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

Jeff Chase
Duke University
What is a distributed system?

"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)
A service

client ---> server
     \    / request
       \  /     reply
         \ /
          \-
server

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!
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

A

lock service

B

x = x + 1

acquire
grant
release

acquire
grant
release

x = x + 1
A lock service in the real world

A

acquire

grant

acquire

???

B

???

B

X

x=x+1
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

A

acquire

grant

B

acquire

release

x=x+1

x=x+1

grant

x=x+1
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

A

acquire

grant

x=x+1

???

B

acquire

grant

x=x+1

release
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

Step 1
Propose.
Each P proposes a value to the others.

Step 2
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
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.

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

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).

Dr. Eric Brewer
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.
Dr. Eric Brewer

“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.

C-A-P “choose two”
Properties for Correct Consensus

- **Termination**: All correct processes eventually decide.
- **Agreement**: All correct processes select the same $d_i$.
  - Or...(stronger) all processes that do decide select the same $d_i$, 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)
“The brain of an enemy”

Finite State Machine (FSM)
Dogs and Cats

[Diagram showing the states and transitions of dogs and cats based on actions and events.]

FSM Basics

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

• Finite set of states and actions (inputs)
• Deterministic transition function: $F(state, input) \rightarrow state$
• 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.
• $F(state, input) \rightarrow (state, output)$
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

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

At any moment, machine exists in a “state”

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.

Client: What is your state?

My state is “2”
State machines

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

“actions” change the machine’s state
State machines

“actions” change the machine’s state

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

Client

What is the state?

Client

What is the state?
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.

Client

Apply action a.

Leader=L

Leader=L

Leader=L

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

1375: 

Γωνδα is the new cheese inspector

But Δικοστρα 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 Δικοστρα and Γωνδα 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 Δικοστρα the cheese inspector might read

2716: 8:30 15 Jan 72 — Δικοστρα is cheese inspector for 3 months

???
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.
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.

Figure 1: VR Architecture; the figure shows the configuration when $f = 1$. 
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 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.
- 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).
- 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.
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 $f = 1$. 
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.")
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 \textit{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 \textit{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.
How to know if a leader is “new”?

• The protocol runs as a sequence of **views** or **terms**.
  – VR: views. Raft: **terms**. Paxos: **ballots**.
• Views are numbered: monotonically increasing.
• During each view, there is at most one leader.
• Every message in the protocol carries the sender’s current view #. If the receiver has a higher view #, it ignores the message and responds with its view #.
• Leaders compete on who has the biggest view #.
• **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**
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.
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

---

**Server States**

**Follower**
- start
- discover current server or higher term
- “step down”

**Candidate**
- timeout, start election
- receive votes from majority of servers
- timeout, new election

**Leader**
- discover server with higher term
Liveness timeouts

• All consensus algorithms 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.

• Paxos: unspecified, but discussed in PMMC “pragmatics”.

• Raft: randomized timeouts to avoid contending leaders.
Log entry = index, term, command

Log stored on stable storage (disk); survives crashes

Entry **committed** if known to be stored on majority of servers
  - Durable, will eventually be executed by state machines
Log Consistency

High level of coherency 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

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

Leader changes can result in log inconsistencies:

Extraneous Entries

Missing Entries
Bringing a new leader up to date

- **Safety**: All majority-accepted operations must survive into future views.

- **Key idea**: since both commitment and leader election require a majority (f+1 votes), at least one server that votes for a leader saw every committed update.

- **Solution**:
  1. Piggyback history on leader votes.
  2. New leader learns history from this information.
  3. New leader pushes authoritative history to acceptors.
  4. Continue into new view.
Raft: leadership safety condition

• Raft does things a little differently.
• Elections choose the most up-to-date leader: one with a top log index (op-number) at least as high as all its voters.
• Candidates step down when they see a better (more up-to-date) candidate.
• So the new leader is one who has already seen and remembers at least all committed updates.
• Avoids state transfer from acceptors to leader.
• Drawbacks?
Picking the Best Leader

- Can’t tell which entries are committed!

- During elections, choose candidate with log most likely to contain 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”:
    \[(\text{lastTerm}_V > \text{lastTerm}_C) \lor (\text{lastTerm}_V == \text{lastTerm}_C) \land (\text{lastIndex}_V > \text{lastIndex}_C)\]
  - Leader will have “most complete” log among electing majority
Raft: safety

**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.
Summary

• 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 master?**
  – Master itself might fail or be isolated by a network partition.
  – Requires a scheme for “leader election” to choose a new master.
  – 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.
  – We will study a variant of Consensus with a simplified presentation, called Raft.
Quorum write

write
“accept”
commit
write
lagging accepts
Primary or coordinator backups
Quorum read

write

“accept”

read

stale results

read complete

fresh result

lagging accepts

Primary or coordinator backups

commit

write

complete
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.)
Google App Engine

[From Spark Plug to Drive Train: The Life of an App Engine Request, Along Levi, 5/27/09]
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.
How to Build a Highly Available System Using Consensus

Butler W. Lampson

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]
Nodes may compete to serve as leader, and may interrupt one another’s rounds. It can take many rounds to reach consensus.