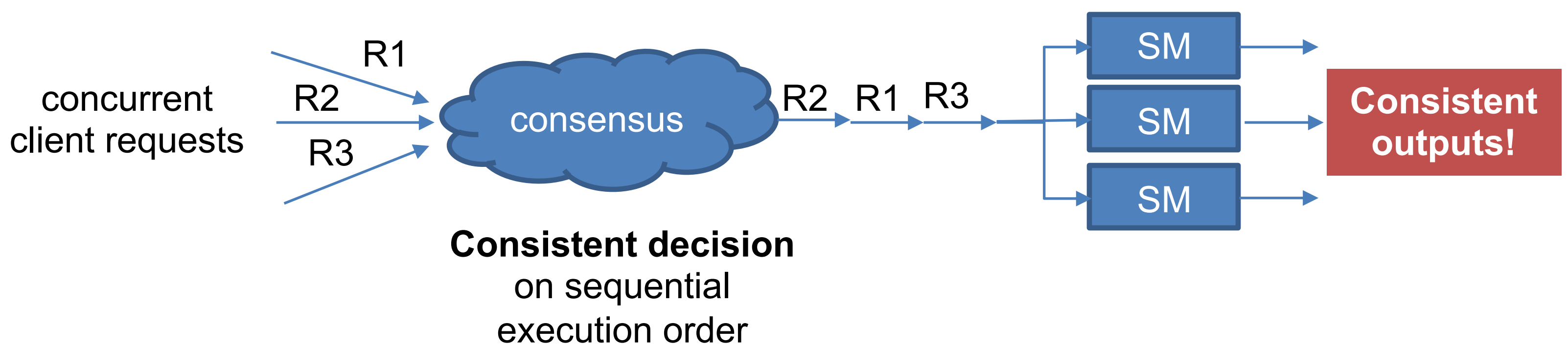
# Consensus vs. 2PC

- Distributed commit requires
  1. Global agreement (consensus) on a decision
  2. A *global constraint* on the value: local abort -> global abort
- Paxos : how to reach consensus in distributed systems that can tolerate non-malicious failures?
  - No global constraint: every local proposal can become global
  - Use case: switching to a new view
  - Use case: state machine replication



# Paxos: fault-tolerant agreement

- Implementation of consensus
- Paxos lets nodes agree on the same value despite:
  - node failures, network failures and delays
- General approach
  - One (or more) nodes decides to be leader (aka proposer)
  - Leader proposes a value and solicits acceptance from others
  - Leader announces result or tries again
- Proposed independently by Lamport and Liskov
  - Widely used in real systems in major companies

# Paxos Requirements

- Safety
  – No two participants ever agree on different values
  – Agreed value X was proposed by some participant
- Liveness (eventually correct participants terminate) if
  1. Less than N/2 participants fail
  2. All participants happens to see the same leader (using a best-effort leader election protocol)
- Why is agreement hard?
  – Network partitions
  – Leader crashes during solicitation or after deciding but before announcing results,
  – New leader proposes different value from already decided value,
  – More than one node becomes leader simultaneously....

# Paxos Setup

- Consider agreeing on a single value
- Entities: Proposer (leader), acceptor, learner
  - Leader proposes value, solicits acceptance from acceptors
  - Acceptors are nodes that want to agree; announce chosen value to learners
- Each proposal has a unique *ballot number*
  - Each leader can choose any high number to try to get proposal accepted
  - An acceptor can accept a proposal only if it has a higher ballot number than previously accepted proposal
- Learners check if a proposal has been chosen by a quorum of acceptors

# Paxos Protocol

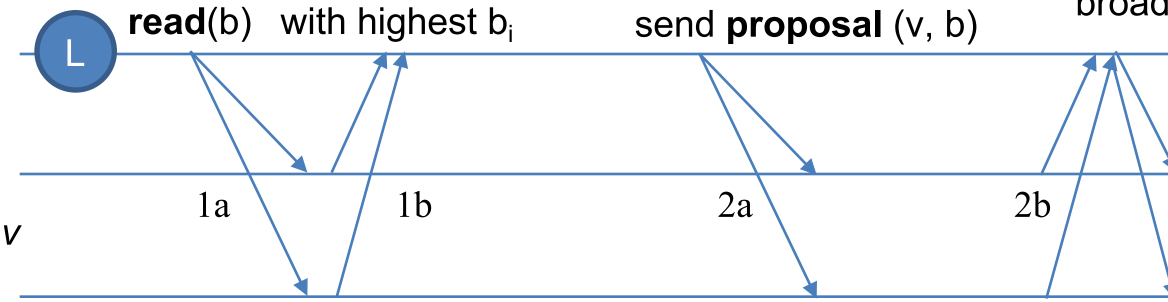
Newly elected leader picks unique ballot number *b*. It has its own proposed value *v*

send **read**(b)

**wait** for n-f replies
If some reply is ($v_i$, $b_i$), set v to $v_i$ with highest $b_i$

send **proposal** (v, b)

**wait** for n-f acks, then **decide** on v and broadcast decision




*If progress gets stuck (not enough replies), the leader picks a larger ballot number and restarts the protocol. Eventually, there will be a single leader with a large enough ballot number which completes all the steps*

If the replica has previously accepted a proposal ($v_i$, $b_i$) and b > $b_i$
1. **reply** with ($v_i$, $b_i$)
2. **promise** not to accept messages with ballot < b

If no prior accepted proposal **reply** with ack

1. **accept** (v, b) unless this breaks **promise**
2. if accept, reply **ack**

Reference: L. Lamport. "*Paxos made simple*"

# Paxos: Why is it Safe to Deliver?

- Definition of chosen proposal (v,b):
  - Accepted by a majority of replicas at a given point in time
- Proposal (v,b) decided by one replica => (v,b) chosen at some point in time
- Invariant:
  - Once (v,b) chosen, future proposals (v', b') from different leaders such that b' > b have v = v'
  - Proposals from old leaders cannot overwrite the ones from newer leaders

# Raft Consensus Protocol

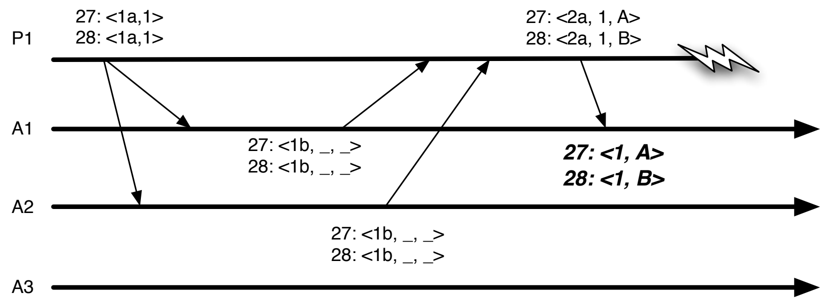
- Paxos is hard to understand (single vs multi-paxos)
- Raft - popular variant of Paxos consensus protocol
  - Each node has a replicated log (the operations to execute)
  - Leader election protocol is integrated with consensus
    - Sequential sequence of terms (instead of ballot numbers)
    - Each term has one leader
    - At the beginning of the term, participants vote replica with the longest committed history
    - The leader with the majority commits its log and then continues

# Active vs. Passive Replication

- Paxos is for *active* replication, where replicas agree on *operations*
- What about *passive* (primary-backup) replication?
  - Primary executes operation and broadcasts state updates to backups
  - State update is a function of operation + state of the primary
  - Backups need to be in the same state as the primary when they apply an update
  - New primary must sync its state with old primaries before sending updates

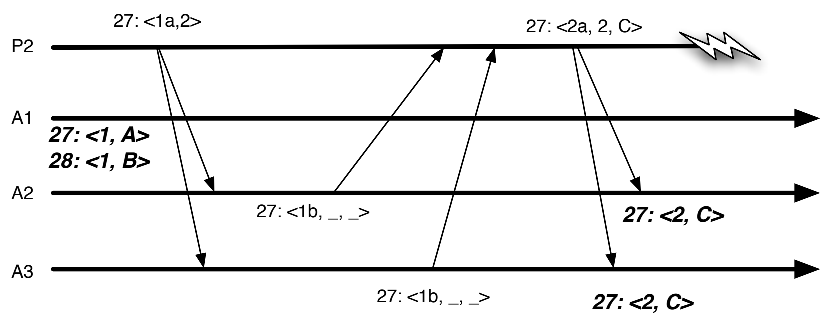
# Paxos and Passive Replication

P1 becomes leader and crashes



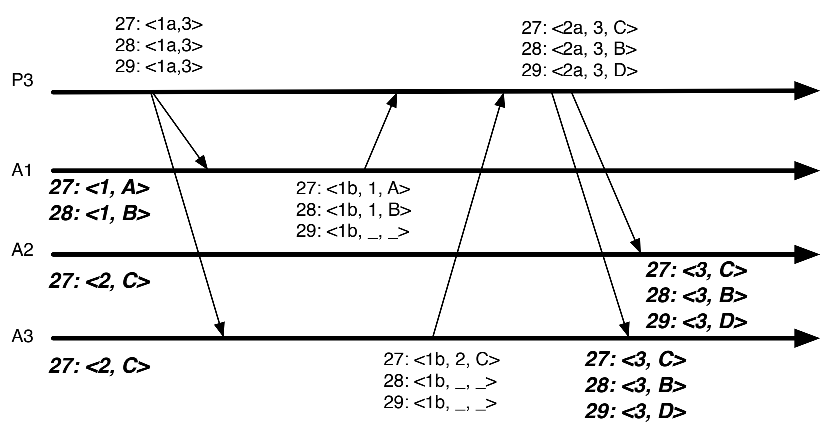

*P1 is primary and generates two state updates A and B for sequence numbers 27 and 28. They must be applied in this order. None is delivered yet.*

P2 becomes leader and crashes




*P2 becomes primary and generates state update C that is an alternative execution branch to A and B*

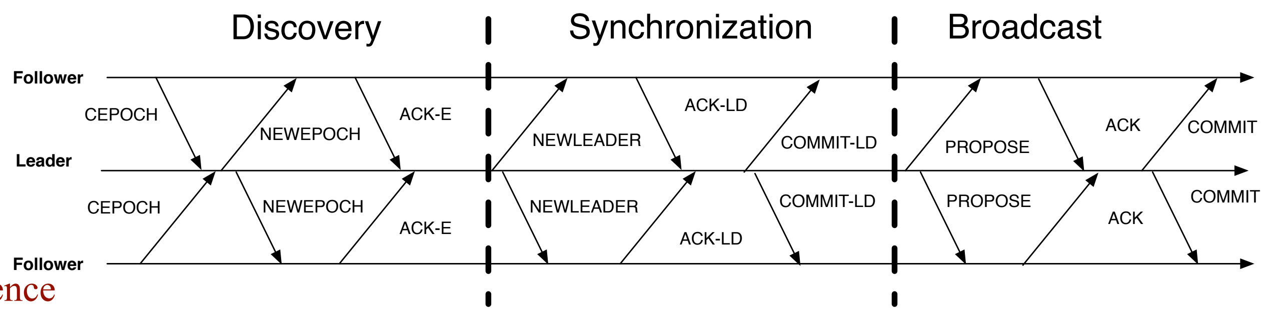
P3 becomes leader




*State update B was generated (and thus must be applied) after A, not C!*

# Zookeeper Atomic Broadcast

- Zab: Consensus + primary order properties
  - Establishes a total order of primaries (leaders)
  - *Local primary order:* Updates from the same primary delivered in FIFO order
  - *Global primary order:* Updates from different primary delivered in primary order
  - *Primary integrity:* Primary delivers all delivered updates from previous primaries *before* it starts sending its own updates
- Used by the Zookeeper coordination protocol
- Idea: Before a new primary starts proposing, replicas must agree on which updates from previous primaries are ever going to be delivered

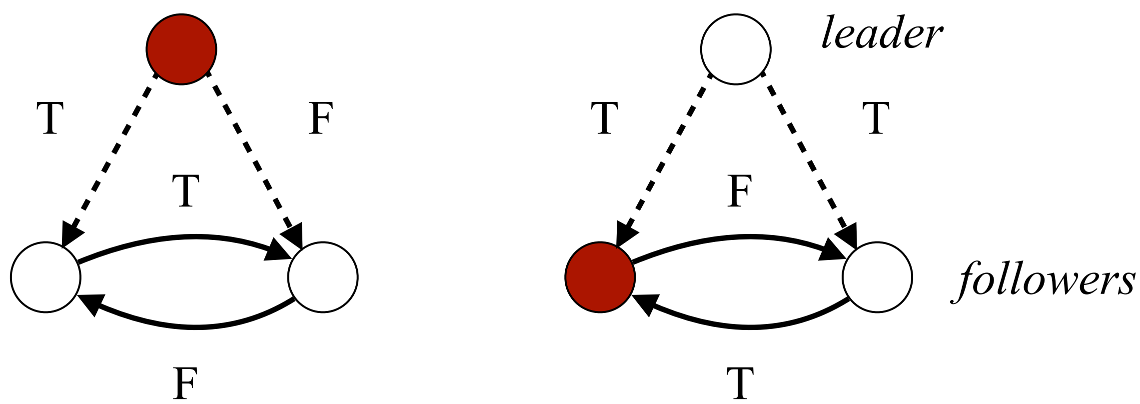


# Byzantine Faults

- Until now: crash faults
- Byzantine faults: arbitrary (adversarial) model for faulty processes
- Metaphor: Byzantine generals
  - Can N generals reach agreement with a perfect channel if some of them are traitors and lies?
- Byzantine agreement
  - Every correct process delivers the same value
  - If the primary is correct, correct processes decide on the primary's proposal
- We consider synchronous communication (accurate timeouts) for simplicity

# Why *3f* Replica Not Enough

- Example with $f = 1$ and $3$ replicas
  - First round: leader broadcasts. Second round: follower exchange what they got
  - Can detect fault but cannot distinguish whether leader or follower is faulty




- With $3f+1=4$ replicas, the 3 follower can use majority voting
  - If primary correct, there is a majority of correct followers
  - If primary faulty, the three followers will all reach consistent decision
    - If there is no majority then the primary was definitely faulty so all replicas

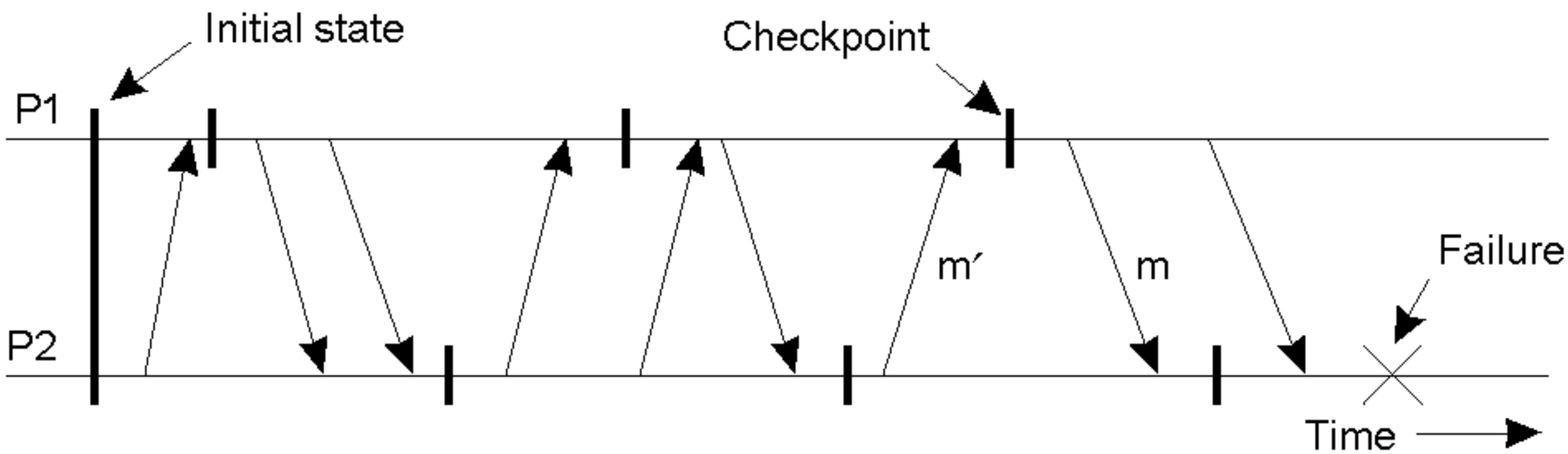
# Byzantine Fault Tolerance

- Detecting a faulty process is easier
  - 2f+1 to detect f faults
- Reaching agreement is harder
  - Need 3f+1 processes (2/3rd majority needed to eliminate the faulty processes)
  - PBFT: extension of Paxos for Byzantine consensus
    - Safe in asynchronous system with leader election
    - Still requires 3f+1
- Implications on real systems:
  - How many replicas?
  - Separating agreement from execution provides savings

# Recovery

- Techniques thus far allow fault *tolerance*
- Fault *recovery:* return to a correct state after a failure
- Based on checkpointing
  - Periodically checkpoint state
  - Upon a crash roll back to a previous checkpoint with a *consistent state*

# Independent Checkpointing




- Each processes periodically checkpoints independently of other processes
- Upon a failure, work backwards to locate a consistent cut
- Problem: if most recent checkpoints form inconsistenct cut, will need to keep rolling back until a consistent cut is found
- Cascading rollbacks can lead to a domino effect.

# Coordinated Checkpointing

- Take a distributed snapshot [already discussed]

- Upon a failure, roll back to the latest snapshot
  - All process restart from the latest snapshot

# Logging

- Logging : a common approach to handle failures
  - Log requests / responses received by system on separate storage device / file (stable storage)
    - Used in databases, filesystems, ...
- Failure of a node
  - Some requests may be lost
  - Replay log to "roll forward" system state

# Message Logging

- Checkpointing is expensive
  - All processes restart from previous consistent cut
  - Taking a snapshot is expensive
  - Infrequent snapshots => all computations after previous snapshot will need to be redone [wasteful]
- Combine checkpointing (expensive) with message logging (cheap)
  - Take infrequent checkpoints
  - Log all messages between checkpoints to local stable storage
  - To recover: simply replay messages from previous checkpoint
    - Avoids recomputations from previous checkpoint