Distributed Transactions: 2 Phase Commit

CS 475, Spring 2019
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;
}
```

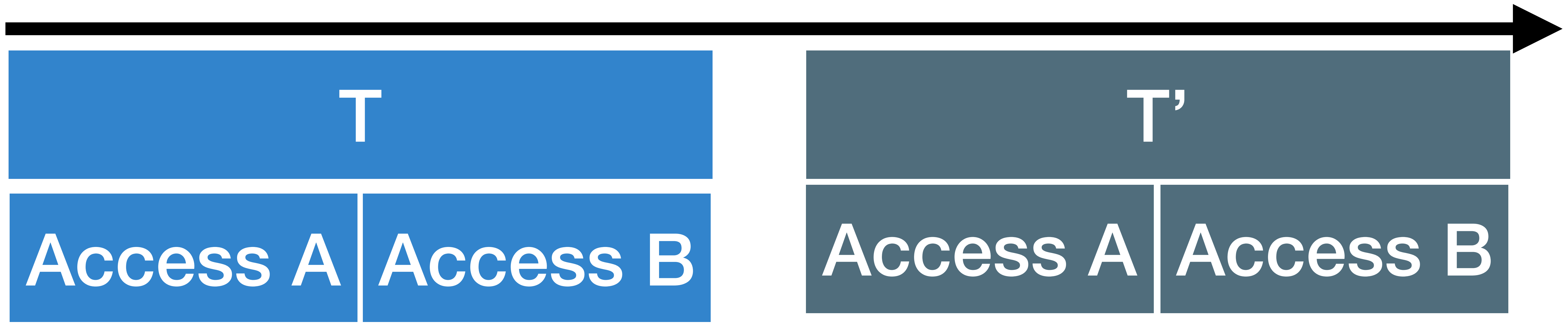
What can go wrong here?

Review: 2-phase locking

- Simple solution for isolation
- Phase 1: acquire locks (all that you might need)
- Phase 2: release locks
  - You can’t get any more locks after you release any
  - Typically: locks released when you say “commit” or “abort”

Review: 2-Phase Locking Ensures Serializability of Transactions

- Allows serializability to be considered at the level of transactions, which might include multiple variables
- If a transaction T accesses variables A and B, and T’ accesses variables A and B, then either:



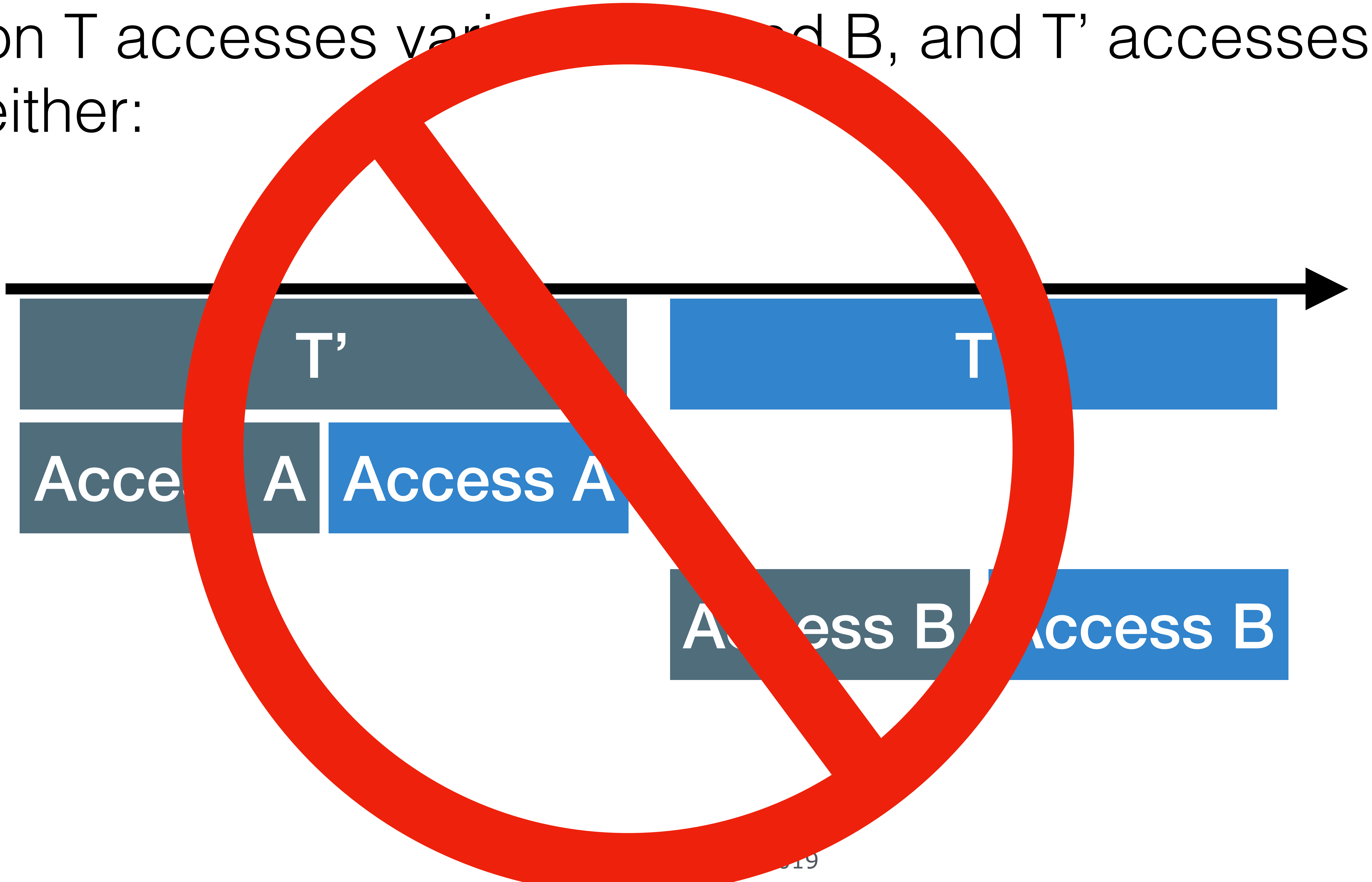
Review: 2-Phase Locking Ensures Serializability of Transactions

- Allows serializability to be considered at the level of transactions, which might include multiple variables
- If a transaction T accesses variables A and B, and T’ accesses variables A and B, then either:

Review: 2-Phase Locking Ensures Serializability of Transactions

Individual variable acesses are sequentially consistent, but transactions are not serializable!

- If a transaction T accesses var… d B, and T’ accesses variables A and B, then either:



Review: Fault Recovery

- How do we recover transaction state if we crash?
- Goal:
  - Committed transactions are not lost
  - Non-committed transactions either continue where they were or aborted
- Plan:
  - Write ahead logging
  - Replay to recover

Today

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

Agreement

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

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

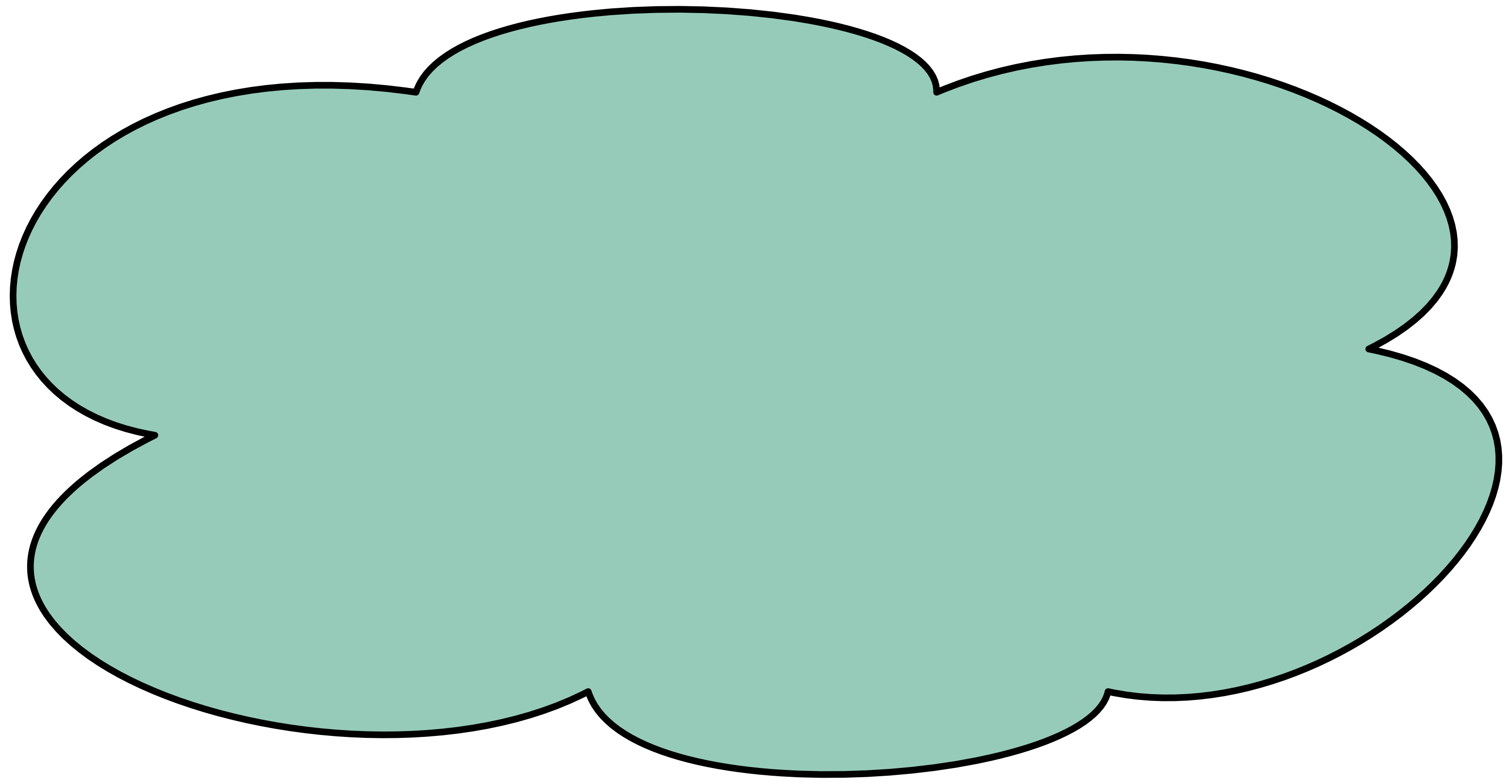
Distributed Transactions

```
transferMoney("from": Barney@Goliath National,
    "to": Mortimer@ Duke&Duke, "amount"=$1)
    Initially: Barney.balance= $10000, Mortimer.balance=$10000
```

Goliath National Bank

transferMoney:

```
add(Mortimer,1)
add(Barney,-1)
```



Duke & Duke Partners

auditRecords:

```
tmp1 = get(Mortimer)
tmp2 = get(Barney)
print tmp1, tmp2
```

What can we hope for if these two actions happen at once?

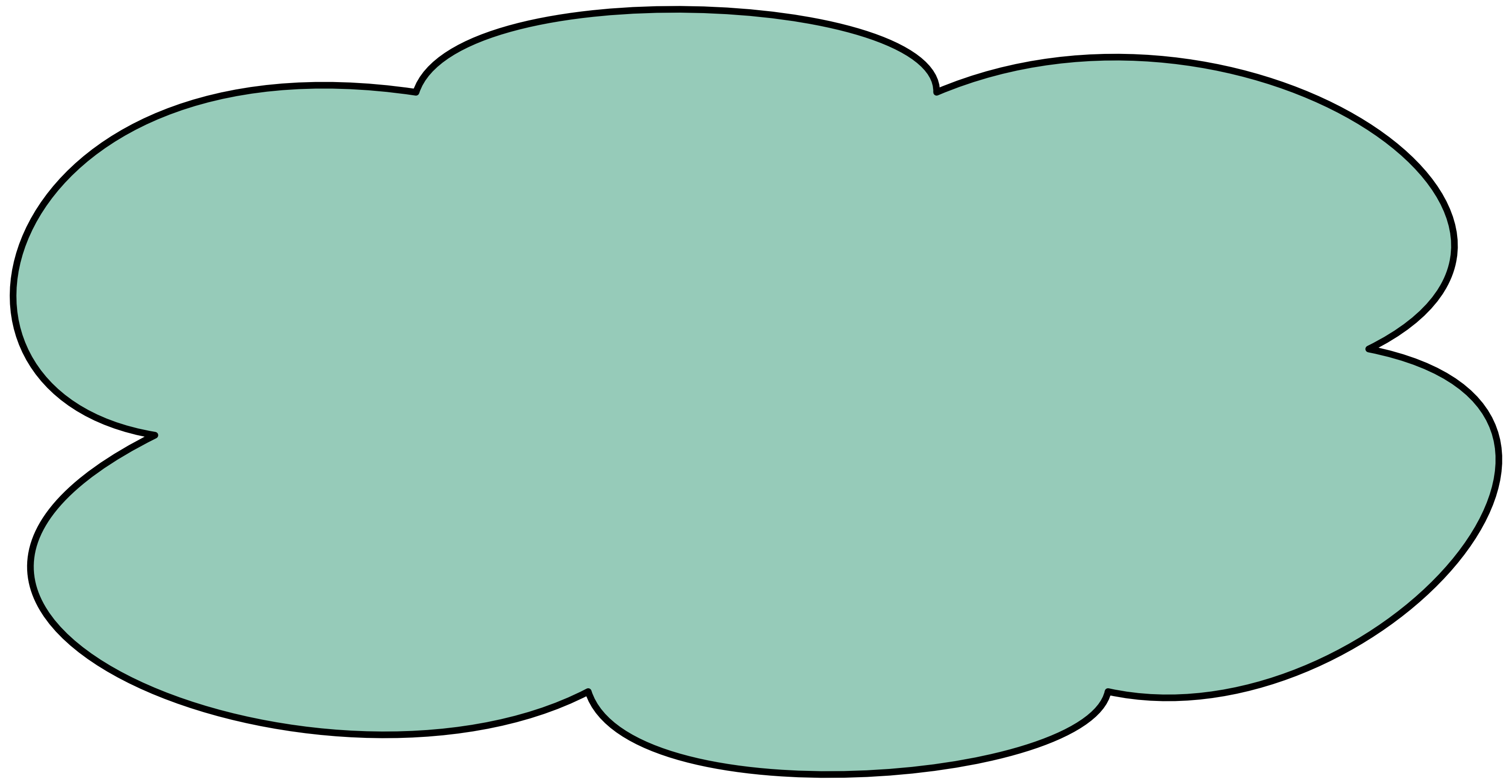
10,000 printed twice, or:
10,001 and 9,999
(Atomicity of the transfer)

Distributed Transactions

```
transferMoney("from": Barney@Goliath National,
    "to": Mortimer@ Duke&Duke, "amount"=$1)
    Initially: Barney.balance= $10000, Mortimer.balance=$10000
```

Goliath National Bank

```
transferMoney:
add(Mortimer,1)
add(Barney,-1)
```



Duke & Duke Partners

```
auditRecords:
tmp1 = get(Mortimer)
tmp2 = get(Barney)
print tmp1, tmp2
```

...But why is this hard? What can go wrong?

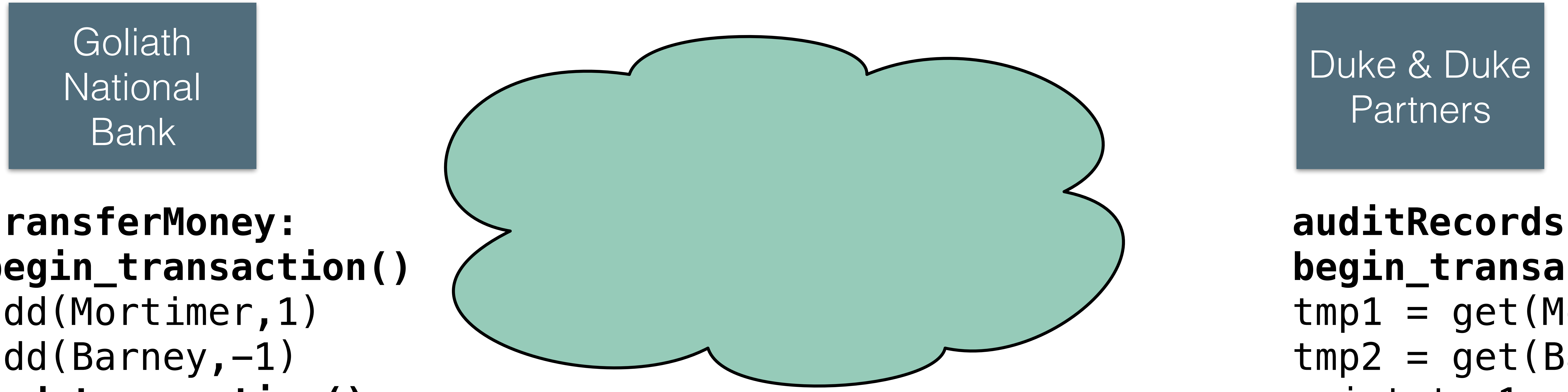
auditRecords is interleaved with transferMoney?

Server or network failure on either end

Mortimer or Barney's account might not even exist

Distributed Transactions

- We can easily solve our transfer problem by making this two transactions!
- Client tells the transaction system when to start/end each transaction
- System arranges transactions to ensure our ACID properties
- Today's focus: how do we build that transaction system?




```
transferMoney:
begin_transaction()
add(Mortimer,1)
add(Barney,-1)
end_transaction()
```

```
auditRecords:
begin_transaction()
tmp1 = get(Mortimer)
tmp2 = get(Barney)
print tmp1, tmp2
end_transaction()
```

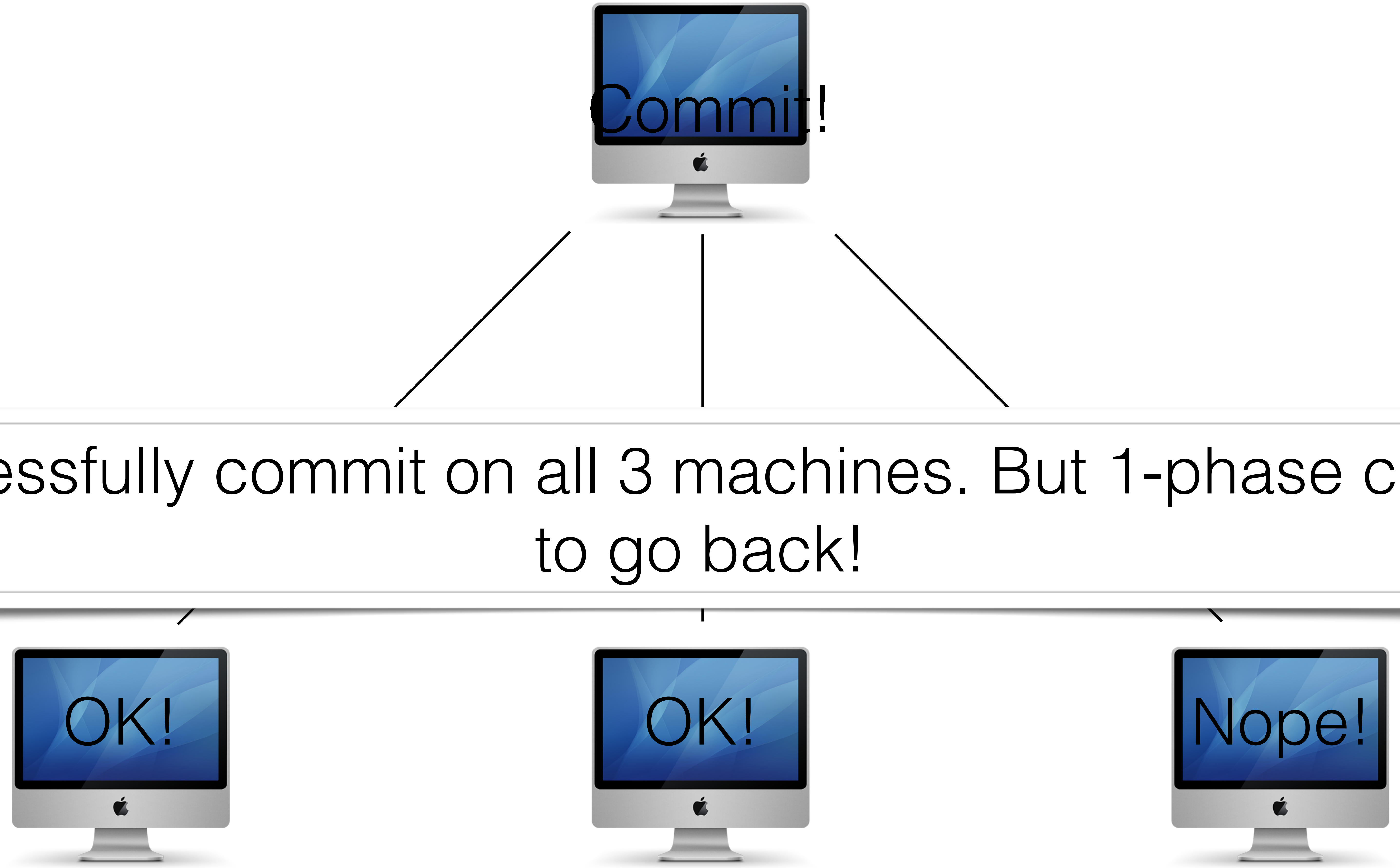
Distributed Transactions

- Will focus much more on how to abort - because more can go wrong:
  - Abort must undo any in-progress modifications
  - Voluntary abort - some client validation fails (e.g. bank account doesn’t exist)
  - Abort might come from failure (server or network crash)
  - System might deadlock and need to abort
- Two big components, just like non-distributed transactions:
  - Concurrency control (2 phase locking, just like non-distributed)
  - Atomic commit

Distributed Transactions

- Coordinator: Begins a transaction
  - Assigns a unique transaction ID
  - Responsible for commit + abort
- Participants: everyone else who has the data used in the transaction
- In principle, any client can be the coordinator, but all participants need to agree on who is the coordinator

1-Phase Commit (no transactions)



We couldn’t successfully commit on all 3 machines. But 1-phase commit has no way to go back!

1-Phase Non-Transaction Commit

- Naive protocol: coordinator broadcasts out “commit!” continuously until participants all say “OK!”
- Problem: what happens when a participant doesn’t want to commit? How do the other participants know that they shouldn’t have really committed and they need to abort?

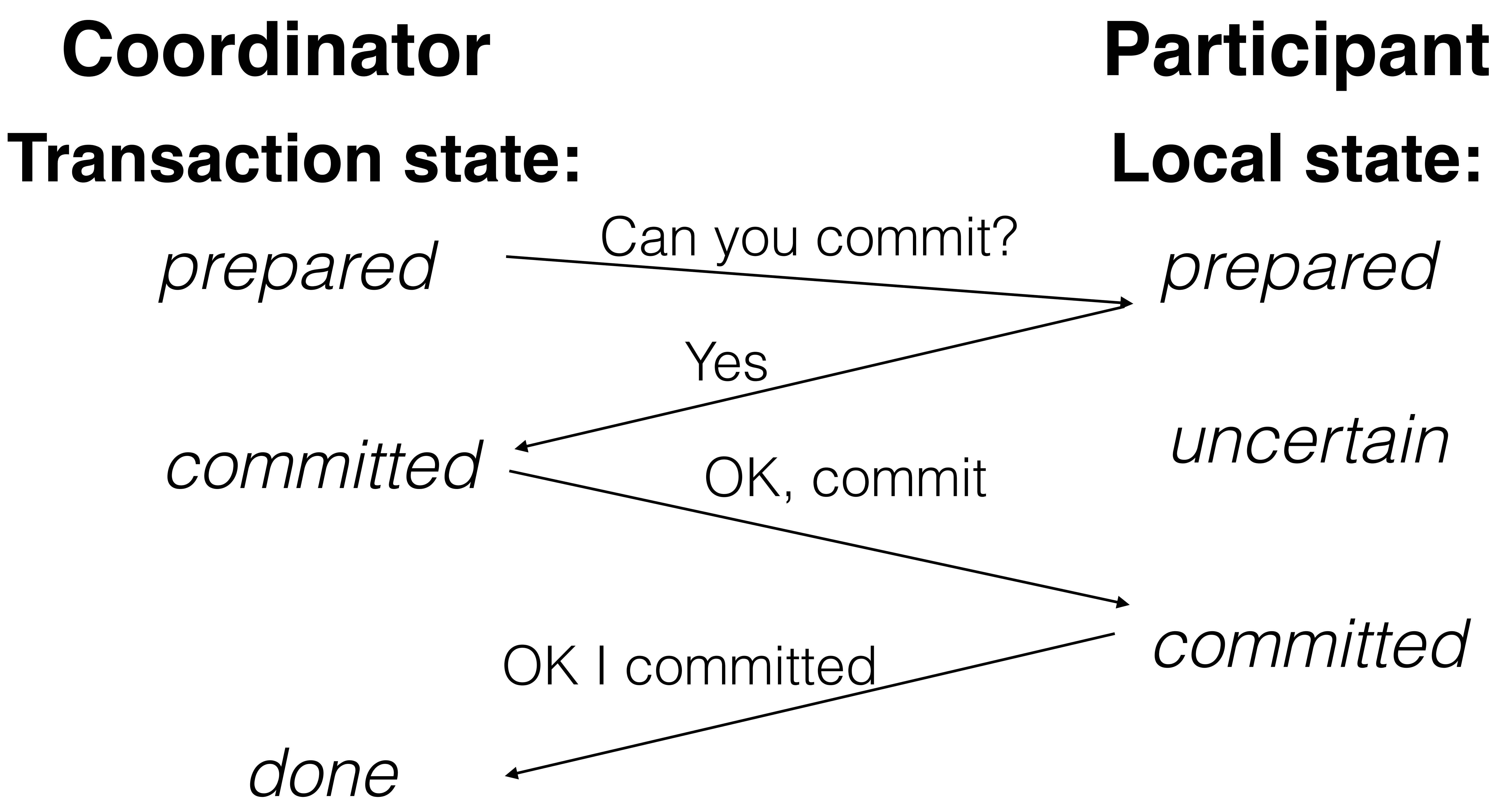
2-Phase Commit

- 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: Voting

- Coordinator asks each participant: can you commit for this transaction?
- Each participant prepares to commit BEFORE answering yes
  - e.g. save transaction to disk for later recovery
  - Can not abort after saying yes
- Outcome of transaction is unknown until the coordinator receives all votes and says “do abort” or “do commit”

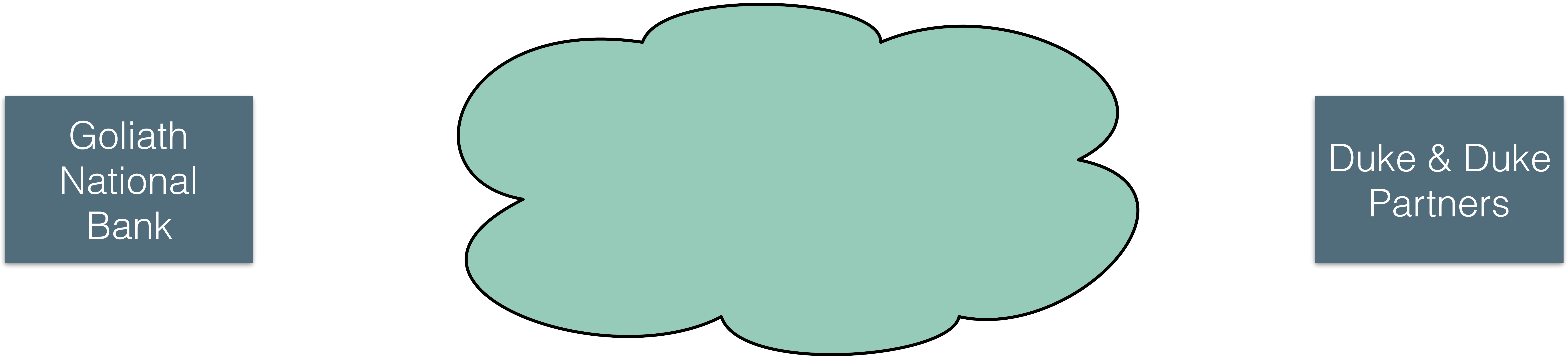
2PC Event Sequence



2PC Example

```
transferMoney(“from”: Barney@Goliath National,
    “to”: Mortimer@ Duke&Duke, “amount”=$1)

Initially: Barney.balance= $10000, Mortimer.balance=$10000
```



Requirements:

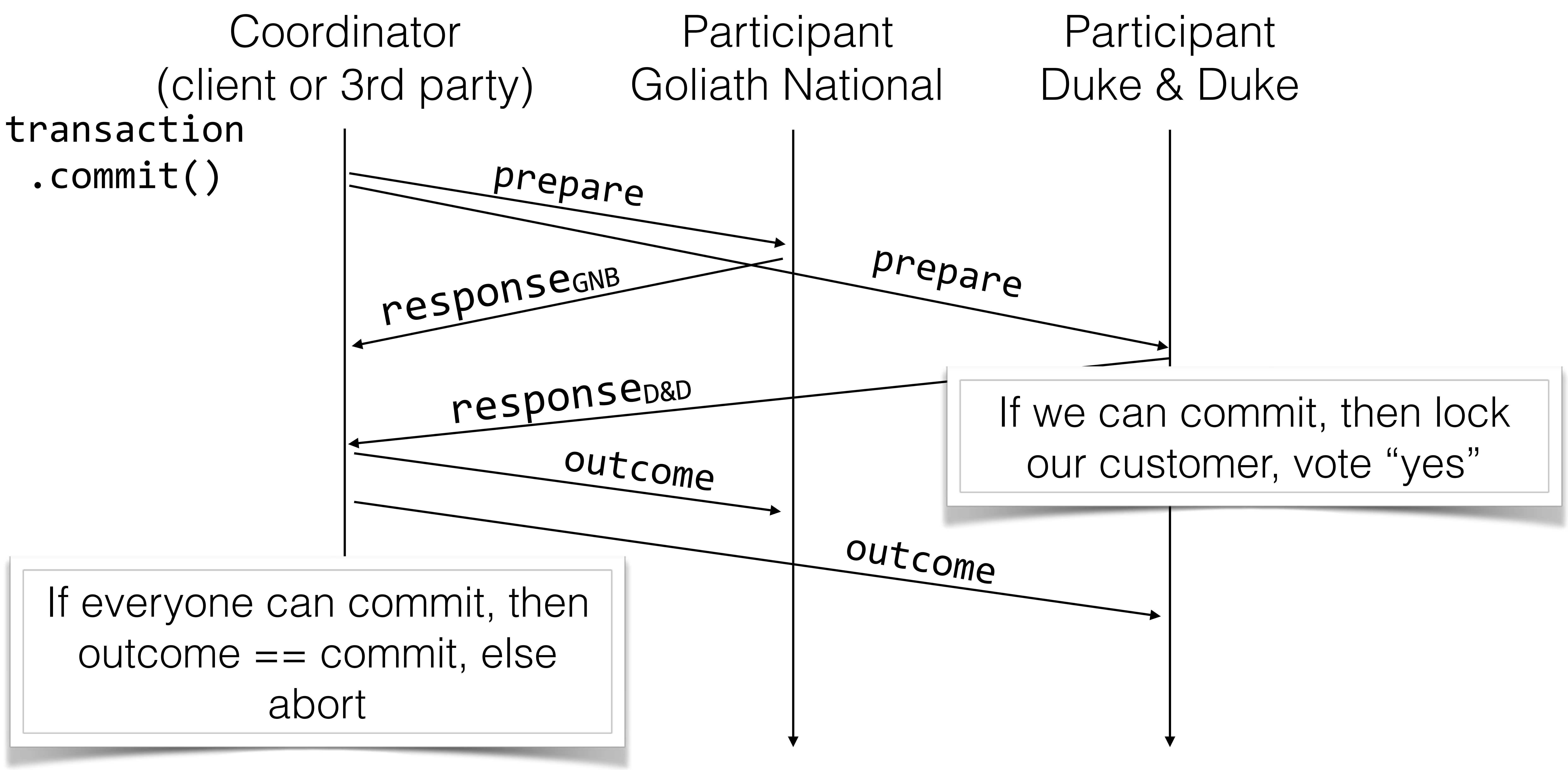
1. Atomicity (transfer happens or doesn’t)
2. Concurrency control (serializability)

2PC Example

For simplicity, let's assume transfer is:

```
int transfer(src, dst, amt) {
    transaction = begin();
    src.bal -= amt;
    dst.bal += amt;
    return transaction.commit();
}
```

2PC Example



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

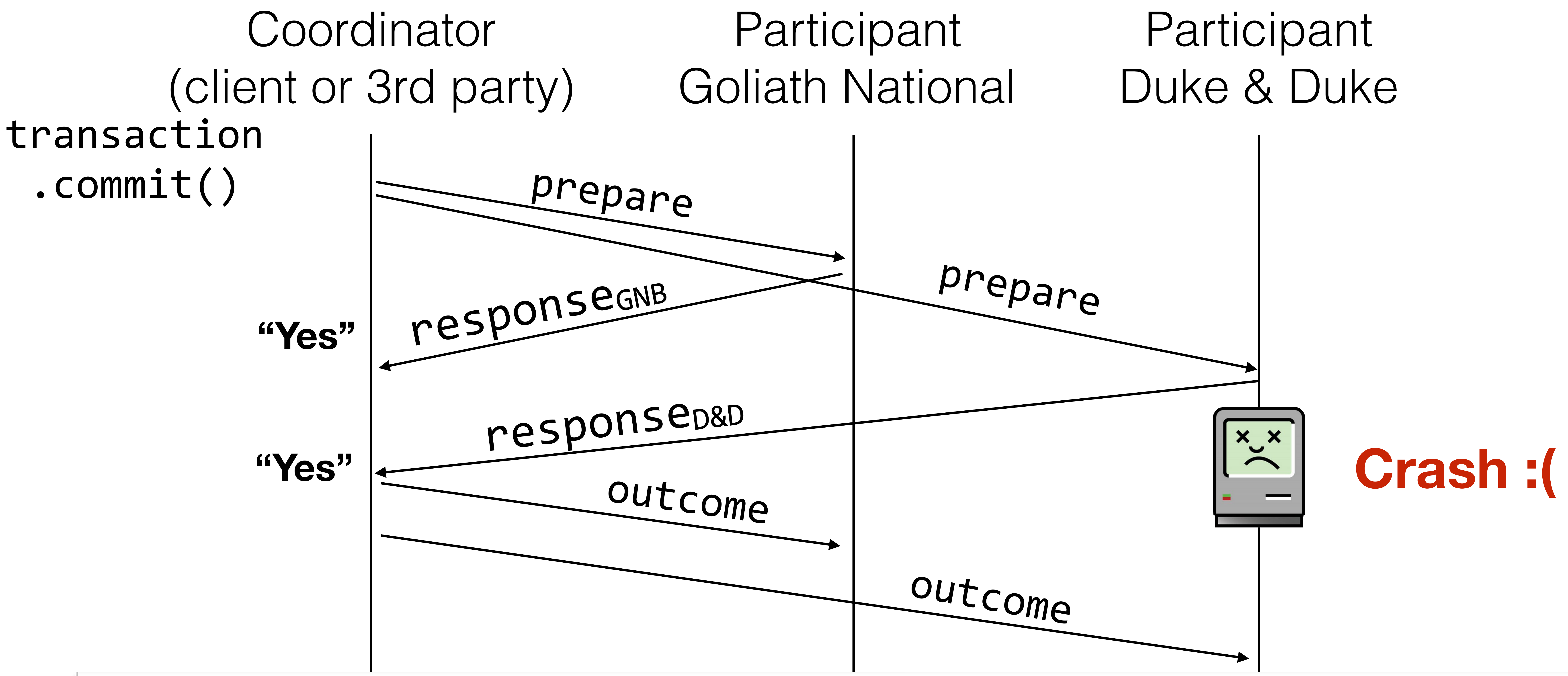
2PC Correctness (Safety)

- Remember the two kinds of properties we want to get:
  - 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
- As presented so far, 2PC guarantees safety, because no participant can proceed with the commit

Fault Recovery

- How do we recover transaction state if we crash?
- Goal:
  - Committed transactions are not lost
  - Non-committed transactions either continue where they were or aborted
- First: lay out various failure modes and discuss intuitions for solutions
  - Crashes for participant and coordinator; timeouts for same
- Then: formalize a policy for recovery in 2PC

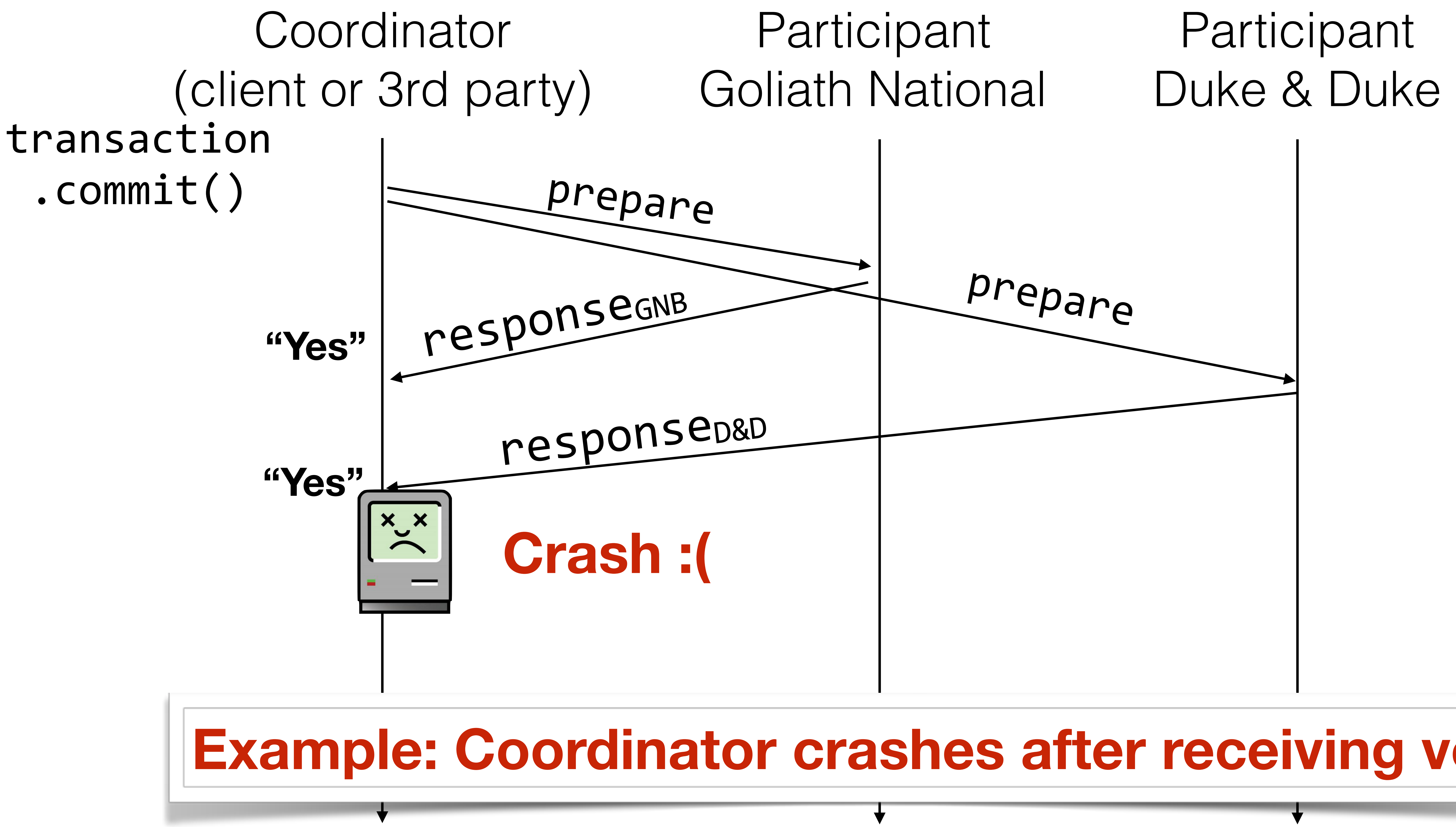
Fault Recovery Example



Example: Participant crashes after voting "yes" to commit

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

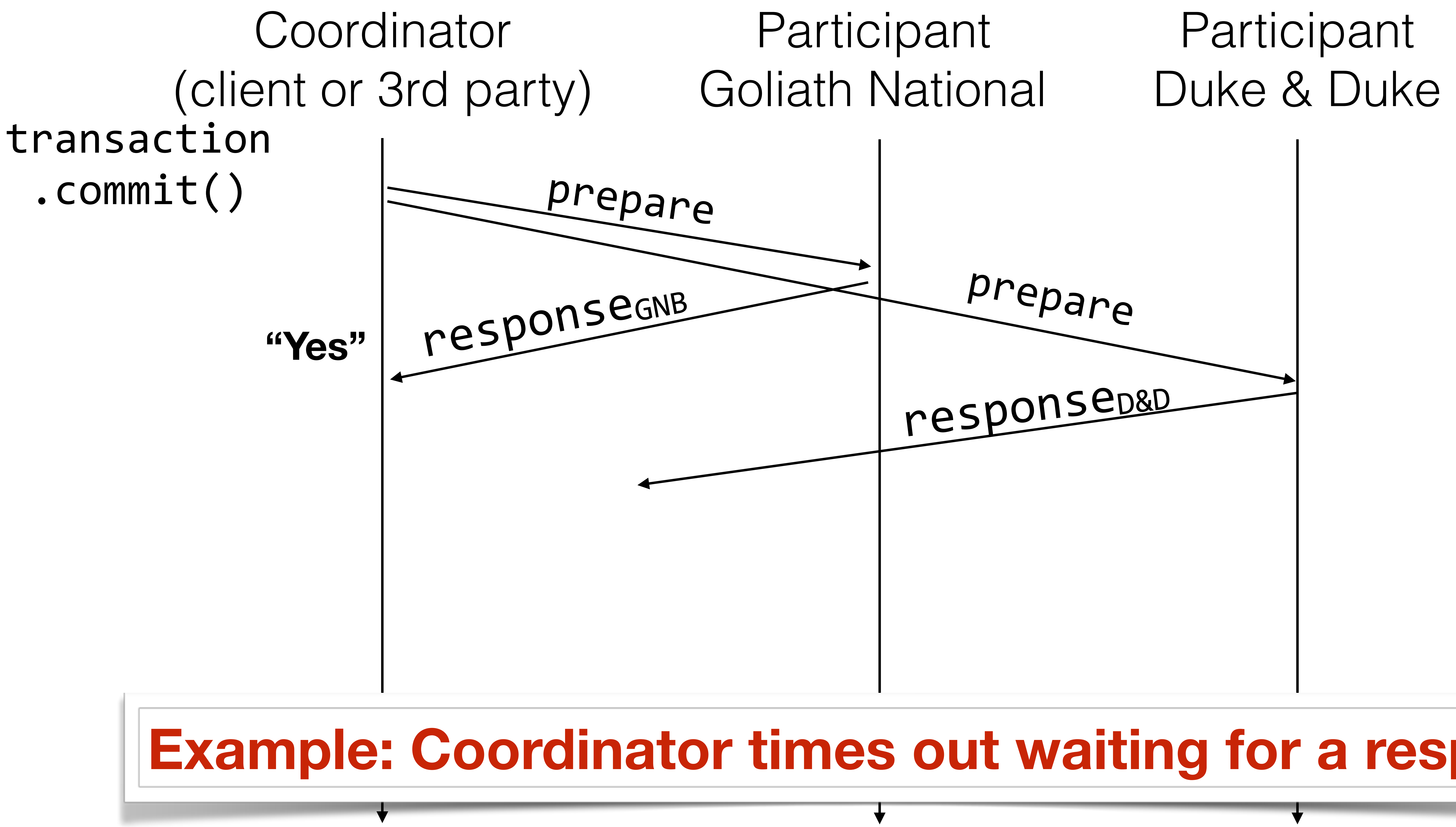
Fault Recovery Example



Example: Coordinator crashes after receiving votes

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

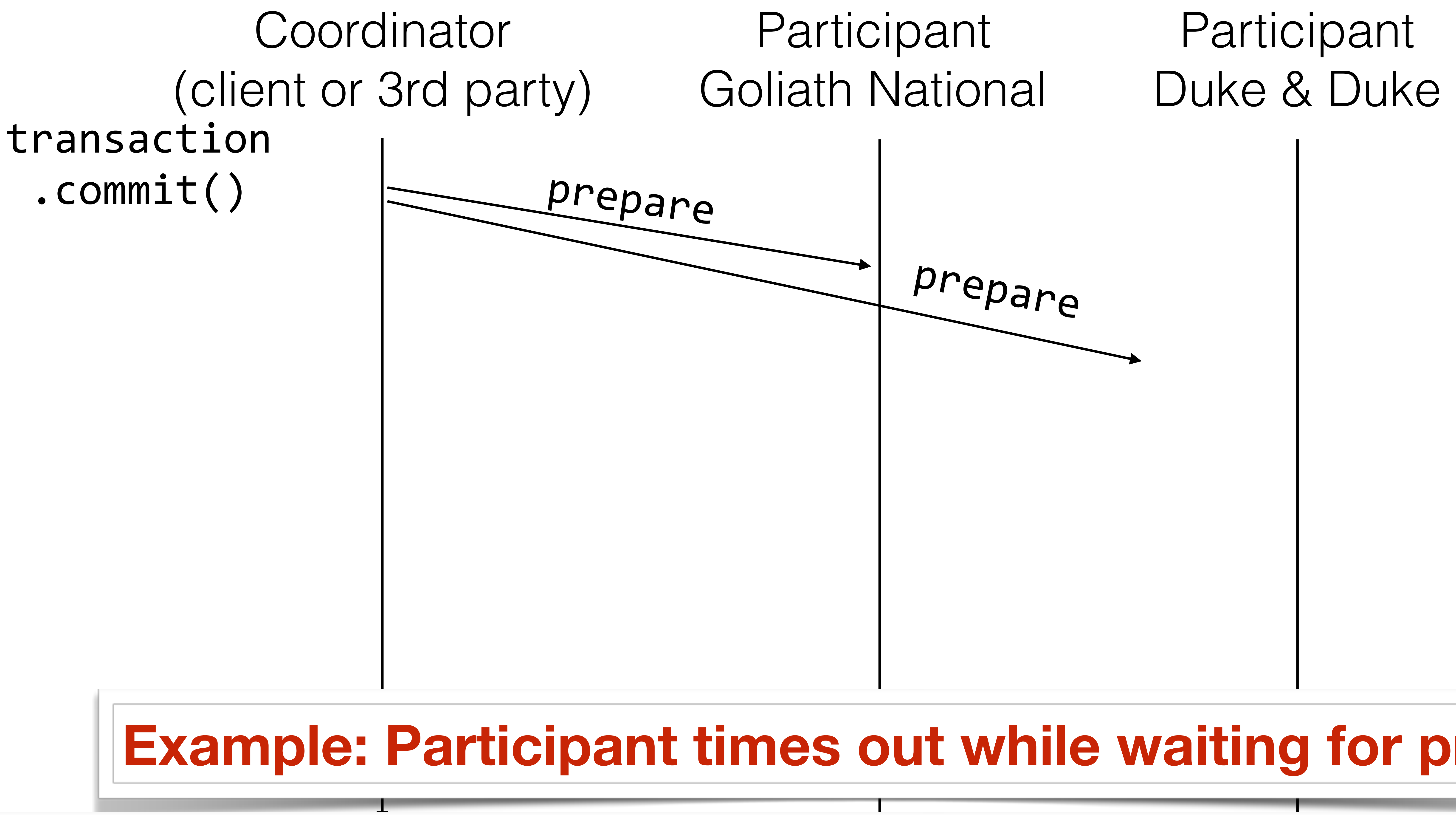
Fault Recovery Example



Example: Coordinator times out waiting for a response

Solution: Coordinator can default to "abort" on timeout

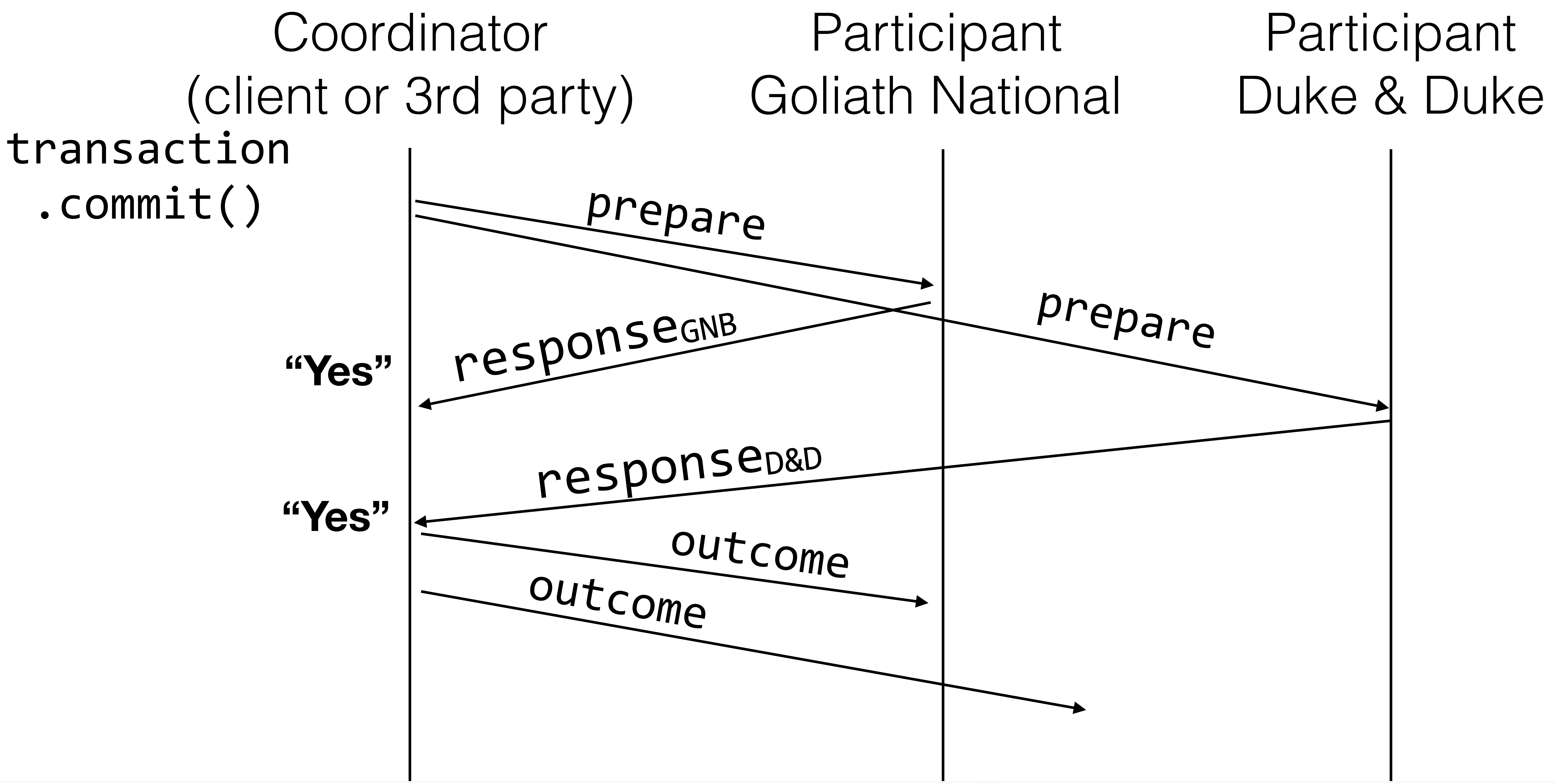
Fault Recovery Example



Example: Participant times out while waiting for prepare

Solution: If we never got prepare message, we never voted "yes," so NOBODY else could commit -> safe to abort

Fault Recovery Example



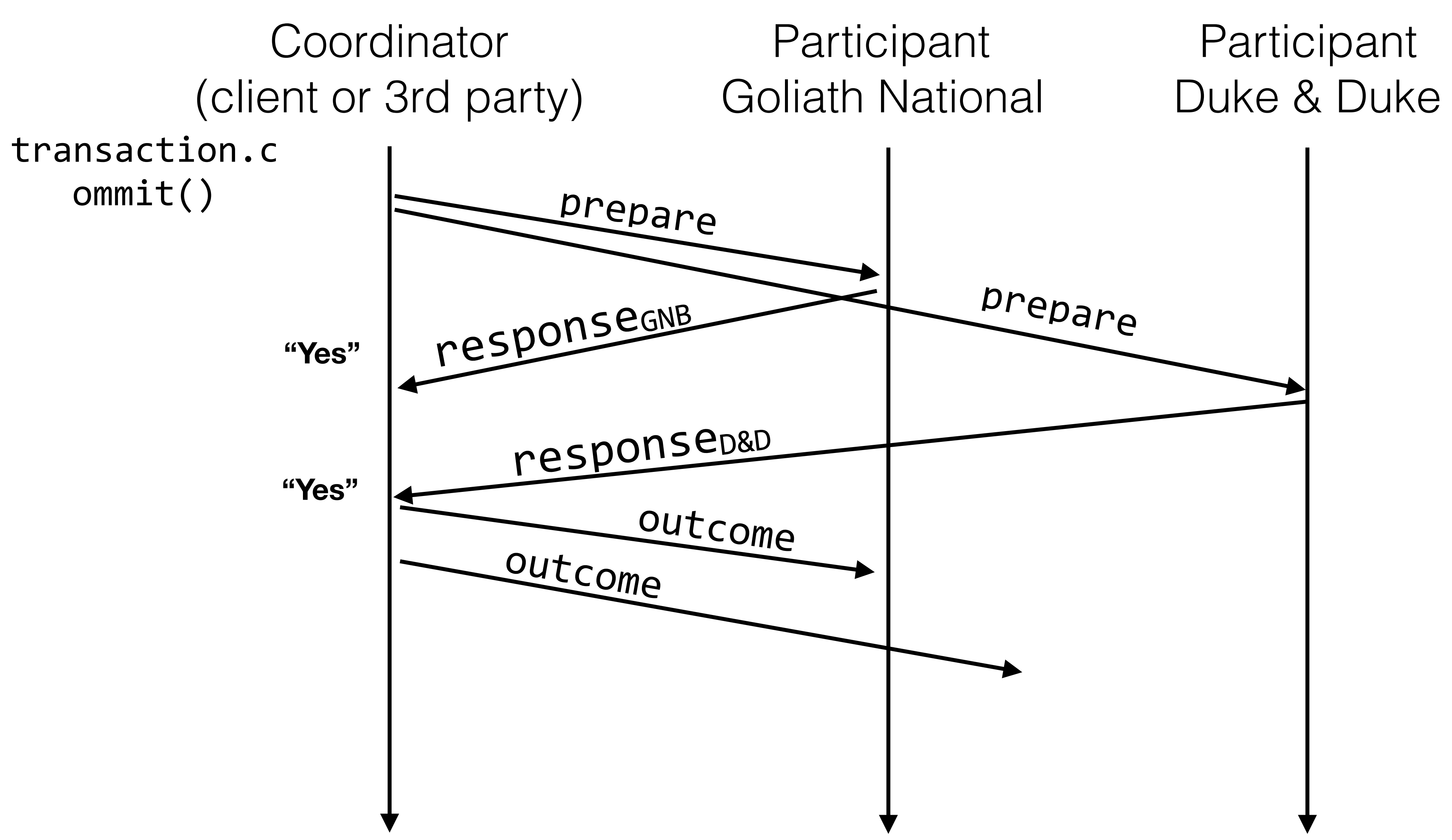
Example: Participant times out while waiting to hear the outcome

Problem: Can the participant unilaterally determine the outcome?

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!

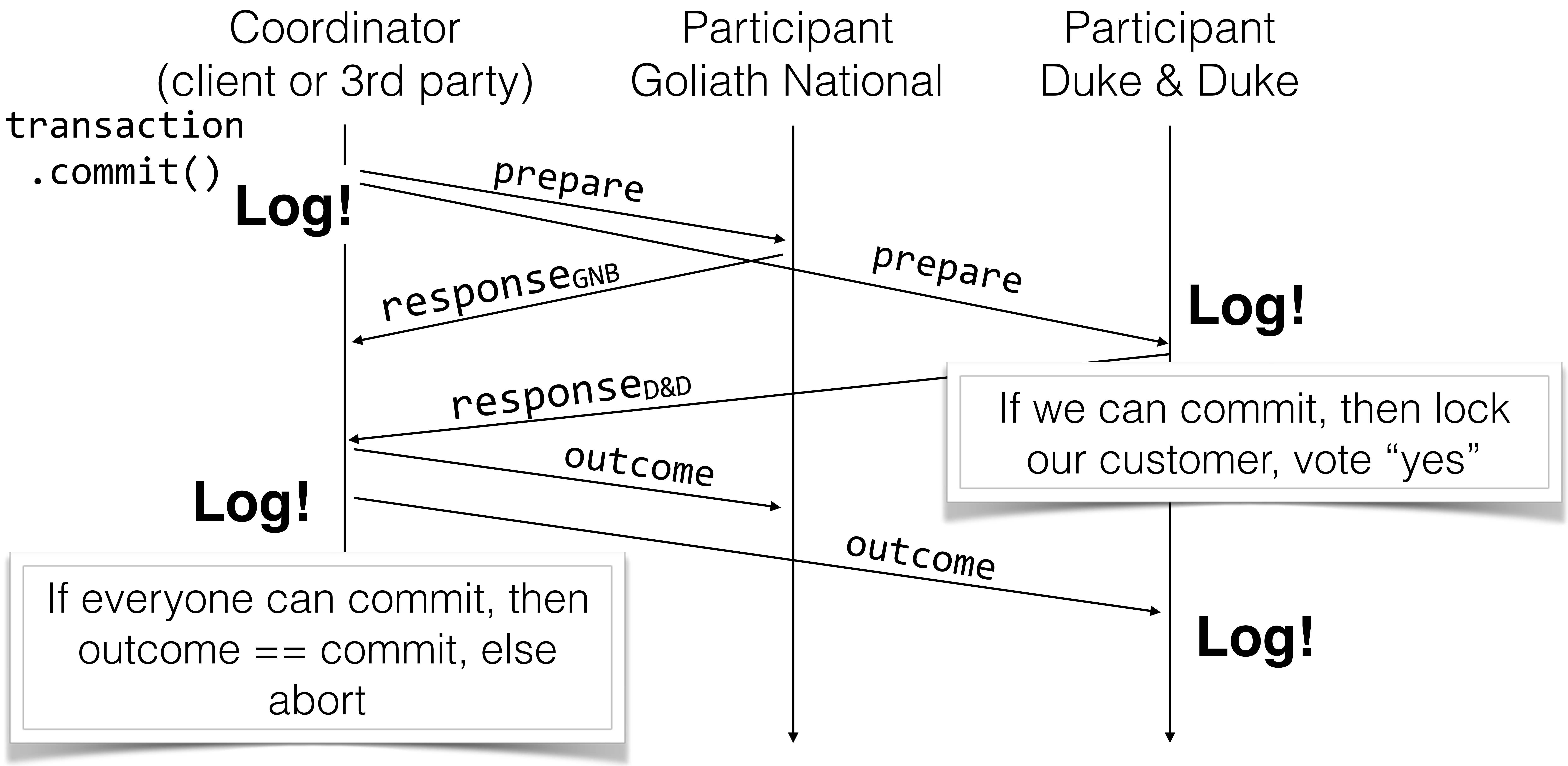
Recovery in 2PC

- What to log?
  - State changes in protocol
  - Participants: prepared; uncertain; committed/aborted
  - Coordinator: prepared; committed/aborted; done
  - These messages are idempotent - can be repeated
- Recovery depends on failure
  - Crash + reboot + recover
  - Timeout + recover

Crash + Reboot Recovery

- Nodes can’t back out once commit is decided
- If coordinator crashes just AFTER deciding “commit”
  - Must remember this decision, replay
- If participant crashes after saying “yes, commit”
  - Must remember this decision, replay
- Hence, all nodes need to log their progress in the protocol

2PC Example with logging



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

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
- We'll come back to this "discuss amongst yourselves" kind of transactions next week

2PC Summary

- Guarantees safety, but not liveness - there are situations in which the protocol can stall indefinitely
- Recovery requires considerable logging
- Relatively few messages required though, for each transaction (low latency)