Transactions

CS 475, Spring 2018
Concurrent & Distributed Systems

GEORGE MASON UNIVERSITY

Review: Transactions

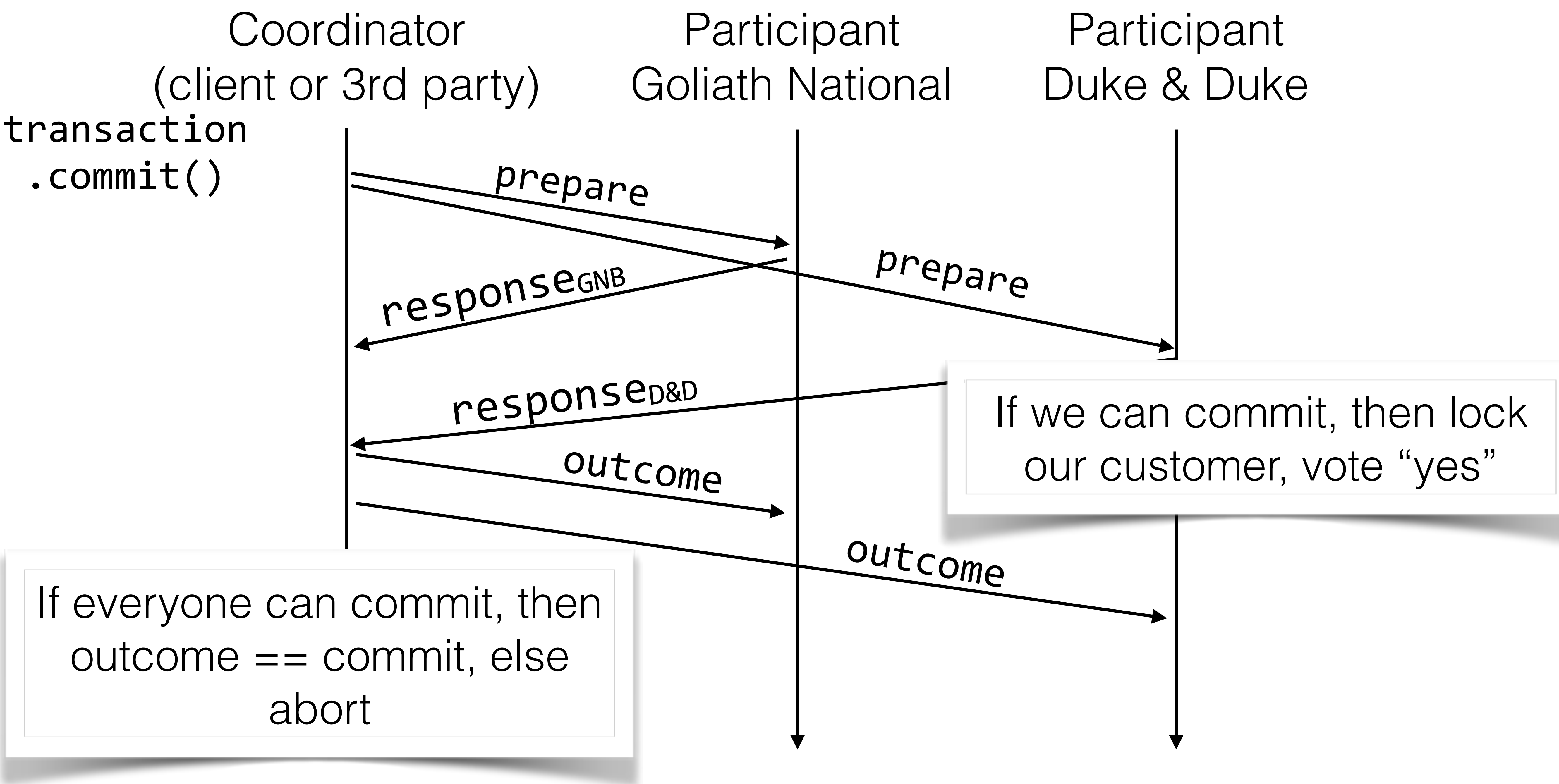
```
boolean transferMoney(Person from, Person
to, float amount){
    if(from.balance >= amount)
    {
        from.balance = from.balance -
amount;
        to.balance = to.balance + amount;
        return true;
    }
    return false;
}
```

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


Coordinator
(client or 3rd party)
Participant
Goliath National
Participant
Duke & Duke
transaction
.commit()
prepare
prepare
responseGNB
responseD&D
outcome
outcome
If we can commit, then lock our customer, vote “yes”
If everyone can commit, then outcome == commit, else abort

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?

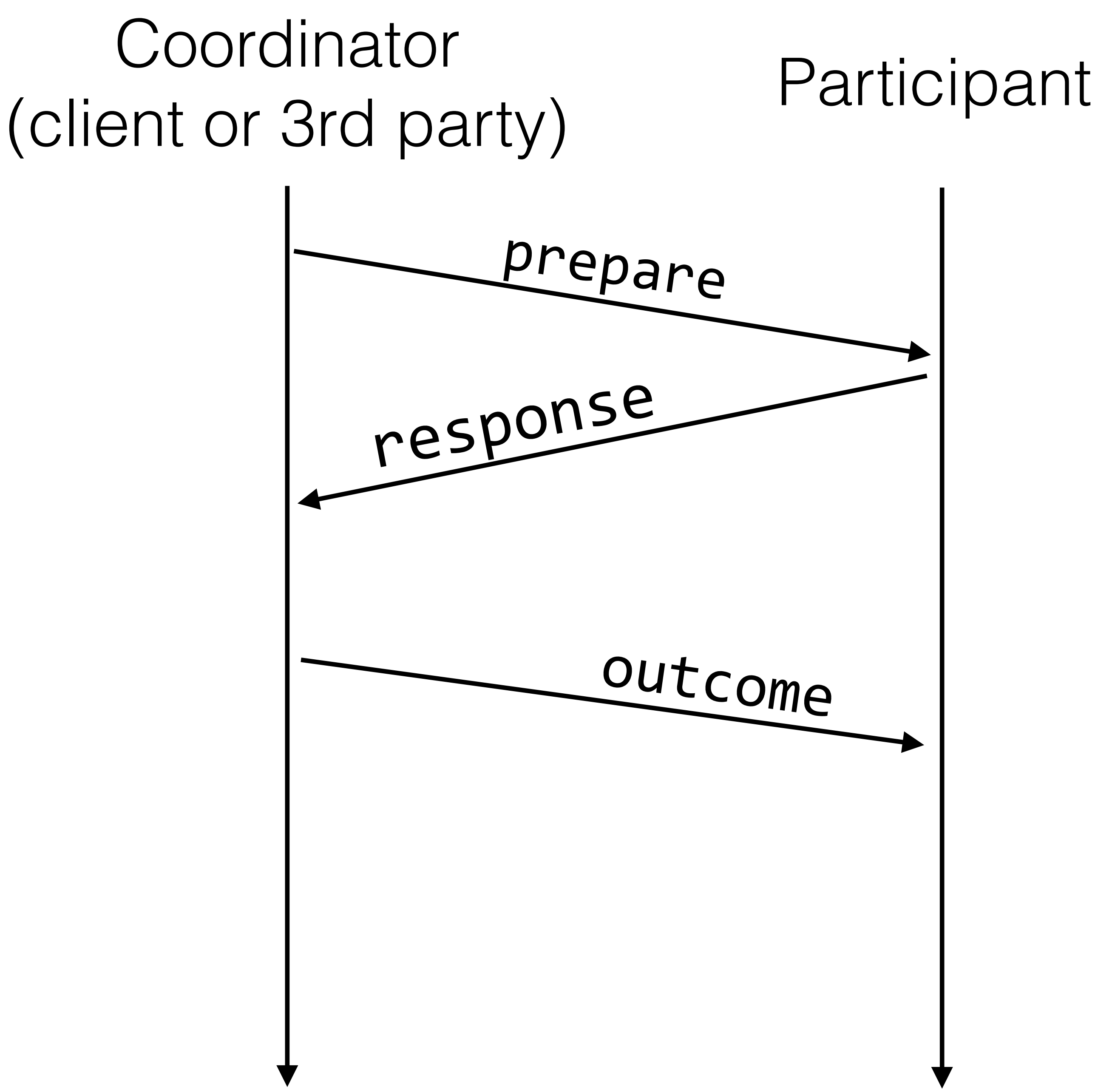
Handling Bank Timeouts

- 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

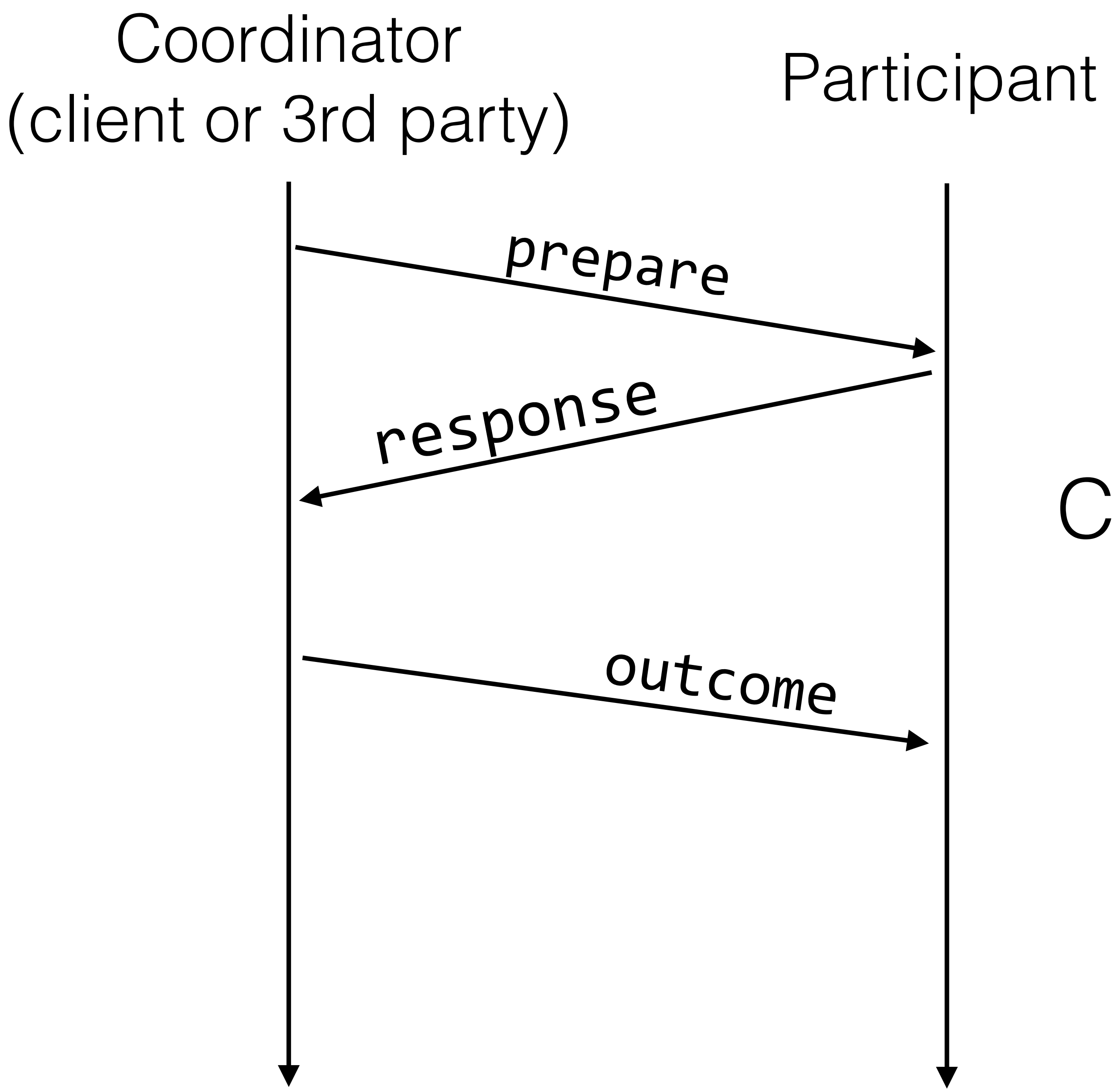
2PC Exercise



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

	Participants			
Coord	A	B	C	D

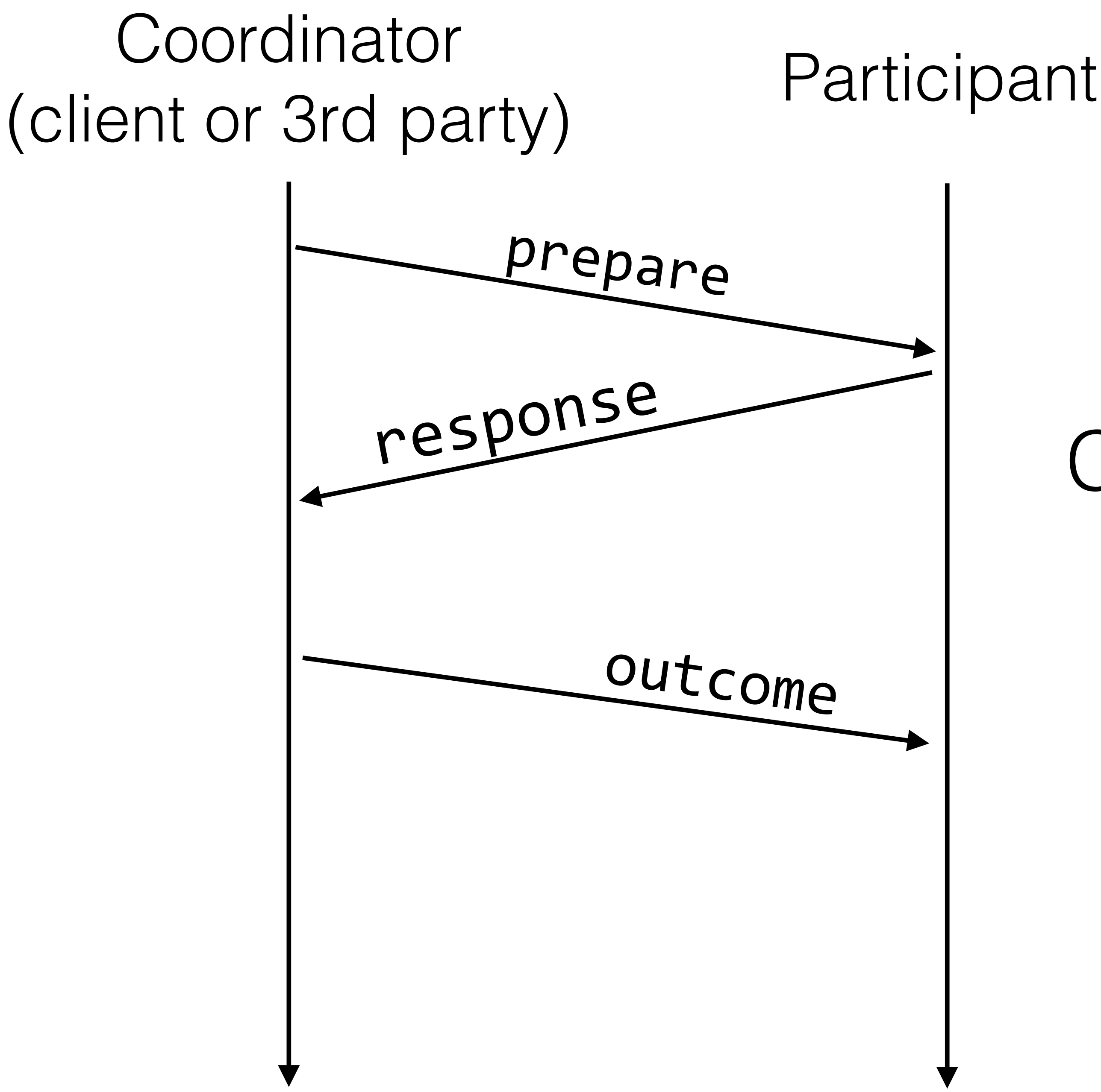
2PC Exercise



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

	Participants			
Coord	A	B	C	D

2PC Exercise



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

	Participants			
Coord	A	B	C	D

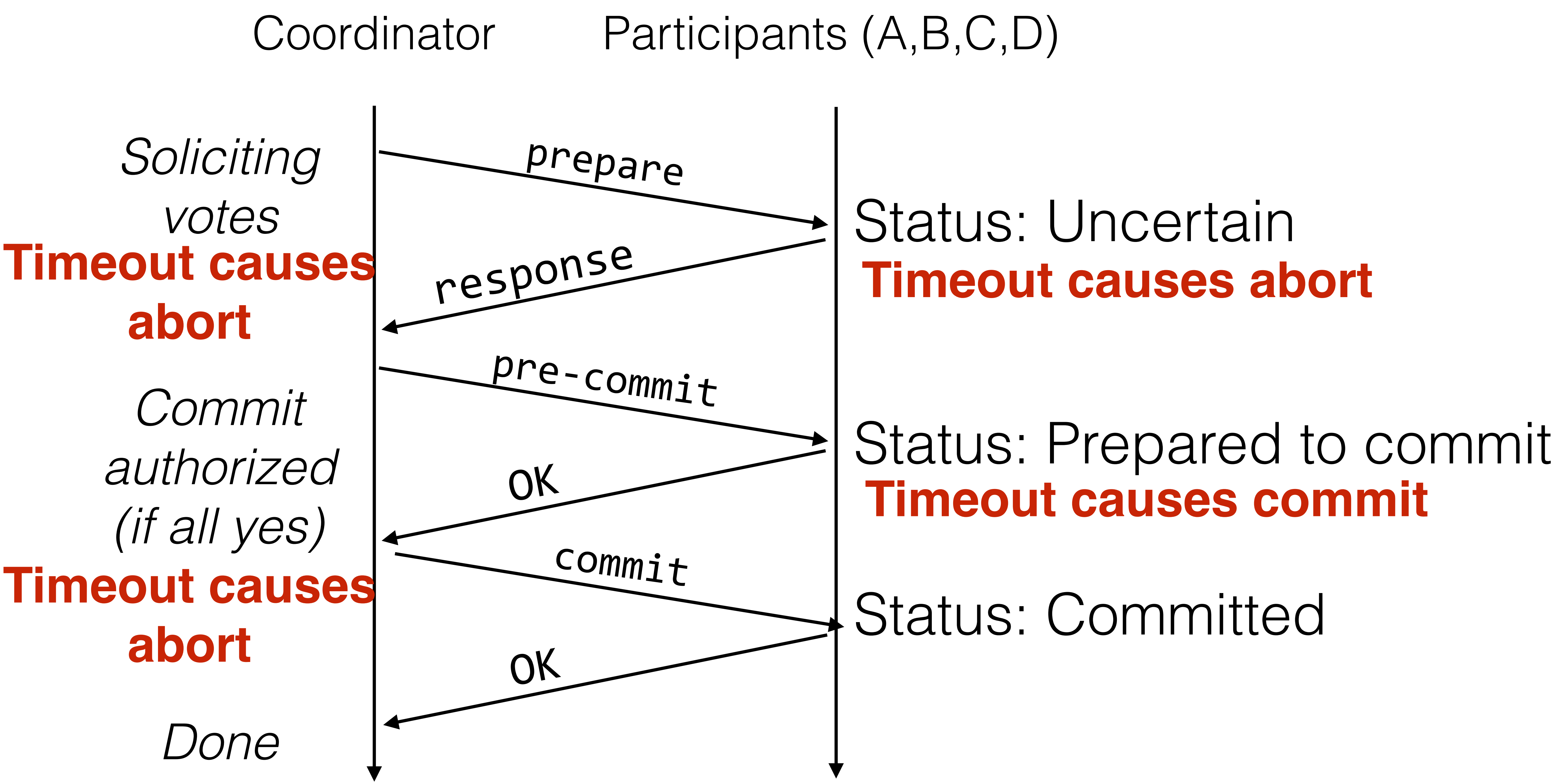
3 Phase Commit

- Goal: Eliminate this specific failure from blocking liveness

3 Phase Commit

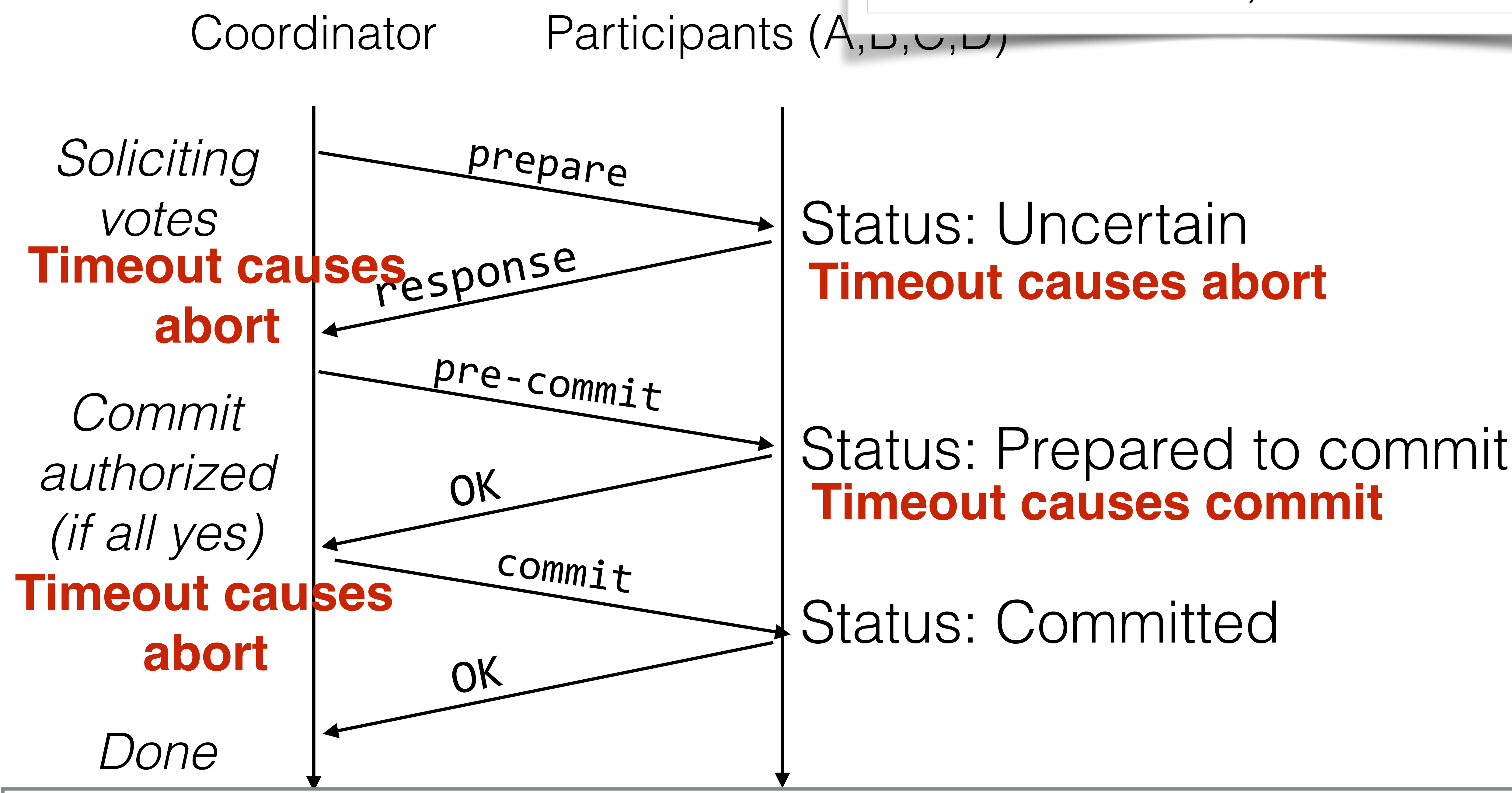
- Goal: Avoid blocking on node failure
- How?
  - Think about how 2PC is better than 1PC
    - 1PC means you can never change your mind or have a failure after committing
    - 2PC **still** means that you can’t have a failure after committing (committing is irreversible)
- 3PC idea:
  - Split commit/abort 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
Timeout causes abort
commit
Status: Committed
OK
Done

3PC Ex

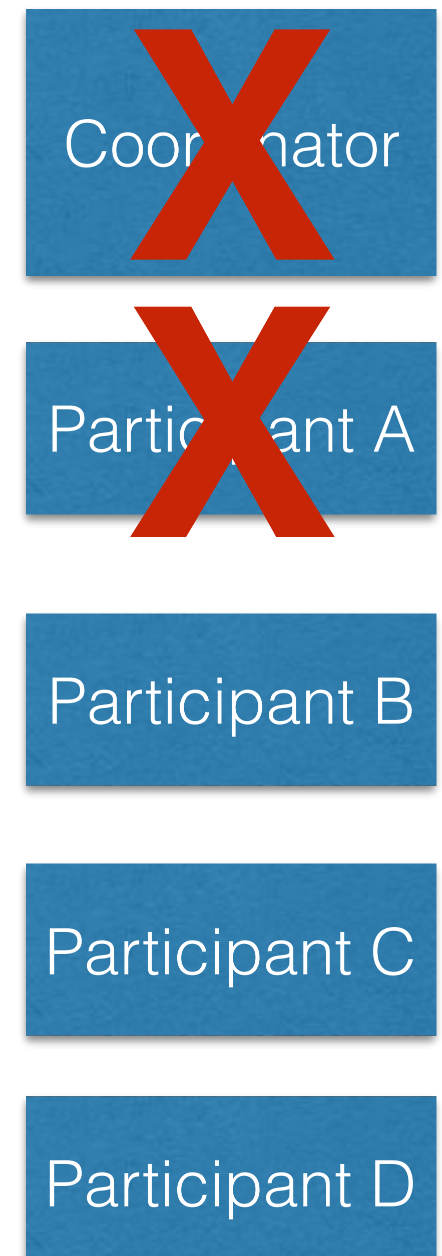
Scenario:
1 Coordinator, 4 participants
No failures, all commit



	Participants			
Coord	A	B	C	D

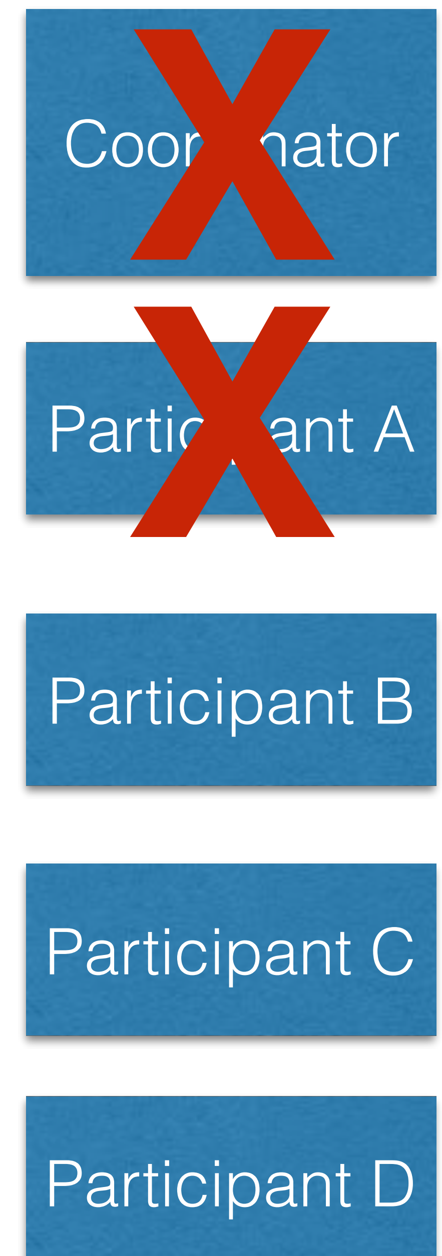
3PC Crash Handling

- 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

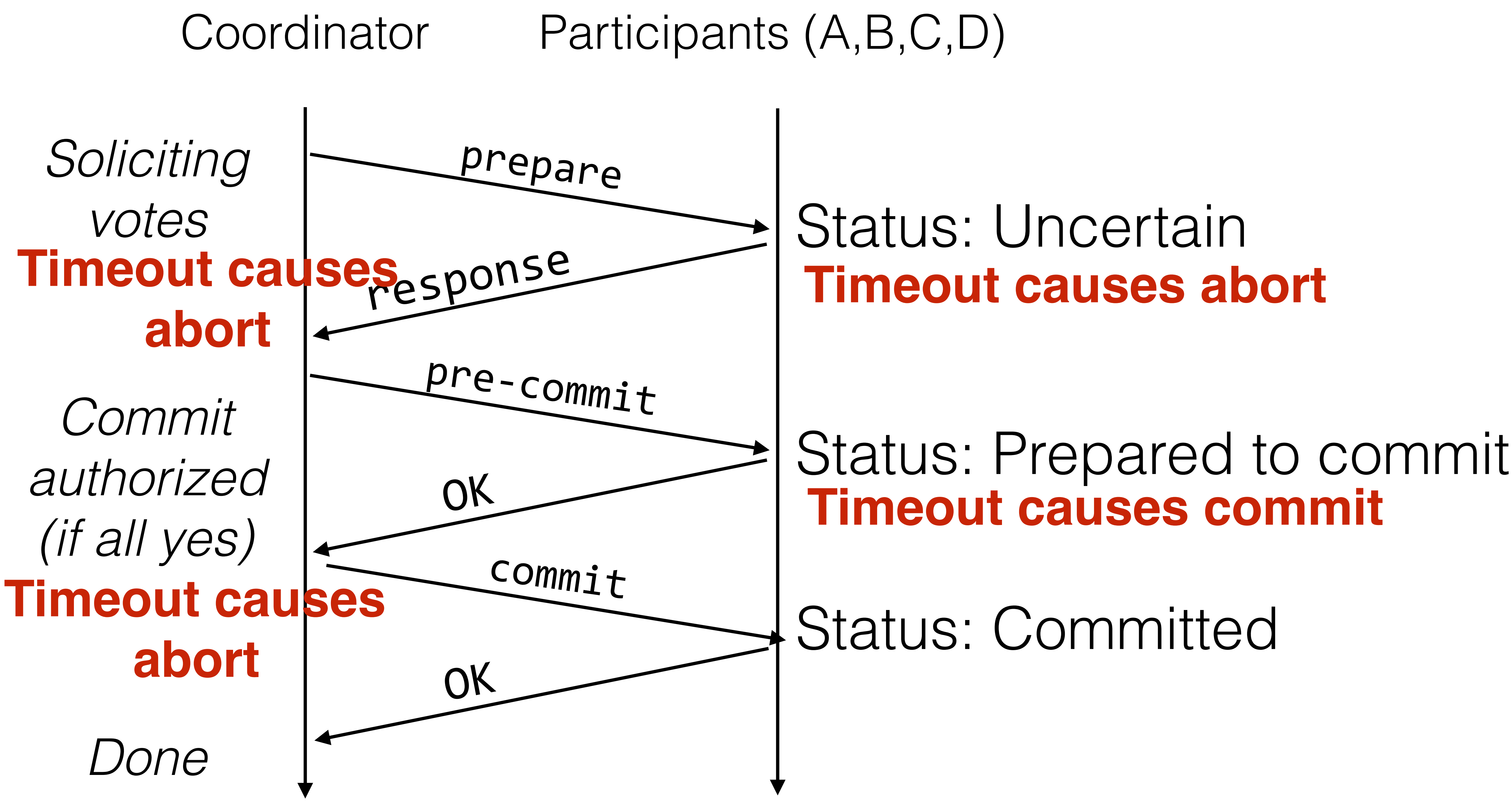


3PC Crash Handling

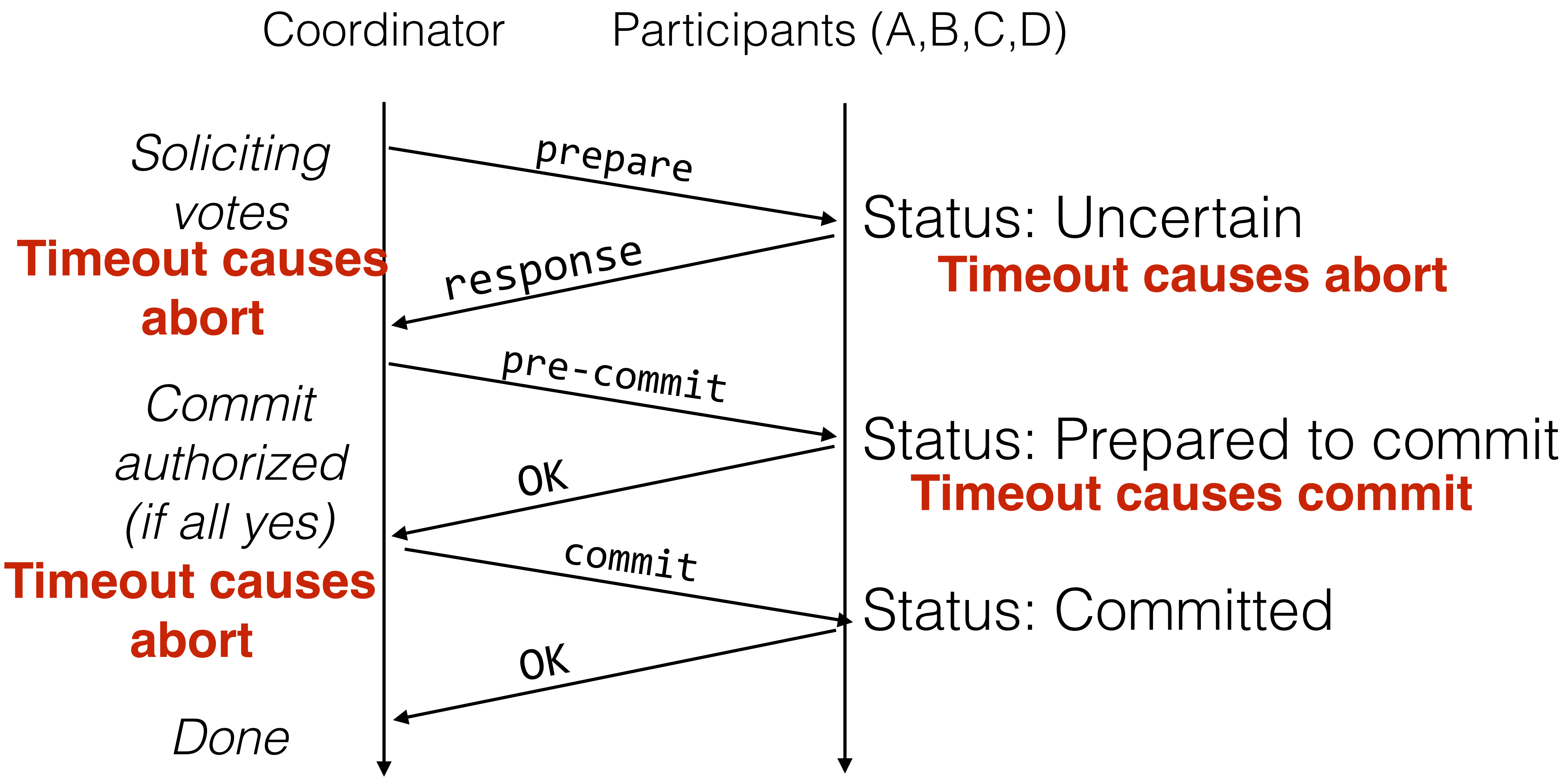
- Can B/C/D reach a safe decision…
  - If NONE of them has received preCommit?
    - YES! It is safe to abort, because A can not have committed (because it couldn't commit until B/C/D receive and acknowledge the pre-commit)
    - This is the big strength of the extra phase over 2PC
- Summary: Any node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

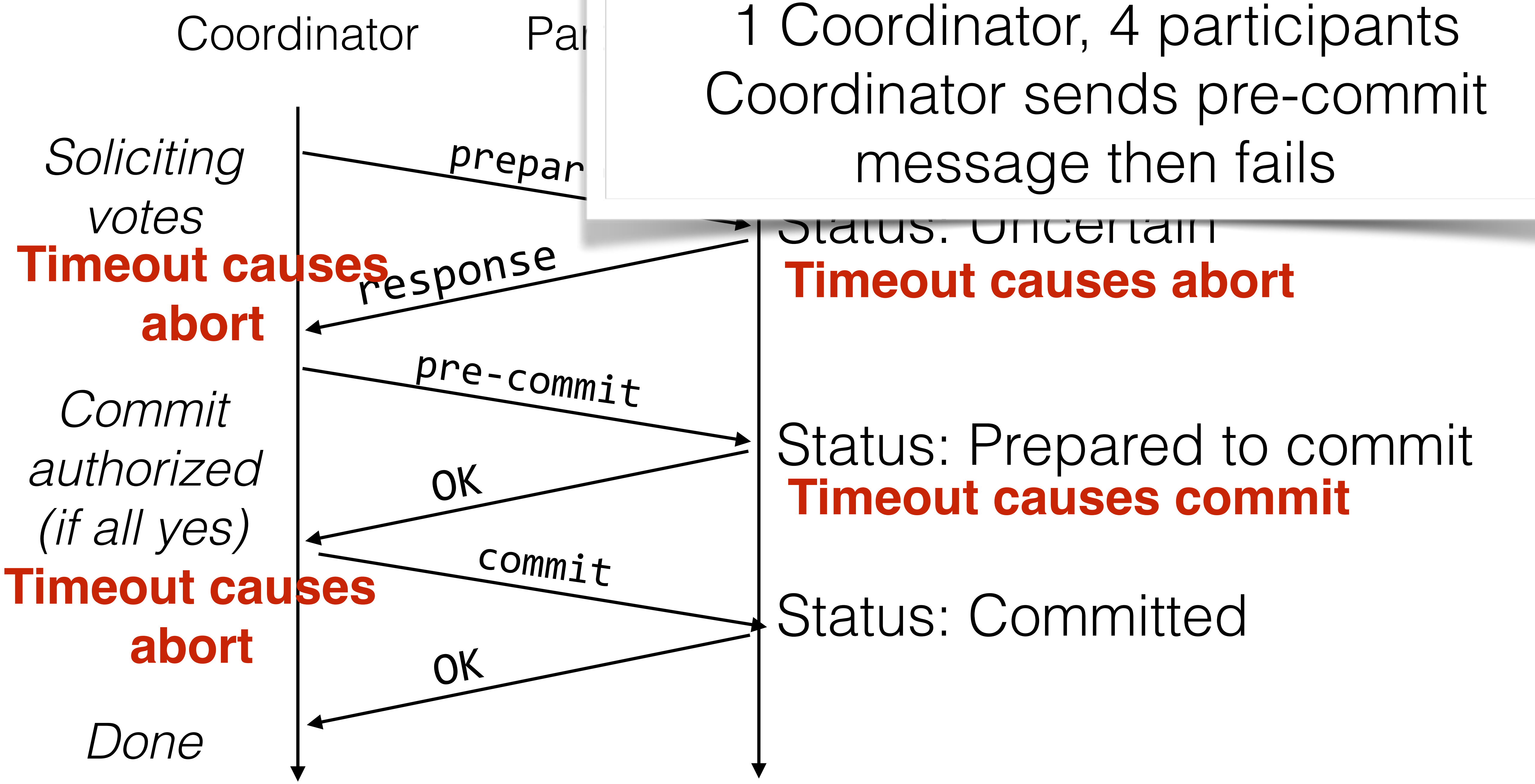


3PC Timeout Handling



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

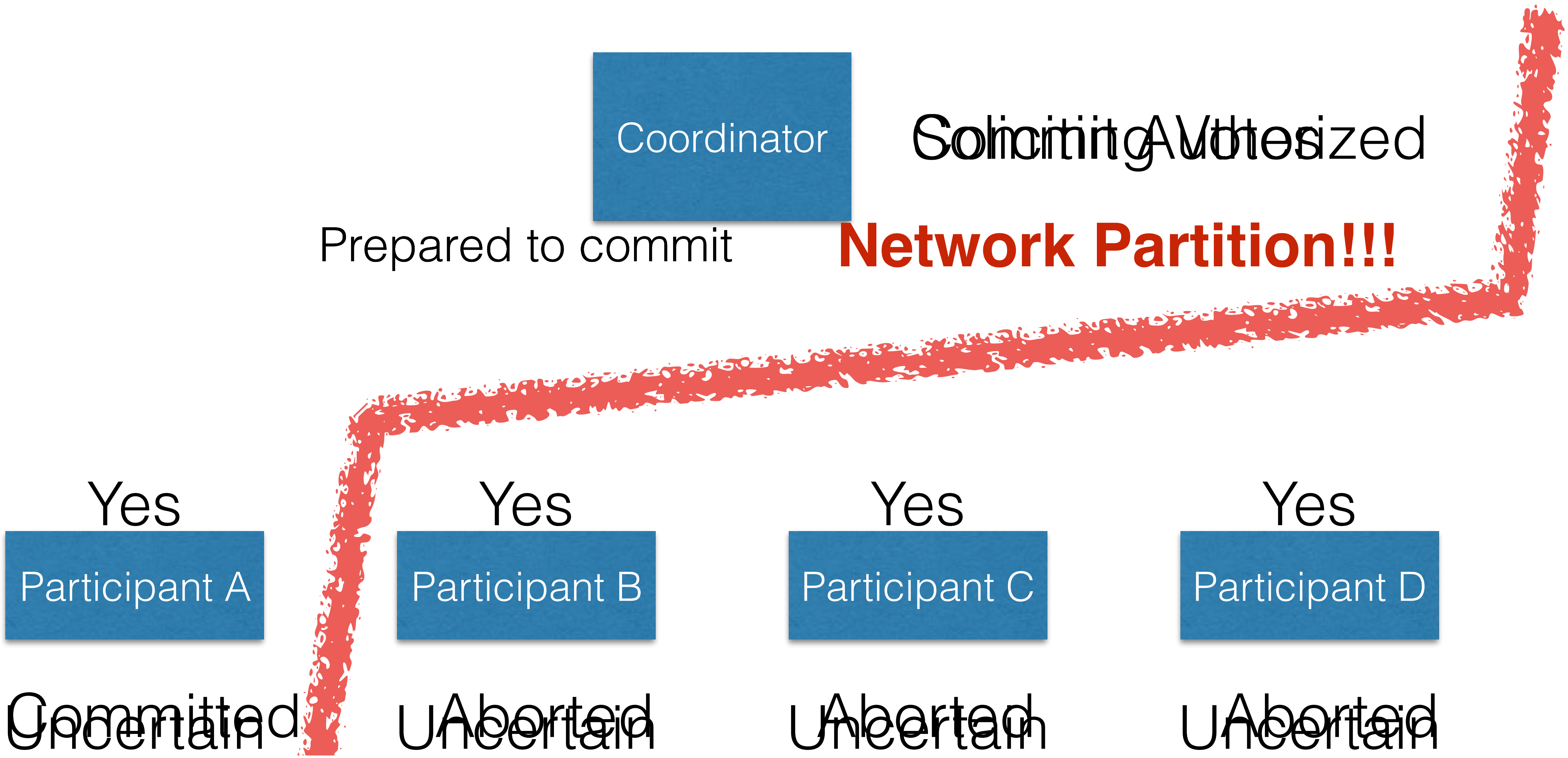
Properties of Agreement

- **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

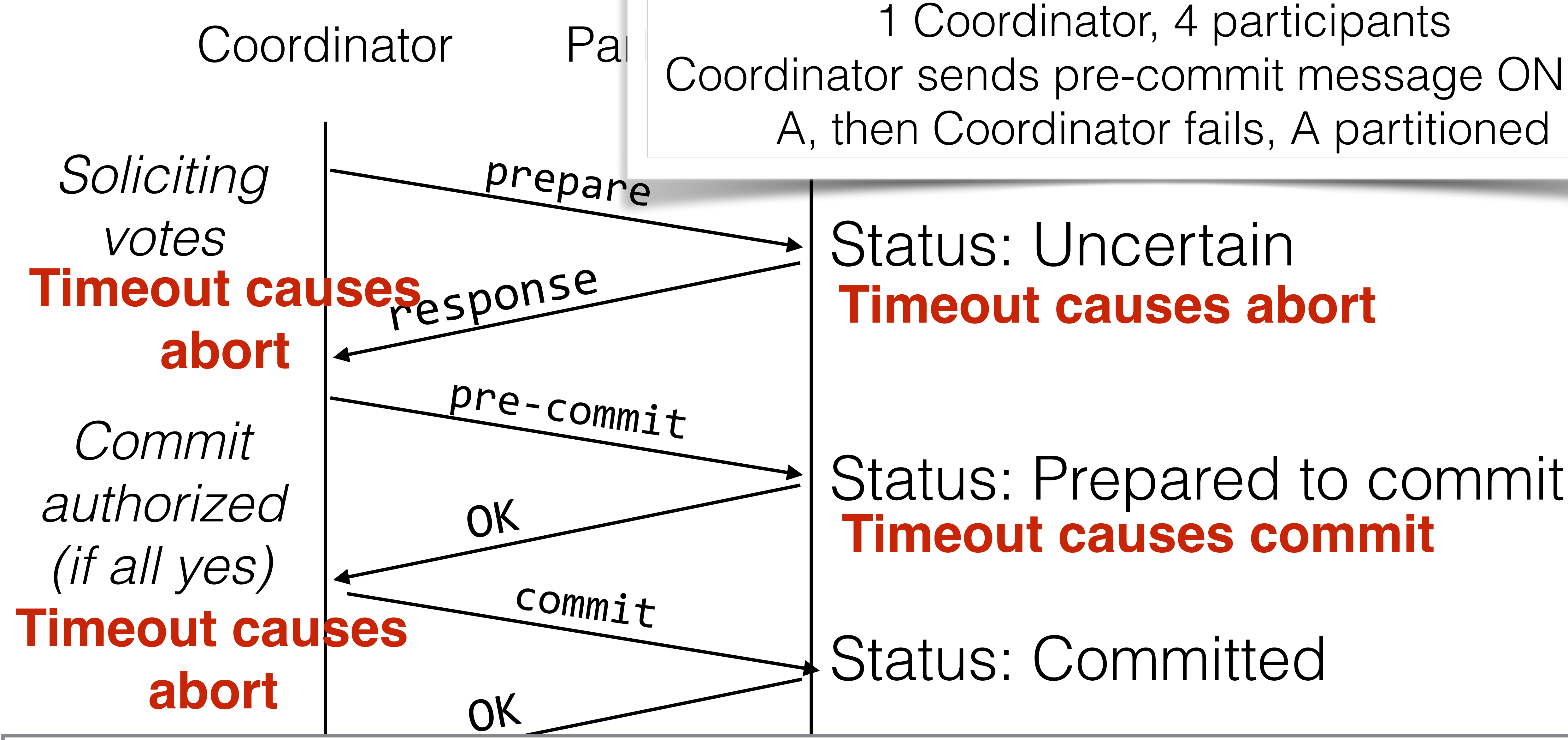


Timeout behavior: Commit!

Timeout behavior: abort

3PC Exercise

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



	Participants			
Coord	A	B	C	D

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?

Participant A	Participant B	Participant C	Participant D
Yes	Yes	Yes	Yes
Committed	Aborted	Aborted	Aborted
Uncertain	Uncertain	Uncertain	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

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.

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