Distributed Systems
Principles and Paradigms

Maarten van Steen
Chapter 8: Fault Tolerance

vrije Universiteit amsterdam

Dependability

- A component provides services to clients. To provide services, the component may require the services from other components → a component may depend on some other component.
- A component C depends on C* if the correctness of C's behavior depends on the correctness of C*'s behavior.
- Note: in the context of distributed systems, components are generally processes or channels.

Availability	Readiness for usage
Reliability	Continuity of service delivery
Safety	Very low probability of catastrophes
Maintainability	How easily can a failed system be repaired

Reliability versus Availability

- Reliability $R(t)$: probability that a component has been up and running continuously in the time interval $[0,t)$.
- Some traditional metrics:
  - Mean Time To Failure (MTTF): Average time until a component fails.
  - Mean Time To Repair (MTTR): Average time it takes to repair a failed component.
  - Mean Time Between Failures (MTBF): MTTF + MTTR

Reliability versus Availability

- Availability $A(t)$: Average fraction of time that a component has been up and running in the interval [0,t)
  - (Long term) availability A: $A(\infty)$
- Note:
  - A = MTTF/MTBF = MTTF/(MTTF + MTTR)

Observation

Reliability and availability make sense only if we have an accurate notion of what a failure actually is

Terminology

Term	Description	Example
Failure	May occur when a component is not living up to its specifications	A crashed program
Error	Part of a component that may lead to a failure	A programming bug
Fault	The cause of an error	A sloppy programmer

Terminology

Term	Description	Example
Fault prevention	Prevent the occurrence of a fault	Don't hire sloppy programmers
Fault tolerance	Build a component such that it can mask the occurrence of a fault	Build each component by two independent programmers
Fault removal	Reduce the presence, number, or seriousness of a fault	Get rid of sloppy programmers
Fault forecasting	Estimate current presence, future incidence, and consequences of faults	Estimate how a recruiter is doing when it comes to hiring sloppy programmers

Failure models

- Crash failures: Halt, but correct behavior until halting
- General omission failures: failure in sending or receiving messages
  - Receiving omissions: sent messages are not received
  - Send omissions: messages are not sent that should have
- Timing failures: correct output, but provided outside a specified time interval.
  - Performance failures: the component is too slow
- Response failures: incorrect output, but cannot be accounted to another component
  - Value failures: wrong output values
  - State transition failures: deviation from correct flow of control (Note: this failure may initially not even be observable)
- Arbitrary failures: any (combination of) failure may occur, perhaps even unnoticed

Dependability versus security

- Omission failure: A component fails to take an action that it should have taken
- Commission failure: A component takes an action that it should not have taken

Observations

Deliberate failures, be they omission or commission failures, stretch out to the field of security

There may actually be a thin line between dependability and security

Halting failures

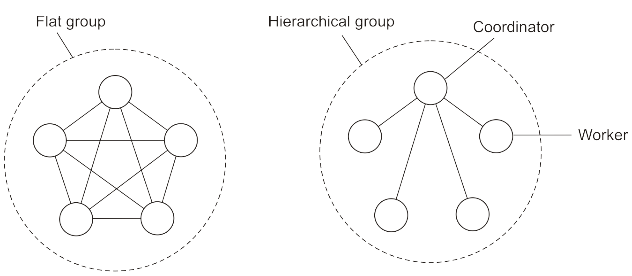
- Scenario: C no longer perceives any activity from C* — a halting failure? Distinguishing between a crash or omission/timing failure may be impossible:
  - Asynchronous system: no assumptions about process execution speeds or message delivery times → cannot reliably detect crash failures.
  - Synchronous system: process execution speeds and message delivery times are bounded → 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 → can normally reliably detect crash failures.

Halting failures

- Assumptions we can make:
  - 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 (can't do any harm).
  - Fail-arbitrary: Arbitrary, with malicious failures

Process reslience

- Basic idea: protect yourself against faulty processes through process replication:



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

Groups and failure masking

- Important:
  - All members are identical
  - All members process commands in the same order
- Result:
  - Only then do we know that all processes are programmed to do exactly the same thing.

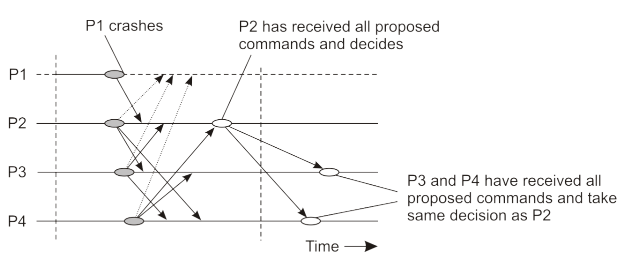
Observation

The processes need to have consensus on which command to execute next

Flooding-based consensus

- Assume:
  - Fail-crash semantics
  - Reliable failure detection
  - Unreliable communication
- Basic idea:
  - Processes multicast their proposed operations
  - All apply the same selection procedure → all process will execute the same if no failures occur
- Problem:
  - Suppose a process crashes before completing its multicast

Flooding-based consensus



Paxos

- Assumptions (rather weak ones):
  - An asynchronous system
  - Communication may be unreliable (meaning that messages may be lost, duplicated, or reordered)
  - Corrupted messages are detectable (and can thus be discarded)
  - All operations are deterministic
  - Process may exhibit halting failures, but not arbitrary failures, nor do they collude.

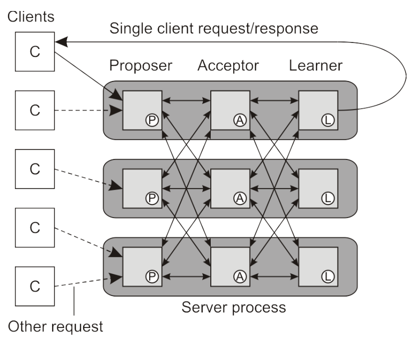
Essential Paxos

- A collection of (replicated) threads, collectively fulfilling the following roles:
  - Client: a thread that requests to have an operation performed
  - Learner: a thread that eventually performs an operation
  - Acceptor: a thread that operates in a quorum to vote for the execution of an operation
  - Proposer: a thread that takes a client's request and attempts to have the requested operation accepted for execution

Essential Paxos

- Safety (nothing bad will happen):
  - Only proposed operations will be learned
  - At most one operation will be learned (and subsequently executed before a next operation is learned)
- Liveness (something good will eventually happen):
  - If sufficient processes remain nonfaulty, then a proposed operation will eventually be learned (and thus executed)

Essential Paxos



Paxos: Phase 1a (prepare)

- A proposer P:
  - has a unique ID, say i
  - communicates only with a quorum of acceptors
  - For requested operation *cmd*:
  - Selects a counter n higher than any of its previous counters, leading to a proposal number $r = (m,i)$.
    Note: $(m,i) < (n,j)$ iff $m < n$ or $m = n$ and $i < j$
  - Sends *prepare*(r) to a majority of acceptors
- Goal:
  - Proposer tries to get its proposal number anchored: any previous proposal failed, or also proposed *cmd*.
    Note: previous is defined wrt proposal number

Paxos: Phase 1b (promise)

- What the acceptor does:
  - If r is highest from any proposer:
  - Return *promise*(r) to p, telling the proposer that the acceptor will ignore any future proposals with a lower proposal number.
  - If r is highest, but a previous proposal (r',*cmd'*) had already been accepted:
  - Additionally return (r',*cmd'*) to p. This will allow the proposer to decide on the final operation that needs to be accepted.
  - Otherwise: do nothing – there is a proposal with a higher proposal number in the works

Paxos: Phase 2a (accept)

- It's the proposer's turn again:
  - If it does not receive any accepted operation, it sends *accept*(*r*,*cmd*) to a majority of acceptors
  - If it receives one or more accepted operations, it sends *accept*(*r*,*cmd**), where
  - *r* is the proposer's selected proposal number
  - *cmd** is the operation whose proposal number is highest among all accepted operations received from acceptors.

Paxos: Phase 2b (learn)

- An acceptor receives an *accept*(*r*,*cmd*) message:
  - If it did not send a *promise*(*r'*) with *r'* > *r*, it must accept *cmd*, and says so to the learners: *learn*(*cmd*).
- A learner receiving learn(cmd) from a majority of acceptors, will execute the operation *cmd*.

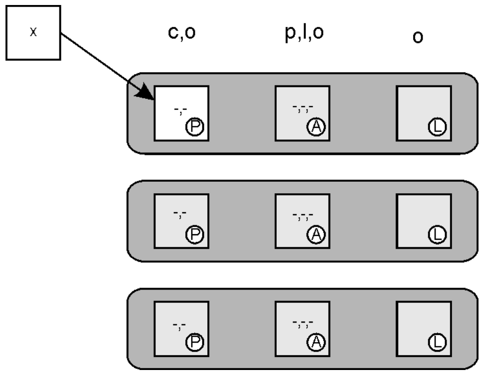
Observation

The essence of Paxos is that the proposers drive a majority of the acceptors to the accepted operation with the highest anchored proposal number

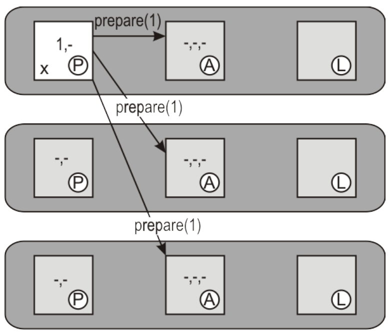
Essential Paxos: Hein Meling

Associate professor @ University Stavanger

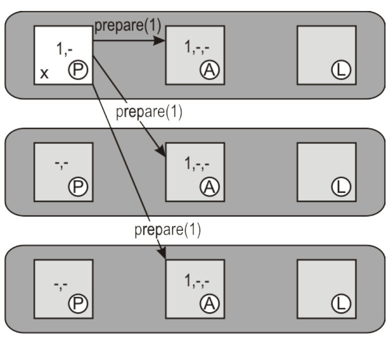
Essential Paxos: Normal case



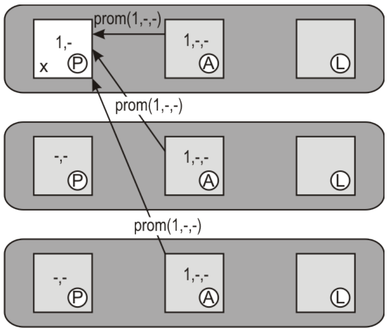
Essential Paxos: Normal case



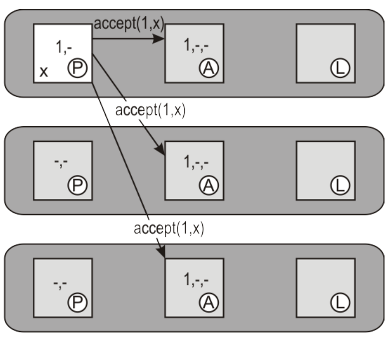
Essential Paxos: Normal case



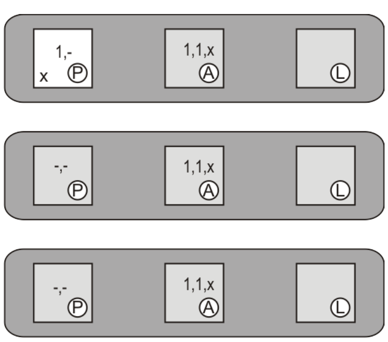
Essential Paxos: Normal case



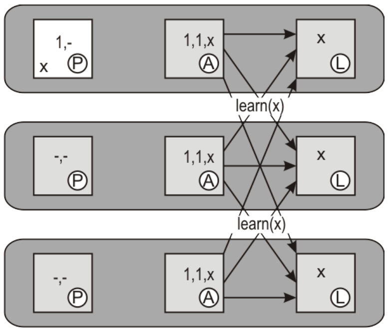
Essential Paxos: Normal case



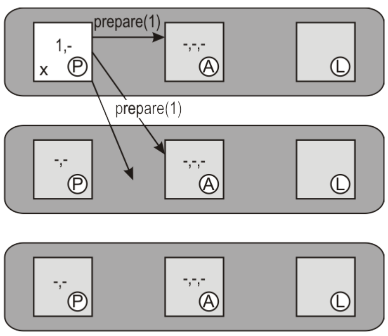
Essential Paxos: Normal case



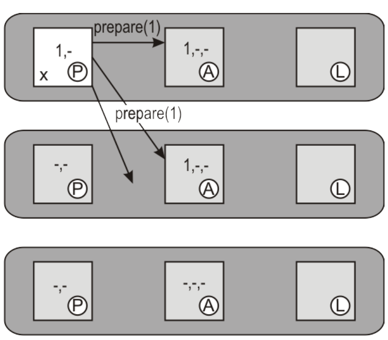
Essential Paxos: Normal case



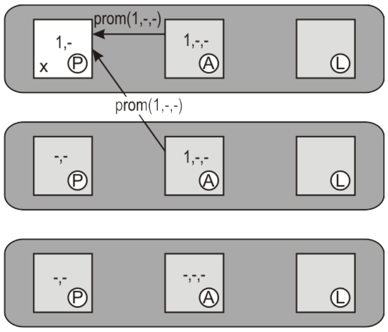
Essential Paxos: Problematic case



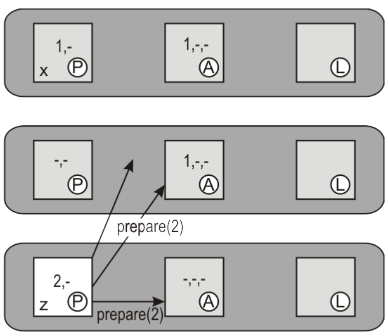
Essential Paxos: Problematic case



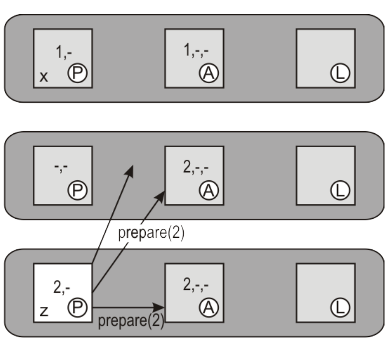
Essential Paxos: Problematic case



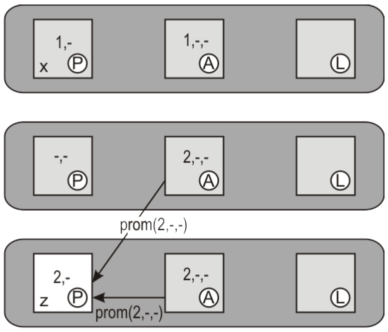
Essential Paxos: Problematic case



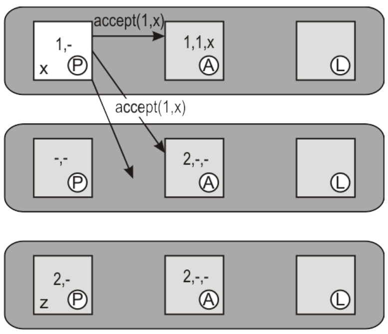
Essential Paxos: Problematic case



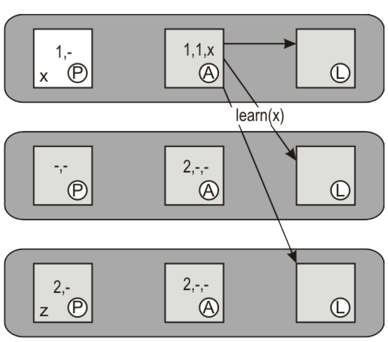
Essential Paxos: Problematic case



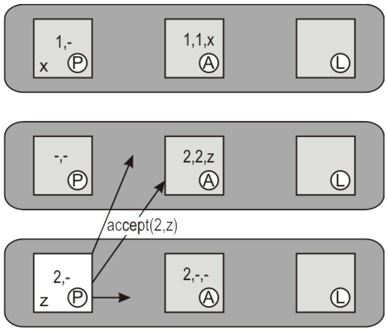
Essential Paxos: Problematic case



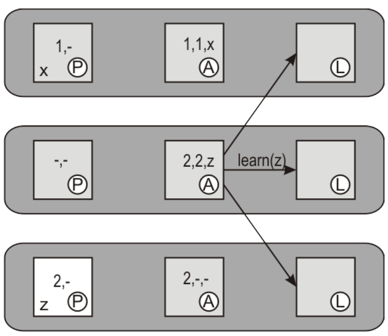
Essential Paxos: Problematic case



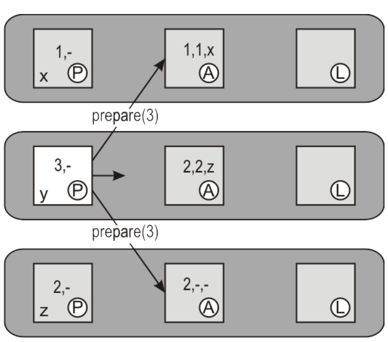
Essential Paxos: Problematic case



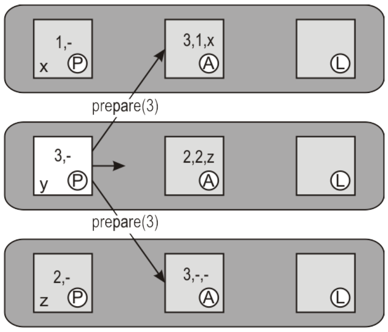
Essential Paxos: Problematic case



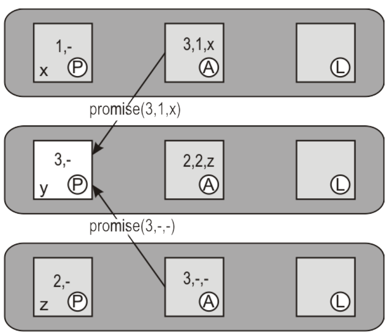
Essential Paxos: Problematic case



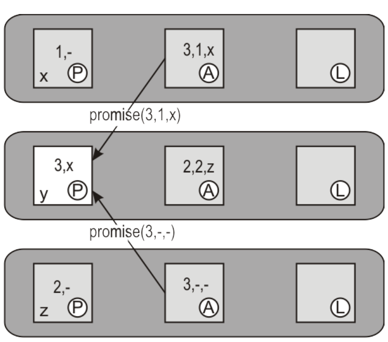
Essential Paxos: Problematic case



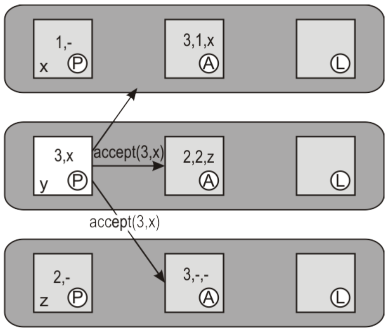
Essential Paxos: Problematic case



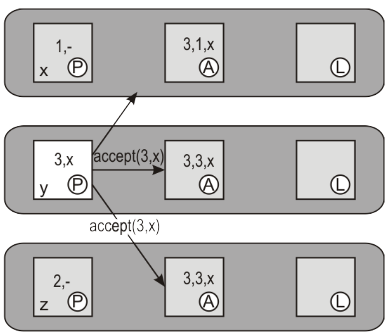
Essential Paxos: Problematic case



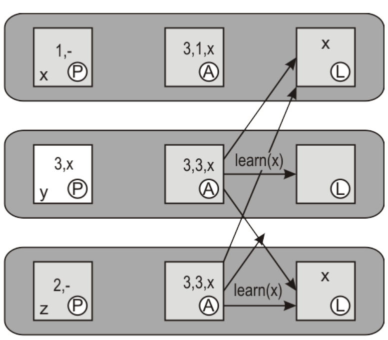
Essential Paxos: Problematic case



Essential Paxos: Problematic case



Essential Paxos: Problematic case



Failure detection

Issue

How can we reliably detect that a process has actually crashed?

- General model:
  - Each process is equipped with a failure detection module
  - A process p probes another process q for a reaction
  - q reacts → q is alive
  - q does not react within t time units → q is suspected to have crashed
- Note: in a synchronous system:
  - a suspected crash is a known crash
  - Referred to as a perfect failure detector

Failure detection

- Practice: the eventually perfect failure detector
- Has two important properties:
  - Strong completeness: every crashed process is eventually suspected to have crashed by every correct process.
  - Eventual strong accuracy: eventually, no correct process is suspected by any other correct process to have crashed.
- Implementation:
  - If p did not receive heartbeat from q within time t → p suspects q.
  - If q later sends a message (received by p):
  - p stops suspecting q
  - p increases timeout value t
  - Note: if q does crash, p will keep suspecting q.