Fault Tolerance: Consensus

Distributed Systems Spring 2019

Lecture 18

Agenda

Last time

- Types of faults
- Reliability modeling
- Byzantine Generals

Today

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

Asynchronous vs. Synchronous Systems

C no longer perceives any activity from C^* — a **halting failure**? Distinguishing between a **crash** or **omission/timing failure** may be impossible.

Asynchronous versus synchronous systems

- **Asynchronous system:** no assumptions about process execution speeds or message delivery times $\rightarrow$ **cannot reliably detect crash failures**.
- **Synchronous system:** process execution speeds and message delivery times are bounded $\rightarrow$ **we can reliably detect omission and timing failures**.
- In practice we have **partially synchronous systems:** most of the time, we can assume the system to be synchronous, yet there is no bound on the time that a system is asynchronous $\rightarrow$ **can normally reliably detect crash failures**.
- Partially synchronous also called *eventually* synchronous.

Halting failures

Assumptions we can make

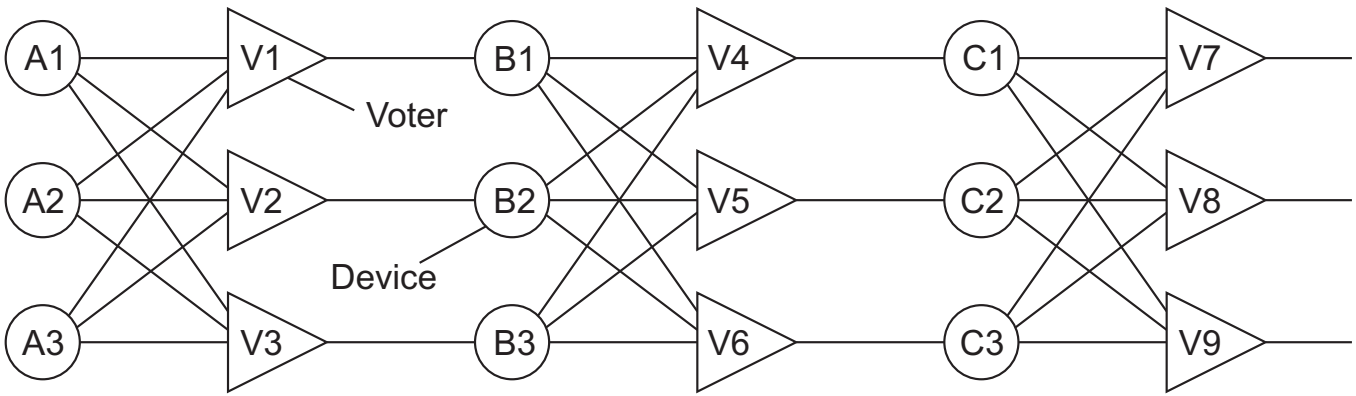
Halting type	Description
Fail-stop	Crash failures, but reliably detectable
Fail-noisy	Crash failures, eventually reliably detectable
Fail-silent	Omission or crash failures: clients cannot tell what went wrong
Fail-safe	Arbitrary, yet benign failures (i.e., they cannot do any harm)
Fail-arbitrary	Arbitrary, with malicious failures

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.

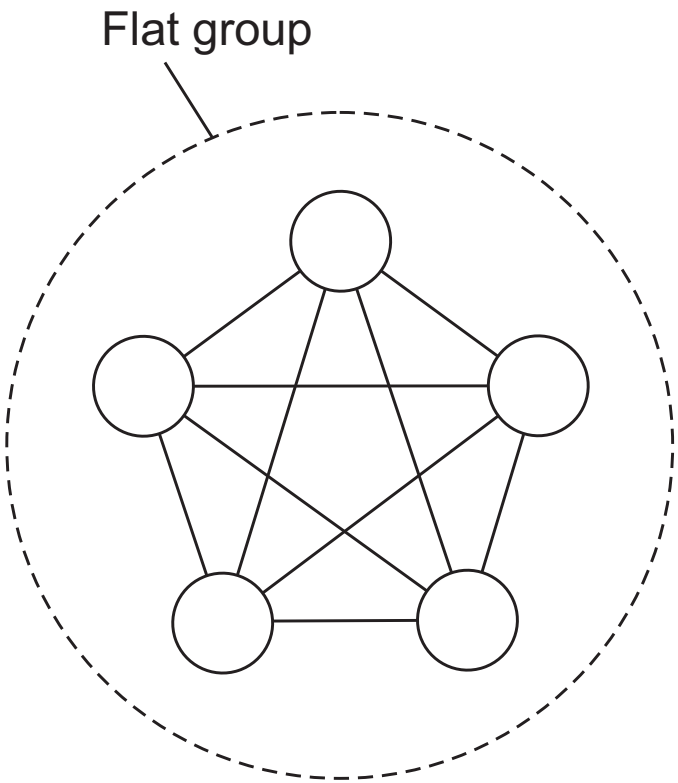
Triple Modular Redundancy

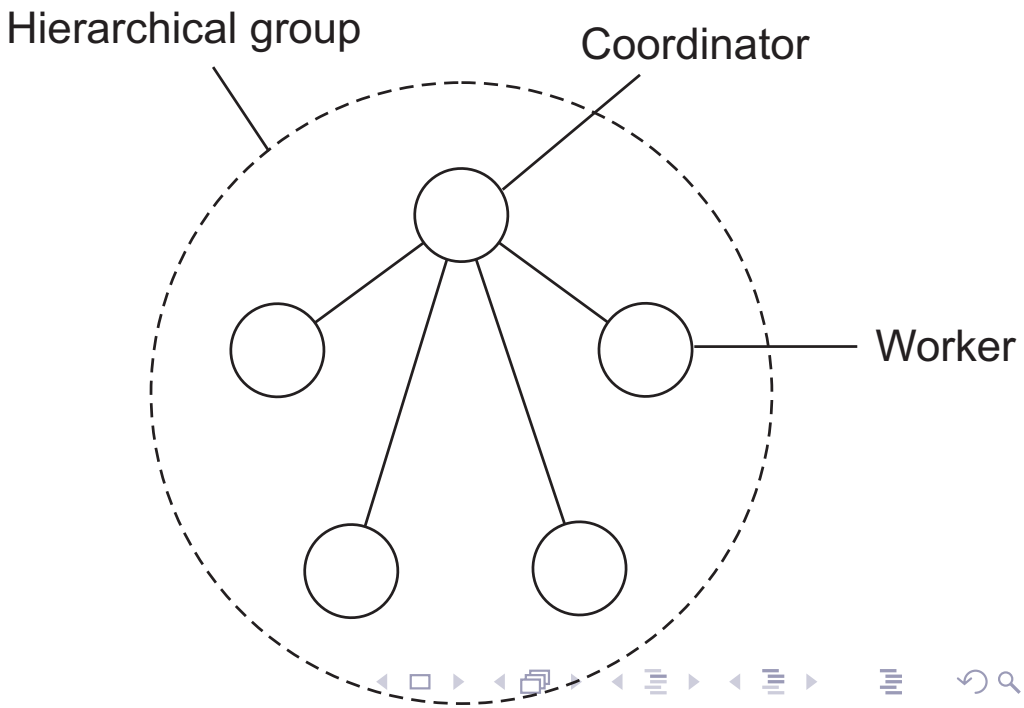


Process resilience

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

Result: We can now be sure that all processes do exactly the same

Consensus

Prerequisite

In a fault-tolerant process group, each nonfaulty 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, agreement, and validity

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

Proof

Proof ommitted

Flooding-based consensus

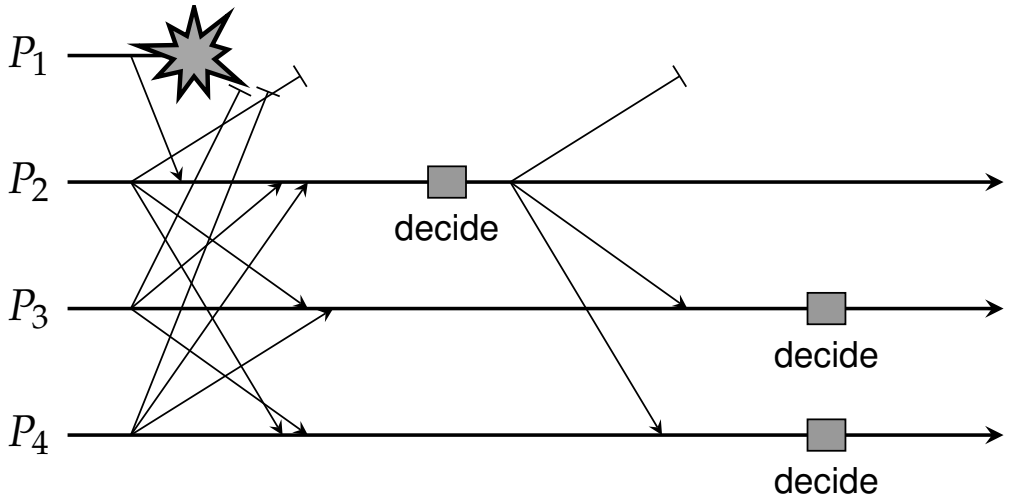
System model

- A process group $P = \{P_1, \ldots, P_n\}$
- Fail-stop failure semantics, i.e., with **reliable failure detection**
- A client contacts a P_i requesting it to execute a command
- Every P_i maintains a list of proposed commands

Basic algorithm (based on rounds)

1. In **round** r, P_i multicasts its known set of commands C_i^r to all others
2. At the end of r, each P_i merges all received commands into a new C_i^{r+1}.
3. Next command cmd_i selected through a **globally shared, deterministic function**: $\text{cmd}_i \leftarrow select(C_i^{r+1})$.

Flooding-based consensus: Example



Observations

- P_2 received all proposed commands from all other processes $\Rightarrow$ **makes decision**.
- P_3 may have detected that P_1 crashed, but does not know if P_2 received anything, i.e., P_3 cannot know **if it has the same information** as P_2 $\Rightarrow$ **cannot make decision** (same for P_4).

PAXOS

Realistic Consensus: Paxos

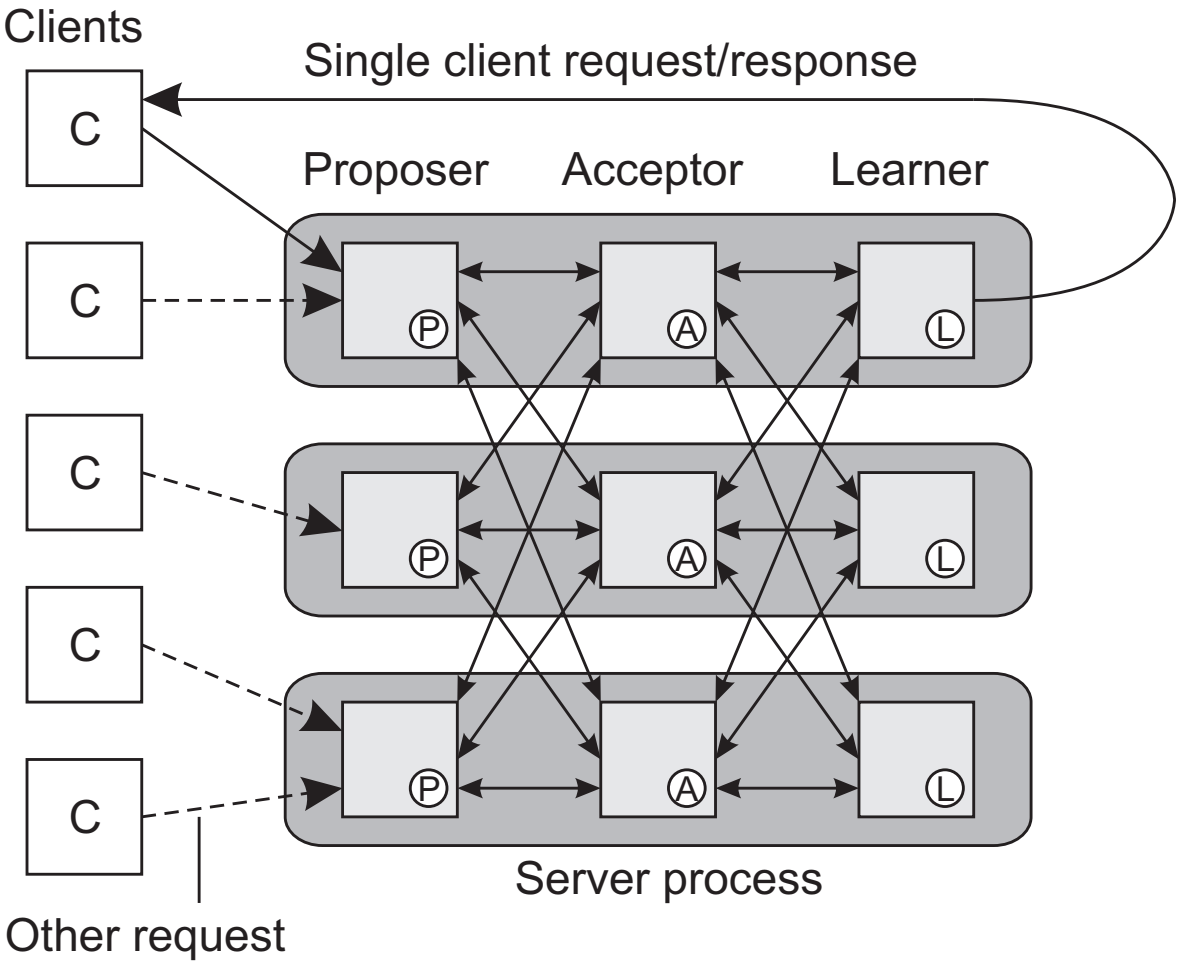
Assumptions (rather weak ones, and realistic)

- 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**.

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

Paxos Components



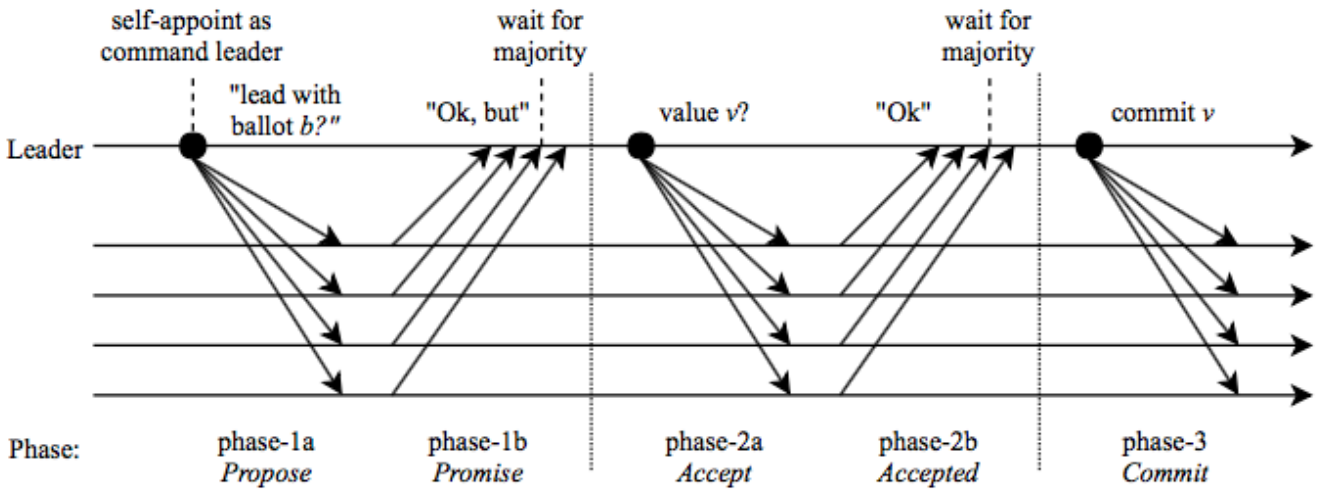
Paxos Properties

- 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

Rounds and Ballots

- Paxos proceeds in rounds. Each round has uniquely numbered ballot. Each round has three phases.
- 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.

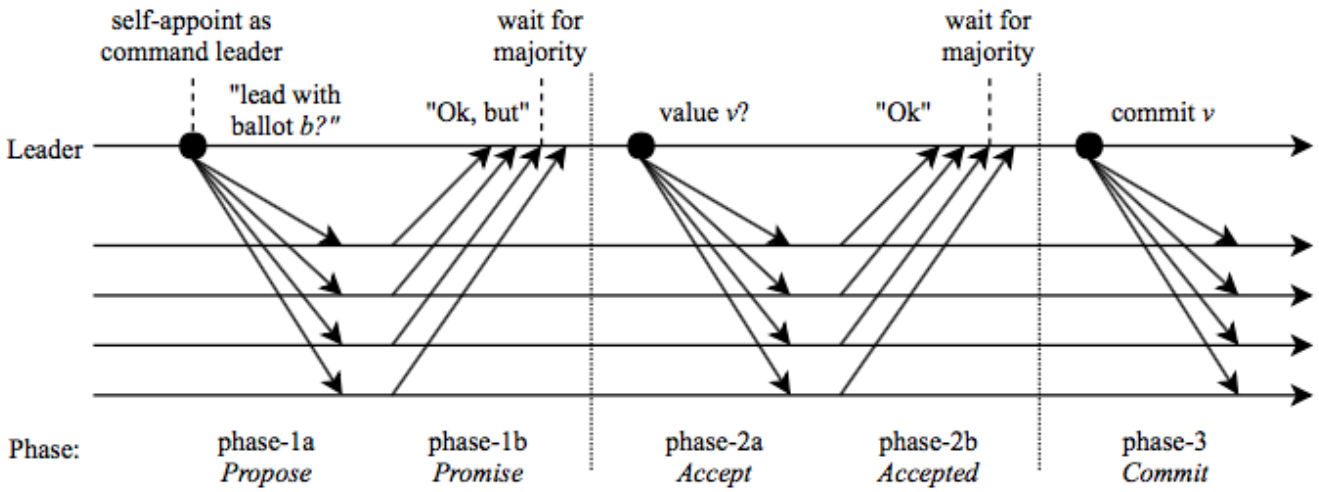
A

lso called the "Prepare" phase.

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

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.

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

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

Distributed Algorithm

Persistent State of acceptors

n_p: Highest prepare seen
n_a, v_a: Highest accept seen

Proposer

While not decided:

1. Choose unique ballot number n
2. Send prepare(n) to all servers including self
3. If prepare_(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

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. $n_p = n$; reply prepare_ok(n, n_a, v_a)
3. Else, reply prepare_reject

Handling accept messages

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

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
-

Usecases

- Fault-tolerant storage of metadata
- Coordinating replica sets

Resources

- Lamport. Part time Parliament (1988)
- Lamport. Paxos made simple
- Paxos made moderately complex http://paxos.systems
- Paxos made live (real-world implementation issues)
- Consensus in the Cloud: Paxos Systems Demystified

Next Time

- More Paxos examples
- Paxos in practice