# Stumbling over consensus research: Misunderstandings and issues

Unfortunately, consensus has two commonly adopted interfaces.

- The p-interface used by the Paxos algorithm.
- The r-interface used by all other consensus algorithms, including algorithms based on randomization, partial synchrony or failure detection.

With the r-interface, all non-faulty processes initially propose a value, while with the p-interface, any positive number of non-faulty processes initially propose a value.

# Stumbling over consensus research: Misunderstandings and issues

The reason for having two interfaces is historic.

- The r-interface appeared as a variation of the interactive-consistency problem, in which every process starts with some initial value.
- The p-interface was later proposed as an interface more directly applicable to the state machine approach.

Aguilera M.K. (2010) .

*Replication - Lecture Notes in Computer Science*, Vol. 5959, pp. 1-15.

# The Part-Time Parliament

Lamport, L. (1998)
*ACM Transactions on Computer Systems*
Vol. 16, No. 2, pp. 133-169.

Technical Report SRC-DEC, September 1989

# Paxos Parliament Protocol

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 Paxos parliament's protocol provides a new way of implementing the state machine approach to the design of distributed systems.

# Replicated State Machine

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.

The Paxos algorithm is the most fault-tolerant way to get consensus without real-time guarantees.

# Paxos Parliament Protocol

- (Single-decree) Synod protocol — Basic Paxos
- (Multi-decree) Parliamentary protocol — Complete Multi Paxos

  - citizen ↔ client program
  - legislator ↔ server
  - decree ↔ command
  - ledger ↔ list of commands
  - law book ↔ SM state

  - president ↔ leader / primary

# Greek Friends

- Λινχδ: **Lynch** - Legislator
- Φισδερ: **Fischer** - Legislator
- Τωυεγ: **Toueg** - Legislator
- Ωκι: **Oki** - Legislator
- Στωκμε˘ιρ: **Stockmeyer** - Legislator

- Λισκωφ: **Liskov** - Merchant
- Σκεεν: **Skeen** - Merchant

- Λαμπσων: **Lampson** - General
- Γρα˘ι: **Gray** - Priest

- Δ˘ικστρα: **Dijkstra** - Cheese inspector

- Δφωρκ: **Dwork** - President
- Σδν˘ιδερ: **Schneider** - Citizen

# The Paxos Algorithm or How to Win a Turing Award

Leslie Lamport

*SPTDC Talks 2019*

https://www.youtube.com/watch?v=tw3gsBms-f8

# The Problem Solved by Paxos

- Multiple clients can send requests to a system.
- The system must choose in what order to handle them.
- The system is implemented by multiple computers.
- They must choose a single ordering even if some computers fail.

Paxos solves this problem by running multiple solutions to the consensus problem.

# The Consensus Problem

- Multiple clients can each send a request to a system.
- The system must choose which one to handle next.
- The system is implemented by multiple computers.
- They must choose a single request even if some computers fail.

# Implementation Details X

- Paxos is efficient because it simultaneously executes the first part (phase) of all the consensus solutions.
- But we're interested in why Paxos is correct, not why it's efficient.

- So, we abstract away such implementation details.

# Abstraction

- We are going to abstract the consensus problem to make it simpler, easier to understand.
- Abstraction, abstraction, abstraction.
- That is how you win a Turing award.

# The Consensus Problem

- Multiple clients can each send a request to a system.
- The system must choose which one to handle next.
- The system is implemented by multiple computers.
- They must choose a single request even if some computers fail.

# A More Abstract Problem

- Forget about clients.
- The system must choose which one to handle next.
- The system is implemented by multiple computers.
- They must choose a single request even if some computers fail.

# A More Abstract Problem

- The system must choose which one to handle next.
- The system is implemented by multiple computers.
- They must choose a single request even if some computers fail.
- The computers may choose any request in the set *Value*.

# A More Abstract Problem

- The system must choose which one to handle next.
- The system is implemented by multiple computers.
- They must choose a single value even if some computers fail.
- The computers may choose any value in the set *Value*.

# Paxos Made Simple

Lamport, L. (2001)

*ACM SIGACT News (Distributed Computing Column)*
Vol. 32, No. 4, pp. 51-58.

# The Problem

Assume a collection of processes that can propose values.

A consensus algorithm ensures that a single one among the proposed values is chosen.

If no value is proposed, then no value should be chosen.

If a value has been chosen, then processes should be able to learn the chosen value.

# Requirements

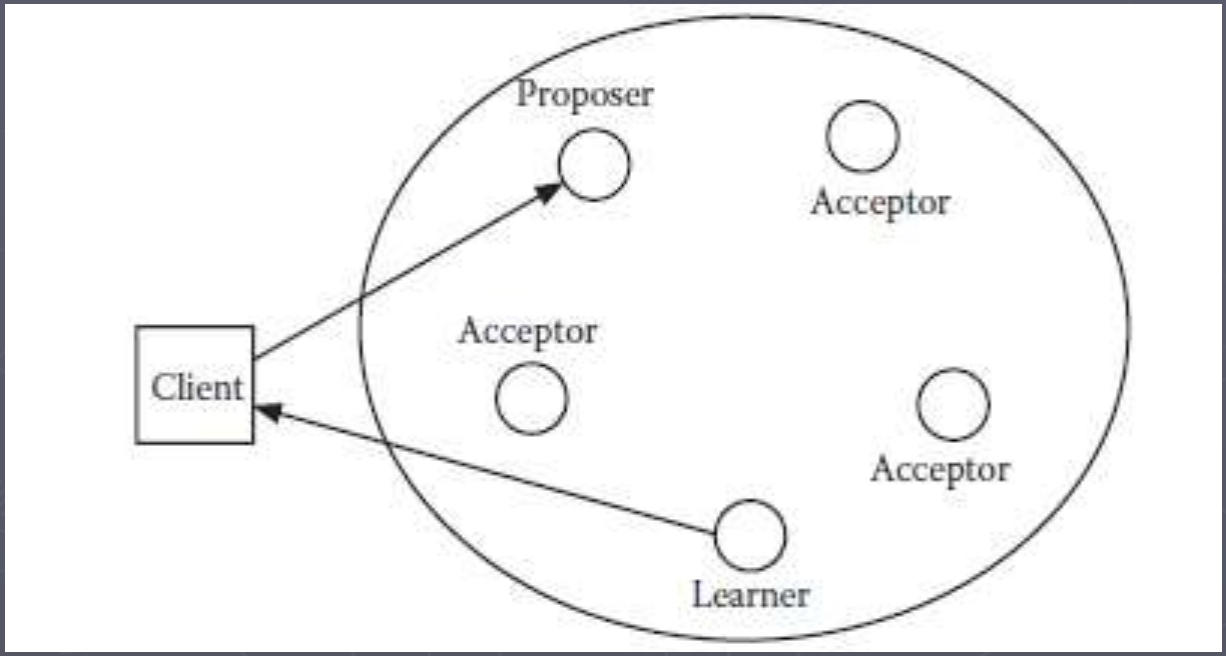
- Safety
  1. Only a value that has been proposed may be chosen.
  2. Only a single value is chosen.
  3. A node never learns that a value has been chosen unless it actually has been.

- Liveness
  - Some proposed value is eventually chosen.
  - If a value has been chosen, then a process can eventually learn the value.

# Classes of Agents

- Client: issues a request and waits for a response.
- Acceptor (Voters): Acceptors are collected into groups called quorums.
- Proposer: Proposer advocates a client request and tries to convince the members of a quorum to agree on it; acts as a coordinator.
- Learner: A learner acts on the request once a client request has been agreed upon by the acceptors.

# Classes of Agents




In an implementation, each process plays the role of proposer, acceptor, and learner.

# System Model

- Agents can communicate with one another by sending messages.
- Asynchronous, non-byzantine model:
  - Agents operate at arbitrary speed, may fail by stopping, and may restart.
  - Messages can take arbitrarily long to be delivered, can be duplicated, and can be lost, but they are not corrupted.
- Since all agents may fail after a value is chosen and then restart, some information must be remembered by an agent that has failed and restarted.

# Choosing a Value

- The algorithm operates in two phases
  - Phase 1: *Prepare* request → ← response
  - Phase 2: *Accept* request → ← response

# Phase 1: Prepare

(1) A proposer selects a (new unique) proposal number n, and sends a *prepare* request with number n ("prepare", n) to a majority of acceptors:

(a) Can I make a proposal with number n?

(b) If yes, do you suggest some value for my proposal?

# Phase 1: Prepare

(2) If an acceptor receives a *prepare* request ("prepare", n) with number n greater than that of any *prepare* request to which it has already responded, it sends out ("ack", n, n', v') or ("ack", n, $\perp$ , $\perp$)

(a) responds with a promise not to accept any more proposals numbered less than n.

(b) suggest the value v of the highest-number proposal that it has **accepted** if any, else $\perp$.

# Phase 2: Accept

(3) If the proposer receives a response to its *prepare* requests (numbered n) from a majority of acceptors, then it sends an *accept* request (“accept”, n , v) to each of those acceptors for a proposal numbered n with a value v:

(a) n is the number that appears in the *prepare* request.

(b) v is the value of the highest-numbered proposal among the responses or is any value if the responses reported no proposals.

# Phase 2: Accept

(4) If the acceptor receives an *accept* request (“accept”, n , v), it accepts the proposal unless it has already responded to a *prepare* request having a number greater than n.

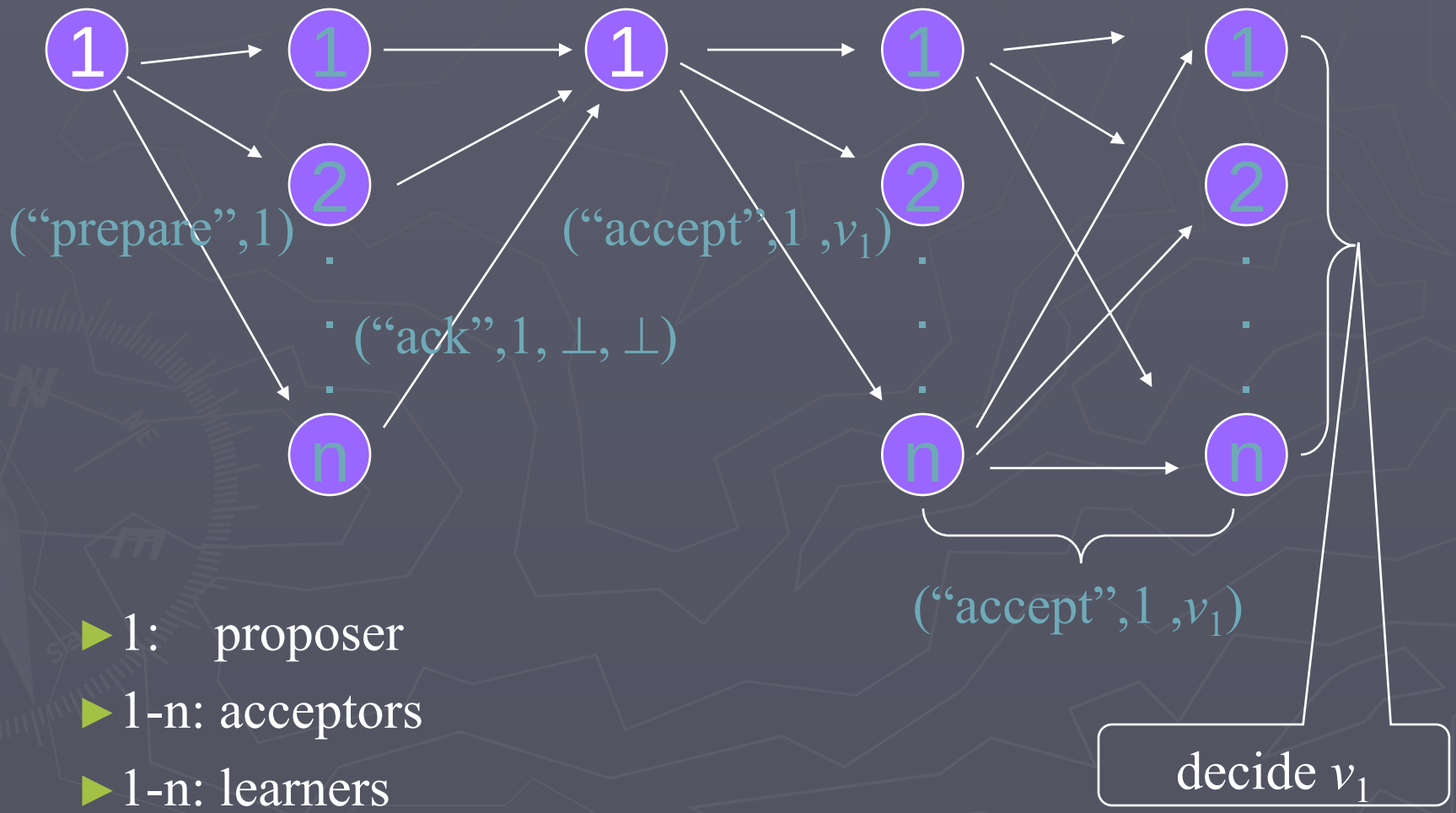
# Learning a Chosen Value

To learn that a value has been chosen, a learner must find out that a proposal has been accepted by a majority of acceptors.

First option:

- The obvious algorithm is to have each acceptor, whenever it accepts a proposal, respond to all learners, sending them the request (“accept”, n , v).

# In Well-Behaved Runs




- 1: proposer
- 1-n: acceptors
- 1-n: learners

# Learning a Chosen Value

To learn that a value has been chosen, a learner must find out that a proposal has been accepted by a majority of acceptors.

Second option:

- We can have the acceptors respond with their acceptances to a **distinguished learner**, which in turn informs the other learners when a value has been chosen.
- This approach requires a **third phase** for all the learners to discover the chosen value.

# Liveness

To guarantee progress, a **distinguished proposer** must be selected as the only one to try issuing proposals.

- If the distinguished proposer can communicate successfully with a majority of acceptors, and if it uses a proposal with number greater than any already used, then it will succeed in issuing a proposal that is accepted.
- By abandoning a proposal and trying again if it learns about some request with a higher proposal number, the distinguished proposer will eventually choose a high enough proposal number.

# The Implementation

Stable storage, preserved during failures, is used to maintain the information that agents must remember.

- An acceptor records its intended response before actually sending the response.
- A proposer remembers the highest-numbered proposal it has tried to issue and begins a new phase 1 with a higher proposal number than any it has already used.

# The Implementation

Different proposers choose their numbers from disjoint sets of numbers, so two different proposers never issue a proposal with the same number.

The algorithm chooses a **leader**, which plays the roles of the distinguished proposer and the distinguished learner.

# Implementing a State Machine

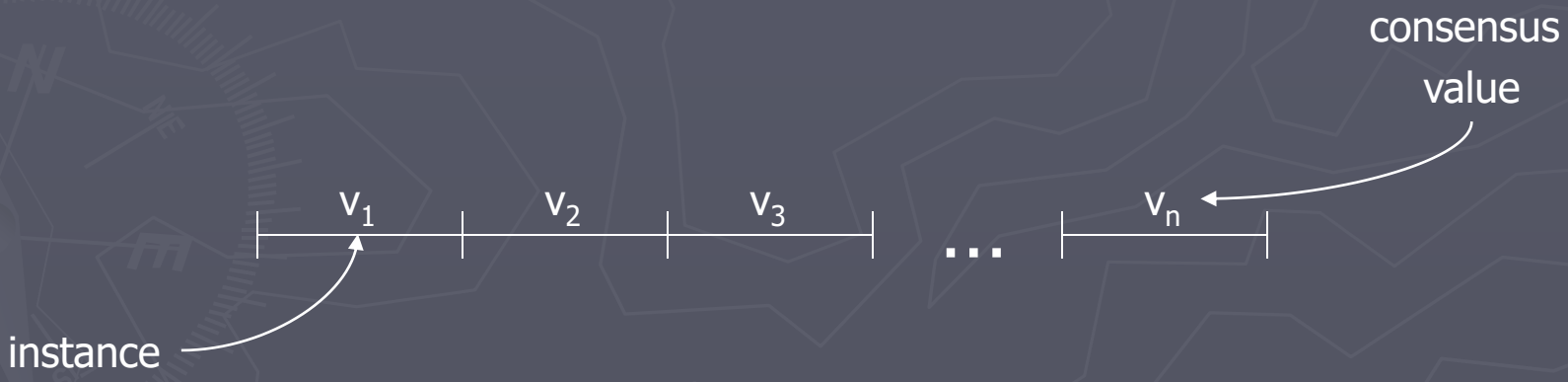
A simple way to implement a distributed system is as a collection of clients that issue commands to a central server.

An implementation that uses a single central server fails if that server fails.

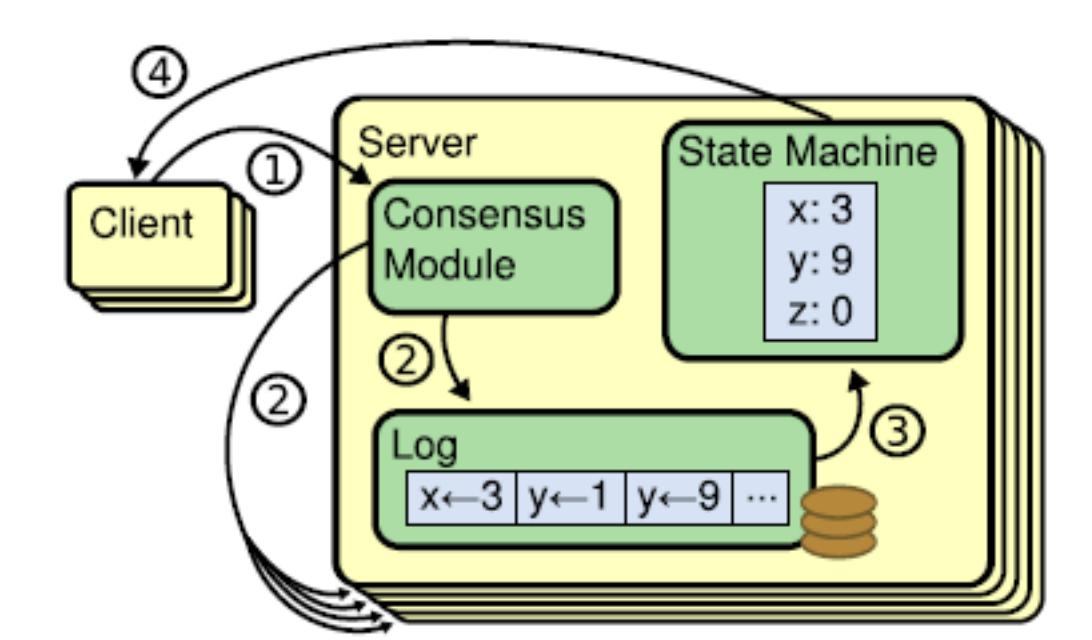
We therefore instead use a collection of servers, each one independently implementing the state machine.

# Implementing a State Machine

To guarantee that all servers execute the same sequence of state machine commands, we implement a sequence of separate instances of the Paxos consensus algorithm, the value chosen by the ith instance being the ith state machine command in the sequence.



# Implementing a State Machine



# Implementing a State Machine

In normal operation, a single server is elected to be the leader, which acts as the distinguished proposer (the only one that tries to issue proposals) in all instances of the consensus algorithm.

Clients send commands to the leader, who decides where in the sequence each command should appear.

# Implementing a State Machine

To increase bandwidth, the leader can run several Paxos instances in parallel.

That is, the leader can propose commands i+1 through i+$\alpha$, after commands 1 through i are chosen.

# Implementing a State Machine

A gap of up to $\alpha$-1 commands could arise if the command i+$\alpha$ is chosen but all the messages of previous commands are lost and then the leader fails.

# Implementing a State Machine

A newly elected leader will fill the gap executing phase 1 of the missing instances.

- The outcome of these executions may determine the value to be proposed in some instances but leave the proposed value unconstrained in all other instances.
- For the unconstrained instances, the leader selects a special “noop” command that leaves the state unchanged.

The leader then executes phase 2 of the missing instances.

# (Basic) Paxos

Determines the authoritative value for a single variable.

Several proposers may offer different values to set the variable to.

The system converges on a single agreed-upon decision (chosen) value.

# (Complete) Paxos

Multi-Paxos streamlines a series of consensus decisions.

Phase 1 is done once by a new leader.

Phase 2 is repeated multiple times by the same leader.

# Safety and Liveness

Safety must be preserved at all times, and hence, its implementation must not rely on synchrony assumptions.

Liveness, on the other hand, may be hampered during periods of instability, but eventually, when the system resumes normal behavior, progress should be guaranteed.

In Paxos, liveness hinges on a separate (unspecified) leader election algorithm; but safety is ensured regardless of the success or failure of the election.

# Paxos Made moderately Complex

van Renesse, R., and Altinbuken, D. (2015)
*ACM Computing Surveys*,
Vol. 47, No. 3, Article 42.

# State Machine Replication

- SMR is a technique to mask failures, particularly crash failures.

- A collection of *replicas* of a deterministic state machine are created.
- The replicas are then provided with the same sequence of operations, so they go through the same sequence of state transitions and end up in the same state and produce the same sequence of outputs.
- It is assumed that at least one replica never crashes, but we do not know a priori which replica this is.

# Types of processes in Paxos

| Process Type | Description | Minimum Number |
|---|---|---|
| Replica | Maintains application state<br>Receives requests from clients<br>Asks leaders to serialize the requests so all replicas see the same sequence<br>Applies serialized requests to the application state<br>Responds to clients | $f+1$ |
| Leader | Receives requests from replicas<br>Serializes requests and responds to replicas | $f+1$ |
| Acceptor | Maintains the fault tolerant memory of Paxos | $2f+1$ |

*Note:* $f$ is the number of failures tolerated.

- Replica — Learner / SM
- Leader — Proposer

# Clients and commands

- When a client wants to execute a command c, its stub routine broadcasts a message to all replicas and waits for a response from one of the replicas.

- A command is a triple (*κ, cid, operation)*, where *κ* is the identifier of the client that issued the command and *cid* is a client-local unique command identifier (e.g., a sequence number).

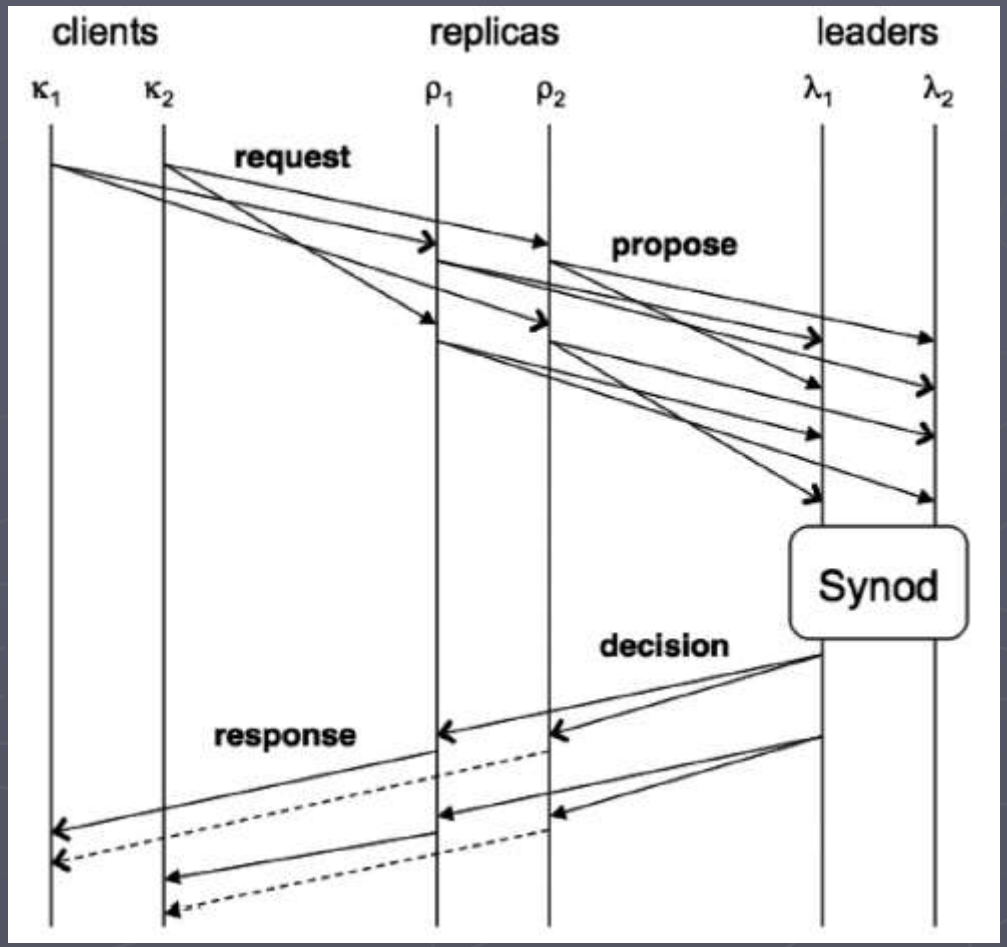
# Replicas and slots

- The replicas can be thought of as having a sequence of *slots* that need to be filled with commands that make up the inputs to the state machine.
- Each slot is indexed by a *slot number*.
- Replicas receive requests from clients and assign them to specific slots, creating a sequence of commands.
- A replica, on receipt of a request, *c* message, proposes command *c* for its lowest unused slot.

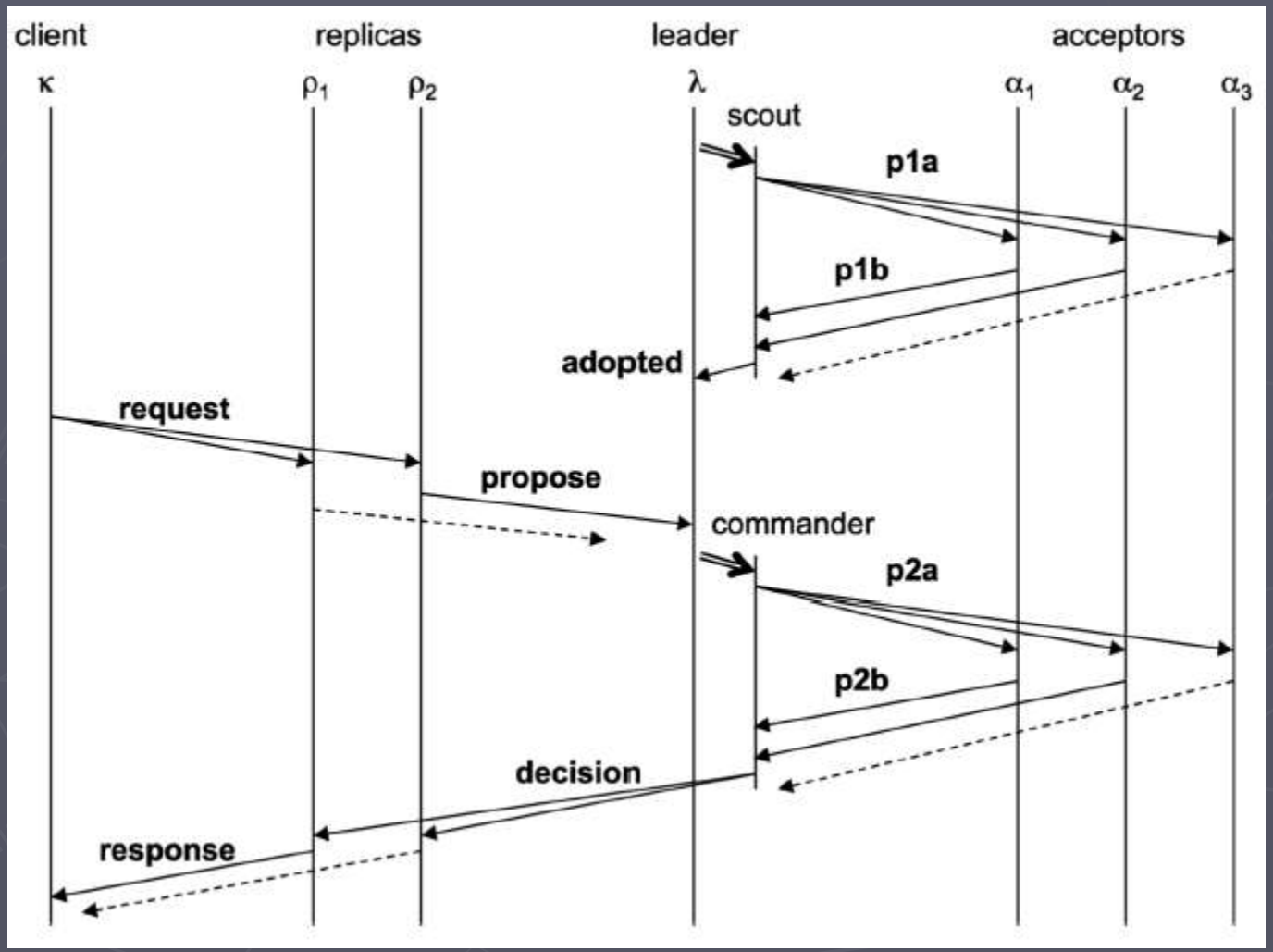
# Leaders and decisions

- Leaders receive proposed commands from replicas and are responsible for *deciding* a single command for the slot.
- A replica awaits the decision before actually updating its state and computing a  response to send back to the client that issued the request.

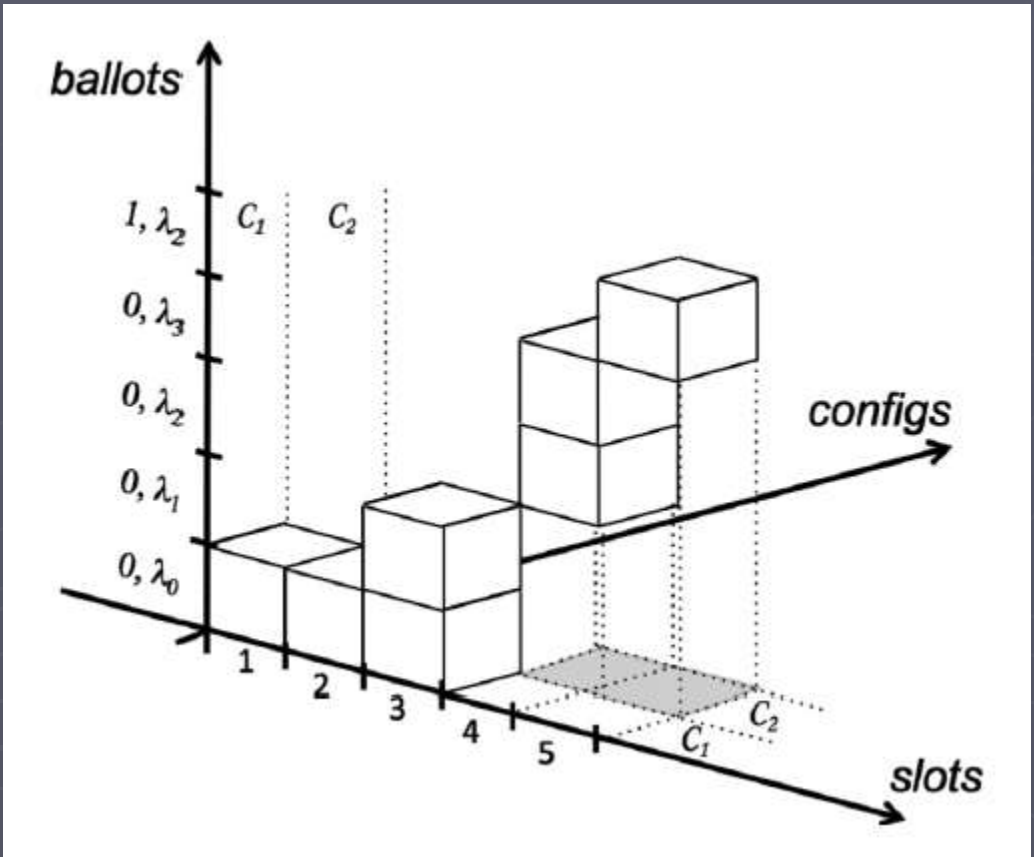
# Clients, replicas and leaders



# Acceptors


client
replicas
leader
acceptors
κ
ρ1
ρ2
λ
α1
α2
α3
scout
p1a
p1b
adopted
request
propose
commander
p2a
p2b
decision
response

# Ballots, slots and configurations



# How to Build a Highly Available System Using Consensus

Lampson, B. (1996)

*Distributed Algorithms – Lecture Notes in Computer Science,* Vol. 1151, pp. 1–17.

# Essential properties

Paxos is run by a set of leader processes that guide a set of agent processes to achieve consensus (decide on an input value).

Paxos is safe (partially correct) no matter how many simultaneous leaders there are and no matter how often leader or agent processes fail and recover, how slow they are, or how many messages are lost, delayed, or duplicated.

# Essential properties

It terminates if there is a single leader for a long enough time during which the leader can talk to a majority of the agent processes twice.

It may not terminate if there are always too many leaders (fortunate, since we know that guaranteed termination is impossible).

# Essential properties

To get a complete consensus algorithm we combine this with a sloppy timeout-based algorithm for choosing a single leader.

If the sloppy algorithm leaves us with no leader or more than one leader for a time, the consensus algorithm may not terminate during that time.

But if the sloppy algorithm ever produces a single leader for long enough the algorithm will terminate, no matter how messy things were earlier.

# The framework

There is a set of agent processes. The behavior of an agent is deterministic; an agent does what it's told. An agent has 'persistent' storage that survives crashes. The set of agents is fixed for a single run of the algorithm (though it can be changed using the Paxos algorithm itself).

There are also some leader processes that tell the agents what to do. Leaders can come and go freely, they are not deterministic, and they have no persistent storage.

# The idea

The key idea of Paxos comes from a non-fault-tolerant majority consensus algorithm. That algorithm gets into trouble if the agents can't agree on a majority, or if some members of the majority fail so that the rest are unsure whether consensus was reached.

To fix this problem, Paxos has a sequence of rounds. Round n has a single leader who tries to get a majority for a single value $v_n$. If one round gets into trouble, another one can make a fresh start. If round n achieves a majority for $v_n$ then $v_n$ is the outcome.

# The idea

Clearly for this to work, any two rounds that achieve a majority must have the same value. The tricky part of the Paxos algorithm is to ensure this property.

In each round the leader

- queries the agents to learn their status for past rounds,
- chooses a value and commands the agents, trying to get a majority to accept it, and
- if successful, distributes the value as the outcome to everyone.

# The ABCD's of Paxos

Lampson, B. (2001)

*Twentieth Annual ACM Symposium on Principles of Distributed Computing (PODC),* New Port, Rhode Island, 16 p.

# The idea of Paxos

The simplest form of consensus decides when a majority of agents choose the same value.

This is not very fault-tolerant for two reasons: there may never be a majority, and even when there is, it may remain permanently invisible if some of its agents stop.

Since we can't distinguish a stopped agent from a slow one, we can't tell whether the invisible majority will reappear, so we can't ignore it.

# The idea of Paxos

To avoid these problems, Paxos uses a sequence of views.

A majority in any view decides (or more generally, a decision quorum), but if a view doesn't work out, a later view can supersede it.

This makes the algorithm fault-tolerant but introduces a new problem: decisions in all views must agree.

# The idea of Paxos

The key idea of Paxos is that a later view *v* need not know that an earlier view decided in order to agree with it.

Instead, it's enough to classify each earlier view *u* into one of two buckets: either it can *never* decide, in which case we say that it's *out*, or it has made a *choice* and it must decide for that choice if it decides at all.

In the latter case, *v* just needs to know *u*'s choice.

# The idea of Paxos

Thus, a view chooses and then decides. The choice can be superseded, but the decision cannot.

On the other hand, the choice must be visible unless the view is visibly out, but the decision need not be visible because we can run another view to get a visible decision.

This separation between decision and visibility is the heart of the algorithm.

# The idea of Paxos

A decision will be unique as long as every later choice agrees with it. We ensure this by anchoring the choice: if all previous views are out, *v* can choose any input value; if not, it can take the choice of the latest previous view that isn't known to be out.

By induction, this ensures that *v* will agree with any previous decision.

# The idea of Paxos

To keep from blocking the algorithm, we must be able to make each previous view visibly out unless it has a visible choice.

# The idea of Paxos

A sequence of *views*; get a decision quorum in one of them.

Each view *v chooses* an *anchored* value (equals any possible earlier decision).

If a quorum *accepts* the choice, decision!

- Decision is irrevocable, may be invisible, but is any later view's choice.
- Choice is changeable, must be visible.

# Additional Notes

**Basic Paxos** assumes a static membership, and it uses an elected leader to coordinate the **agreement protocol**. It runs on a completely connected network of n processes and tolerates up to f failures, where $n \geq 2f+1$.

**Multi Paxos** is a **state machine replication protocol** that allows a set of distributed servers, exchanging messages via asynchronous communication, to totally order client requests in the benign-fault, crash-recovery model.

The idea of Paxos is to have a sequence of views until one of them forms a write quorum, consisting of a majority of agents, that is noticed. So, each view has three phases:

- Choose an input value that is anchored: guaranteed to be the same as any previous decision (phase 1).
- Try to get a decision quorum of agents to accept the value (phase 2).
- If successful, finish the algorithm by recording the decision at the agents (phase 3).

It is possible for multiple proposers to submit proposals with increasing values of sequence numbers in such a manner that none of those are ever accepted.

To guarantee progress, a distinguished proposer and distinguished learner (**coordinator**) must be selected as the only one allowed to issuing proposals (start off new rounds).

In this way, no conflict occurs and if the distinguished proposer can communicate successfully with a majority of acceptors, and if it uses a proposal with number greater than any already used, then it will succeed in issuing a proposal that is accepted (chosen value).

By abandoning a proposal and trying again if it learns about some request with a higher proposal number, the distinguished proposer will eventually choose a high enough proposal number.

If the leader crashes or becomes unreachable, the other servers elect a new leader; a view change occurs, allowing progress to safely resume in the new view under the reign of the new leader.

We can assume that the processes have access to a **failure detector Ω** that indicates who is the leader.

At each process p, the module Ω outputs a single process that is currently considered the leader by process p. Ω essentially implements an eventual leader election, where all non-faulty processes eventually trust the same non-faulty process as the leader.

Proposer p considers itself the coordinator if its Ω module outputs p. Eventually, all the proposers will agree and therefore the system is guaranteed to progress.

Multi-Paxos streamlines a series of consensus decisions. Protocol becomes a 2-phase commit with a 3-phase commit when leader fails.

A new leader p carries out the prepare phase once for all its proposals. After the completion of the prepare phase, p carries out only the accept phase for each proposal until a new leader emerges by initiating a new prepare phase. To achieve this, **the instance number "i" is included along with each new view *v***.

In the prepare phase, p sends a prepare message to all processes to declare the ballot number it uses for its proposals, learns about all the existing proposals, and requests promises that no smaller ballot numbers be accepted afterwards. The prepare phase is completed once p receives acknowledgments from n−f processes.

Once the prepare phase is completed, to have a proposal committed, leader p initiates the accept phase by sending an accept message to all processes with the proposal and the ballot number it declares in the prepare phase. The proposal is committed when p receives acknowledgments from f+1 processes.

Whenever a higher ballot number is encountered in the prepare phase or the accept phase, the leader has to initiate a new prepare phase with an even higher ballot number.

This could happen if there are other processes acting as leaders, unavoidable in an asynchronous system.

We make the following observations:

1) After an initial prepare phase, in order for a leader p to make timely progress, it suffices for p to obtain timely responses for its accept message from any set of f+1 processes (or f processes besides itself). The set could change for different accept messages.
2) Any leader change incurs the cost of an extra round of communication for the prepare phase.

Paxos allows any server to be leader provided it then updates its command log to ensure it is up-to-date.

# Paxos made live: An engineering perspective

Chandra, T., Griesemer, R. and Redstone, J. (2007) *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing (PODC),* Portland, Oregon, pp. 398-407.