Fault Tolerance: Consensus

Distributed Systems

Agenda

Today

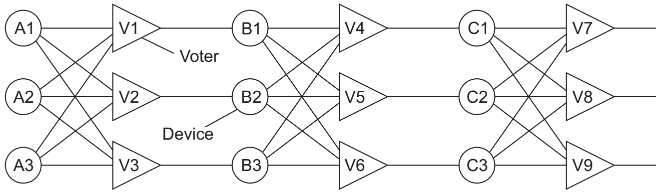
- Paxos
- How to design a fault-tolerant distributed algorithm?
  - Which algorithm? Why, Totally Ordered Multicast, ofcourse!

Redundancy for failure masking

Types of redundancy

- **Information redundancy**: Add extra bits to data units so that errors can recovered when bits are garbled.
- **Time redundancy**: Design a system such that an action can be performed again if anything went wrong. Typically used when faults are transient or intermittent.
- **Physical redundancy**: add equipment or processes in order to allow one or more components to fail. This type is extensively used in distributed systems.

Triple Modular Redundancy

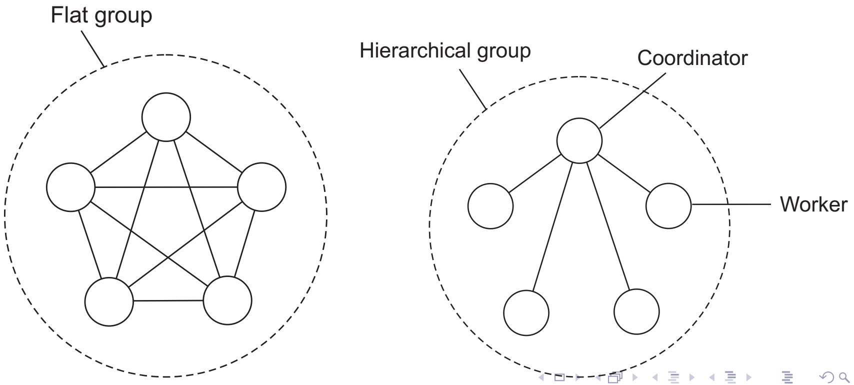


Often used in safety-critical systems such as avionics

Process resilience

Basic idea

Protect against malfunctioning processes through **process replication**, organizing multiple processes into **process group**. Distinguish between **flat groups** and **hierarchical groups**.



Groups and failure masking

k-fault tolerant group

When a group can mask any k concurrent member failures (k is called **degree of fault tolerance**).

How large does a k-fault tolerant group need to be?

- With **halting failures** (crash/omission/timing failures): we need a total of $k+1$ members as **no member will produce an incorrect result, so the result of one member is good enough**.
- With **arbitrary failures**: we need $2k+1$ members so that the correct result can be obtained through a majority vote.

Important assumptions

- All members are identical
- **All members process commands in the same order**

State Machine Replication: We can now be sure that all processes do exactly the same thing.

Consensus

In a fault-tolerant process group, each non-faulty process executes the same commands, and in the same order, as every other nonfaulty process.

Reformulation

Nonfaulty group members need to reach **consensus** on which command to execute next.

- **Termination**: All non-faulty processes must eventually decide on a value
- **Agreement**: All non-faulty processes agreee on same value
- **Validity**: Agreed upon value must be the same as the initial proposed "source" value

Totally Ordered Multicast

- Applicable IFF no failures
- How to handle missing acknowledgements?

FLP Consensus Impossibility

Fisher,Lynch, and Patterson—1985

- If we assume totally *asynchronous* system model
- And if failures are fail-stop
- Then it is impossible to have a deterministic consensus protocol

Asynchronous: no assumptions about process execution speeds or message delivery times

PAXOS

Realistic Consensus: Paxos

Assumptions (rather weak ones, and realistic)

- A **partially synchronous** system (in fact, it may even be asynchronous).
- **Communication** between processes may be **unreliable**: messages may be lost, duplicated, or reordered.
- **Corrupted message can be detected** (and thus subsequently ignored).
- All **operations are deterministic**: once an execution is started, it is known exactly what it will do.
- Processes may exhibit **crash failures**, but **not arbitrary failures**.
- Processes **do not collude**.

Checksums

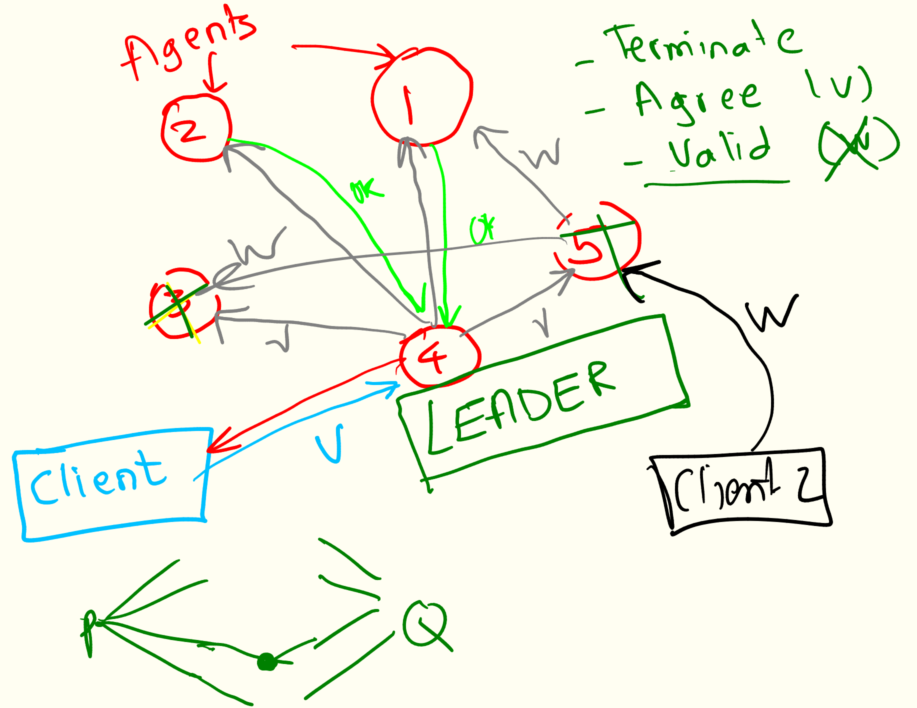
No Byzantine Failures

Essence of Paxos

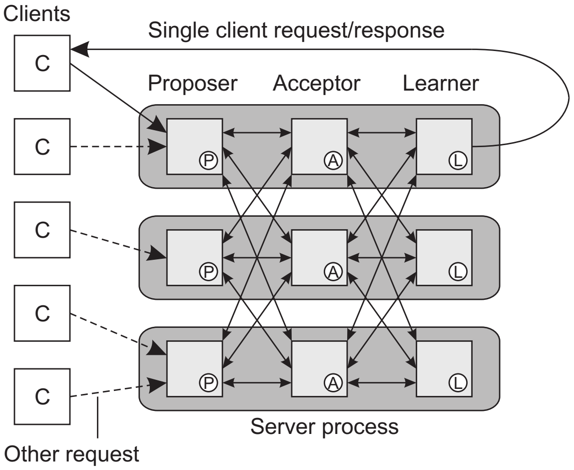
- Out of N nodes, some (ideally, one) act as a leader
- Leader presents the consensus value to the *acceptors*, counts the ballots for acceptance of the majority, and notifies acceptors of success
- Paxos can mask failure of a minority of N nodes
- Agent processes have persistent storage that survives crashes
- Leaders have no persistent storage

Why majority consensus is required

- Assume two concurrent leaders P and Q
- If P and Q receive $\lceil n/2 \rceil + 1$ acks, at least one process must be common



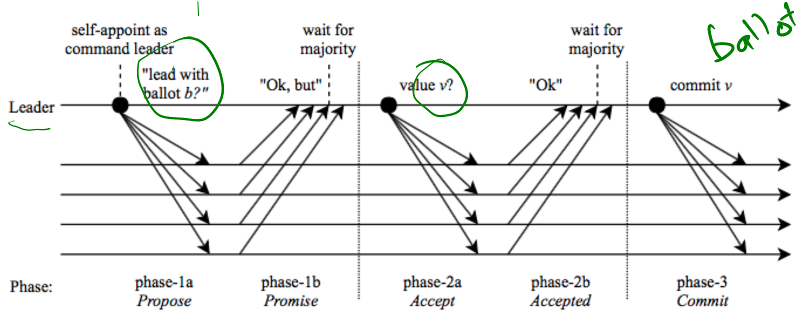
Paxos Components



Rounds and Ballots

- Paxos proceeds in rounds. Each round has three phases.
- **Each round has uniquely numbered ballot** [ballot-id. Totally ordered]
- If no failures, then consensus reached in one round
- Any would-be leader can start a new round on any (apparent) failure
- Consensus is reached when some leader successfully completes a round

Paxos Phases

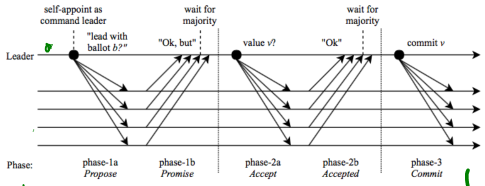


Phase 1: Leader election

1. Would-be leader chooses unique ballot ID (round #)
2. Proposes "Can I lead?"
3. Other processes return highest ballot ID seen so far. Can only lead if these are smaller than ballot ID proposed.
4. If majority respond, and no one knows of a higher ballot number, then you are the leader for this round.

Also called the "Prepare" phase.

Else → Terminate the round.

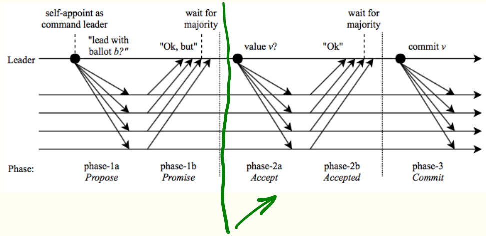


Round

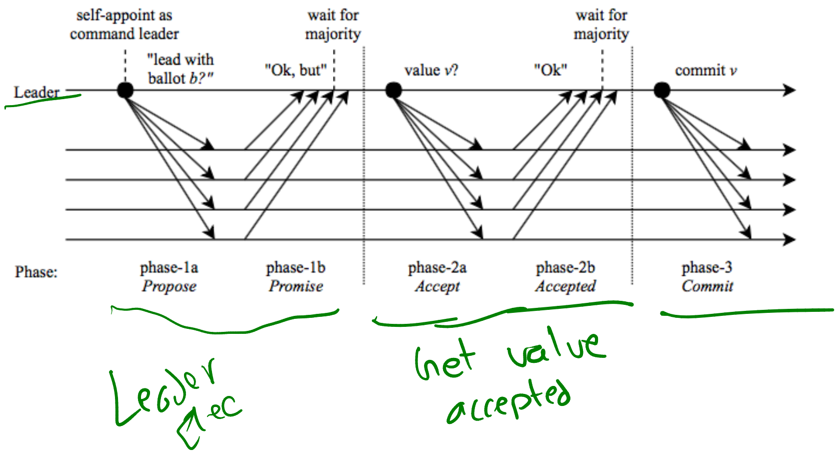
Phases 2–3: Leading a round

- Choose “suitable value” v for this ballot/round
- Ask agents to accept value
- If majority respond and agree, then tell everyone the round succeeded.
- Else, move on, and ask for another round

Assume for now (1) v is client supplied

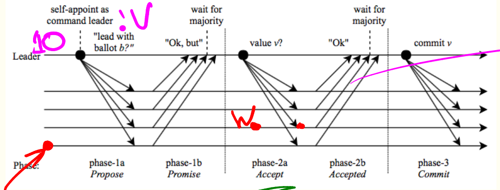


Paxos Phases

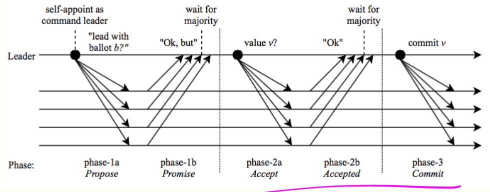


Choosing a suitable value

- Assume a majority of agents responded
- If no agent accepted a value from some previous round/ballot, then can choose any value leader wants
- Else, they tell you ballot ID and value. Find most recent value that any corresponding agent accepted, and choose it for this ballot too.



Distributed Algorithm



Persistent State of acceptors

n_p: Highest prepare seen [Phase 1]
n_a, v_a: Highest accept seen [phase 2]

Proposer

While not decided:

1. Choose unique ballot number n
2. Send prepare(n) to all servers including self
3. If promise(n, n_a, v_a) from majority:
4. $v' = v_a$ with highest n_a. Otherwise choose own v
5. Send accept(n, v') to all
6. If accept_ok(n) from majority, send decided(v') to all

phase 2

phase

IF max$\{n_a\} < n$, then choose v

Algorithm for Acceptors

Persistent State

n_p: Highest prepare seen

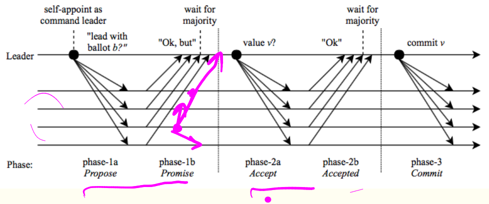
n_a, v_a: Highest accept seen

Handling Prepare Messages

1. If $n > n_p$:
2. $\quad n_p = n$; reply promise(n, n_a, v_a)
3. Else, reply prepare_reject

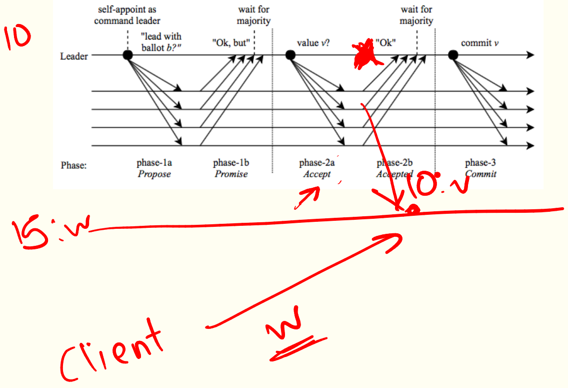
Handling accept messages

1. If $n >= n_p$:
2. $\quad n_p = n$; $n_a = n$; $v_a = v$
3. $\quad$ reply accept_ok(n)
4. Else, reply accept_reject



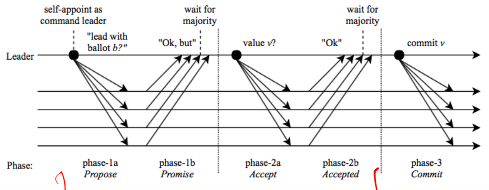
Anchoring a value

- A round "anchors" if majority of agents hear the Accept command and obey
- The round may then fail if many agents fail, many command messages are lost, or if another leader usurps.
- Safety: Once a round anchors, no subsequent round can change it
- System may have another round, possibly with different leader, until all nodes learn of the success.
- Reminder: Agents read persistent log after crash restarts



Paxos Properties

- Run by a set of leader processes that guide a set of agent processes
- It is correct no matter how many simultaenous leaders there are
- It is correct no matter how often processes fail/recover, their speeds, message losses/delays/duplicated
- Terminates if there is a single leader for long enough time during which the leader can talk to majority of processes twice
- It may not terminate if there are always too many leaders



Why Paxos Works

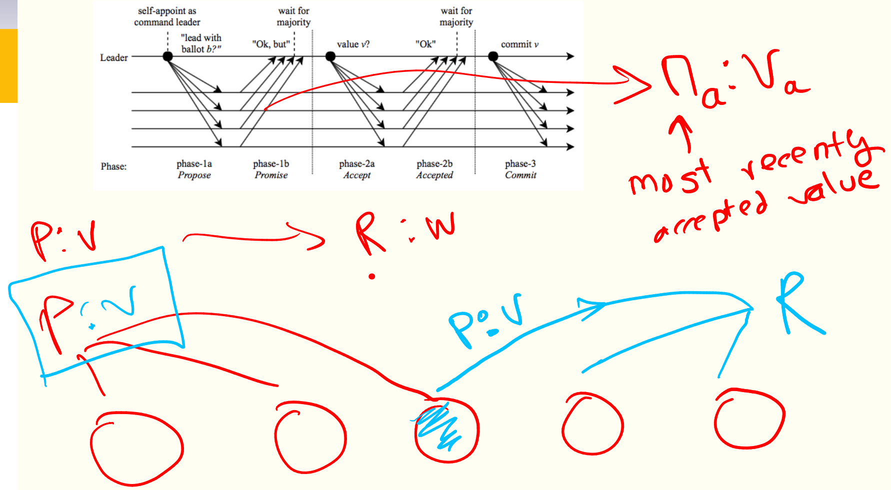
Key invariant

If some round commits, then any subsequent round chooses the same value, or it fails

- Leader L or round R that follows a successful round P with value v.
- Either L learns of (P,v), or R fails
- P got responses from majority. If R does too, then some agent responds to both.
- If L does learn of (P,v), then L must choose v as the suitable value

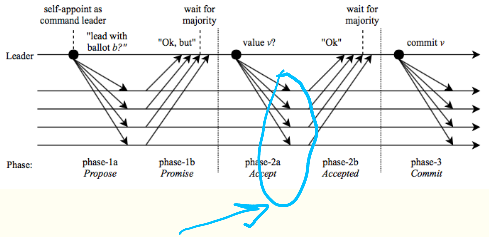
Algo terminates once a value is committed

— Agents dont store ballot id of committed only of the accepted.



Anchoring and agreement

- Once a value is decided, the decision is final and no different value can be chosen
- Agreement if $\lfloor n/2 \rfloor + 1$ acceptors out of n are up and able to communicate
- Acceptors broadcast agreement to Learners, and learners must acknowledge!
- Acceptors check if learned value matches their stored agreement value



TOM vs Paxos

- Totally Ordered Multicast with no failures gives consensus
- With failures, cannot afford to wait for all responses
- Hence can have multiple leaders in Paxos
- Fault-tolerant version of TOM : "atomic multicast"
- Atomic multicast is equivalent to consensus
- Used in ZooKeeper (ZAB: Zookeeper Atomic Broadcast)

Paxos Simulation Scenarios

1. Simple case: 1 leader
2. 2 leaders
3. Acceptor failure in phase 1
4. Acceptor failure in phase 2
5. Leader fails after phase 1

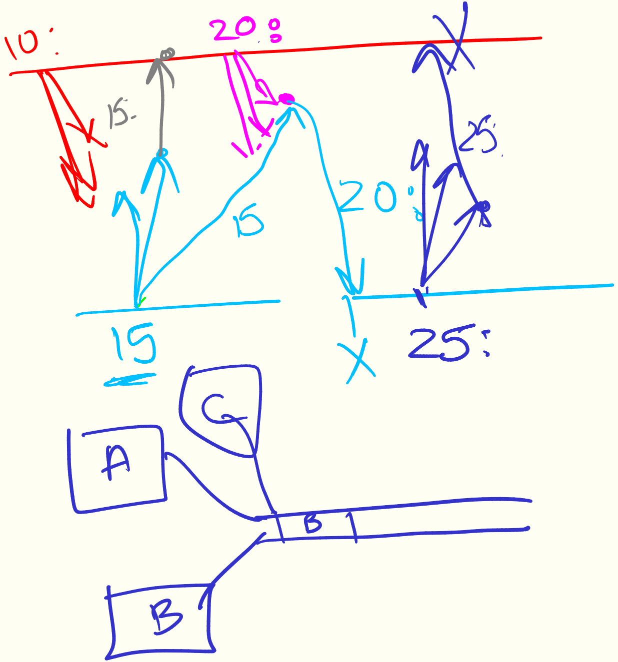
FINALS!

Duelling Leaders

- Liveness can be compromised if there are two leaders
- If higher ballot number is seen, then phase 2 cannot succeed
- Potential solution: Randomized waiting

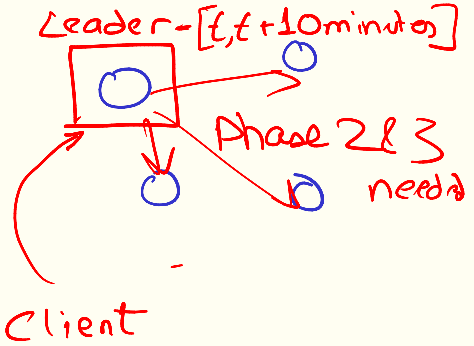
+ exponential backoff

~ Ethernet packet sending



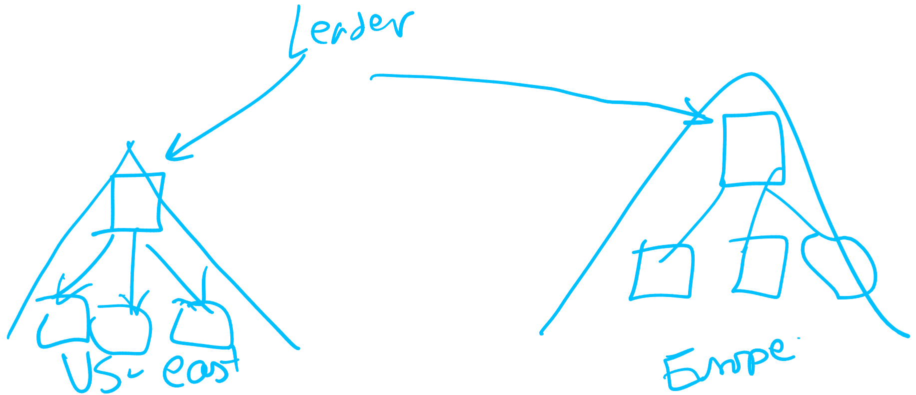
Multi-paxos

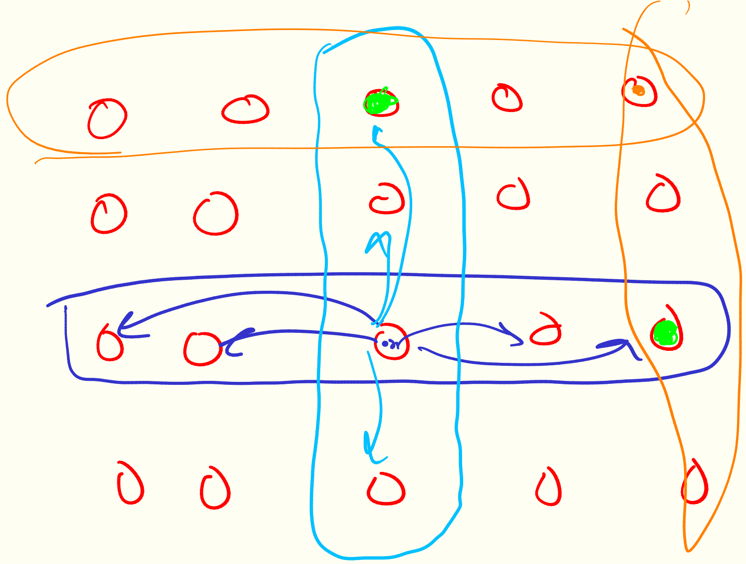
- Optimization to reduce number of phases
- "Master leases": avoid first round of messages
- Leader serves until lease expires.
- Replicas cannot process messages from other wannabe leaders while lease holds



Quorums

- Vanilla paxos: Majority of all acceptors
- Can use quorums of acceptors in phase 2 and 3
- Quorum acceptance suffices





Usecases

- Fault-tolerant storage of metadata
- State machine replication
- Log replication (Apache Kafka)
- Coordinating replica sets
- Leader election
- Synchronization (Mutual exclusion, distributed barriers...)
- Message queues (not ideal!)

When to use paxos

- Paxos provides strong consistency
- Should not be in critical path
- All reads should not have to go through paxos
- Use paxos for small amount of metadata
- Carefully consider replica placement if over a Wide Area Network

Real life use cases

- Google's chubby lock service
  - First known use of paxos in large scale environment?
- Apache Zookeeper

Implementations of Paxos

- Raft. “Easier” to understand alternative to Paxos
- OpenReplica
- libpaxos
- WPaxos

Resources

- Lamport. Part time Parliament (1988)
- Lamport. Paxos made simple
- Butler Lampson. How to Build a Highly Available System Using Consensus
- Paxos made moderately complex http://paxos.systems
- Paxos made live (real-world implementation issues)
- Consensus in the Cloud: Paxos Systems Demystified