Distributed Systems, Failures, and Consensus

Jeff Chase

Duke University

The Players

- Choose from a large set of interchangeable terms:
  - Processes, threads, tasks,…
  - Processors, nodes, servers, clients,…
  - Actors, agents, participants, partners, cohorts…
- I prefer to use the term "node" or "actor"
  - Short and sweet
  - A logical/virtual entity: may be multiple logical nodes per physical machine.
  - General with regard to role and internal structure
  - Tend to use "actor" if self-interest is an issue

Properties of nodes/actors

- Essential properties typically assumed by model:
  - Private state
    - Distributed memory: model sharing as messages
  - Executes a sequence of state transitions
    - Some transitions are reactions to messages
    - May have internal concurrency, but hide that
  - Deterministic vs. nondeterministic
  - Unique identity vs. anonymous nodes
  - Local clocks with arbitrary drift vs. global time strobe (e.g., GPS satellites)

Node faults and failures

- Fail-stop. Nodes/actors may fail by stopping.
- Byzantine. Nodes/actors may fail without stopping.
  - Arbitrary, erratic, unexpected behavior
  - May be malicious and disruptive
- Unfaithful behavior
  - Actors may behave unfaithfully from self-interest.
  - If it is rational, is it Byzantine?
  - If it is rational, then it is expected.
  - If it is expected, then we can control it.
  - Design in incentives for faithful behavior, or disincentives for unfaithful behavior.

Node recovery

- Fail-stopped nodes may revive/restart.
  - Retain identity
  - Lose messages sent to them while failed
  - Arbitrary time to restart…or maybe never
- Restarted node may recover state at time of failure.
  - Lose state in volatile (primary) memory.
  - Restore state in non-volatile (secondary) memory.
  - Writes to non-volatile memory are expensive.
  - Design problem: recover complete states reliably, with minimal write cost.

Messages

- Processes communicate by sending messages.
- Unicast typically assumed
  - Build multicast/broadcast on top
- Use unique process identity as destination.
- <u>Optional</u>: cryptography
  - (optional) Sender is authenticated.
  - (optional) Message integrity is assured.
  - E.g., using digital signatures or Message Authentication Codes.

Distributed System Models

- Synchronous model
  - Message delay is bounded and the bound is known.
  - E.g., delivery before next tick of a global clock.
  - Simplifies distributed algorithms
    - "learn just by watching the clock"
    - absence of a message conveys information.
- Asynchronous model
  - Message delays are finite, but unbounded/unknown
  - More realistic/general than synchronous model.
    - "Beware of any model with stronger assumptions." - Burrows
  - Strictly harder/weaker than synchronous model.
    - Consensus is not always possible

Messaging properties

- Other possible properties of the messaging model:
  - Messages may be lost.
  - Messages may be delivered out of order.
  - Messages may be duplicated.
- Do we need to consider these in our distributed system model?
- Or, can we solve them within the asynchronous model, without affecting its foundational properties?
  - E.g., reliable transport protocol such as TCP

The network

- Picture a cloud with open unicast and unbounded capacity/bandwidth.
  - Squint and call it the Internet.
- Alternatively, the network could be a graph:
  - Graph models a particular interconnect structure.
  - Examples: star, ring, hypercube, etc.
  - Nodes must forward/route messages.
  - Issues: cut-through, buffer scheduling, etc.
  - Bounded links, blocking send: may deadlock.
  - For that take CPS 221 (Parallel Architectures)

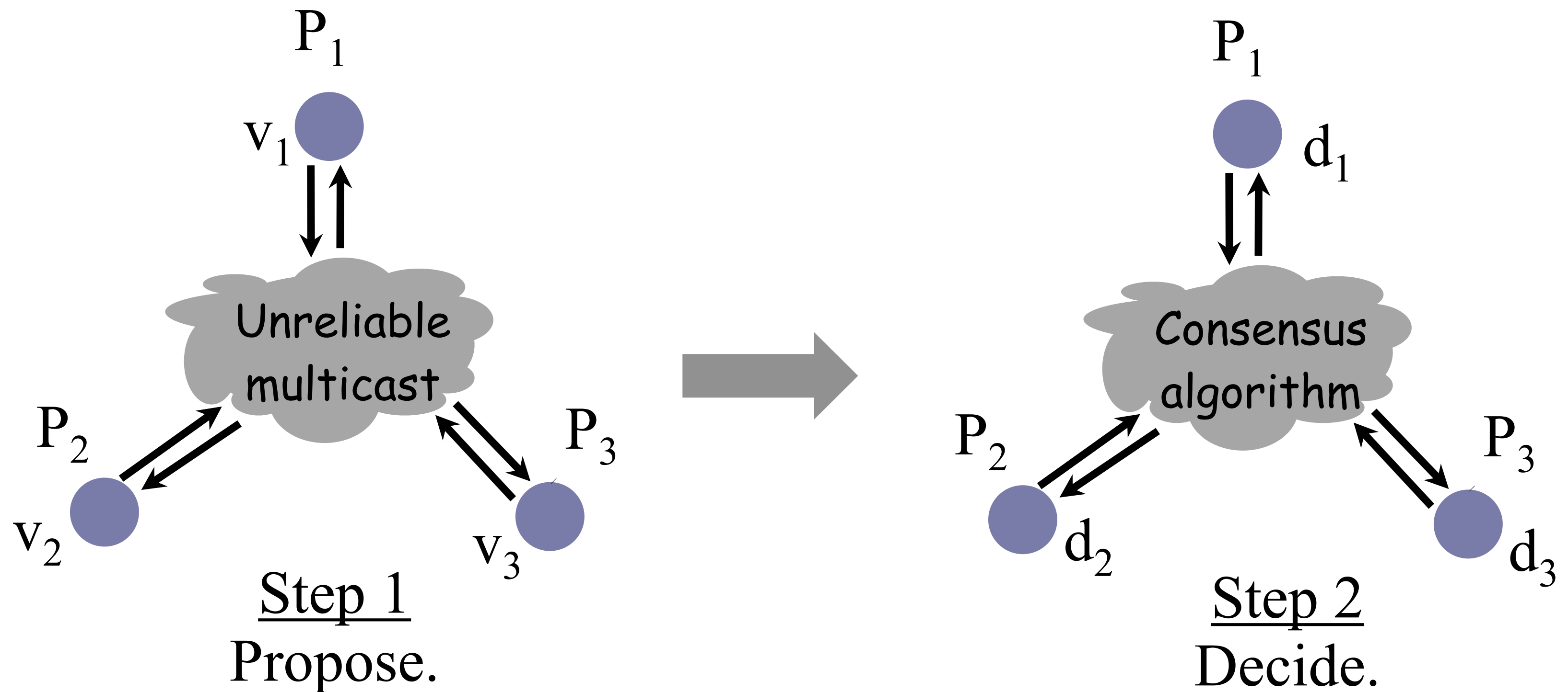
Standard assumptions

- For this class, we make reasonable assumptions for general Internet systems:
  - Nodes with local state and (mostly) local clocks
  - Asynchronous model: unbounded delay but no loss
  - Fail-stop or Byzantine
  - Node identity with (optional) authentication
    - Allows message integrity
  - No communication-induced deadlock.
    - Can deadlock occur? How to avoid it?
  - Temporary network interruptions are possible.
    - Including partitions

Coordination

- If the solution to availability and scalability is to decentralize and replicate functions and data, how do we coordinate the nodes?
  - data consistency
  - update propagation
  - mutual exclusion
  - consistent global states
  - group membership
  - group communication
  - event ordering
  - distributed consensus
  - quorum consensus

Consensus



Generalizes to N nodes/processes.

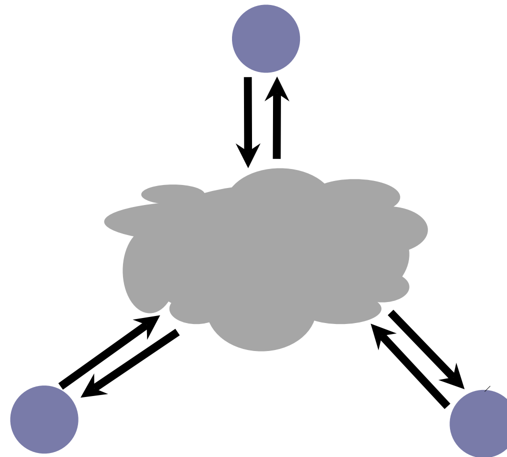
Properties for Correct Consensus

- Termination: All correct processes eventually decide.
- Agreement: All correct processes select the same d_i.
  - Or…(stronger) all processes that do decide select the same d_i, even if they later fail.
  - Called uniform consensus: "Uniform consensus is harder then consensus."
- Integrity: All deciding processes select the "right" value.
  - As specified for the variants of the consensus problem.

Properties of Distributed Algorithms

- Agreement is a safety property.
  - Every possible state of the system has this property in all possible executions.
  - I.e., either they have not agreed yet, or they all agreed on the same value.
- Termination is a liveness property.
  - Some state of the system has this property in all possible executions.
  - The property is stable: once some state of an execution has the property, all subsequent states also have it.

Variant I: Consensus (C)



$d_i = v_k$

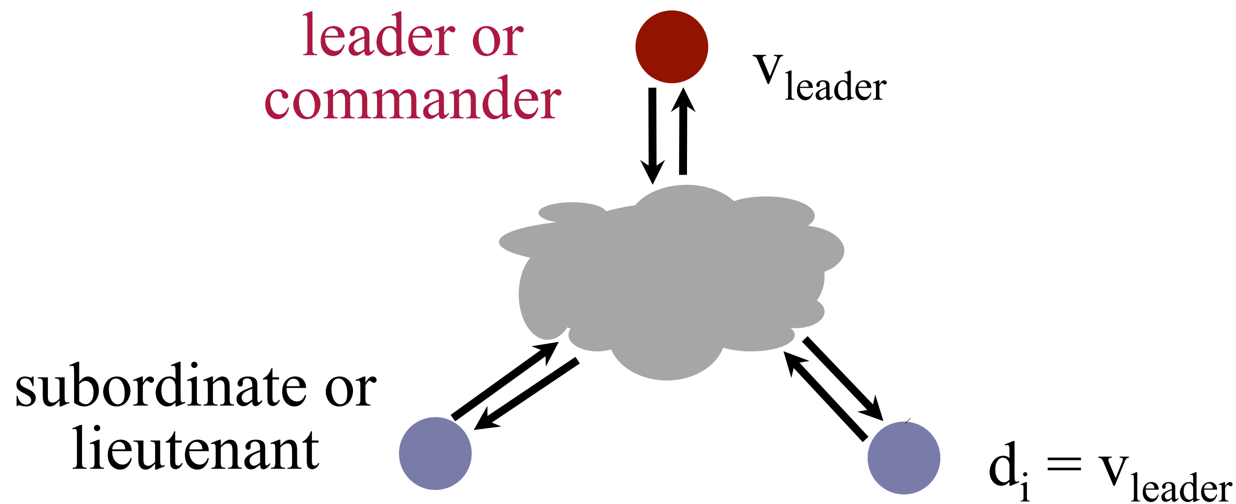
P_i selects d_i from $\{v_0, \ldots, v_{N-1}\}$.

All P_i select d_i as the same v_k .

If all P_i propose the same v, then $d_i = v$, else d_i is arbitrary.

Coulouris and Dollimore

Variant II: Command Consensus (BG)



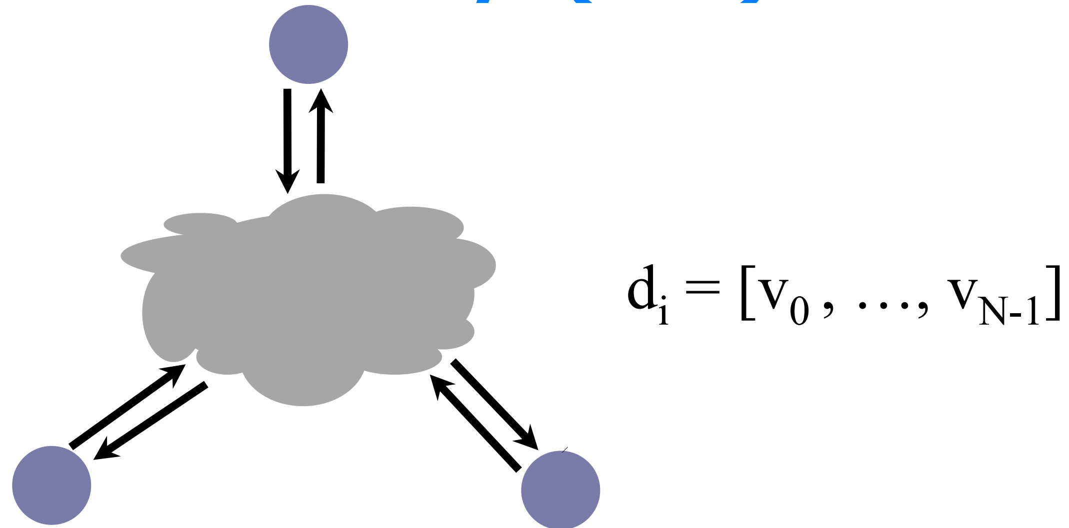
P_i selects $d_i = v_{leader}$ proposed by designated leader node P_{leader} if the leader is correct, else the selected value is arbitrary.

As used in the *Byzantine generals* problem.

Also called *attacking armies*.

Coulouris and Dollimore

Variant III: Interactive Consistency (IC)



P_i selects $d_i = [v_0 , \ldots, v_{N-1}]$ vector reflecting the values proposed by all correct participants.

Coulouris and Dollimore

Equivalence of Consensus Variants

- If any of the consensus variants has a solution, then all of them have a solution.
- Proof is by reduction.
  - IC from BG. Run BG N times, one with each P_i as leader.
  - C from IC. Run IC, then select from the vector.
  - BG from C.
    - Step 1: leader proposes to all subordinates.
    - Step 2: subordinates run C to agree on the proposed value.
  - IC from C? BG from IC? Etc.

Fischer-Lynch-Patterson (1985)

- No consensus can be guaranteed in an asynchronous communication system in the presence of any failures.
- Intuition: a “failed” process may just be slow, and can rise from the dead at exactly the wrong time.
- Consensus may occur recognizably, rarely or often.
  - e.g., if no inconveniently delayed messages
- FLP implies that no agreement can be guaranteed in an asynchronous system with Byzantine failures either. (More on that later.)

Consensus in Practice I

- What do these results mean in an asynchronous world?
  - Unfortunately, the Internet is asynchronous, even if we believe that all faults are eventually repaired.
  - Synchronized clocks and predictable execution times don't change this essential fact.
- Even a single faulty process can prevent consensus.
- The FLP impossibility result extends to:
  - Reliable ordered multicast communication in groups
  - Transaction commit for coordinated atomic updates
  - Consistent replication
- These are practical necessities, so what are we to do?

Consensus in Practice II

- We can use some tricks to apply synchronous algorithms:
  - Fault masking: assume that failed processes always recover, and reintegrate them into the group.
    - If you haven't heard from a process, wait longer…
    - A round terminates when every expected message is received.
  - Failure detectors: construct a failure detector that can determine if a process has failed.
    - A round terminates when every expected message is received, or the failure detector reports that its sender has failed.
- But: protocols may block in pathological scenarios, and they may misbehave if a failure detector is wrong.

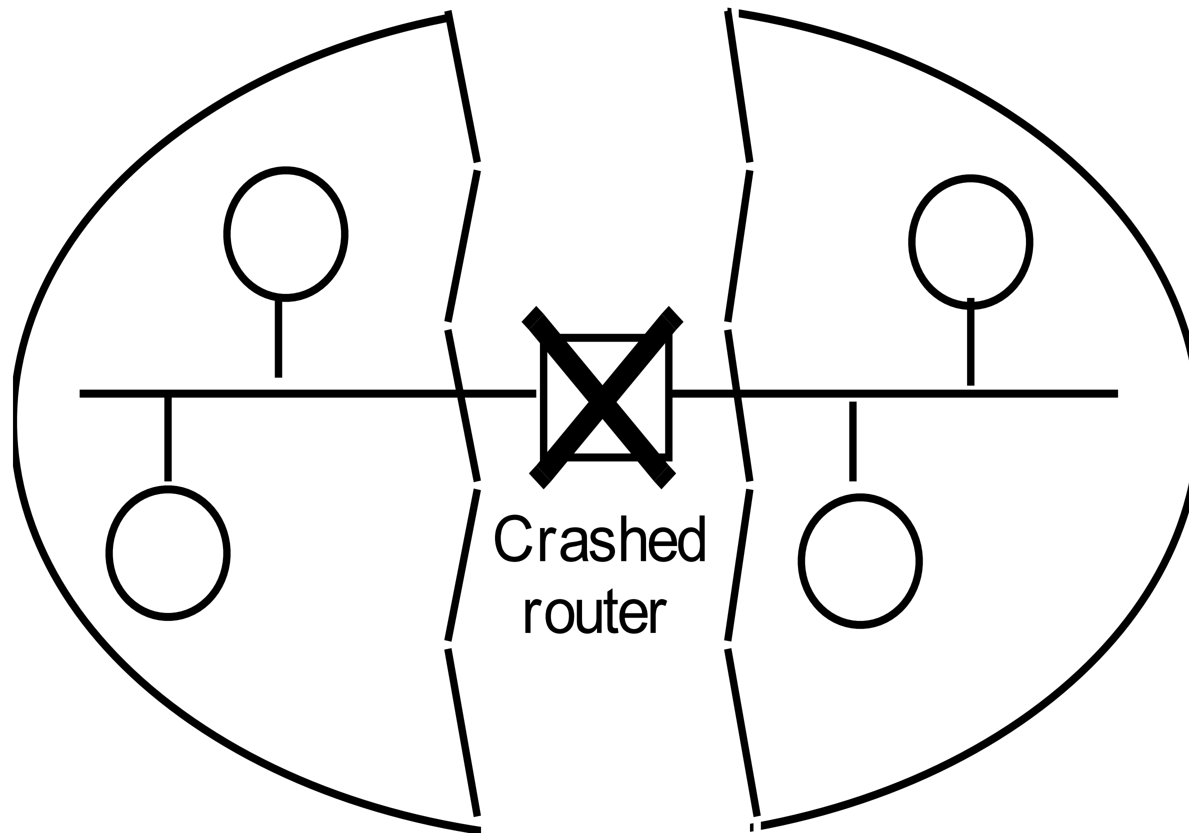
Failure Detectors

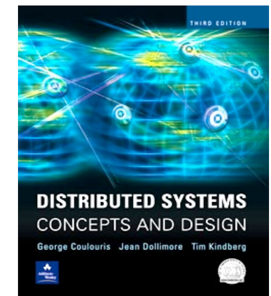
- How to detect that a member has failed?
  - pings, timeouts, beacons, heartbeats
  - recovery notifications
    - "I was gone for awhile, but now I'm back."
- Is the failure detector accurate?
- Is the failure detector live (complete)?
- In an asynchronous system, it is possible for a failure detector to be accurate or live, but not both.
  - FLP tells us that it is impossible for an asynchronous system to agree on anything with accuracy and liveness!

Failure Detectors in Real Systems

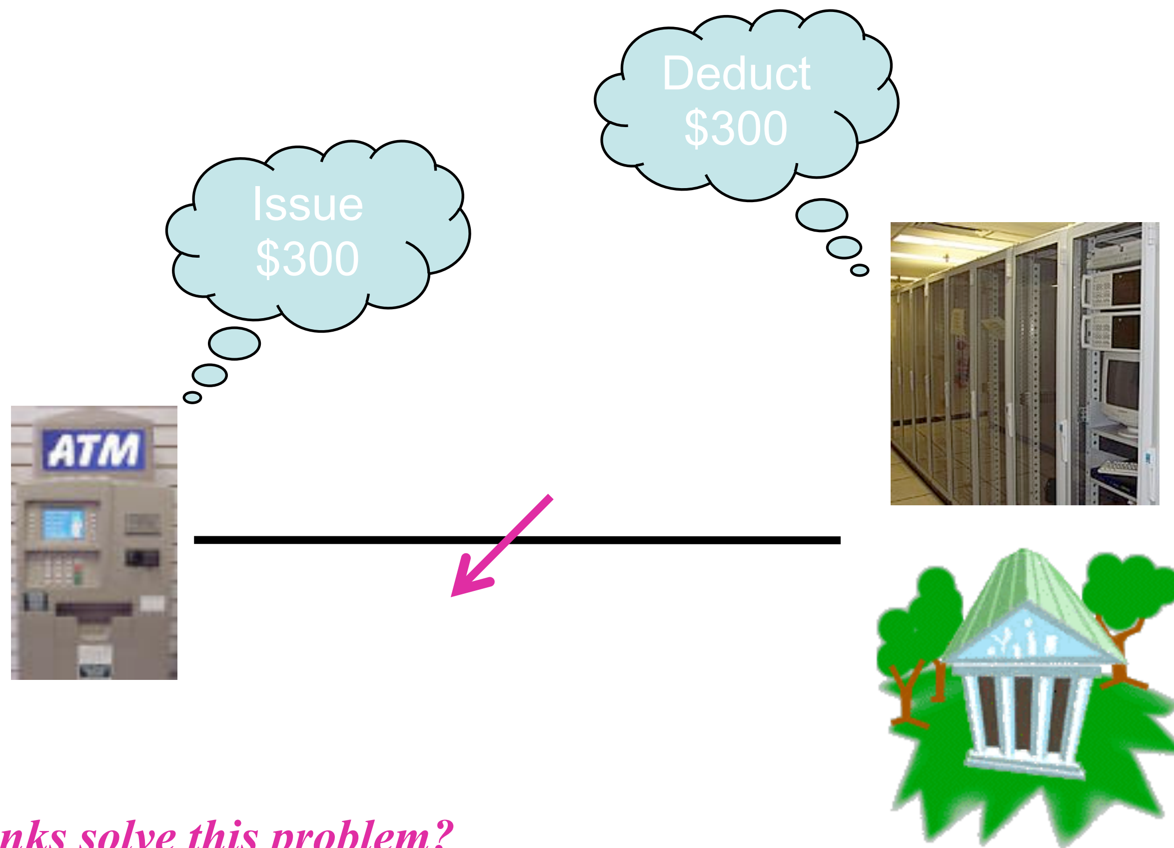
- Use a detector that is accurate but not live.
  - "I'm back....hey, did anyone hear me?"
  - Can't wait forever...
- Use a detector that is live but not accurate.
  - Assume bounded processing delays and delivery times.
  - Timeout with multiple retries detects failure accurately with high probability. Tune it to observed latencies.
  - If a "failed" site turns out to be alive, then restore it or kill it (fencing, fail-silent).
  - Example: leases and leased locks
- What do we assume about communication failures? How much pinging is enough? What about network partitions?

A network partition





Two Generals in practice



How do banks solve this problem?

Keith Marzullo

Committing Distributed Transactions

- Transactions may touch data at more than one site.
- <u>Problem</u>: any site may fail or disconnect while a commit for transaction T is in progress.
  - Atomicity says that T does not "partly commit", i.e., commit at some site and abort at another.
  - Individual sites cannot unilaterally choose to abort T without the agreement of the other sites.
  - If T holds locks at a site S, then S cannot release them until it knows if T committed or aborted.
  - If T has pending updates to data at a site S, then S cannot expose the data until T commits/aborts.

Commit is a Consensus Problem

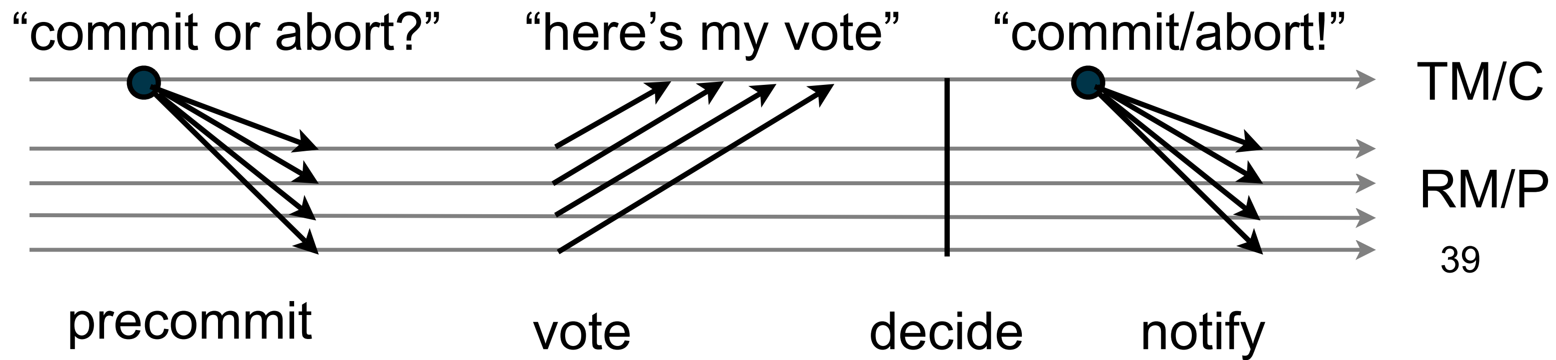
- If there is more than one site, then the sites must agree to commit or abort.
- Sites (Resource Managers or RMs) manage their own data, but coordinate commit/abort with other sites.
  - "Log locally, commit globally."
- We need a protocol for distributed commit.
  - It must be safe, even if FLP tells us it might not terminate.
- Each transaction commit is led by a coordinator (Transaction Manager or TM).

Commit Protocols

- <u>Two phase commit (2PC)</u>
  - Widely taught and used
  - Might block forever if coordinator (TM) fails or disconnects.
- 3PC: Add another phase
  - Reduce the window of vulnerability
- Paxos commit: works whenever it can (nonblocking)
  - Lamport/Gray based on Paxos consensus
  - If TM fails, another steps forward to take over and restart the protocol.

2PC

If unanimous to commit
decide to commit
else decide to abort



RMs validate Tx and prepare by logging their local updates and decisions

TM logs commit/abort (commit point)

2PC: Phase 1

- ✓ 1. *Tx* requests commit, by notifying coordinator (*C*)
  - *C* must know the list of participating sites/RMs.
- ✓ 2. Coordinator *C* requests each participant (*P*) to *prepare*.
- ✓ 3. Participants (RMs) validate, prepare, and vote.
  - Each *P* validates the request, logs validates updates locally, and responds to *C* with its vote to *commit* or *abort*.
  - If *P* votes to commit, *Tx* is said to be "prepared" at *P*.

2PC: Phase 2

✓ 4. Coordinator (TM) commits.

- Iff all ***P*** votes are unanimous to commit
  - ***C*** writes a **commit** record to its log
  - ***Tx*** is committed.
- Else **abort**.

✓ 5. Coordinator notifies participants.

- ***C*** asynchronously notifies each ***P*** of the outcome for ***Tx***.
- Each ***P*** logs the outcome locally
- Each ***P*** releases any resources held for ***Tx***.

Handling Failures in 2PC

- How to ensure consensus if a site fails during the 2PC protocol?
- 1. A participant P fails before preparing.
    - Either P recovers and votes to abort, or C times out and aborts.
- 2. Each P votes to commit, but C fails before committing.
    - Participants wait until C recovers and notifies them of the decision to abort. The outcome is uncertain until C recovers.

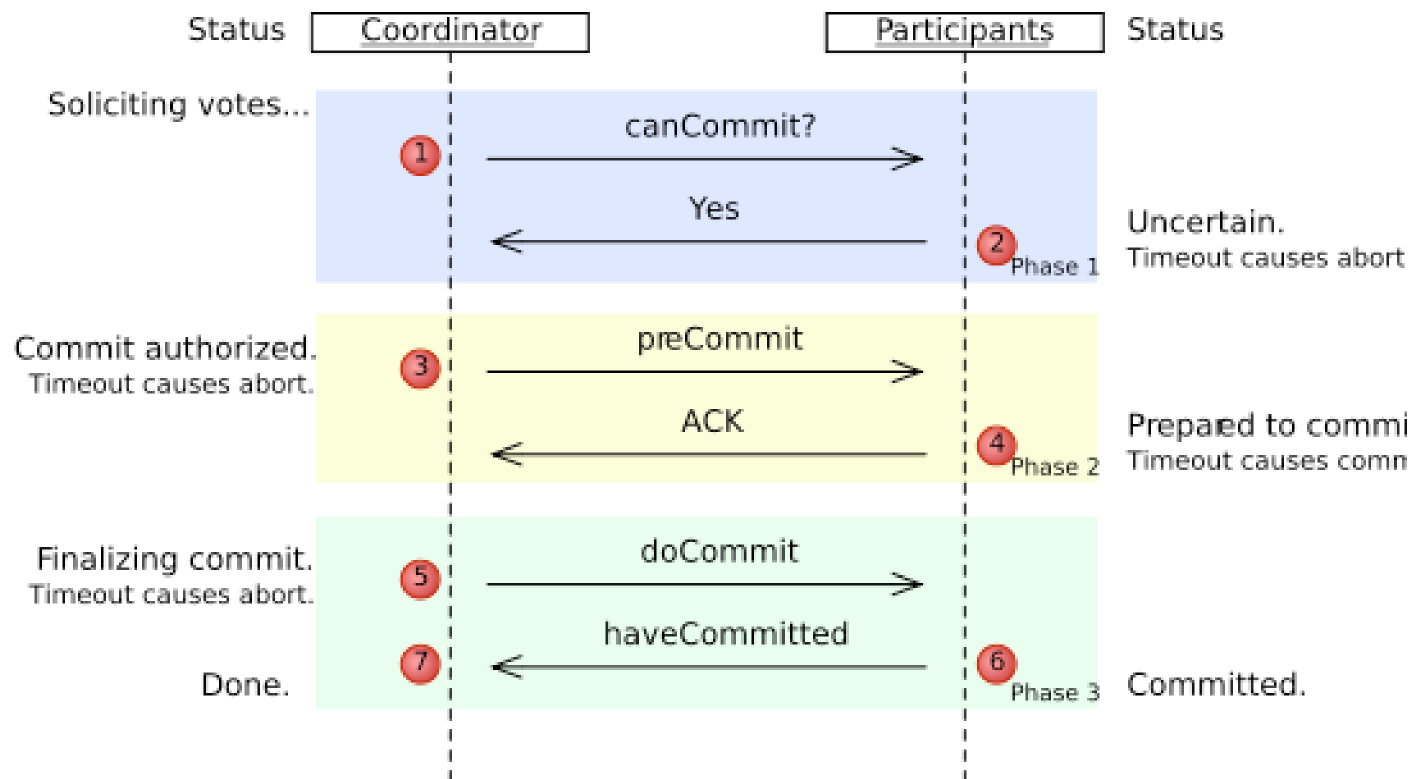
Handling Failures in 2PC, continued

- 3. P or C fails during phase 2, after the outcome is determined.
    - Carry out the decision by reinitiating the protocol on recovery.
    - Again, if C fails, the outcome is uncertain until C recovers.

Notes on 2PC

- Any RM (participant) can enter the prepared state at any time. "The TM's prepare message can be viewed as an optional suggestion that now would be a good time to do so. Other events, including real-time deadlines, might cause working RMs to prepare. This observation is the basis for variants of the 2PC protocol that use fewer messages." Lamport and Gray.
- 3N-1 messages, some of which may be local.
- Non-blocking commit: "failure of a single process does not prevent other processes from deciding if the transaction is committed or aborted.
  - E.g., 3PC.

3PC



General Asynchronous Consensus: Paxos

Paxos? Simple? Really?

- The original Lamport paper “Part-Time Parliament”?
  - These ancient rituals are opaque to most of us.
- Lampson: “How to Build a...System Using Consensus”?
  - Wonderfully clear and elegant...
  - But not simple: Lampson has too much to teach us.
- The Lamport reprise “Paxos Made Simple”?
  - Not clear: too many false leads, paths not taken.
- “Paxos Made Live”: clear, insightful, but too simple.
  - “Read the following papers for the details.”

Paxos Really Made Simple

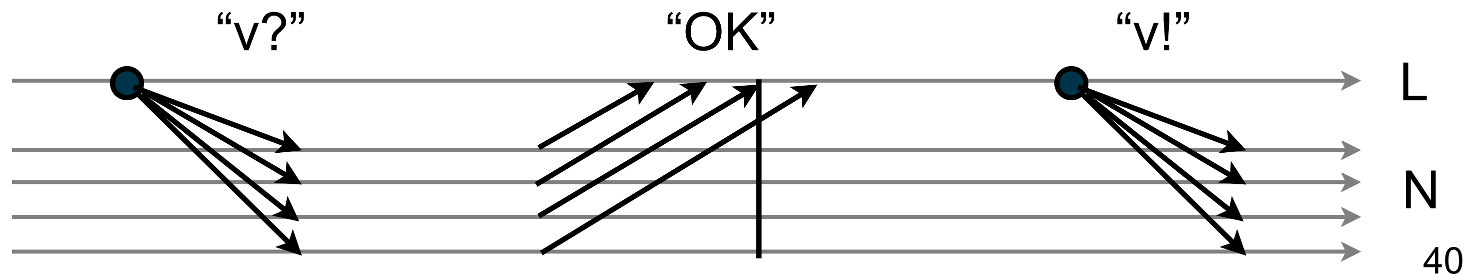
- Today: “Paxos Really Made Simple”.
  - Or at least I try...

Paxos: Properties

- Paxos is an asynchronous consensus algorithm.
- FLP result says no asynchronous consensus algorithm can guarantee both safety and liveness.
- Paxos is guaranteed safe.
  - Consensus is a stable property: once reached it is never violated; the agreed value is not changed.
- Paxos is not guaranteed live.
  - Consensus is reached if "a large enough subnetwork...is nonfaulty for a long enough time."
  - Otherwise Paxos might never terminate.

Paxos: the Players

- N acceptors/agents
  - Majority required for consensus.
  - Paxos can tolerate a minority of acceptors failing.
- Leader/proposer/coordinator
  - Presents a consensus value to the acceptors and counts the ballots for acceptance of the majority.
  - Notifies the agents of success.
- Note: any node/replica may serve either/both roles.



Paxos: Leader Election

- In general, leader election is a consensus problem!
  - No fair for a consensus algorithm to depend on it.
- Paxos is safe with multiple leaders.
  - In essence leader election is "built in": no need to agree in advance who the "real" leader is.
  - Robust: if consensus (appears) stalled, anyone can (try to) take over as self-appointed leader.
- But: too many would-be leaders can cause livelock.

Leaders and Liveness

- In Paxos any new leader usurps previous leaders.
  - New leader can intrude before protocol completes.
- How aggressively should nodes claim leadership?
  - Too aggressive?
    - Costly, and protocol might fail to terminate.
  - Not aggressive enough?
    - Protocol stalls.
- Solution: use Paxos sparingly and with suitable leader timeouts [Lampson].

Paxos in Practice

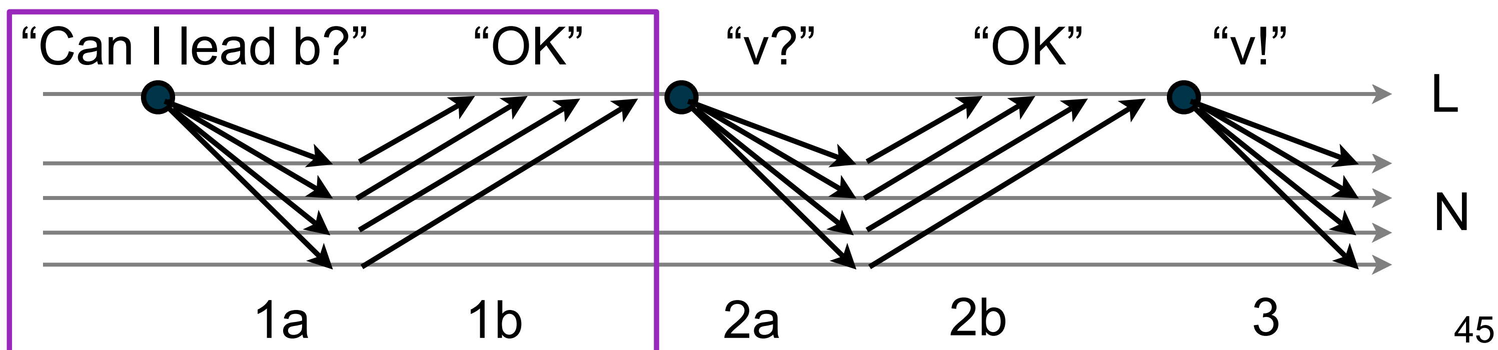
- Lampson: "Since general consensus is expensive, practical systems reserve it for emergencies."
  - e.g., to select a primary such as a lock server.
    - Frangipani
    - Google Chubby service ("Paxos Made Live")
- Pick a primary with Paxos. Do it rarely; do it right.
  - Primary holds a "master lease" with a timeout.
    - Renewable by consensus with primary as leader.
  - Primary is king as long as it holds the lease.
  - Master lease expires? Fall back to Paxos.

Rounds and Ballots

- The Paxos protocol proceeds in rounds.
  - Each round has a uniquely numbered ballot.
- If no failures, then consensus is 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.
- It might take even more rounds for the acceptors (agents) to learn that consensus was reached.

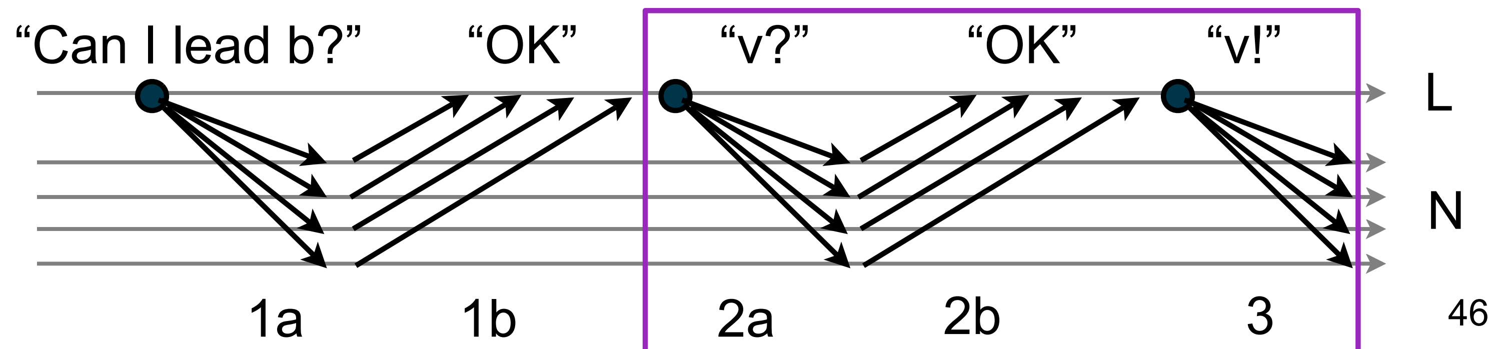
Phase 1: Proposing a Round

- Would-be leader chooses a unique ballot ID.
- Propose to the acceptors/agents (1a).
  - Will you consider this ballot with me as leader?
- Agents return the highest ballot ID seen so far (1b).
  - Seen one higher than yours? That's a rejection.
- If a majority respond and know of no higher ballot number, then you are their leader (for this round).



Phase 2-3: Leading a Round

- Congratulations! You were accepted to lead a round.
  - Choose a "suitable value" for this ballot.
  - Command the agents to accept the value (2a).
  - If a majority hear and obey, the round succeeds.
- Did a majority respond (2b) and assent?
  - Yes: tell everyone the round succeeded (3).
  - No: move on, e.g., ask for another round.

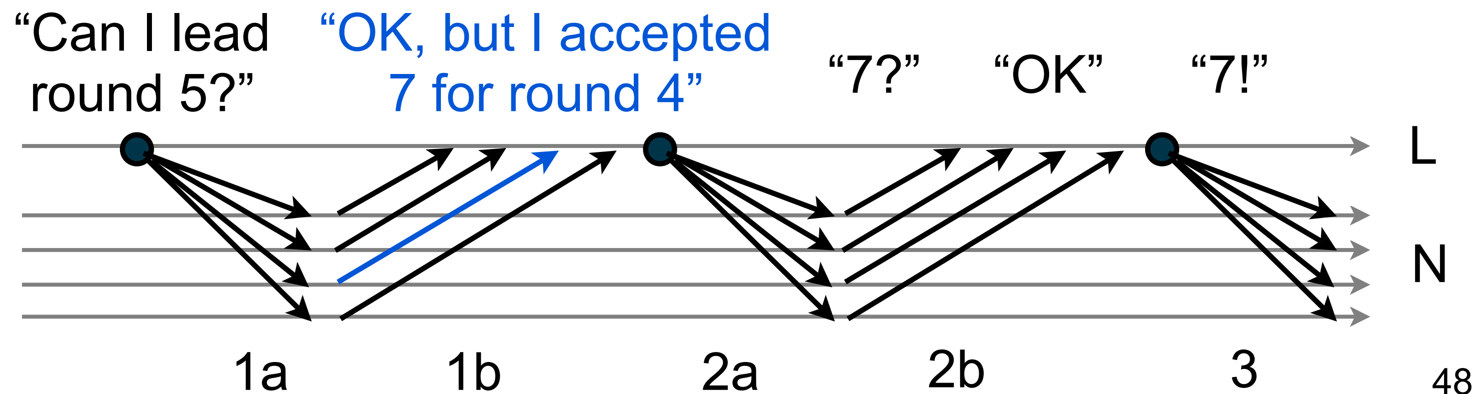


The Agents

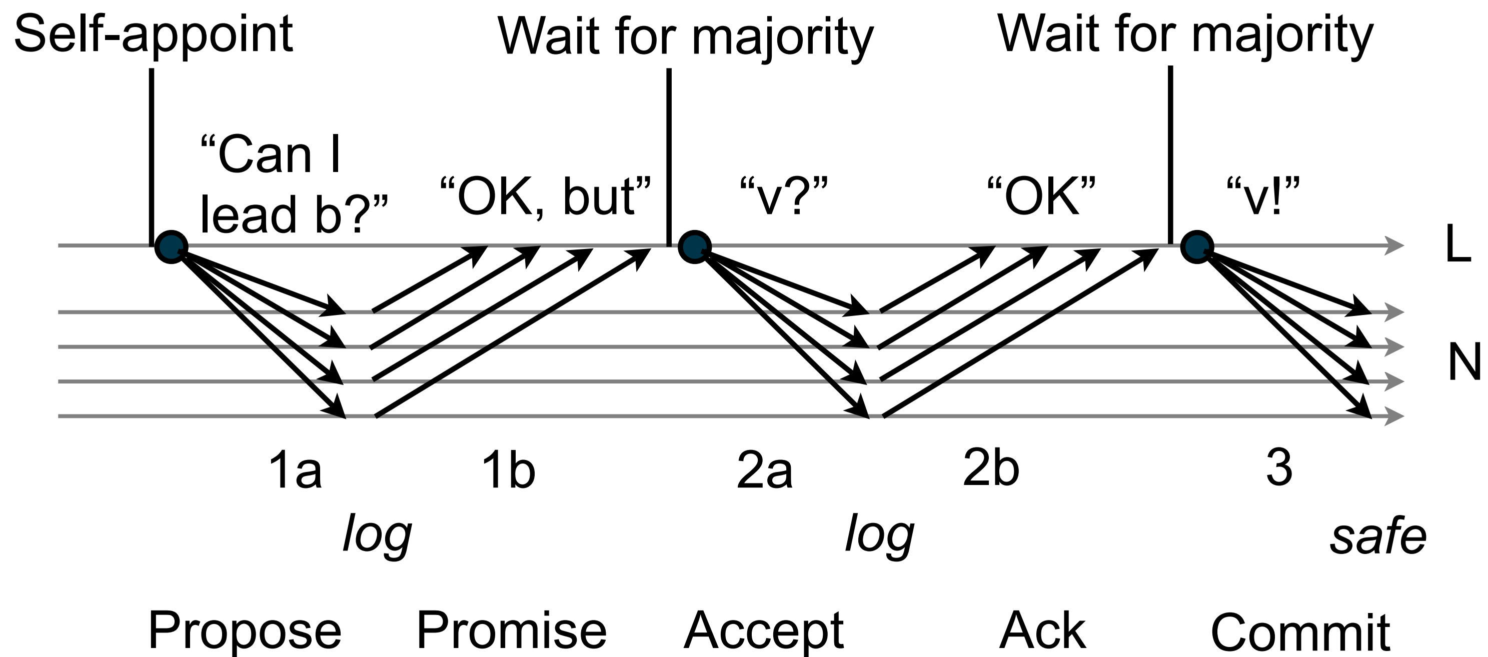
- Proposal for a new ballot (1a) by a would-be leader?
  - If this ballot ID is higher than any you have seen so far, then accept it as the current ballot.
    - Log ballot ID in persistent memory.
  - Respond with highest previous ballot ID "etc." (1b)
- Commanded (2a) to accept a value for current ballot?
  - Accept value and log it in persistent memory.
  - Discard any previously accepted value.
  - Respond (2b) with accept, or deny (if ballot is old).

Leader: Choosing a Suitable Value

- A majority of agents responded (1b): good. Did any accept a value for some previous ballot (in a 2b message)?
  - Yes: they tell you the ballot ID and value.
    - That was the "etc." in 1b.
    - Find the most recent value that any responding agent accepted, and choose it for this ballot too.
  - No: choose any value you want.



A Paxos Round



Where is the consensus “point of no return”?

Success and Failure

- A round succeeds if a majority of agents hear the command (2a) and obey.
- A round fails if too many agents fail, too many command messages are lost, or another leader usurps.
  - But some agents may survive and hear the command, and obey it even though the round failed.
- Liveness requires that agents are free to accept different values in subsequent rounds.
- But: safety requires that once some round succeeds, no subsequent round can change it.

What Happens After Success?

- The leader may not know that the round succeeded.
  - Leader may fail. Agent responses (2b) may be lost.
- The agents may not know that the round succeeded.
  - They won't know until the leader knows and tells them (3), which might not happen (e.g., see above).
  - Even if the leader knows, it may fail or go quiet.
- Even so, it succeeded: consensus has been reached!
  - We can never go back.
- Solution: have another round, possibly with a different leader, until you learn of your success.

Safety: Outline

- Key invariant: If some round succeeds, then any subsequent round chooses the same value, or it fails.
- To see why, consider the leader L of a round R.
  - If a previous round S succeeded with value v, then either L learns of (S, v), or else R fails.
    - Why? S got responses from a majority: if R does too, then some agent responds to both.
  - If L does learn of (S,v), then by the rules of Paxos L chooses v as a “suitable value” for R.
  - (Unless there was an intervening successful round.)

More on Safety

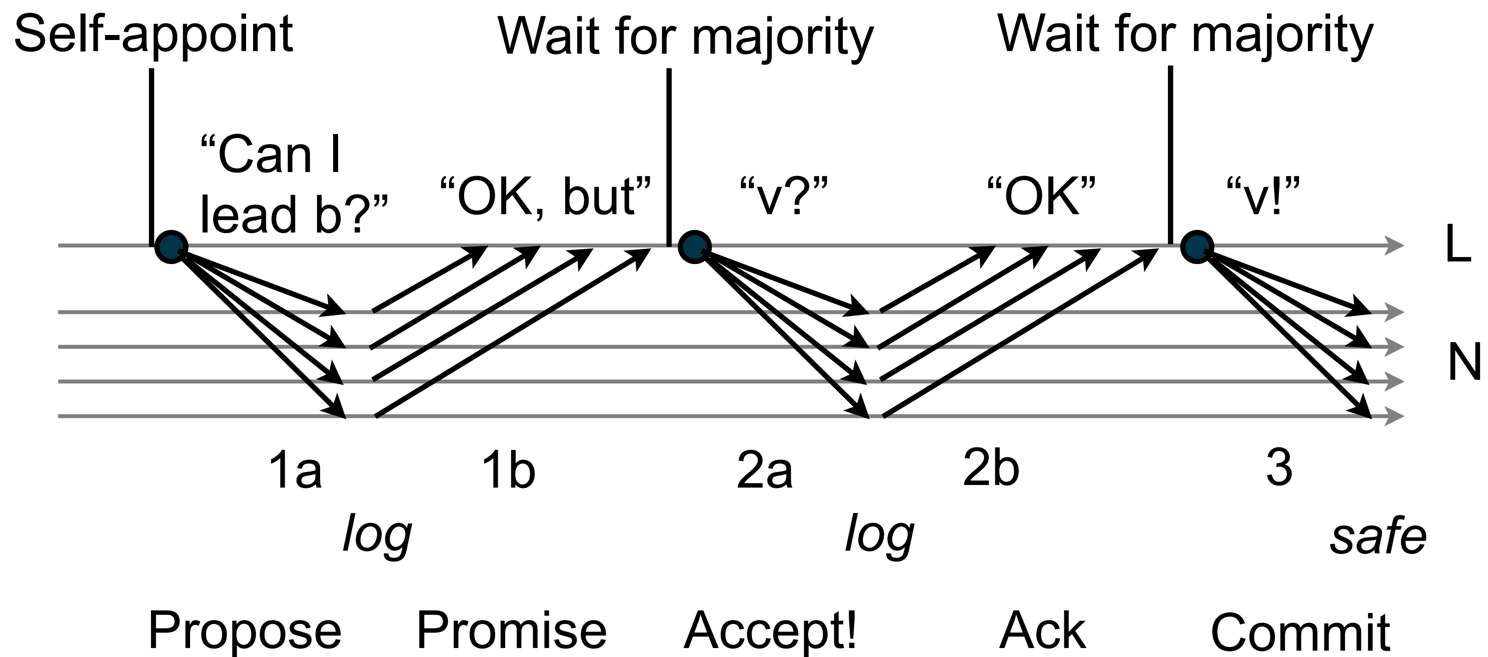
- All agents that accept a value for some round S accept the same value v for S.
  - They can only accept the one value proposed by the single leader for that unique round S.
- And if an agent accepts a value (after 2a), it reports that value to the leader of a successor round (in 1b).
- Therefore, if R is the next round to succeed, then the leader of a R learns of (S,v), and picks v for R.
  - Success requires a majority, and majority sets are guaranteed to intersect.
  - Induction to all future successful rounds.

Consensus

Jenkins, if I want another yes-man, I'll build one!

[Cribbed from a presentation by Ken Birman.]

A Paxos Round



Some Failure Cases

- Agent fails.
  - Go on without it, and/or it restarts with its hard state: (last ballot ID, last accepted value).
- Leader fails.
  - Abandon the round and follow a new leader. If the round succeeded, we'll find out and stick to it.
- Leaders race.
  - Eventually some 1b-approved leader will push 2a to a majority of agents before being usurped by a new leader's 1a. Eventually?
- Messages dropped.
  - If things get ugly, have another round.

Dynamic Uniformity in Paxos

- Dynamic uniformity: if any node X accepts v as the consensus value, then every operational node Y also accepts v....eventually....even if X crashes.
- Suppose some subset of nodes accept v by Paxos in round S, and then all crash or disconnect.
- The S nodes made up a majority, so no subsequent round R can succeed until at least one recovers.
  - Might never recover: Paxos is not guaranteed live.
- But if enough recover for a subsequent round R to succeed, then R will have the same value v!
  - Agents log (S, v) at 2a and report it for 1b.

Paxos vs. 2PC

- The fundamental difference is that leader failure can block 2PC, while Paxos is non-blocking.
  - Someone else can take over as leader.
- Both use logging for continuity across failures.
- But there are differences in problem setting...
  - 2PC: agents have "multiple choice with veto power".
    - Unanimity is required, at least to commit.
  - Paxos: consensus value is dictated by the first leader to control a majority.
- Can we derive a nonblocking commit from Paxos?

From 2PC to Paxos Commit

- One way: if coordinator (TM) fails, each RM leads a Paxos instance so the others learn its vote.
  - Gray and Lamport, "Paxos Commit Algorithm".
- "2PC is a degenerate case of the Paxos Commit algorithm with a single coordinator, guaranteeing progress only if the coordinator is working."
- Or: add multiple-choice voting to Paxos?
  - Left as an exercise.

Ken Birman on Paxos

- "Dynamic uniformity is very costly...there are major systems (notably Paxos, the group management protocol favored in some recent Microsoft products and platforms) in which an ABCAST primitive with dynamic uniformity is the default. Such a primitive is very slow and expensive but can be used safely for almost any purpose. On the other hand, the cost is potentially so high that the resulting applications may be unacceptably sluggish compared to versions coded to use one of the cheaper primitives, particularly a non-uniform ABCAST implemented using token passing and FBCAST."

Paxos vs. ABCAST

- ABCAST: agree on the next message in a stream of messages to a group with multiple senders.
  - How to establish a total order on the messages?
  - Run a Paxos instance for each next message?
- Dynamic uniformity: if any node X accepts message m as next, then every operational node Y also accepts m as next....eventually...even if X crashes.
  - "Safe": X can take externally visible actions based on m before X knows that Y knows m is next.
  - E.g., put cash out the ATM on transaction commit.

Keith Marzullo on Paxos

- Keith Marzullo et. al. support the equivalence of Paxos consensus and dynamically uniform ABCAST.
- On why would we use consensus (e.g., Paxos):
  - "One of the most important applications of consensus is state-machine replication, where clients propose commands and servers run a sequence of consensus instances. Each instance then selects a single command for the replicated service to execute."
  - In other words, we need a total order on the commands, i.e., ABCAST delivery of the commands.

Is Paxos "Unacceptably Sluggish"?

- Paxos is slow...if multiple nodes fight for leadership (e.g., ABCAST with multiple senders).
- But: consider one Paxos instance with no conflict among leaders, and no failure:
  - takes 2.5 message rounds for full protocol
  - two disk writes (per agent) in the latency path
  - not SO bad
- But keep trying if there's a failure or conflict.
  - "Get it done as fast as you can, but take as long as it takes to get it done."

Paxos: Summary

- Non-blocking asynchronous consensus protocol.
  - Safe, and live if not “too many” failures/conflicts.
- Paxos is at the heart of many distributed and networked systems.
  - Often used as a basis for electing the primary in primary-based systems (i.e., “token passing”).
- Related to 2PC, but robust to leader failure if some leader lasts long enough to complete the protocol.
  - The cost of this robustness is related to the rate of failures and competition among leaders.

Byzantine Consensus

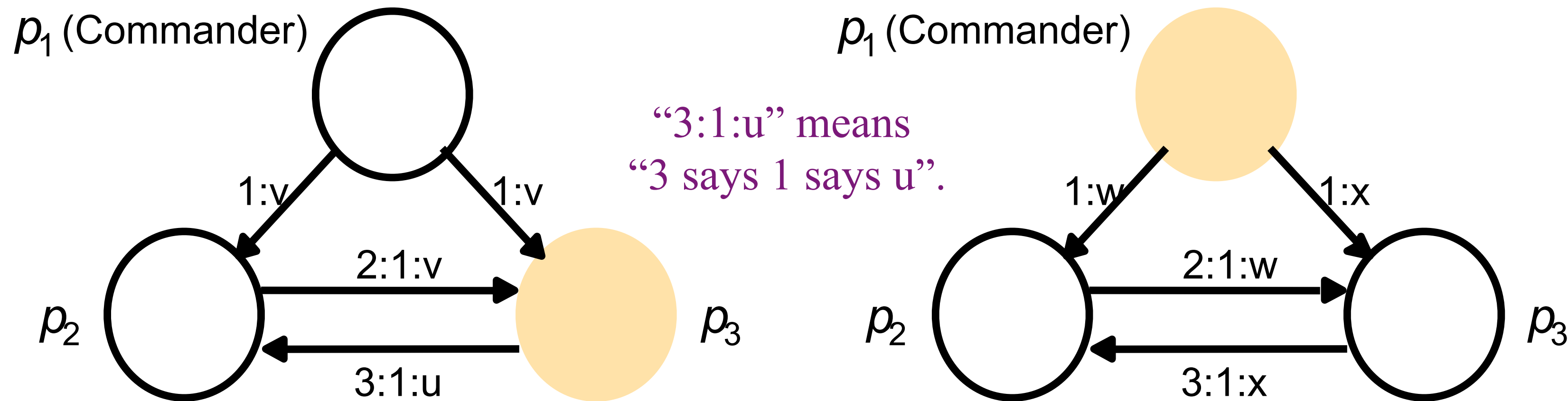
Byzantine Fault Tolerance

- What happens to 2PC or Paxos if the participants lie or act inconsistently?
  - Failure model: fail-stop or disconnect.
  - What about "Byzantine" failures? e.g., compromise or subversion by an attacker.
- "Byzantine fault tolerance" (BFT): a hot topic
  - Replicate functions, compare notes, and vote.
  - Consistent/correct outcome even if some fail.
  - Note: assumes independent failures.
    - vulnerable to "groupthink"

Lamport's 1982 Result, Generalized by Pease

- The Lamport/Pease result shows that consensus is impossible:
  - with byzantine failures,
  - if one-third or more processes fail ($N \leq 3F$),
    - Lamport shows it for 3 processes, but Pease generalizes to N.
  - even with synchronous communication.
- Intuition: a node presented with inconsistent information cannot determine which process is faulty.
- The good news: consensus can be reached if $N > 3F$, no matter what kinds of node failures occur.

Impossibility with three byzantine generals



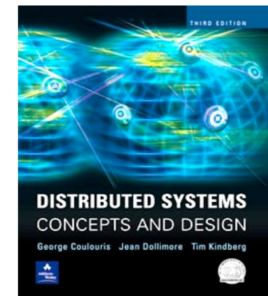
"3:1:u" means "3 says 1 says u".

Faulty processes are shown shaded

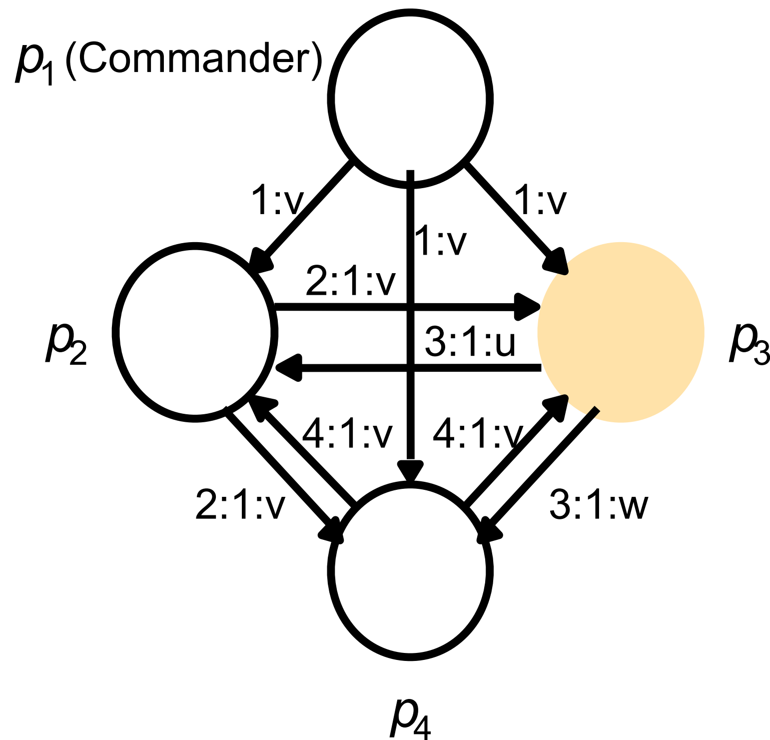
[Lamport82]

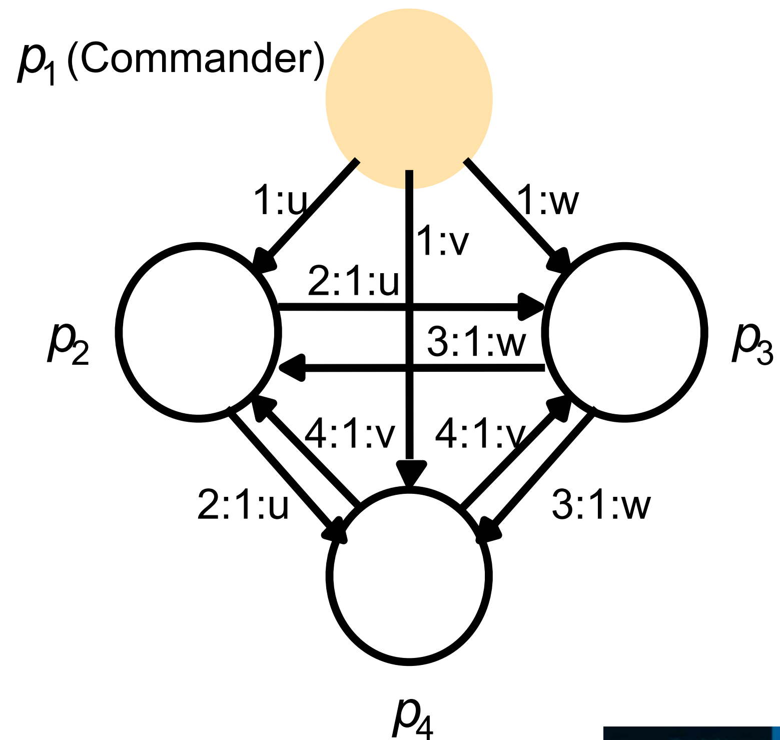
Intuition: subordinates cannot distinguish these cases.

Each must select the commander's value in the first case, but this means they cannot agree in the second case.

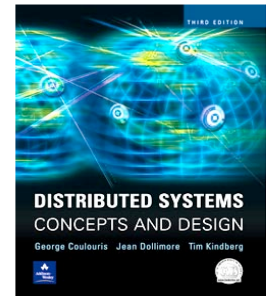


Solution with four byzantine generals





Faulty processes are shown shaded

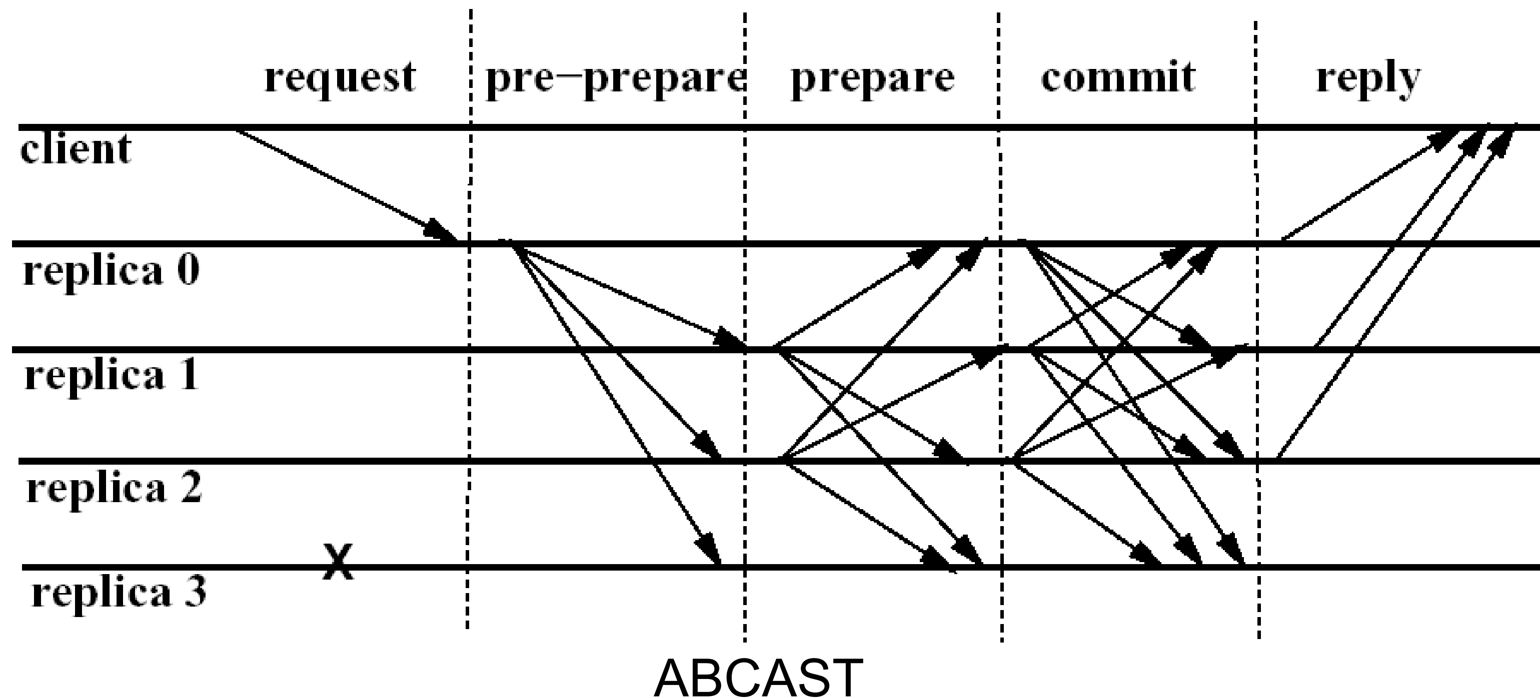


Intuition: vote.

Summary: Byzantine Consensus

- A solution exists if less than one-third are faulty (N > 3F).
- But FLP applies: it works only if communication is synchronous.
- Like fail-stop consensus, the algorithm requires F+1 rounds.
- The algorithm is very expensive and therefore impractical.
    - Number of messages is exponential in the number of rounds.
- Signed messages make the problem easier (authenticated byzantine).
    - In general case, the failure bounds (N > 3F) are not affected.
    - Practical algorithms exist for N > 3F. [Castro&Liskov]

Practical BFT



ABCAST

- Castro/Liskov is a widely cited paper with lots of follow-on work. Practical BFT consensus using Message Authentication Codes and voting.

Practical BFT: Overview

- Nodes share a sequence of views of group membership, with a designated leader/primary.
  - If the primary appears to fail, start a new view (like Paxos, and Viewstamped Replication [Liskov]).
- Clients send requests to primary.
  - Primary sequences all requests for state machine replication model.
    - i.e., in essence, dynamically uniform ABCAST using a Paxos-elected primary as a sequencer.
- Secondaries vote on whether to commit, then send their view on the outcome of the vote to the client.

BFT Protocol Summary

- Primary multicasts “prepare” message for next request.
- Replicas multicast their acceptance to all replica.
- When a replica sees enough prepare-accepts, it commits locally and multicasts its commit vote to all replicas.
- When a replica sees enough commit votes, it considers the request to have committed.
- Each replica sends its view of the outcome back to the client.
- Client compares the outcomes.
- Lots of grunge around view changes.

Weak Synchrony

- Castro and Liskov: Consensus protocols can be "live if delay(t) does not grow faster than (t) indefinitely". This is a "weak synchrony assumption" that is "likely to be true in any real system provided that network faults are eventually repaired, yet it enables us to circumvent the impossibility result" in FLP.

Aside: View Changes

- How to propagate knowledge of failure and recovery events to other nodes?
  - Surviving nodes should agree on the new view (regrouping).
  - Convergence should be rapid.
  - The regrouping protocol should itself be tolerant of message drops, message reorderings, and failures.
    - and ordered with respect to message delivery
  - The regrouping protocol should be scalable.
  - The protocol should handle network partitions.
- This is another instance of a consensus problem.
- Explored in depth for process group and group membership systems (e.g., virtual synchrony).

Final Thoughts: CAP

consistency

C

Fox&Brewer “CAP Theorem”:
C-A-P: choose two.

Claim: every distributed system is on one side of the triangle.

CA: available, and consistent, unless there is a partition.

CP: always consistent, even in a partition, but a reachable replica may deny service without agreement of the others (e.g., quorum).

A

Availability

AP: a reachable replica provides service even in a partition, but may be inconsistent.

P

Partition-resilience

CAP Examples

- CP: Paxos, or any consensus algorithm, or state machine replication with a quorum required for service.
  - Always consistent, even in a partition. But might not be available, even without a partition.
- AP: Bayou
  - Always available if any replica is up and reachable, even in a partition. But might not be consistent, even without a partition.
- CA: consistent replication (e.g., state machine with CATOCS) with service from any replica.
  - What happens in a partition?

CAP: CA

- A CA system is consistent and available, but may become inconsistent in a partition.
  - Basic state machine replication with service from any replica.
  - Coda read-one-write-all-available replication.
- These are always consistent in the absence of a partition.
  - But they could provide service at two or more isolated/ conflicting replicas in a partition ("split brain").
- To preserve consistency in a partition requires some mechanism like quorum voting to avoid a "split brain".
  - That makes the system a CP: it must deny service when it does not have a quorum, even if there is no partition.