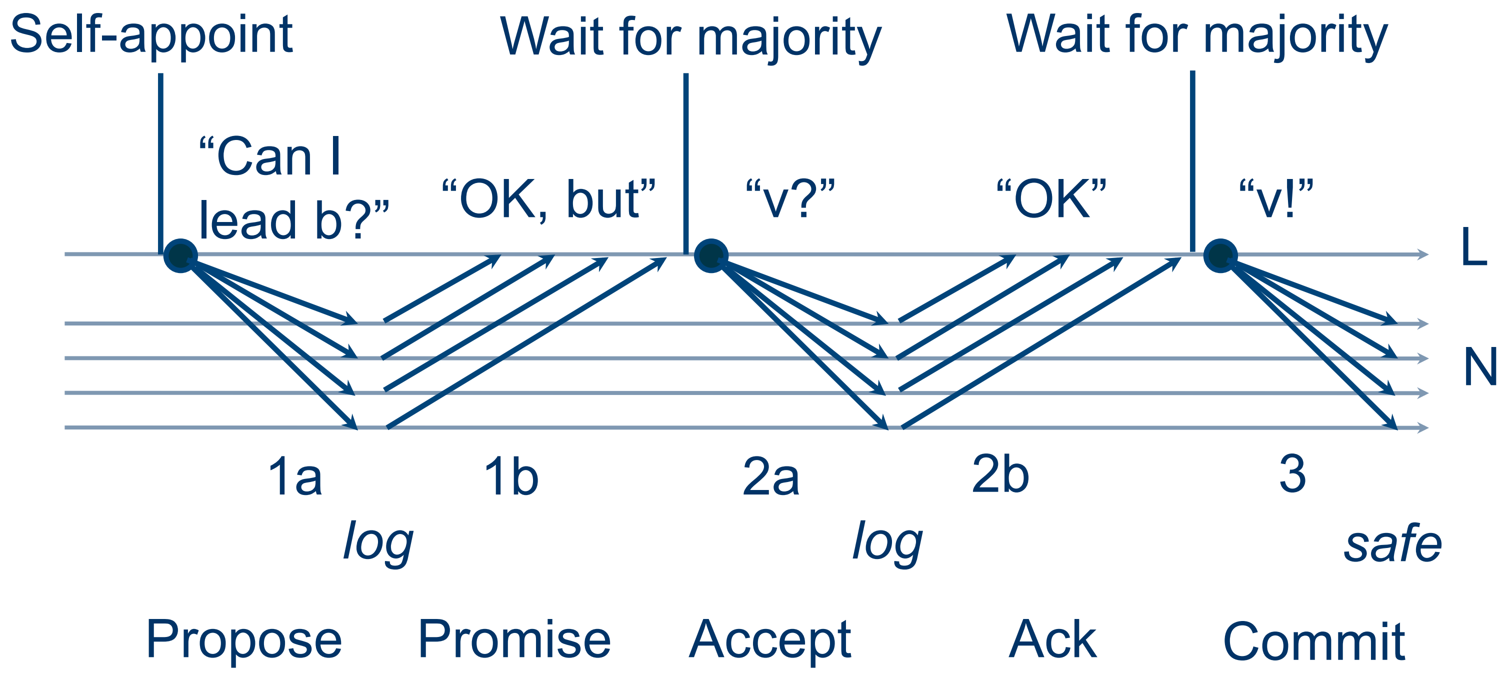
CA: available, and consistent, unless there is a partition.

CP: always consistent, even in a partition, but a reachable replica may deny service if it is unable to agree with the others (e.g., quorum).

AP: a reachable replica provides service even in a partition, but may be inconsistent.

Dr. Eric Brewer

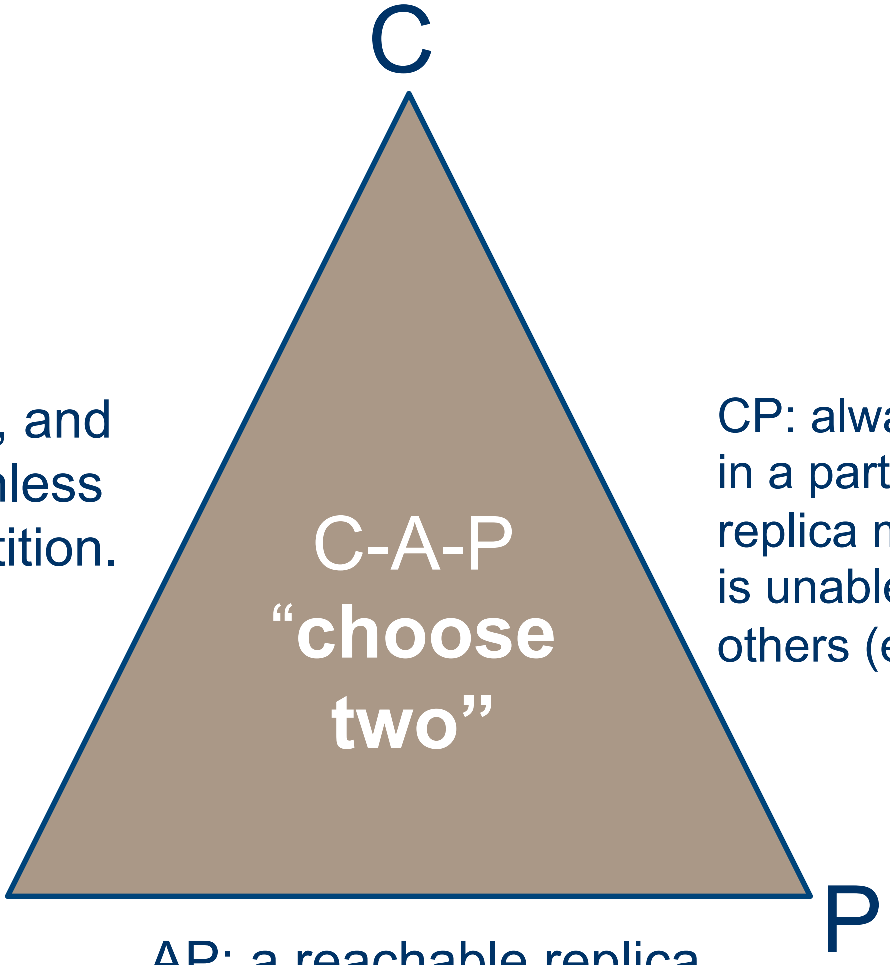
Paxos: voting among groups of nodes



You will see references to **Paxos state machine**: it refers to a group of nodes that cooperate using the Paxos algorithm to keep a system with replicated state safe and available (to the extent possible under prevailing conditions). We will discuss it later.

“CAP theorem”

Dr. Eric Brewer



CA: available, and consistent, unless there is a partition.

CP: always consistent, even in a partition, but a reachable replica may deny service if it is unable to agree with the others (e.g., quorum).

AP: a reachable replica provides service even in a partition, but may be inconsistent.

Properties for Correct Consensus

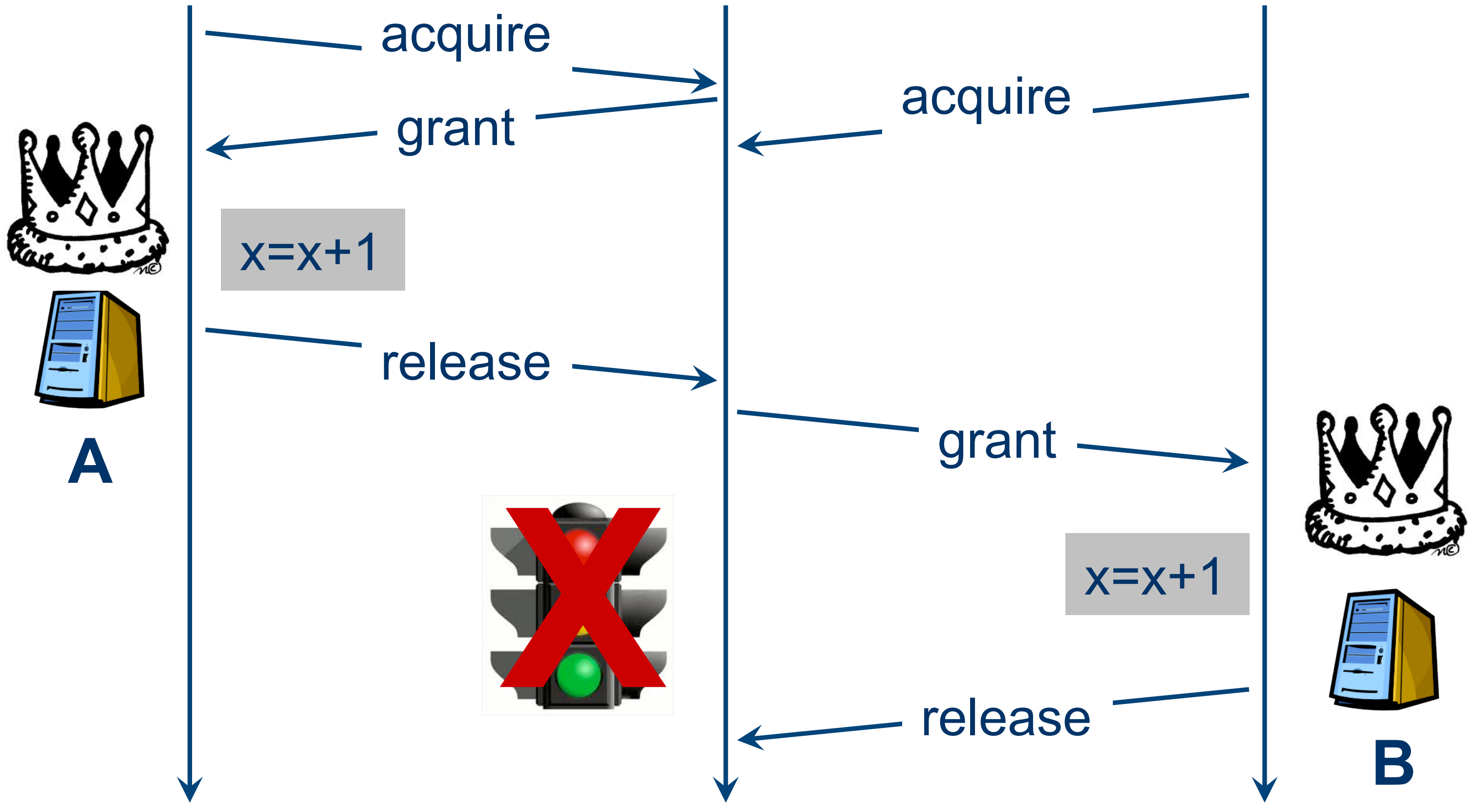
- Termination: All correct processes eventually decide.
- Agreement: All correct processes select the same d_j.
  - Or…(stronger) all processes that do decide select the same d_j, even if they later fail.

- Consensus "must be" both safe and live.
- FLP and CAP say that a consensus algorithm can be safe or live, but not both.

Now what?

- We must build practical, scalable, efficient distributed systems that **really work** in the real world.
- But the theory says it is **impossible** to build reliable computer systems from unreliable components.
- **So what are we to do?**

Recap: replicated lock service



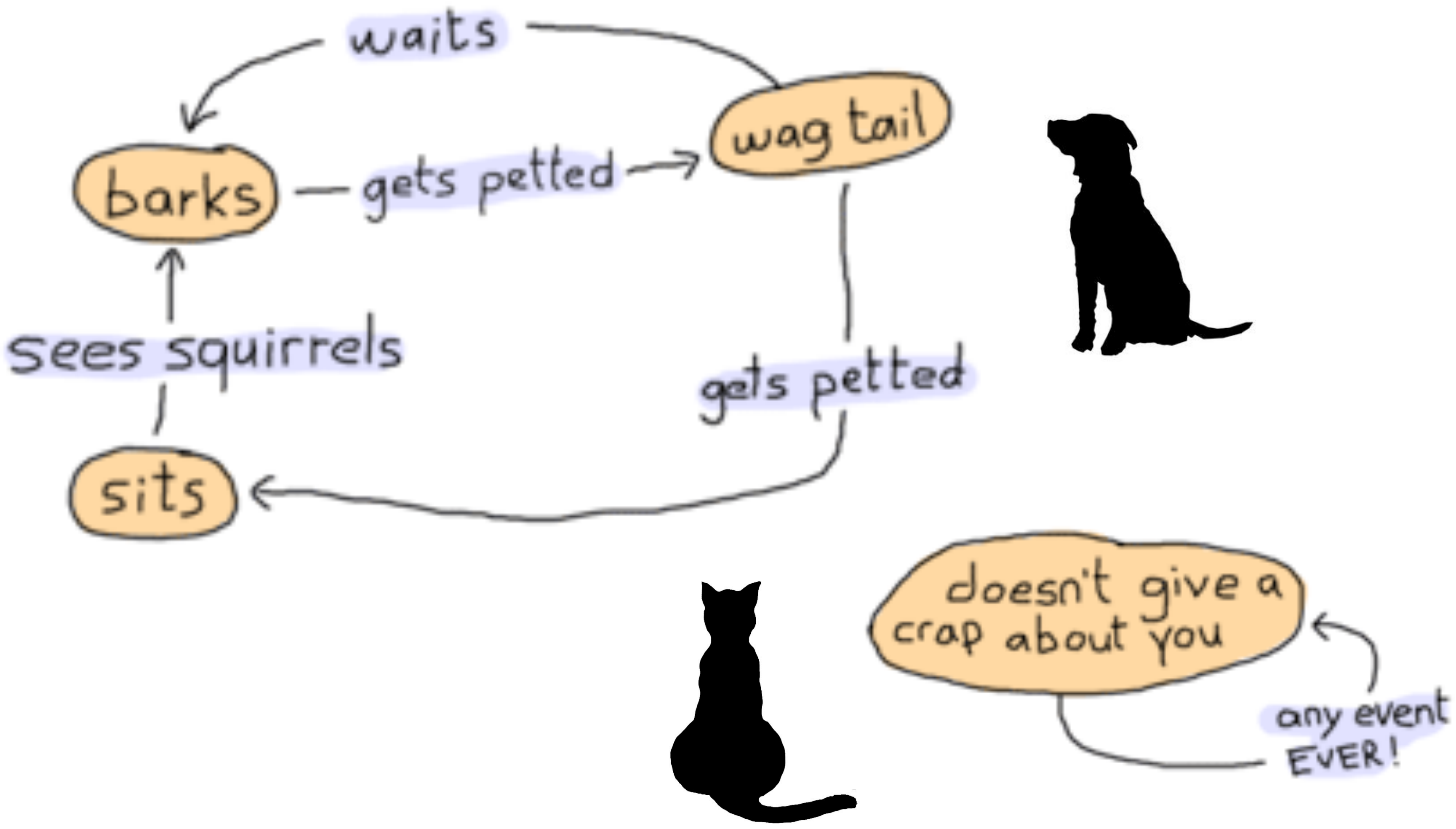
How to handle failure of the lock server? **Replicate it.**

Coordination services and consensus

- It is common to build cloud service apps around a **coordination service**.
  - Locking, failure detection, atomic/consistent update to small file-like objects in a consistent global name space.
- Fundamental building block for scalable services.
  - Chubby (Google)
  - **Zookeeper** (Yahoo! / Apache)
  - Centrifuge (Microsoft)
- They have the same consensus algorithm at their core (with minor variations): Paxos/VR/Raft
- For p3 we use Raft for **State Machine Replication**.

Finite State Machine (FSM)

Dogs and Cats



http://learnyousomeerlang.com/finite-state-machines

FSM Basics

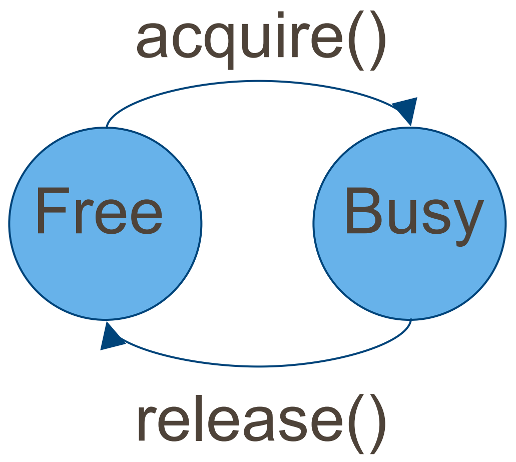
It’s a useful formal model for thinking about programs.

- Finite set of states and actions (inputs)
- Deterministic transition function: *F(state, input) → state*
- State determines behavior (e.g., also emit an output).

We can think of servers as FSMs.

- States: all possible combinations of internal data values.
- Inputs: client requests change data + generate output.
- F(state, input) → (state, output)

Lock server as an FSM



It’s an FSM, but we also have to represent waiters, and actions like waking up a waiter on a state transition.

An FSM for two locks

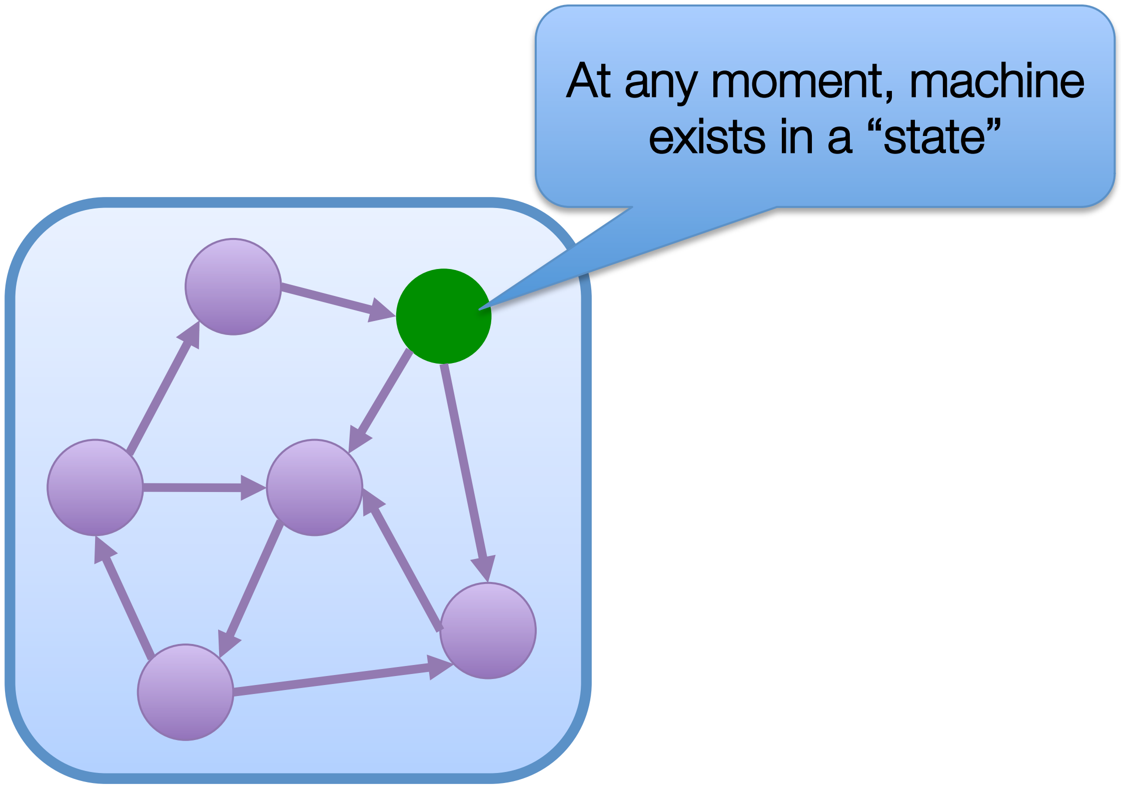
$$\begin{tikzcd}
1F2F \arrow[r, bend left, "A(1)"] \arrow[d, bend right=60, "A(2)"'] & 1B2F \arrow[l, bend left, "R(1)"] \arrow[d, shift left, "A(2)"] \\
1F2B \arrow[u, shift left, "R(2)"] \arrow[r, bend left, "A(1)"] & 1B2B \arrow[l, bend left, "R(1)"] \arrow[u, bend right=60, "R(2)"']
\end{tikzcd}$$

It gets complicated fast. But the point is that, formally, a lock server with N locks may be viewed as an FSM.

Why bother with FSMs?

- We can represent any server as an FSM – theoretically.
- The point is that if we are going to replicate servers, we want all the replicas to give the same responses.
- And that just means that they must be in the same state.
- And that will happen if they receive **exactly** the same inputs (requests) in **exactly** the same order.
- That is what we mean when we say we view a set of server replicas as a **replicated state machine** (RSM).

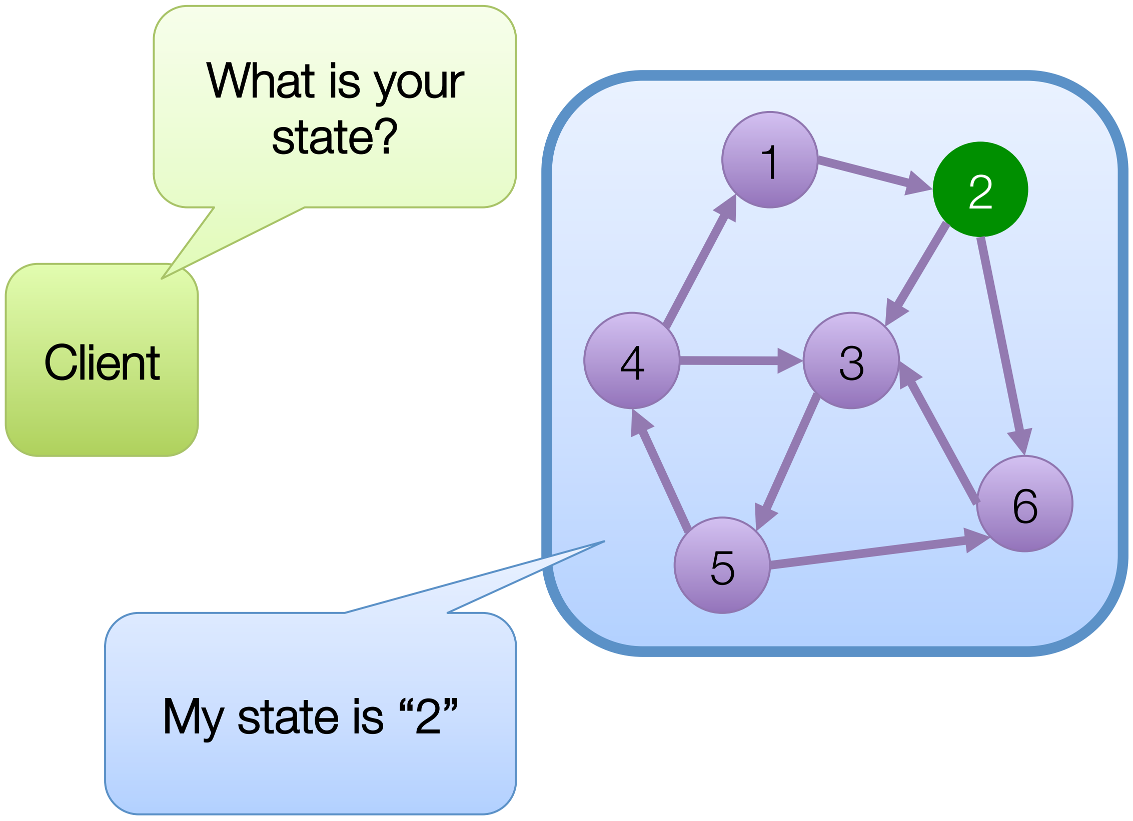
State machines



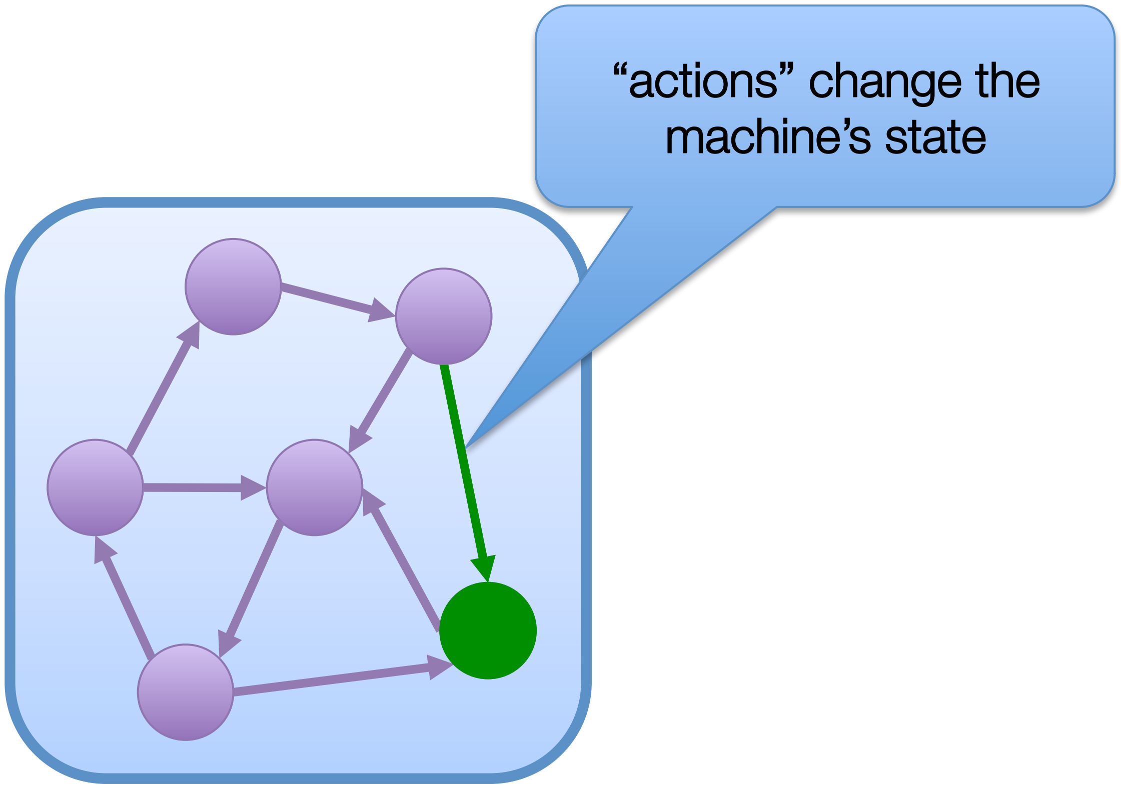
What is a state? Should think of as a set of named variables and their values

State machines

Clients can ask a machine about its current state.

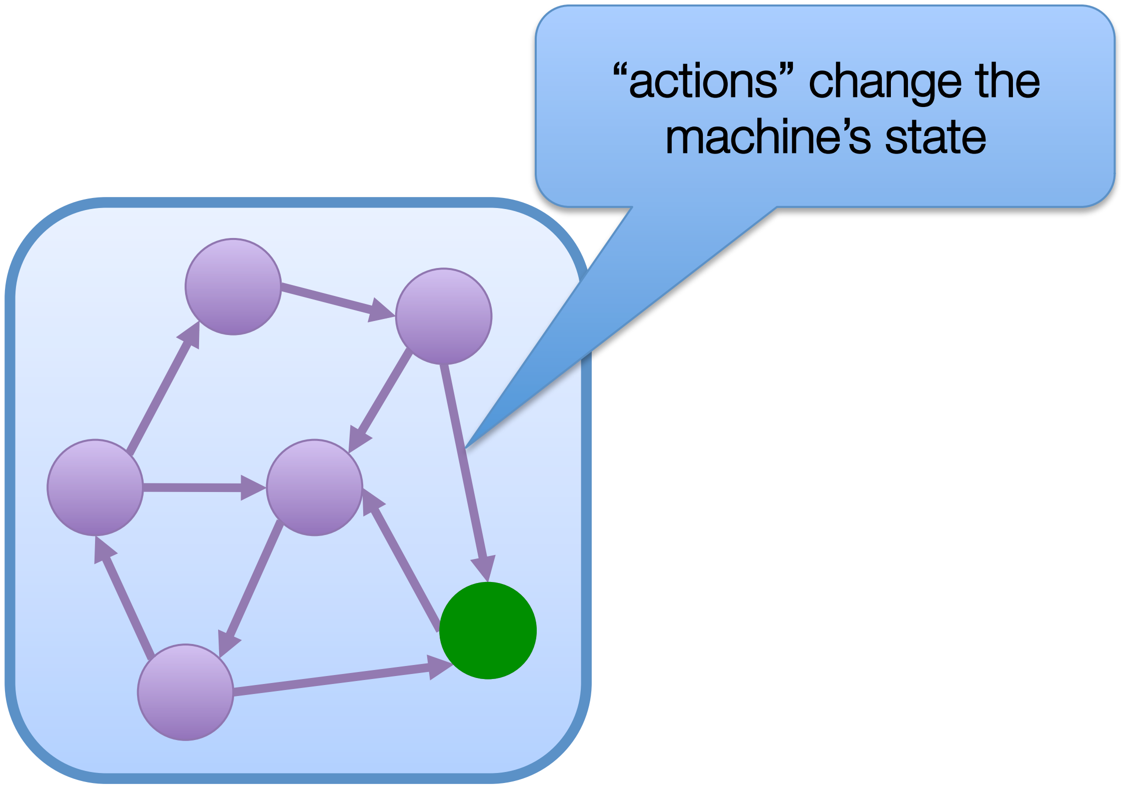


State machines



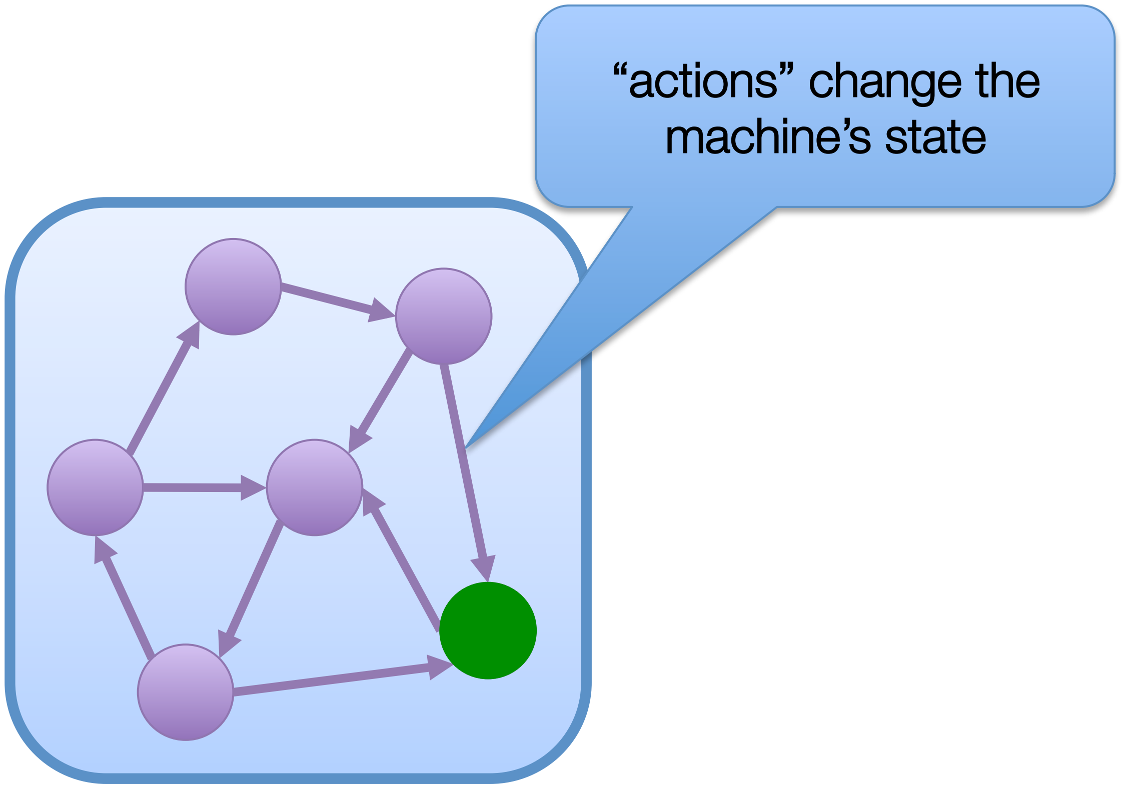
What is an action? Command that updates named variables' values

State machines



Is an action’s effect deterministic? For our purposes, yes. Given a state and an action, we can determine next state w/ 100% certainty.

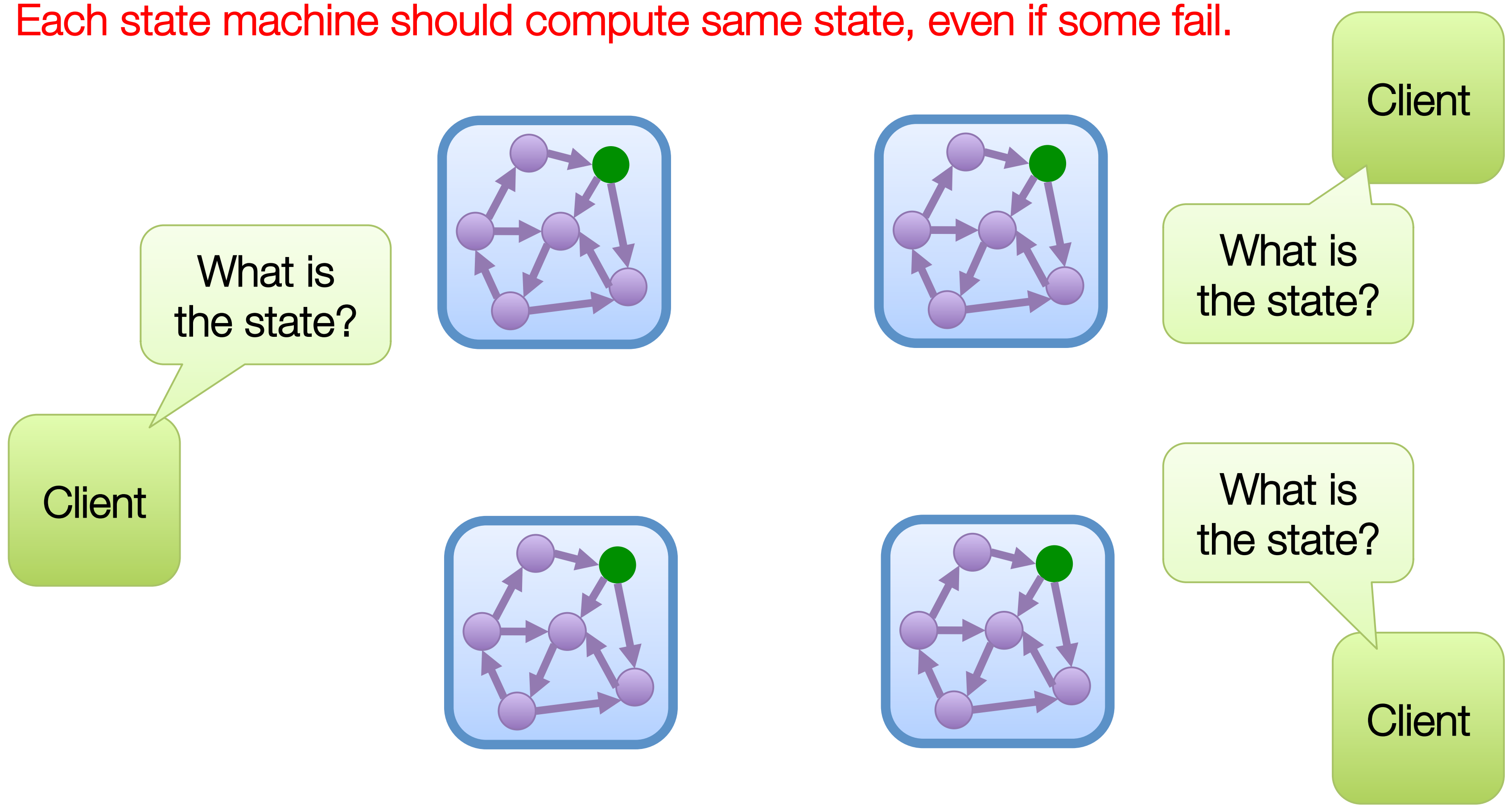
State machines



Is the effect of a sequence of actions deterministic? Yes, given a state and a sequence of actions, can be 100% certain of end state

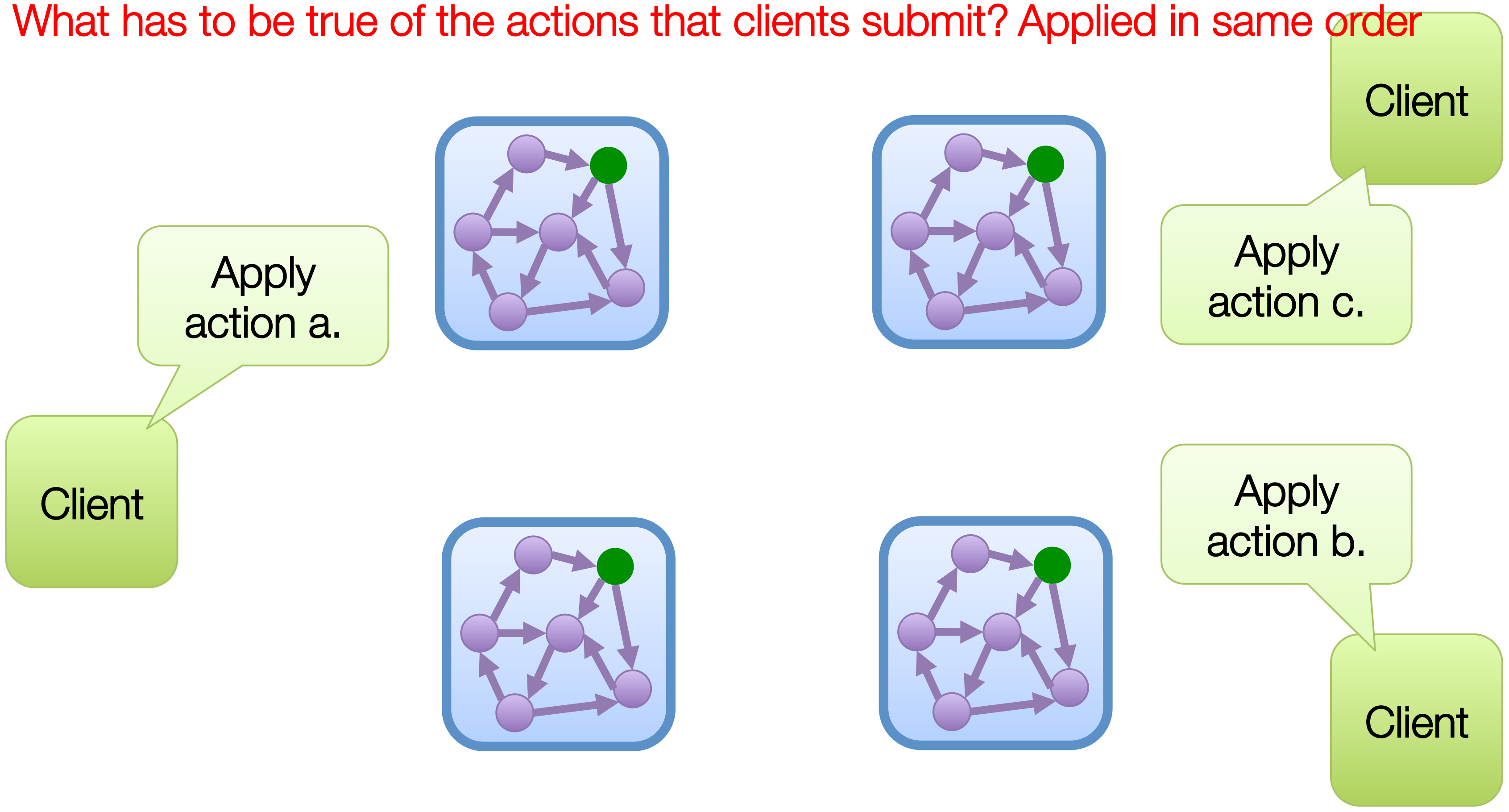
Replicated state machines

Each state machine should compute same state, even if some fail.



Replicated state machines

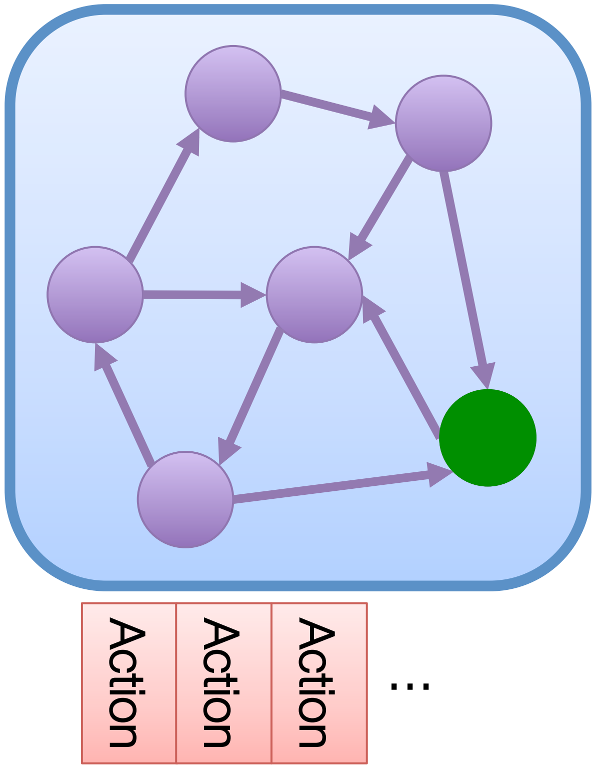
What has to be true of the actions that clients submit? Applied in same order



State machines

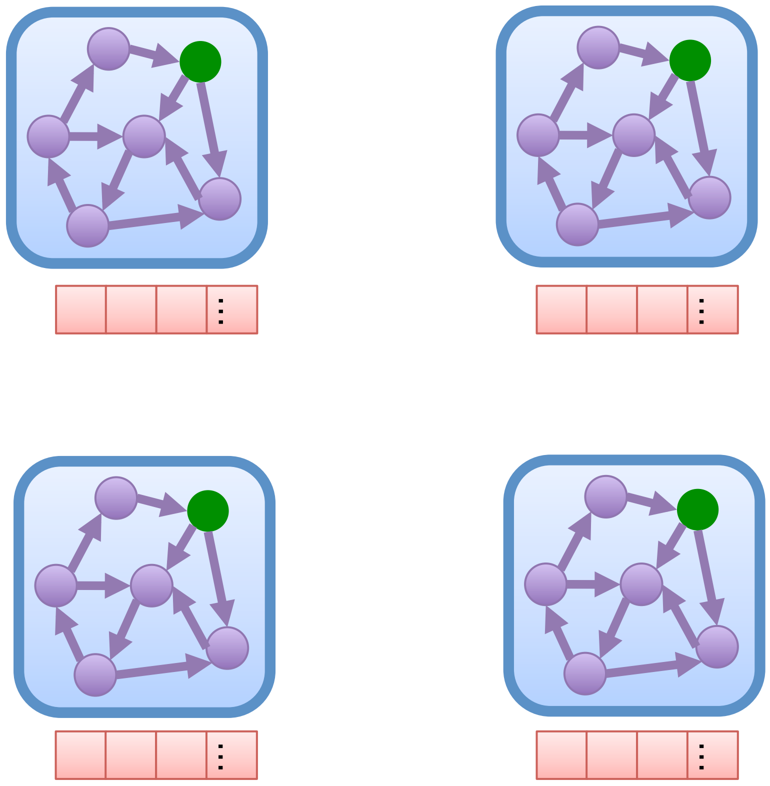
How should a machine make sure it applies action in same order across reboots?

Store them in a log!



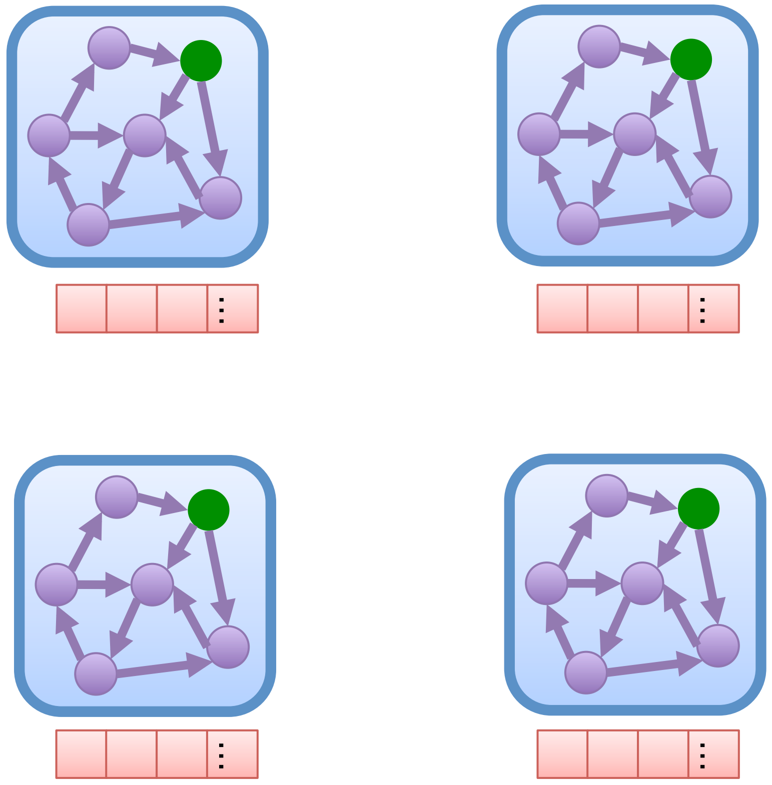
Replicated state machines

Can reduce problem of consistent, replicated states to consistent, replicated logs



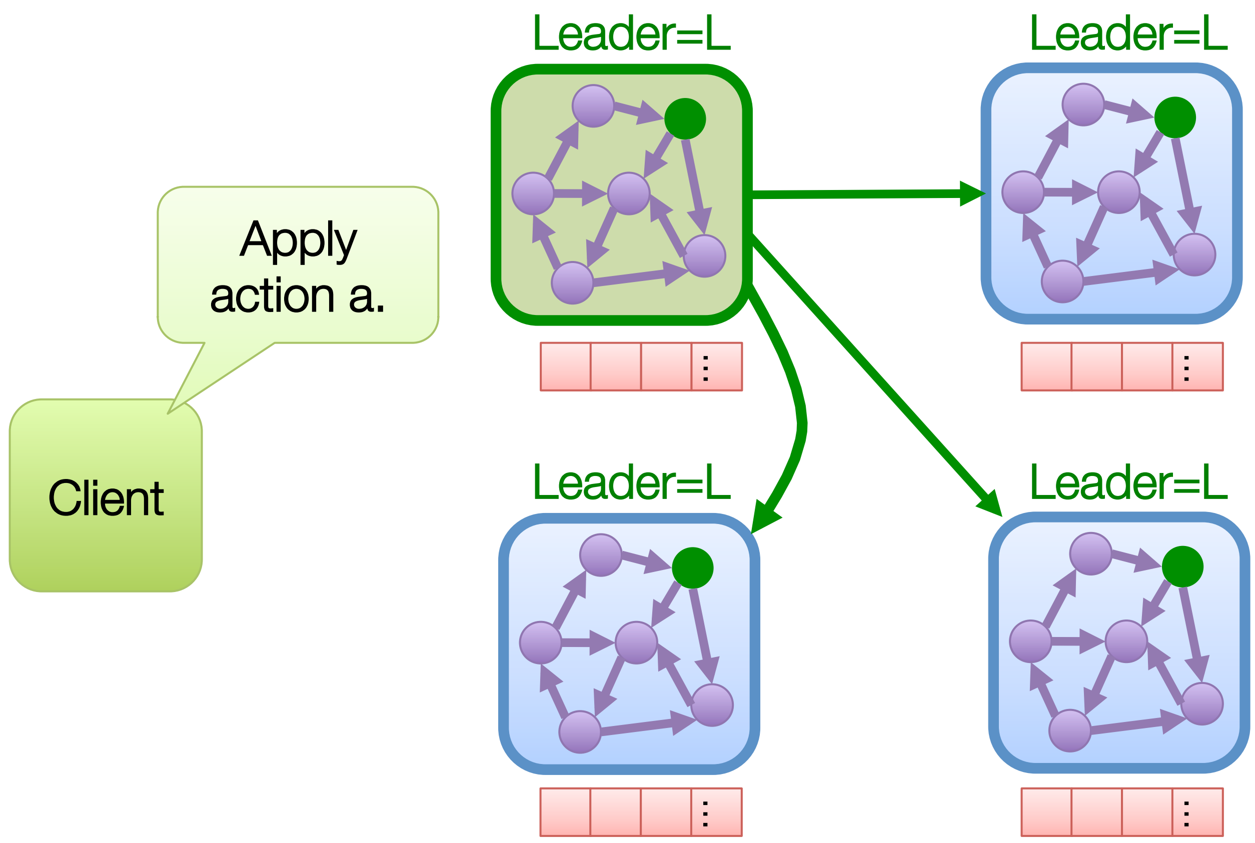
Replicated state machines

How to make sure that logs are consistent? Two-phase commit? …



Replicated state machines

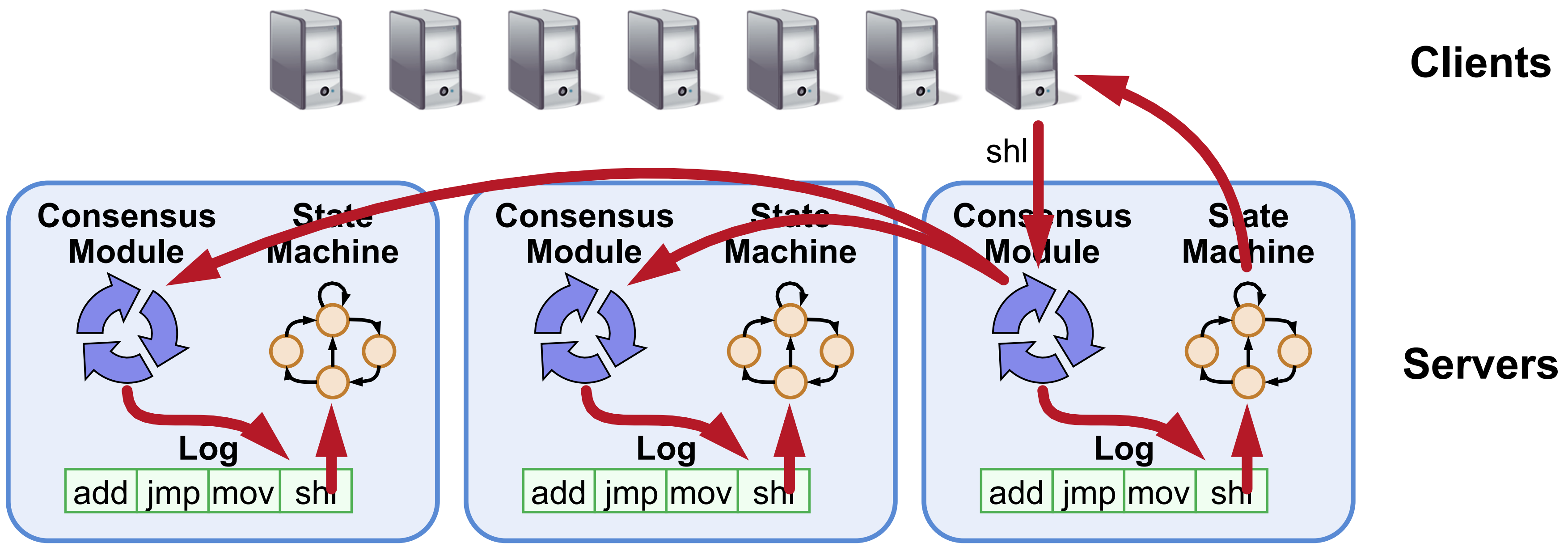
What is the heart of the matter? Have to agree on the leader, outside of the logs.



RSM and consensus

- In this setting, consensus means that all replicas agree on a sequence (or **log**) of **actions** (requests, inputs, ops).
- This **strong ordering condition** is **necessary** to ensure that all replicas converge (in the general case).
  - In more specific cases, we might be able to relax it. E.g., for a file service that executes reads/writes on distinct files.
- And it is also a **sufficient** condition for convergence.
  - Presuming the server program (“service code” or “server state machine”) is in fact deterministic.
- **So now we have a clear goal!**

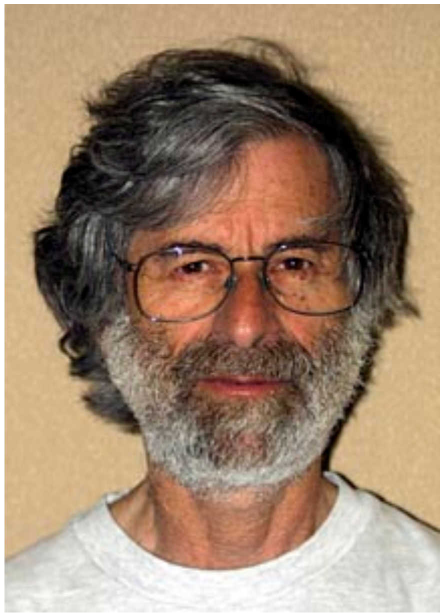
Goal: Replicated Log

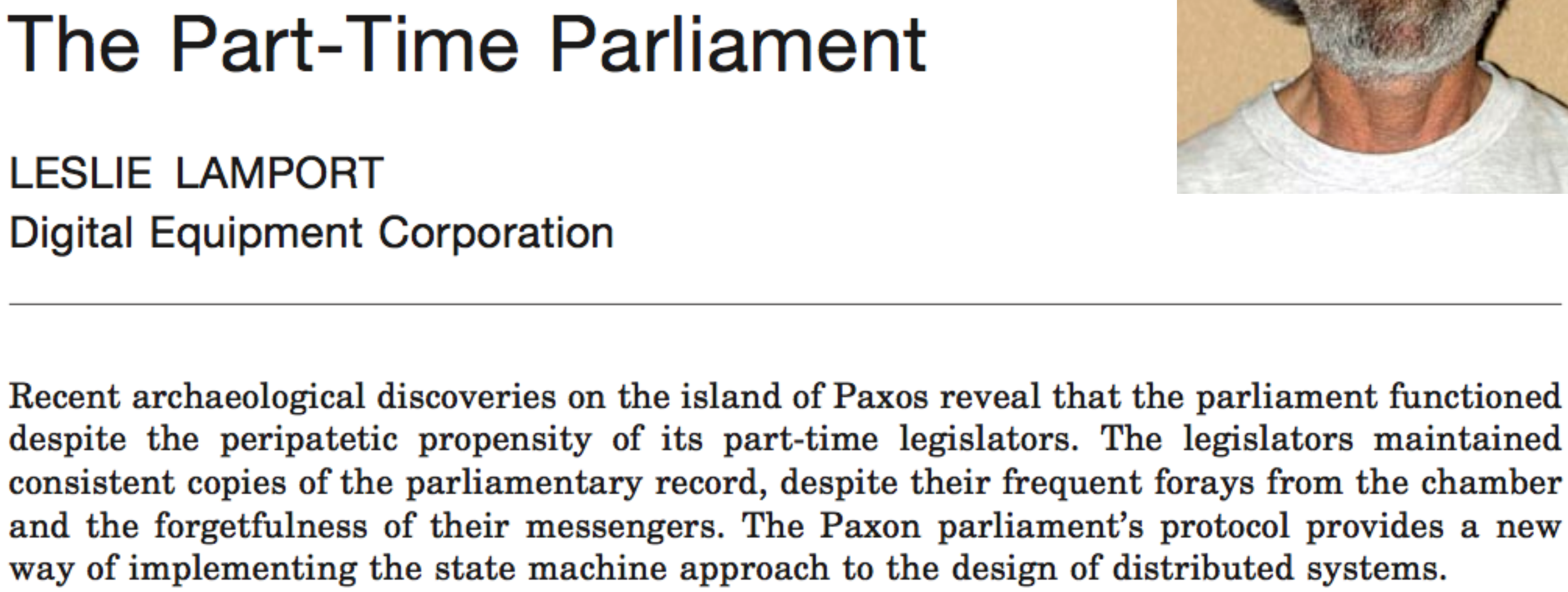


- **Replicated log => replicated state machine**
  - All servers execute same commands in same order
- **Consensus module ensures proper log replication**
- **System makes progress as long as any majority of servers are up**
- **Failure model: fail-stop (not Byzantine), delayed/lost messages**

Consensus: the classic paper

- **ACM TOCS:**
  - Transactions on Computer Systems
- **Submitted: 1990. Accepted: 1998**
- **Introduced:** *Γωυδα is the new cheese inspector*





The Part-Time Parliament

LESLIE LAMPORT
Digital Equipment Corporation

Recent archaeological discoveries on the island of Paxos reveal that the parliament functioned despite the peripatetic propensity of its part-time legislators. The legislators maintained consistent copies of the parliamentary record, despite their frequent forays from the chamber and the forgetfulness of their messengers. The Paxon parliament's protocol provides a new way of implementing the state machine approach to the design of distributed systems.

1. THE PROBLEM

1.1 The Island of Paxos

Early in this millennium, the Aegean island of Paxos was a thriving mercantile center.[1] Wealth led to political sophistication, and the Paxons replaced their ancient theocracy with a parliamentary form of government. But trade came before civic duty, and no one in Paxos was willing to devote his life to Parliament. The Paxon Parliament had to function even though legislators continually wandered in and out of the parliamentary Chamber.

The problem of governing with a part-time parliament bears a remarkable correspondence to the problem faced by today's fault-tolerant distributed systems, where legislators correspond to processes, and leaving the Chamber corresponds to failing. The Paxons' solution may therefore be of some interest to computer scientists. I present here a short history of the

Paxos beaches & resorts
Click on a location to explore some of the beaches & resorts of Paxos
Lakka
Harami
Monodendri
Loggos beach
Loggos
Marmari
Kipadi
Kaka Lagada
Gaios
Paxos
Key
beaches
resorts
Vrika
Voutoumi
Anti Paxos
For more information about exploring Paxos see the Top Trips page
You can also purchase maps, trails & DVDs from the Travel Shop

strict and was rejecting perfectly good cheese. Parliament then replaced him by passing the decree

1375: Γωυδα *is the new cheese inspector*

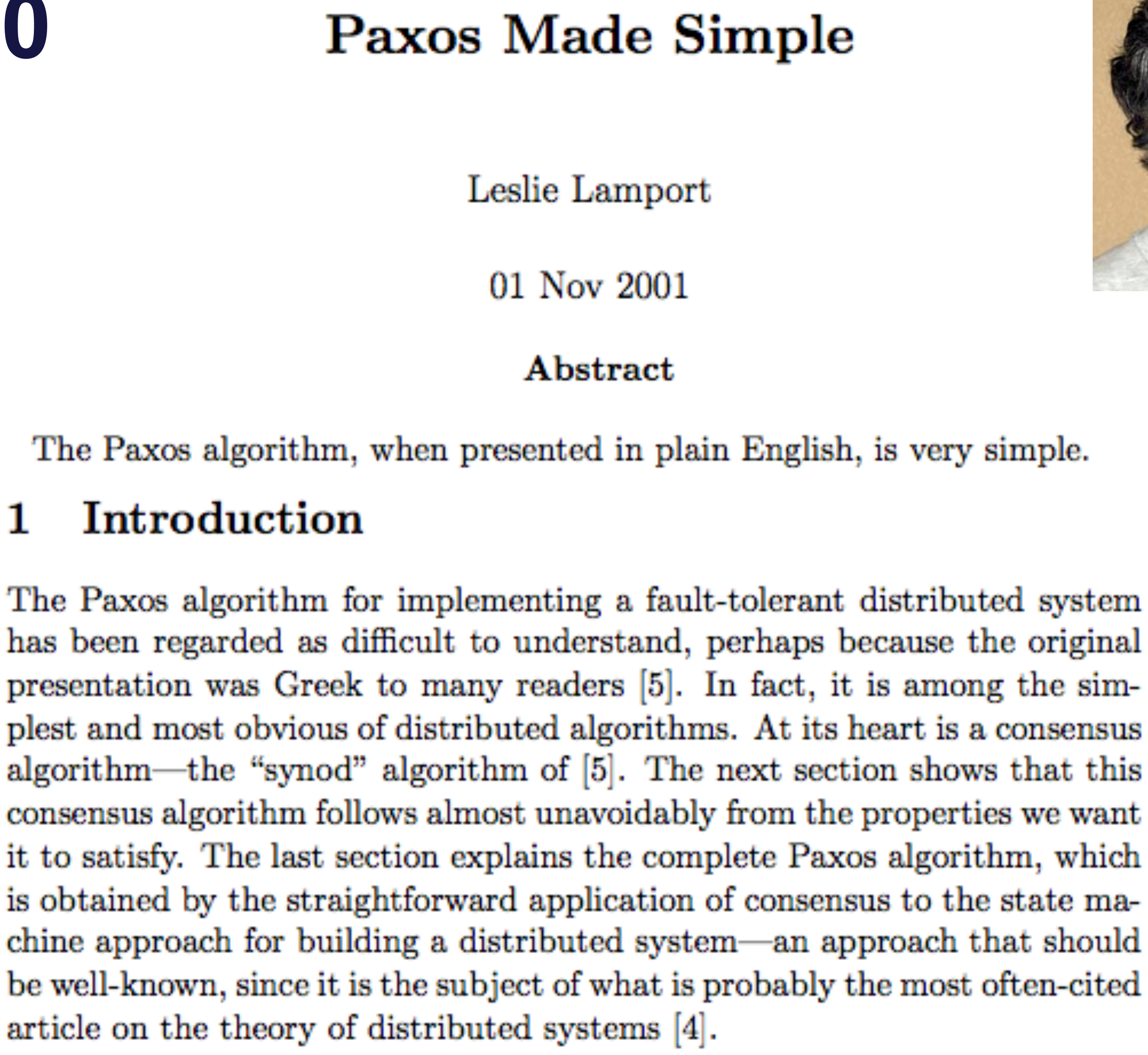
But Δῐκστρα did not pay close attention to what Parliament did, so he did not learn of this decree right away. There was a period of confusion in the cheese market when both Δῐκστρα and Γωυδα were inspecting cheese and making conflicting decisions.

To prevent such confusion, the Paxons had to guarantee that a position could be held by at most one bureaucrat at any time. To do this, a president included as part of each decree the time and date when it was proposed. A decree making Δῐκστρα the cheese inspector might read

2716: *8:30 15 Jan 72* – Δῐκστρα *is cheese inspector for 3 months*

???

v2.0



Paxos Made Simple

Leslie Lamport

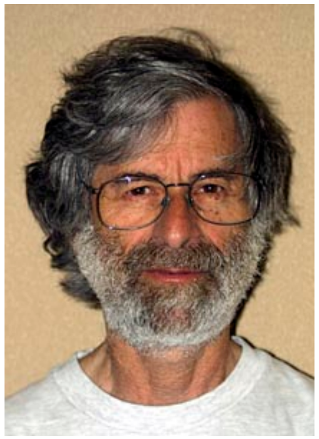
01 Nov 2001

Abstract

The Paxos algorithm, when presented in plain English, is very simple.

1 Introduction

The Paxos algorithm for implementing a fault-tolerant distributed system has been regarded as difficult to understand, perhaps because the original presentation was Greek to many readers [5]. In fact, it is among the simplest and most obvious of distributed algorithms. At its heart is a consensus algorithm—the "synod" algorithm of [5]. The next section shows that this consensus algorithm follows almost unavoidably from the properties we want it to satisfy. The last section explains the complete Paxos algorithm, which is obtained by the straightforward application of consensus to the state machine approach for building a distributed system—an approach that should be well-known, since it is the subject of what is probably the most often-cited article on the theory of distributed systems [4].



“Other” consensus algorithms

- **Viewstamped Replication (VR)**
  - Barbara Liskov / Brian Oki 1988
  - Chapter on “Replication”, 2010
- **Raft**
  - Diego Ongaro, John Ousterhout et. al., 2014

VR is the same as Paxos, but explained more directly. It was ahead of its time, and its significance was not recognized.

“Everything I know about systems I learned from Barbara Liskov.”

Raft is the same as VR, but uses different vocabulary and minor differences to the message protocol and leader election.

Systems and terminology, and p3

- Raft, VR, and Paxos are “the same”.
- These slides mix graphics and terms from all three of these consensus presentations.
- What is said applies to Raft and the lab p3.
- For p3, we focus on two parts of Raft: **leader election** and **log repair**.
- For p3, there are no clients and no new requests to the service. It is “just as if” the servers all restart after a series of failures, and they must agree on the history of actions.

VR

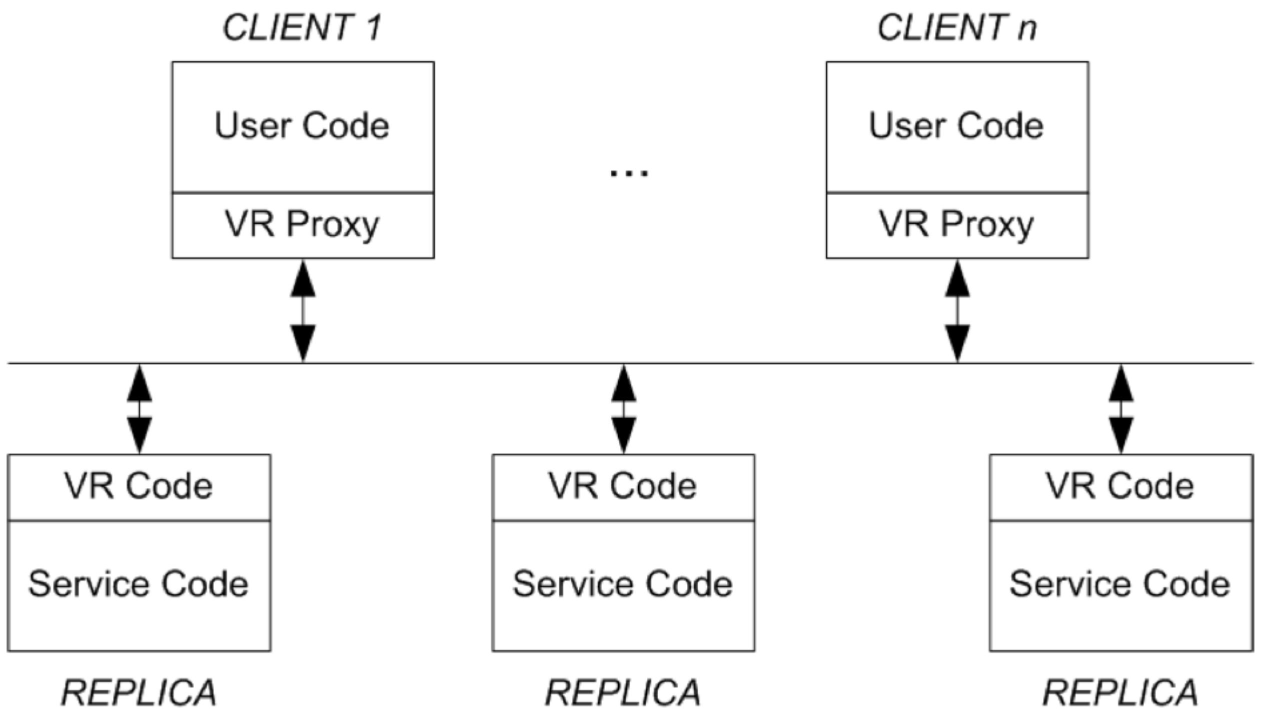
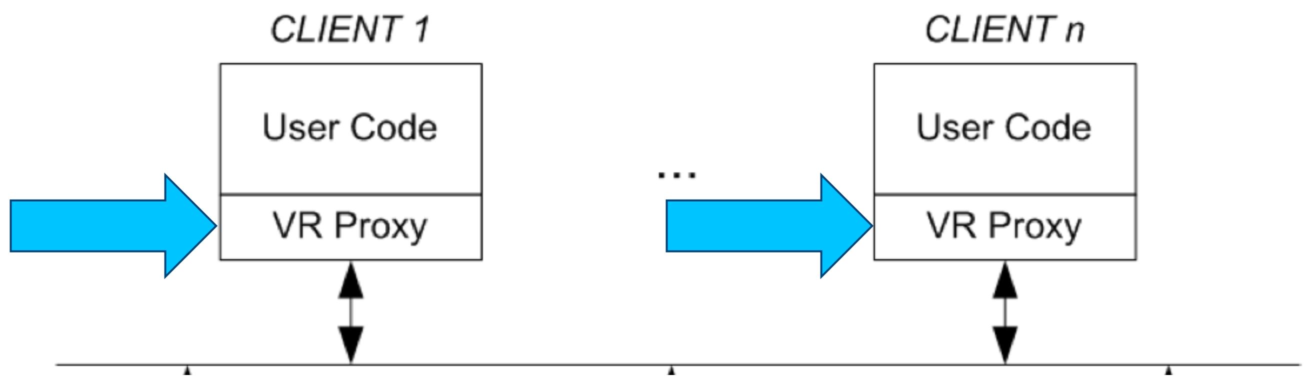


Figure 1: VR Architecture; the figure shows the configuration when $f = 1$.

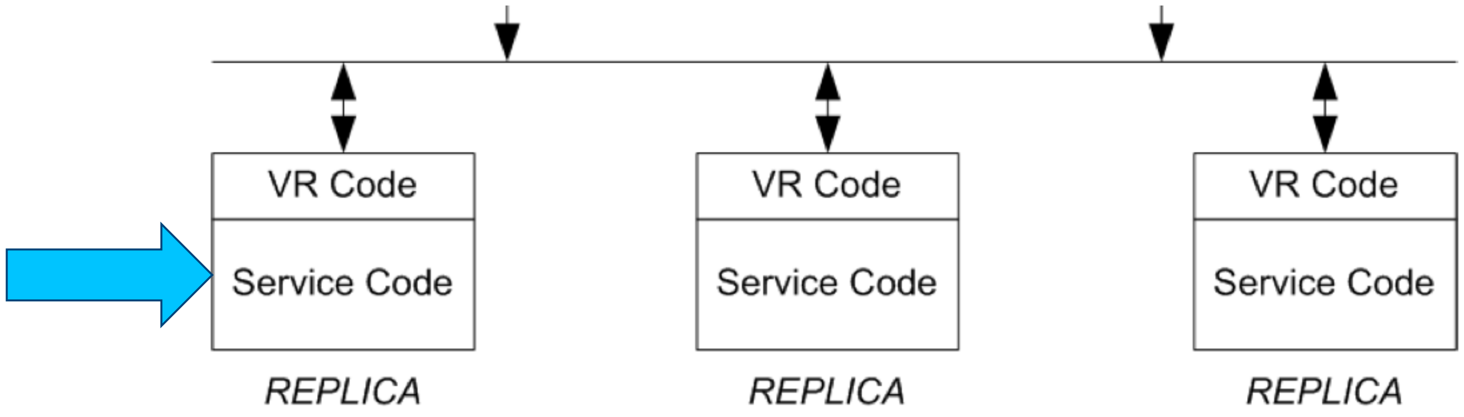
In VR, the leader is chosen from among the replicas; the non-leader replicas (2f of them) serve as backup servers to tolerate up to f concurrent failures.

VR: proxy



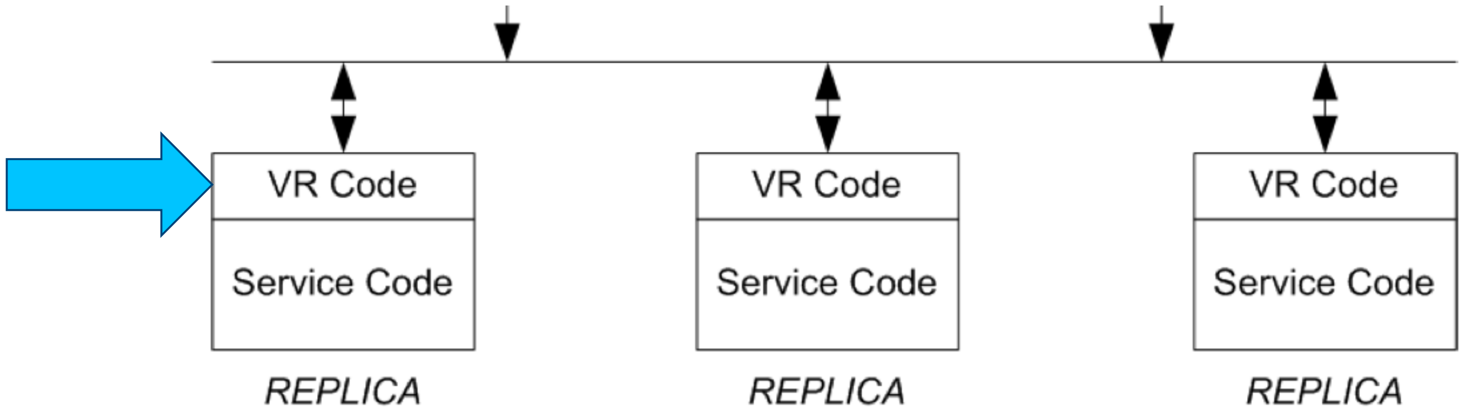
- **VR proxy** code runs in each client.
  - Discover/track the leader and send requests to it.
  - Tag requests with a monotonic sequence number.
  - Suppress duplicate replies.
- The **user code** forms the requests: VR is independent of the application, so we say nothing more about it.

Service Code



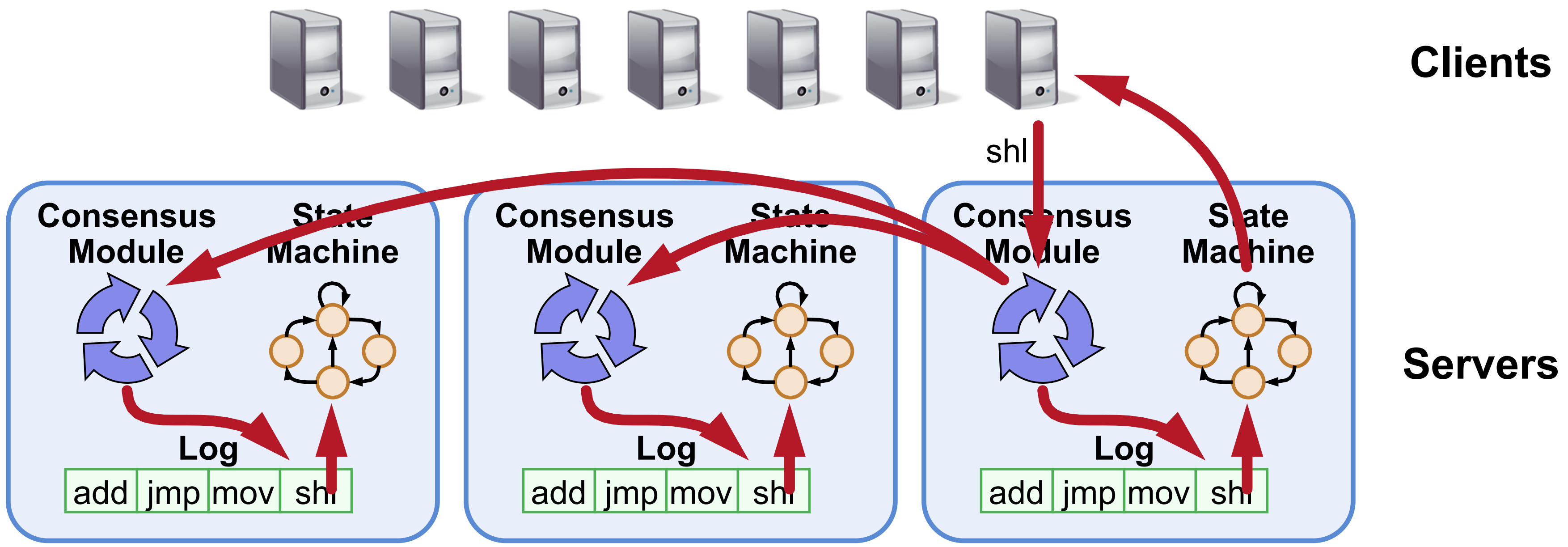
- Each replica runs a copy of the application-defined **Service Code**, which maintains the application state.
- The Service Code receives a sequence of commands/ operations and executes them in order (RSM).

VR Code



- The VR code accepts requests and sequences them.
- When a requested operation has **committed**, the VR code passes it to the the Service Code (RSM) to execute.
- Once the Service Code receives an operation, you can't take it back! It is committed for all time!
- It maintains operation history as an append-only **log**.

Goal: Replicated Log



- **Replicated log => replicated state machine**
  - All servers execute same commands in same order
- **Consensus module ensures proper log replication**
- **System makes progress as long as any majority of servers are up**
- **Failure model: fail-stop (not Byzantine), delayed/lost messages**

The operation log / sequence

- The committed operation log has a sequence of **entries**.
  - Paxos: **slots**
  - VR: **op-numbers**
  - Raft: log **entry** / log **index**
- **Goal**: agree on an action/op/command for each index.
- Each replica maintains its **log**: a sequence of actions that it has accepted (agreed to).
  - **Note**: each replica might keep its log in memory, or on stable storage ("disk"), as it chooses. There is no requirement that any part of the log on any replica survives restarts—**if** our failure assumption is true: a majority of replicas are up, at any time, always, no matter what.
- Let us suppose that each protocol round is concerned only with choosing an action for the "next" log entry.

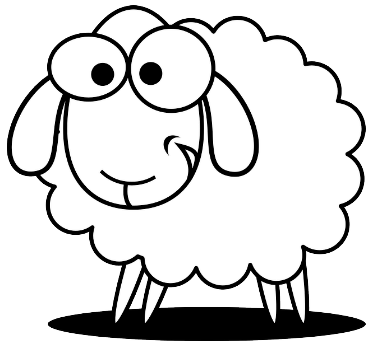
How to agree on the next entry?

- This is not rocket science: we can make it sound hard, but let's **try** to make it sound easy.
- It **is** easy:
  - Pick a **leader** (**primary**) from among the replicas.
  - The leader receives requests/commands/**actions** from clients.
  - The leader picks a sequence for the actions, and tells the other replicas (the **secondary** replicas).
  - Once a majority of replicas have heard and agreed on each (index, action) pair, the action is **committed** for that index.
  - The leader responds to the clients after commit.
  - All replicas apply committed actions in the agreed commit order.
  - → All replicas converge to the same state.

The players

Leader / primary

1. Become leader.
2. Rewrite history.
3. Dictate the future.
4. Stay leader forever.
5. If deposed **goto** step 1.



Acceptor / secondary

1. Adopt leader.
2. Tell it your history.
3. Accept whatever the leader says.
4. Write it all down.
5. If a **new** leader appears, **goto** step 1.

VR: Reaching consensus

- It’s easy if you have a leader and everybody follows!

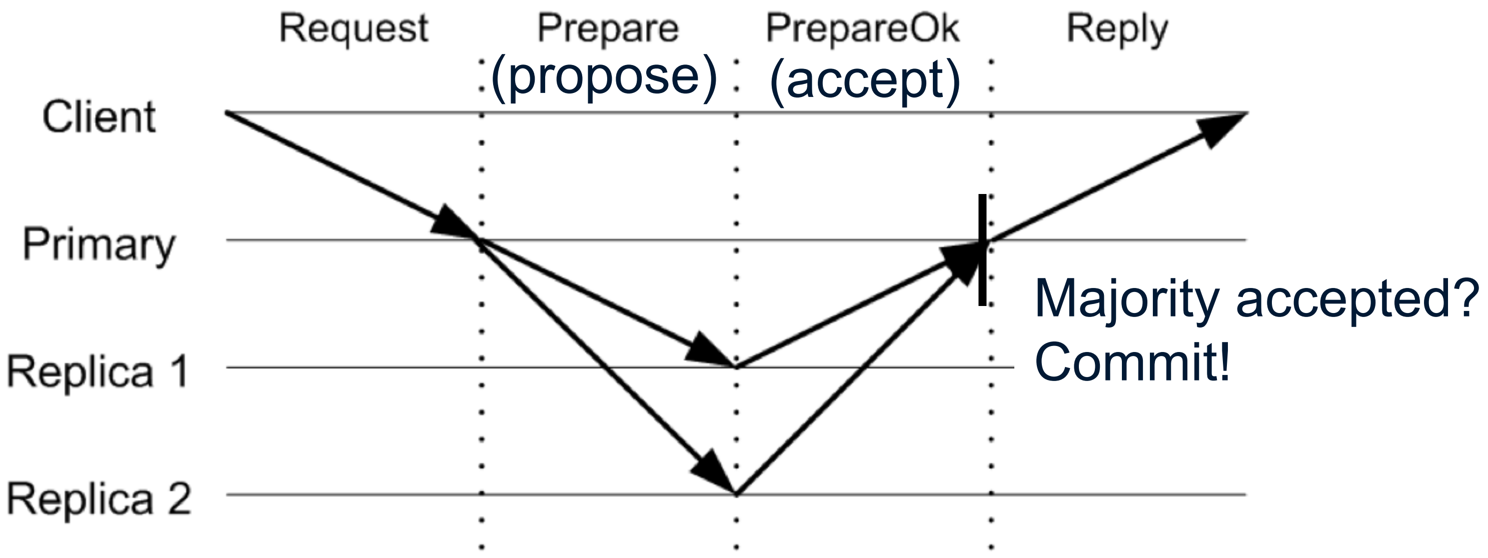
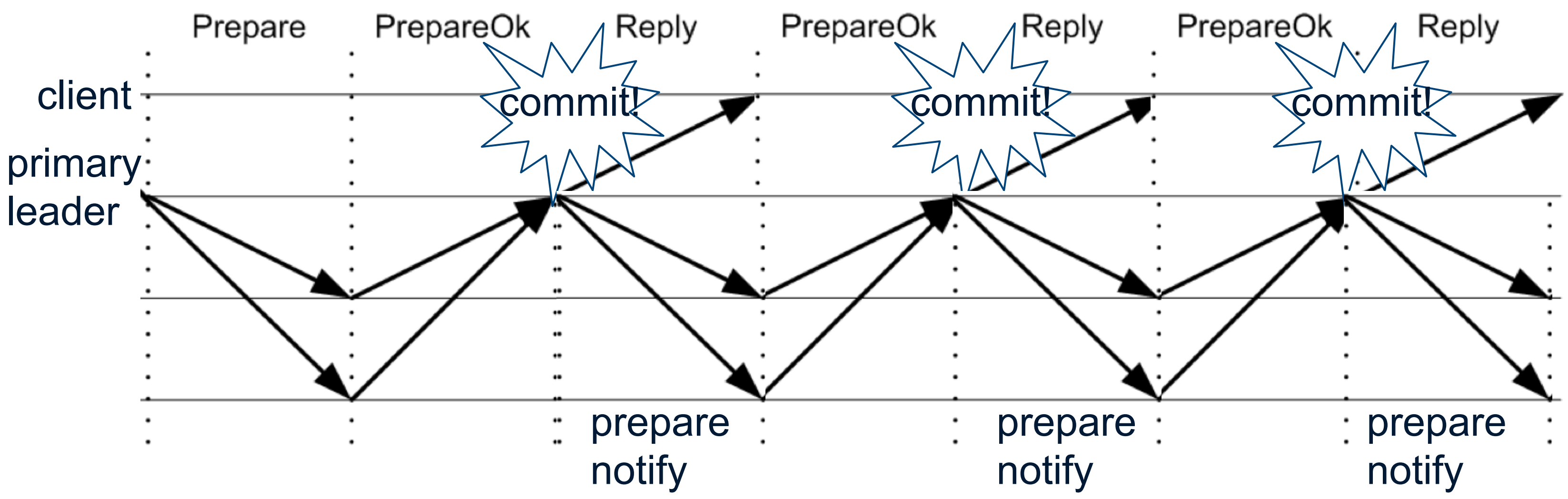


Figure 3: Normal case processing in VR for a configuration with $f = 1$.

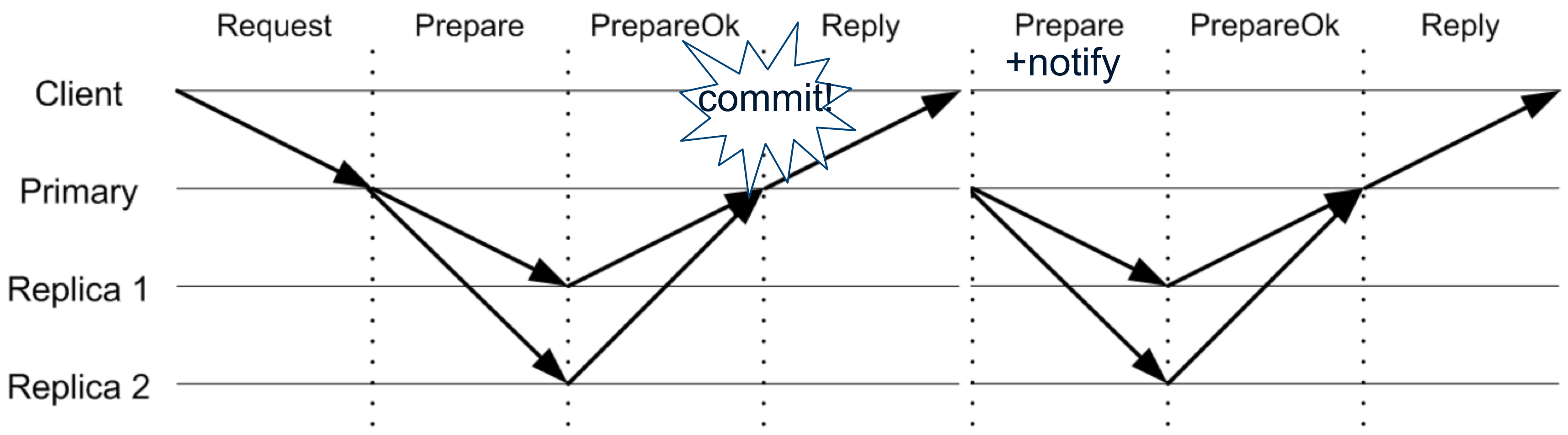
The easy part: a stable view



A leader may send multiple ops to accepters in each **prepare** message (e.g., as in Raft AppendRPCs).

Safety: All majority-accepted operations must survive into future views, even if failure strikes and nobody knows that they have committed. (“First writer wins forever.”)

VR: Learning of commitment



At some point after the operation has committed, the primary informs the other replicas about the commit. This need not be done immediately. A good time to send this information is on the next PREPARE message, as piggy-backed information; only the *op-number* of the most recent committed operation needs to be sent.

When a non-primary replica learns of a commit, it waits until it has executed all earlier operations and until it has the request in its *log*. Then it executes the operation by performing the upcall to the service code.

Does this algorithm work?

That’s really all there is to it, **if we have good leaders**:

1. Leaders rule only with consent of the governed: they rule only if a majority of acceptors adopt and follow.
2. Leaders don’t fight: if somebody else is leading, then they follow or get out of the way.
3. Leaders accept and promulgate the consensus view of history. They don’t try to change the past.
4. Leaders decide and apply their decisions consistently.

Also: acceptors may fail and forget, but they do not lie.

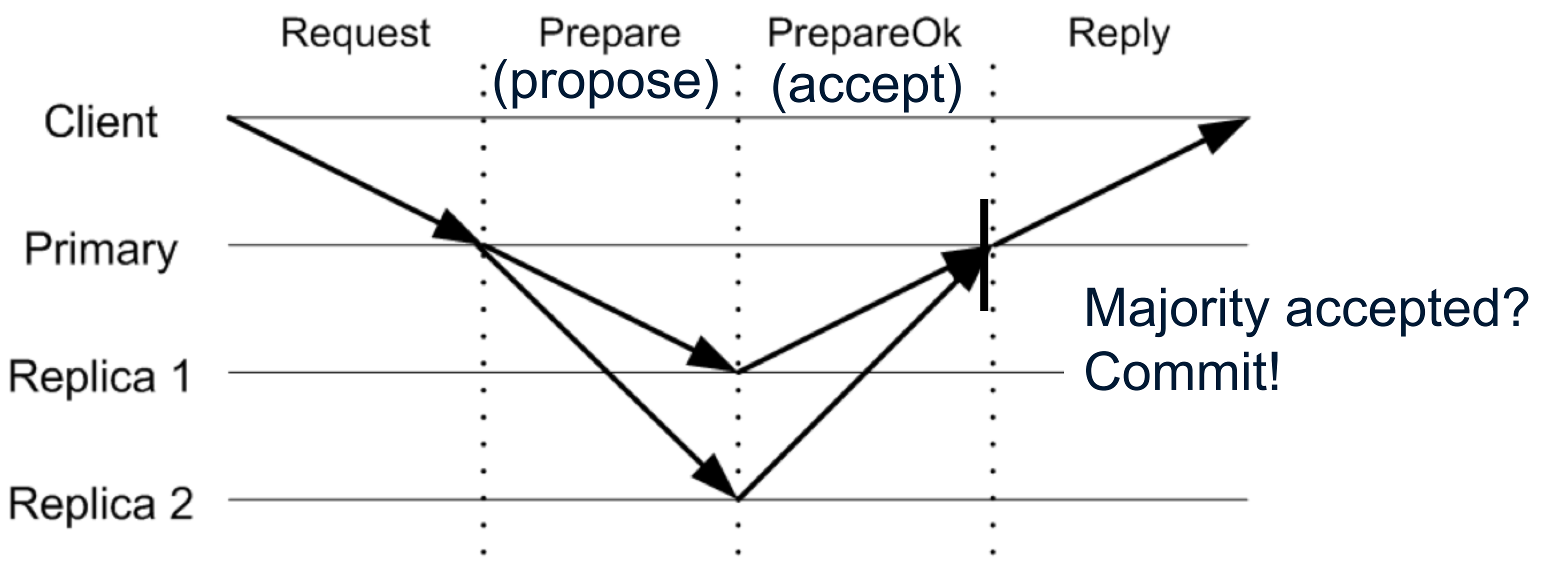
Why is consensus hard?

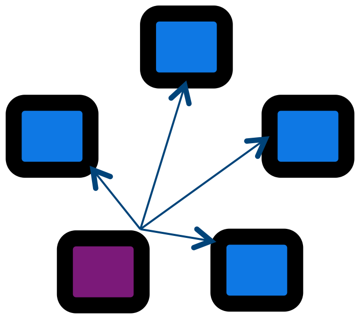
OK, maybe it **is** rocket science…

- What if the leader fails?
- What if the network is partitioned? Could there are leaders on either side of the partition?
- Or it there **appear** to be failures and partitions because the network is slow?
- What if a partition heals, so now there are two leaders?
- What if replicas stall, or fail and then recover? How do they get back up to date?

Answer: establish clear rules for who the leader is, for every contingency.   Keep the leader up to date. The leader keeps others up to date. And vote: majority rules.

Recap: an RSM replica group

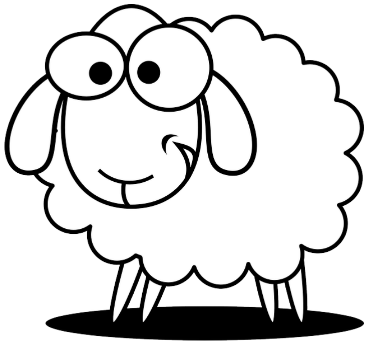




Each replica is a copy of the application server, with all of its state. Replicas keep a **log** of operations (at least in memory) and apply them in order to their state. The consensus protocol ensures that their states converge. We assume that only that the app is deterministic and that **always** some majority is functioning correctly; faulty replicas just stop, and may restart "empty" and resync.

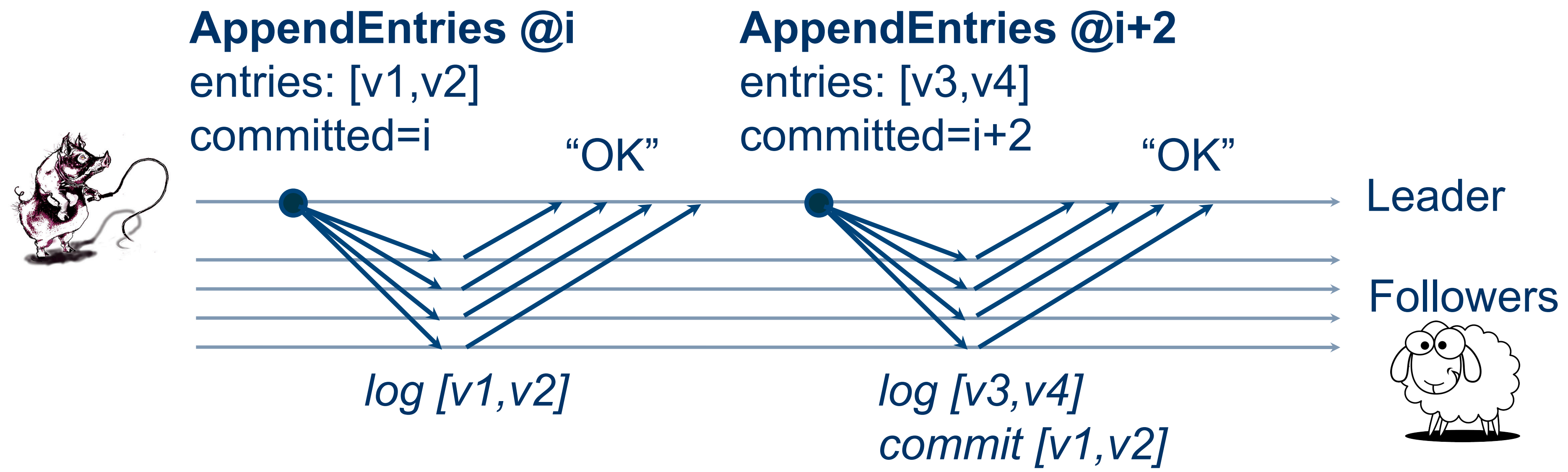
Consensus oversimplified

1. If leader appears failed and now is a good time to run, declare candidacy.
2. Become leader by majority vote.
3. Discover and affirm history.
4. Propose values for new log slots in order; notify others if majority accepts.



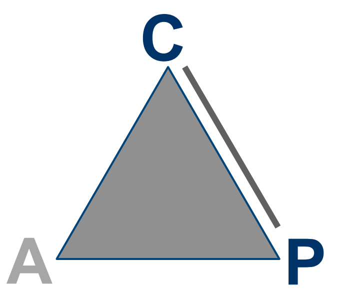
1. Adopt leader.
2. Tell it your history.
3. Accept whatever the leader says.
4. Write it all down.
5. If a new leader appears, **goto** step 1.

Raft in normal operation

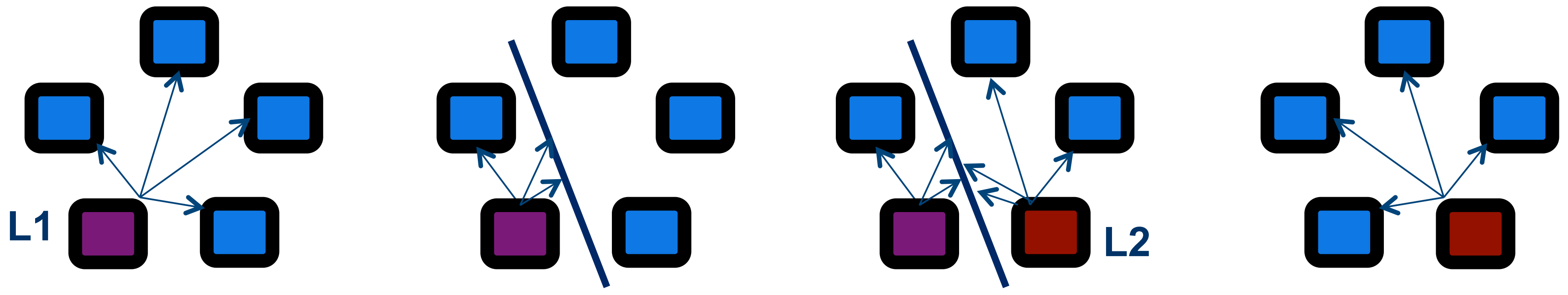


Followers accept whatever the leader proposes.
Accept **everything**, even if it overwrites log history.

A nasty scenario…



1. **Network partition: leader L1 survives with minority.**
   - How to make progress where we can do it safely?
   - How to avoid accepting requests where it might be unsafe?
2. **Partition heals → L1 and L2 both try to lead.**
   - How to establish which leader is "current", so L1 steps down?
   - How to reconcile conflicting histories in the logs?



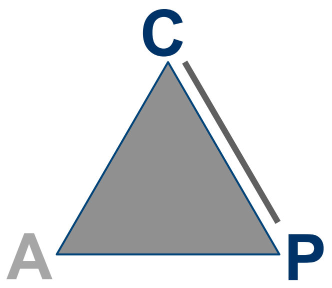
A happy group of 5 replicas.

Oops, network partition. Leader L1 survives on minority side.

L2 is elected on majority side; L1 struggles on.

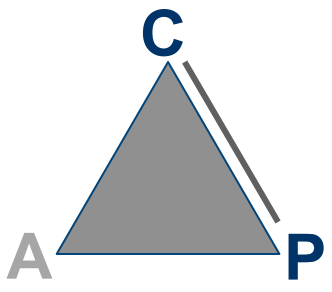
Partition heals. L1 hears L2 and steps down.

A nasty scenario…



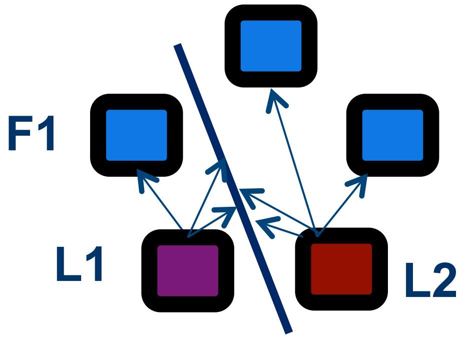
1. **Network partition: leader survives in minority side.**
   - How to make progress where we can do it safely?
   - How to avoid accepting requests where it might be unsafe?
   - **Answer: majority rules**. Leader requires a majority vote (**quorum**). Majority side elects new leader (L2) and continues.
   - No requests can commit under L1: no majority. But they try!
2. **Partition heals → L1 and L2 both try to lead.**
   - How to establish which leader is "current", so L1 steps down?
   - How to reconcile conflicting histories in the logs?
   - **Answer**: monotonically increasing **terms** / **views**. Everyone can see that L1 is an old expired leader from an earlier term. L1 steps down and yields to L2. Uncommitted log entries written under L1 are overwritten with L2's history.

The importance of majority (quorum)



- In order to lead and serve client requests, a leader L1 must continually receive votes from a **majority** of the group (a **quorum**).
- The **quorum rule** protects consistency (**C**) in a network partition: at most subgroup commits to the log, so it does not diverge.
- But it sacrifices availability (**A**). If a majority of replicas fail, it would be safe for the survivors to serve clients. But they **must not**, because this case is **indistinguishable** from a network partition.

L1 and F1 cannot commit log entries because they do not have a quorum. So they cannot serve clients → no **A**vailability if the majority had merely failed.

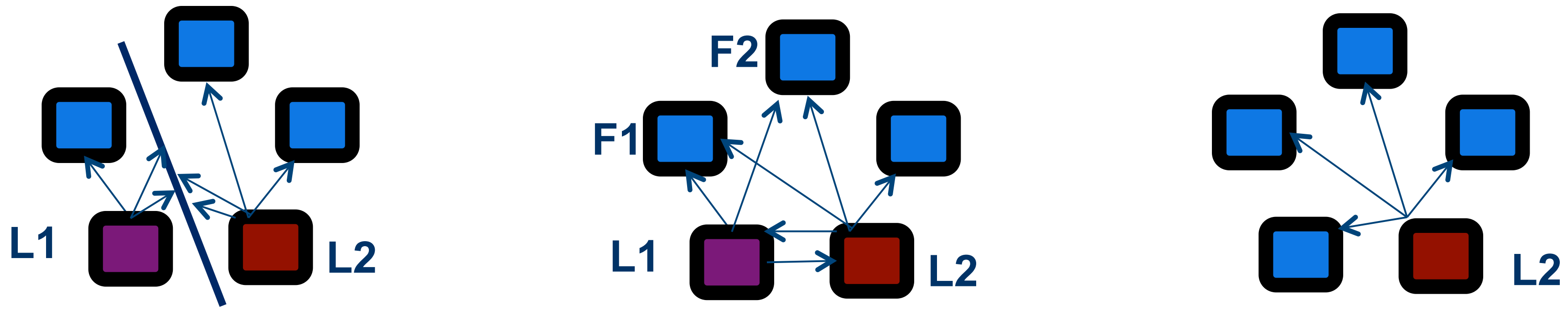


L2 can gain a quorum from the other followers, so they can continue to serve requests together in close cooperation.

The quorum rule protects **C**onsistency of the shared log in a partition. At most one side of a partition makes progress: **there cannot be two disjoint majorities**.

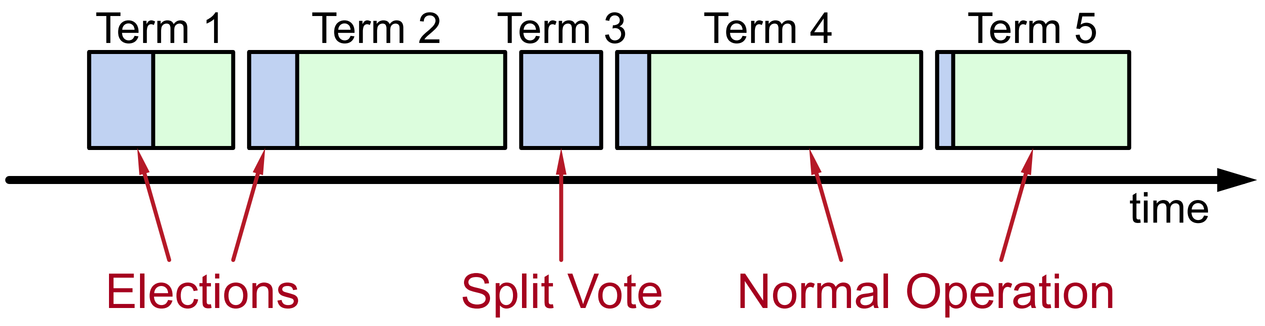
Rejecting a leader from the past

- The protocol runs as a sequence of **views** or **terms**.
- A new view/term begins when some participant (e.g., L2) declares an **election** (e.g., because L2 does not hear the old leader L1).
- Winning requires a quorum → at most one leader per view/term.
- Nobody pays attention to a leader/candidate L1 from the past, and L1 steps down if L1 learns of a term later than its own.



During the partition, L2 declares candidacy in a new term and wins a quorum. When the partition heals, F2 rejects messages from L1, whose term has expired. F1 learns of the new term and defects to L2, also rejecting L1. L1 learns of the new term and steps down.

Terms



- **Time divided into terms:**
  - Election
  - Normal operation under a single leader
- **At most 1 leader per term**
- **Some terms have no leader (failed election)**
- **Each server maintains current term value**
- **Key role of terms: identify obsolete information**

Spelling it out: views/terms

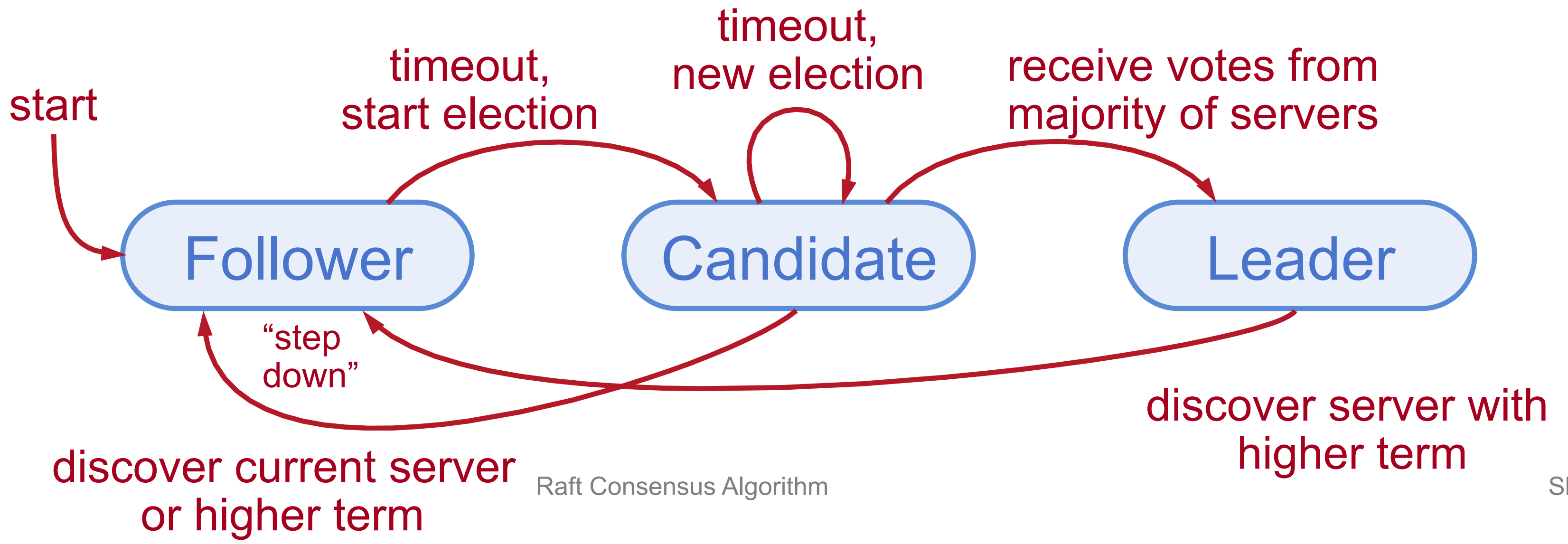
- In general, a participant P may leave a group temporarily due to loss of connectivity or failure/restart.
- The group moves on, and when P rejoins the group, P's state is seen to be **stale**: P can't participate correctly.
  - E.g., in consensus, P's log has diverged or has missing entries, P's view of the identity of the group's leader is no longer correct.
- Consensus (and other distributed protocols) need some way to detect stale participants and reintegrate them.
- Consensus uses view/term for this purpose.
  - Tag messages with the view/term, tag state with the view/term, and compare tags to detect staleness.

Numbering the views (terms)

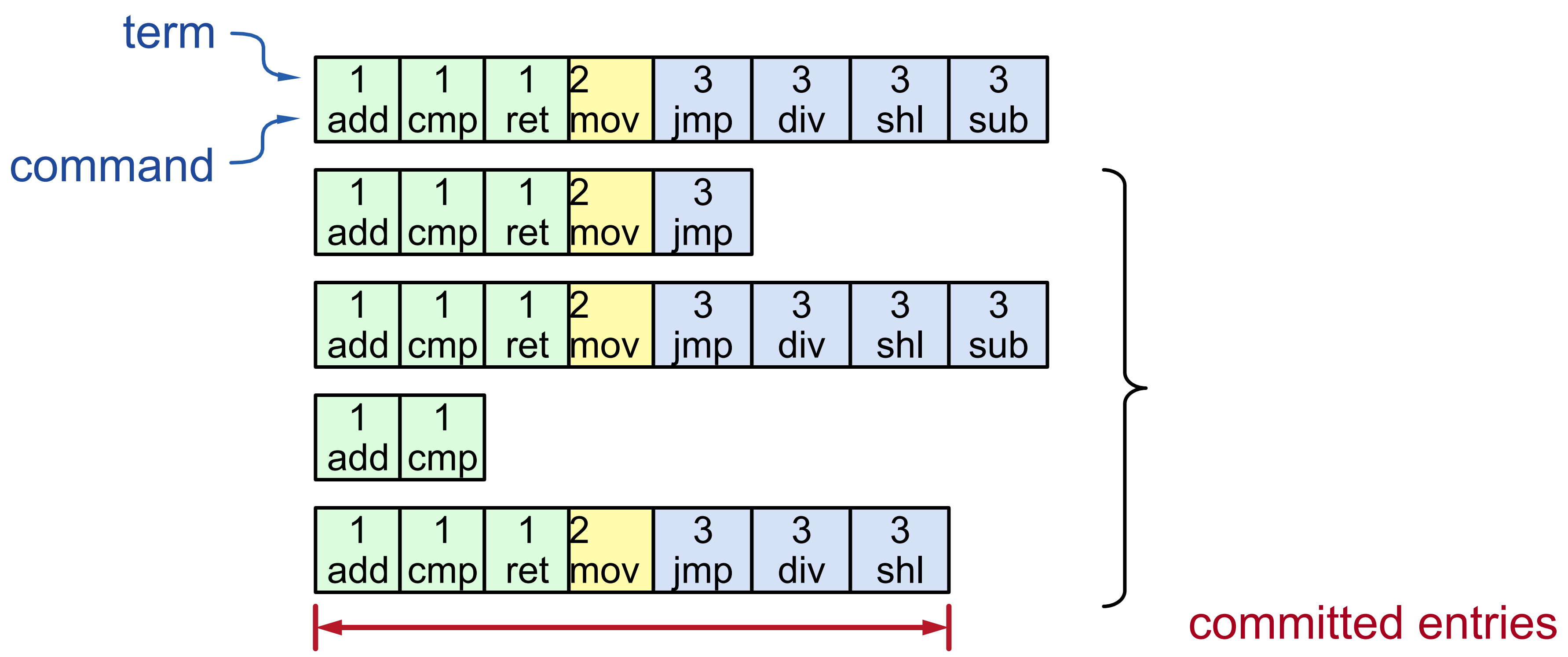
- Every message in the protocol carries the sender's current view# (term).
- If the receiver has a higher view# than the sender, it **ignores and rejects** the message.
  - Stale participants can do no damage.
- If a participant learns of a view# **v** higher than its own, it sets its view# to **v.**
  - The view# is an example of a **logical clock**, a foundational concept in distributed systems.
- When a participant learns it is stale, it comes up to date.
  - E.g., in the "nasty scenario" example, L1 steps down, and L1 and F1 accept updates from new leader L2 to repair their logs.

Server States

- **At any given time, each server is either:**
  - Leader: handles all client interactions, log replication
    - At most 1 viable leader at a time
  - Follower: completely passive (issues no RPCs, responds to incoming RPCs)
  - Candidate: used to elect a new leader
- **Normal operation: 1 leader, N-1 followers**


start
timeout, start election
timeout, new election
receive votes from majority of servers
Follower
Candidate
Leader
"step down"
discover current server or higher term
discover server with higher term

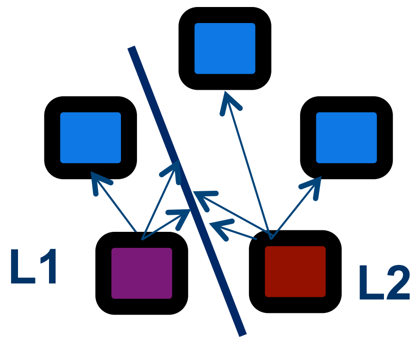
Log Structure

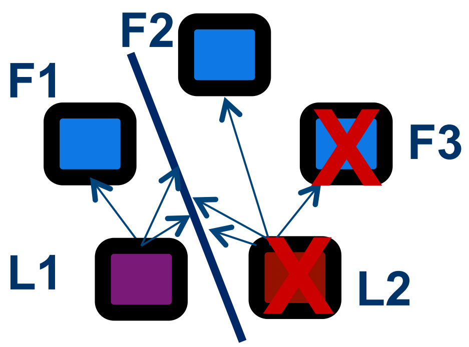


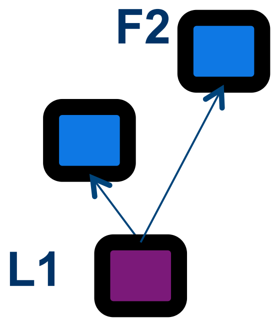
- **Log entry = index, term, command**
- **Every entry is tagged with a term. These tags are used to detect stale entries, and in leader election and log repair.**

Nasty scenario: getting nastier

- A bare majority of replicas commit new entries under L2.
- The minority writes new uncommitted entries under L1.
- Then L2 and a follower (F3) fail, and the minority rejoins.
- Progress is possible, but the replicas have divergent logs. All committed updates **must** be preserved.
- How to recognize the lone survivor (F2) who knows the committed history, and who can pass it to the others?



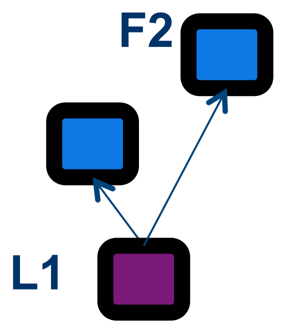


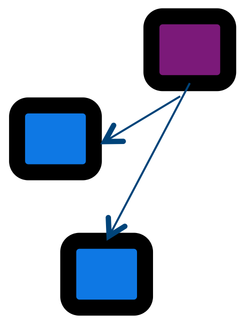


Nasty scenario: getting nastier

- **Answer**: look at the terms in the log entries.
  - Every entry is tagged with a view#/term.
  - This is the key to the log repair protocol.
- F2 rejects L1 because F2 knows a higher term than L1. L1 learns its term has expired, and L1 steps down.
- A new election is declared → new term and new leader.
- F2 has log entries from a higher term than the others. Higher terms dominate → the others accept F2's entries.

F2 rejects L1.
L1 steps down.
A new term begins.
An election starts.



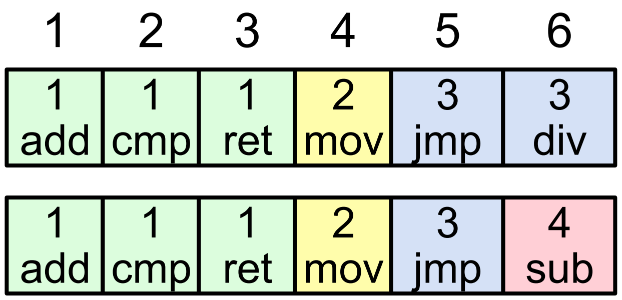


In Raft, F2 wins because its log is most advanced (higher terms). F2's log is used to repair the others. And the group continues.

Log Consistency

High level of consistency between logs:

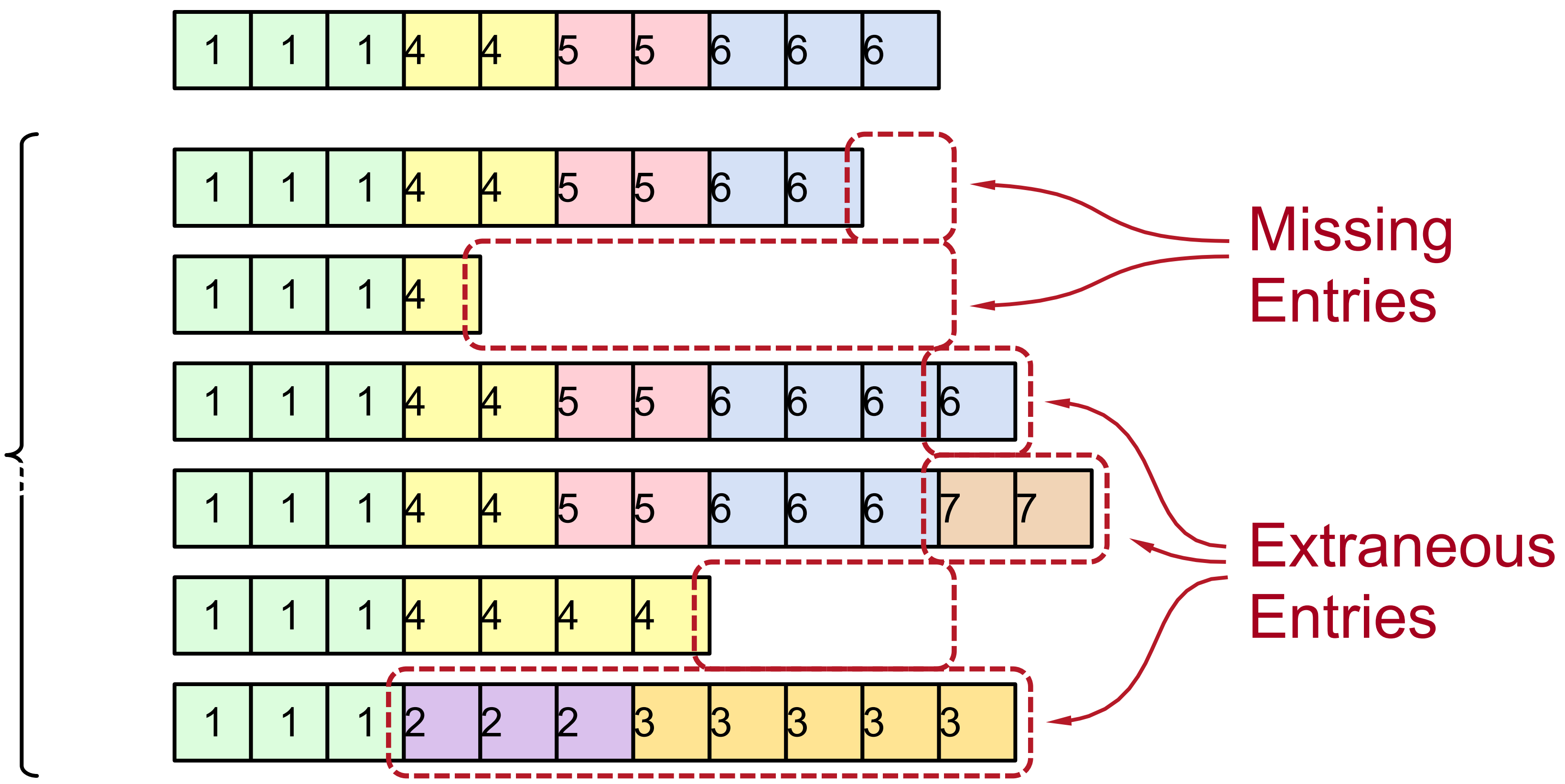
- **If log entries on different servers have same index and term:**
  - They store the same command
  - The logs are identical in all preceding entries


1 2 3 4 5 6
1 add | 1 cmp | 1 ret | 2 mov | 3 jmp | 3 div
1 add | 1 cmp | 1 ret | 2 mov | 3 jmp | 4 sub

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

Log Inconsistencies

Leader changes can result in log inconsistencies:



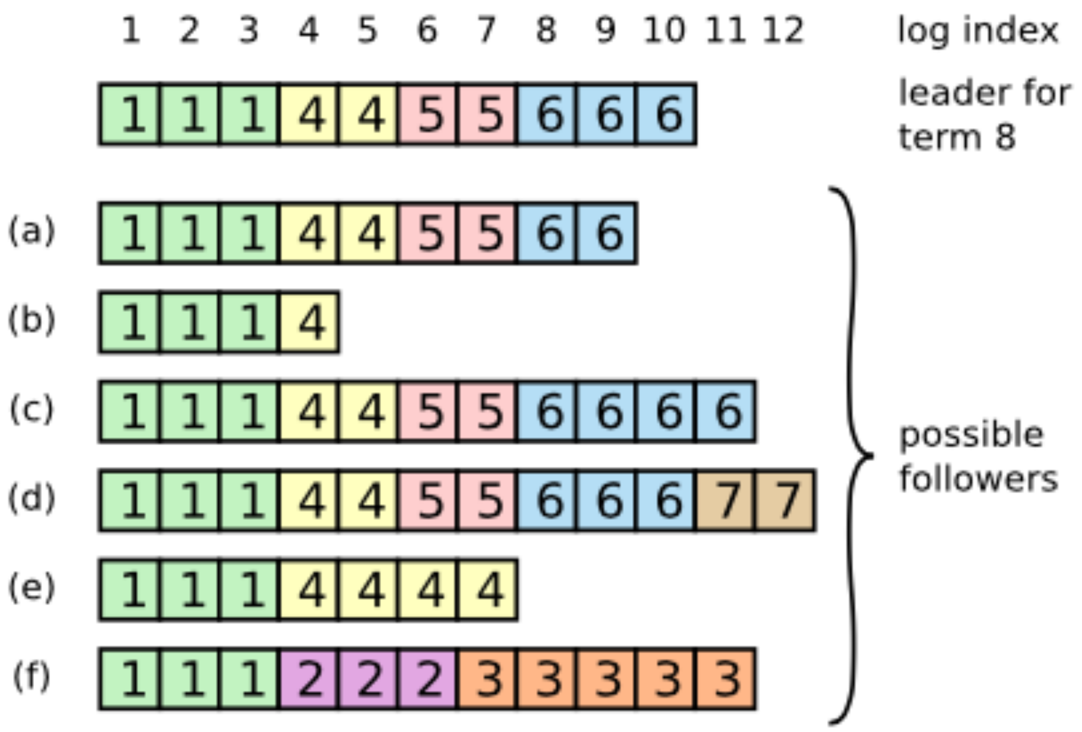
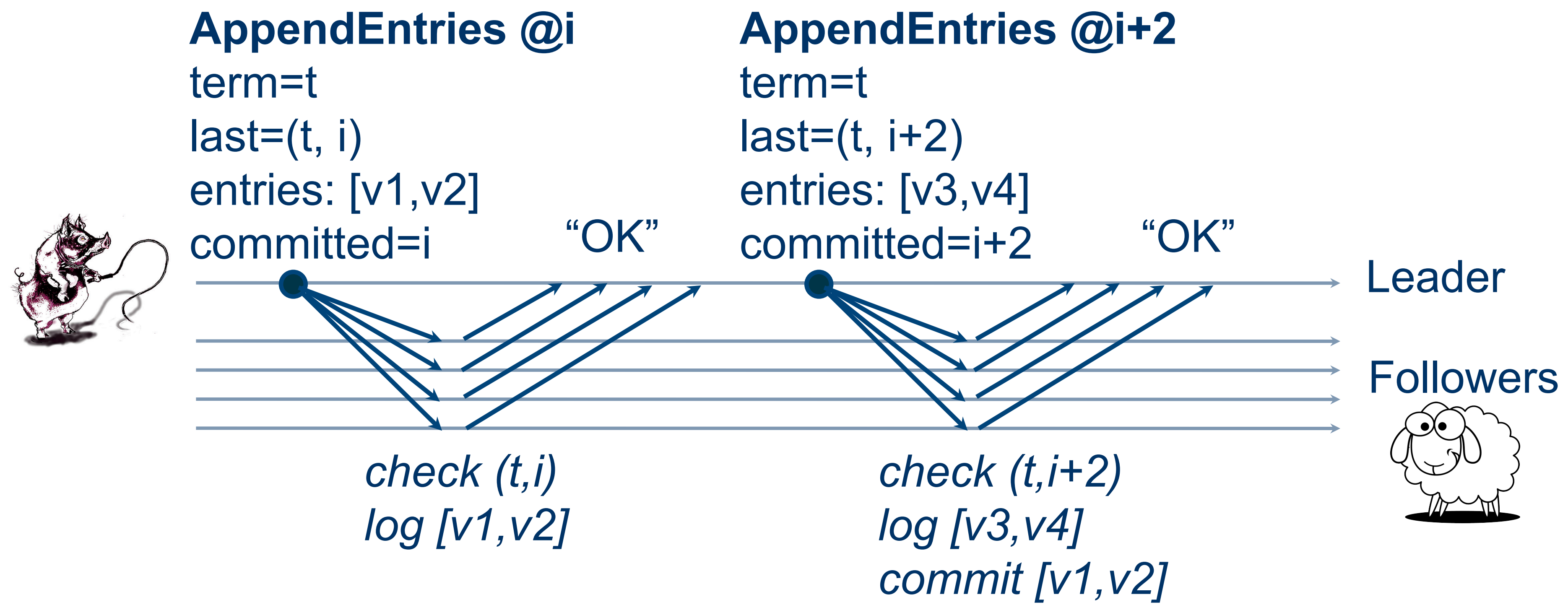


Figure 7: When the leader at the top comes to power, it is possible that any of scenarios (a–f) could occur in follower logs. Each box represents one log entry; the number in the box is its term. A follower may be missing entries (a–b), may have extra uncommitted entries (c–d), or both (e–f). For example, 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.

Raft in normal operation (stable term)



Followers accept whatever the leader proposes.
But first: validate that (term, index) are "in sync".
Leader's term is behind? → reject proposal: "I won't follow you!"
Leader's last=(t,i) is ahead? → reject proposal: "Catch me up!"
Once in sync, accept **everything**, even if it overwrites log history.

Raft: log repair

To bring a follower's log into consistency with its own, the leader must find the latest log entry where the two logs agree, delete any entries in the follower's log after that point, and send the follower all of the leader's entries after that point.

All of these actions happen in response to the consistency check performed by AppendEntries RPCs. The leader maintains a nextIndex for each follower, which is the index of the next log entry the leader will send to that follower. When a leader first comes to power, it initializes all nextIndex values to the index just after the last one in its log (11 in Figure 7). If a follower's log is inconsistent with the leader's, the AppendEntries consistency check will fail in the next AppendEntries RPC.

After a rejection, the leader decrements nextIndex and retries the AppendEntries RPC. Eventually nextIndex will reach a point where the leader and follower logs match. When this happens, AppendEntries will succeed, which removes any conflicting entries in the follower's log and appends entries from the leader's log (if any).

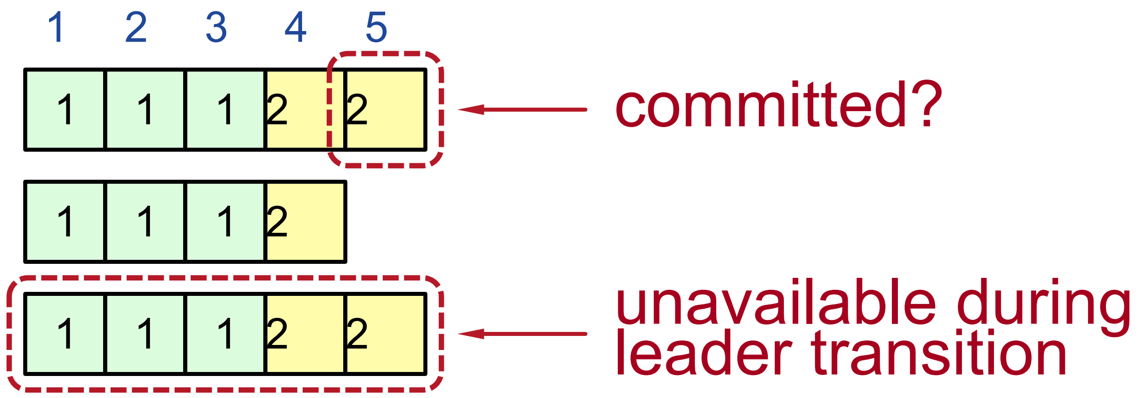
Once AppendEntries succeeds, the follower's log is consistent with the leader's, and it will remain that way for the rest of the term.

Raft: leadership safety condition

- Elections choose a **sufficiently up-to-date leader**:
  - Leader's last(term) is at least as high as all its voters.
  - Leader's last(index) is at least as high as all its voters with the same last(term).
  - Candidates/leaders step down when they see a better (more up-to-date) candidate/leader.
- **Safety property**: a new leader has already seen (and remembers) **at least all committed entries**.
  - The new leader *L* won a majority, and so *L* is at least as up to date as a majority of replicas. Since any committed entry *e* was accepted by a majority of replicas, at least one of the voters *v* for *L* must have entry *e*, and since *L* is at least as up to date as *v*, *L* must have *e* as well.

Picking the Best Leader

- **Can't tell which entries are committed!**



- **During elections, choose candidate with log that contains all committed entries**
  - Candidates include log info in RequestVote RPCs (index & term of last log entry)
  - Voting server V denies vote if its log is "more complete":
    $(\text{lastTerm}_V > \text{lastTerm}_C) \,||$
    $(\text{lastTerm}_V == \text{lastTerm}_C) \,\&\&\, (\text{lastIndex}_V > \text{lastIndex}_C)$

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:

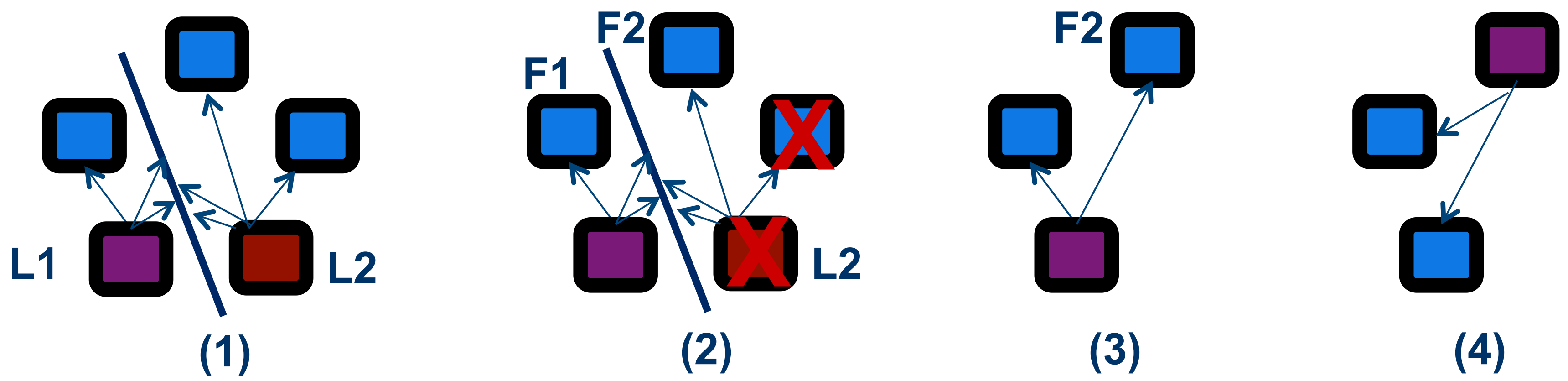
1. Reply false if term < currentTerm (§5.1)
2. If votedFor is null or candidateId, and candidate's log is at least as up-to-date as receiver's log, grant vote (§5.2, §5.4)

Note

- Leader election is the key difference of Raft from VR and Paxos. It leads to a little more thrashing around to pick a leader, but it avoids the need for new mechanisms to update the leader’s state.

Safety: committed entries always survive

- In an earlier scenario, one replica (F2) survives to propagate committed log entries into a new term.



- In fact: for any committed log entry *e*, at least **one replica who knows of *e* survives** into any new term, **always**.
- Why? It requires a quorum to commit entries or elect a new leader, and **any two quorums must intersect**.
  - It is not possible to have two disjoint majorities! This is the key to the proof that Consensus is safe and consistent.

Raft: safety

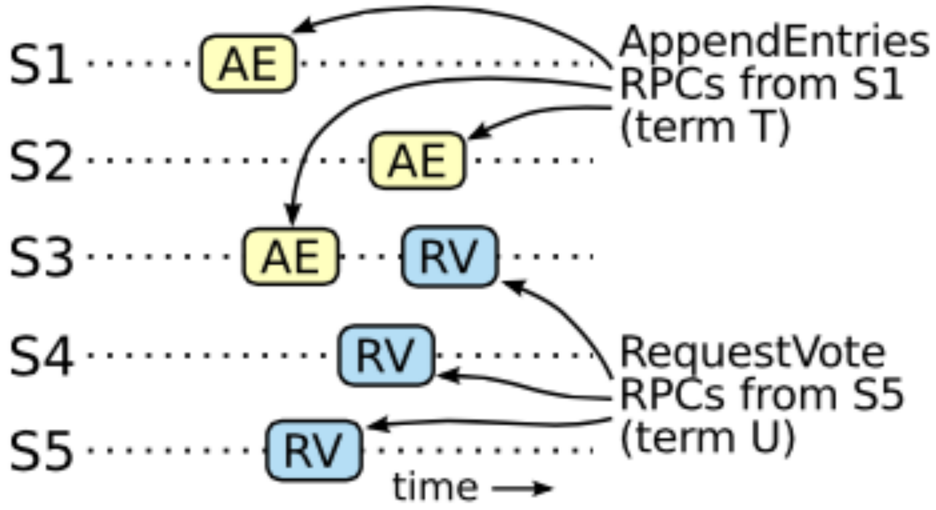

S1
S2
S3
S4
S5
AE
AE
AE
RV
RV
RV
AppendEntries RPCs from S1 (term T)
RequestVote RPCs from S5 (term U)
time

Figure 9: If S1 (leader for term T) commits a new log entry from its term, and S5 is elected leader for a later term U, then there must be at least one server (S3) that accepted the log entry and also voted for S5.

Note: when a new leader L propagates older entries into a new term, L does not and cannot (always) know if those entries committed or not in the old term. But it does not matter: the set of "most up to date" entries is guaranteed to include all committed entries, so all committed entries survive into the new term. And once L propagates them to a quorum, they are known to have committed.

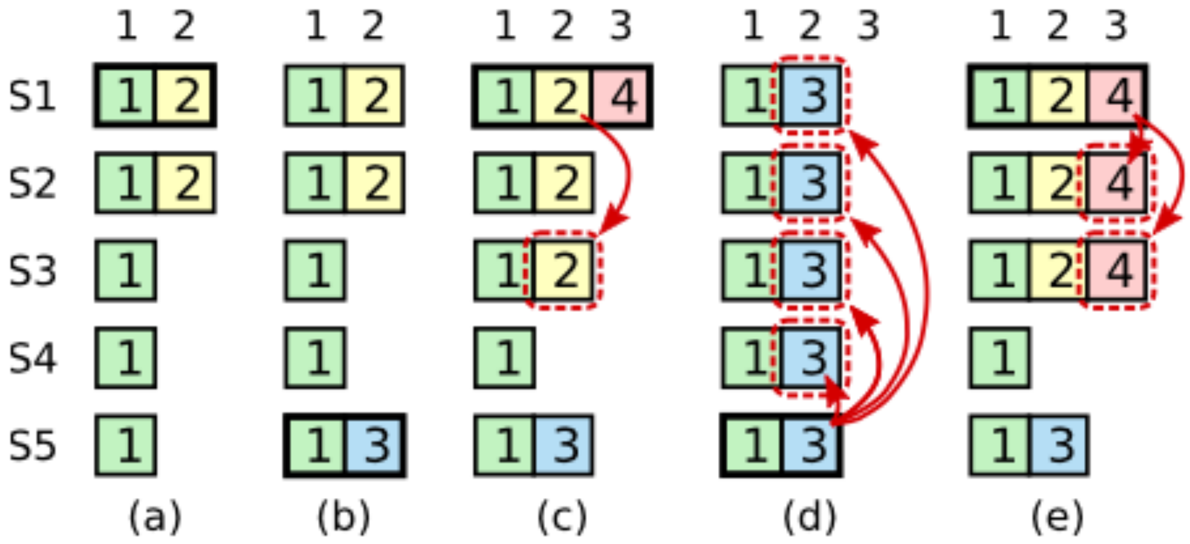


Figure 8: A time sequence showing why a leader cannot determine commitment using log entries from older terms. In (a) S1 is leader and partially replicates the log entry at index 2. In (b) S1 crashes; S5 is elected leader for term 3 with votes from S3, S4, and itself, and accepts a different entry at log index 2. In (c) S5 crashes; S1 restarts, is elected leader, and continues replication. At this point, the log entry from term 2 has been replicated on a majority of the servers, but it is not committed. If S1 crashes as in (d), S5 could be elected leader (with votes from S2, S3, and S4) and overwrite the entry with its own entry from term 3. However, if S1 replicates an entry from its current term on a majority of the servers before crashing, as in (e), then this entry is committed (S5 cannot win an election). At this point all preceding entries in the log are committed as well.

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

Figure 3: Raft guarantees that each of these properties is true at all times. The section numbers indicate where each property is discussed.

Summary

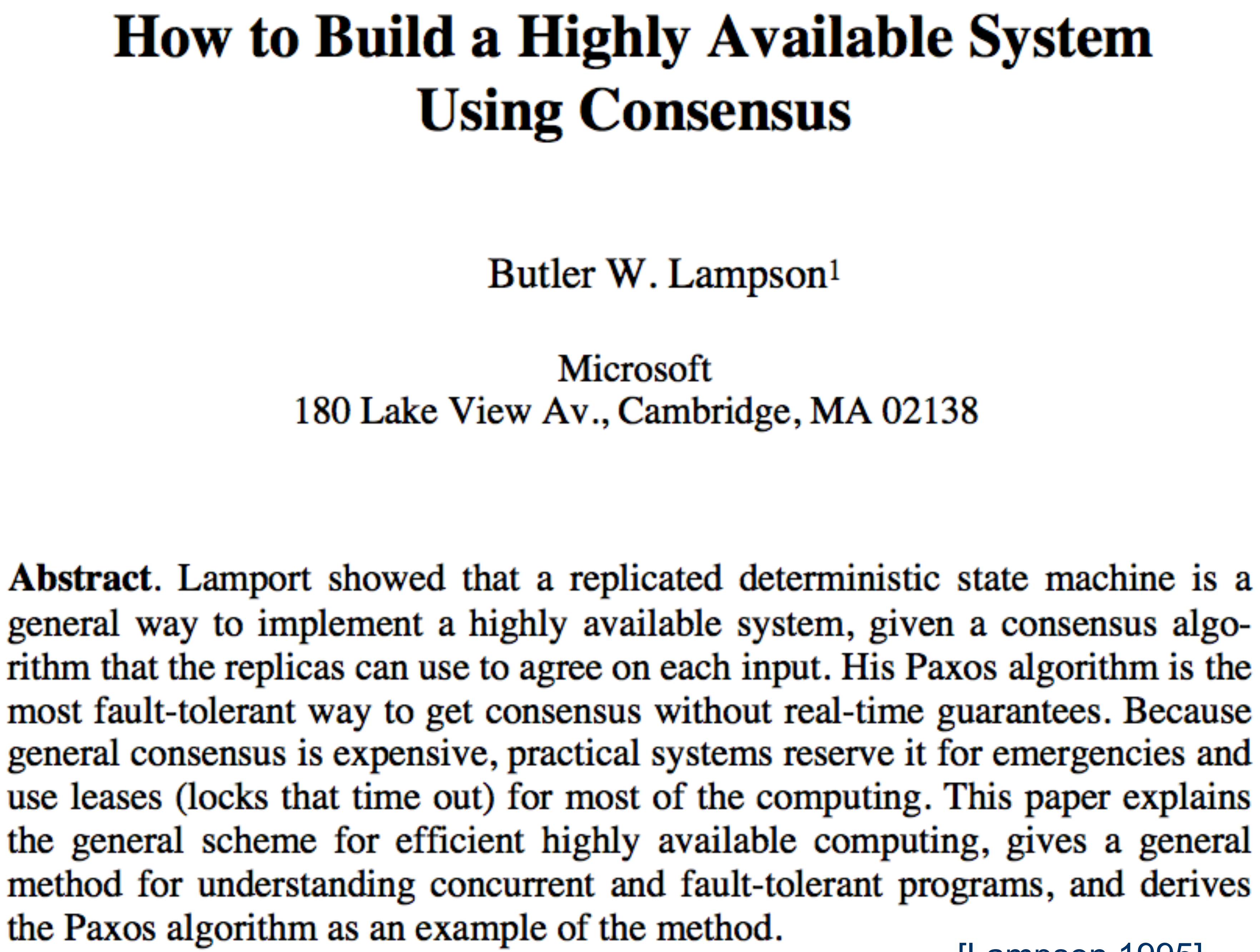
- Leader/master coordinates, **dictates consensus**
  - View the service as a **deterministic state machine**.
  - Master (also called “primary”) dictates order of client operations.
  - → All non-faulty replicas reach the same state.
- Remaining problem: **who is the leader**?
  - Leader itself might fail or be isolated by a network partition.
  - Requires a scheme for “leader election” to choose a new leader.
  - Consensus is **safe** but not **live**: in the worst case (multiple repeated failures) it might not terminate.
  - But in practice Consensus gets the job done…if it can be done.
  - Raft is one variant of Consensus, with minor differences from older variants (VR and Paxos).

Liveness timeouts

- All consensus systems rely on careful timeouts for liveness, because an election or view change disrupts any consensus in progress.
  - Contending leaders can “livelock” the consensus algorithm.
- VR: “More generally **liveness depends on properly setting the timeouts** used to determine whether the primary is faulty so as to avoid unnecessary view changes.”
  - **Note:** also want fast failover times → tight timeouts!
  - Must balance these competing considerations.
- **Raft**: randomized timeouts to avoid contending leaders.

Consensus in Practice

- Lampson: "Since general consensus is expensive, practical systems reserve it for emergencies."
  - e.g., to select a primary/master, e.g., a lock server.
    - Zookeeper
    - Google Chubby service ("Paxos Made Live")
- Pick a primary with Paxos. Do it rarely; do it right.
  - Primary holds a "master lease" with a timeout.
    - Renew by consensus with primary as leader.
  - Primary is "czar" as long as it holds the lease.
  - Master lease expires? Fall back to Paxos.
  - (Or BFT.)



How to Build a Highly Available System Using Consensus

Butler W. Lampson[1]

Microsoft
180 Lake View Av., Cambridge, MA 02138

Abstract. Lamport showed that a replicated deterministic state machine is a general way to implement a highly available system, given a consensus algorithm that the replicas can use to agree on each input. His Paxos algorithm is the most fault-tolerant way to get consensus without real-time guarantees. Because general consensus is expensive, practical systems reserve it for emergencies and use leases (locks that time out) for most of the computing. This paper explains the general scheme for efficient highly available computing, gives a general method for understanding concurrent and fault-tolerant programs, and derives the Paxos algorithm as an example of the method.

[Lampson 1995]

Butler W. Lampson

http://research.microsoft.com/en-us/um/people/blampson

Butler Lampson is a Technical Fellow at Microsoft Corporation and an Adjunct Professor at MIT…..He was one of the designers of the SDS 940 time-sharing system, the Alto personal distributed computing system, the Xerox 9700 laser printer, two-phase commit protocols, the Autonet LAN, the SPKI system for network security, the Microsoft Tablet PC software, the Microsoft Palladium high-assurance stack, and several programming languages. He received the ACM Software Systems Award in 1984 for his work on the Alto, the IEEE Computer Pioneer award in 1996 and von Neumann Medal in 2001, the Turing Award in 1992, and the NAE's Draper Prize in 2004.