

# Transactions

CS 475, Spring 2018  
Concurrent & Distributed Systems

# Review: Transactions

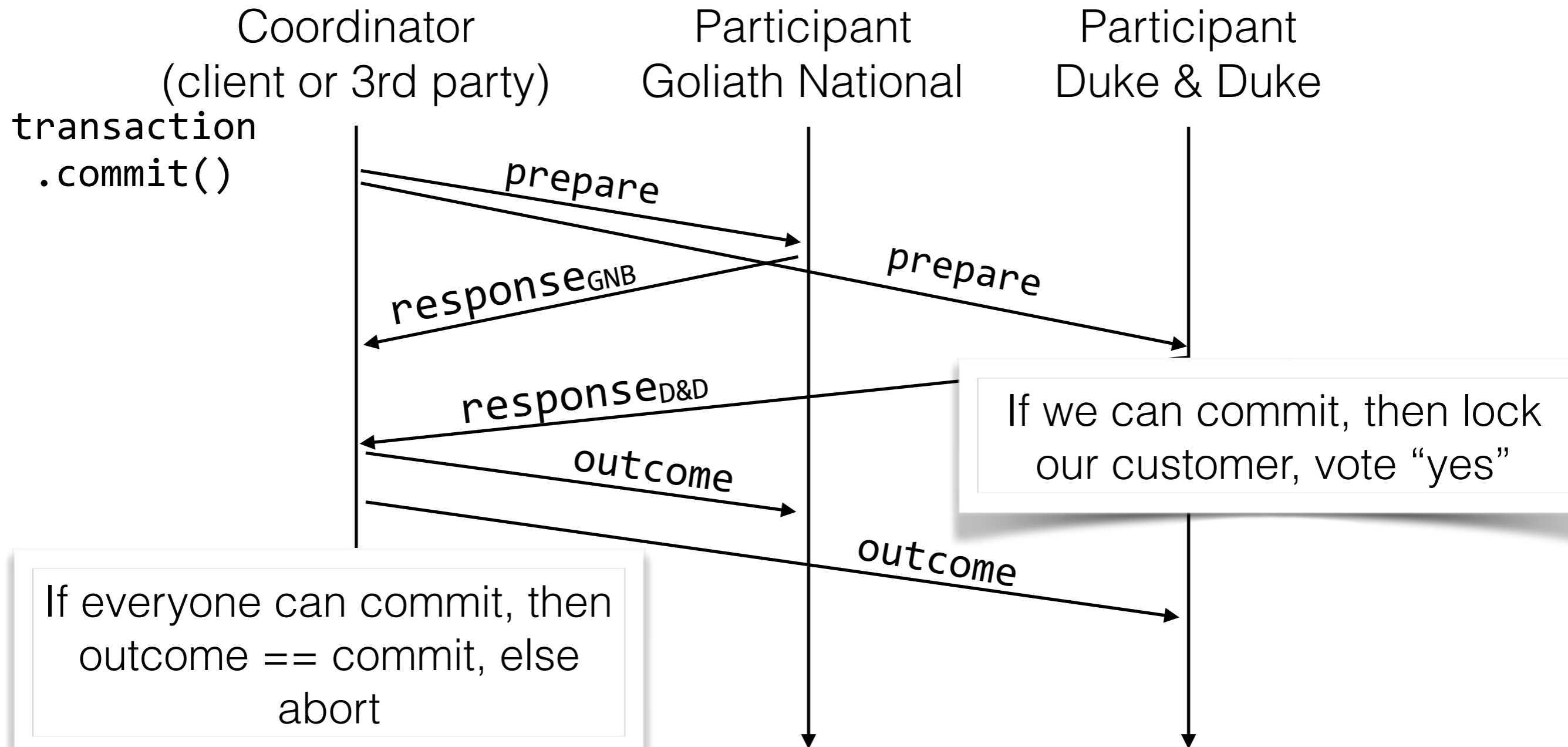
```
boolean transferMoney(Person from, Person
to, float amount){
    if(from.balance >= amount)
    {
        from.balance = from.balance -
amount;
        to.balance = to.balance + amount;
        return true;
    }
    return false;
}
```

Assume running on a single machine:  
What can go wrong here?

# Review: Properties of Transactions

- Traditional properties: ACID
- **Atomicity**: transactions are “all or nothing”
- **Consistency**: Guarantee some basic properties of data; each transaction leaves the database in a valid state
- **Isolation**: Each transaction runs as if it is the only one; there is some valid serial ordering that represents what happens when transactions run concurrently
- **Durability**: Once committed, updates cannot be lost despite failures

# Review: 2PC



# Review: Recovery on Reboot

- If coordinator finds no “commit” message on disk, abort
- If coordinator finds “commit” message, commit
- If participant finds no “yes, ok” message, abort
- If participant finds “yes, ok” message, then replay that message and continue protocol

# Announcements

- HW4 is out!
  - <http://www.jonbell.net/gmu-cs-475-spring-2018/homework-4/>
- Today:
  - Agreement & transactions in distributed systems (continued)
  - Reminder: lecture from last week is posted on YouTube
- Additional readings:
  - <http://the-paper-trail.org/blog/consensus-protocols-two-phase-commit/>
  - <http://the-paper-trail.org/blog/consensus-protocols-three-phase-commit/>
  - Tannenbaum Note 8.13 (“Advanced”!)

# 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!

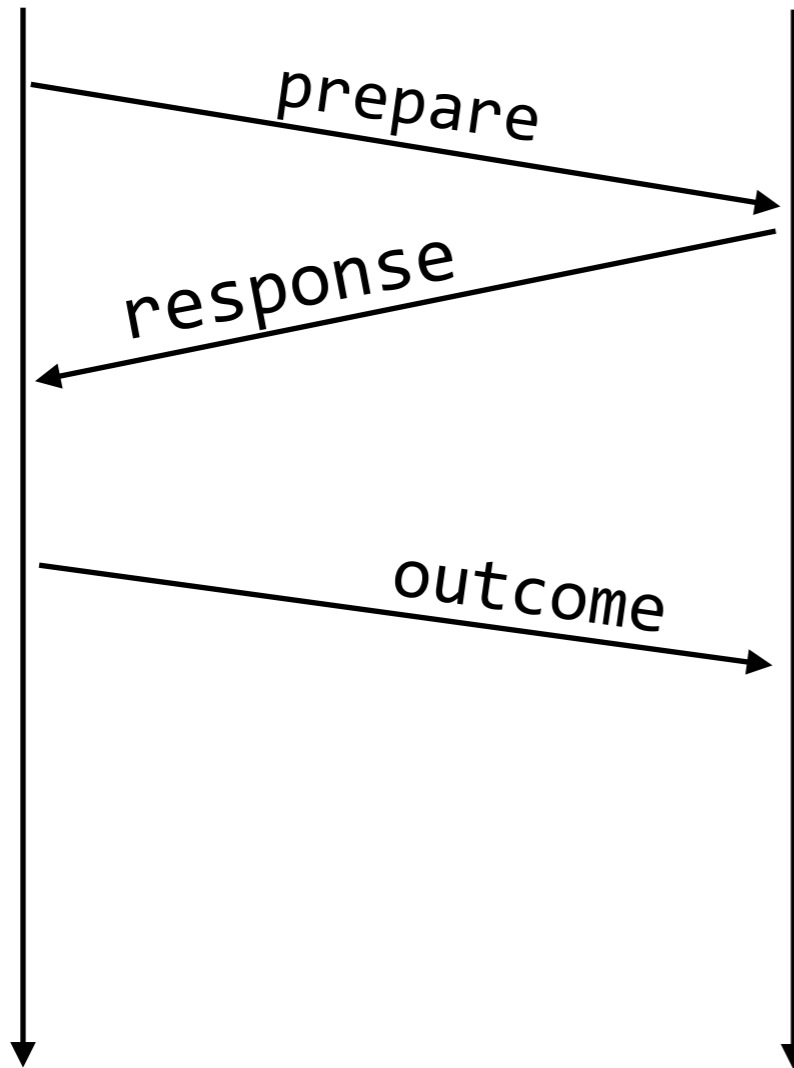
# 2PC Timeouts

- 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

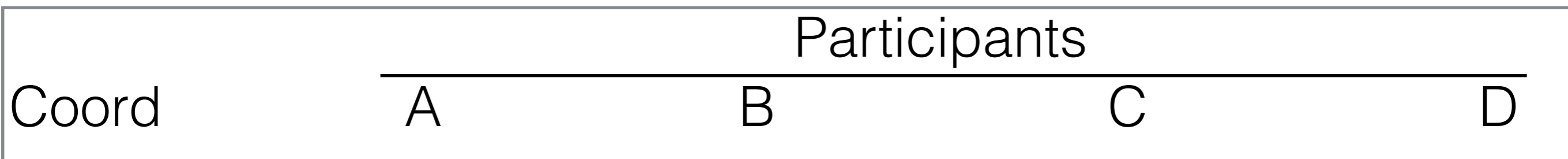
# 2PC Exercise

Coordinator  
(client or 3rd party)

Participant



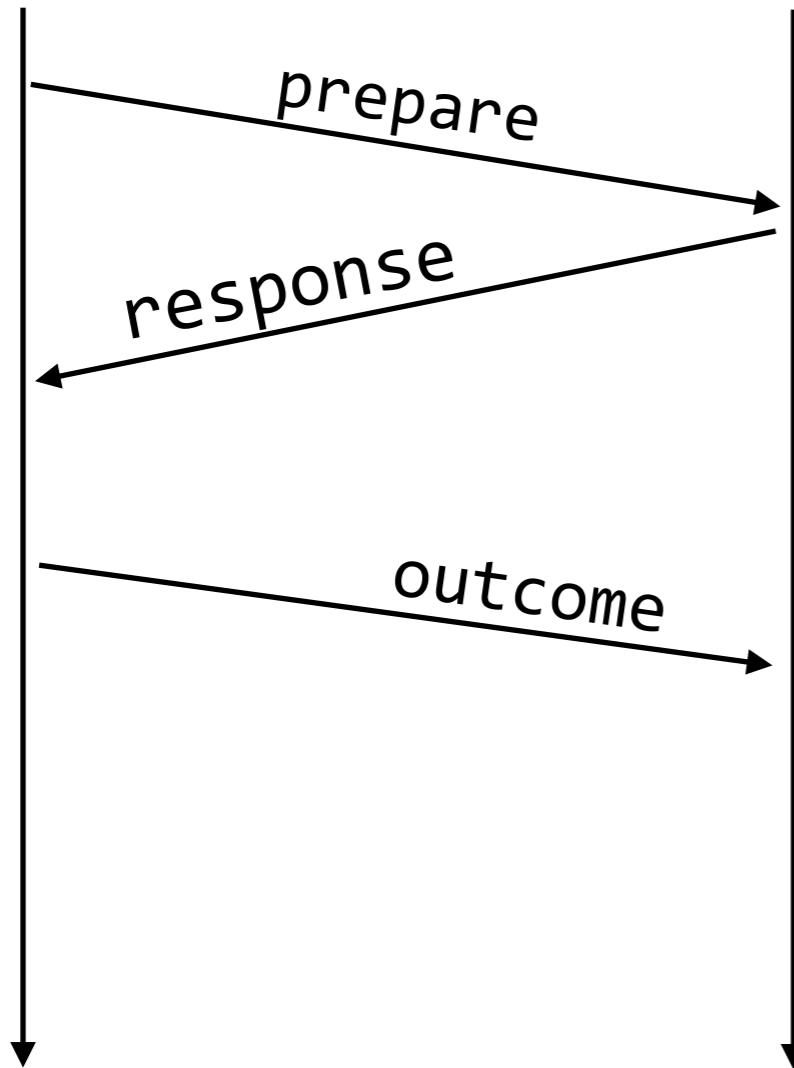
Exercise round 1:  
1 Coordinator, 4 participants  
No failures, all commit



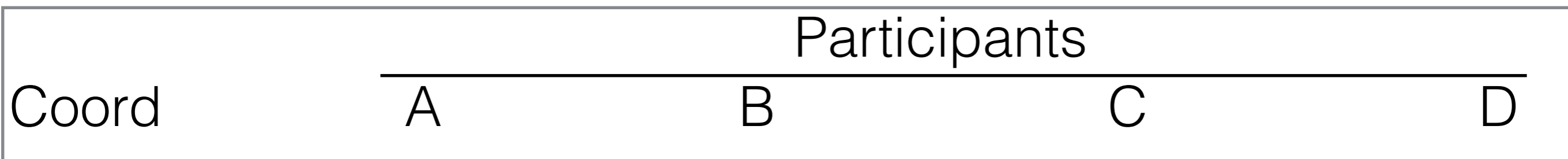
# 2PC Exercise

Coordinator  
(client or 3rd party)

Participant



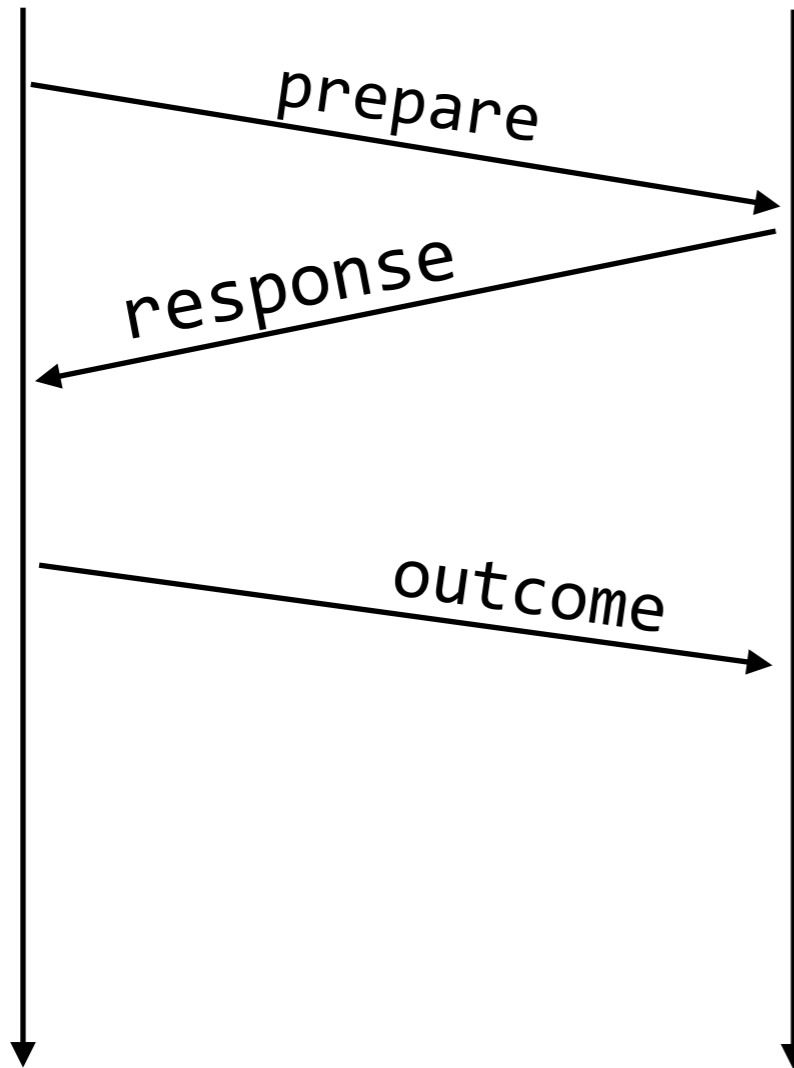
Exercise round 2:  
1 Coordinator, 4 participants  
Coordinator fails before providing  
outcome



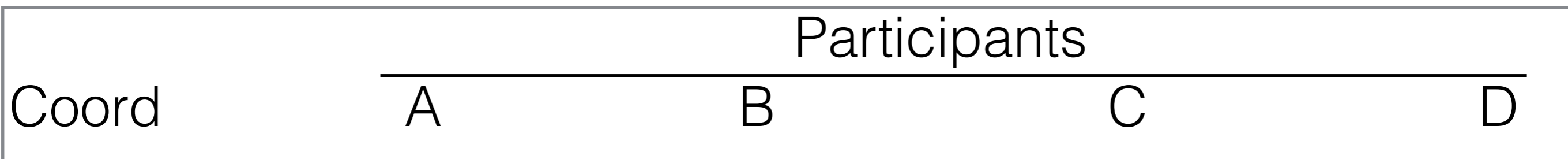
# 2PC Exercise

Coordinator  
(client or 3rd party)

Participant



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



Voted yes  
Heard back "commit"



Voted yes  
**Did not hear result**



Voted yes  
**Did not hear result**



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
  - 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*

Status: Uncertain

**Timeout causes abort**

*response*

*pre-commit*

*Commit  
authorized  
(if all yes)*

Status: Prepared to commit

**Timeout causes commit**

*OK*

*commit*

**Timeout causes  
abort**

Status: Committed

*OK*

*Done*

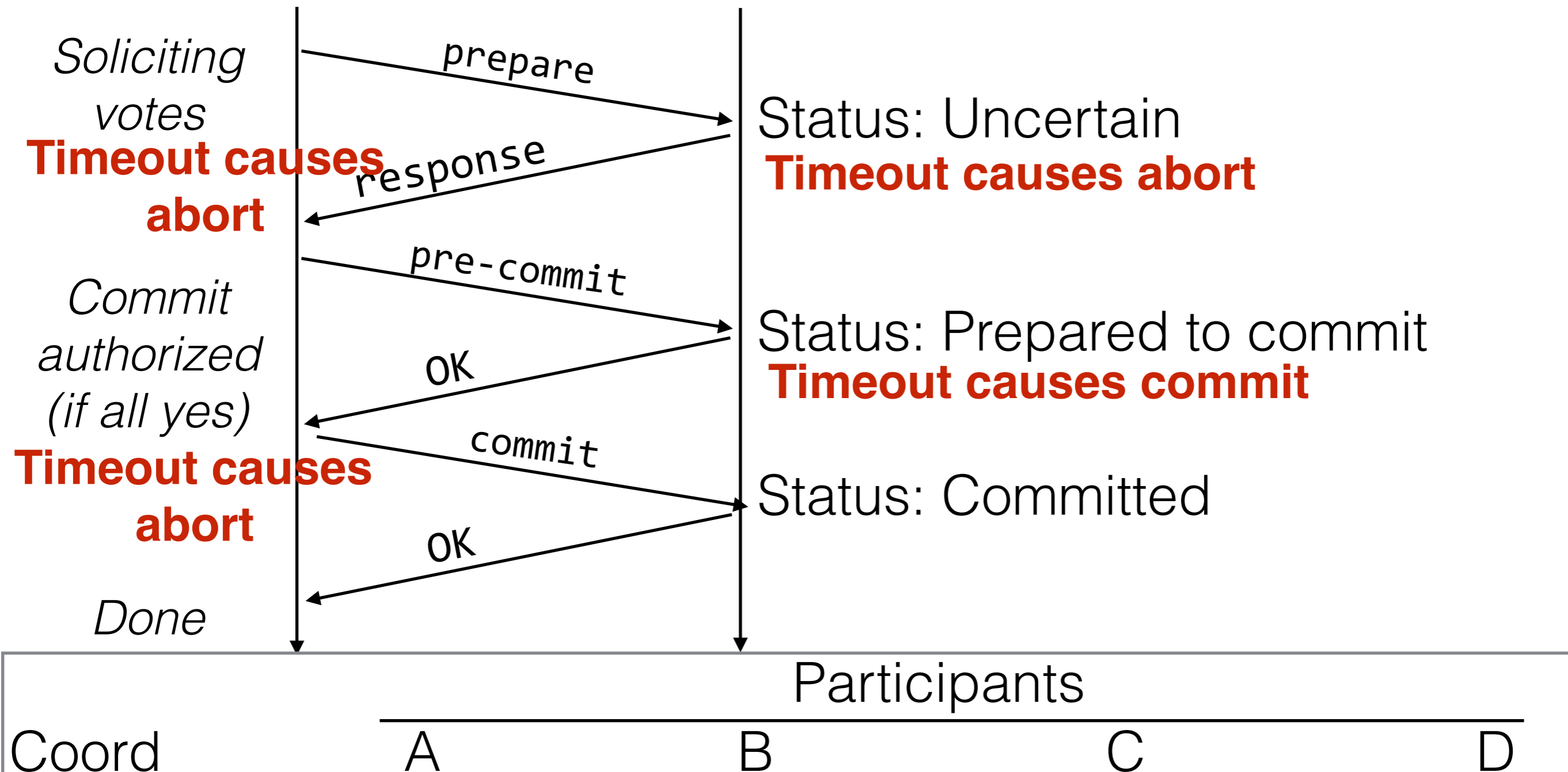
# 3PC Ex

Scenario:

1 Coordinator, 4 participants  
No failures, all commit

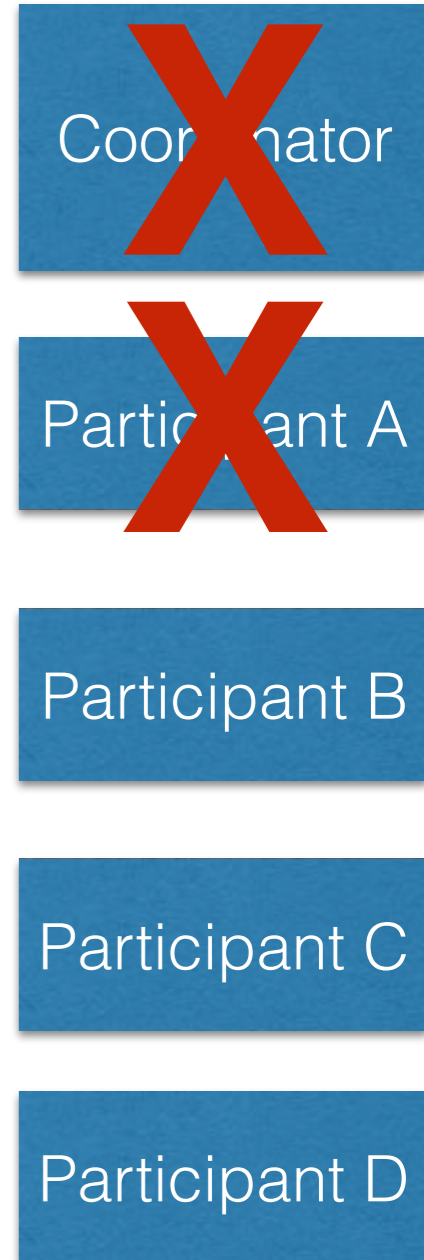
Coordinator

Participants (A,B,C,D)



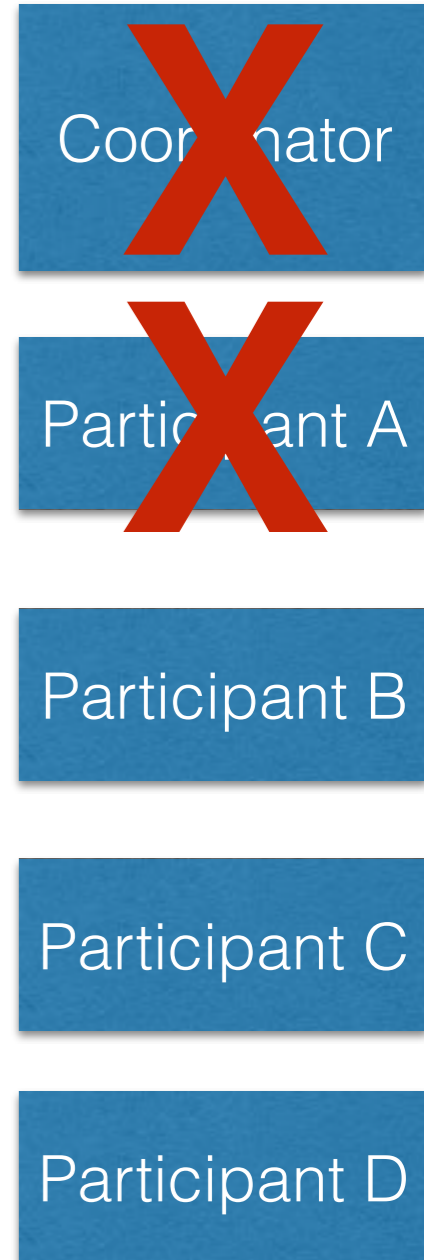
# 3PC Crash Handling

- 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.



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

Coordinator

Participants (A,B,C,D)

*Soliciting  
votes*

**Timeout causes  
abort**

*Commit  
authorized  
(if all yes)*

**Timeout causes  
abort**

*Done*

*prepare*

*response*

*pre-commit*

*OK*

*commit*

*OK*

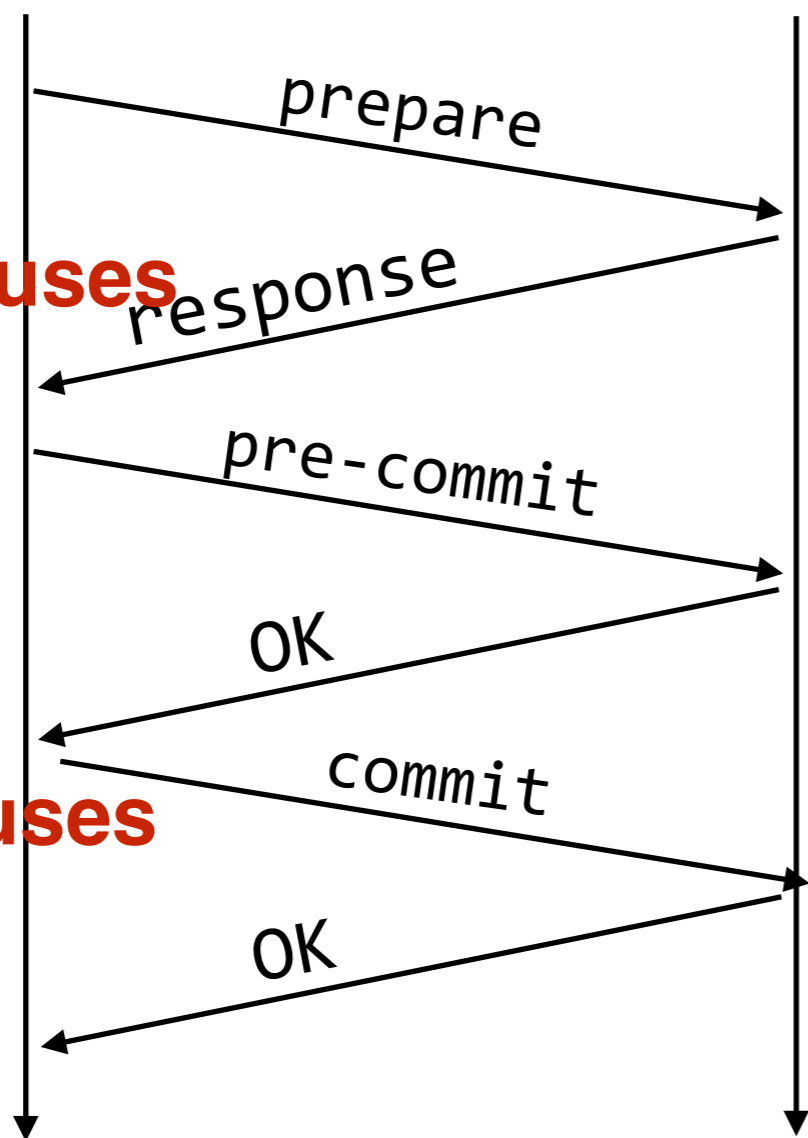
Status: Uncertain

**Timeout causes abort**

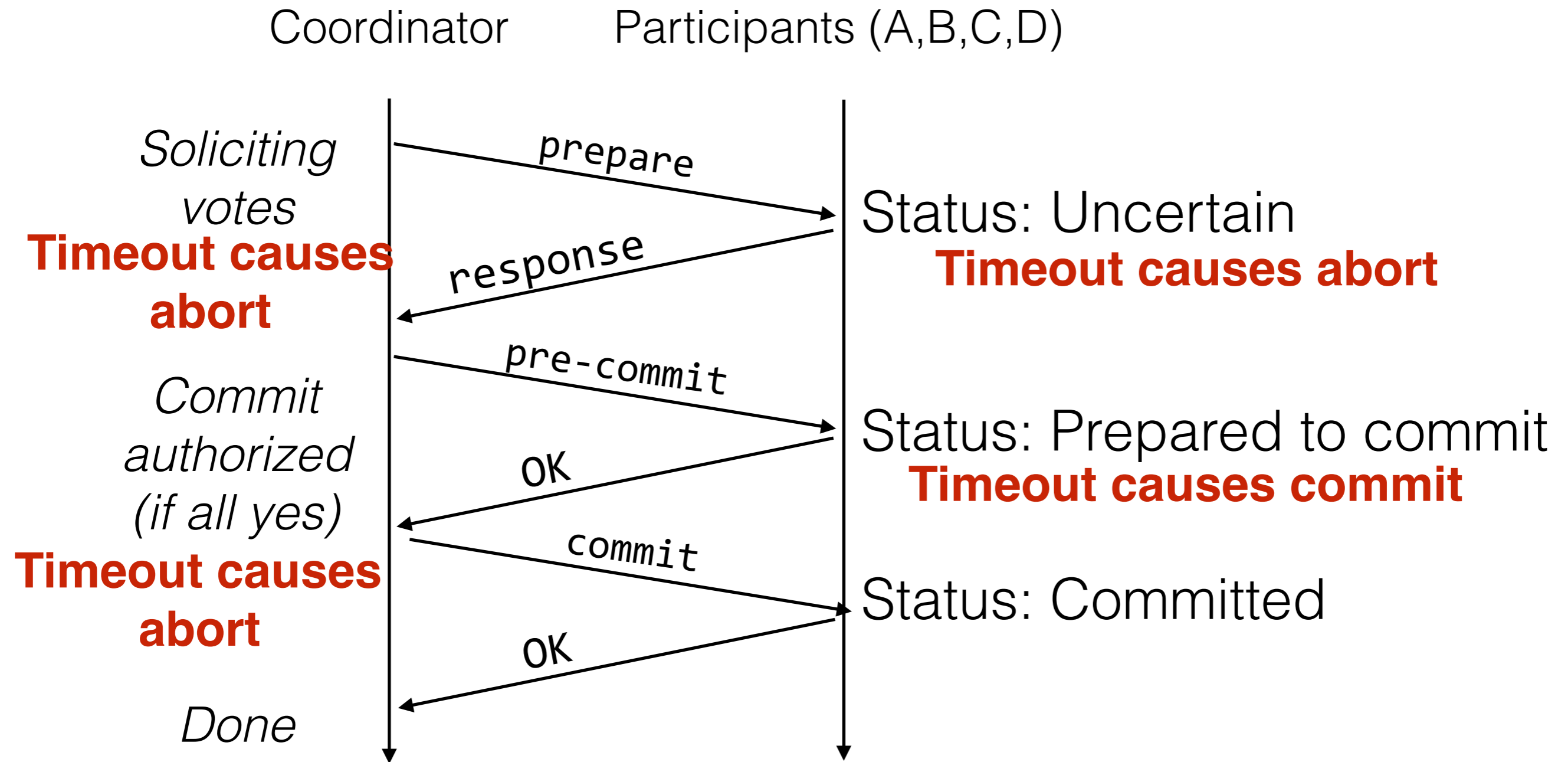
Status: Prepared to commit

**Timeout causes commit**

Status: Committed



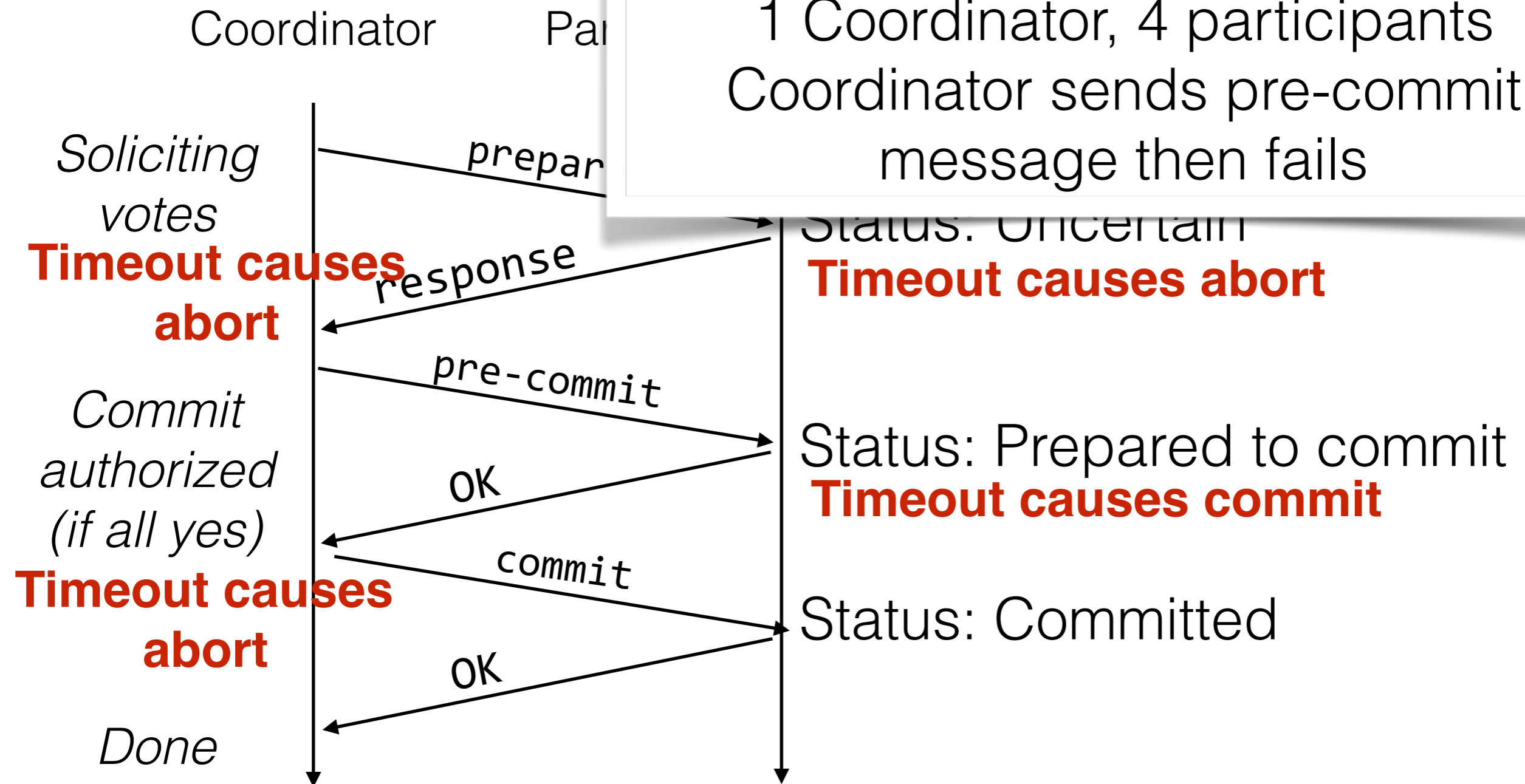
# 3PC Timeout Handling



# 3PC Exercise

Exercise round 2:

1 Coordinator, 4 participants  
Coordinator sends pre-commit  
message then fails



# 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 file

# 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

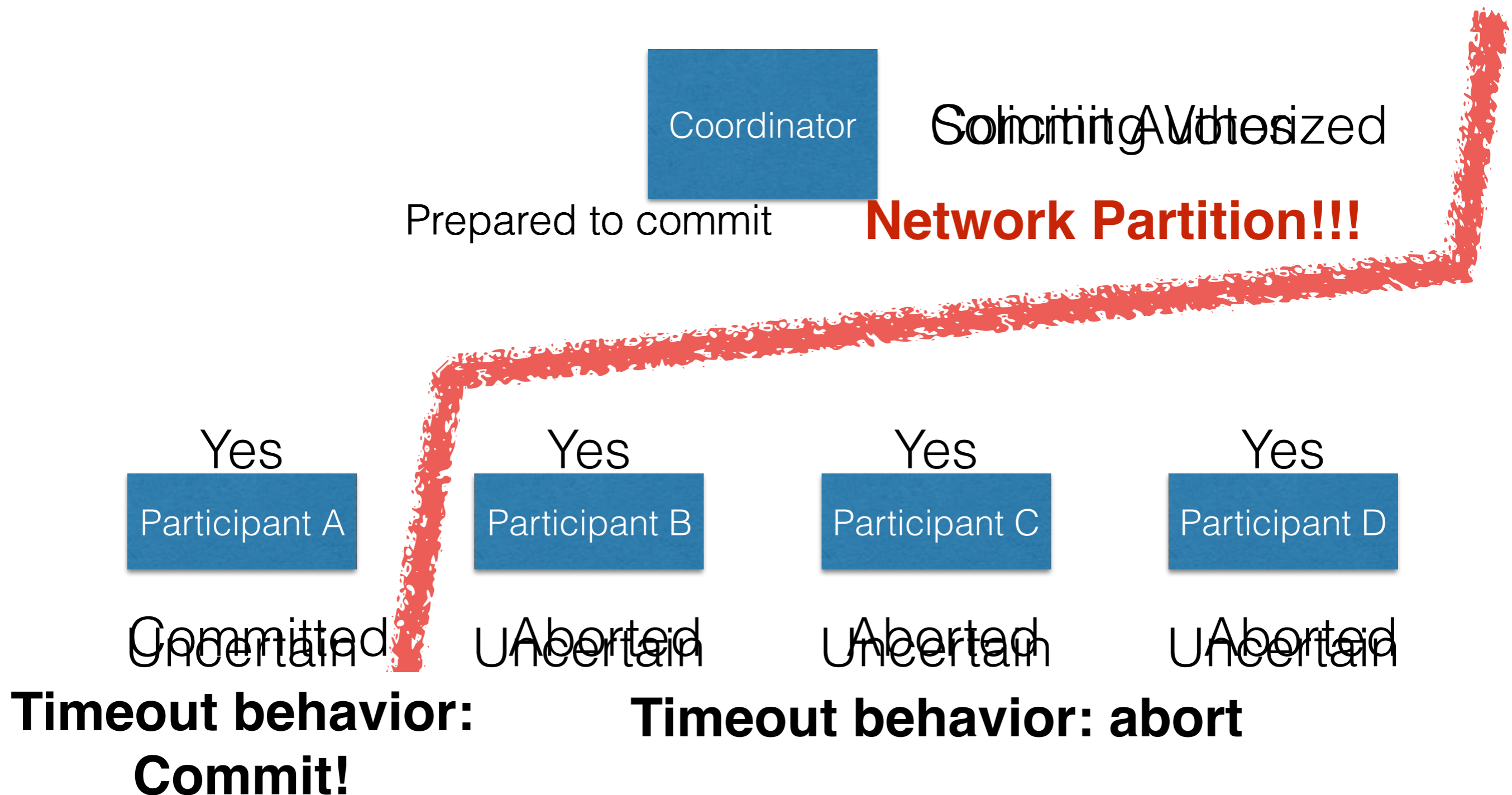
# Properties of Agreement

- **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

# Does 3PC guarantee agreement?

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

# Partitions

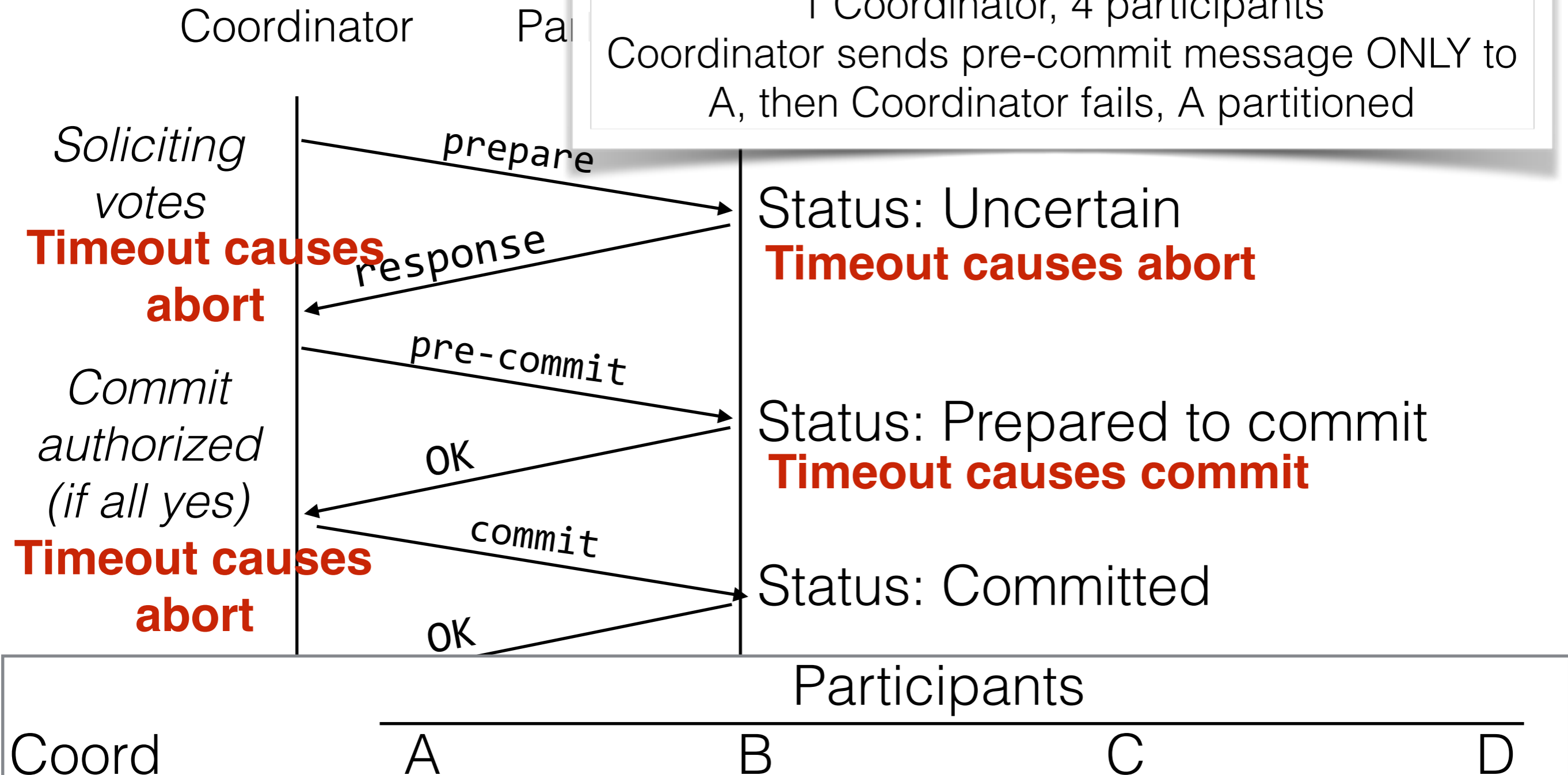


# 3PC Exercise

Scenario:

1 Coordinator, 4 participants

Coordinator sends pre-commit message ONLY to A, then Coordinator fails, A partitioned



# 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 managing a replicated log. It produces a result equivalent to (or better than) Paxos; it is as efficient as Paxos, but its structure is simpler than Paxos; this makes Raft more understandable and easier to implement than Paxos and also provides a better foundation for building practical systems. In order to enhance its safety, Raft separates the key elements of consensus into three modules: leader election, log replication, and safety. This separation allows for a stronger degree of coherency to reduce the number of states that must be considered. Results from a formal analysis demonstrate that Raft is easier to study and implement than Paxos. Raft also includes a new mechanism for managing the cluster membership, which uses timeouts 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 simplicity, a key role in building reliable large-scale systems. Paxos [15, 16] has dominated the discussion of consensus algorithms over the last decade; most of consensus are based on Paxos or inspired by Paxos. Paxos has become the primary vehicle for research and discussion about 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 ease of use. 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 impact

# 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 - 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