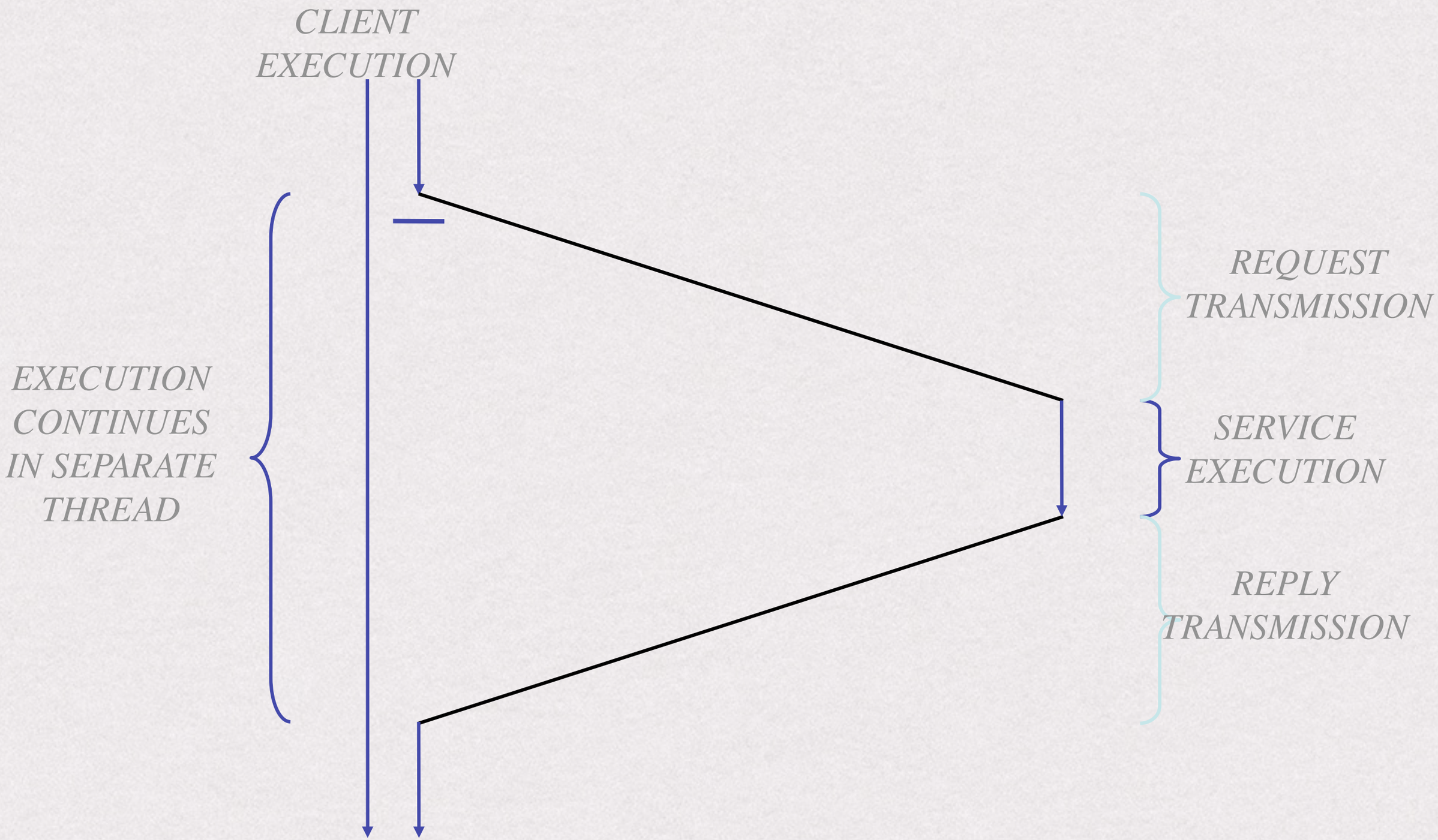
DISTRIBUTED SYSTEMS

FAULT TOLERANCE, 2PC, 3PC, PAXOS

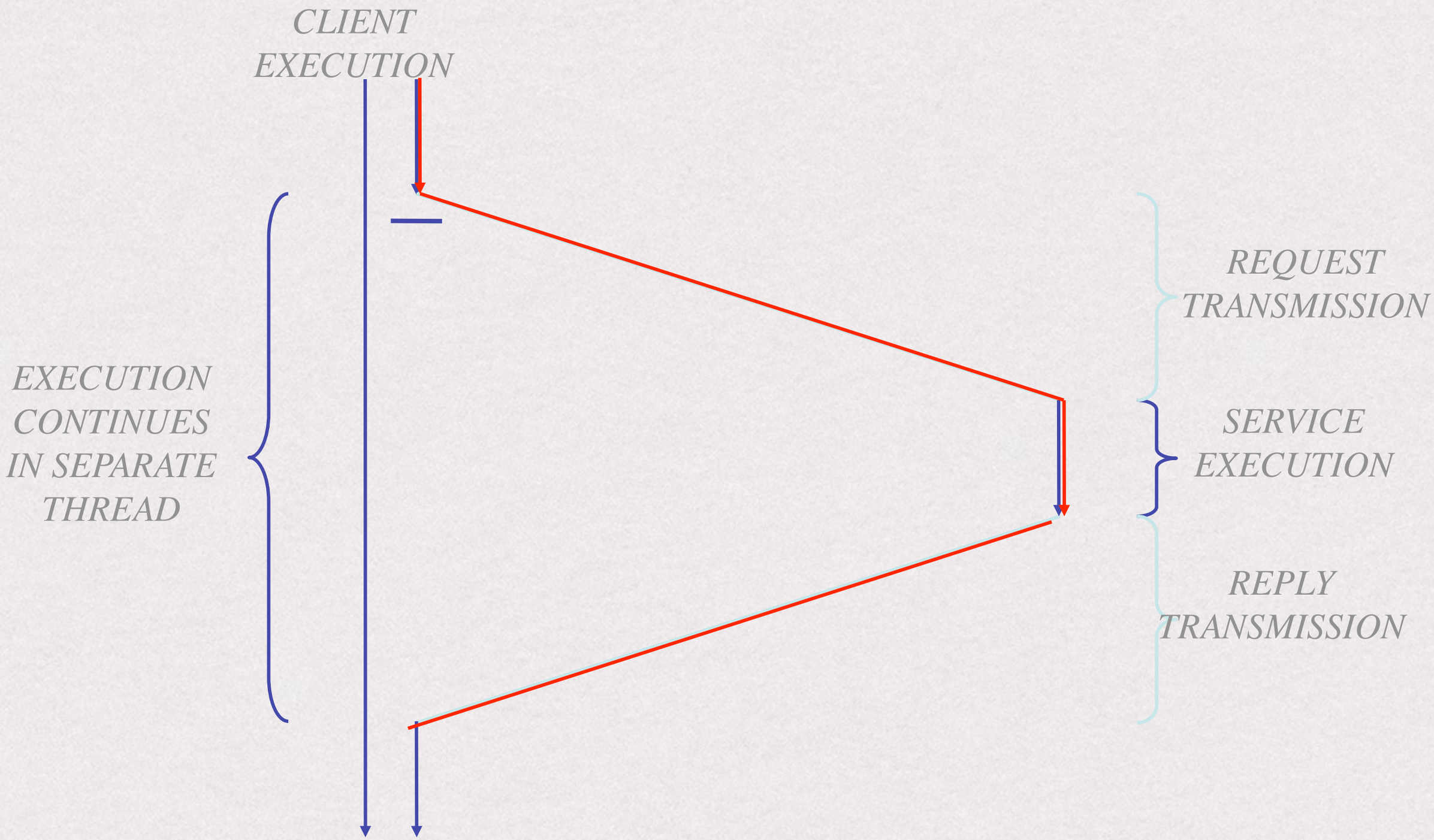
Remote Calls



Questions

- What can fail between the client starting an RPC and the end of that RPC?

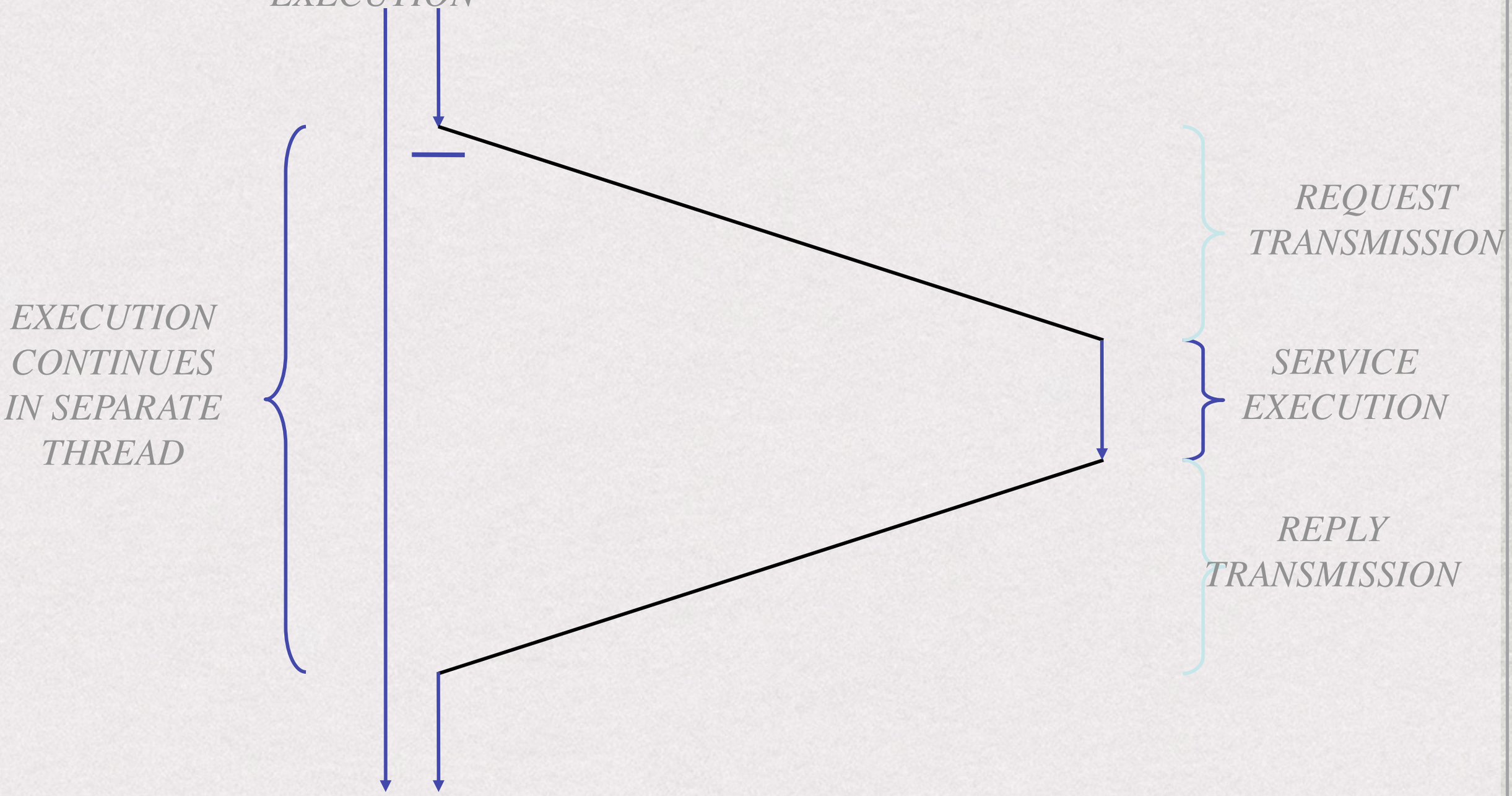
Remote Calls



Questions

- ***What can fail between the client starting an RPC and the end of that RPC?***
- ***What happens to the client when each sort of failure occurs?***

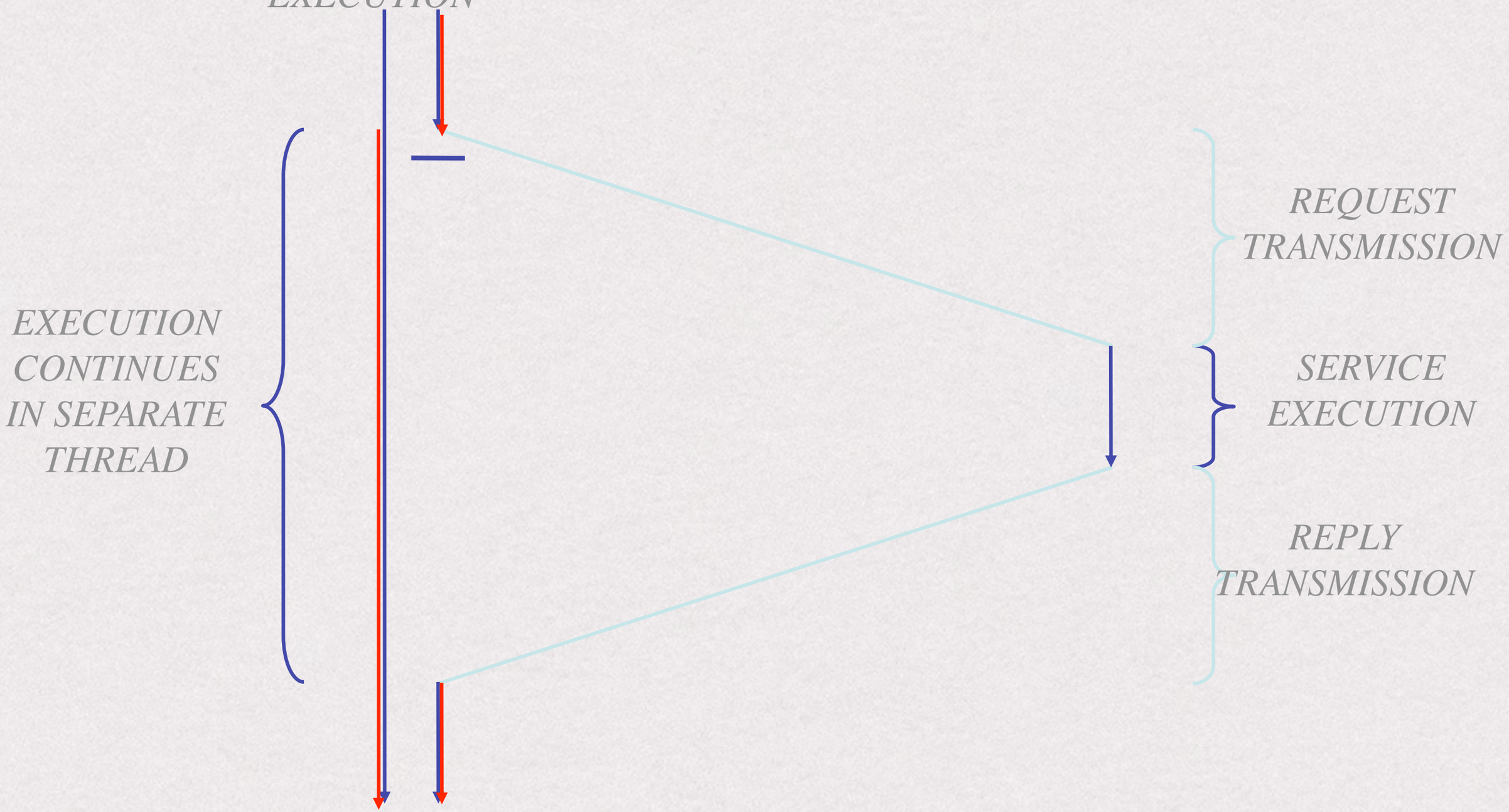
Remote Calls



Questions

- **What can fail between the client starting an RPC and the end of that RPC?**
- **What happens to the client when each sort of failure occurs?**
- **What happens to the server when each sort of failure occurs?**

Remote Calls



Questions

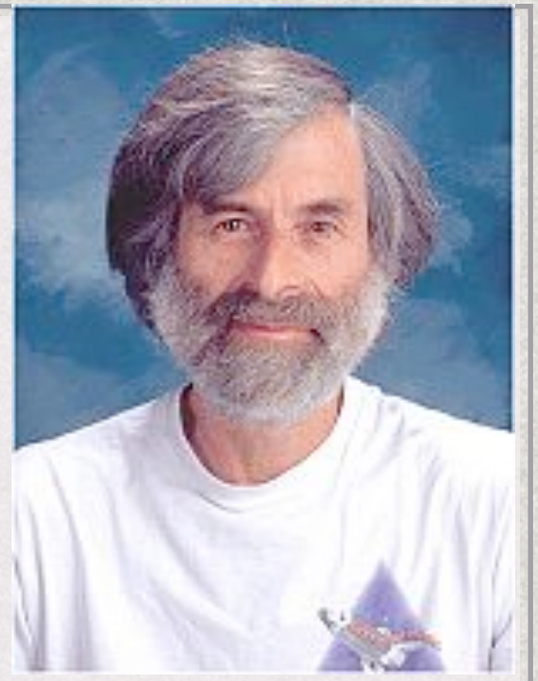
- What can fail between the client starting an RPC and the end of that RPC?
- What happens to the client when each sort of failure occurs?
- What happens to the server when each sort of failure occurs?
- What happens to the distributed system as a whole in the presence of failures?

Questions

- What can fail between the client starting an RPC and the end of that RPC?
- What happens to the client when each sort of failure occurs?
- What happens to the server when each sort of failure occurs?
- What happens to the distributed system as a whole in the presence of failures?
- Or more positively:

How can we build distributed systems that do useful work in spite of the propensity to failure?

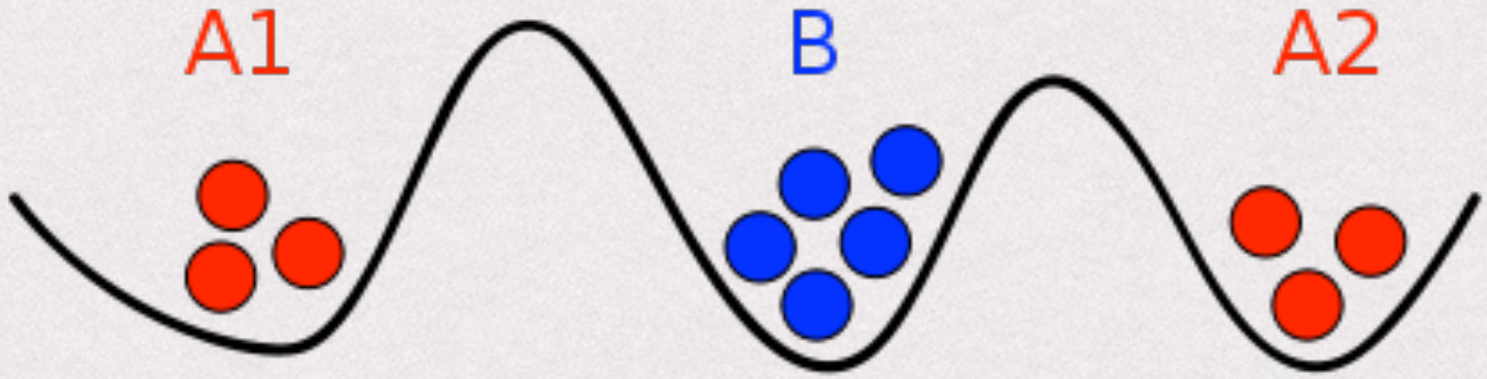
Leslie Lamport



- Seminal work in distributed systems
  - Time, clocks and the order of events
  - The byzantine general's problem
    - Reaching agreement in the presence of faults
  - Mutual exclusion using the bakery algorithm
  - Snapshot algorithms for determining consistent global states
- The initial developer of LaTeX

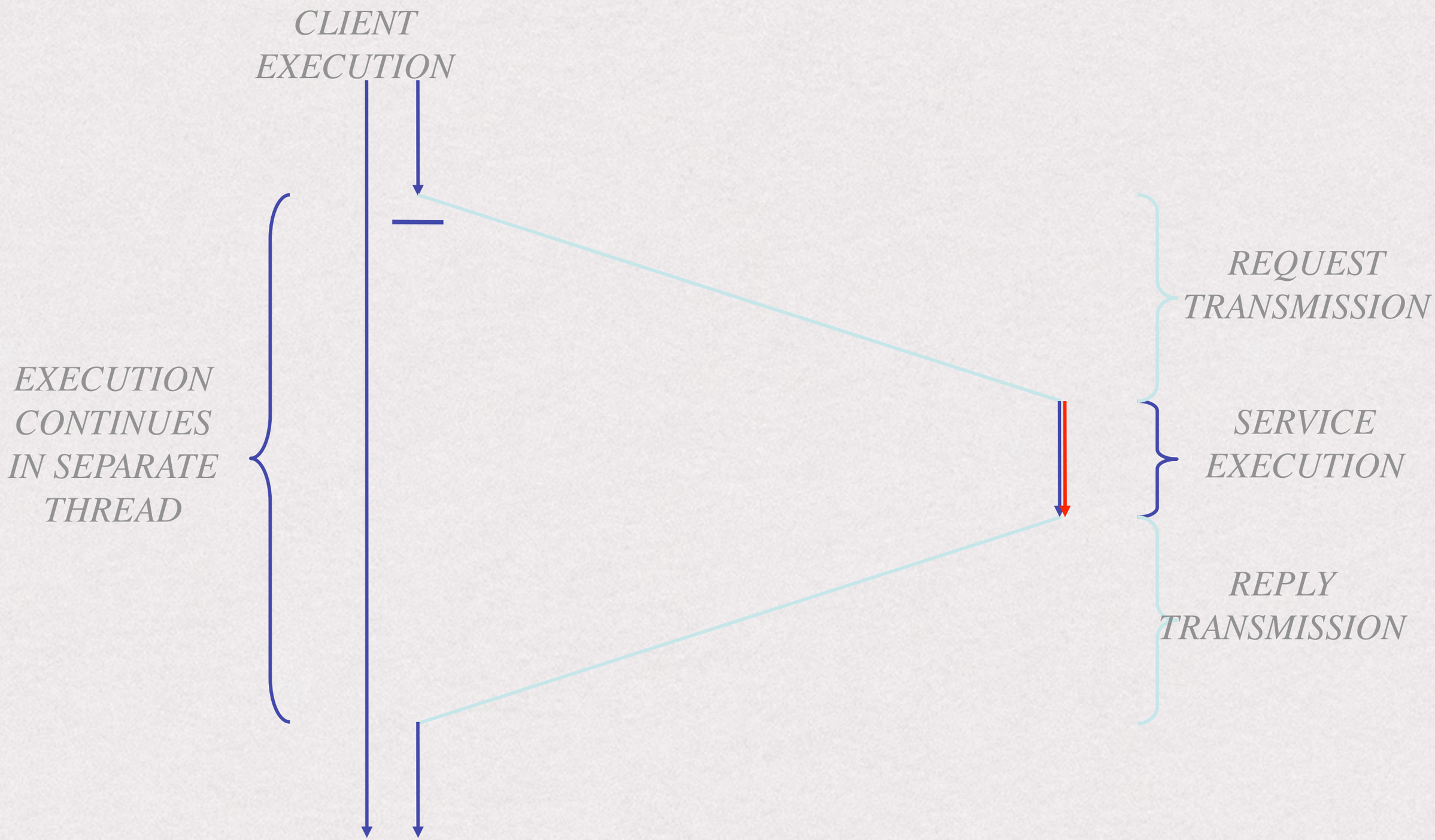
Model of Failure

- The network may lose both request and reply messages.



- Servers may crash, possibly whilst RPCs are in progress, and may then subsequently restart and resume receiving requests
- Clients may crash whilst RPCs are in progress, and may then subsequently restart and resume receiving replies.
- Messages are delivered in the order sent:
  - There are no message re-ordering failures

Remote Calls



Server Failure

- Server crashes and the loss of requests and replies by the network creates a problem for the client.
- The client will use a ***FAILURE DETECTOR*** to discern when such a problem occurs:
  - In principle a failure detector will detect when failures occur.
  - In practice, failure detectors operate by failing to determine that an operation has succeeded within a given time!

Failure Detectors

- A failure detector returns the following:
  - It has not been possible to determine that an operation X has succeeded within time T.
- this is interpreted as:
  - The operation X has failed.
- The operation may have failed, or it may be that the time T is too short:
  - What can one do about this “false failure” result?

Use of Failure Detectors

- Prior to sending a request message, the client sets a timer to expire at some point in the future:
  - If the timer expires before a reply to the request arrives, then the call is ***DECLARED FAILED***.
  - However, this declaration might be wrong – it might just be that the timer duration is too short for processing of the particular request.
- Clearly an infinitely long timer will avoid false failures:
  - This approach is not useful however, since it will not detect true failures either!

THE AUTHOR OF THE WINDOWS FILE COPY DIALOG VISITS SOME FRIENDS.

Adaptive Timeouts

- Instead, RPC systems use an ***ADAPTIVE TIMEOUT*** approach:
  - If the initial timer expires, the client sets a new timer and sends the server a ***PROBE*** message.
  - The server is configured to acknowledge probes immediately if the request is still being processed.
  - If the client receives the probe acknowledgement before the new timer expires, it knows the server is still working on the request.
  - After receiving a probe acknowledgement, the client sets another timer.
  - Then, at the expiry of this timer, it sets (yet) another timer and sends off another probe.
- This process continues until either the reply is received or one of the probe acknowledgements fails to arrive:
  - For efficiency, the inter probe delay may increase up to a point.

Discriminating between Failures

- The (adaptive) timeout approach does not discriminate between different possible failures:
  - Request (or probe) loss on the way to the server.
  - Server crash.
  - Reply (or probe acknowledgement) loss on the way back to the client.

Discriminating between Failures

- Client's perspective:
  - all failures looks essentially the same – an expected message does not arrive in time.
- It is sometimes possible to find out more detail:
  - Once the client has received at least one probe acknowledgment from the server, it is possible to deduce that the request made it to the server.
  - However, one can't conclude from the absence of a probe acknowledgement that the request did not make it to the server – perhaps the probe or probe acknowledgement was lost.

Re-processing of Requests

- Consider a server operation that adds one to a number:
  - For the RPC system to re-process such a request in the event of failure detection would be incorrect, since its not possible to tell whether the request was processed (to the extent of adding one) previously.
  - Re-processing could lead to the number being increased by two!

```
server.increaseCounter();
```

Re-processing of Requests

- So, re-transmission (and consequent re-processing) of requests runs the risk that a request may be executed more than once:
  - This could be bad, for example a request to deduct some money from a bank account.
  - In these situations, the RPC system should provide AT-MOST-ONCE semantics – a given request must never be processed more than once by the server:
    - The corollary is that sometimes a request will not be executed at all!
- As a result, it is not always permissible for an (at-most-once) RPC system to re-process a request when a failure is detected.

An Optimisation

- Server discriminates lost request failures under certain conditions:
  - Server has an ID that is unique each time it re-starts:
    - For example, the number of times the server has started.
  - The client obtains the server's ID each time it binds, and sends that ID with each request to that server.
  - Each client request is allocated a unique sequence number.
  - On failure detection, instead of a reporting an error, the client resends a copy of the request with its original sequence number.
  - The server maintains (in memory) a list of requests it has processed.
  - When the server receives a request, if it has the current server ID:
    - If in the list of processed requests, send a copy of the reply.
    - If not, process the request.
  - If the request has the wrong server ID, the server has recently crashed, and it can't tell whether it has been processed, so discard it and send back an error message.
- This technique allows the system to overcome some lost request and lost reply failures whilst retaining at-most-once semantics.

An Optimisation

1. CLIENT BINDS TO THE SERVER AND GETS ID

2. WHENEVER THE CLIENT TALKS TO THE SERVER, IT WILL USE SID AND A REQUEST ID

An Optimisation

3. SERVER PROCESSES REQUEST AND SAVES RESULT SOMEWHERE

CLIENT

SERVER

SERVER ID: 1
RID 1: 42

4. IF CLIENT DOES NOT GET A RESPONSE FROM THE SERVER, IT WILL RESEND

SERVER

SERVER ID: 1
RID 1: 42

SERVER ID: 1
RID 1: 42

An Optimisation

5. IF THE SERVER ID IS THE SAME, THE SERVER WILL RE-SEND THE REPLY

6. IF SERVER HAS CRASHED IN THE MEANTIME, THE SERVER IDS WILL BE DIFFERENT; ERROR MESSAGE WILL BE SENT

Client Failures

- When a client crashes, it is wasteful to continue processing any requests from that client.
- More importantly, the optimisation described previously relies on **unique** sequence numbers, and a client's sequence numbers are reset when the client restarts:
  - Request messages need to include a unique client ID.
  - This is derived in the same way as the server ID.
  - Taken together, these form a session ID:
    - used instead of the server ID
- Also, the client needs to differentiate a reply to a request it has sent since it booted from a reply to a request that it sent before it crashed:
  - Reply messages also need to have session ID and sequence numbers.
  - Ignore replies from a previous session.

At-least-once Semantics

- There is an alternative RPC semantics:
  - With ***AT-LEAST-ONCE*** semantics, each request is guaranteed to execute to completion.
  - The corollary is that each request may in fact execute (wholly or partially) more than one time.
- Implementation of at-least-once semantics is trivial:
  - Re-send each request until a reply is received!
- So, why is at-least-once semantics not the answer to all problems?
- Operations that are suitable for calling with at-least-once semantics are said to be IDEMPOTENT

Other Semantics

- The at-most-once semantics is more generally useful, but:
  - Even the optimisation to at-most-once doesn't give certainty in all cases – it's still at-most-once, just that the "once" part is more likely.
  - In the event a failure is detected, the operation may have executed partially and then crashed – the RPC system says nothing about this.
- Other possible semantics:
  - ***ZERO-OR-ONCE*** – each operation either executes completely or not at all – failures due to partial executions are prevented.
  - ***RELIABLE NETWORK*** – failures due to loss of request and replies are prevented.
  - ***EXACTLY-ONCE*** – each operation executes to completion.
- How do you think we can go about achieving exactly-once semantics?
  - Hint, think about the optimisation to at-most-once, and what caused it to break down.

System model: Everything can fail!

- Need to establish the system model in order to determine algorithm
- System model fails:
  - Communication
  - Node reliability
  - Node behaviour

Communication failures

- Message loss
- Message reordering
- Message duplication
  - resend on message loss
- Message corruption
  - accidental, deliberate
- Message transmission delays
- Network partitioning

Node reliability

- Nodes can crash (how many?)
- … and then restart
- “Correct” and “incorrect” nodes

Byzantine Failure Model

- Any behaviour is expected/permitted of incorrect nodes
  - refuse to follow protocol
  - forge/fake messages from other nodes
  - collusion to deceive correct nodes
  - interfere with communication

System models

- **The weaker the system model (e.g., the more failures) the stronger the algorithm in place**
- Lamport's Mutex:
  - service: distributed mutual exclusion
  - system model:
    - reliable message delivery
    - in-order delivery
    - reliable nodes

Impossibility Theorems

- CAP - Consistency, Availability, Partitioning
  - we cannot have all three at the same time
- Fischer, Lynch, Paterson (FLP)
  - no consensus is possible in an asynchronous network with unreliable nodes
  - can’t tell difference between message delay and failure

Bounds

- Consensus: N>= 2f+1
- Byzantine consensus: N>= 3f + 1

Example

- Suppose we have an application requirement that an employee's manager is always paid more than the employee:

  emp.salary < emp.manager.salary

- This is referred to as a ***CONSISTENCY CONSTRAINT***:
  - We need to be able to write applications that can manipulate data whilst maintaining such requirements.
  - In a client/server application, we need to be able to support manipulation of such data by several remote operations executing concurrently.
  - ***Atomic transactions*** provide one way to write such applications.

Example - Incomplete Executions

- Suppose we have the code:
  - emp.salary = emp.salary * 1.1
    if (emp.salary >= emp.manager.salary) {
      emp.manager.salary = emp.manager.salary * 1.1
    }
- Now, if the server crashes after executing the first assignment, but not the if statement, it is possible that the consistency constraint will be violated.
- This situation, where only part of an execution is completed, is termed a failure to achieve ***ATOMICITY***.

Example - Incomplete Executions

- Suppose we have the code:
  - emp.salary = emp.salary * 1.1

- Clearly, since this does not check the manager's salary at all, this code can trivially lead to a violation of the consistency constraint.

- This code can fail to preserve ***CONSISTENCY***, even when it executes to completion.

Interference

- Suppose we have two concurrent executions of the code:
  - ```
    emp.salary = emp.salary * 1.1
    if ( emp.salary >= emp.manager.salary ) {
      emp.manager.salary = emp.manager.salary * 1.1
    }
    ```
- Now, consider the following:
  - Execution A executes the first line.
  - Execution B executes the first line.
  - At this point, the salary is 1.21 times its original value.
  - A executes the if statement, and calculates, but does not assign, the new manager salary (1.1 times the original).
  - B executes the if statement, multiplying the manager's salary by 1.1.
  - A completes, assigning 1.1 times the original manager's salary.
- The employees salary has increased by 21%, the manager's by 10%, so the consistency constraint may be violated.
- This can be attributed to a lack of ***ISOLATION*** of concurrent executions:
  - An alternative (non-transactional) analysis identifies a failure of cooperation.
    ```

Durability

- Finally, we can't reliably do anything unless we can be sure that at some point an execution has had a permanent effect.
- If we can't guarantee this, then we can't guarantee that our system will ever get anything done in any lasting sense.
- A failure to provide this guarantee is said to be a failure to provide ***DURABILITY***.
  - guarantees that once a transaction has been committed, it will remain committed even in the case of a system failure

Transaction Properties

- ***ATOMICITY*** – transactions are indivisible with respect to crashes. That is, transactions either ***COMMIT*** and have full effect or ***ABORT*** and have no effect at all; it is not possible for a crash to lead to only part of the effect of a transaction being recorded.
- ***CONSISTENCY*** – when considered by itself, each transaction moves the application from one consistent state to another.
- ***ISOLATION*** – transactions are indivisible with respect to each other. This is, it not possible for a transaction to see a temporary state during the execution of a concurrent transaction.
  - A related idea is ***SERIALISABILITY***, this describes the situation that when two transaction execute concurrent, the effect is the same as if one executed after the other.
- ***DURABILITY*** – the effects of those transactions that commit will survive subsequent failures:
  - Not all failures, but certainly server crashes, and possibly others.

Transaction systems

- The hard part about building a transaction processing system is dealing with the following:
  - ***CONCURRENCY CONTROL*** – ensuring that concurrent transactions execute in isolation and achieve serialisability.
  - ***RECOVERY*** – ensuring both that the effects of committed transactions survive crashes (durability) and that only the effects of committed transactions survive (atomicity).
- Concurrency control and recovery are not independent, so a transaction system must co-ordinate its concurrency control and recovery mechanisms.

Transaction Commit

- A transaction is deemed to commit if and only if the commit record for that transaction reaches the disk:
  - The commit record is preceded by enough information to ensure the transaction's full effect is durable – either the modified data is forced to disk (very inefficient), or updates are recorded in the log (redo records).
  - Writing the commit record is an all-or-nothing event.
- However, this only works for transactions confined to a single server:
  - The existence or absence of a commit record in a single, sequential, log determines a transaction's fate.
- In a ***DISTRIBUTED TRANSACTION***, there will be log records in the logs of each server involved in the transaction:
  - We need an end-of-transaction record in each log, to determine whether the complete effect has been recorded.
  - But, each end-of-transaction record is written independently – so it's possible that some will be written and not others – unlike writing a commit record, writing the end-of-transaction records is not an all or nothing event.

Distributed Transactions

- The ***ATOMICITY OF TRANSACTIONS*** requires that when a distributed transaction comes to an end, either all of its operations are carried out or none of them.
- In a ***DISTRIBUTED TRANSACTION***, there will be log records in the logs of each server involved in the transaction:
  - We need an end-of-transaction record in each log, to determine whether the complete effect has been recorded.
  - But, each end-of-transaction record is written independently – so it's possible that some will be written and not others – unlike writing a commit record, writing the end-of-transactions records is not an all or nothing event.

Group Activity

- Come up with a protocol in which a number of databases decide whether to commit or not commit a distributed transaction

One Phase Commit

- Simple ***ONE-PHASE ATOMIC COMMIT PROTOCOL***
  - The client sends the commit or abort request to all of the participants in the transactions.
  - Keep on repeating the request until all of them have acknowledged that they had carried it out.
- This approach is simple but will not work:
  - When the client requests a commit, it does not allow a server to make a unilateral decision to abort.

Distributed Transactions

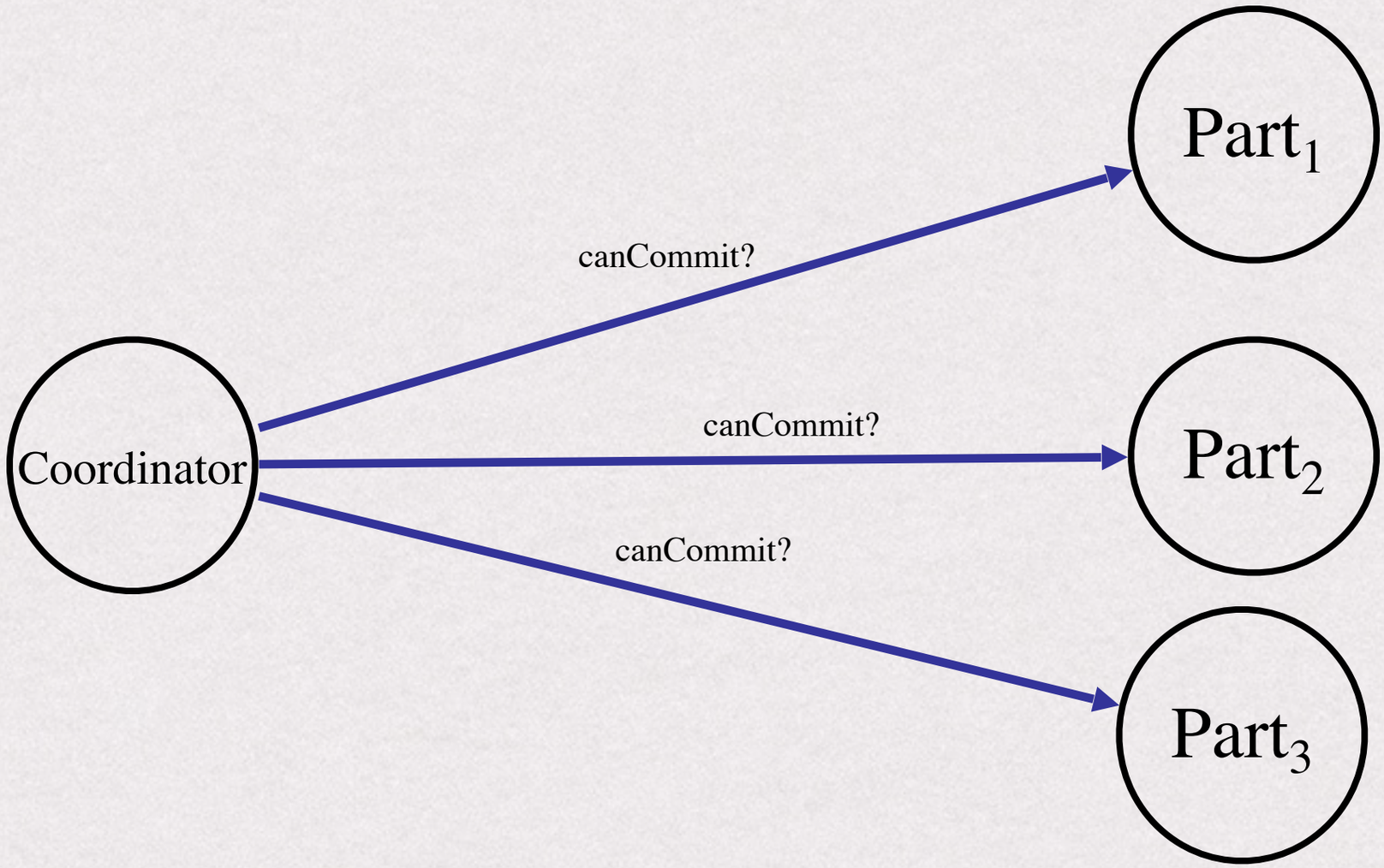
- Efficient ***DISTRIBUTED TRANSACTION PROCESSING*** requires more sophisticated commit and recovery techniques:
  - We shall examine the simplest efficient distributed commit algorithm, namely ***DISTRIBUTED TWO-PHASE COMMIT***.
  - Note that this has nothing to do with two-phase locking.
- The distributed two phase commit algorithm for a given transaction includes all processes involved in that transaction:
  - As a preliminary, one of these processes is elected or appointed as the ***COORDINATOR***, and the remaining processes are termed the ***PARTICIPANTS***.
  - In a client/server system, the client that started the transaction is typically the coordinator.
- Since it is distributed algorithm, it needs to cope with these processes failing and restarting during a commit operation.

Distributed 2PC

- The algorithm is divided into two phases, a ***PREPARE*** or ***DECISION*** phase and a ***COMMIT PROPAGATION*** phase.
- The ***prepare*** phase:
  - The coordinator sends a *prepare* message to each participant.
  - If the participant determines that it cannot commit, it sends an *abort* response to the coordinator and can abort the transaction locally.
  - If the participant determines that it can commit, it writes a *prepare* record to the tail of the log, forces the log, and then sends a *prepare* message to the coordinator – the participant may not commit the transaction at this point.

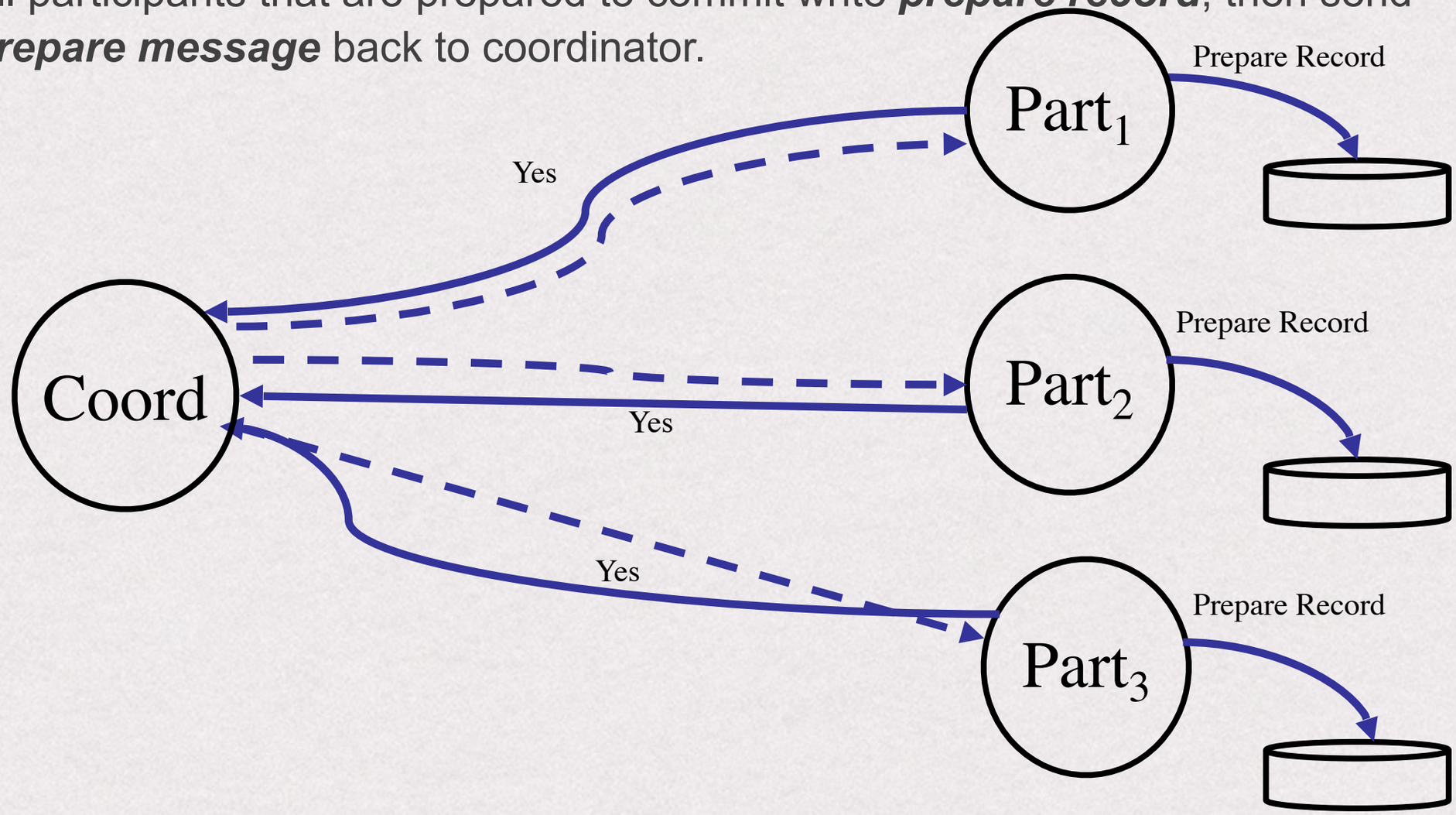
Prepare Phase

1. Coordinator sends ***prepare message*** to all participants.



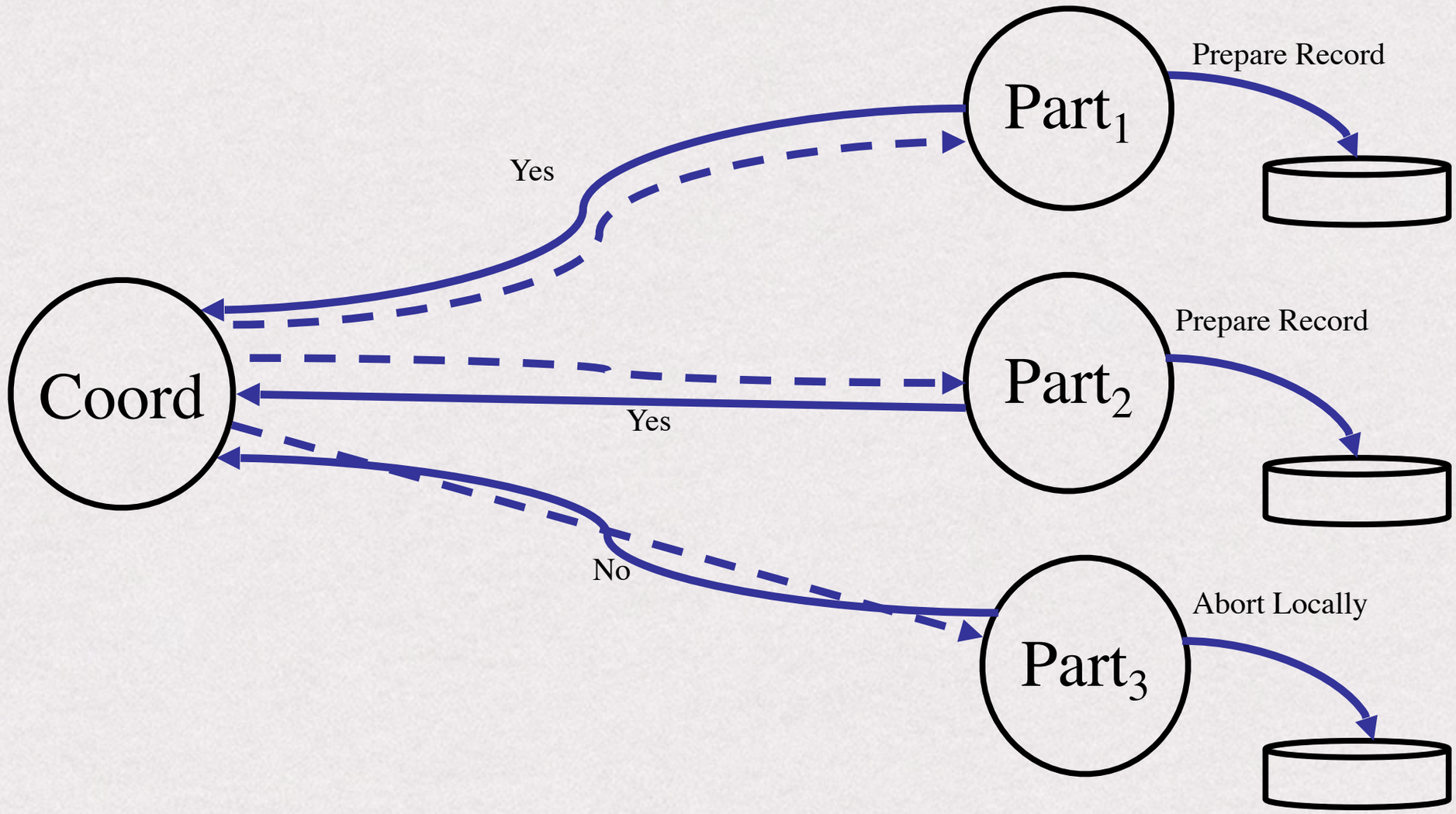
Prepare Phase

1. Coordinator sends ***prepare message*** to all participants.
2. All participants that are prepared to commit write ***prepare record***, then send ***prepare message*** back to coordinator.



Prepare Phase

1. Some participants are prepared to commit, while some decide to abort.

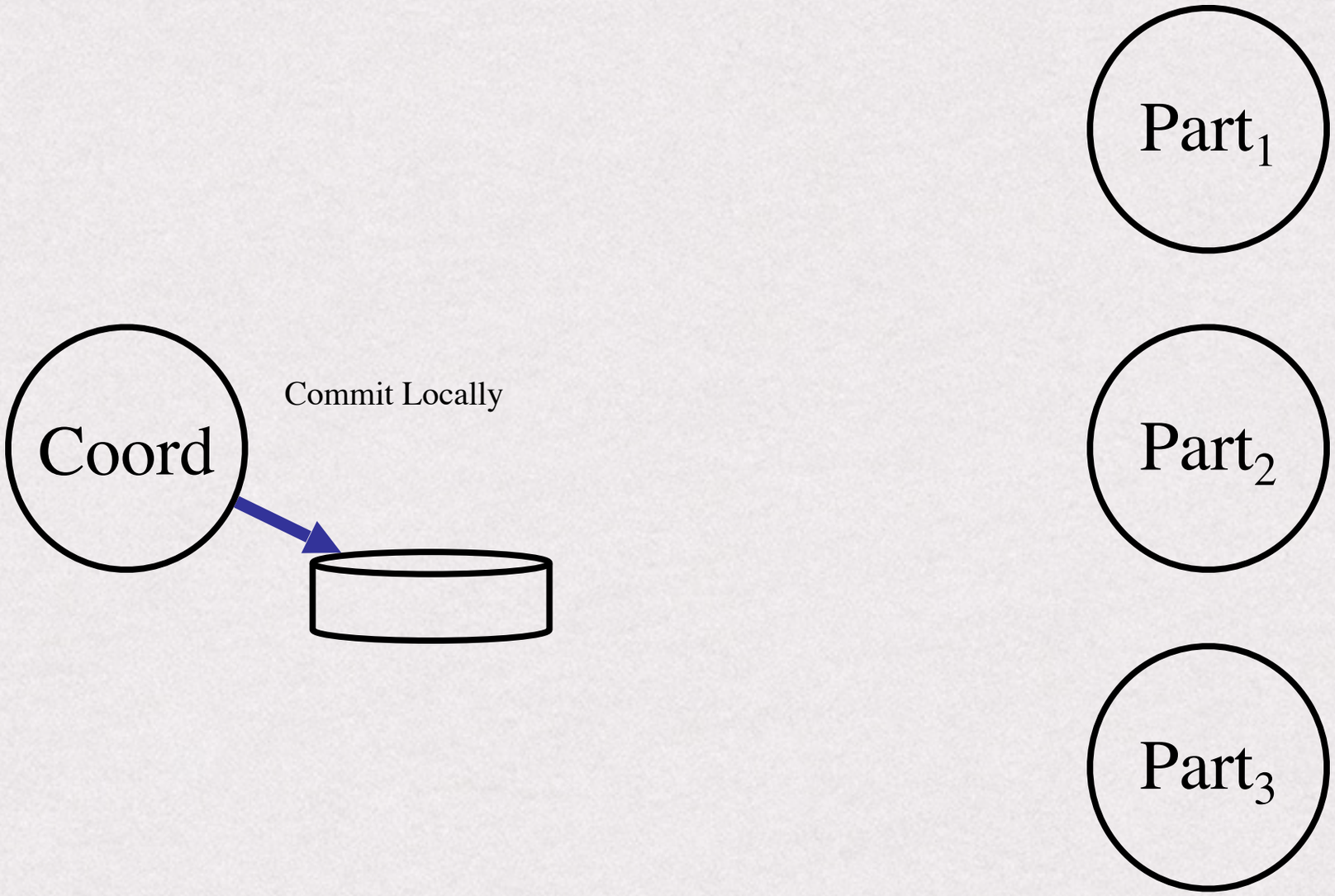


Distributed 2PC

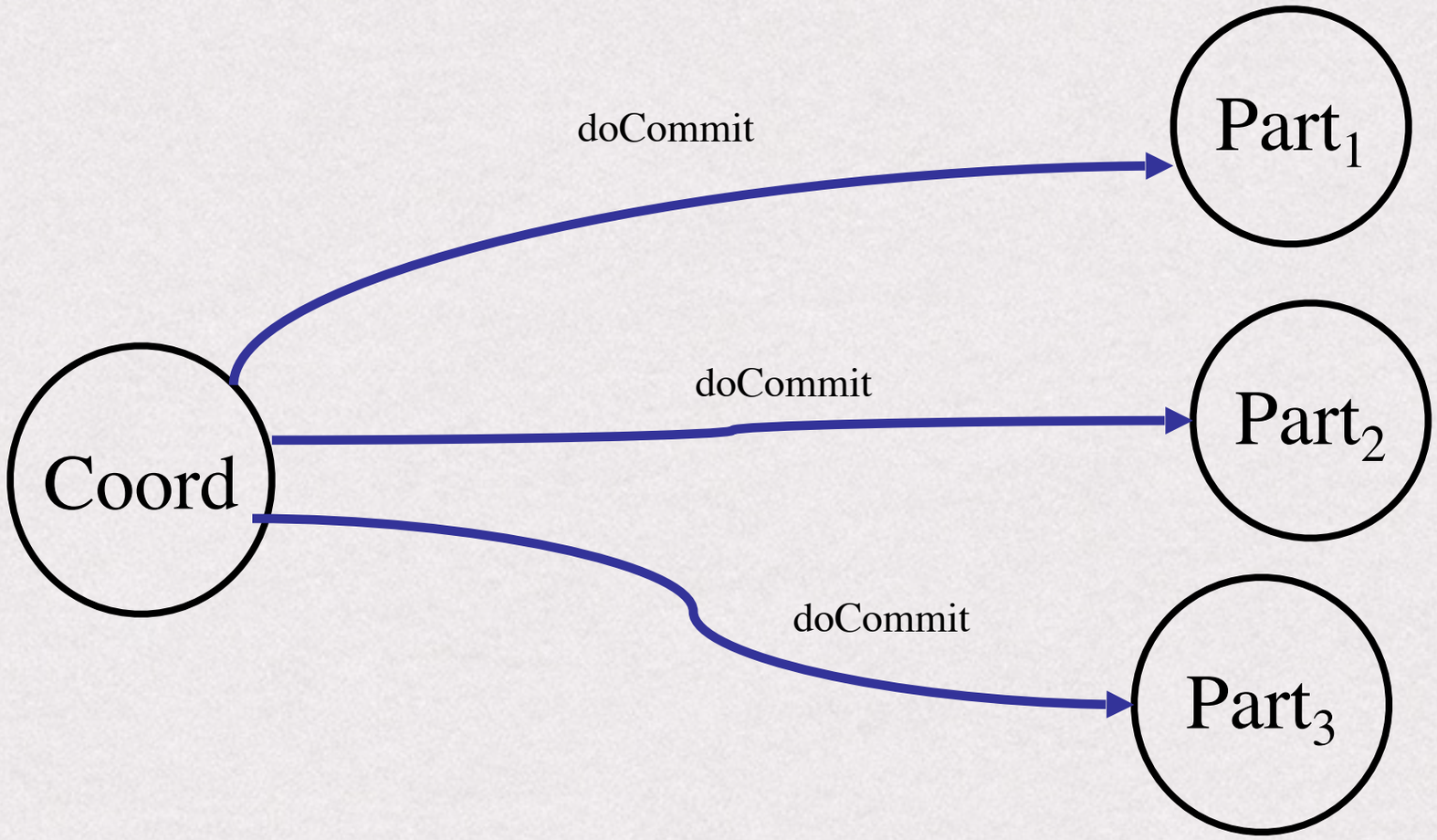
- The *commit propagation* phase:
- If the coordinator receives *prepare* responses from all participants, it decides to commit the transaction. Otherwise, the coordinator decides to abort the transaction.
- If the coordinator decides to commit, then:
  - It writes a *commit decision* record to the log, and forces the log.
  - It sends a commit message to each participant, with an at-least-once protocol.
  - When each participant receives a *commit* message, it writes a *commit* record to its log, and forces the log.
  - Each participant sends a acknowledgement to the coordinator after forcing the commit record.
- If the coordinator decides to abort, it must inform (at least) those participants that sent *prepare* message replies in the first phase:
  - This enables those participants to abort the transaction locally.

Propagation Phase

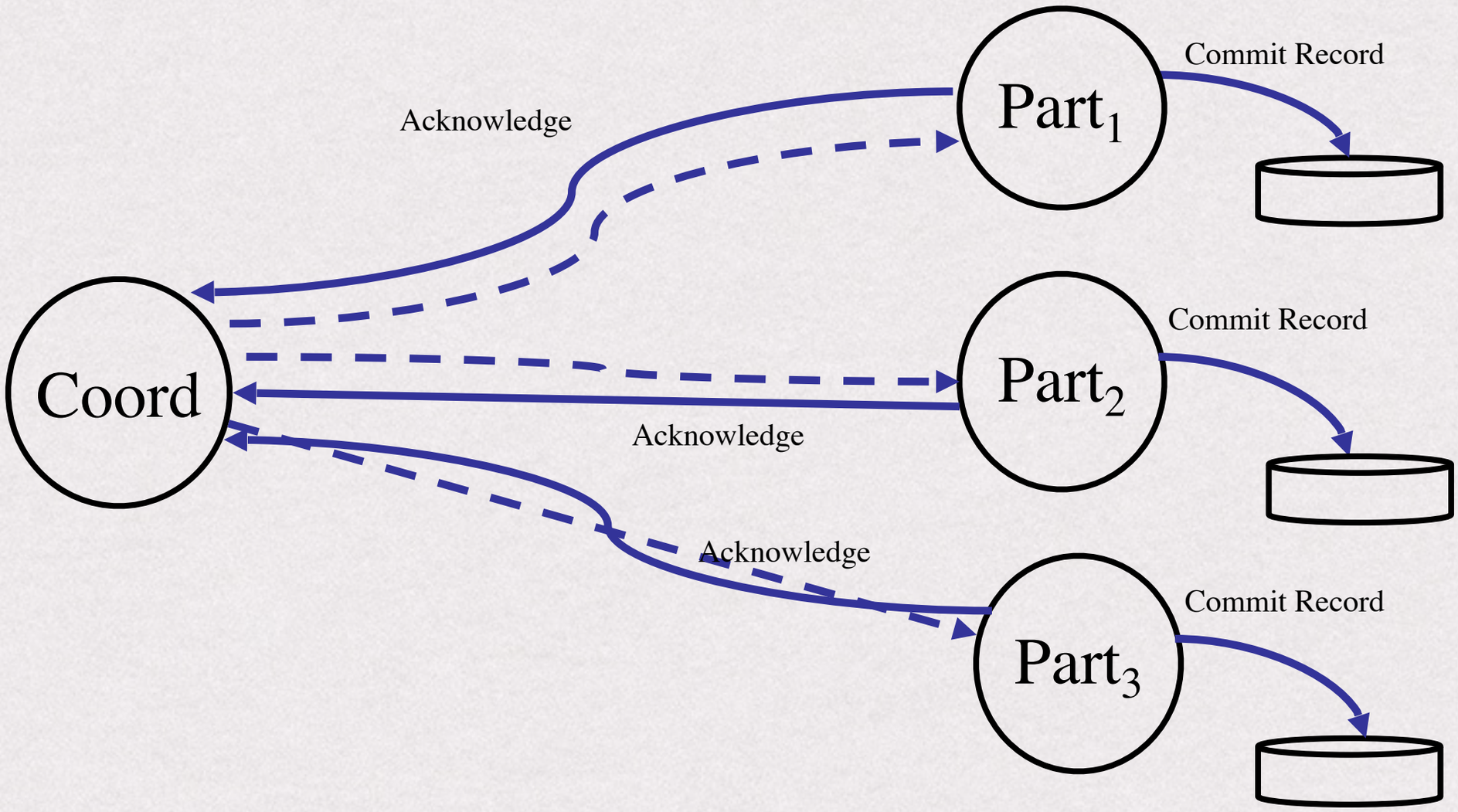
- Coordinator writes ***commit decision record*** to its log.



3. Coordinator writes ***commit decision record*** to its log.
4. Coordinator sends ***commit message*** to all participants.



3. Coordinator writes ***commit decision record*** to its log.
4. Coordinator sends ***commit message*** to all participants.
5. All participants write ***commit record*** to log, then send ***acknowledge message*** to coordinator.



Decision to commit

- ***How do participants decide whether to commit?***
- If a participant has crashed between the first time the transaction reaches that participant and when the participant receives the *prepare* message, then:
  - It may have lost some updates, either in the form of redo records not reaching the disk, or data updates not reaching the disk.
  - In such cases, it must respond with an abort message.
- This can be implemented as follows:
  - Each process records a crash count in stable storage, which is incremented each time the computer reboots.
  - When a given transaction reaches a process, write a *start-transaction* record, including the current crash count.
  - When a *prepare* message reaches a participant, it responds with *prepare* iff the crash count in the transaction's start-transaction record is the same as the current crash count.

3PC Motivation

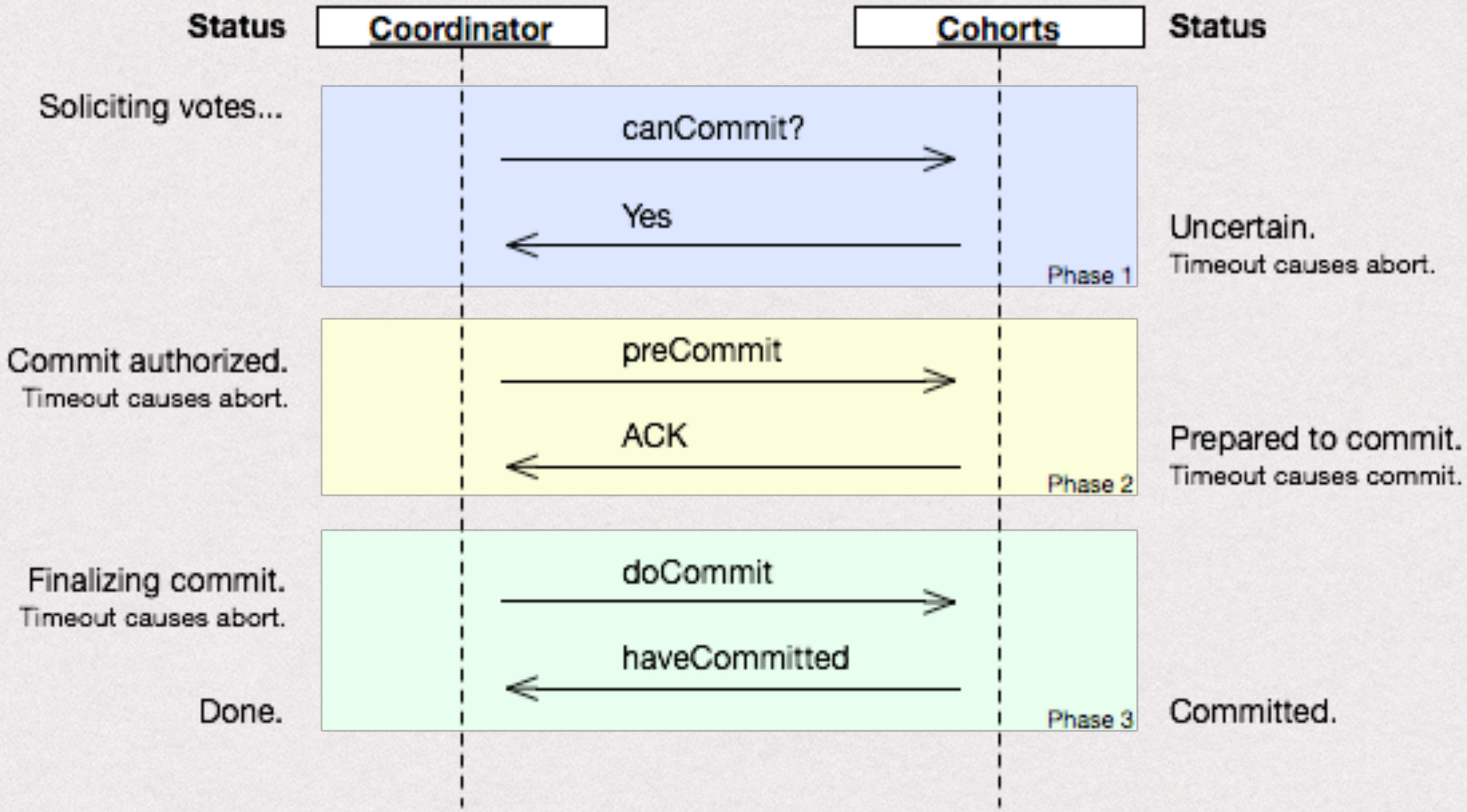
- 2PC is most awesome
  - However, it can block on failures
  - How? Why?

- The distributed two phase commit protocol is subject to blocking:
  - At some points, processes involved in a given transaction must wait for messages, and can neither abort nor commit the transaction until it receives the required messages.
- Participants that respond with prepare messages must block, waiting for the result (commit or abort) from the coordinator:
  - If the coordinator crashes, the participant could be waiting for a long time.
  - Similarly, if the network partitions, a participant in a different partition to the coordinator will have to wait for the network to recover.
- More sophisticated protocols, such as *three phase commit* protocols, can avoid blocking in the event of coordinator crash.
- Blocking is unavoidable in the event of network partitions.

3PC

- Proposed 10 years after 2PC
- Goal: Turn 2PC into a live (non-blocking) protocol
  - 3PC should never block on node failures as 2PC did
- Insight: 2PC suffers from allowing nodes to irreversibly commit an outcome before ensuring that the others know the outcome, too
- Idea in 3PC: split“commit/abort”phase into two phases
  - First communicate the outcome to everyone
  - Let them commit only after everyone knows the outcome

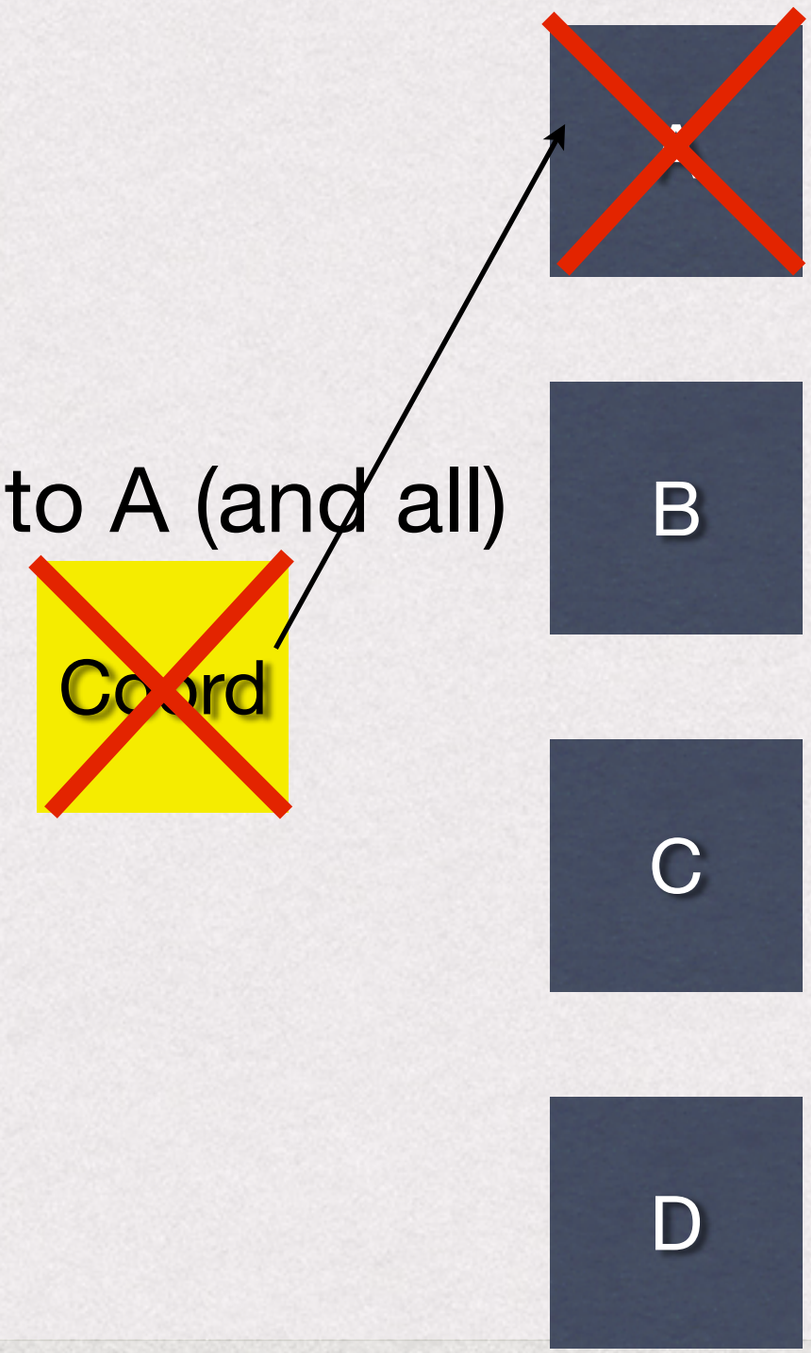
3PC



- http://en.wikipedia.org/wiki/Three-phase_commit_protocol

Crash Scenario

- Coord sends preCommit to A (and all)
- A gets it and commits
- They both crash

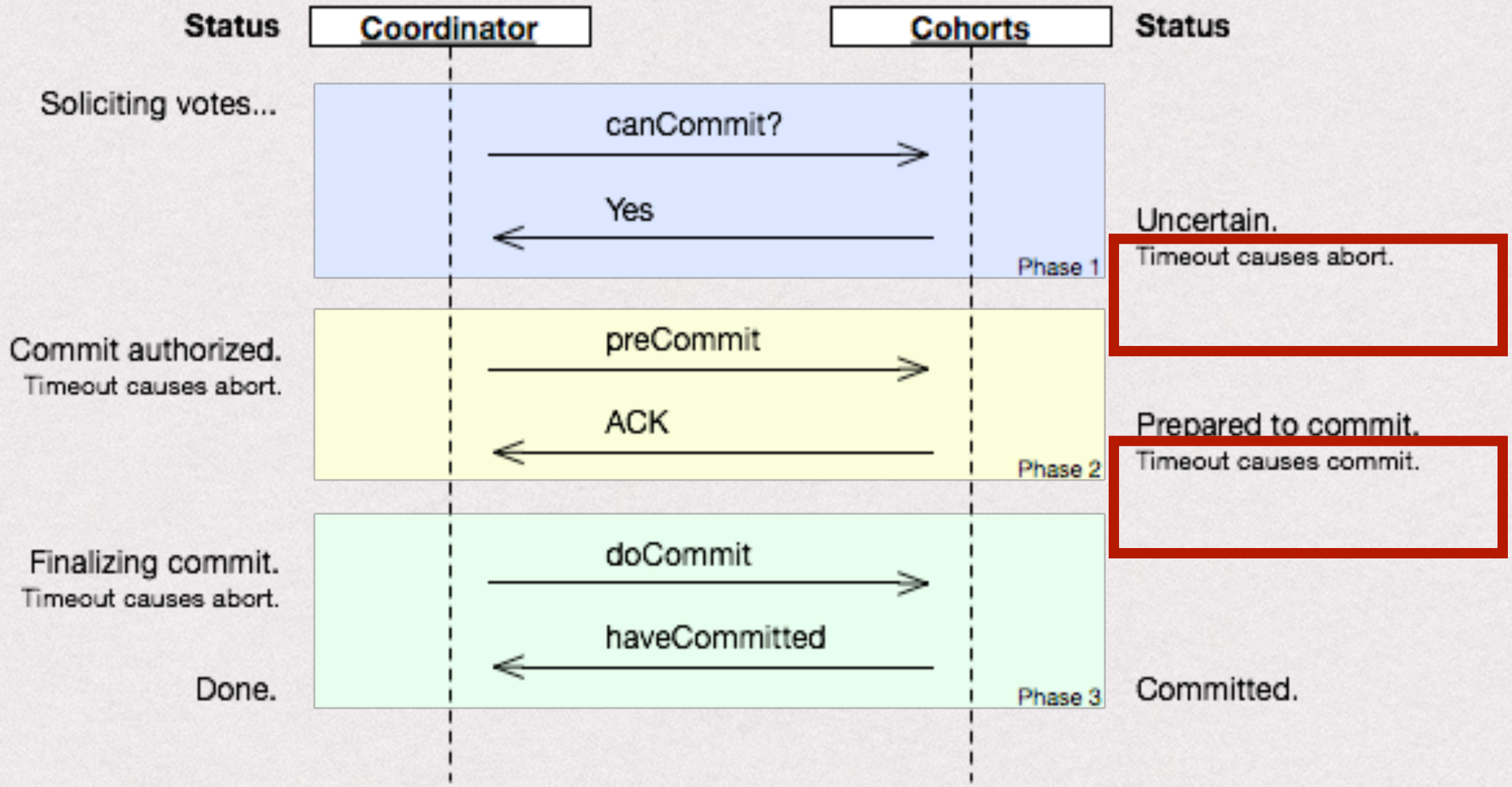


What would happen in 2PC?

What would happen in 2PC?

- Everybody else waits for the coordinator to come back online

3PC

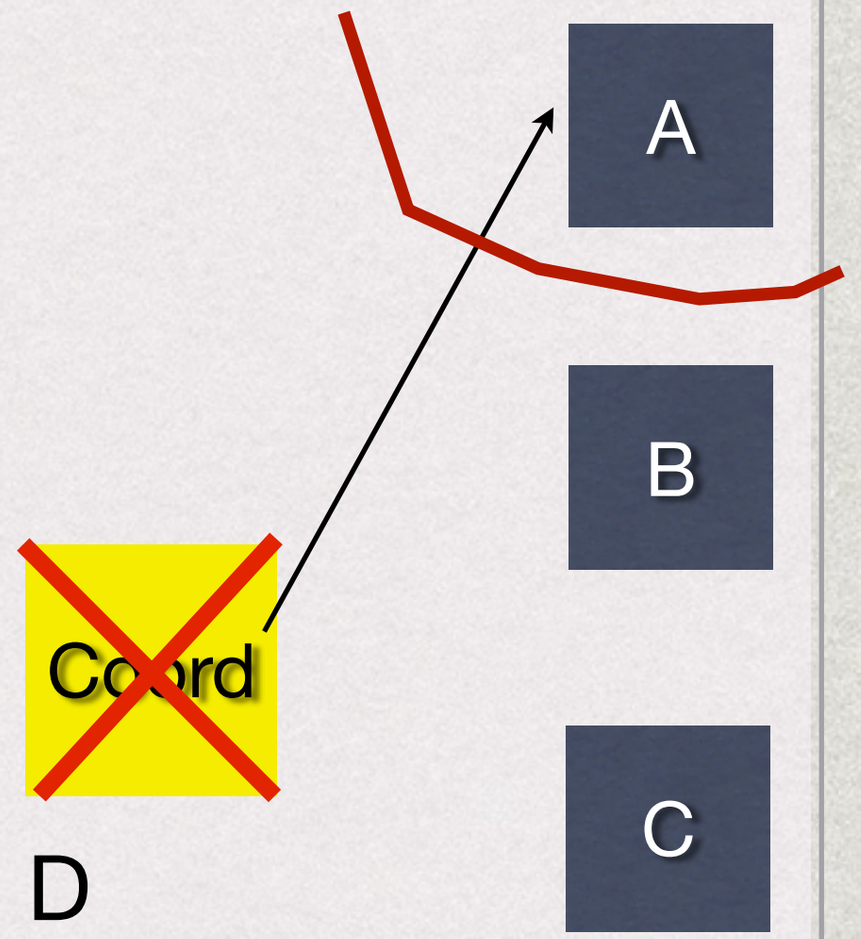


- http://en.wikipedia.org/wiki/Three-phase_commit_protocol

3PC properties

- Liveness: Yay
  - does not block, always makes progress
- Correctness: Nay
  - what about network partitions?
  - A or Coordinator are not crashed, they are just inaccessible

Partition Scenario



- Coord sends preCommit to A
- A becomes partitioned from B, C, D
- Coordinator crashes
- A has received preCommit so will commit upon timeout
- B/C/D have not received it and thus will abort

Question

- Can we design a protocol that is both correct and alive?


YES WE CAN

Group Activity

- Design a protocol that allows you and your friends to decide where you will go out to dinner:
  - some of you never answer text messages
  - some of you answer text messages late and have very strong opinions OR dietary requirements (or both)
  - some of you communicate only via snail mail
  - all of you are really hangry

Paxos

- The only known completely-safe and largely-live agreement protocol
- Lets all nodes agree on the same value despite node failures, network failures, and delays
  - Only blocks in exceptional circumstances that are vanishingly rare in practice
- Extremely useful, e.g.:
  - nodes agree that client X gets a lock
  - nodes agree that Y is the primary
  - nodes agree that Z should be the next operation to be executed

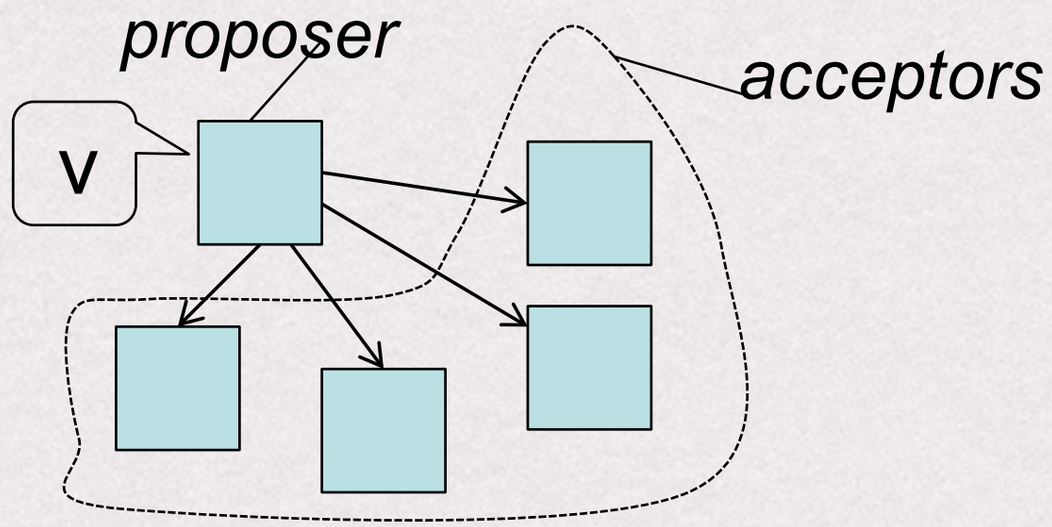
Paxos

- Widely used in industry and academia
- Google: Chubby (Paxos-based distributed lock service)
  - Bigtable and other Google services use Chubby
- Yahoo: Zookeeper (Paxos-based distributed lock service)
- Open source:
  - libpaxos (Paxos-based atomic broadcast)
  - Zookeeper is open-source and integrates with Hadoop
- Strong recommendation: Paxos is hard to get right, so use an existing implementation!

Paxos

- Correctness (safety)
  - If agreement is reached, everyone agrees on the same value
  - The value agreed upon was proposed by some node
- Fault tolerance (i.e., as-good-as-it-gets liveness)
  - If less than half the nodes fail, the rest of the nodes reach agreement *eventually*
- No guaranteed termination (i.e., imperfect liveness)
  - Paxos may not always converge on a value, but only in very degenerate cases that are improbable in the real world

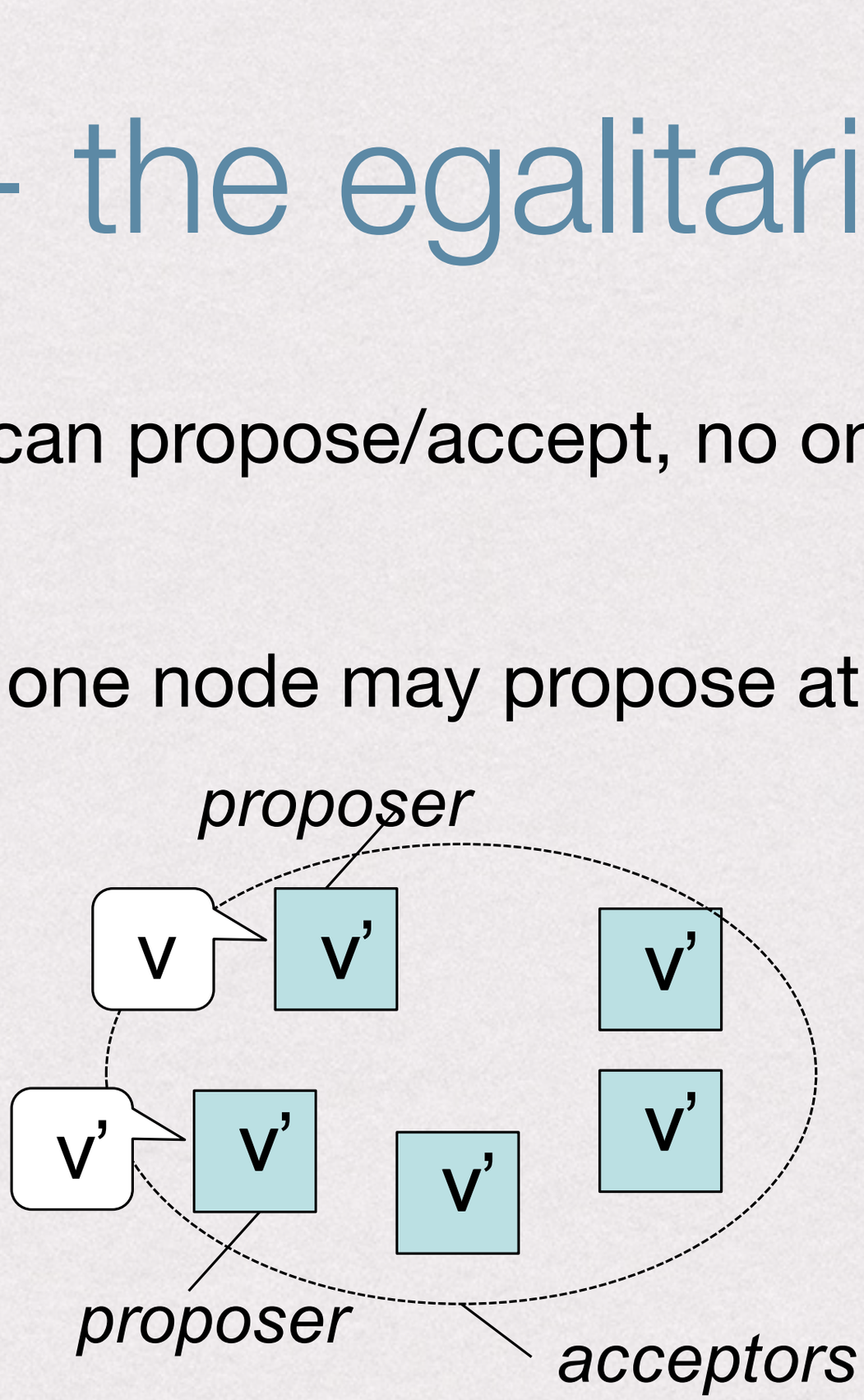
The Basics



- Paxos is similar to 2PC, but with some twists
- One (or more) node decides to be coordinator (proposer)
- Proposer proposes a value and solicits acceptance from others (acceptors)
- Proposer announces the chosen value or tries again if it's failed to converge on a value

Paxos - the egalitarian

- Any node can propose/accept, no one has special powers
- More than one node may propose at one time



Questions

- What if multiple nodes become proposers *simultaneously*?
- What if the new proposer *proposes different values* than an already decided value?
- What if there is a *network partition*?
- What if a proposer *crashes* in the middle of solicitation?
- What if a proposer *crashes after deciding but before announcing results*?
- Scenario: groups of friends organize a party.

Important mechanisms

- Proposal ordering
  - Lets nodes decide which of several concurrent proposals to accept and which to reject
- Majority voting
  - 2PC needs all nodes to vote Yes before committing => may block
  - Paxos requires only a majority of the acceptors (half+1) to accept a proposal => will work if nearly half the nodes fail to reply
  - no two majorities can exist simultaneously --> networks partitions are not a problem

Proposal 1

- Each proposer propose to all acceptors
- Each acceptor accepts the first proposal it receives and rejects rest
- If the proposer receives positive replies from a majority of acceptors, it chooses its own value
  - There is at most 1 majority, hence at most a single value is chosen, even if there are partitions
- Proposer sends chosen value to everyone
- Problems?

Proposal 1

- Each proposer propose to all acceptors
- Each acceptor accepts the first proposal it receives and rejects rest
- If the proposer receives positive replies from a majority of acceptors, it chooses its own value
  - There is at most 1 majority, hence at most a single value is chosen, even if there are partitions
- Proposer sends chosen value to everyone
- Problems?
  - what if multiple proposers propose at the same time?
  - what if proposer dies?

Proposal 2

- Enforce a global ordering of all proposals
- Let acceptors recant their older proposals and accept newer ones
- This will allow consistent progress for both simultaneous proposers and dead proposers
- *Problems*?

Proposal 2

- Enforce a global ordering of all proposals
- Let acceptors recant their older proposals and accept newer ones
- This will allow consistent progress for both simultaneous proposers and dead proposers
- *Problems*?
  - What if old proposer isn't dead, but rather just sloooow?
  - It may think that its proposed value has won, whereas a newer proposer's value has won -- getting back on your word creates problems
    - printed invitations have been sent for the party

Paxos Solutions

- Each acceptor must be able to accept multiple proposals
- Order proposals globally by a proposal number, which acceptors use to select which proposals to accept/reject
  - “node-address:node-local-sequence-number,”e.g.:N1:7
- When acceptor receives a new proposal with # n, it looks at the highest-number proposal it has already accepted, # m, and
  - accepts the new proposal only if n>m
  - rejects it otherwise
- If the acceptor decides to accept new proposal # n, then it will ask the new proposer to use the same value as # m’s

Paxos Phase 1

- a) Prepare
  - A proposer (the leader) creates a proposal n
  - n > than any previous proposal number used by this proposer.
  - The proposer sends a proposal with number n to a majority
- b) Promise
  - If n > any previous proposal number received from any proposer by the acceptor
    - Acceptor must return a promise to ignore all future proposals having a number < n
    - If the acceptor accepted a proposal at some point in the past, it must include the previous proposal number and previous value in its response to the proposer
    - send ***prepare-ok***
  - Otherwise, ignore or send NACK

Paxos Phase 2

- a) Accept Request
  - If a proposer receives enough *prepare-ok* from a majority, it needs to set a value to its proposal
    - If any acceptors had sent a value and proposal number to the proposer, then proposer sets the value of its proposal to the value associated with the highest proposal number reported by the acceptors
    - If none of the acceptors had accepted a proposal up to this point, then the proposer may choose any value for its proposal
  - The Proposer sends an Accept Request message to a majority with the chosen value for its proposal.
- b) Accepted
  - If an acceptor receives an Accept Request message for a proposal n, it must accept it : ***accept-ok***
    - iff it has not already promised to only consider proposals having an identifier greater than n -> also implies acceptor considers proposer LEADER
    - if it has, respond with ***accept-reject***

Paxos Phase 3

- Decide
  - If a leader gets accept-ok from a majority, sends *decide* to all nodes
  - If it fails to get accept-ok, restart

When does a value v get chosen?

When does a value v get chosen?

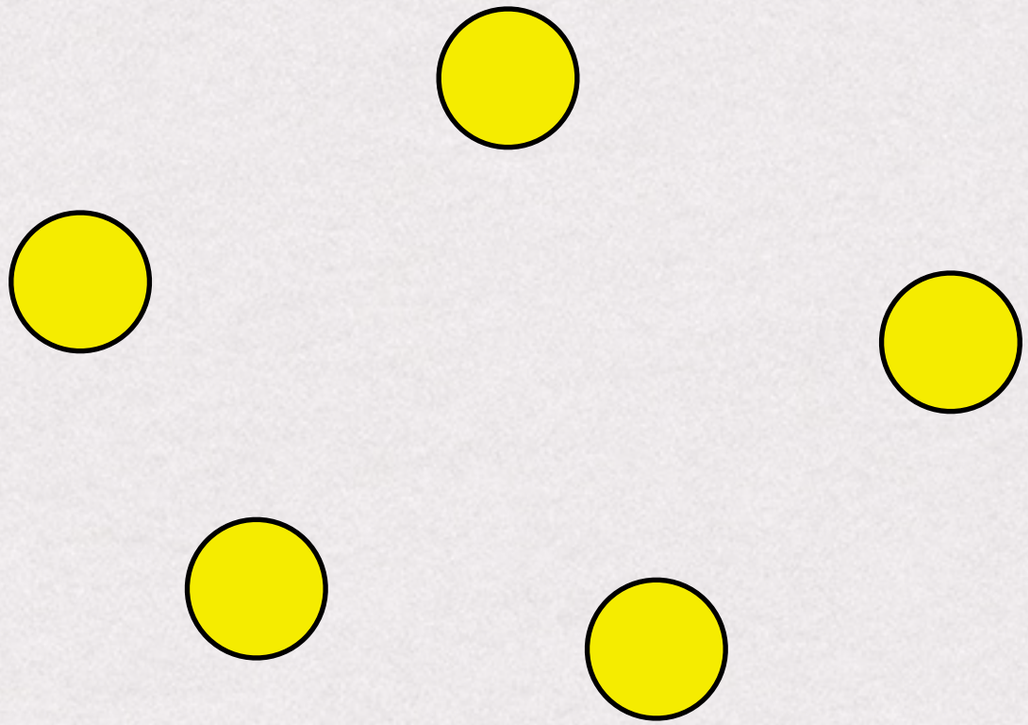
- When leader receives a majority prepare-ok and proposes V
- When a majority nodes accept V
- When the leader receives a majority accept-ok for value V

Majorities

- Paxos requires half+1 to agree on a value before it can be chosen
- Sneaky properties:
  - no two separate majorities can exist simultaneously (not even if we have partitions)
  - If two majorities successively agree on two distinct proposals, #n and #m, respectively, then there is at least one node that was in both majorities and has seen both proposals
  - If another majority agrees then on a third proposal,#p,then there are a set of nodes that collectively will have seen all three proposals, #m, #n, #p
- **Thus, if a proposal with value v is chosen, all higher proposals will preserve value v (so nodes need not recant)**

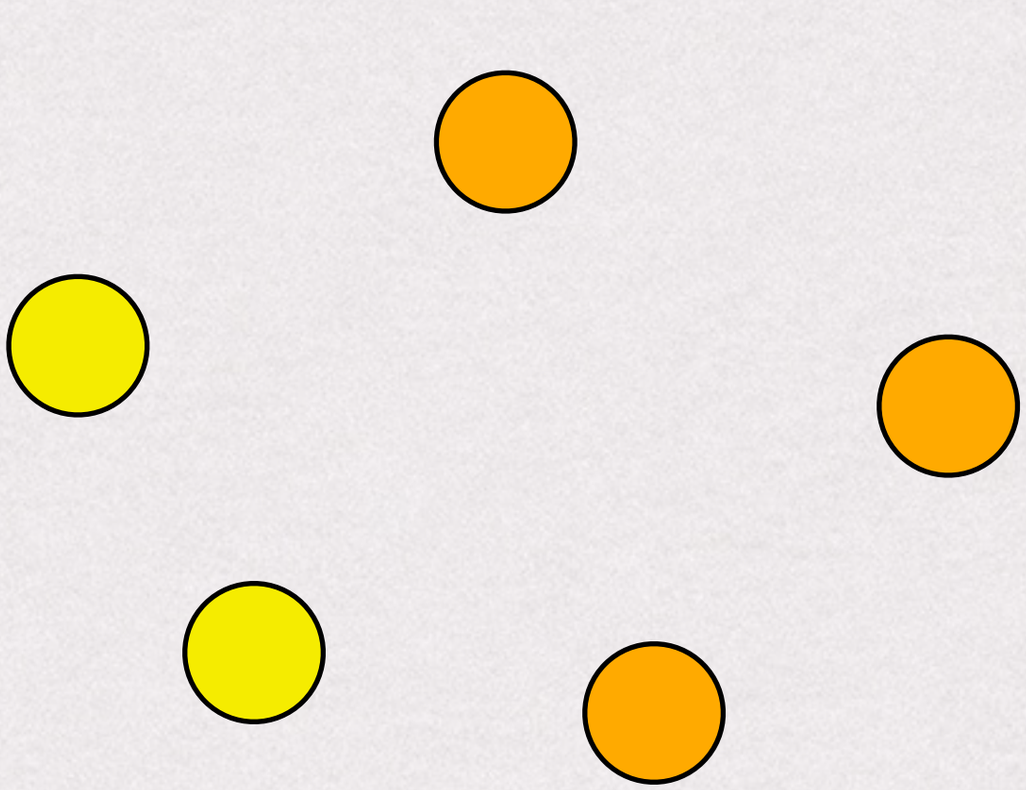
Example - 5 nodes

- half + 1 = 3
- Cannot have two majorities simultaneously

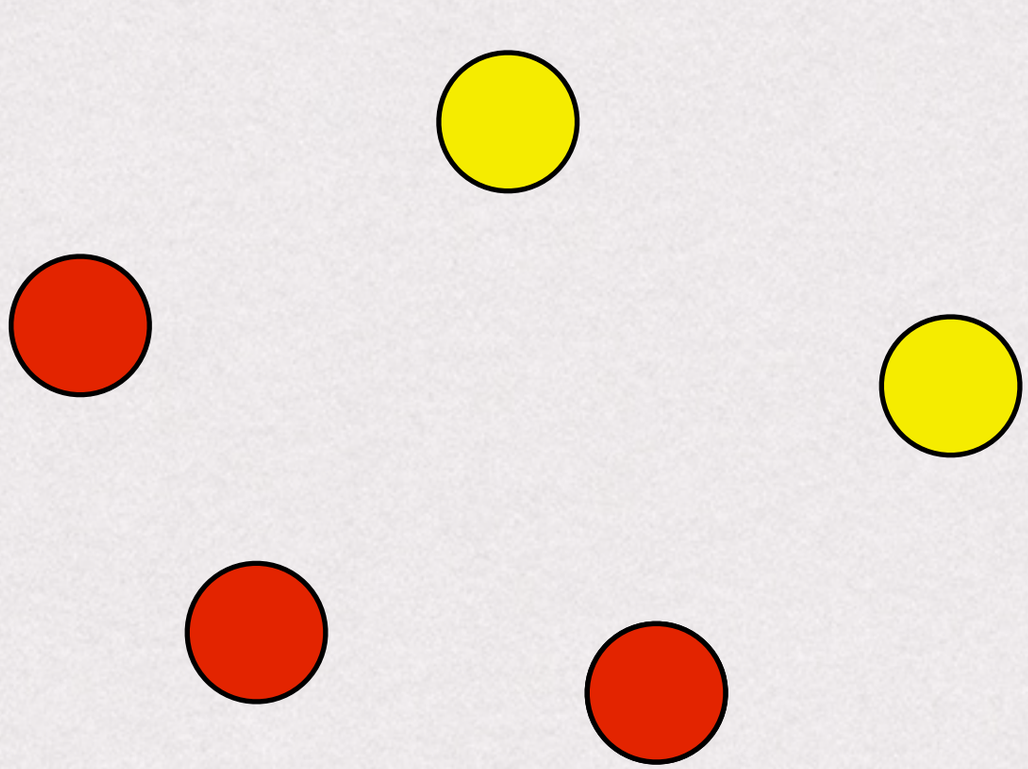


Example - 5 nodes

- If two majorities successively agree on two distinct proposals, #n and #m, respectively, then there is at least one node that was in both majorities and has seen both proposals



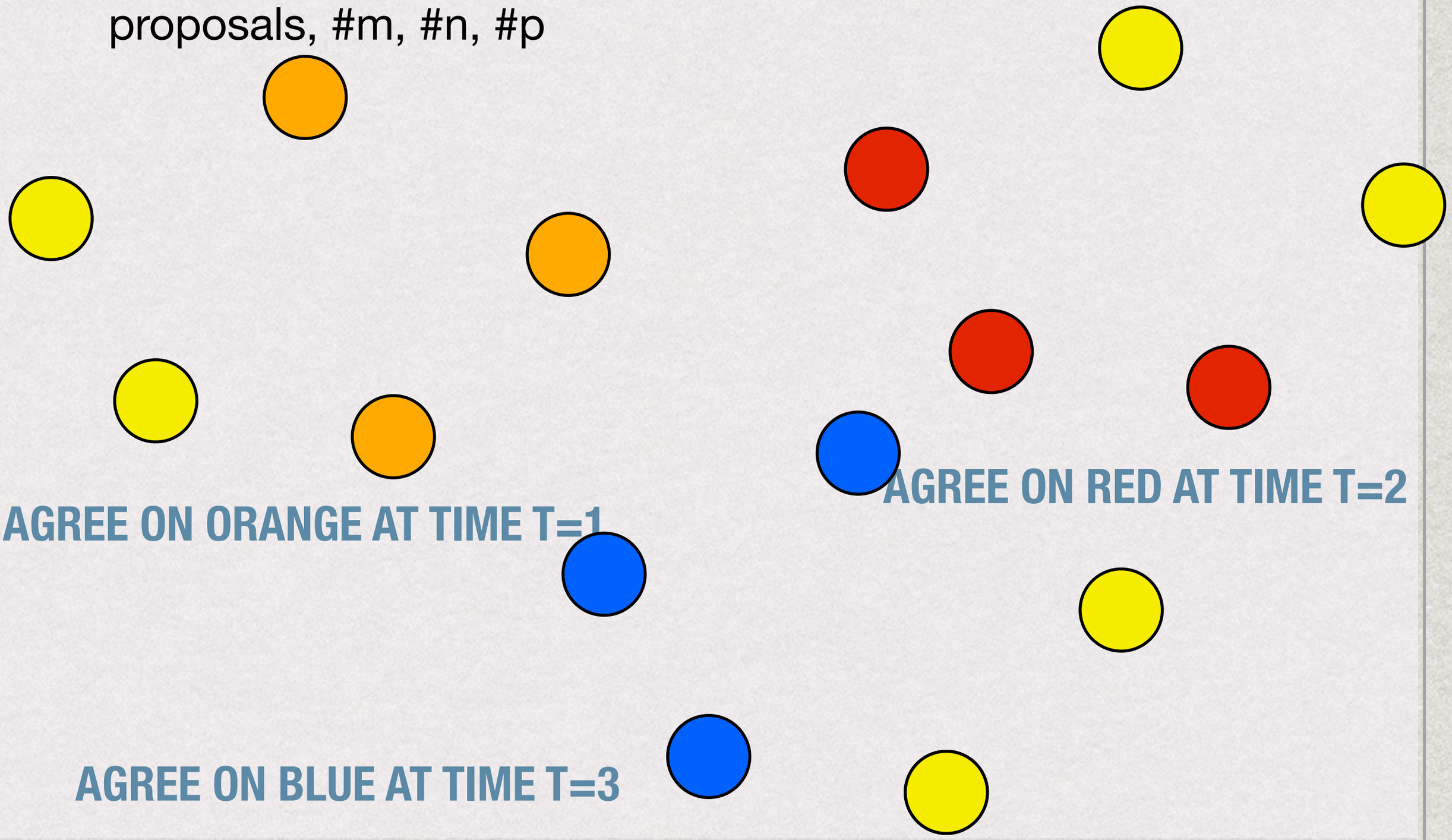
AGREE ON ORANGE AT TIME T=1



AGREE ON RED AT TIME T=2

Example - 5 nodes

- If another majority agrees then on a third proposal,#p,then there are a set of nodes that collectively will have seen all three proposals, #m, #n, #p



Summary

- 2PC proposed in the 1970s. Problem: blocks on failure of single node even in synchronous networks.
- The need for an extra, third phase to avoid blocking is shown in the early 1980s. 3PC is an obvious corollary, but it is not provably correct, and in fact suffers from incorrectness under certain network conditions.
- Paxos is proposed in the 1990s (Leslie Lamport), which is provably correct in asynchronous networks that eventually become synchronous, does not block if a majority of participants are available (so withstands faults) and has provably minimal message delays in the best case.