

# Agreement and Consensus

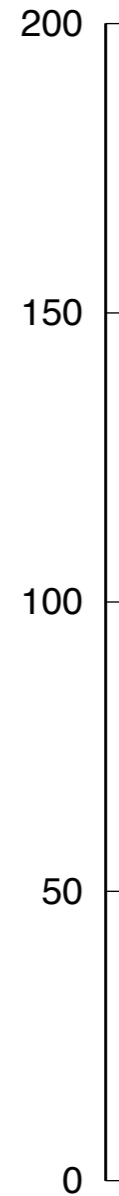
SWE 622, Spring 2017  
Distributed Software Engineering

# Today

- General agreement problems
- Fault tolerance limitations of 2PC
- 3PC
- Paxos + ZooKeeper

# Midterm Recap

GMU SWE 622 Midterm Scores (max possible = 200)



# Agreement

- In distributed systems, we have multiple nodes that need to all agree that some object has some state
- Examples:
  - Who owns a lock
  - Whether or not to commit a transaction
  - The value of a clock

# 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

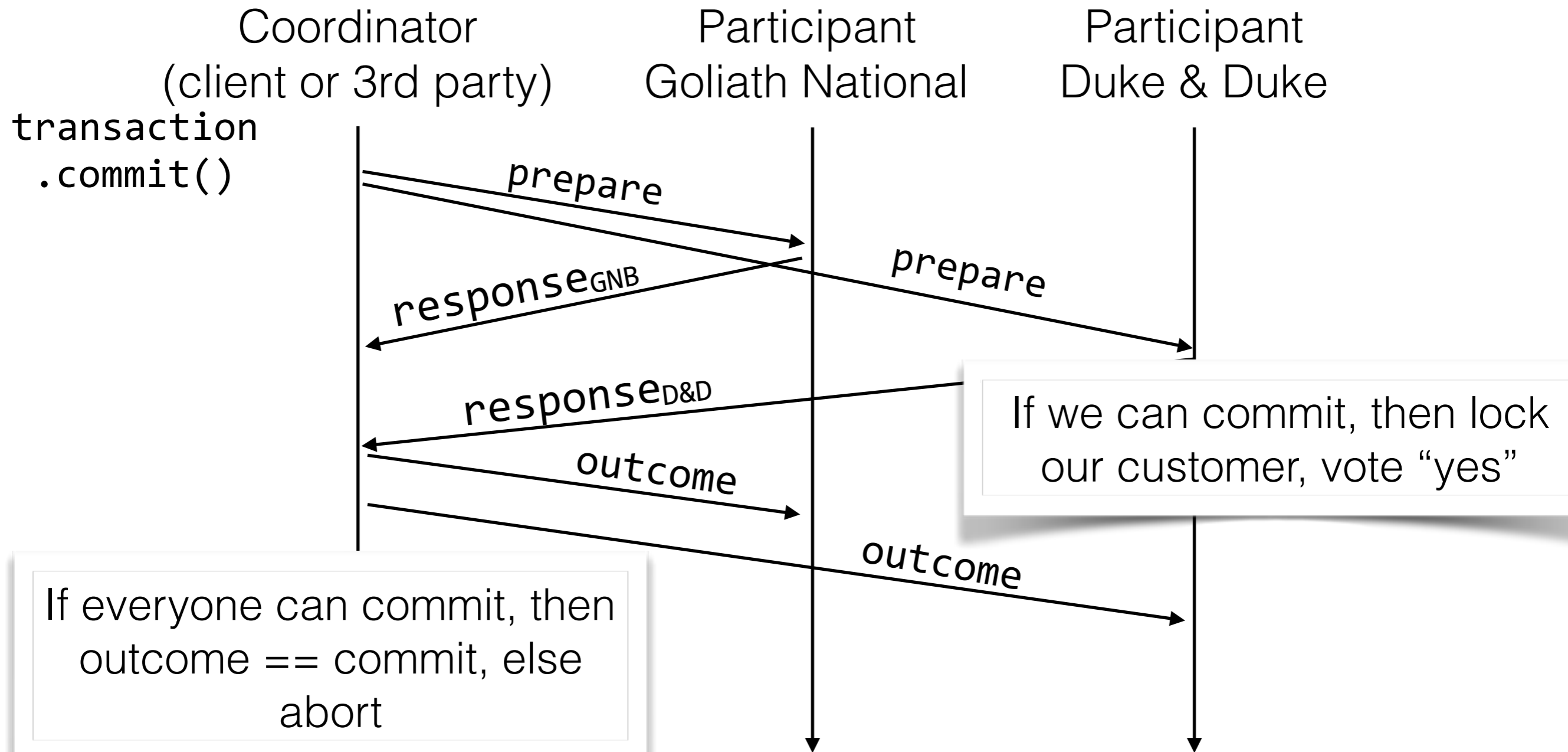
# Properties of Agreement

- 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

# 2PC Example



# Timeouts in 2PC

- Example:
  - Coordinator times out waiting for Goliath National Bank's response
  - Bank times out waiting for coordinator's outcome message
- Causes?
  - Network
  - Overloaded hosts
  - Both are very realistic....

# Coordinator Timeouts

- If coordinator times out waiting to hear from a bank
  - Coordinator hasn't sent any commit messages yet
  - Can safely abort - send abort message
  - Preserves correctness, sacrifices performance (maybe didn't need to abort!)
  - If either bank decided to commit, it's fine - they will eventually abort

# Handling Bank Timeouts

- What if the bank doesn't hear back from coordinator?
- If bank voted "no", it's OK to abort
- If bank voted "yes"
  - It can't decide to abort (maybe both banks voted "yes" and coordinator heard this)
  - It can't decide to commit (maybe other bank voted yes)
- Does bank just wait for ever?

# Handling Bank Timeouts

- Can resolve SOME timeout problems with guaranteed correctness in event bank voted “yes” to commit
- Bank asks other bank for status (if it heard from coordinator)
- If other bank heard “commit” or “abort” then do that
- If other bank didn't hear
  - but other voted “no”: both banks abort
  - but other voted “yes”: no decision possible!

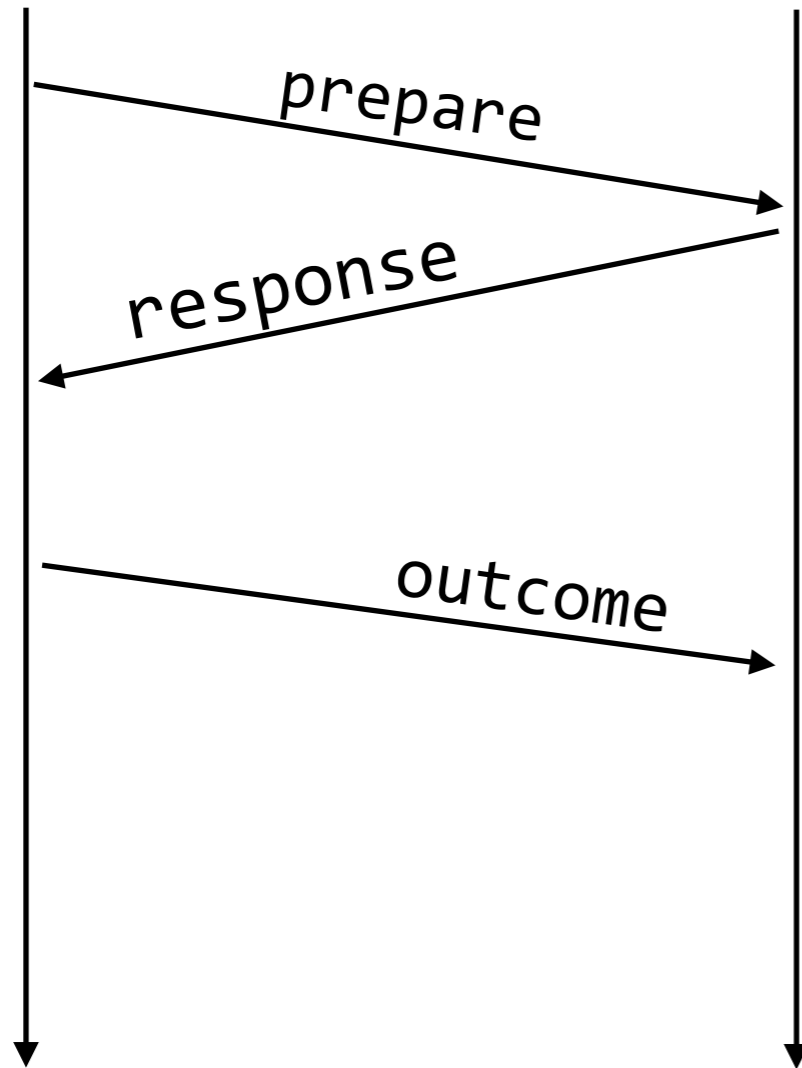
# 2PC Timeouts

- 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
- Hence, 2PC does not guarantee **liveness**: a single node failing can cause the entire set to fail

# 2PC Exercise

Coordinator  
(client or 3rd party)

Participant

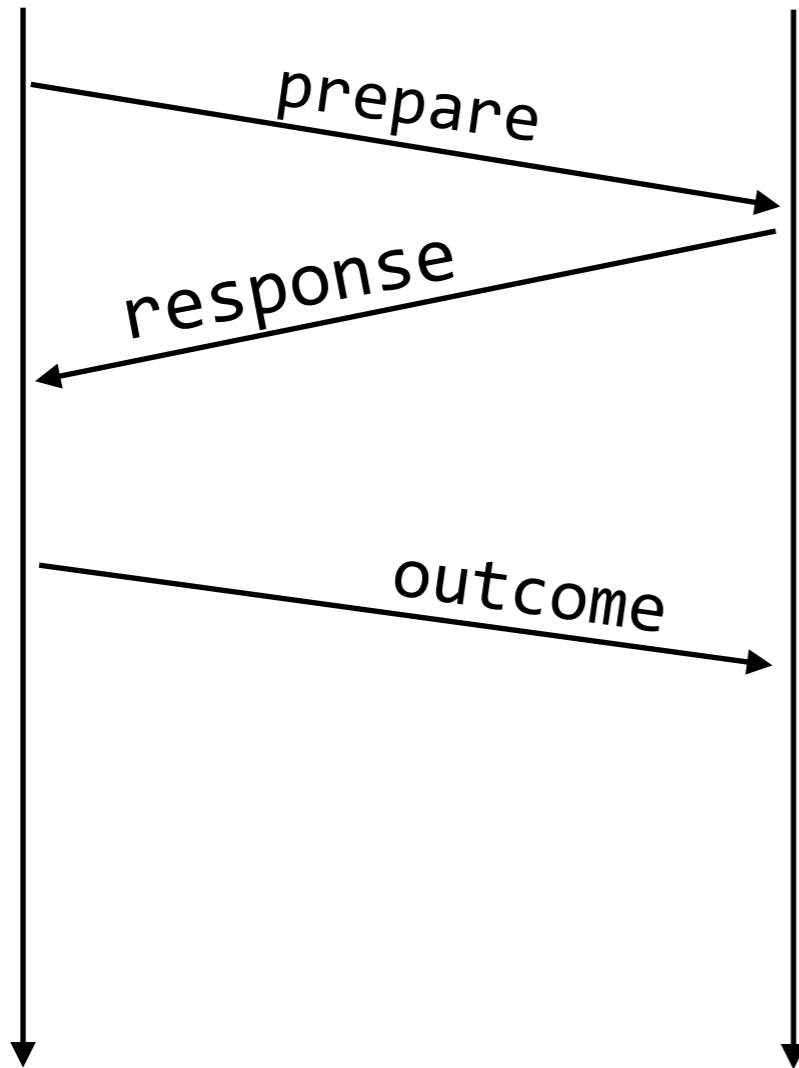


Exercise round 1:  
1 Coordinator, 4 participants  
No failures, all commit

# 2PC Exercise

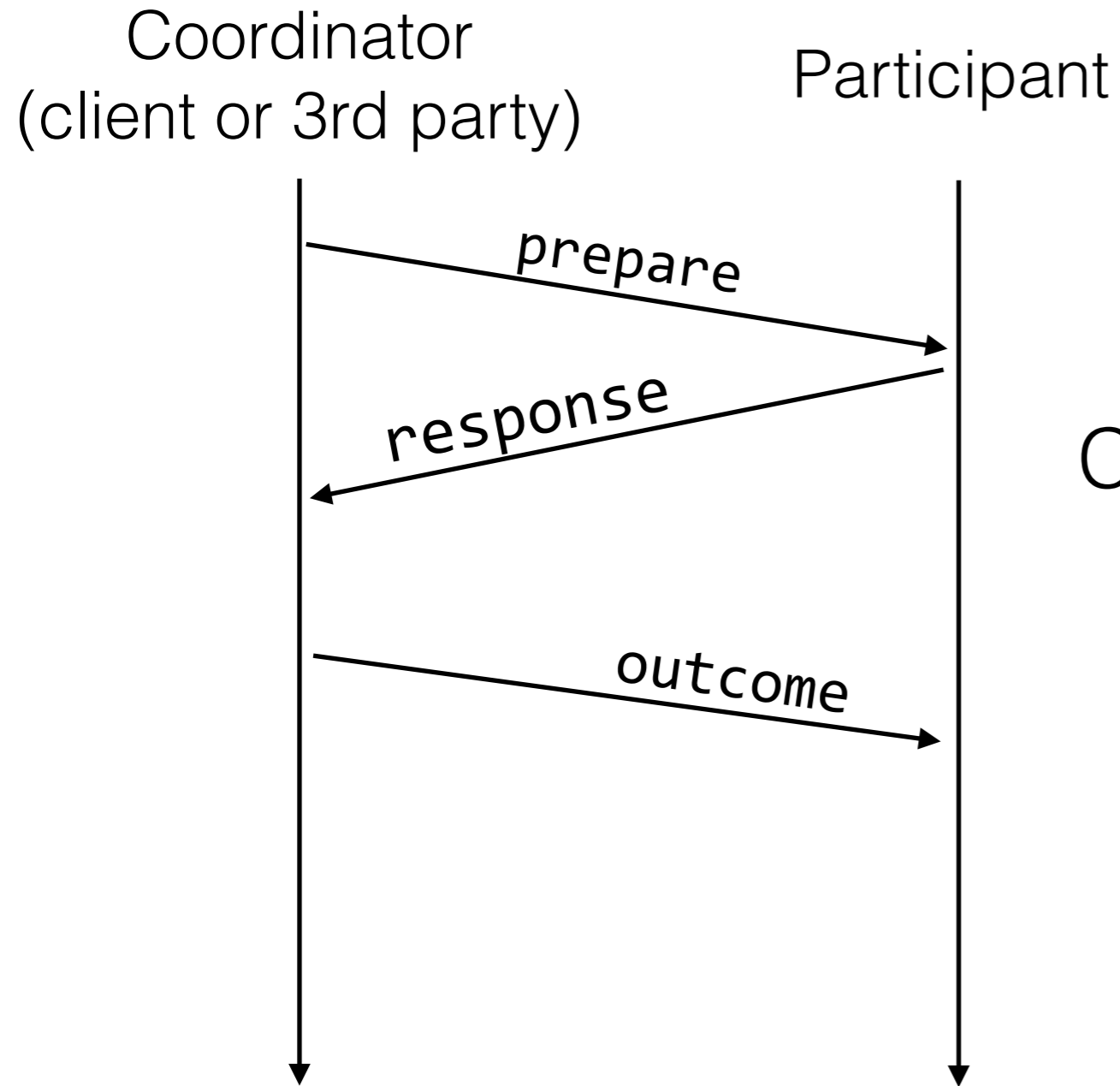
Coordinator  
(client or 3rd party)

Participant



Exercise round 2:  
1 Coordinator, 4 participants  
Coordinator fails before providing  
outcome

# 2PC Exercise



Exercise round 3:  
1 Coordinator, 4 participants  
Coordinator provides outcome to  
1 participant, then coordinator  
and that participant fail

# 3 Phase Commit

- Goal: Eliminate this specific failure from blocking liveness

~~Coordinator~~

~~Participant A~~

Voted yes  
Heard back “commit”

Participant B

Voted yes  
**Did not hear result**

Participant C

Voted yes  
**Did not hear result**

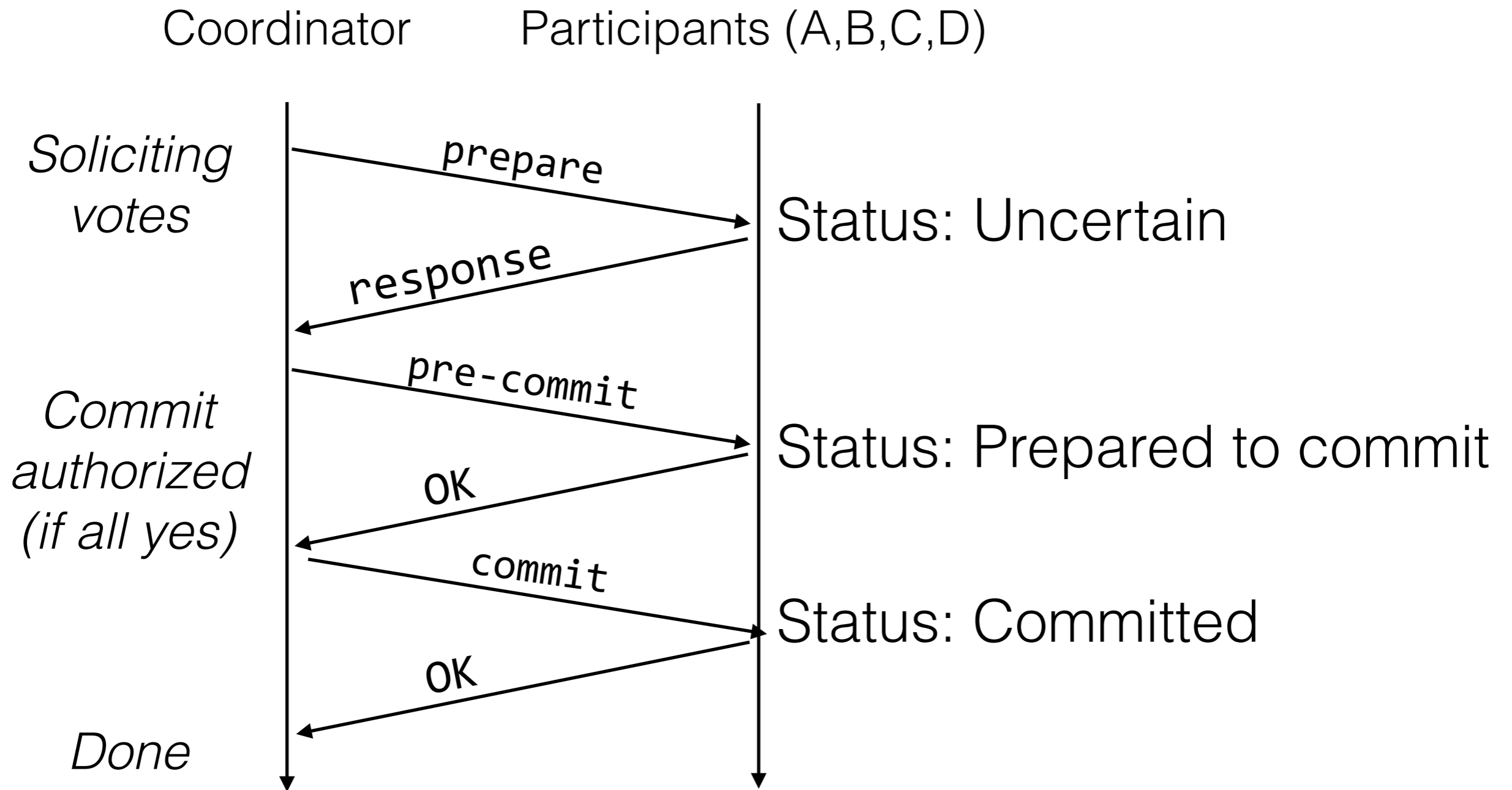
Participant D

Voted yes  
**Did not hear result**

# 3 Phase Commit

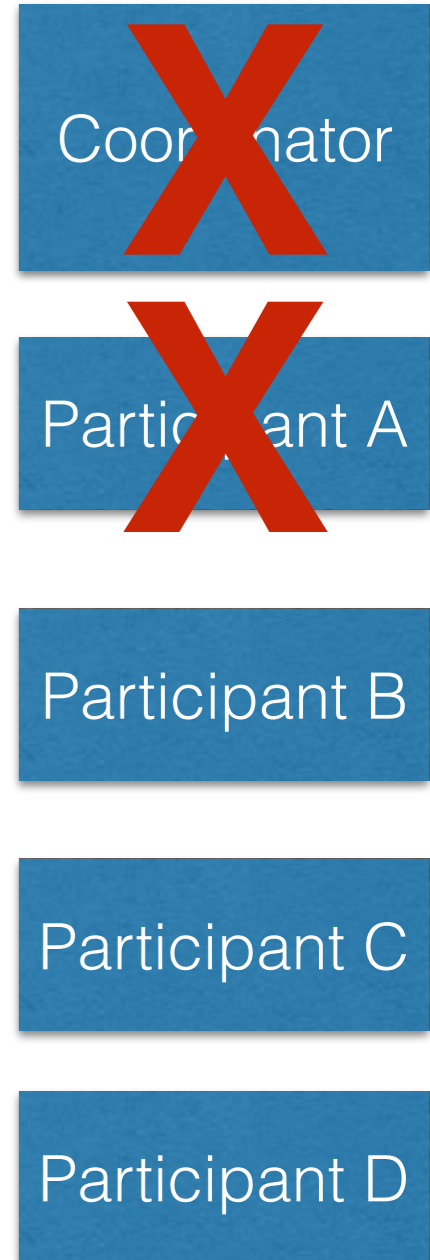
- Goal: Avoid blocking on node failure
- How?
  - Think about how 2PC is better than 1PC
    - 1PC means you can never change your mind or have a failure after committing
    - 2PC **still** means that you can't have a failure after committing (committing is irreversible)
- 3PC idea:
  - Split commit/abort into 2 sub-phases
    - 1: Tell everyone the outcome
    - 2: Agree on outcome

# 3PC Example



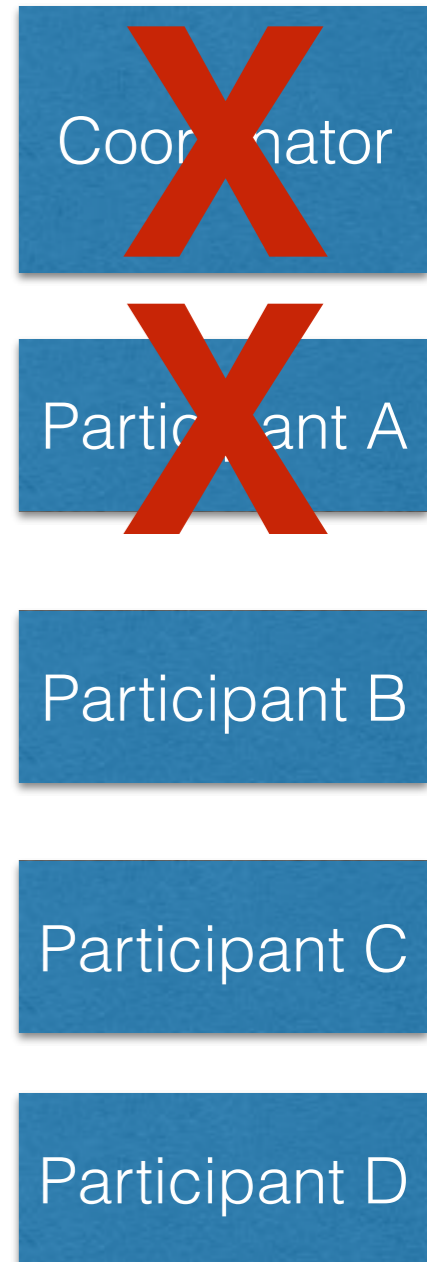
# 3PC Crash Handling

- 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



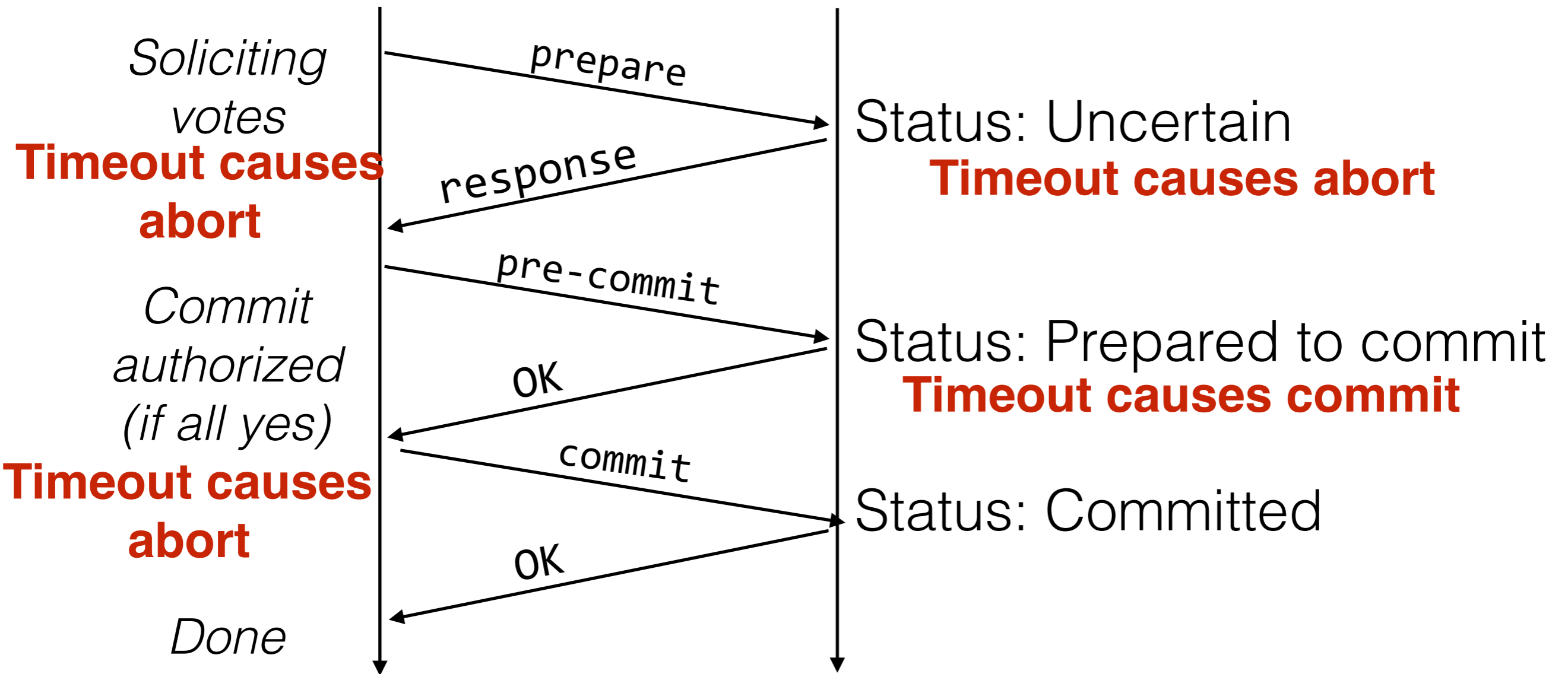
# 3PC Crash Handling

- 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.



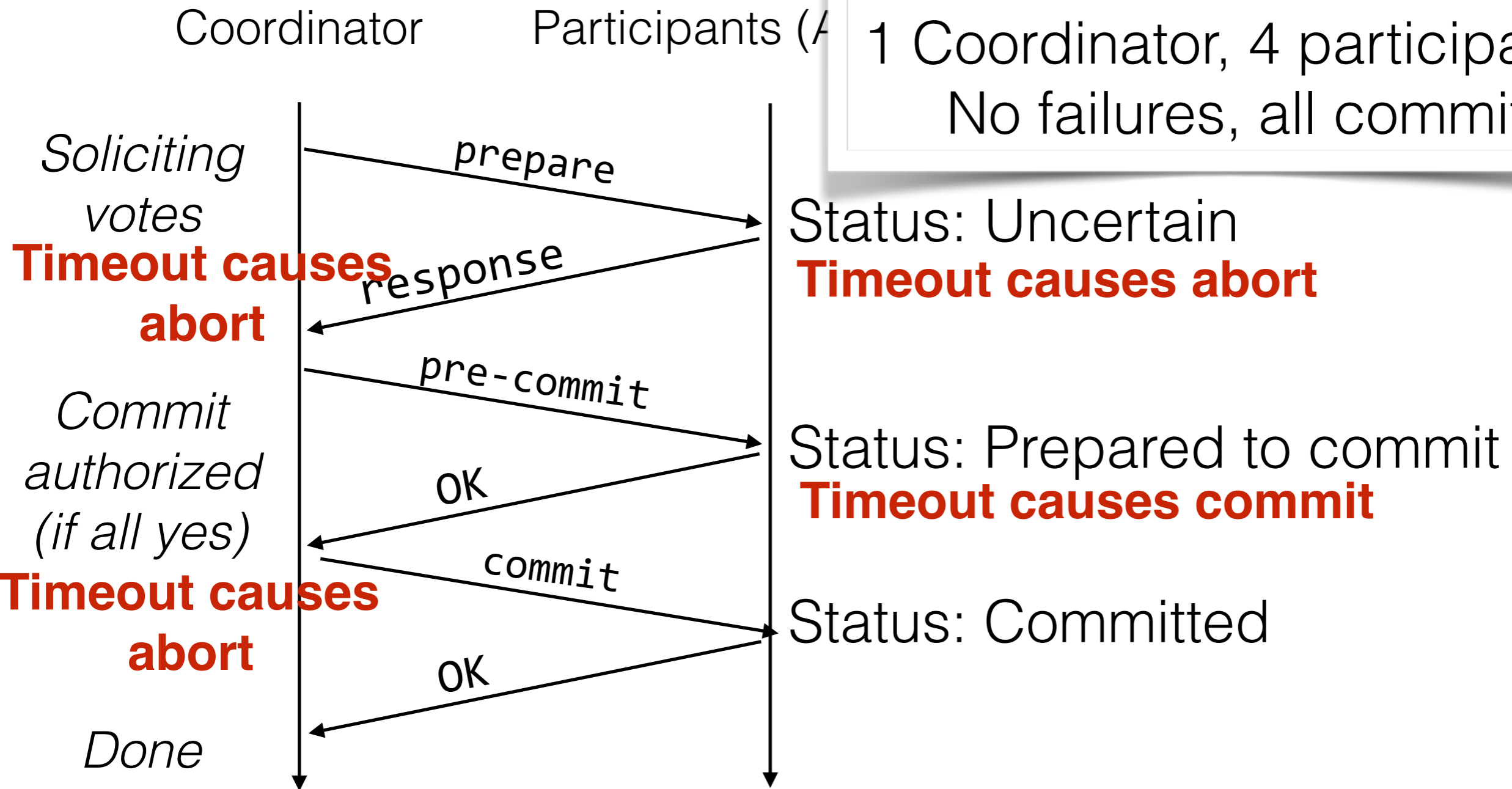
# 3PC Timeout Handling

Coordinator      Participants (A,B,C,D)



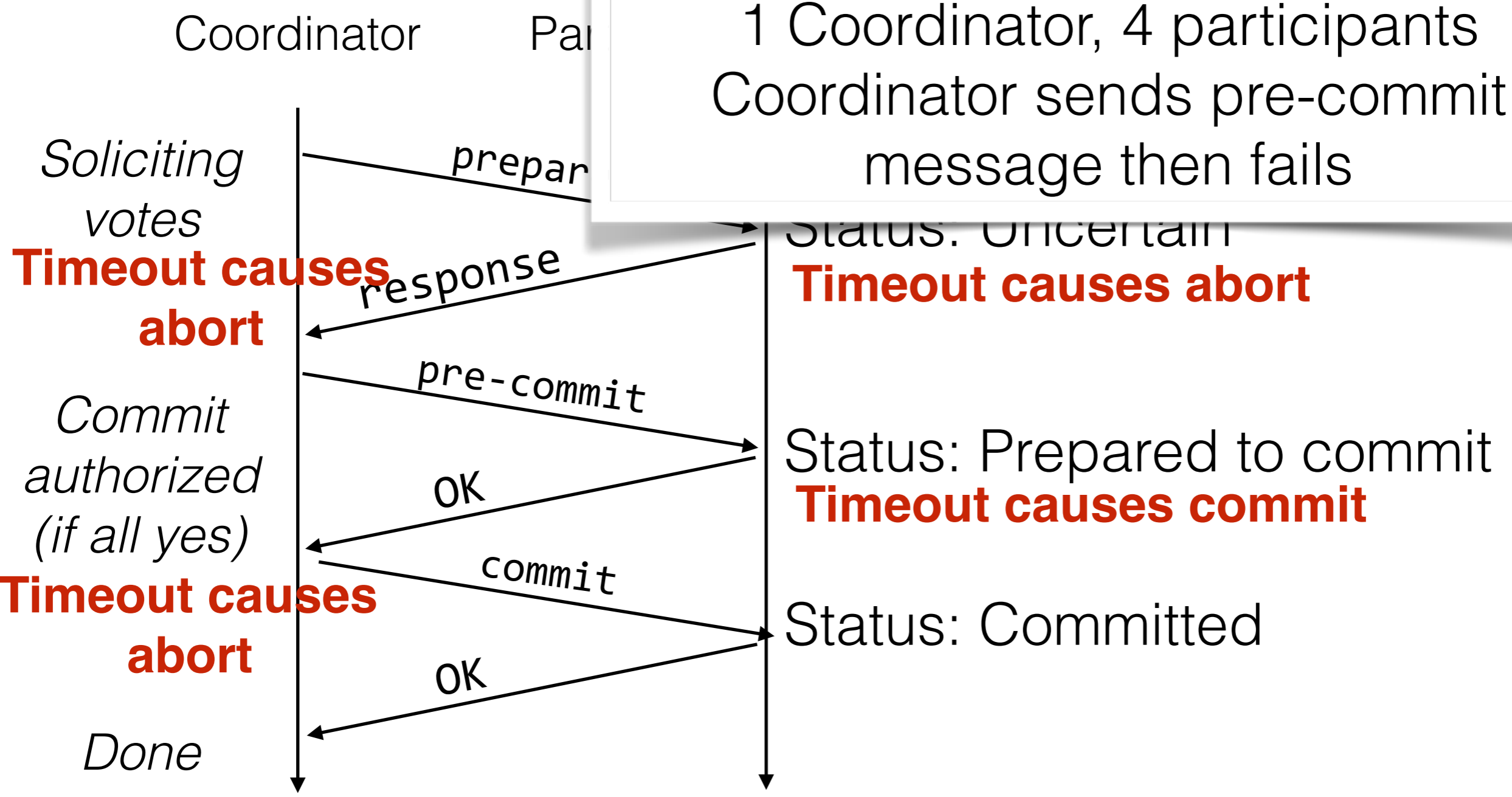
# 3PC Exercise

Exercise round 1:  
1 Coordinator, 4 participants  
No failures, all commit



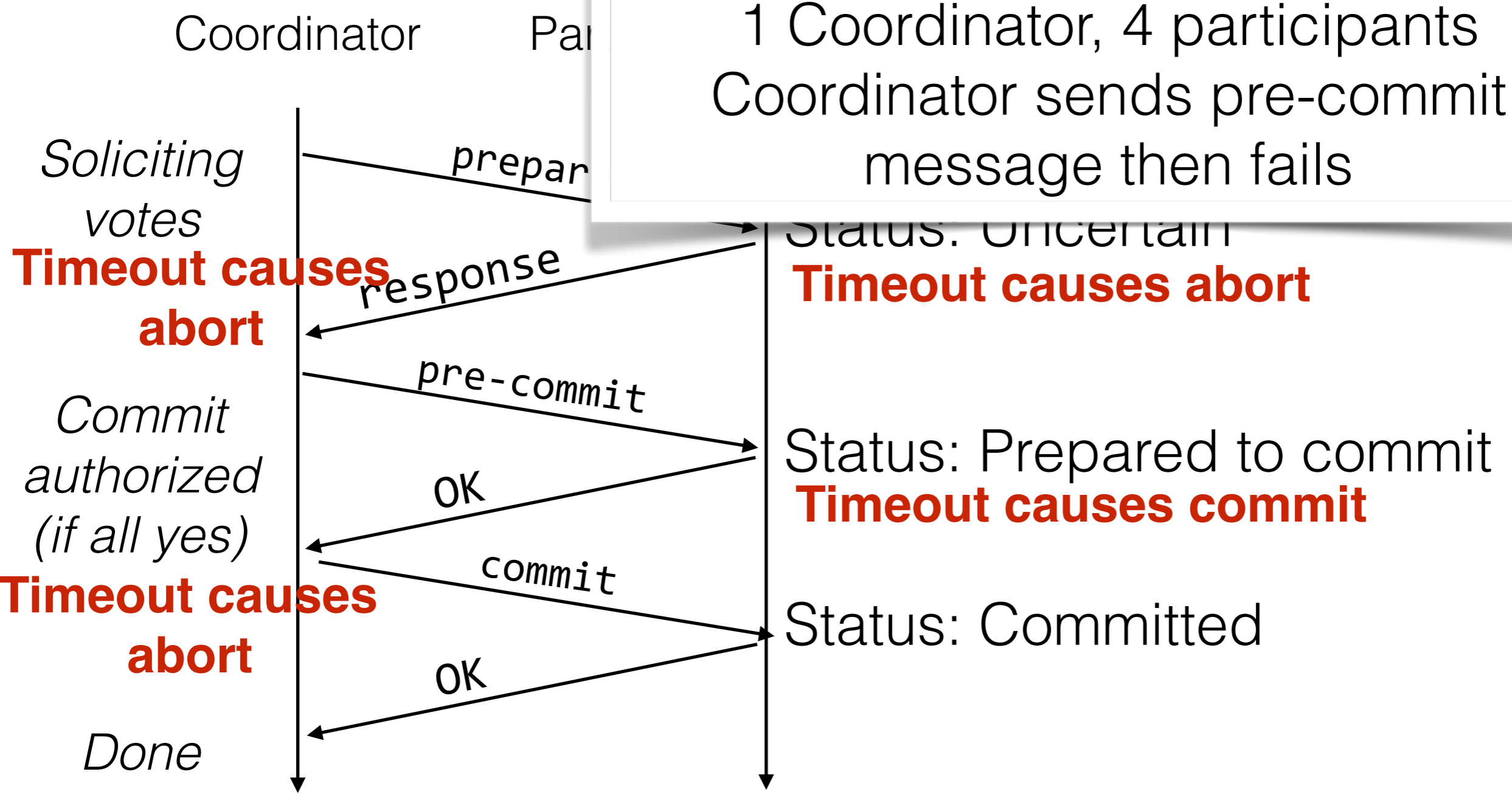
# 3PC Exercise

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



# 3PC Exercise

Exercise round 3:  
1 Coordinator, 4 participants  
Coordinator sends pre-commit message then fails

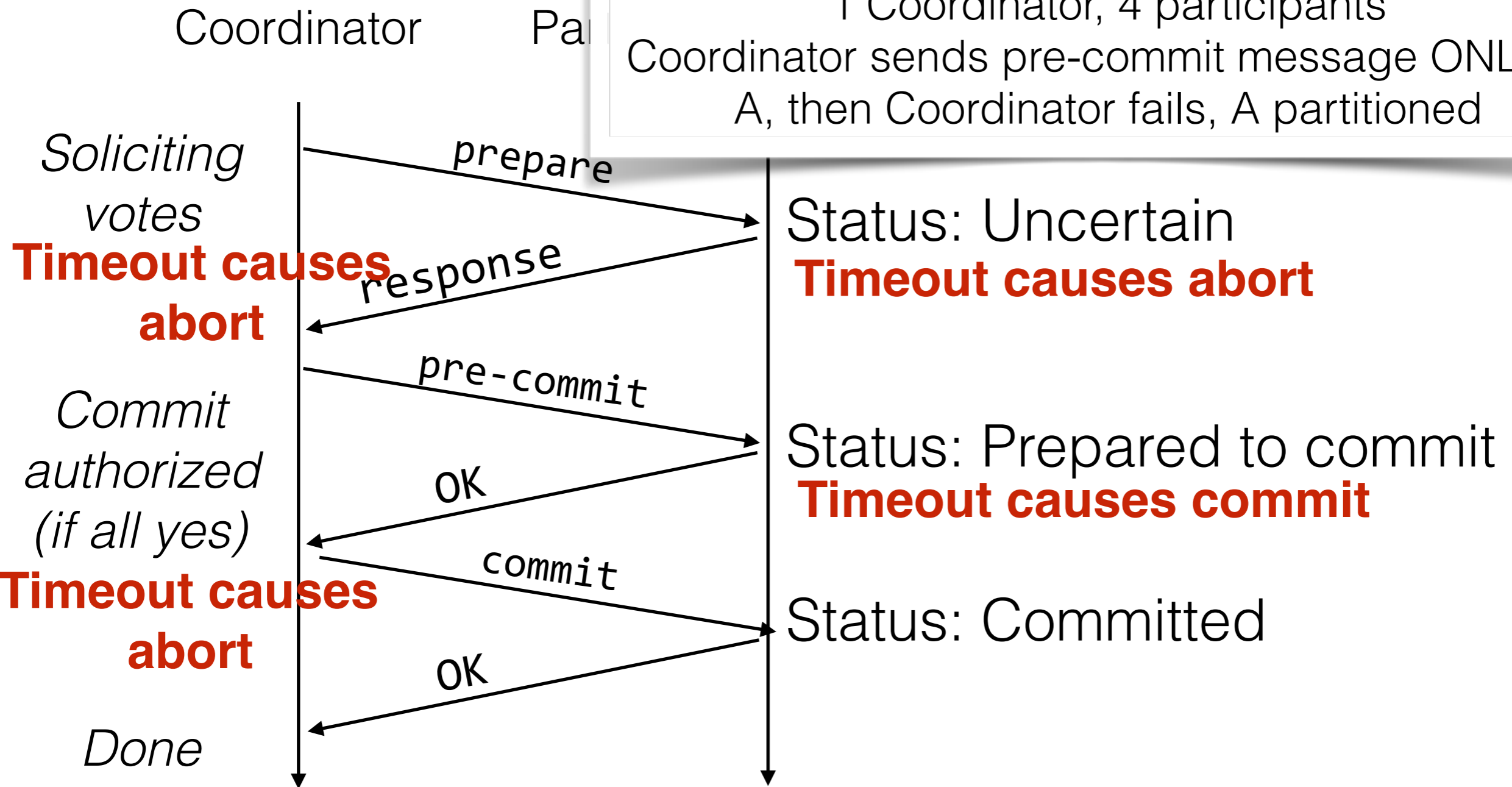


# Does 3PC guarantee consensus?

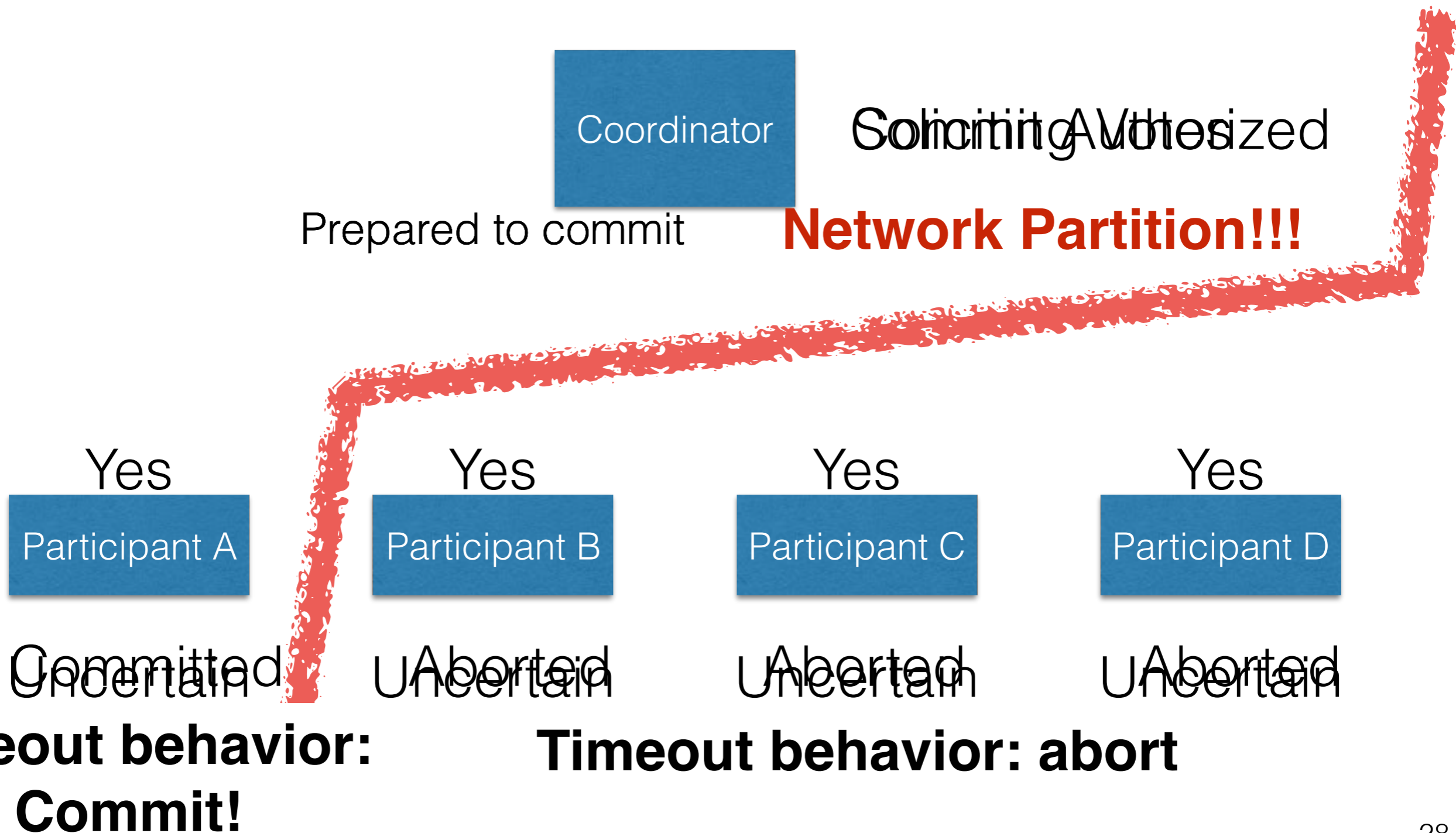
- Reminder, that means:
  - Liveness (availability)
    - **Yes!** Always terminates based on timeouts
  - Safety (correctness)
    - Hmm...

# 3PC Exercise

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



# Partitions



# 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)

Prepared to commit

**Network Partition!!!**

Can we let B, C, D proceed safely while stalling A and D?

Yes

Participant A

Committed  
Uncertain

Yes

Participant B

Aborted  
Uncertain

Yes

Participant C

Aborted  
Uncertain

Yes

Participant D

Aborted  
Uncertain

**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

## In Search of an

### Abstract

Raft is a consensus algorithm for many nodes. It produces a result equivalent to Paxos, but it is as efficient as Paxos, but its structure is simpler than Paxos; this makes Raft more understandable and also provides a better foundation for building practical systems. In order to enhance its safety, Raft separates the key elements of consensus: leader election, log replication, and safety. Raft also provides a stronger degree of coherency to reduce the number of states that must be considered. Results from our experiments demonstrate that Raft is easier to study and implement than Paxos. Raft also includes a new mechanism for managing the cluster membership, which uses only local operations to guarantee safety.

### 1 Introduction

Consensus algorithms allow a collection of processes to work as a coherent group that can tolerate the failure of some of its members. Because of its key role in building reliable large-scale distributed systems, Paxos [15, 16] has dominated the discussion of consensus algorithms over the last decade: most of the consensus algorithms are based on Paxos or inspired by Paxos. Paxos has become the primary vehicle for discussing distributed consensus.

Unfortunately, Paxos is quite difficult to implement in spite of numerous attempts to make it more practical. Furthermore, its architecture requires a lot of state to support practical systems. As a result, many system builders and students struggle with Paxos.

After struggling with Paxos ourselves, we decided to find a new consensus algorithm that could provide a better foundation for system building and implementation. Our approach was unusual in that our primary

## 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 [5] 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 interface 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 interact

# 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

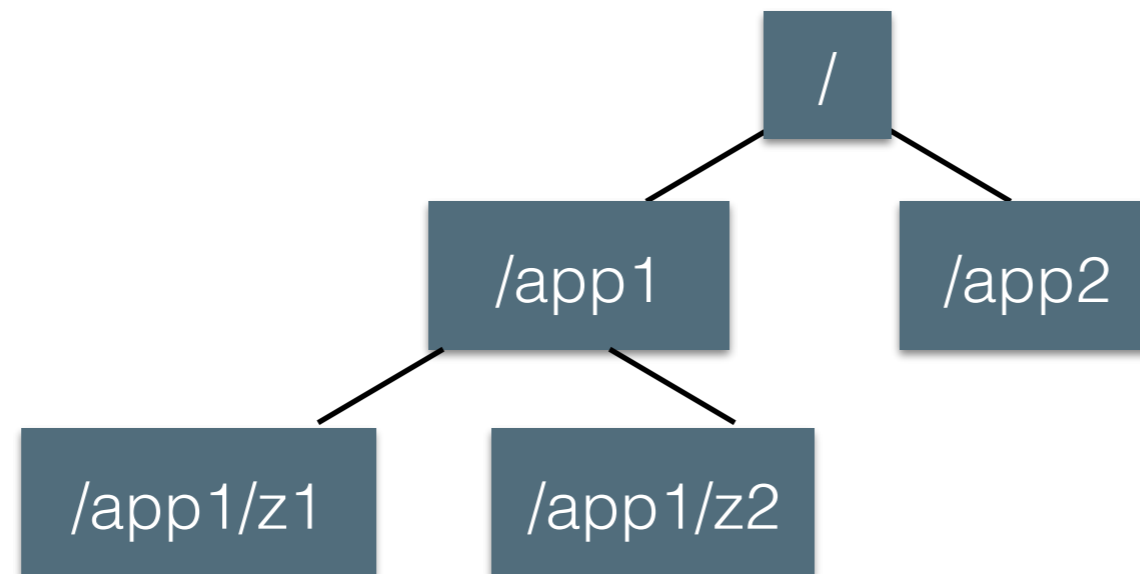
Just kidding!

# 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:
  - Maintaining group membership, distributed data structures, distributed locks, distributed protocol state, etc

# ZooKeeper - Data Model

- Provides a hierarchical namespace
- Each node is called a znode
- ZooKeeper provides an API to manipulate these nodes



# ZooKeeper - ZNodes

- In-memory data
- NOT for storing general data - just metadata (they are replicated and generally stored in memory)
- Map to some client abstraction, for instance - locks
- Znodes maintain counters and timestamps as metadata

# ZooKeeper - Znode Types

- Regular znodes
  - Can have children znodes
  - Created and deleted by clients explicitly through API
- Ephemeral znodes
  - Cannot have children
  - Created by clients explicitly
  - Deleted by clients OR removed automatically when client session that created them disconnects

# ZooKeeper - API

- Clients track changes to znodes by registering a **watch**
- Create(path, data, flags)  
Delete(path, version)  
Exists(path, watch)  
getData(path, watch)  
setData(path, data, version)  
getChildren(path, watch)  
Sync(path)

# ZooKeeper - Guarantees

- **Liveness guarantees:** if a majority of ZooKeeper servers are active and communicating the service will be available
- **Durability guarantees:** if the ZooKeeper service responds successfully to a change request, that change persists across any number of failures as long as a quorum of servers is eventually able to recover

# ZooKeeper - Lock Example

- To acquire a lock called **foo**
- Try to create an ephemeral znode called **/locks/foo**
- If you succeeded:
  - You have the lock
- If you failed:
  - Set a watch on that node. When you are notified that the node is deleted, try to create it again.

# ZooKeeper - Recipes

- Why figure out how to re-implement this low level stuff (like locks)?
- Recipes: <https://zookeeper.apache.org/doc/r3.3.6/recipes.html>
  - And in Java: <http://curator.apache.org>
- Examples:
  - Locks
  - Group Membership

# How Many ZooKeepers?

- How many ZooKeepers do you want?
  - An odd number
  - 3-7 is typical
  - Too many and you pay a LOT for coordination

# ZooKeeper + HW4

- We are NOT going to run a ZooKeeper for each CFS client
- In practice, you really do not want more than ~5 ZooKeepers at any given time. But (in principle) we want more CFS clients.
- Hence, we will replace our 1 lock server with 1 script that starts 5 ZooKeepers

# ZooKeeper - Lab

- Addressbook client...
- No single lock server. Use ZooKeeper (well, a script that will create 5 ZooKeepers)