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)

From http://paxos.systems
A service

Client

Web Server

App Server

DB Server

Store
Scaling a service

Add interchangeable server “bricks” to \textbf{partition} (“shard”) and/or \textbf{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 Eight Fallacies of Distributed Computing

1. The network is reliable.
2. Latency is zero.
3. Bandwidth is infinite.
4. The network is secure.
5. Topology doesn't change.
6. There is one administrator.
7. Transport cost is zero.
8. The network is homogeneous.
9. “Failures are independent.” - Chase

Big Trouble & Painful Learning

Essentially everyone, when they first build a distributed application, makes the following eight assumptions. All prove to be false in the long run and all cause big trouble and painful learning experiences.

Peter Deutsch
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
A lock service in the real world

A

\[ x = x + 1 \]

B

acquire

grant

acquire

???

B

???
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

x = x + 1

release

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

???

acquire

grant

release

B
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 \textit{guaranteed} in an asynchronous system in the presence of failures.
- \textbf{Intuition}: a “failed” process may just be slow, and can rise from the dead at exactly the wrong time.
- Consensus \textit{may} occur recognizably, rarely or often.

![Network partition](image1.png)

![Split brain](image2.png)
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.
choose two

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).

“CAP theorem”
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”

C: consistency
A: availability
P: partition-resilience

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).

“choose two”

Dr. Eric Brewer
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?**
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
strict and was rejecting perfectly good cheese. Parliament then replaced him by passing the decree

1375: Γωνδα is the new cheese inspector

But Δικστρα 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 Δικστρα 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 Δικστρα the cheese inspector might read

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

???
A Paxos Round

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

• 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.
  – Paxos is **safe** but not **live**: in the worst case (multiple repeated failures) it might not terminate.
  – But in practice Paxos will get the job done…if it can be done.
  – We will study a variant of Paxos with a simplified presentation, called Raft.
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
State machines

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

“actions” change the machine’s state
State machines

“actions” change the machine’s state

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?

Client
What is the state?
Replicated state machines

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

Client

Apply action a.

Client

Apply action b.

Apply action c.

Client
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.
Key elements of consensus

• Leader election
  • Who is in charge?

• Log replication
  • What are the actions and what is their order?

• Safety
  • What is true for all states, in all executions (including failures)?
  • e.g., either we haven’t agreed or we all agree on the same value
Quorum write

write

“accept”

commit

write

lagging accepts

Primary or coordinator

backups
Quorum read

- Write
- "Accept"
- Read
- Stale results
- Fresh result
- Lagging accepts
- Primary or coordinator
- Backups

Diagrams illustrate the process of committing and reading with quorum read operations.
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]
Coordination services

• Build your cloud apps around a coordination service with consensus (Paxos) at its core.
• This service is a fundamental building block for consistent scalable services.
  – Chubby (Google)
  – Zookeeper (Yahoo!)
  – Centrifuge (Microsoft)