Agreement in Distributed Systems: Atomic Commitment

Lecture 8

Agreement in Distributed Systems

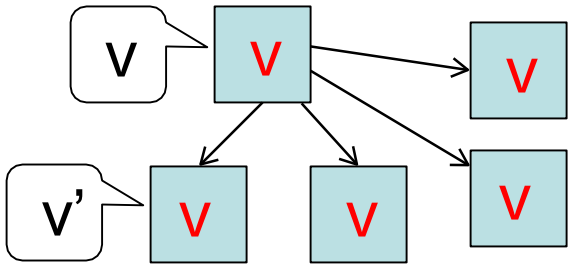
- The crown problem of distributed systems

- Despite having different views of the world, all nodes in a distributed system must act in concert
  - All replicas that store the same object O must apply all updates to O in the same order (consistency)
  - All nodes involved in a transaction must either commit or abort their portion of the transaction (atomicity)

- All that, despite FAILURES
  - Nodes can restart, die, be slow
  - Networks can be slow, as well (but we assume they're reliable here, i.e., all network messages are eventually received)

The Agreement Problem

- Some nodes *propose* values (or actions) by sending them to the others
- All nodes must *decide* whether to accept or reject those values



- Examples of values to agree on:
  - Whether or not to commit a transaction to a DB
  - The value of the clock
  - Who has a lock in a distributed lock service among multiple clients that request it almost simultaneously
  - Whether to move to the next stage of a distributed alg. (a barrier)

Agreement Requirements

- Safety (correctness)
  - All nodes agree on the same value
  - The agreed value X has been proposed by some node
- Liveness (fault tolerance, availability)
  - If less than some fraction of nodes crash, the rest should still reach agreement
- I.e., agreement aims to give the behavior of a single machine with the fault-tolerance of multiple machines

Failure Models

- For these classes, we define agreement in the context of two failure models:
- Synchronous systems: machines and networks can only be delayed by a bounded time
  - I.e., using a sufficiently large timeout, you can tell with certainty whether the machine crashed or it or the network is just slow
- Asynchronous systems: machines and networks can be arbitrarily delayed
  - There's no way you can tell whether a machine has crashed or is just slow
- We'll see that different safety/liveness properties are possible under different models

Agreement problem: Two Flavors

- **Atomic commitment problem**: participants need to agree on a value, but they have specific constraints on whether they can accept any particular value.
  - Ex: A group's decision on when to meet is probably an atomic commitment problem, because each participant has his/her own calendar constraints.

- **Consensus problem**: participants need to agree on a value, but they are willing and capable to accept any value.
  - Ex: A group's decision on where to meet (say, which specific room on campus of those that are of suitable size) can probably be cast as a consensus problem: most likely no one cares where they meet, but they all need to agree on the same value.

Atomic Commitment: Distributed Transactions

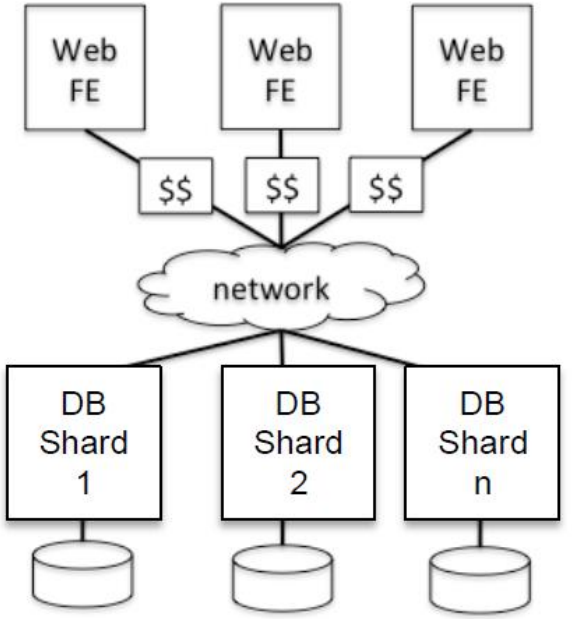
- Previously, we talked about single node database
  - ACID properties of transactions, Isolation with 2PL

- In practice, databases are distributed
  - Data sharded/partitioned for *scalability*
  - Shards replicated for *fault tolerance*

- Distributing data creates semantic challenges

Semantic Challenges

- Architecture of a banking application
  - FEs accept requests from end-users' browsers and process them concurrently
  - The data stored in the DB is *sharded*, say by user ID
    - All data of the first third of the users in ID space is stored in Shard 1, 2nd in shard 2, …
- Sharding provides scalability
  - If workload involves accessing accounts from one of the shards (e.g., users check out mostly their accounts), various DB shards can handle data requests in parallel from multiple different FEs
- But what if transaction spans multiple shards?
  - Each individual shard implements an ACID engine.
  - ***How does each shard decide whether to commit? When to release locks?***
  - ***How is atomicity provided under failures?***



One-phase commit

- Transaction coordinator
  - Begins transaction, assigns unique transaction ID
  - Responsible for commit/abort
  - Many systems allow any client to be the coordinator for its own transactions
- Participants
  - The servers with the data used in the distributed transaction
- “One-phase commit”
  - Coordinator broadcasts “commit!” to participants until all reply
- Problem: What happens if one participant fails?
  - Other participants cannot undo what they have already committed

Two-phase commit: Intuitive Example

- You want to organize outing with 4 friends at 6pm Tuesday
  - Goal: go out only if all friends can make it
- What do you do?
  - Call each of them and ask if can do 6pm on Tuesday (voting phase)
  - If all can do Tuesday, call each friend back to ACK (commit)
  - If one can’t do Tuesday, call other three to cancel (abort)
- Critical details:
  - While you were calling everyone to ask, people who’ve promised they can do 6pm Tuesday must reserve that slot
  - You need to remember the decision and tell anyone whom you haven’t been able to reach during commit/abort phase

2PC

- The commit-step itself is two phases
- Phase 1: Voting
  - Each participant prepares to commit, and votes on whether or not it can commit
- Phase 2: Committing
  - Each participant actually commits or aborts

2PC Operations

- **`canCommit?(T)`** -> yes/no
  - Coordinator asks a participant if it can commit
- **`doCommit(T)`**
  - Coordinator tells a participant to actually commit
- **`doAbort(T)`**
  - Coordinator tells a participant to abort
- **`haveCommitted(participant,T)`**
  - Participant tells coordinator it actually committed
- **`getDecision(T)`** -> yes/no
  - Participant can ask coordinator if T should be committed or aborted

2PC: Voting

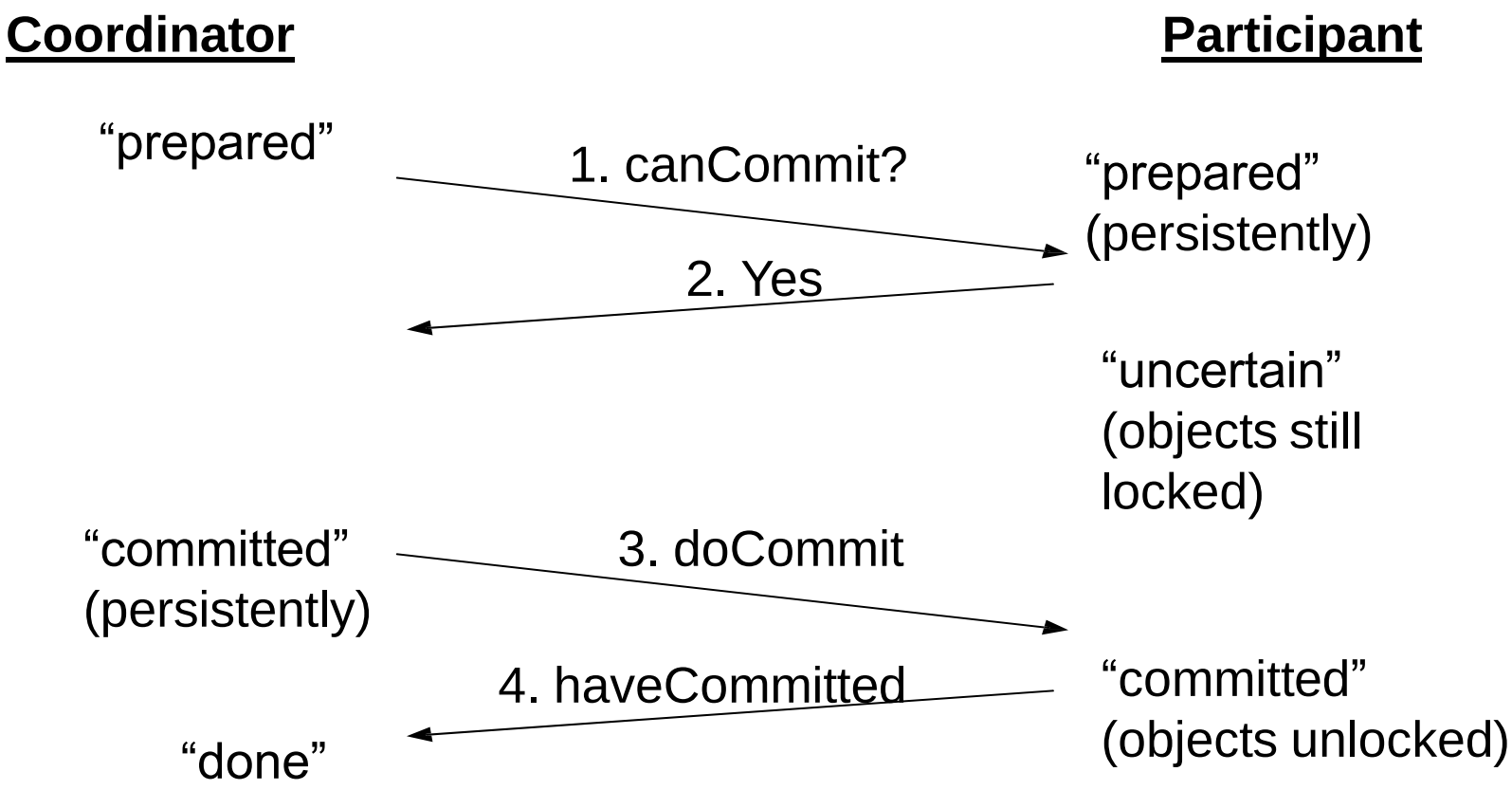
- Coordinator asks each participant: **`canCommit?(T)`**
- Participants must prepare to commit using permanent storage before answering yes
  - Done by writing “prepare-yes” message to the log
- Objects are still locked: Each participant uses 2PL for its objects
  - Once a participant votes “yes”, it is not allowed to cause an abort
- Outcome of **`T`** is uncertain until **`doCommit`** or **`doAbort`**
  - Other participants might still cause an abort

2PC: Commit

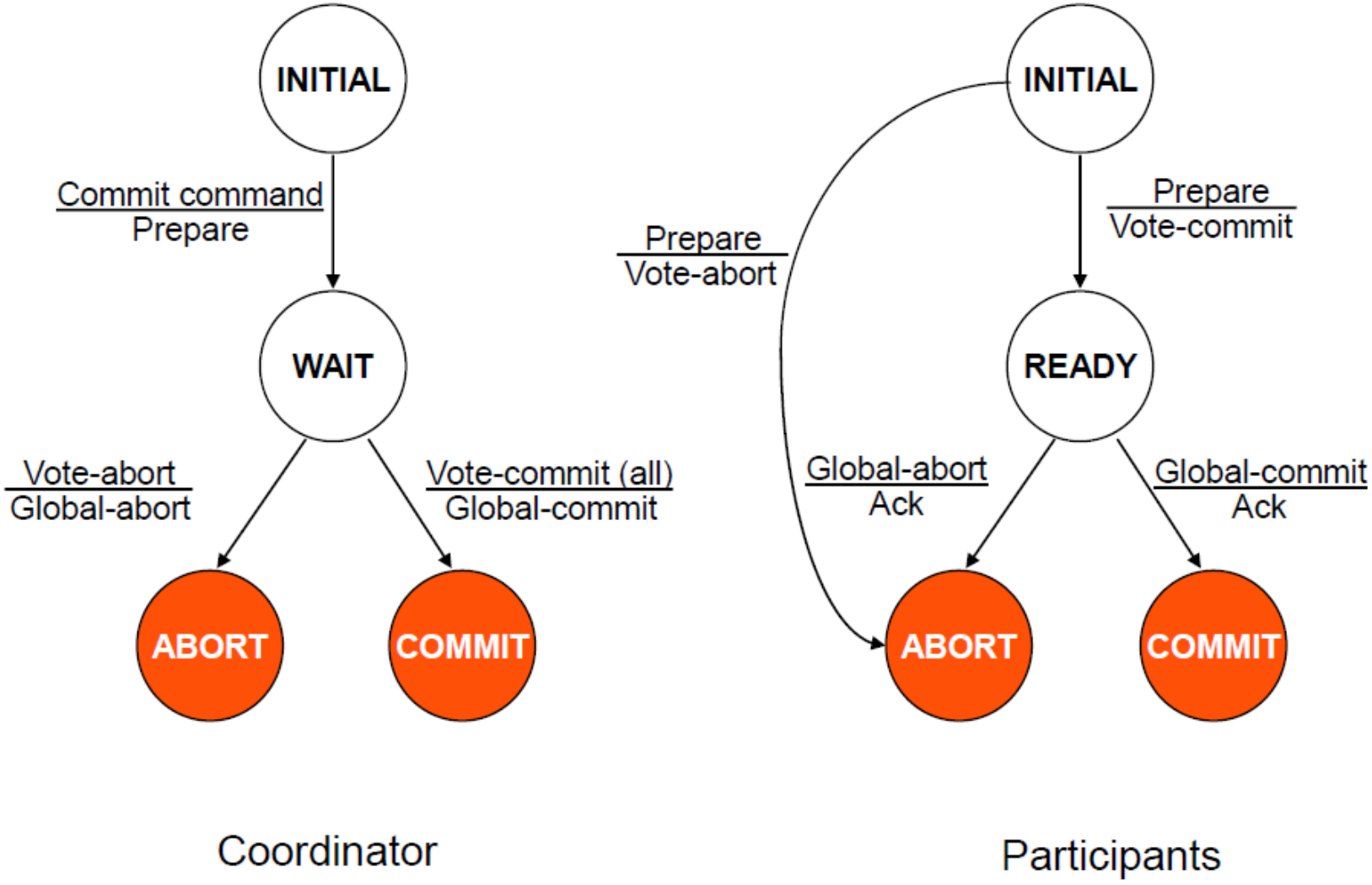
- The coordinator collects all votes
  - If unanimous “yes”, causes commit
  - If any participant voted “no”, causes abort

- The fate of the transaction is decided atomically at the coordinator, once all participants vote
  - Coordinator records fate using permanent storage
  - Then broadcasts **`doCommit`** or **`doAbort`** to participants

2PC event timeline



2PC state transition



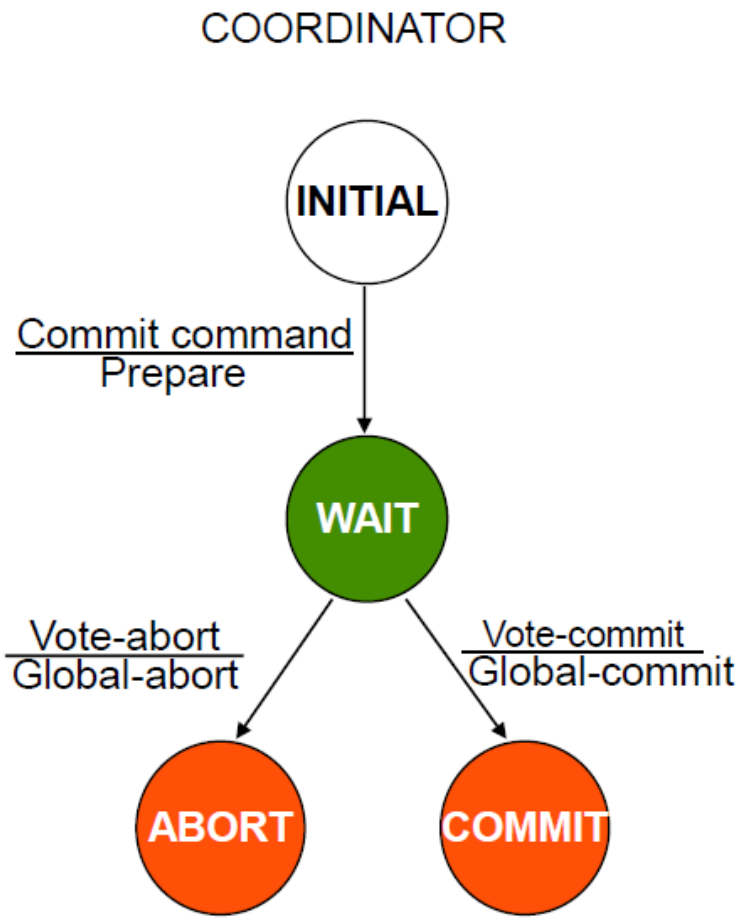
Recovery in Two-Phase Commit

- Easy: just log the state-changes
  - Participants: prepared, uncertain, committed/aborted
  - Coordinator: prepared, committed/aborted, done

- Two cases:
  - Recovery after timeouts
  - Recovery after crashes and reboots
  - Note: you can’t differentiate between the above in a realistic, asynchronous network!

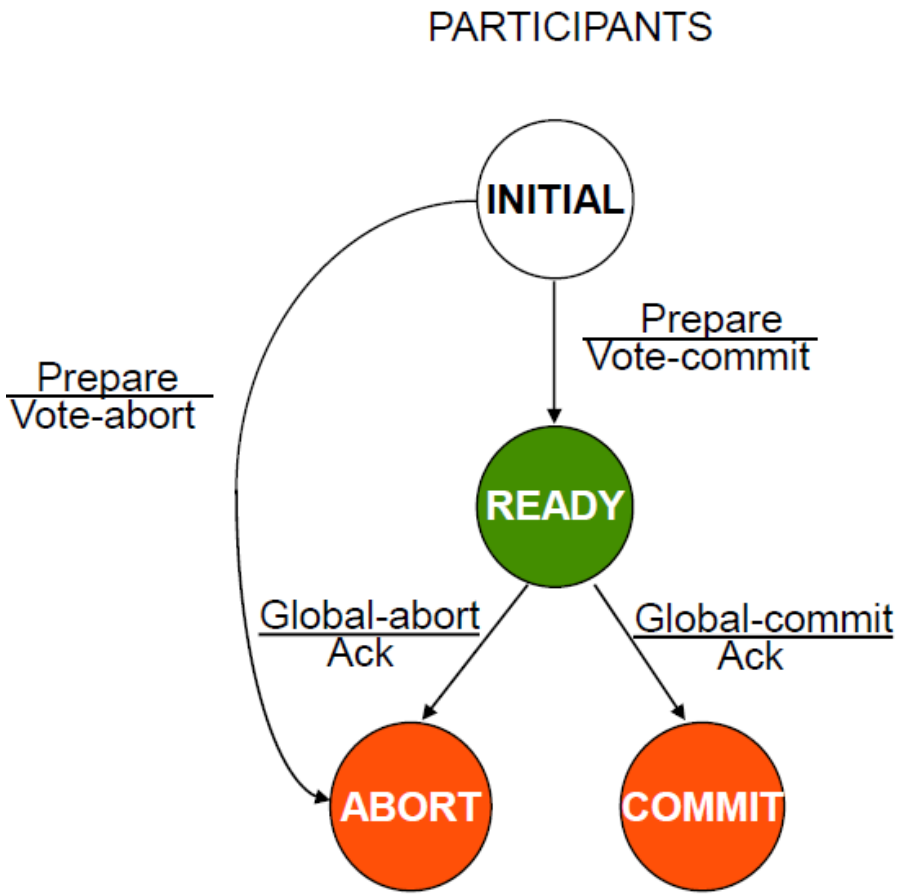
2PC Timeout in Coordinator

- Timeout in INITIAL
  - Who cares
- Timeout in WAIT
  - Cannot unilaterally commit
  - Can unilaterally abort
- Timeout in ABORT or COMMIT
  - Stay blocked and wait for the acks



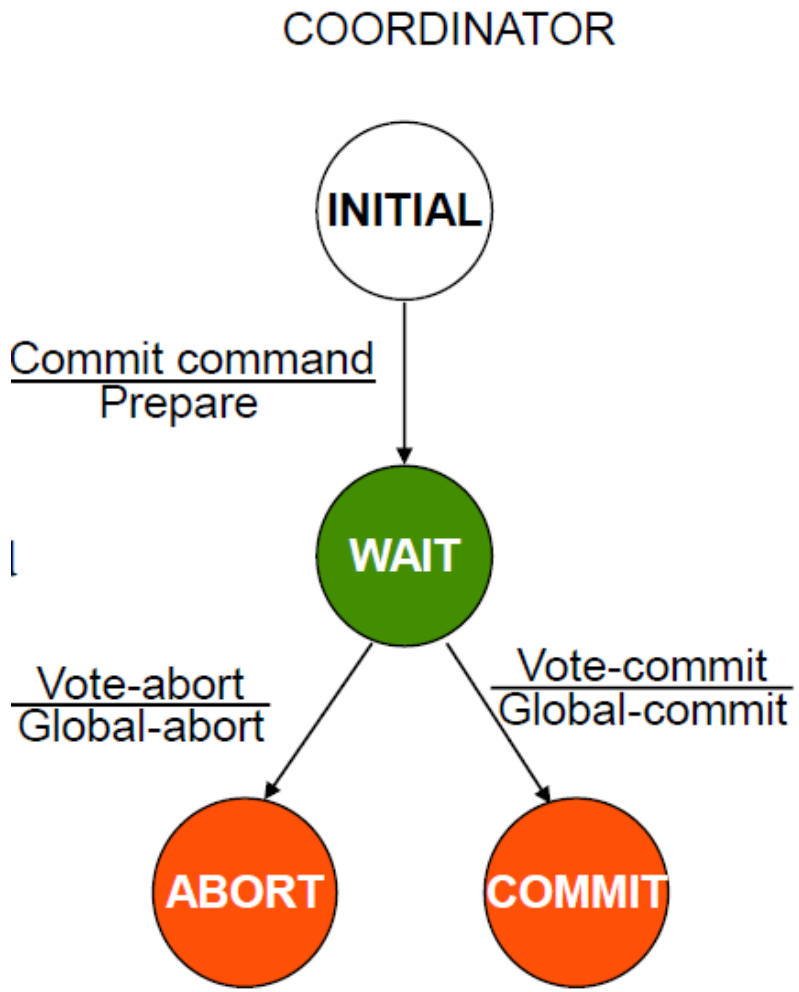
2PC Timeout in Participant

- Timeout in INITIAL
  - Coordinator must have failed in INITIAL state
  - Unilaterally abort
- Timeout in READY
  - Participant is in the “uncertain” period
  - Must use a **termination protocol** as participant does not know coordinator’s decision.
- Termination protocol
  - Blocking: Wait until communication with TC is re-established.
  - Cooperative: (P) Asks other participants (Q) for outcome
    - If Q is COMMIT state, P can move to COMMIT
    - If Q is ABORT state, P can move to ABORT
    - If Q is INIT state, P can move to ABORT
    - If Q is READY state, P has to contact another Q’
  - If all Q’s are in READY STATE, protocol blocks



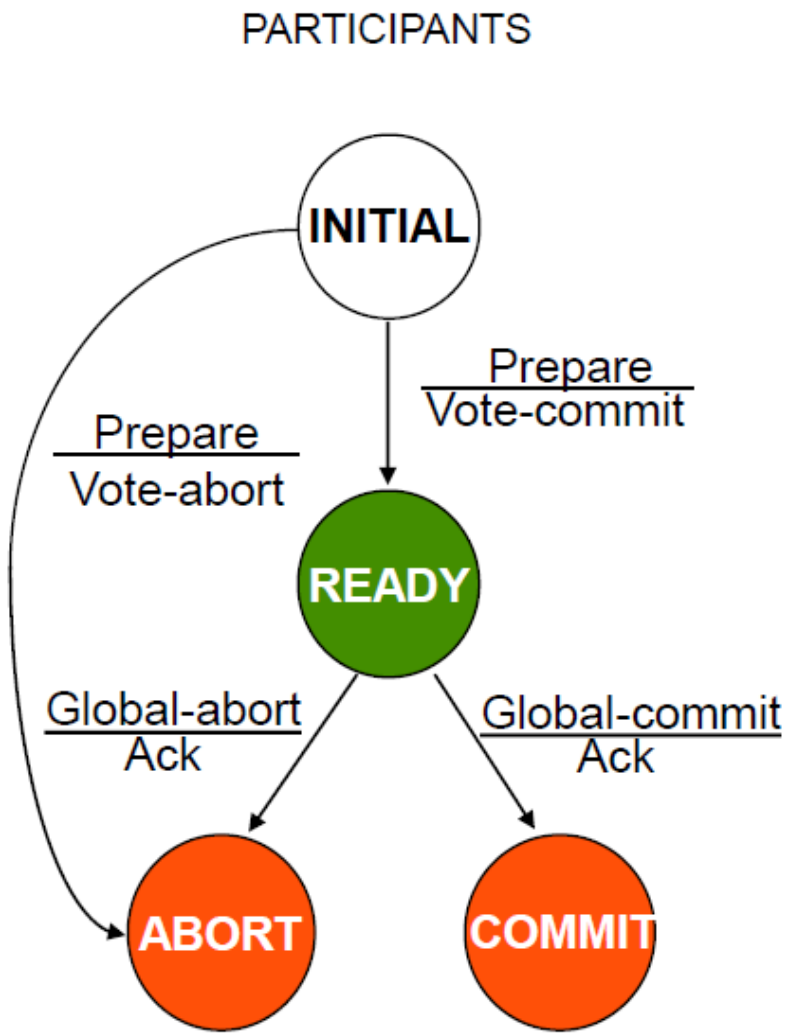
2PC Coordinator Crash Recovery

- Failure in INITIAL
  - Start the commit process upon recovery
- Failure in WAIT
  - Restart the commit process upon recovery
- Failure in ABORT or COMMIT
  - Nothing special if all the acks have been received
  - Otherwise the termination protocol is involved



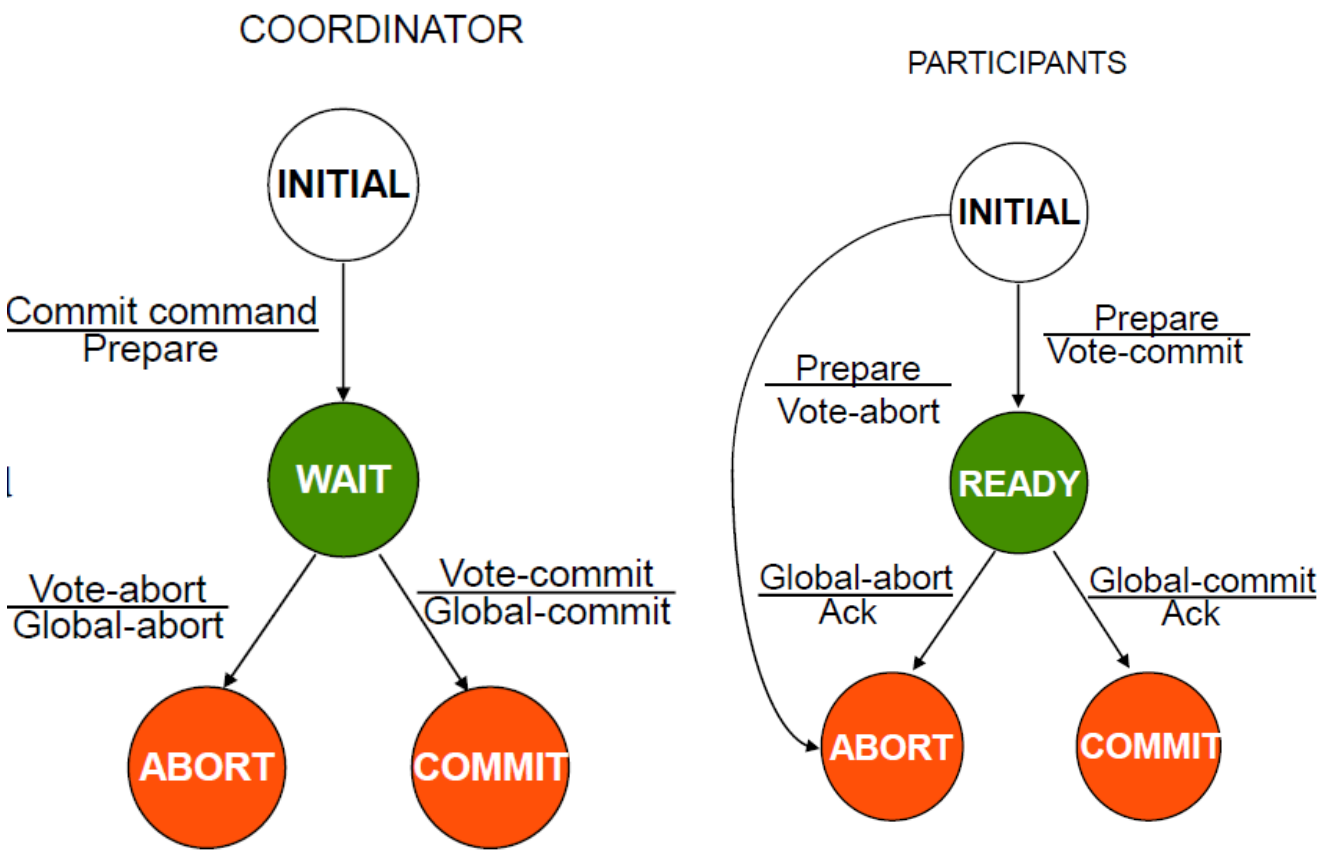
2PC Participant Crash Recovery

- Failure in INITIAL
  - Unilaterally abort upon recovery
- Failure in READY
  - The coordinator has been informed about the local decision
  - Treat as timeout in READY state and invoke the termination protocol
- Failure in ABORT or COMMIT
  - Nothing special needs to be done



Some additional cases to think about

- Ex. 1: Coordinator site fails after writing “begin_commit” log and before sending “prepare” command
  - Treat it as a failure in WAIT state; send “prepare” command
- Ex. 2: Participant site fails after writing “ready” record in log but before “vote-commit” is sent
  - treat it as failure in READY state
- Ex.3: Participant site fails after writing “abort” record in log but before “vote-abort” is sent
  - no need to do anything upon recovery
- In general, arises due to non-atomicity of log and msg send actions
  - Complete 2PC protocol has to deal with these

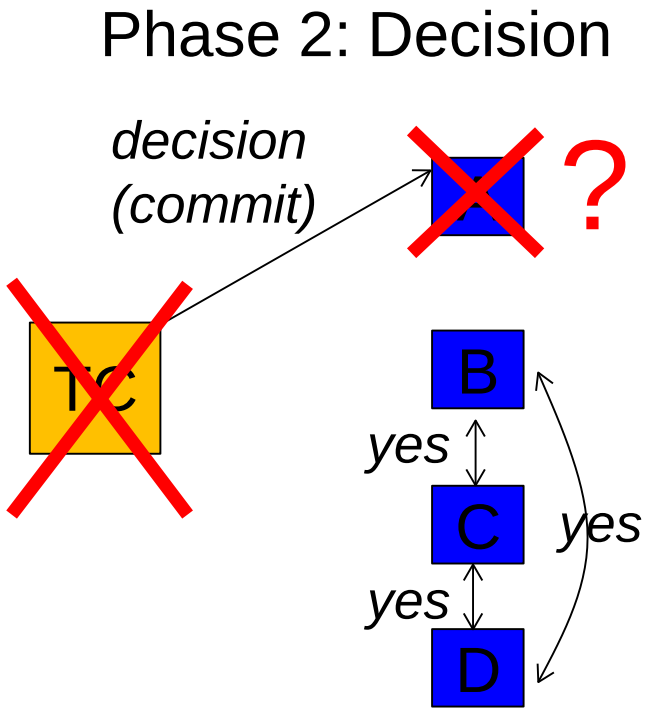


2PC Performance

- 2PC is expensive
  - Holding locks while executing a distributed protocol.
  - If distribution is over geographies, 2PC is very expensive
- Time complexity
  - 2PC requires three message delivery latencies: PREPARE, YES/NO, ABORT/COMMIT
  - On failure, additional rounds may be necessary to recover
- Message complexity
  - common case for n participants plus 1 coordinator: 3n messages

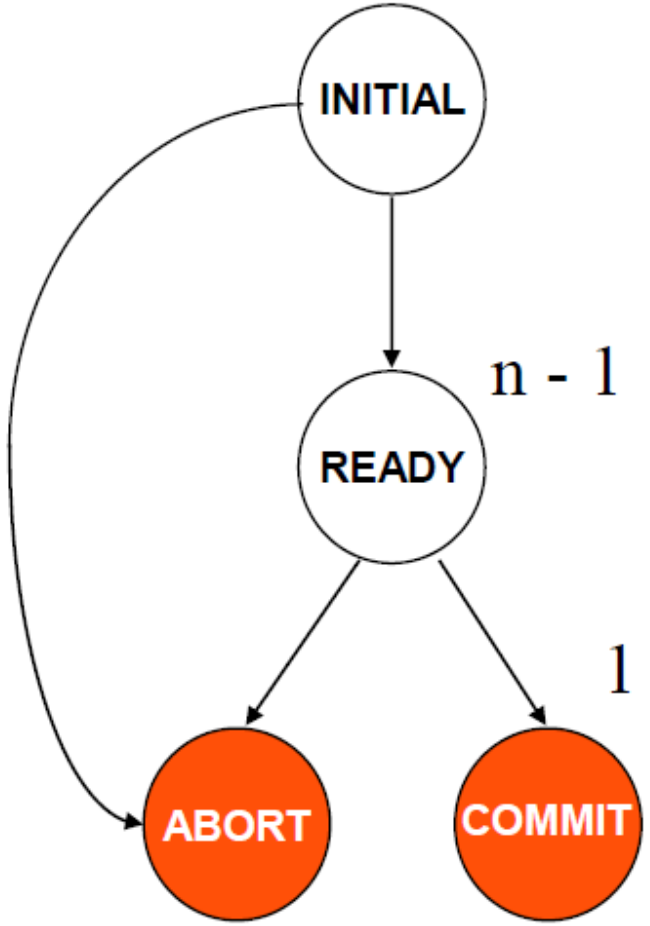
Blocking nature of 2PC

- Scenario:
  - TC sends commit decision to A, A gets it and commits, and then both TC and A crash
  - B, C, D, who voted Yes, now need to wait for TC or A to reappear while holding locks
  - They cannot commit as A could have sent NO to canCommit
  - They cannot abort as A could have sent YES to canCommit, committed, and some other transaction could have read committed data making it impossible for A to abort.
- Worst case: TC is both a coordinator and a participant
  - Protocol is vulnerable to a single-node failure (the TC's failure)!
  - If that takes a long time (e.g., a human needs to replace a hardware component), then the protocol is stuck and availability goes down
- This is why 2 phase commit is called a blocking protocol
  - Blocking means that it doesn't make progress during the failure



Blocking & Fault Tolerance of 2PC

- Generalizing the problem for 2PC
  - it is possible that a coordinator failure can result in n-1 of the participants being in READY state and 1 participant in COMMIT state.
  - The system MUST commit at this point. Therefore, it is not safe to abort the n-1 participants in the READY state if the participant in COMMIT also fails.
  - Only choice is to block even if one (or a few) machines fail
- In context of agreement requirements, 2PC is not fault tolerant: 2PC is safe, but not live



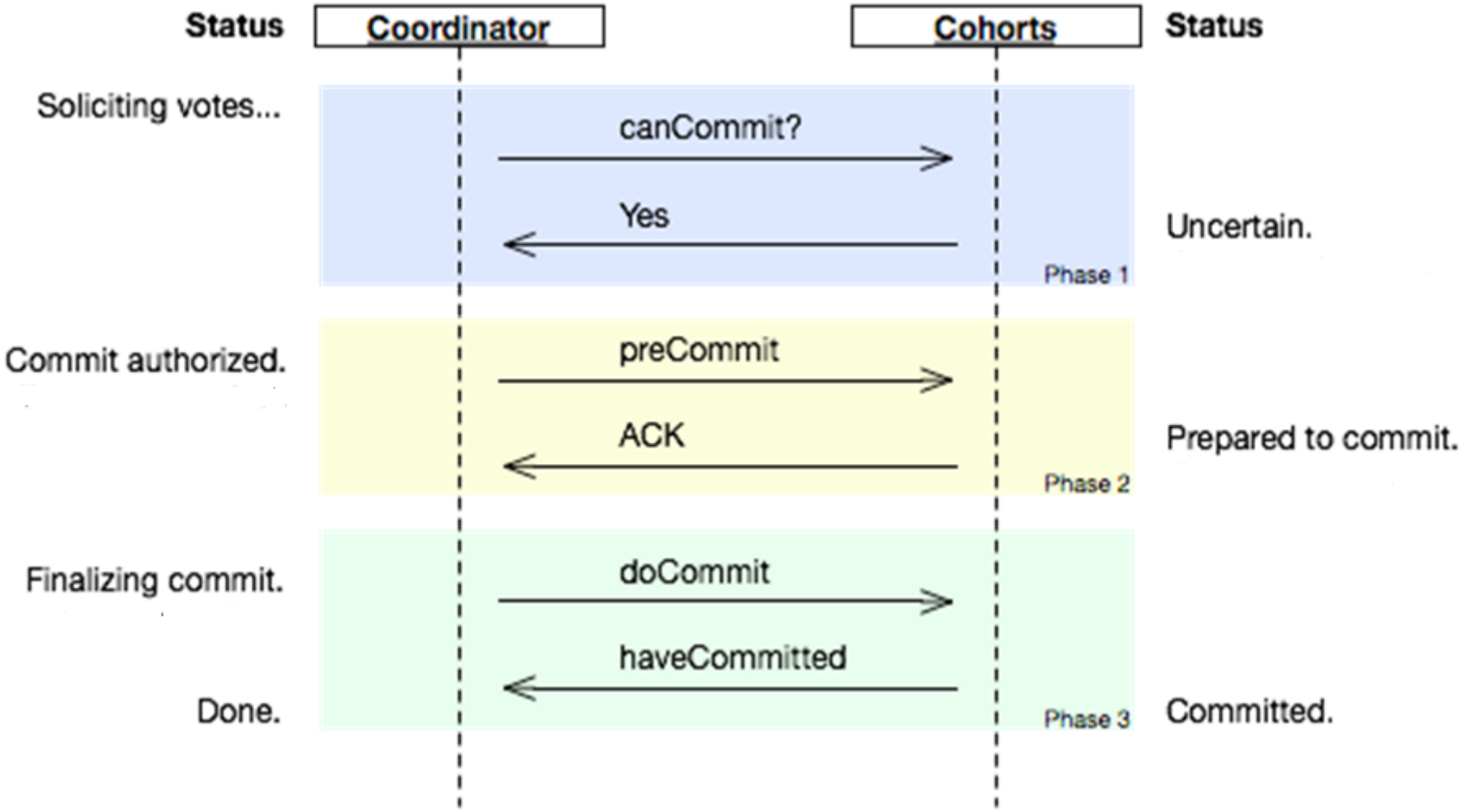
Fixing 2PC with 3PC

- Goal: Turn 2PC into a live (non-blocking) protocol
  - 3PC should never block on node failures as 2PC did

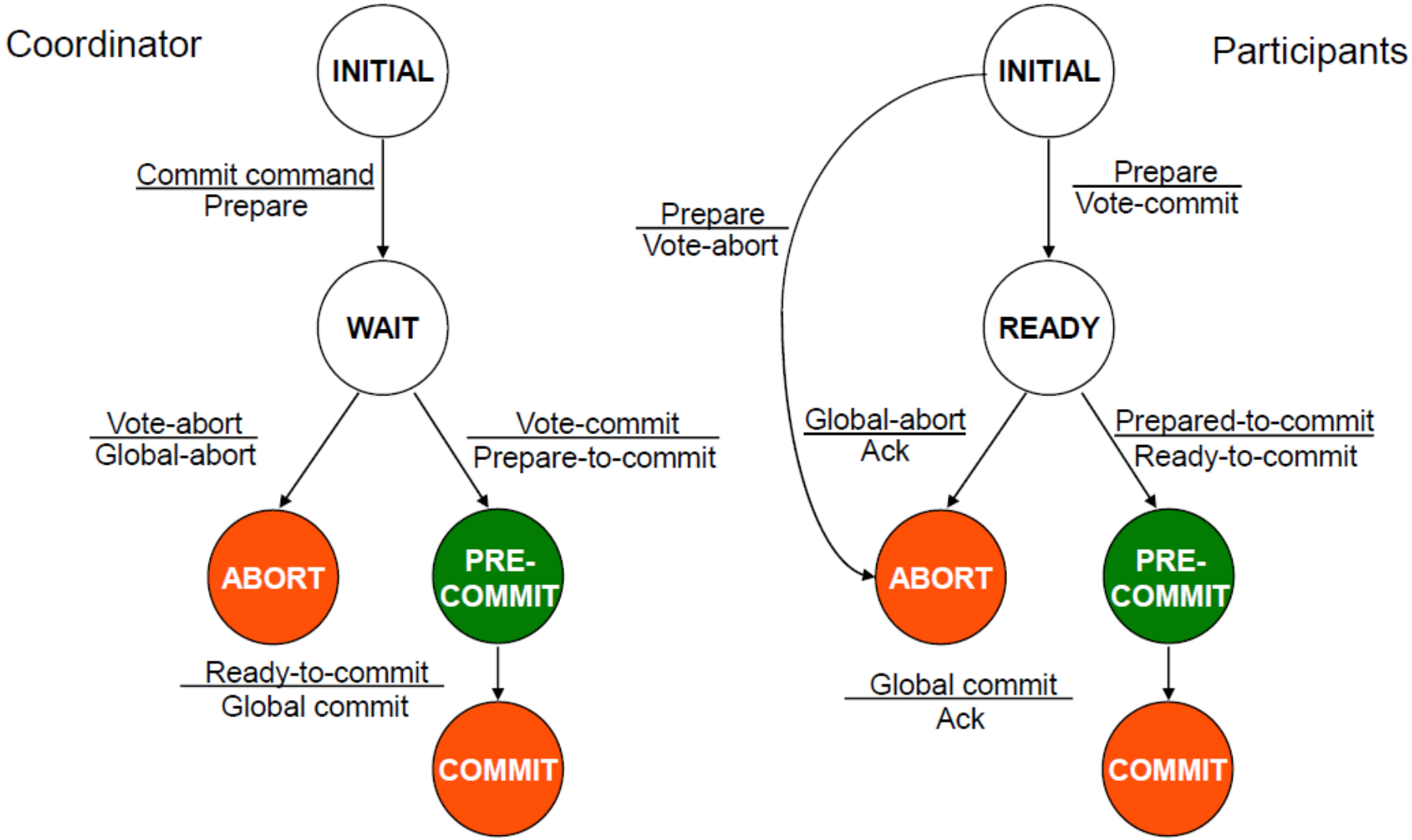
- Insight: 2PC suffers from allowing nodes to irreversibly commit an outcome before ensuring that the others know the outcome, too

- Idea in 3PC: split “commit/abort” phase into two phases
  - First communicate the outcome to everyone
  - Let them commit only after everyone knows the outcome

3PC timeline

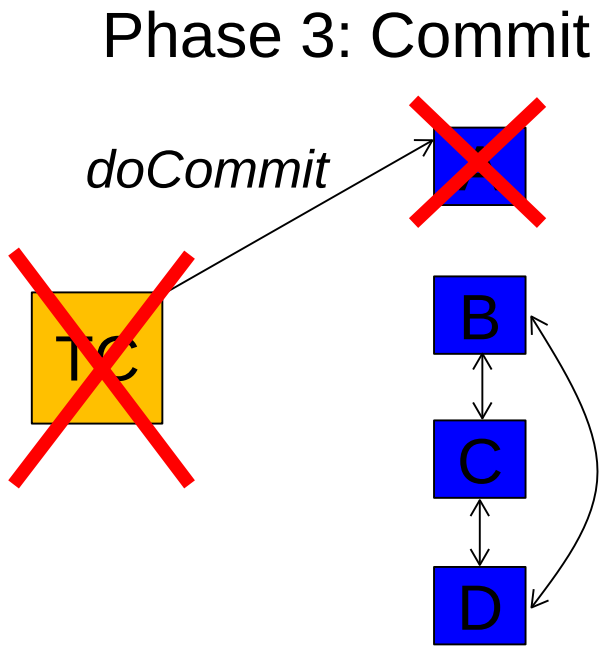


3PC state transitions



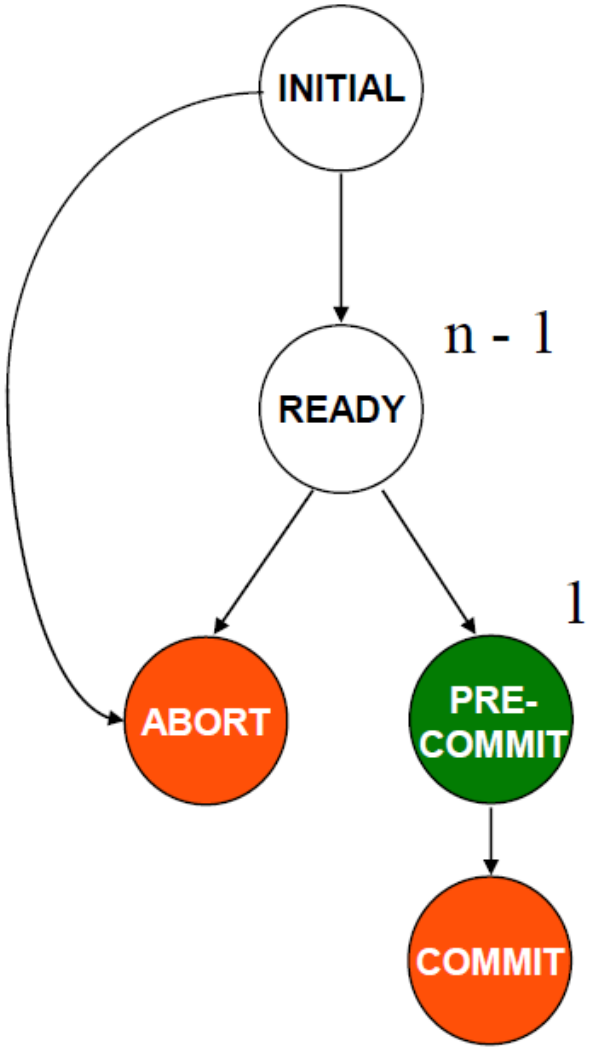
Can 3PC Solve Blocking 2PC Scenario?

- Assuming same scenario as before (TC, A crash), can B/C/D reach a safe decision when they time out?
- If one of them has received preCommit, they can all commit
  - Precommit is sent only if response to canCommit is YES from all participants
- If none of them has received preCommit, they can all abort
  - This is safe, because we know A couldn't have received a doCommit, so it couldn't have committed



3PC & Fault Tolerance: Liveness

- In 3PC a coordinator failure can leave the system with n-1 participants in READY and 1 in PRE-COMMIT.
- Even if the participant in PRE-COMMIT fails, the system can abort as the participant in the PRE-COMMIT state can still abort.
- If instead there is any alive participants in PRE-COMMIT, then the transaction can commit without blocking on the failed nodes
- Liveness (availability): Yes
  - Doesn't block, it always makes progress by timing out



3PC & Fault Tolerance: Safety

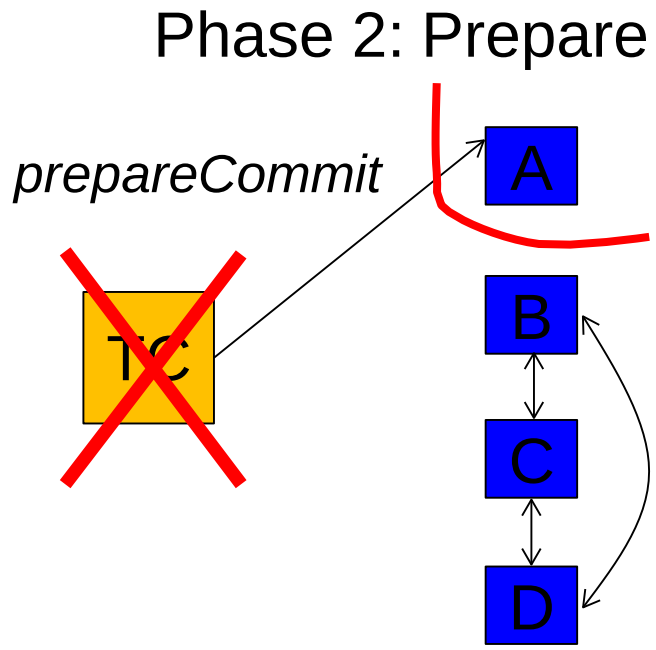
- Safety (correctness): Not in all cases
  - 3PC is safe for node crashes including TC + participant
  - But not in other cases

- Two examples of unsafety in 3PC:
  - A hasn't crashed, it's just offline
  - TC hasn't crashed, it's just offline

} Network partitions

3PC with Network Partitions

- One example scenario:
  - A receives prepareCommit from TC
  - Then, A gets partitioned from B/C/D and TC crashes
  - None of B/C/D have received prepareCommit, hence they all abort upon timeout
  - A is prepared to commit, hence, according to protocol, after it times out, it unilaterally decides to commit
- Similar scenario with partitioned, not crashed, TC



Safety vs. Liveness

- So, 3PC is doomed for network partitions
  - The way to think about it is that this protocol's design trades safety for liveness
- Remember that 2PC traded liveness for safety
- Can we design a protocol that's both safe and live?
- Well, it turns out that it's impossible in the most general case!

Fischer-Lynch-Paterson [FLP'85] Impossibility Result

- It is impossible for a set of processors in an asynchronous system to agree on a binary value, even if only a single process is subject to an unannounced failure

- The core of the problem is asynchrony
  - It makes it impossible to tell whether or not a machine has crashed (and therefore it will launch recovery and coordinate with you safely) or you just can't reach it now (and therefore it's running separately from you, potentially doing stuff in disagreement with you)

- For synchronous systems, 3PC can be made to guarantee both safety and liveness!
  - When you know the upper bound of message delays, you can infer when something has crashed with certainty

FLP Insights

- What FLP says: you can't guarantee both safety and progress when there is even a single fault at an inopportune moment
- What FLP doesn't say: in practice, how close can you get to the ideal (always safe and live)?
- Consensus protocols like Paxos get close in practice
  - More on this next lecture