Distributed Transactions: 3 Phase Commit and Beyond

CS 475, Spring 2019
Concurrent & Distributed Systems

GEORGE MASON UNIVERSITY

Agreement Generally

- Most distributed systems problems can be reduced to this one:
  - Despite being separate nodes (with potentially different views of their data and the world)…
  - All nodes that store the same object O must apply all updates to that object in the same order (consistency)
  - All nodes involved in a transaction must either commit or abort their part of the transaction (atomicity)
- Easy?
  - … but nodes can restart, die or be arbitrarily slow
  - … and networks can be slow or unreliable too

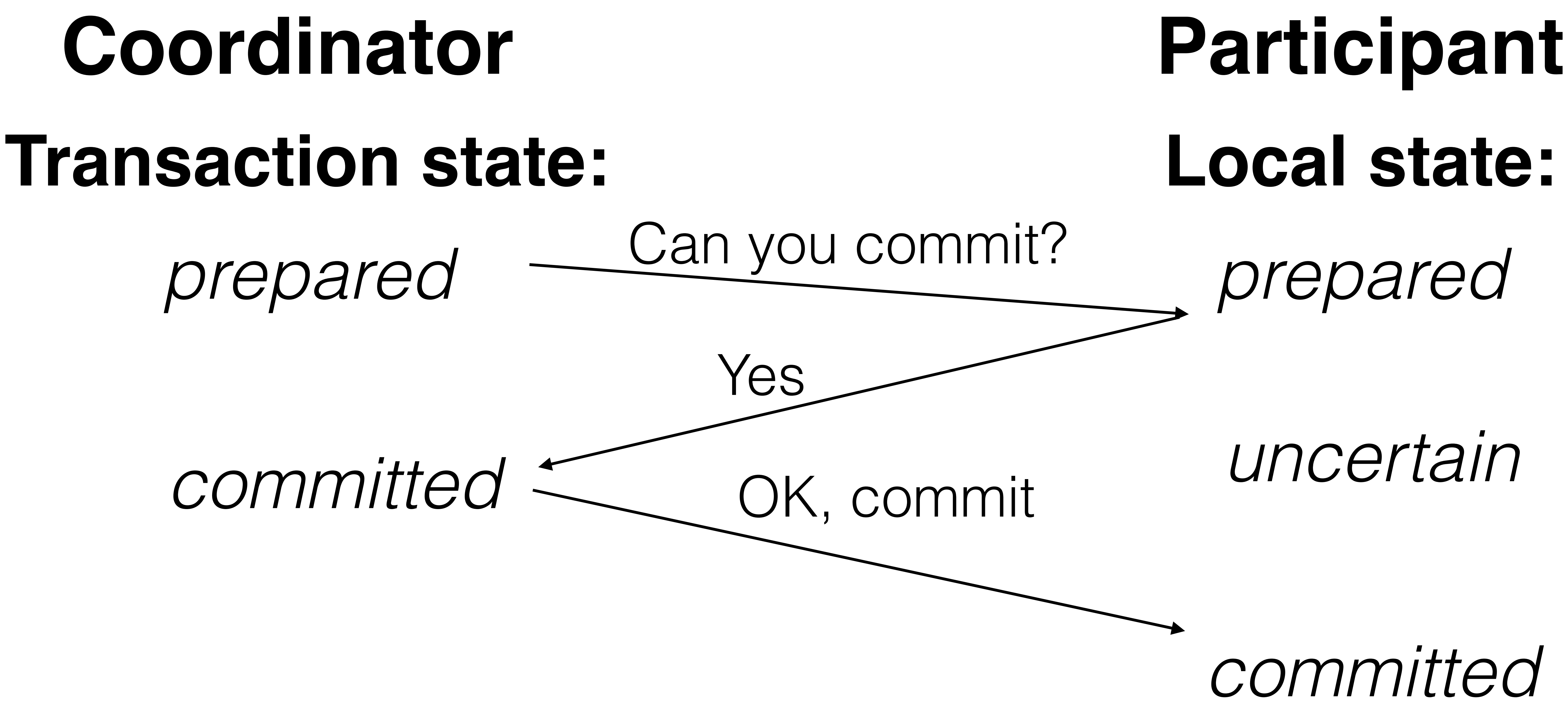
Properties of Agreement

- 2 kinds of properties, just like for mutual exclusion:
- Safety (correctness)
  - All nodes agree on the same value (which was proposed by some node)
- Liveness (fault tolerance, availability)
  - If less than N nodes crash, the rest should still be OK

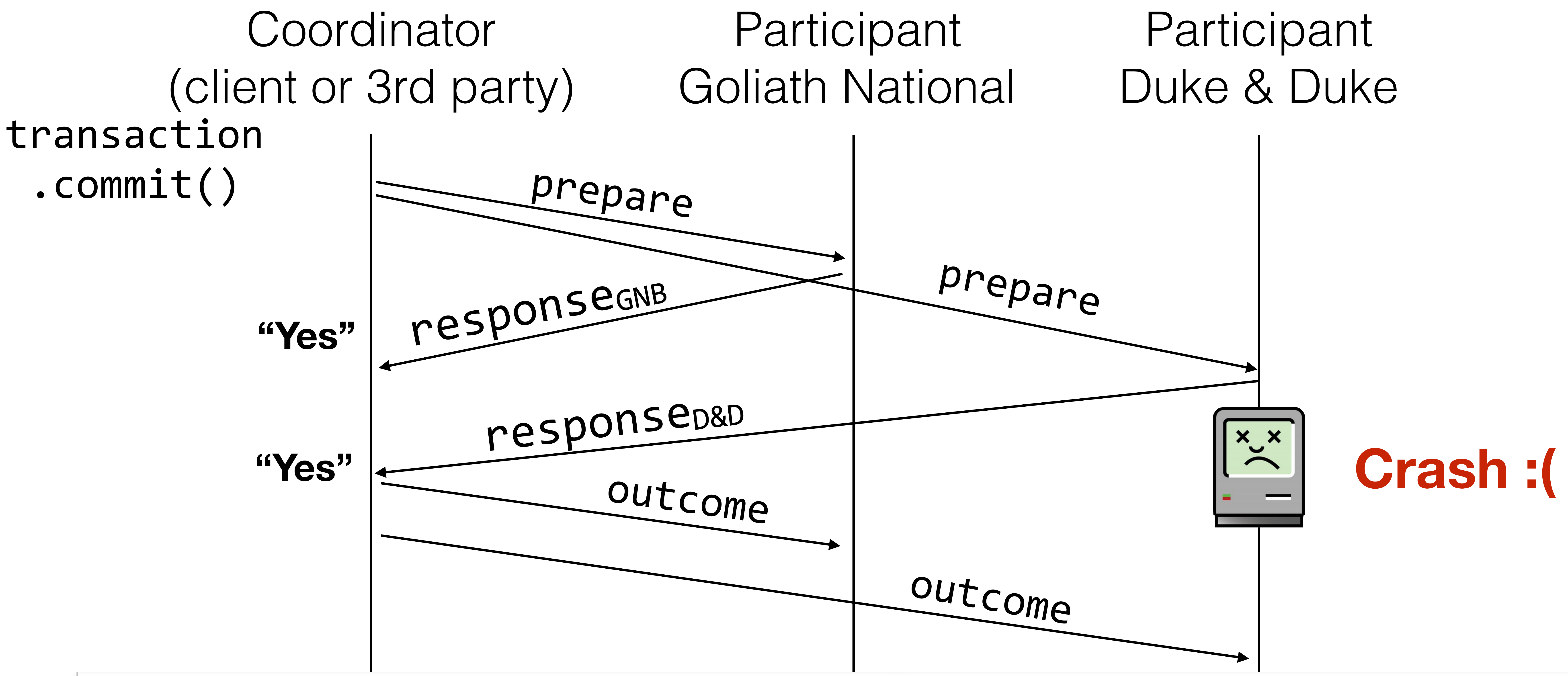
2-Phase Commit

- Separate the commit into two steps:
- 1: Voting
  - Each participant prepares to commit and votes of whether or not it can commit
- 2: Committing
  - Once voting succeeds, every participant commits or aborts
- Assume that participants and coordinator communicate over RPC

2PC Event Sequence



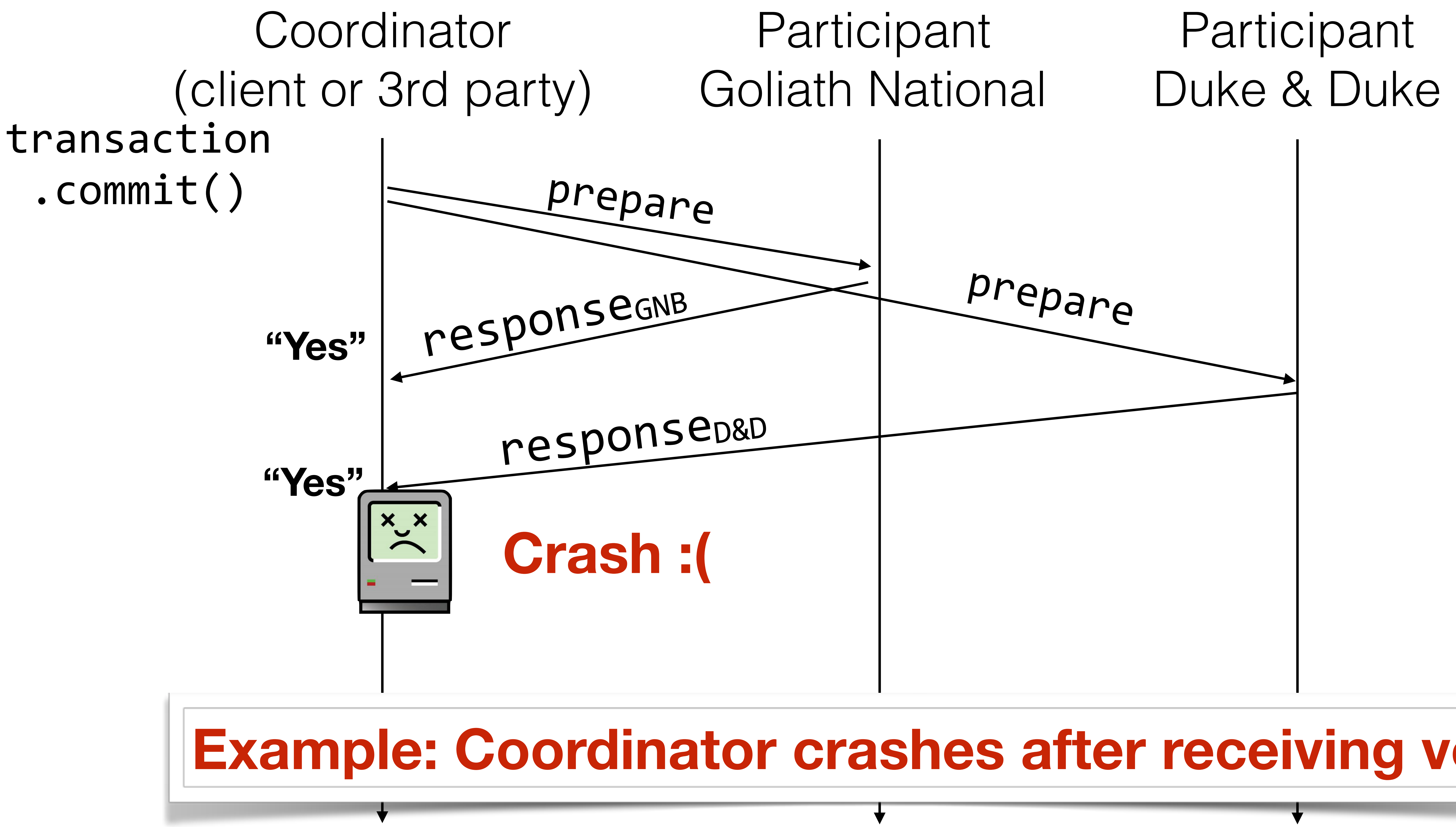
Fault Recovery Example



Example: Participant crashes after voting "yes" to commit

Solution: Participants must keep track of transaction status on persistent storage for recovery on reboot

Fault Recovery Example



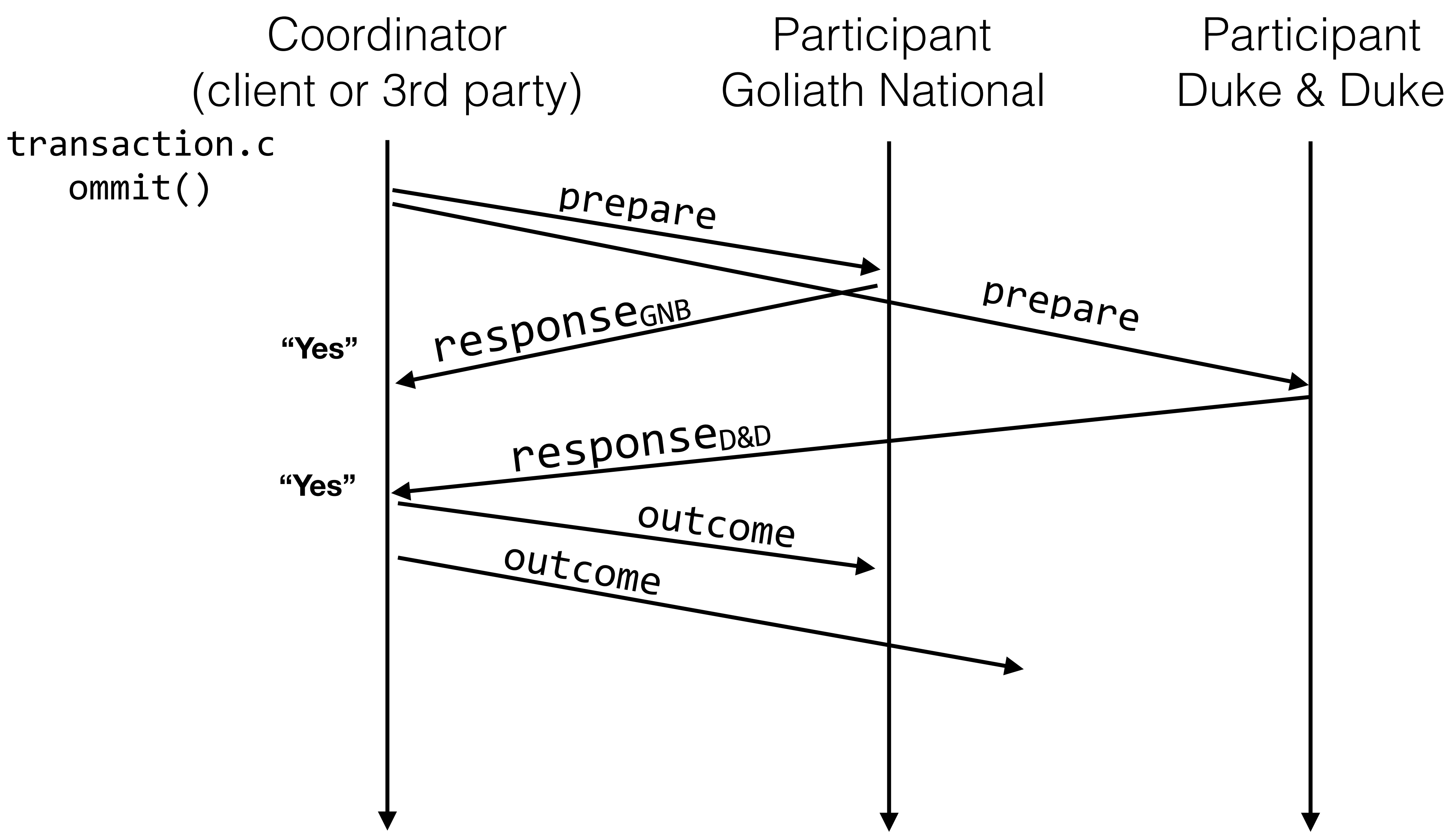
Example: Coordinator crashes after receiving votes

Solution: Coordinator must keep track of transaction status on persistent storage for recovery on reboot

Fault Recovery Example

Example: Participant times out while waiting to hear the outcome

Problem: Can the participant unilaterally determine the outcome?



Participant 1: GNB	Participant 2: D&D	Mutually Agreed Outcome
Votes Yes	Votes No	Abort
Votes No	Votes No	Abort
Votes Yes	Votes Yes	Commit
Votes No	Votes Yes	Abort

Solution: As long as we vote "no" outcome is always abort! If we voted "yes"... no idea!

2PC Timeouts

- We can solve a lot (but not all of the cases) by having the participants talk to each other
- But, if coordinator fails, there are cases where everyone stalls until it recovers
- Can the coordinator fail?… yes
- We'll come back to this "discuss amongst yourselves" kind of transactions today!

Today

- More discussion of fault tolerance, in the context of transactions
- Agreement and transactions in distributed systems - 3PC
- Reminders:
    - HW3 due Thursday!
    - Study opportunity - help improve software engineering, get $40 - https://cs.gmu.edu/~tlatoza/studies/AuthoringDesignRules.pdf

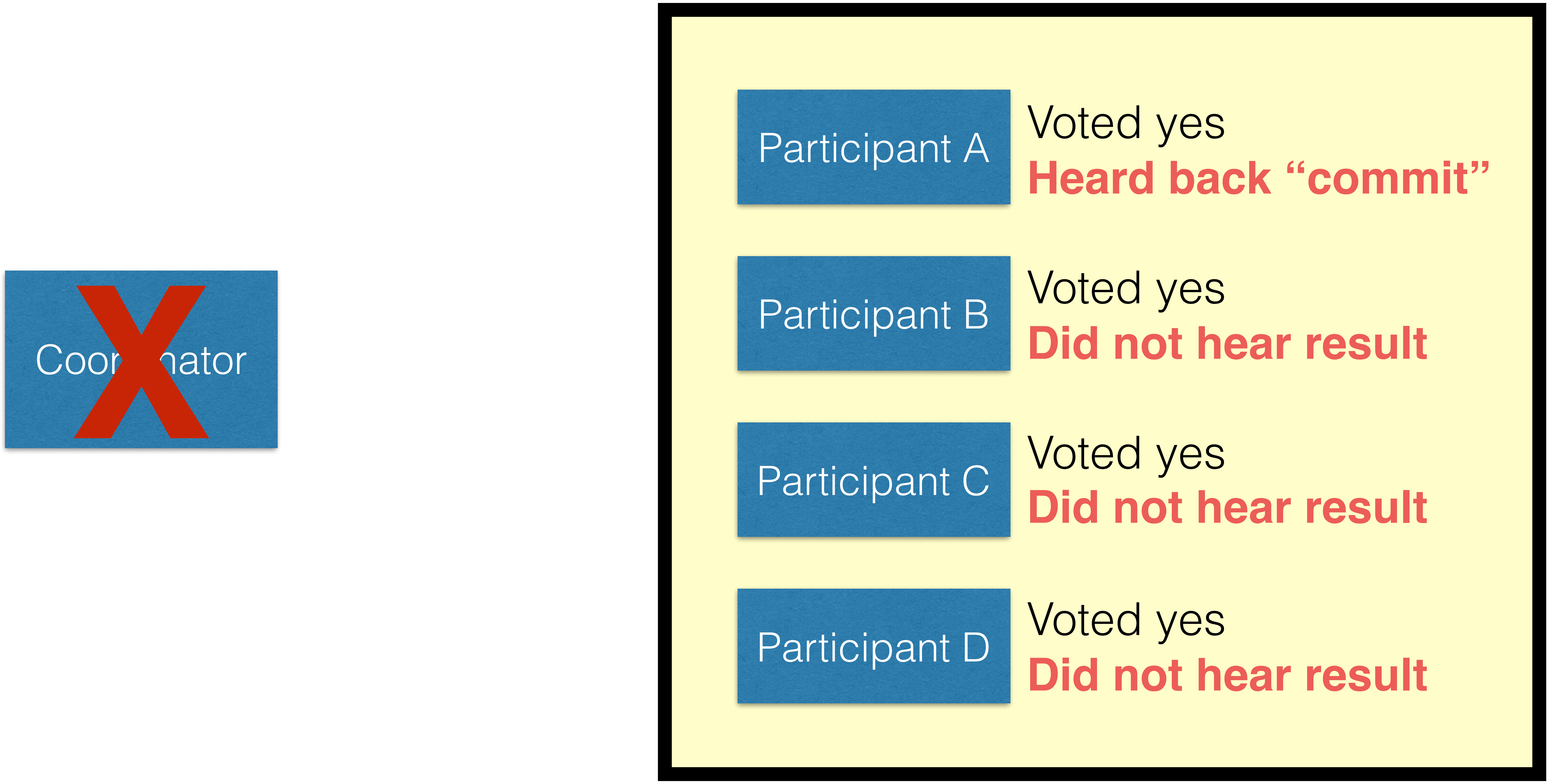
Digging Deeper into 2PC Failures

- Fundamental problem:
  - Once coordinator says commit **we can not go back**
  - That's the property of transactions though!
- In what situations can we reach consensus if the coordinator fails?
- Let's go through some examples again, this time using Socrative to poll your answers

Go to socrative.com and select "Student Login" Room: CS475; ID is your G-Number

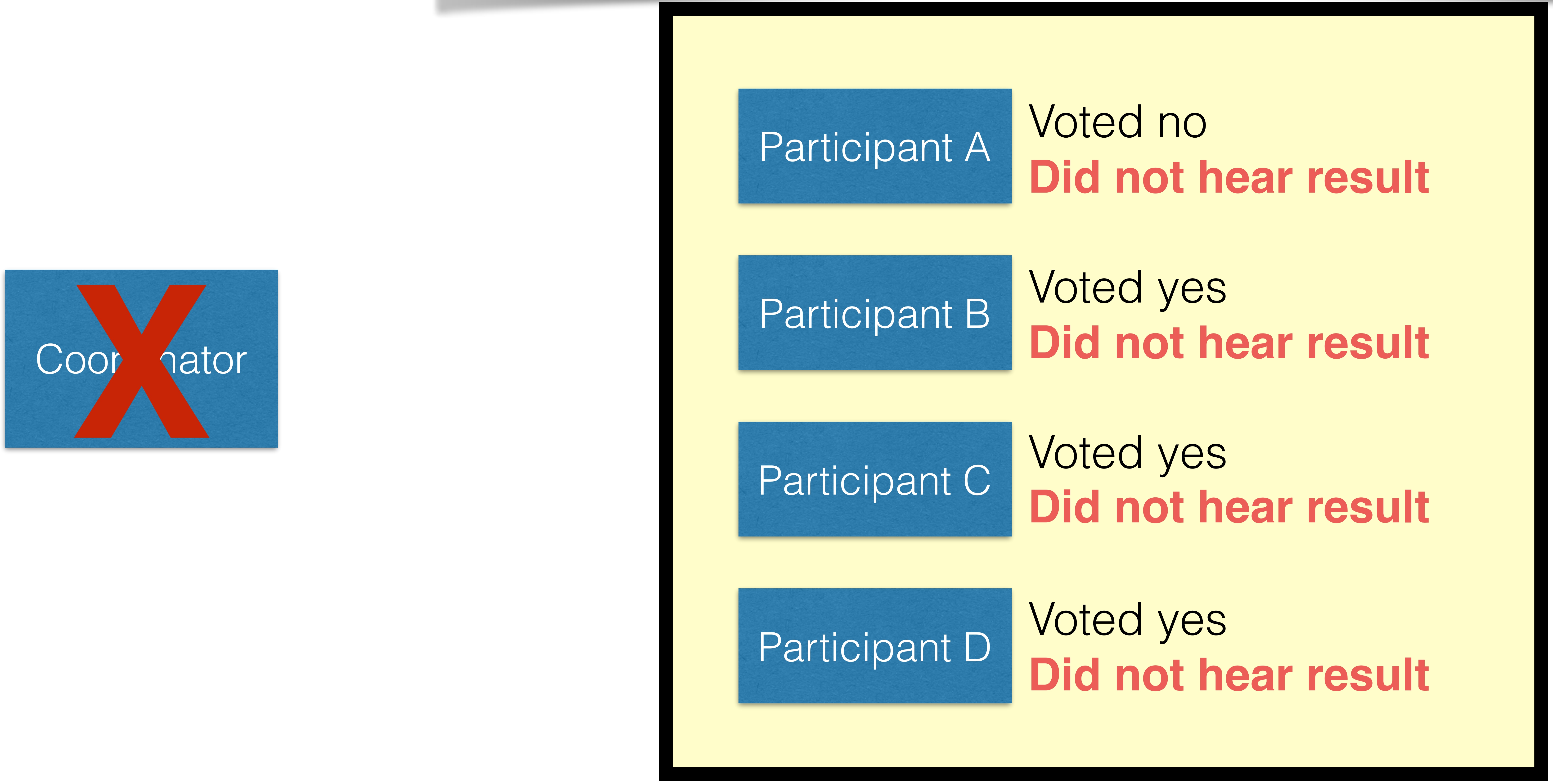
Digging Deeper into 2PC Failures

If they can talk to each other, we know we can commit (good)



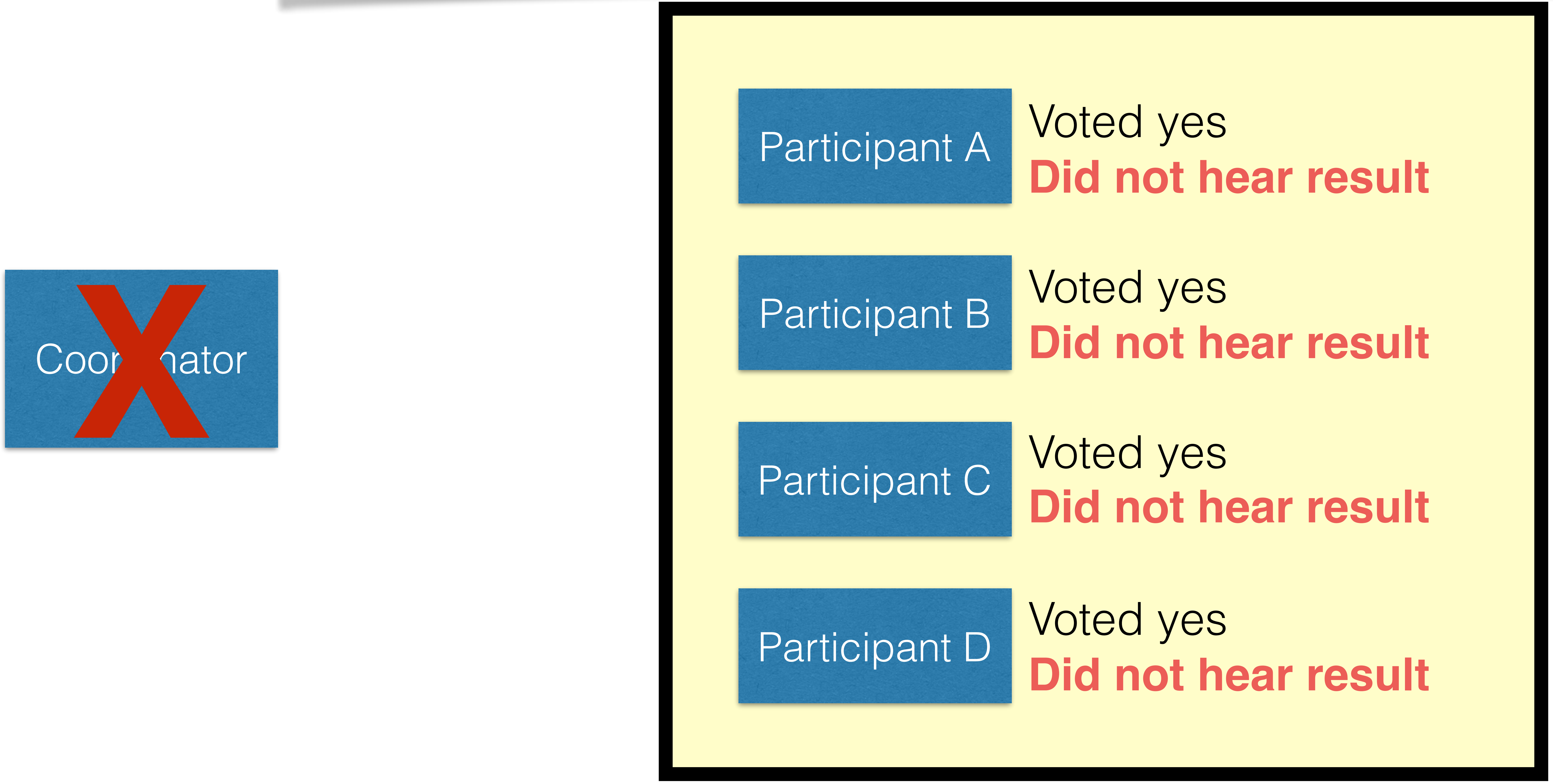
Digging Deeper into 2PC Failures

If they can talk to each other, we know that we can all abort (good)



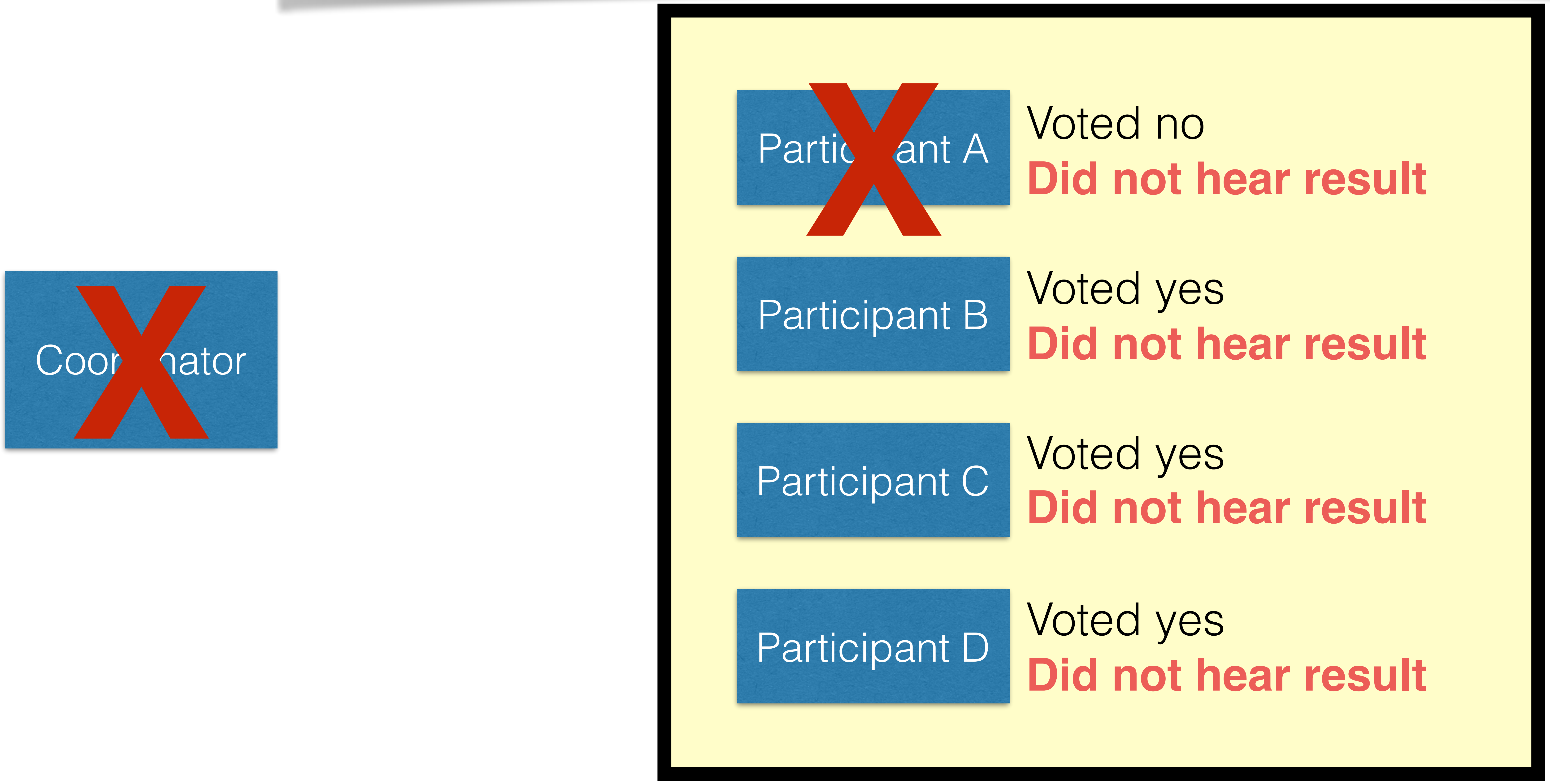
Digging Deeper into 2PC Failures

If they can talk to each other, we do not know if we can commit/abort (who knows what the coordinator will do?)



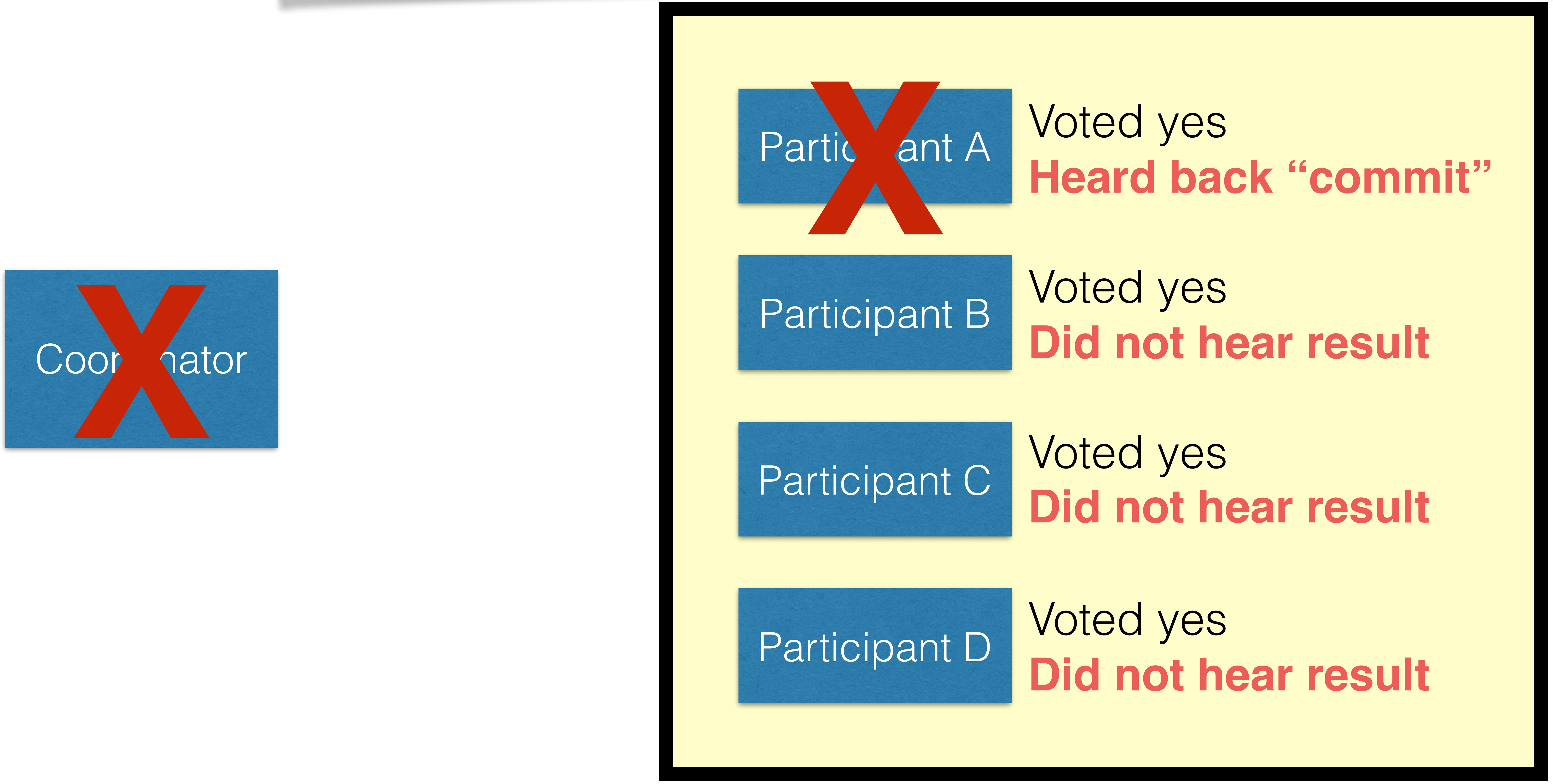
Digging Deeper into 2PC Failures

If they can talk to each other, we do not know if we can commit/abort (who knows that there was a vote no?)



Digging Deeper into 2PC Failures

If they can talk to each other, we do not know if we can commit/abort (do not know what the coordinator heard/said)



3 Phase Commit

- Goal: Eliminate this class of failure from blocking liveness

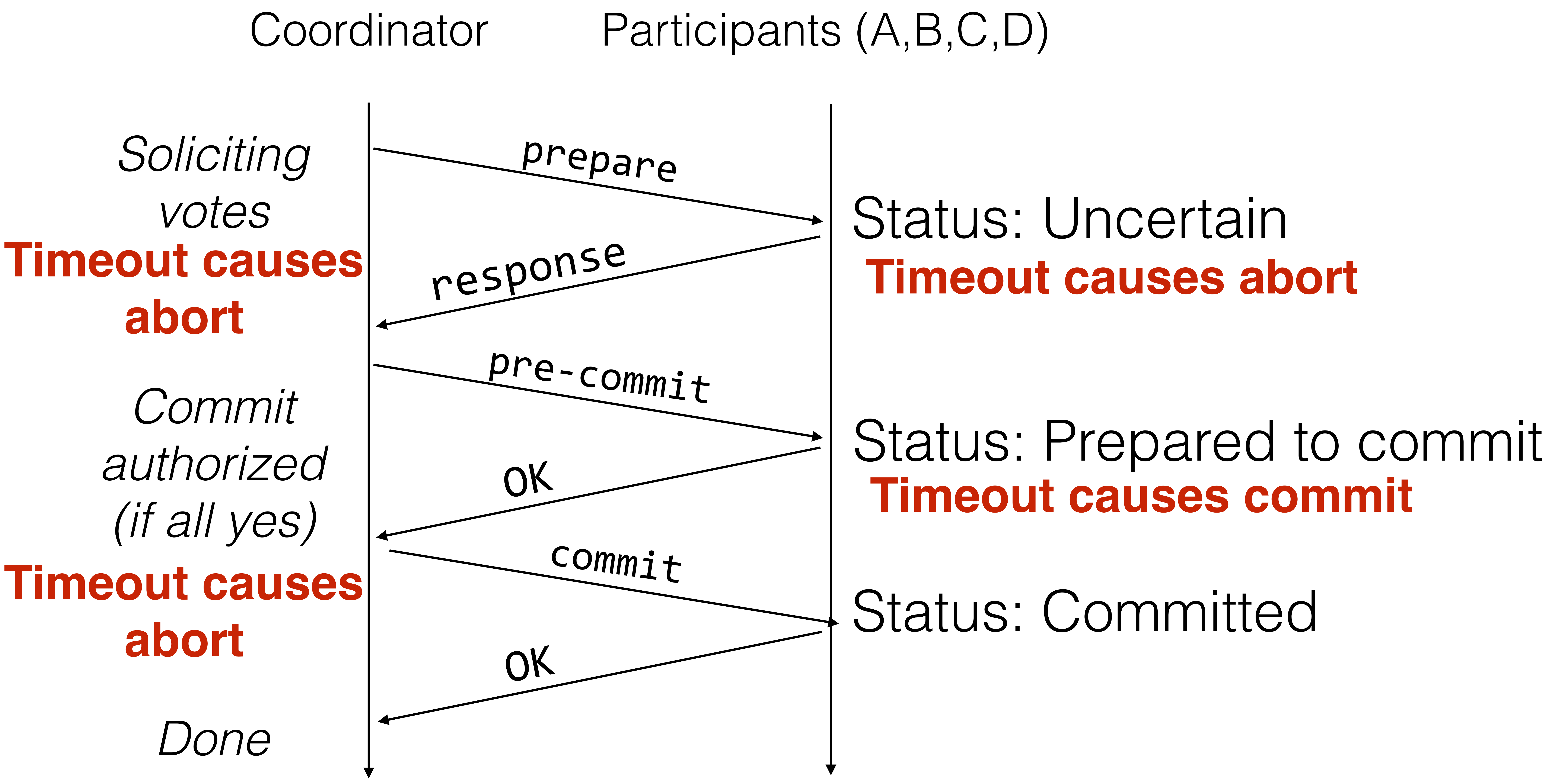
3 Phase Commit

- Goal: Avoid blocking on node failure
- How?
  - Think about how 2PC is better than 1PC
    - 1PC means you can never change your mind or have a failure after committing
    - 2PC **still** means that you can't have a failure after committing (committing is irreversible)

3 Phase Commit

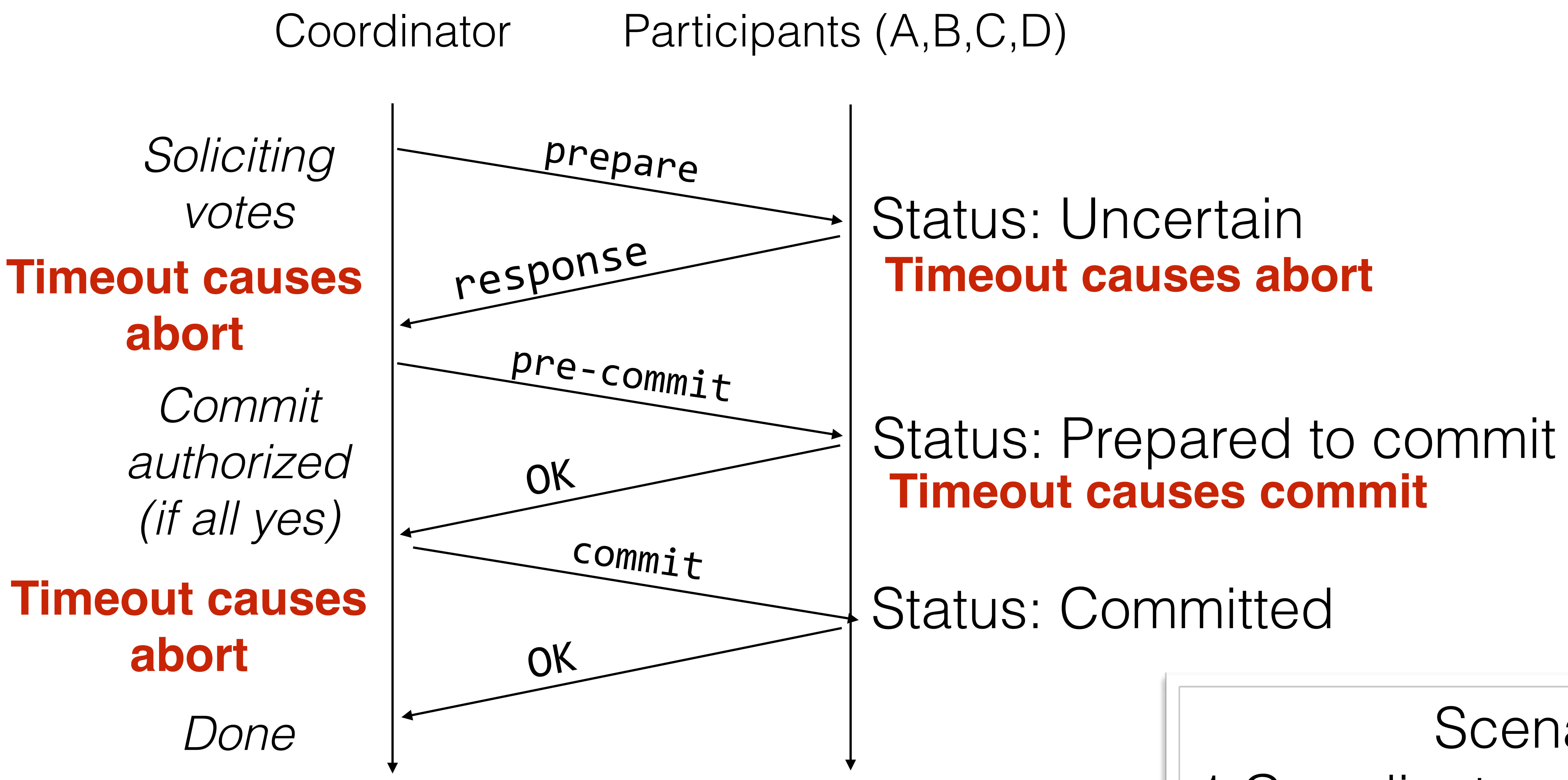
- 3PC idea:
  - Split commit/abort into 2 sub-phases
    - 1: Tell everyone the outcome
    - 2: Agree on outcome
  - Now: EVERY participant knows what the result will be before they irrevocably commit!

3PC Example



Coordinator
Participants (A,B,C,D)
Soliciting votes
Timeout causes abort
prepare
response
Status: Uncertain
Timeout causes abort
pre-commit
Commit authorized (if all yes)
Timeout causes abort
OK
Status: Prepared to commit
Timeout causes commit
commit
Status: Committed
OK
Done

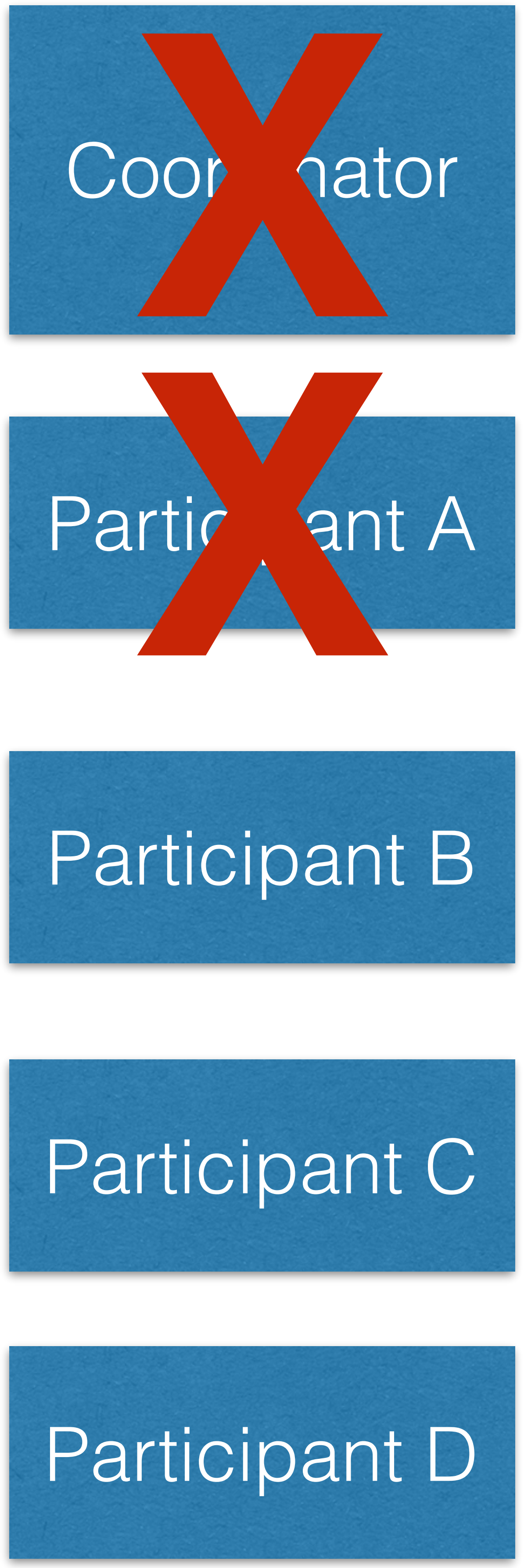
3PC Exercise



Scenario:
1 Coordinator, 4 participants
No failures, all commit

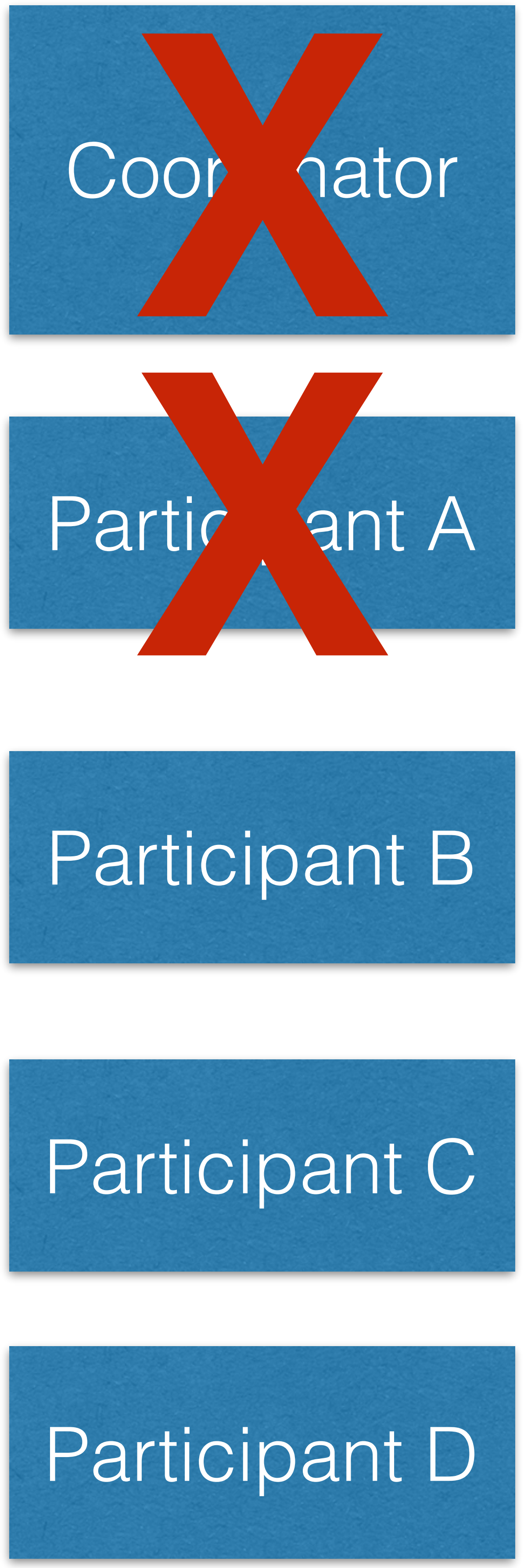
3PC Crash Handling

- Can B/C/D reach a safe decision…
  - If any one of them has received preCommit?
    - YES! Assume A is dead. When A comes back online, it will recover, and talk to B/C/D to catch up.
    - Consider equivalent to in 2PC where B/C/D received the "commit" message and all voted yes

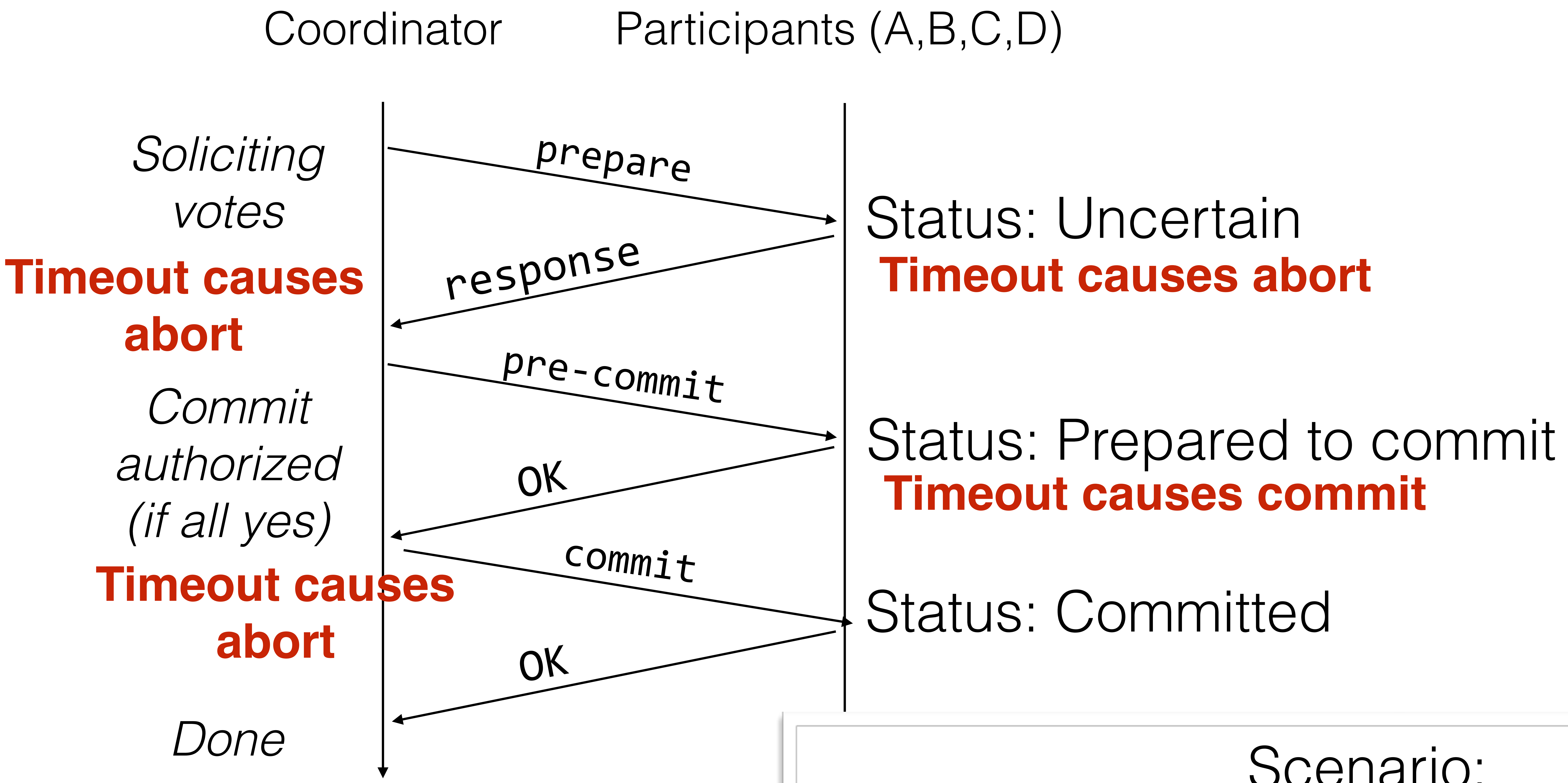


3PC Crash Handling

- Can B/C/D reach a safe decision…
  - If NONE of them has received preCommit?
    - YES! It is safe to abort, because A can not have committed (because it couldn't commit until B/C/D receive and acknowledge the pre-commit)
    - This is the big strength of the extra phase over 2PC
- Summary: Any node can crash at any time, and we can always safely abort or commit.

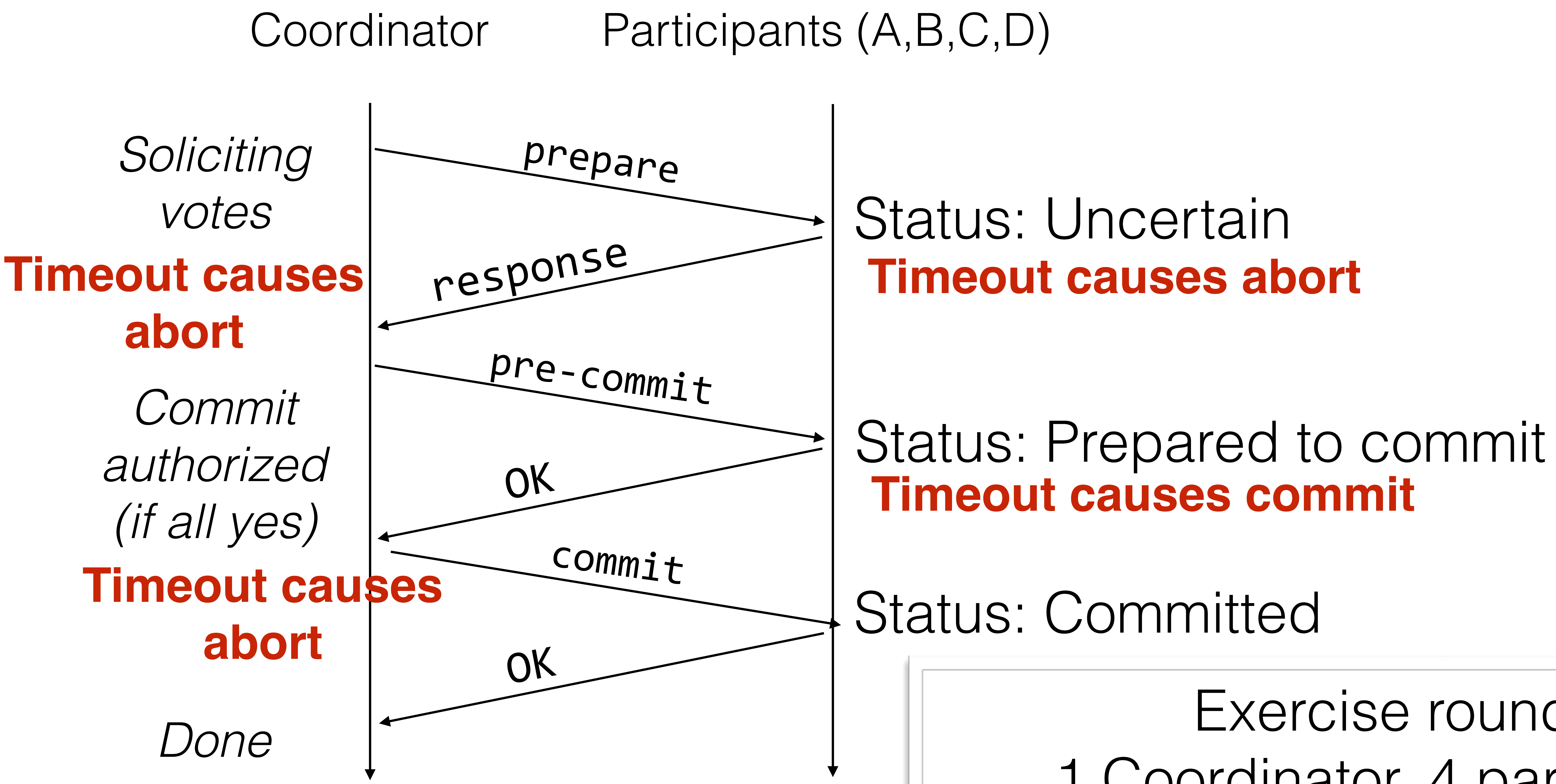


3PC Exercise



Scenario:
1 Coordinator, 4 participants
After pre-commit sent, coordinator and A fail

3PC Exercise



Exercise round 2:
1 Coordinator, 4 participants
Coordinator sends pre-commit message then fails

Properties of Agreement

- **Safety** (correctness)
  - All nodes agree on the same value (which was proposed by some node)
- **Liveness** (fault tolerance, availability)
  - If less than N nodes crash, the rest should still be OK

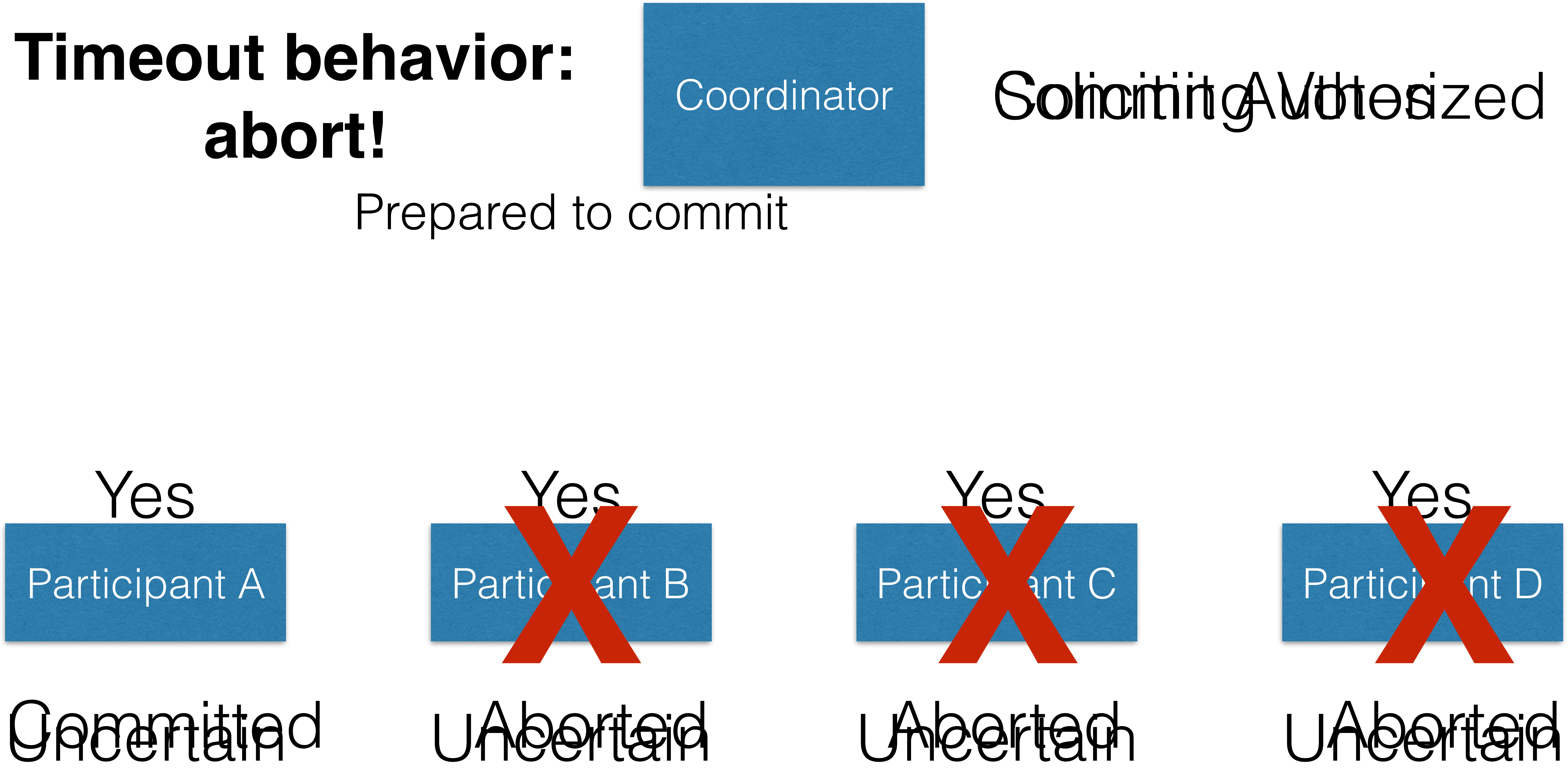
Does 3PC guarantee agreement?

- Reminder, that means:
  - Liveness (availability)
    - **Yes!** Always terminates based on timeouts
  - Safety (correctness)
    - **Yes!***

**Assuming that the only way things fail is by crashing*

Safety in Crashes

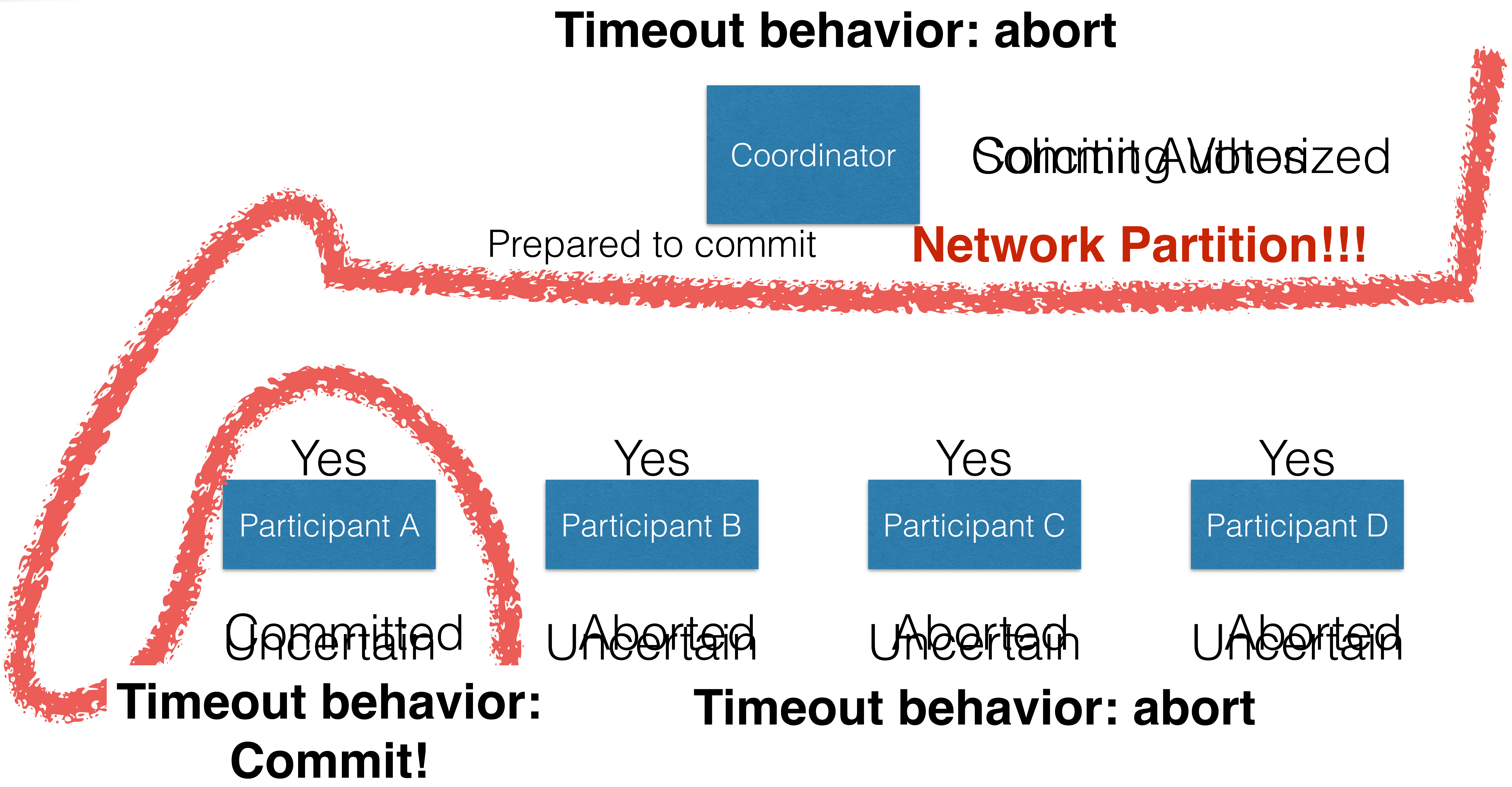
Timeout behavior: abort!



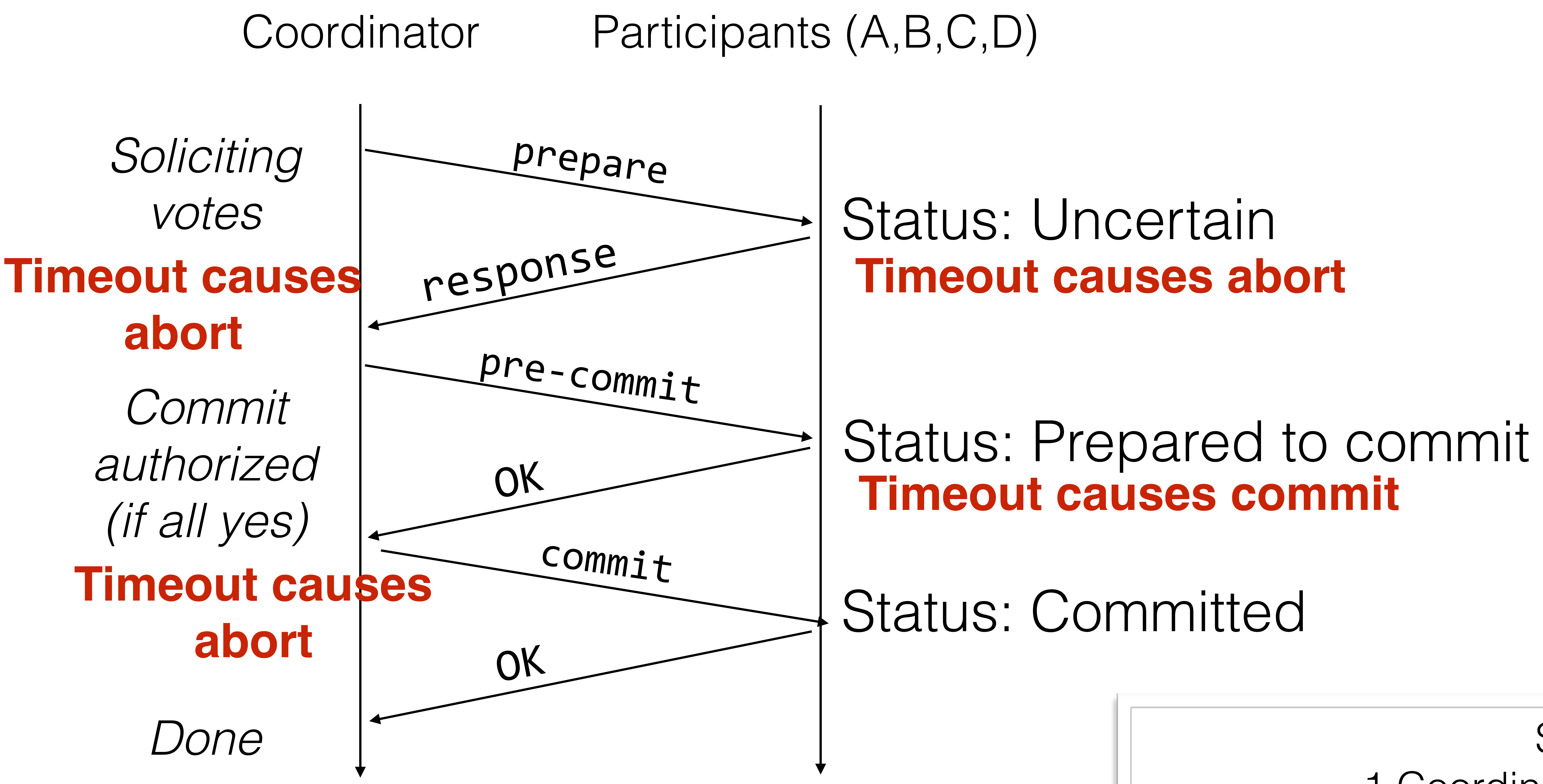
Crashed: do not commit or abort. When recovers, asks coordinator what to do

Partitions

Implication: if networks can delay arbitrarily, 3PC does not guarantee safety!!!!



3PC Exercise



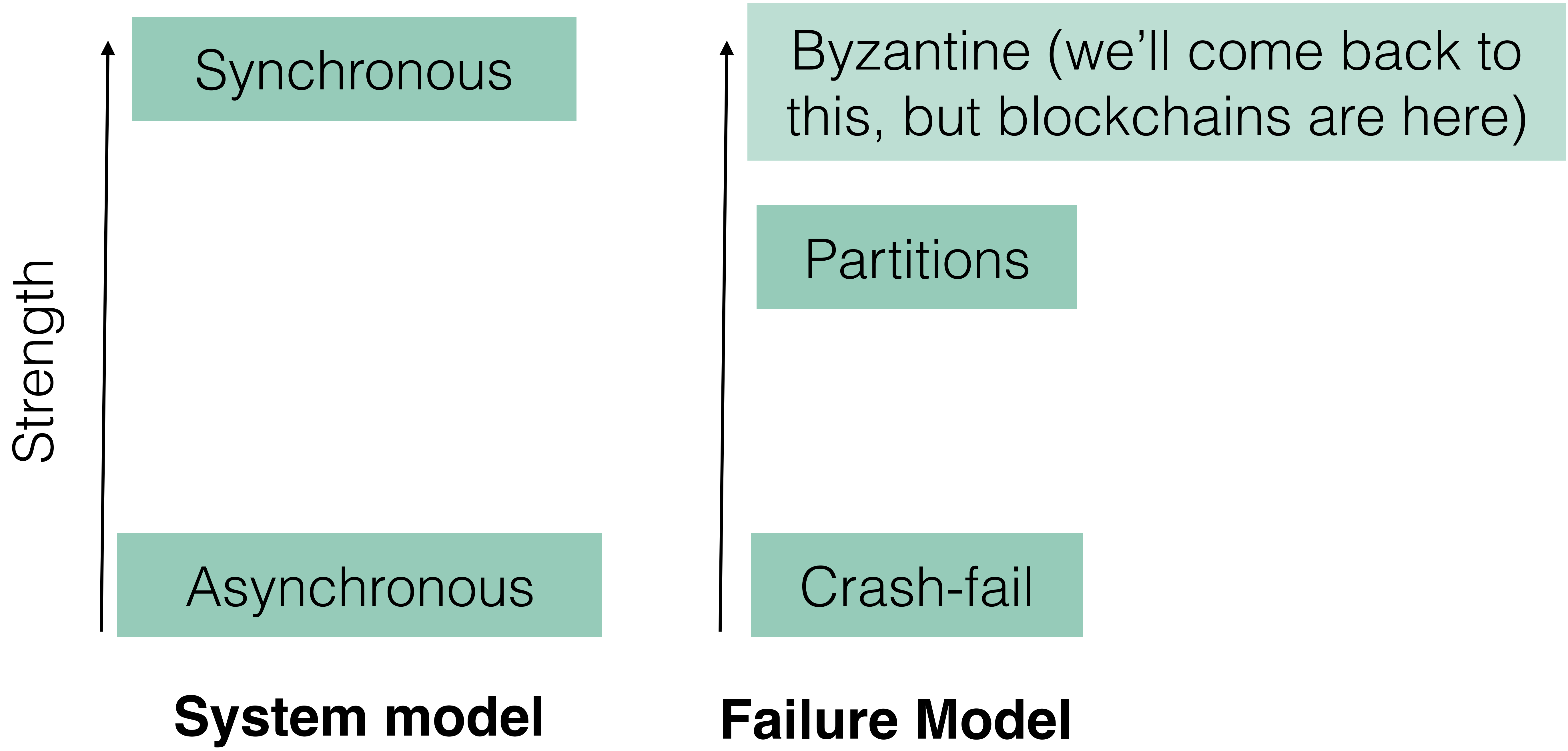
Scenario:
1 Coordinator, 4 participants
Coordinator sends pre-commit message ONLY to A, then Coordinator fails, A partitioned

Modeling our Systems

To help design our algorithms and systems, we tend to leverage abstractions and models to make assumptions

Generally: Stronger assumptions -> worse performance

Weaker assumptions -> more complicated



Synchronous vs Asynchronous Messages

- Synchronous: There is a bound on how long a message takes to arrive
- Asynchronous: There is no bound on how long a message takes to arrive
- Key implication: what does a timeout mean?
  - Synchronous: Something must have crashed
  - Asynchronous: Network might just be slow
- Note: real networks are asynchronous

Failure Models: Crash-Fail vs Partition Tolerant

- Crash-fail: Our system will be correct if the only failures we can ever see are a node crashing
- Partition tolerant: Our system will be correct for crashing failures **and** for arbitrary network delays
- NB: If the network is synchronous, we are partition-tolerant by default (no partitions possible)

2PC vs 3PC

- 2PC
  - Safety (always, for crash and partition failures)
  - Liveness (if 1 node fails, we may block)
- 3PC
  - Safety (assuming the only failure mode is crash, never partition)
  - Liveness (can always proceed if 1 node fails)
- Can we have some hybrid/best of both worlds?

Can we fix it?

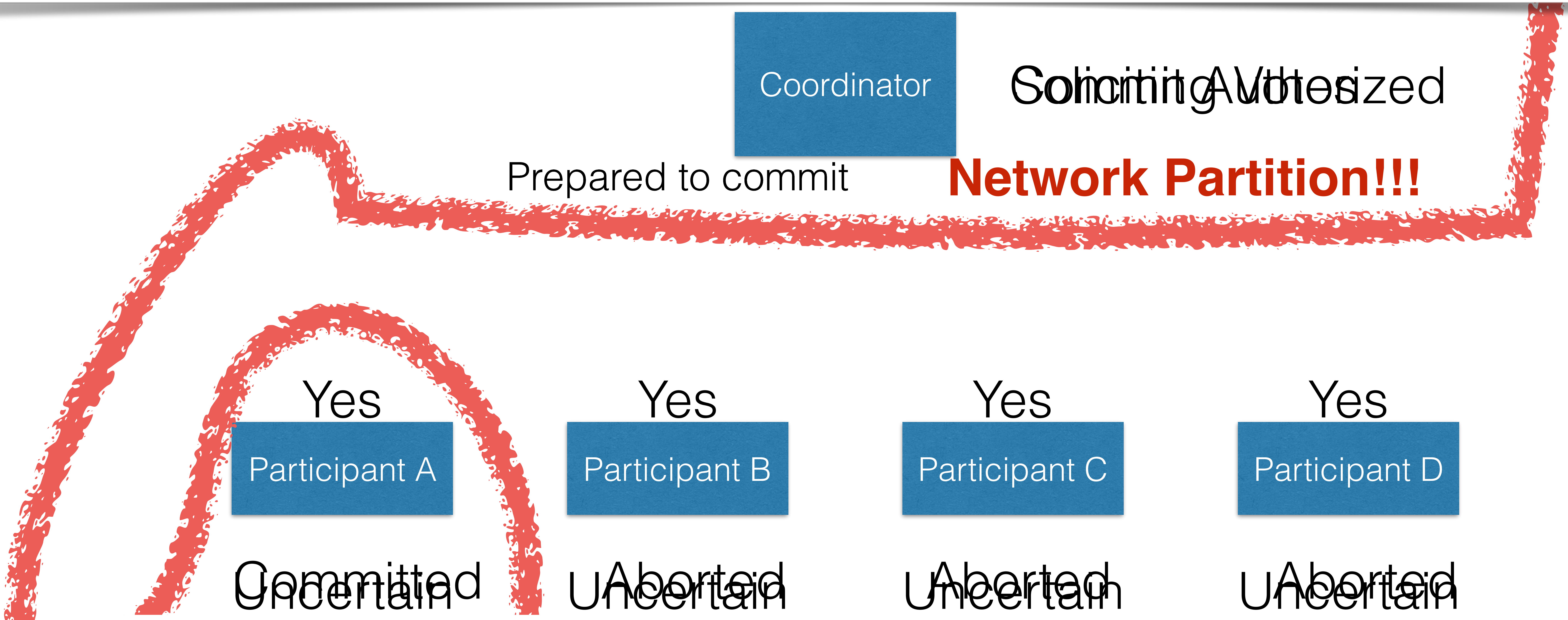
- Short answer: No.
- Fischer, Lynch & Paterson (FLP) Impossibility Result:
  - Assume that nodes can only fail by crashing, network is reliable but can be delayed arbitrarily
  - Then, there can not be a deterministic algorithm for the consensus problem subject to these failures

FLP - Intuition

- Why can’t we make a protocol for consensus/agreement that can tolerate both partitions and node failures?
- To tolerate a partition, you need to assume that **eventually** the partition will heal, and the network will deliver the delayed packages
- But the messages might be delayed **forever**
- Hence, your protocol would not come to a result, until **forever** (it would not have the **liveness** property)

Partitions

Insight: There is a “majority” partition here (B,C,D)
The “minority” know that they are not in the majority (A can only talk to Coordinator, knows B, C, D might exist)



Timeout behavior: Commit!

Timeout behavior: abort

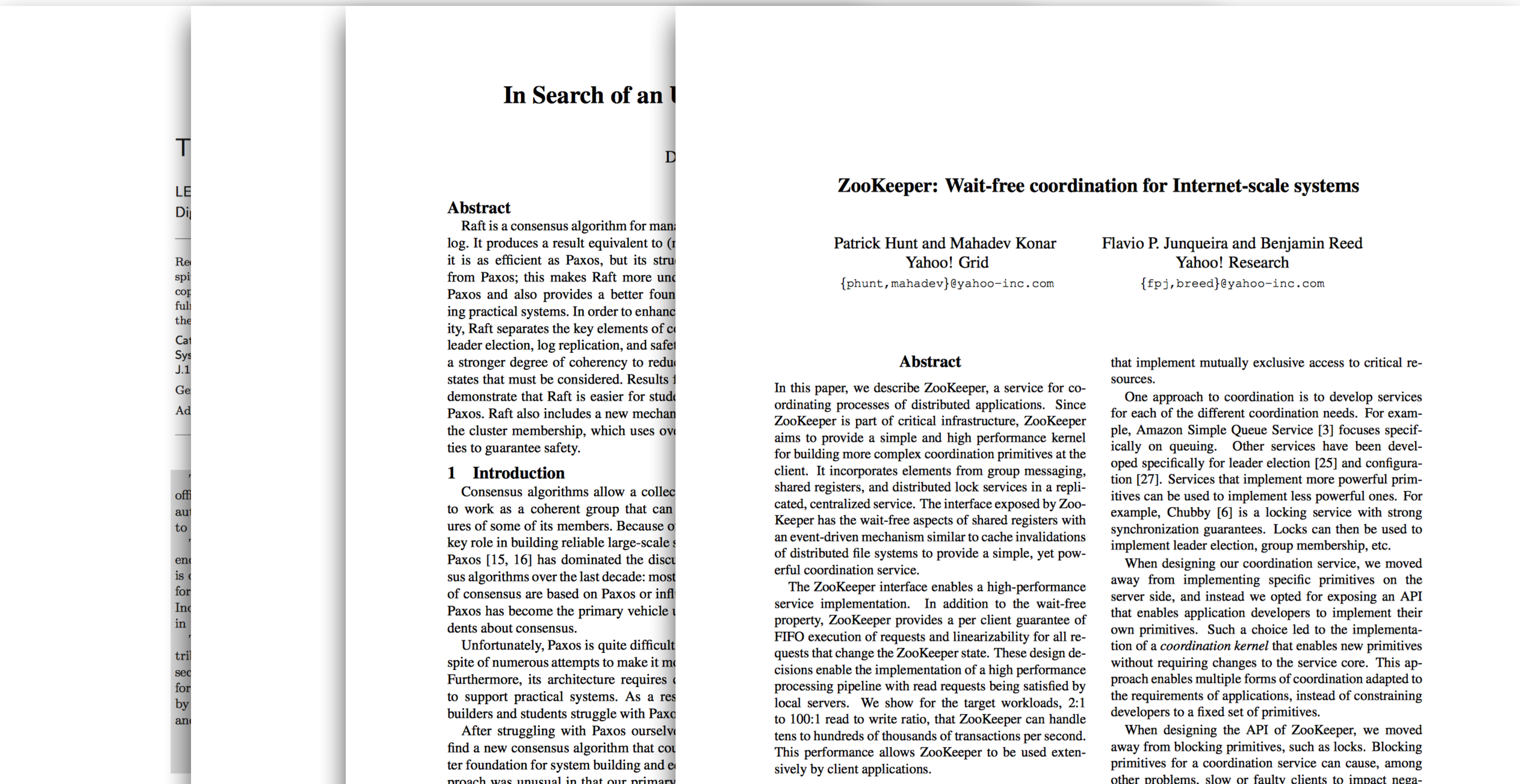
Partition Tolerance

- Key idea: if you always have an odd number of nodes…
- There will always be a **minority** partition and a **majority** partition
- Give up processing in the minority until partition heals and network resumes
- Majority can continue processing

Partition Tolerant Consensus Algorithms

- Decisions made by **majority**
- Typically a fixed coordinator (**leader**) during a time period (**epoch**)
- How does the leader change?
  - Assume it starts out as an arbitrary node
  - The leader sends a heartbeat
  - If you haven't heard from the leader, then you **challenge** it by advancing to the next epoch and try to elect a new one
  - If you don't get a **majority** of votes, you don't get to be leader
  - …hence no leader in a minority partition

Partition Tolerant Consensus Algorithms



ZooKeeper: Wait-free coordination for Internet-scale systems

Patrick Hunt and Mahadev Konar
Yahoo! Grid
{phunt,mahadev}@yahoo-inc.com

Flavio P. Junqueira and Benjamin Reed
Yahoo! Research
{fpj,breed}@yahoo-inc.com

Abstract

In this paper, we describe ZooKeeper, a service for coordinating processes of distributed applications. Since ZooKeeper is part of critical infrastructure, ZooKeeper aims to provide a simple and high performance kernel for building more complex coordination primitives at the client. It incorporates elements from group messaging, shared registers, and distributed lock services in a replicated, centralized service. The interface exposed by ZooKeeper has the wait-free aspects of shared registers with an event-driven mechanism similar to cache invalidations of distributed file systems to provide a simple, yet powerful coordination service.

The ZooKeeper interface enables a high-performance service implementation. In addition to the wait-free property, ZooKeeper provides a per client guarantee of FIFO execution of requests and linearizability for all requests that change the ZooKeeper state. These design decisions enable the implementation of a high performance processing pipeline with read requests being satisfied by local servers. We show for the target workloads, 2:1 to 100:1 read to write ratio, that ZooKeeper can handle tens to hundreds of thousands of transactions per second. This performance allows ZooKeeper to be used extensively by client applications.

that implement mutually exclusive access to critical resources.

One approach to coordination is to develop services for each of the different coordination needs. For example, Amazon Simple Queue Service [3] focuses specifically on queuing. Other services have been developed specifically for leader election [25] and configuration [27]. Services that implement more powerful primitives can be used to implement less powerful ones. For example, Chubby [6] is a locking service with strong synchronization guarantees. Locks can then be used to implement leader election, group membership, etc.

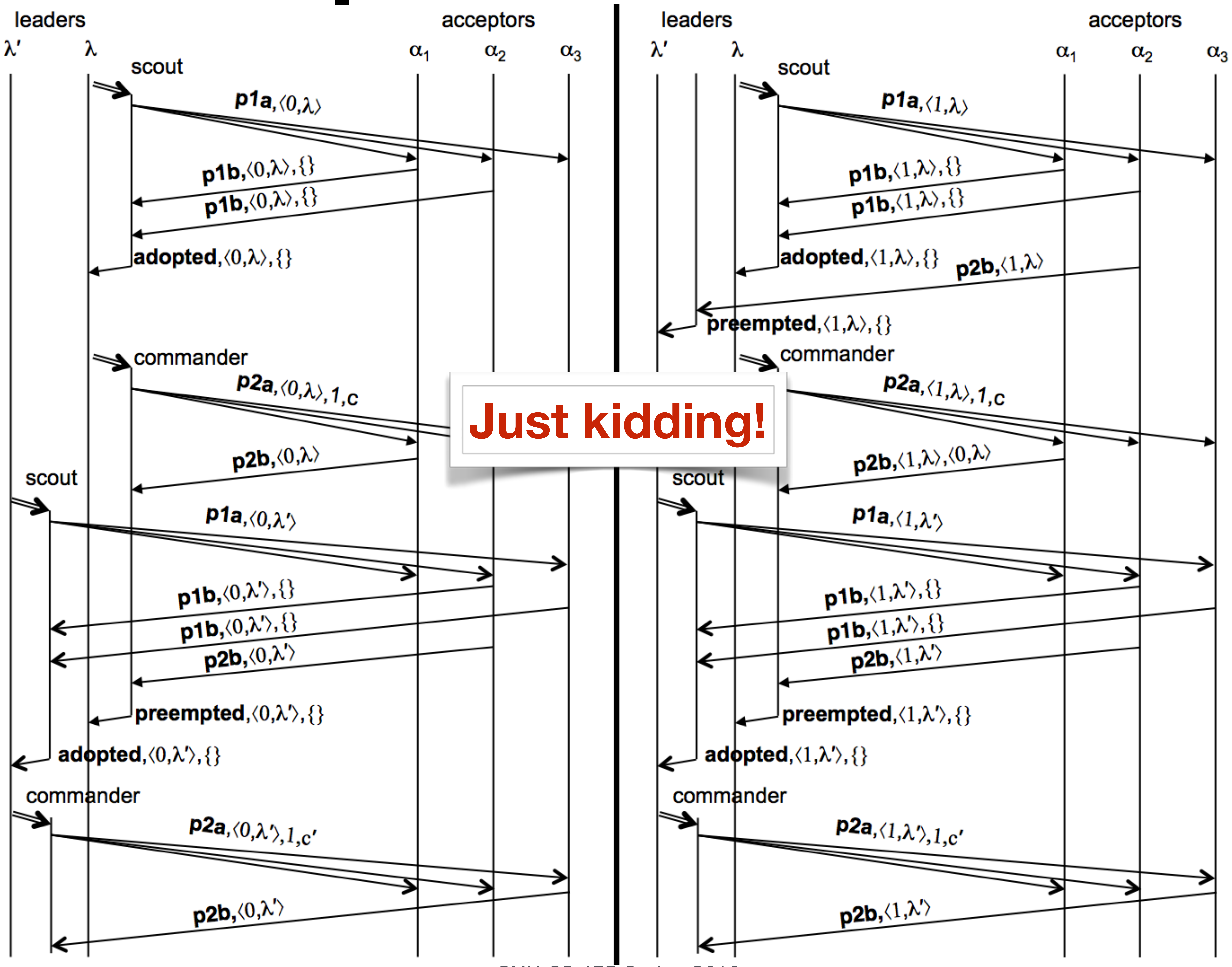
When designing our coordination service, we moved away from implementing specific primitives on the server side, and instead we opted for exposing an API that enables application developers to implement their own primitives. Such a choice led to the implementation of a *coordination kernel* that enables new primitives without requiring changes to the service core. This approach enables multiple forms of coordination adapted to the requirements of applications, instead of constraining developers to a fixed set of primitives.

When designing the API of ZooKeeper, we moved away from blocking primitives, such as locks. Blocking primitives for a coordination service can cause, among other problems, slow or faulty clients to impact nega-

Paxos: High Level

- One (or more) nodes decide to be leader (proposer)
- Leader proposes a value, solicits acceptance from the rest of the nodes
- Leader announces chosen value, or tries again if it failed to get all nodes to agree on that value
- Lots of tricky corners (failure handling)
- In sum: requires only a majority of the (non-leader) nodes to accept a proposal for it to succeed

Paxos: Implementation Details



ZooKeeper

- Distributed coordination service from Yahoo! originally, now maintained as Apache project, used widely (key component of Hadoop etc)
- Highly available, fault tolerant, performant
- Designed so that YOU don't have to implement Paxos for:
  - Distributed transactions/agreement/consensus
- We'll come back to ZooKeeper in a few weeks