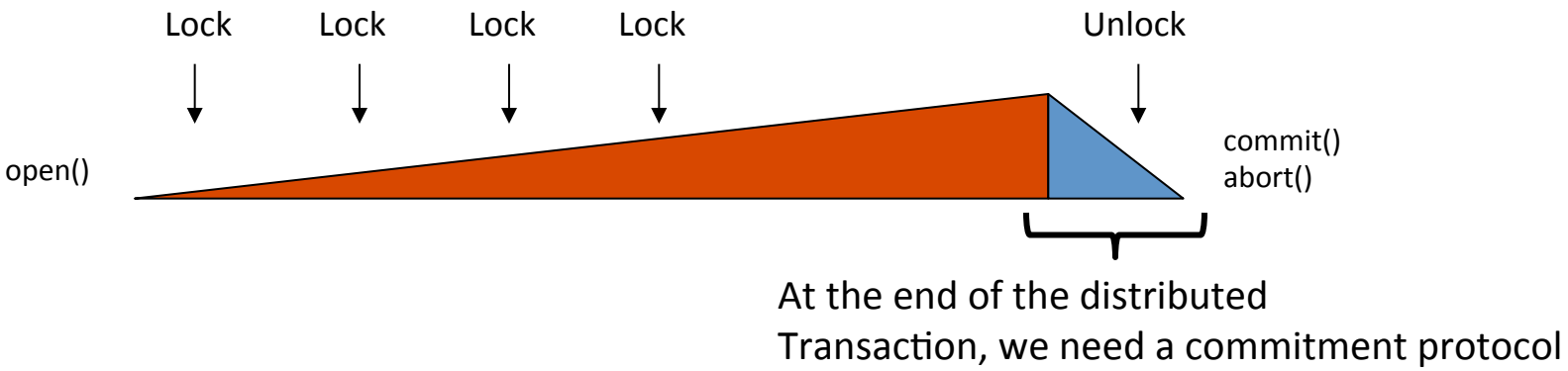
Distributed Transactions II

CS3524 Distributed Systems

Lecture 12

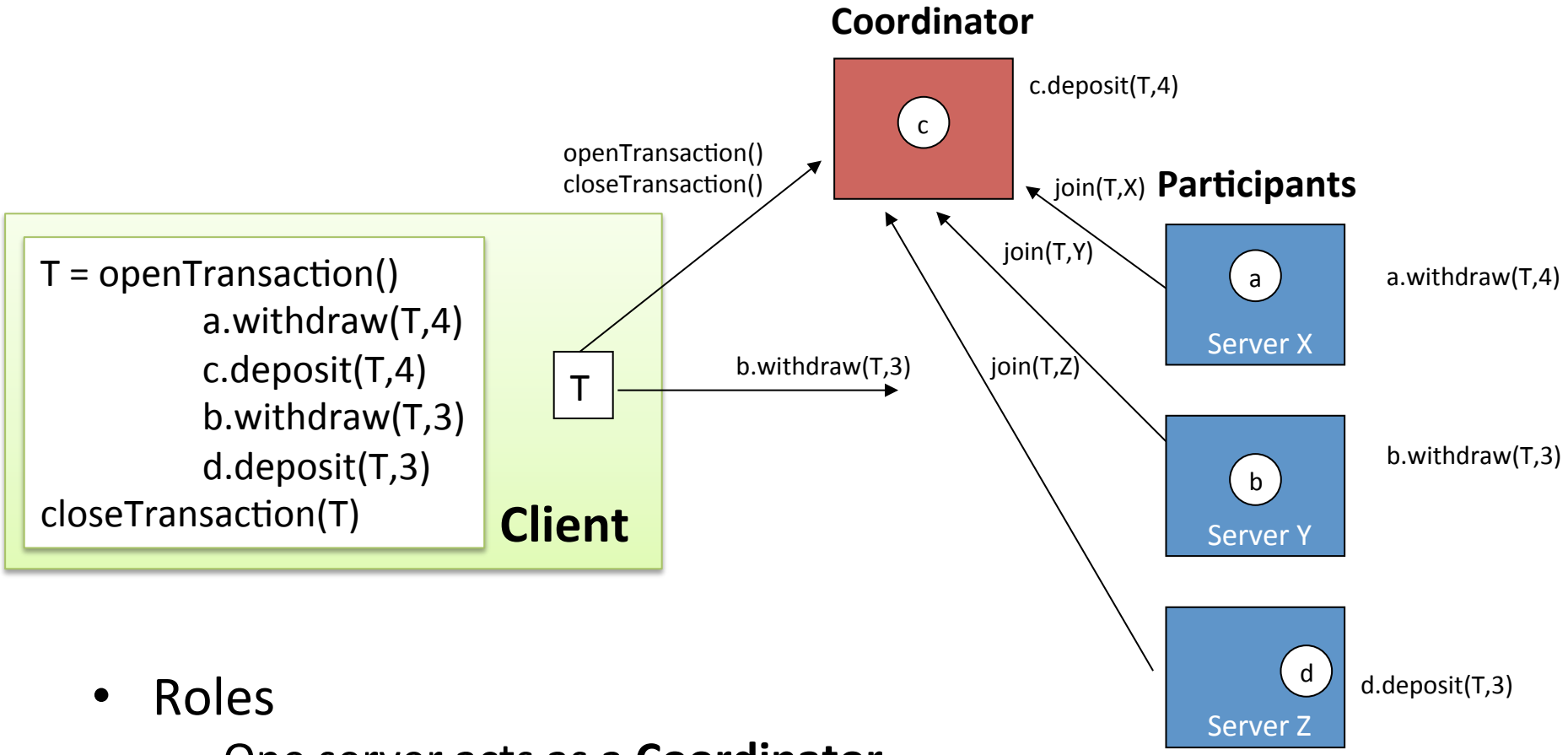
Commit Protocols



- When do we need a commit protocol?
  - To guarantee atomicity for distributed transactions
  - In distributed transactions, a set of servers may manipulate data – these manipulations have to be committed / aborted in an orderly fashion
  - Therefore: servers participating in such a transaction have to communicate and follow a particular protocol to complete their transaction

Two-phase Commit Protocol (2PC)

Coordinator and Participant



- Roles
  - One server acts as a **Coordinator**
  - All other servers act as **Participants**

2-Phase Commit Protocol

- 2 Roles:
  - The coordinator: one of the servers involved will be the "coordinator" of a distributed transaction
  - The participant: each server participating in a distributed transaction (managing particular data objects manipulated by a distributed transaction) is a "participant" and registers with the coordinator
- 2 Phases:

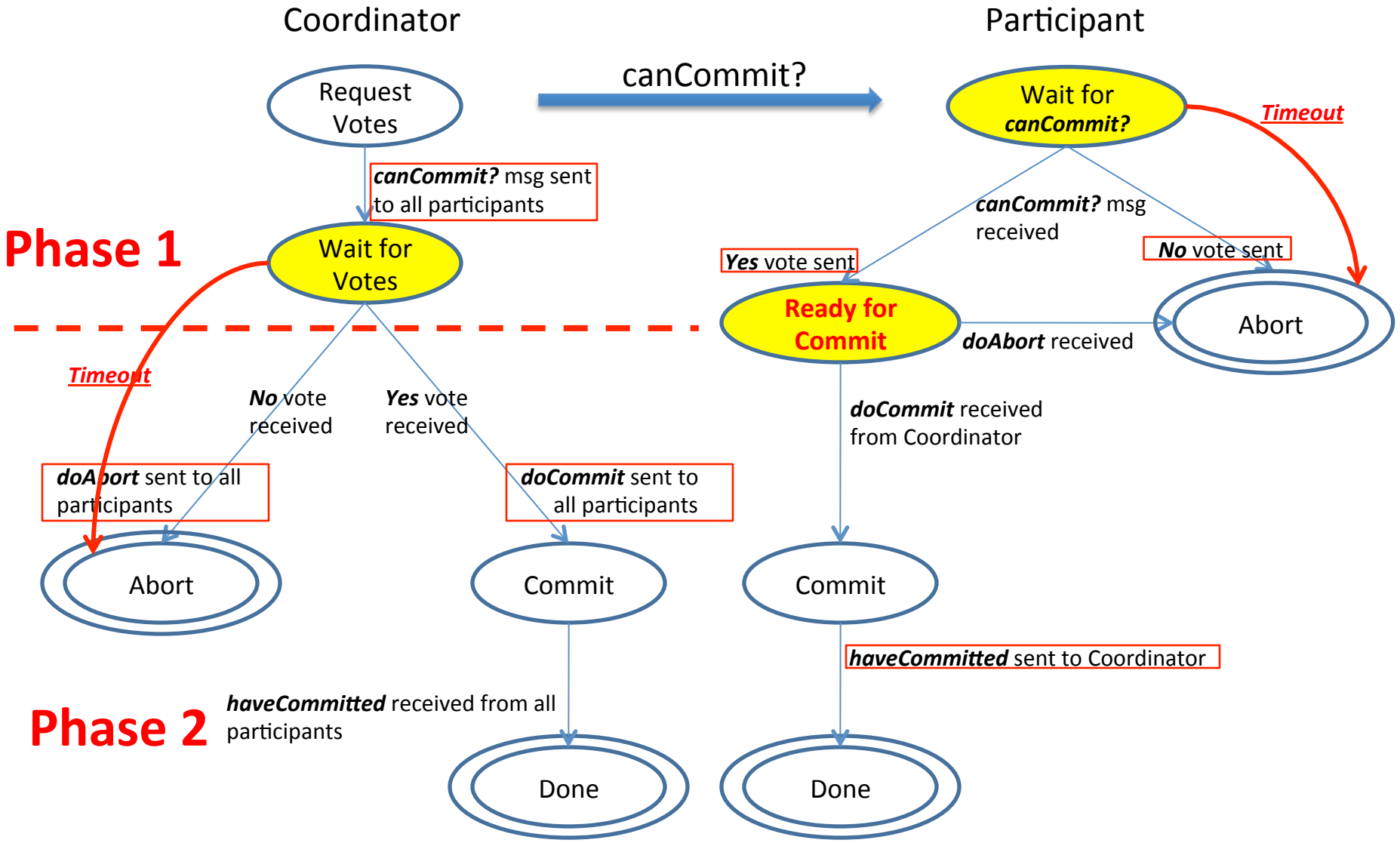
Voting phase – ***prepare to commit***

Coordinator asks participants whether they are ***prepared*** **to commit** or **abort**
Participants send their vote to the coordinator
if they vote "yes", they will move to state "**Ready for commit**"
if they vote "no", they will unilaterally abort

Completion phase – ***commit***

If all votes are for commit, coordinator asks participants to ***commit***
If at least one vote is for abort, coordinator asks all participants to ***abort***

2-Phase Commit Protocol



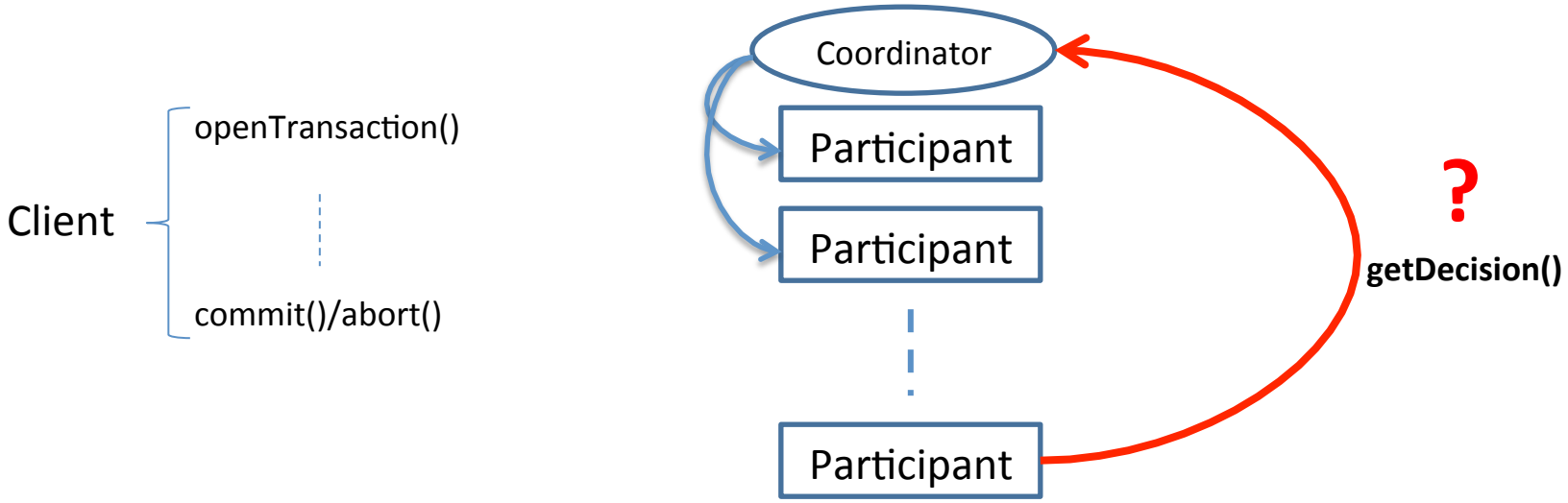
Handling Failure Situations

- Communication Problems:
  - Due to hardware / software / communication failures, it is possible that messages exchanged between Coordinator and Participants get lost
- Goals:
  - A commit protocol should guarantee **atomicity**, **consistency** and **durability** of data in such failure situations
    - even if servers fail and have to recover from these failures, a distributed transaction must be finished in an orderly fashion
  - A commit protocol should be **non-blocking**
    - **Unilateral abort**: none of the servers involved in a distributed transaction (coordinator and participants) should have to wait until a crashed server has fully recovered, but should be able to **unilaterally abort**

Handling Failure Situations Robustness of 2PC

- 2PC works in failure situations because
  - participants save their state (manipulations on data) in permanent storage as a *preparation* for commit
  - This preparation enables recovery in case of system failure
- If a participant recovers from a crash, it can continue from this saved state and complete the interaction with the coordinator
- Participant may retrieve last vote on transaction from Coordinator

Participant asks Coordinator



- After recovery, a participant has to ask Coordinator for its decision
  - Participant sends a "getDecision" message to Coordinator
  - Participant will act according to Coordinator decision

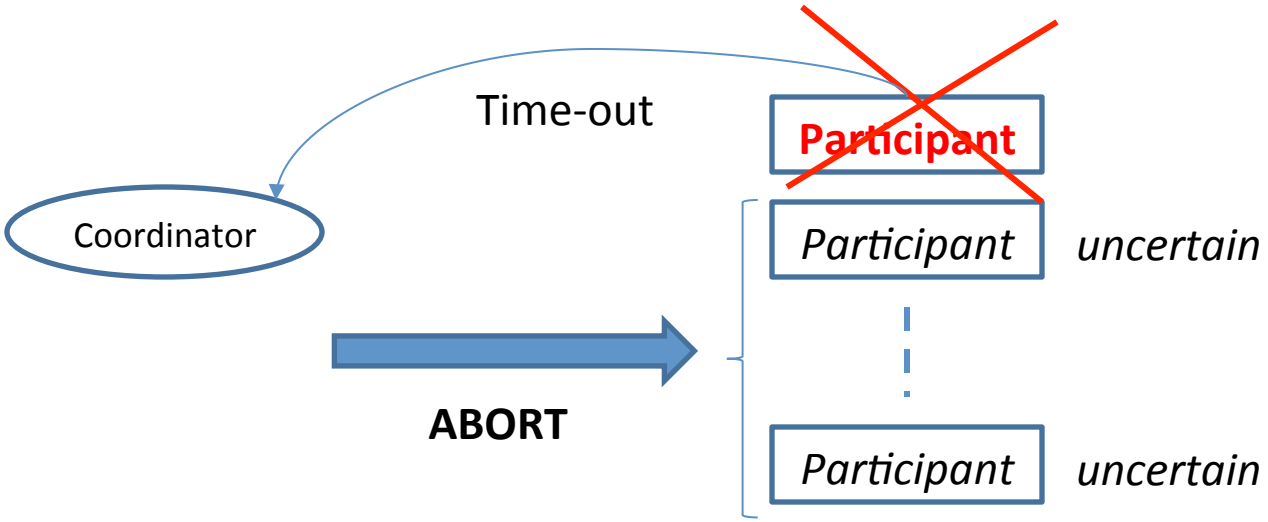
Two-phase Commit Protocol

- As long as Coordinator is alive, distributed transactions can be aborted fast and restarted
  - Coordinator will abort distributed transaction in case of failure
  - When the failed participant recovers, it may ask coordinator about decision (which was abort)
- **HOWEVER: What if the coordinator crashes ?**

Failure Situations

- Failure Situations
  - Participants fail and become unresponsive
  - Coordinator fails and becomes unresponsive
- Participants fail:
  - These are non-critical – they are “non-blocking”, as no other server has to wait for a participant’s recovery
  - Coordinator will abort transaction
- Coordinator fails:
  - This is a critical situation – Participants are left in an uncertain state: how can they get the information **whether to commit or abort**?
  - in the worst case, participants have to wait until the coordinator recovered – **Blocking Situation**

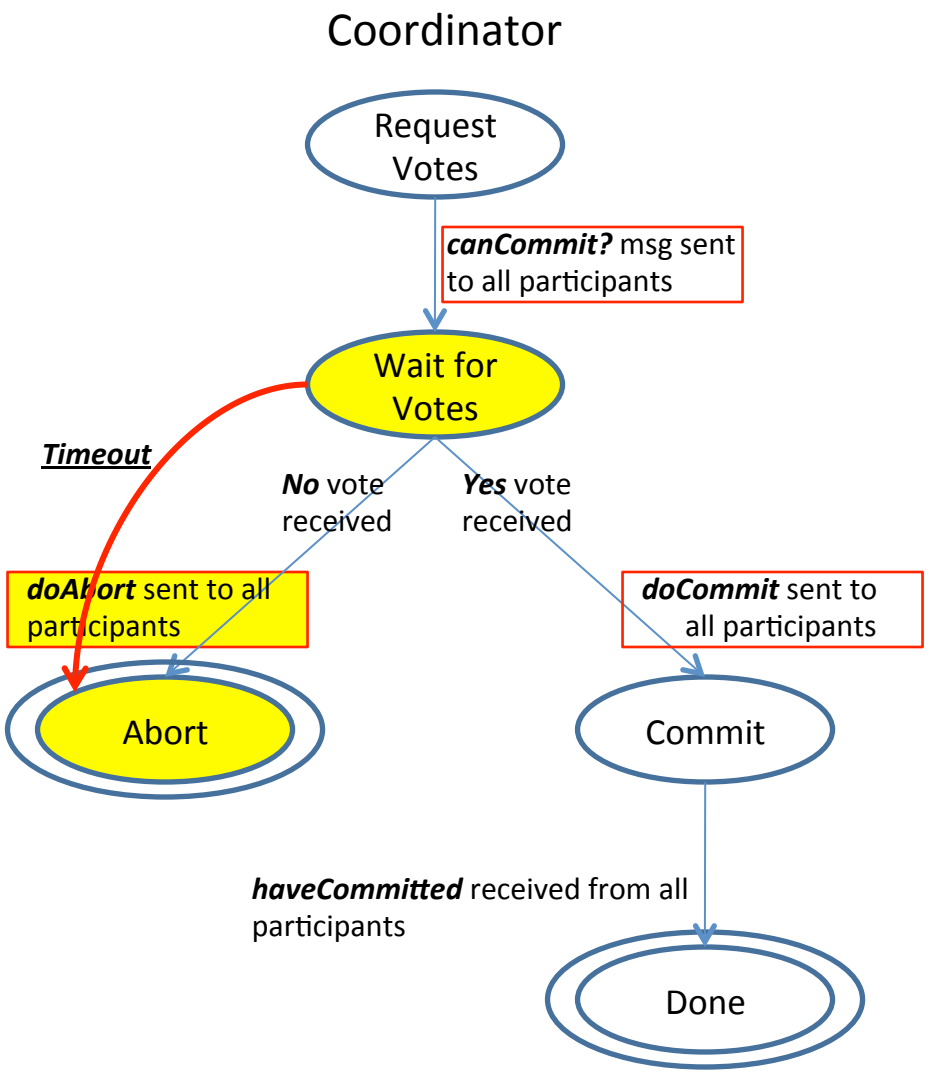
Participant fails in Phase 1



- Coordinator will timeout, as it does not receive vote from this participant
  - will send ***doAbort*** messages to all remaining participants
- "Non-blocking": Coordinator and remaining participants do not have to wait for failed participant to recover
  - when a failed participant recovers later, it can ask the coordinator for its decision (which was an abort) and **abort** itself after recovery

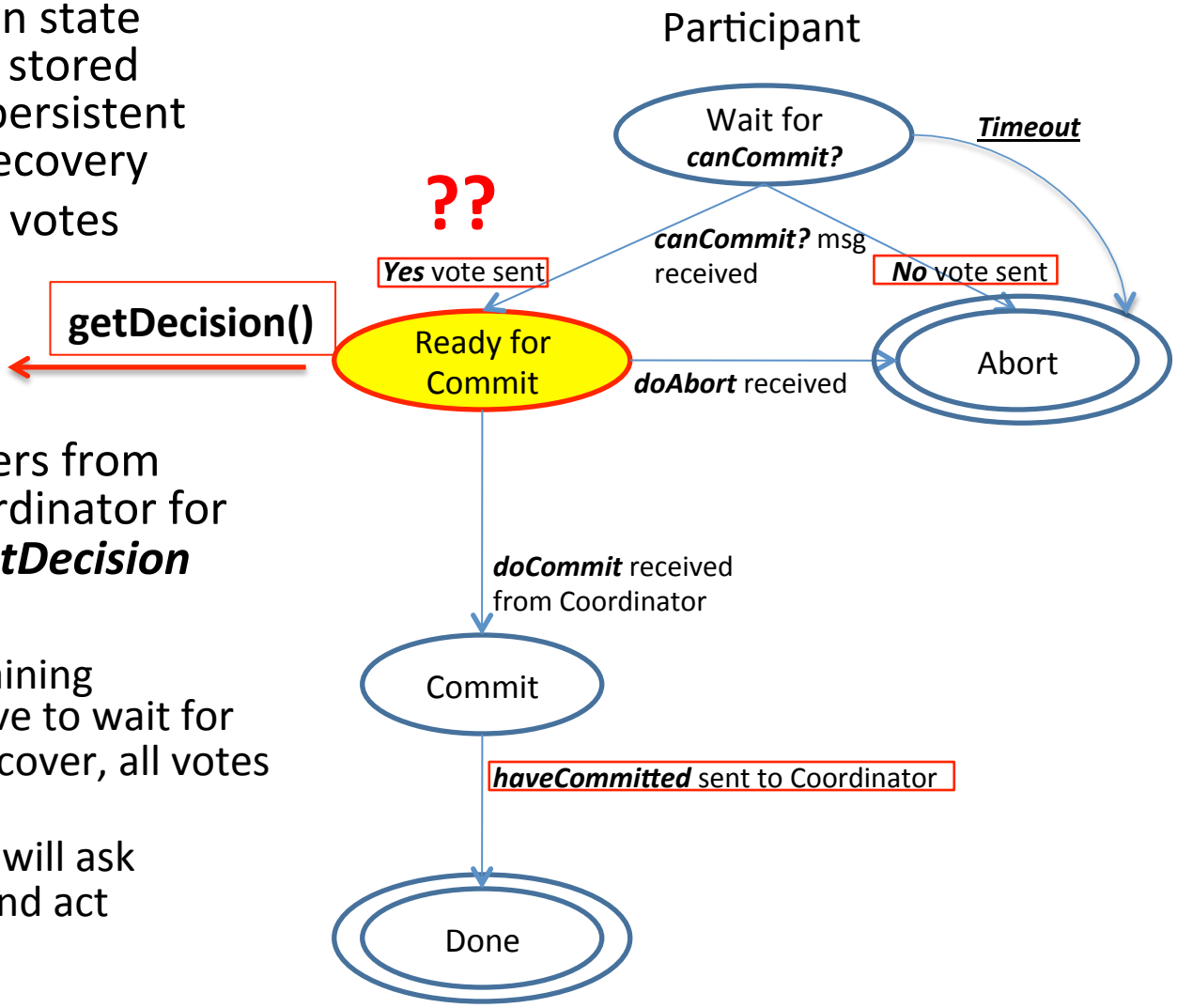
Participant fails in Phase 1

- Coordinator time-out in phase 1
  - No vote received from at least one participant after timeout
    - This is regarded as an indication to abort
  - Coordinator decides to abort the transaction
  - Coordinator sends "doAbort" to participants and aborts unilaterally
  - Coordinator will ignore subsequent votes
- If failed participant recovers it has to ask coordinator for its decision and will abort as well



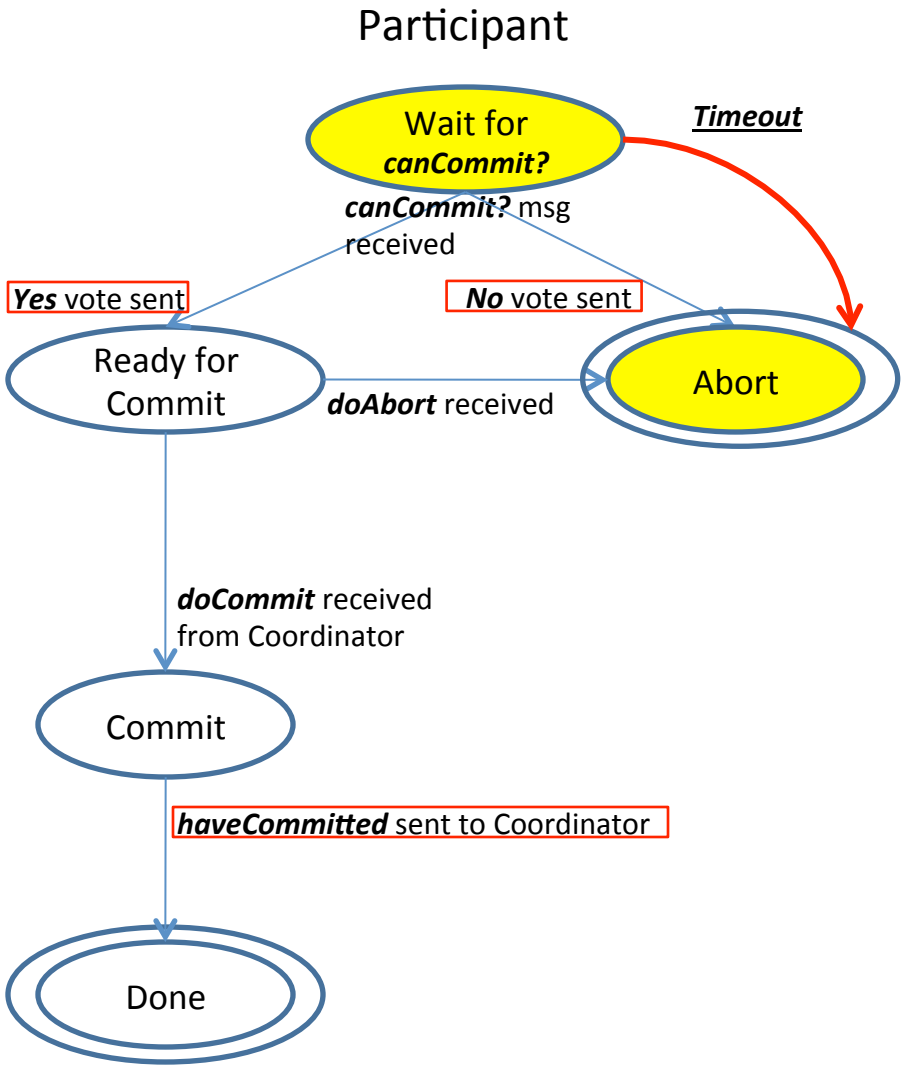
Participant fails in Phase 2

- Participant is in uncertain state "Ready to Commit", has stored intermediate results in persistent storage to prepare for recovery
- Coordinator received all votes
- **Participant fails**
- When participant recovers from failure, it has to ask coordinator for its decision by calling ***getDecision***
- "Non-blocking":
  - Coordinator and remaining participants do not have to wait for failed participant to recover, all votes have been received
  - Recovered participant will ask coordinator decision and act accordingly

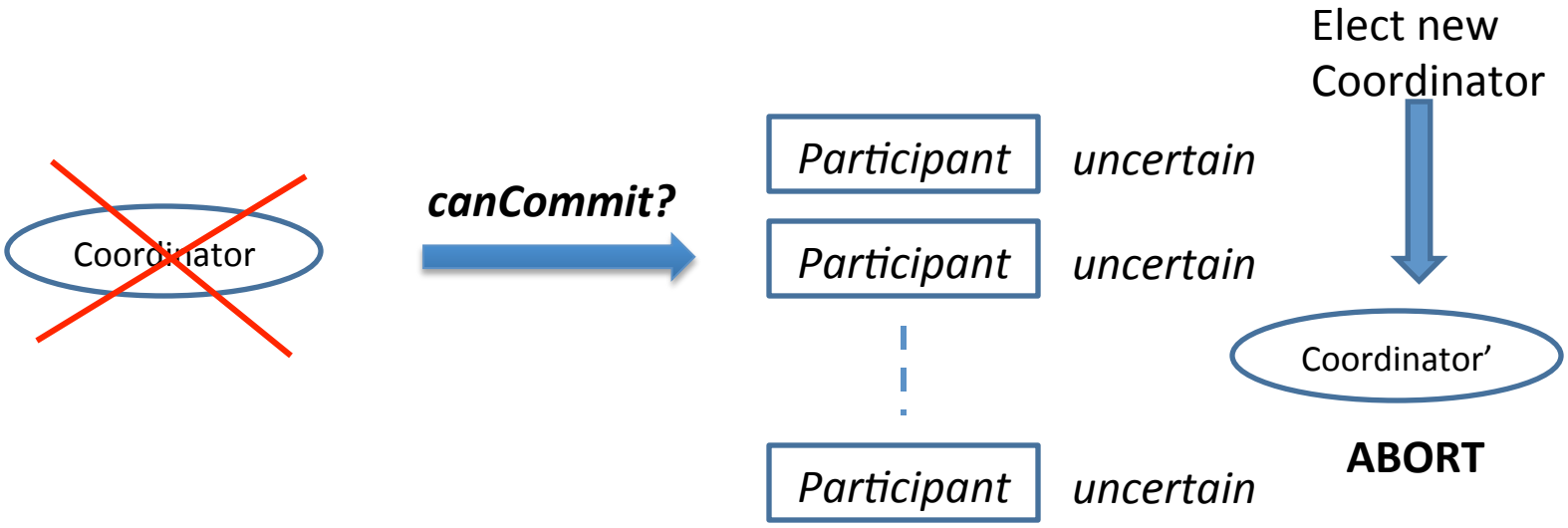


Coordinator fails before Phase 1

- Participant time-out in phase 1
  - Participant finishes its part of the distributed transaction
  - Coordinator is late in sending a request-for-vote (phase 1):
  - the participant reaches its timeout and will unilaterally abort



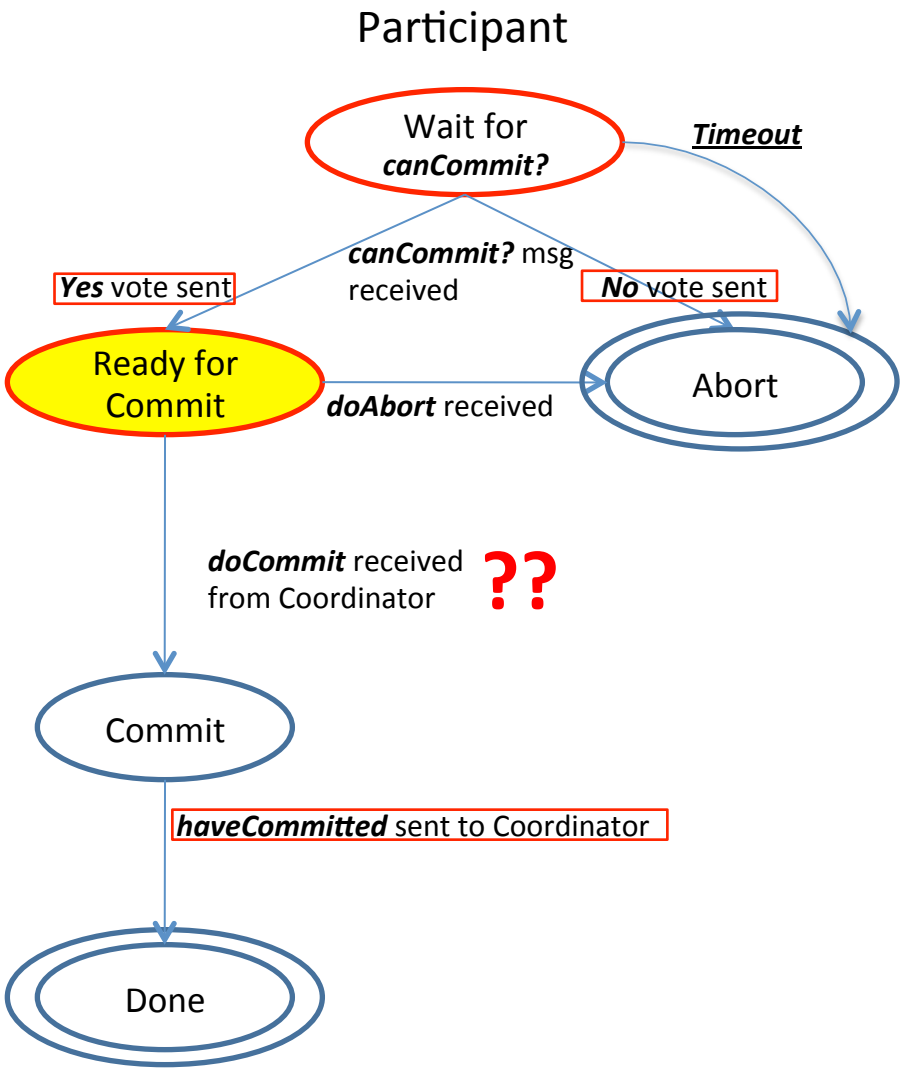
Coordinator fails in Phase 1



- Coordinator fails in Phase 1:
  - Coordinator sent ***canCommit?*** (request for votes) to participants
  - failed after that – cannot receive any votes from participants
- Consequence: Participants will not receive a doCommit or doAbort message and remain in an uncertain state

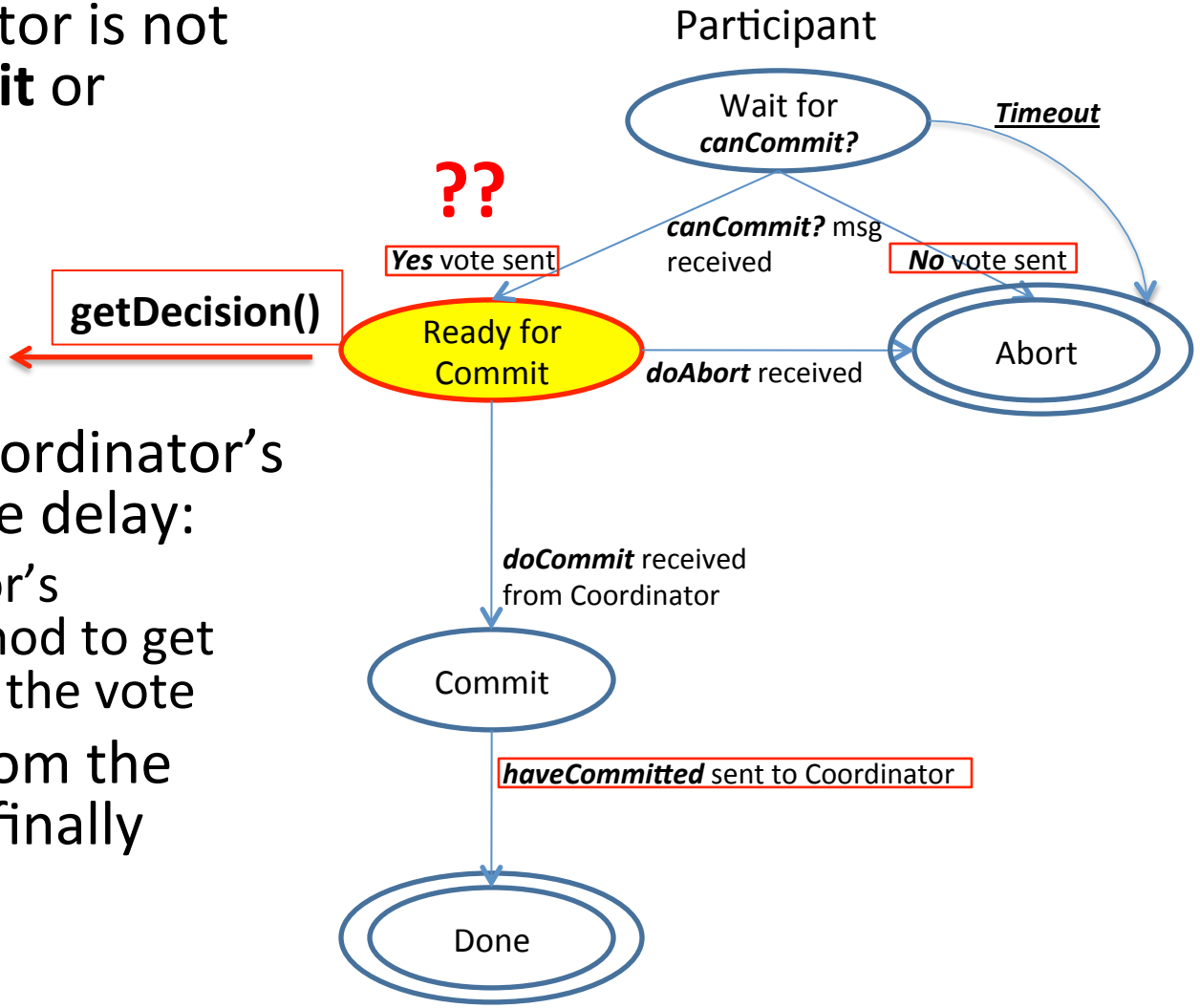
Coordinator fails in Phase 1

- Coordinator fails in Phase 1: Coordinator sent ***canCommit?*** (request for votes) to participants
- Participant received the request for vote in phase 1, is prepared to commit
- Participant **does not receive a doCommit message from the Coordinator**
  - Participant is prepared for commit (in phase 2), but *uncertain* about the outcome of the voting – it cannot proceed, therefore, objects remain locked



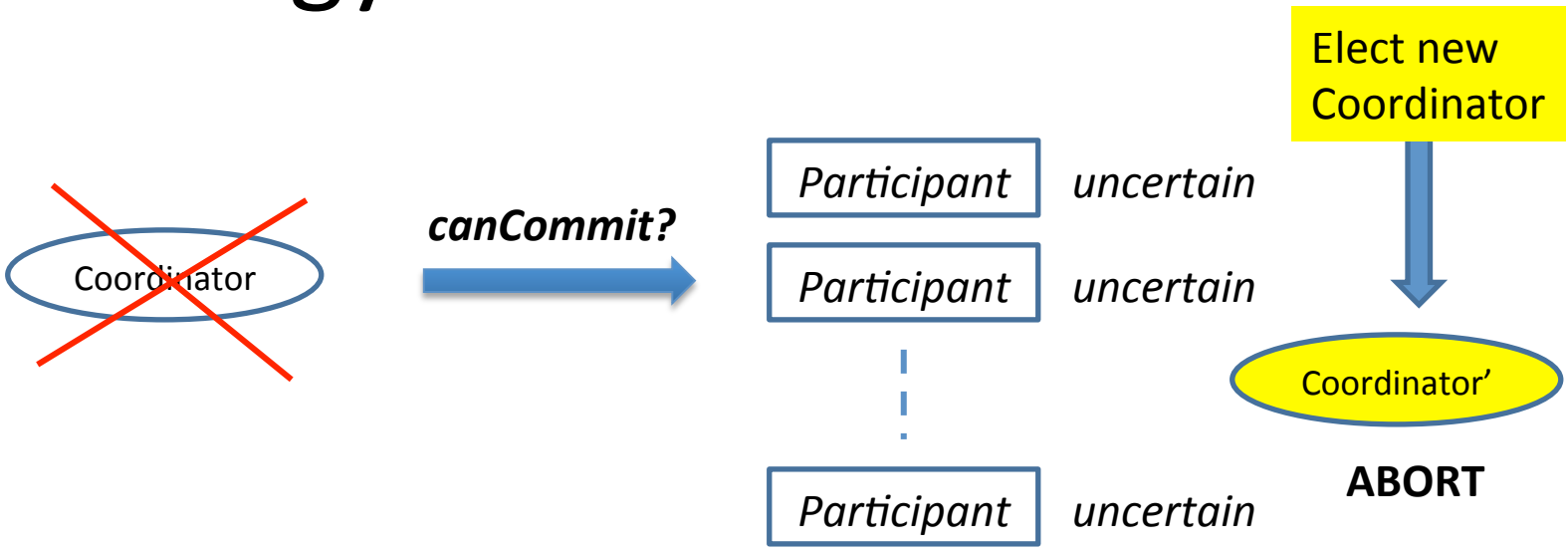
Strategy: Ask for Decision

- Situation: Coordinator is not sending a **doCommit** or **doAbort** (phase 2)



- Strategy: Ask for coordinator's decision after a time delay:
  - call the coordinator's getDecision() method to get information about the vote
- If there is a reply from the coordinator, it can finally commit or abort

Strategy: Elect a new Coordinator



- What if the coordinator has failed after sending a ***coCommit?*** (request for votes) message in phase 1?
  - Participants cannot contact the coordinator, must wait until it is recovered/replaced
  - In the meantime, all locks on data objects involved in the distributed transaction cannot be released
- Participants may elect a new coordinator among them that sends abort messages to the remaining participants

Coordinator fails in Phase 1 - Recovery

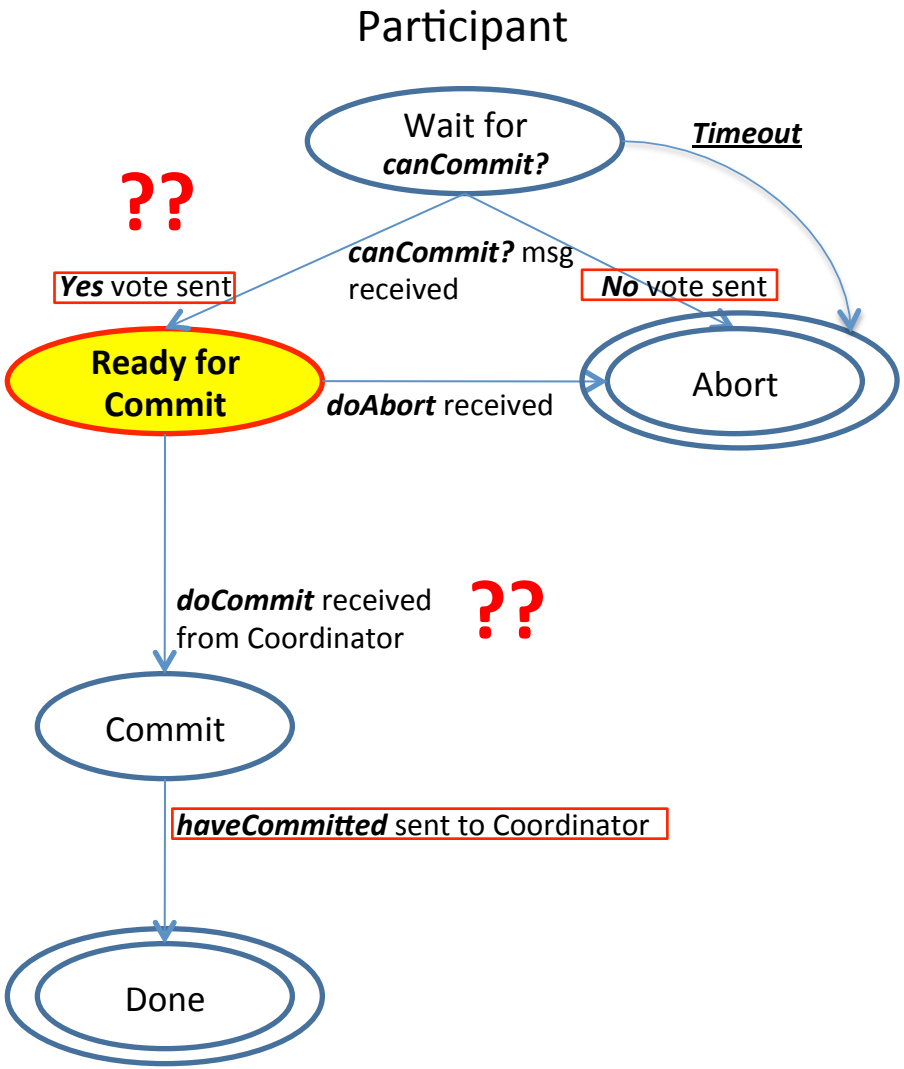
- Recovery:
  - participants elect new coordinator and restart 2PC protocol
    - New coordinator will now send new ***request for vote*** messages (canCommit?) to participants
  - If all Participants are alive:
    - They will send their vote to the new coordinator
  - If one or more participants failed and are not recovered
    - the new coordinator will not receive votes from these participants – it will **timeout** and send ***doAbort*** to the remaining alive participants
  - If the new coordinator also fails, the participants may elect yet another new coordinator and restart the 2PC protocol

Coordinator fails in Phase 2

- Participants sent their vote, they are in an *uncertain* state – have to wait for the coordinator to send its decision in phase 2
  - All manipulated data will remain locked
- Coordinator received all votes and makes a decision to either commit or abort the transaction, then fails
- Uncertain situation:
  - Has the coordinator sent a **doCommit** or **doAbort** before it failed?
  - Has at **least one of the Participants** received this message?

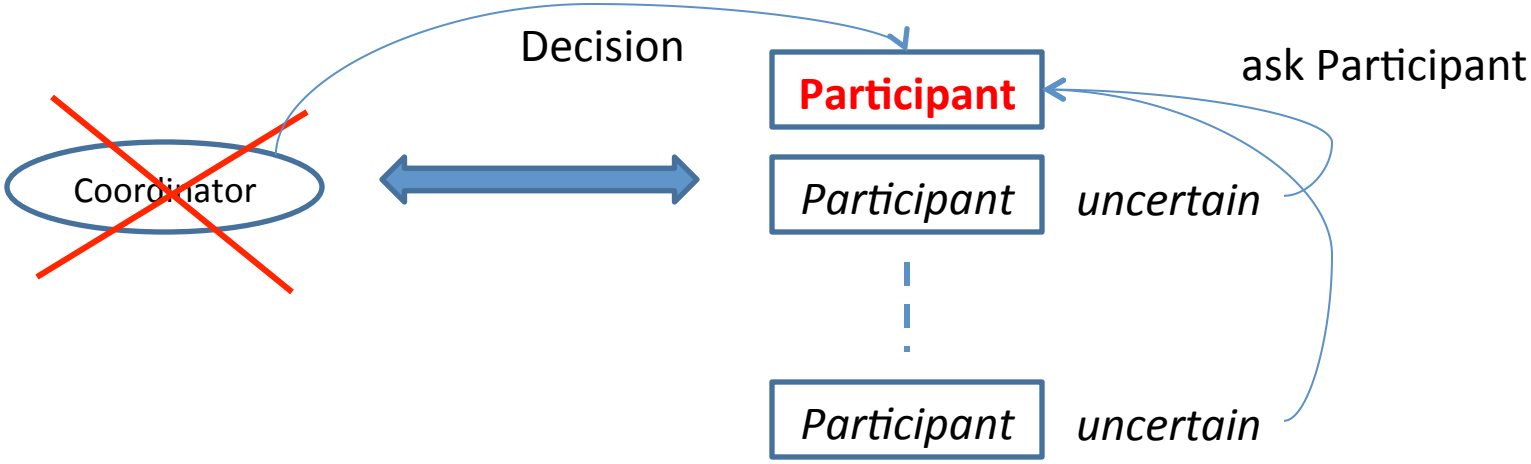
Coordinator fails in Phase 2

- If coordinator fails in phase 2
  - It may have sent a **doCommit** or **doAbort**, before it crashed
  - **One of the Participants may have received this information**
    - Others can ask this Participant for the Coordinator's decision
  - If none of the Participants received a message from the failed coordinator
    - They can wait for Coordinator to recover
- Question: can participants elect a new Coordinator that restarts the 2PC protocol by sending out new requests for a vote?



Coordinator failed in Phase 2

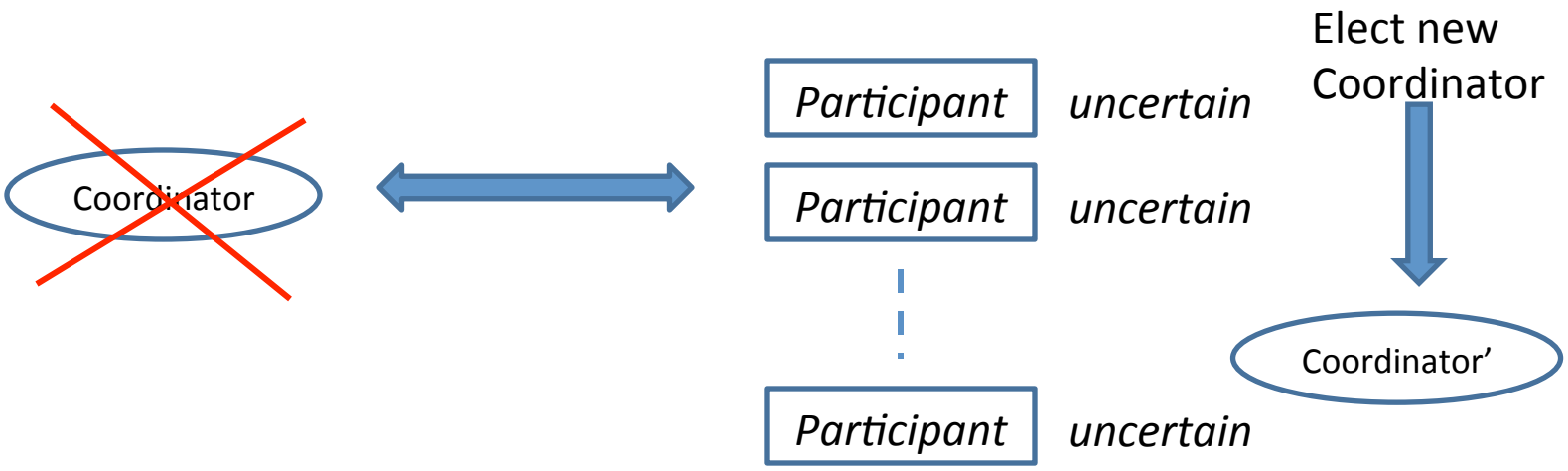
Scenario 1: At least one Participant knows Decision



- Scenario 1: at least one of the alive participants knows the decision of the failed coordinator – it received a ***doCommit*** or ***doAbort*** message
- Recovery: the participant can convey this information to other participants, so that they can commit or abort
  - "Non-blocking": participants do not have to wait for the coordinator to recover

Coordinator fails in Phase 2

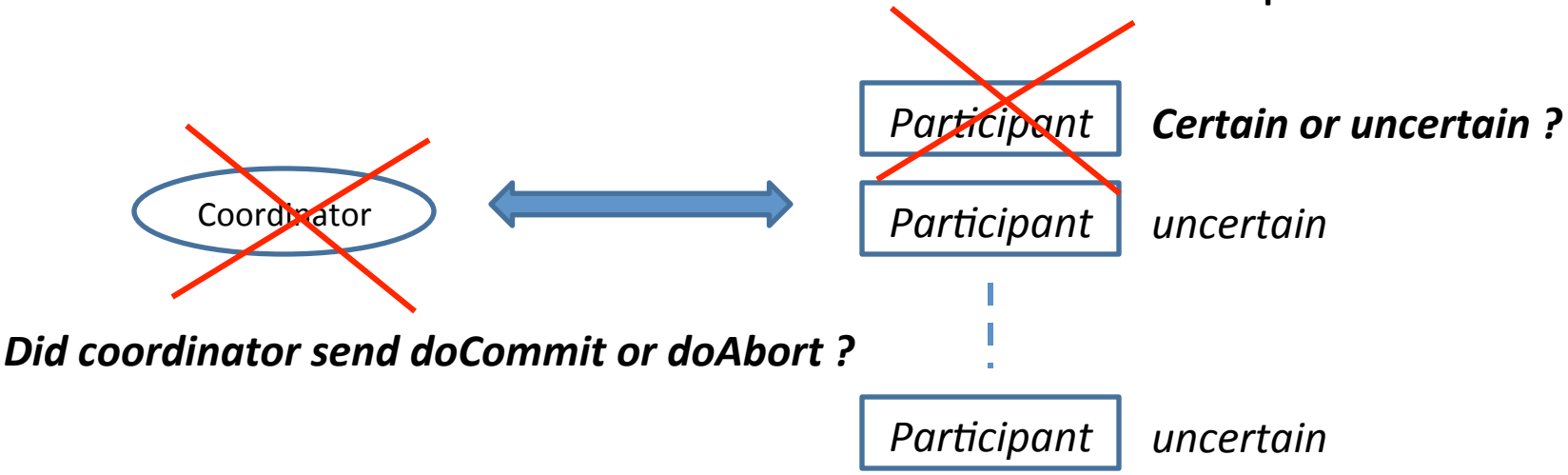
Scenario 2: none of the alive Participants knows Decision



- Scenario 2: **all participants are alive** and none of them knows the coordinator's decision
- Possible Solution:
  - Participants elect a new coordinator, restart the 2PC protocol

Coordinator fails in Phase 2

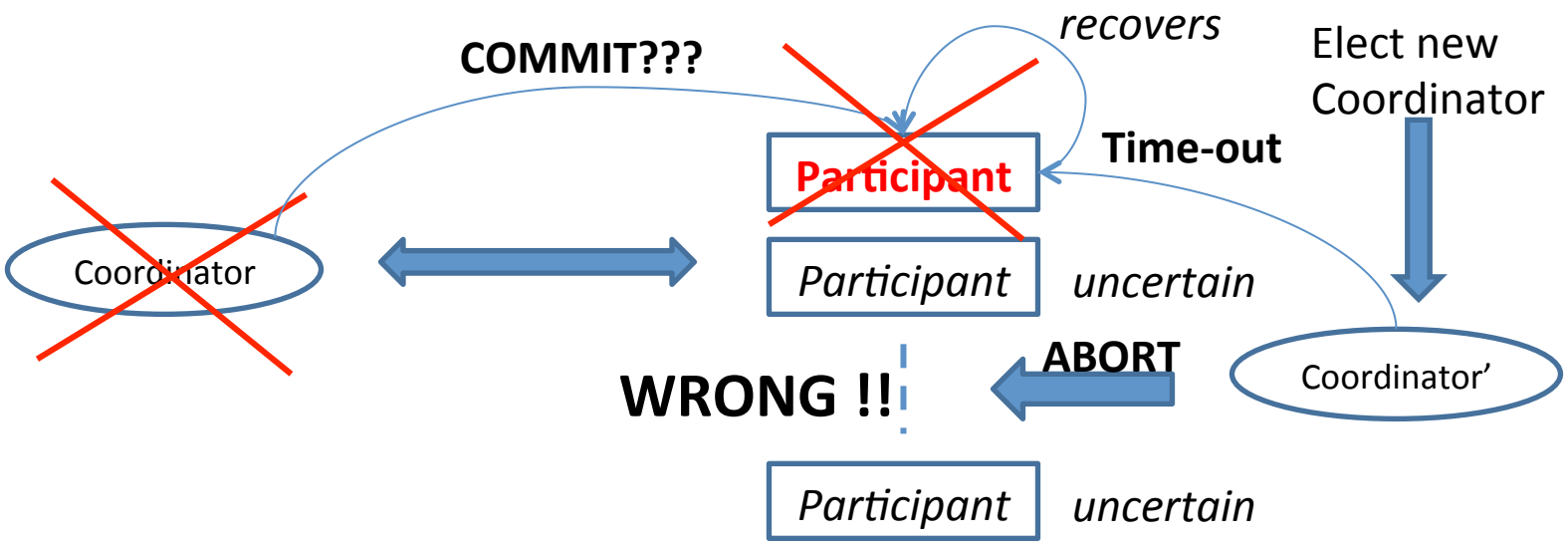
Scenario 3: Both the Coordinator and **one** Participant fail



- **Problem: what if one participant crashed together with the coordinator?**
- Coordinator received all votes and makes a decision to either commit or abort the transaction, then fails
  - Has the coordinator sent a **doCommit** or **doAbort** before it failed?
  - **Has at least one of the Participants received this message**?
- Problem:
  - the rest of the participants may elect a new coordinator who will **abort** the transaction
  - what if the crashed participant received a **doCommit** and will commit after recovery?

Coordinator fails in Phase 2

Scenario 3: Both Coordinator and one Participant fail



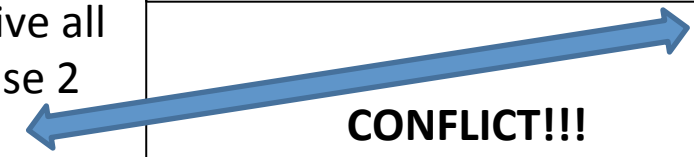
- Coordinator failed, one participant failed, **no alive participant knows** the coordinator's decision
- Problem: The coordinator's decision is **unknown** and it is unknown whether the crashed participant received a decision from the coordinator
  - The remaining participants may elect a new coordinator among them and vote – this new coordinator will time-out as the crashed participant does not deliver a vote and ***make an ABORT-decision***
  - **BUT:** If the crashed participant received a **COMMIT-decision before from the old coordinator**, it ***will commit data manipulations <u>after recovery</u>***
- **Possible CONTRADICTION !!**

Coordinator fails in Phase 2

Scenario 3: Both the Coordinator and one Participant fail

- Most critical case: both the coordinator and one participant failed
  - The failed Participant may have received a doCommit or doAbort, but which one? As it crashed, it cannot inform other participants
- If we assume that situation is resolved by electing a new coordinator / restarting the 2PC protocol:

• Participants elect new Coordinator • New Coordinator restarts 2PC and sends canCommit? in phase 1 • the failed Participant cannot vote • the new coordinator does not receive all votes and will send a **doAbort** in phase 2 • the Participants alive will **abort**	• Failed Participant recovers – we assume it received a **doCommit** from the original failed Coordinator • Recovered Participant **commits**



CONFLICT!!!

- This is a situation where a new Coordinator **cannot** be elected !!
- **Participants have to wait until original Coordinator recovers** and continues to send doCommit or doAbort to the rest of the participants!!

Problem in System Failure Situations

- 2PC is a *blocking* protocol in failure situations:
  - 2PC can cause considerable delays to participants in an *uncertain* state, this occurs when the coordinator fails
  - Locks on data objects remain in place as the transaction cannot finish
- Three-phase commit protocols have been designed to prevent delays due to coordinator or participant failure, but the cost is higher in the failure-free case (more messages required).

Three-Phase Commit Protocol

3-Phase Commit Protocol

- 3PC is non-blocking in failure situations
- 3PC has three phases

Voting phase

Coordinator asks participants whether they are ***prepared*** **to commit** or **abort**

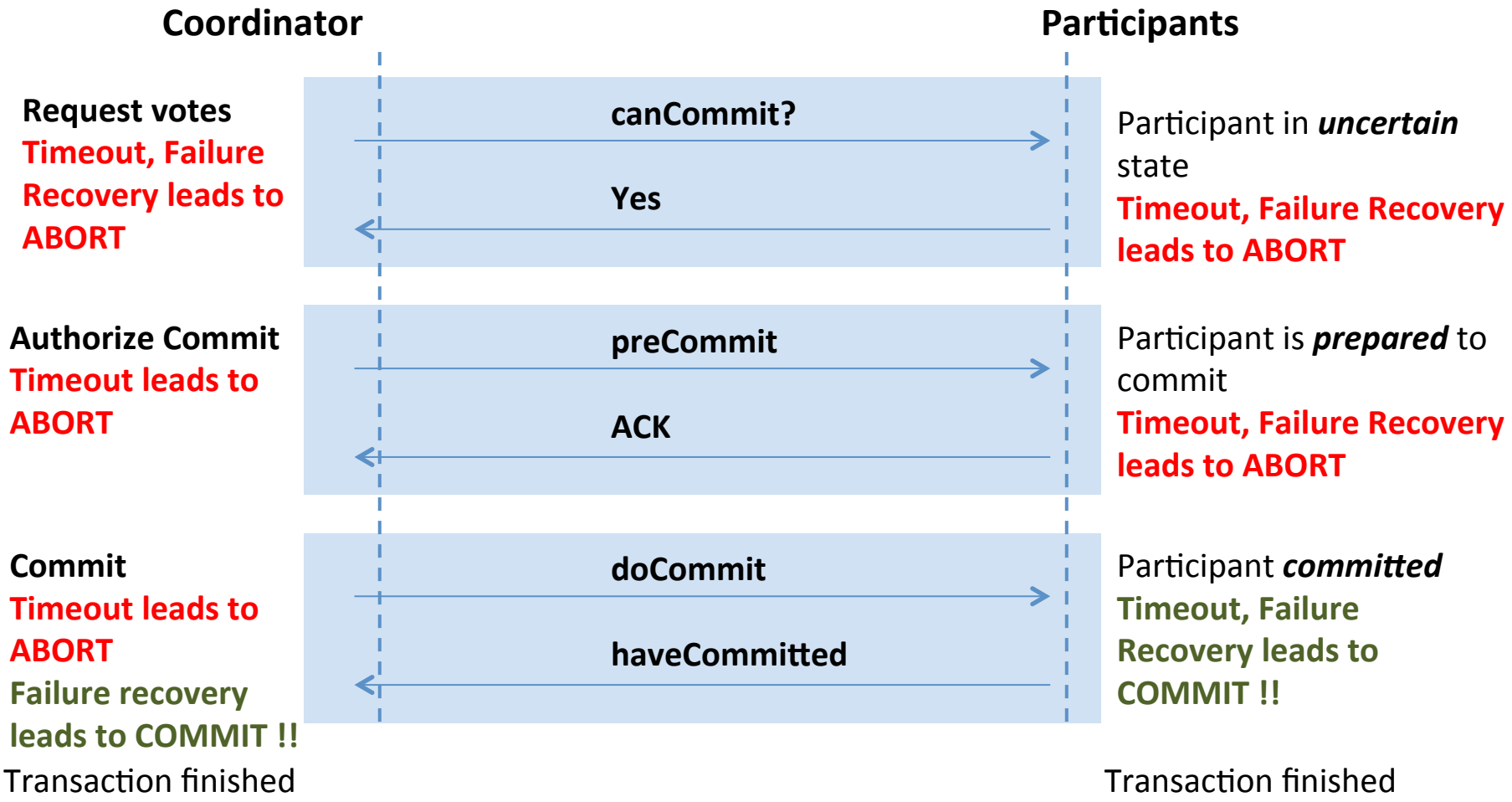
Pre-Commit phase

If all votes are for commit, coordinator asks participants to ***prepare for commit***
If at least one vote is for abort, coordinator asks all participants to ***abort***

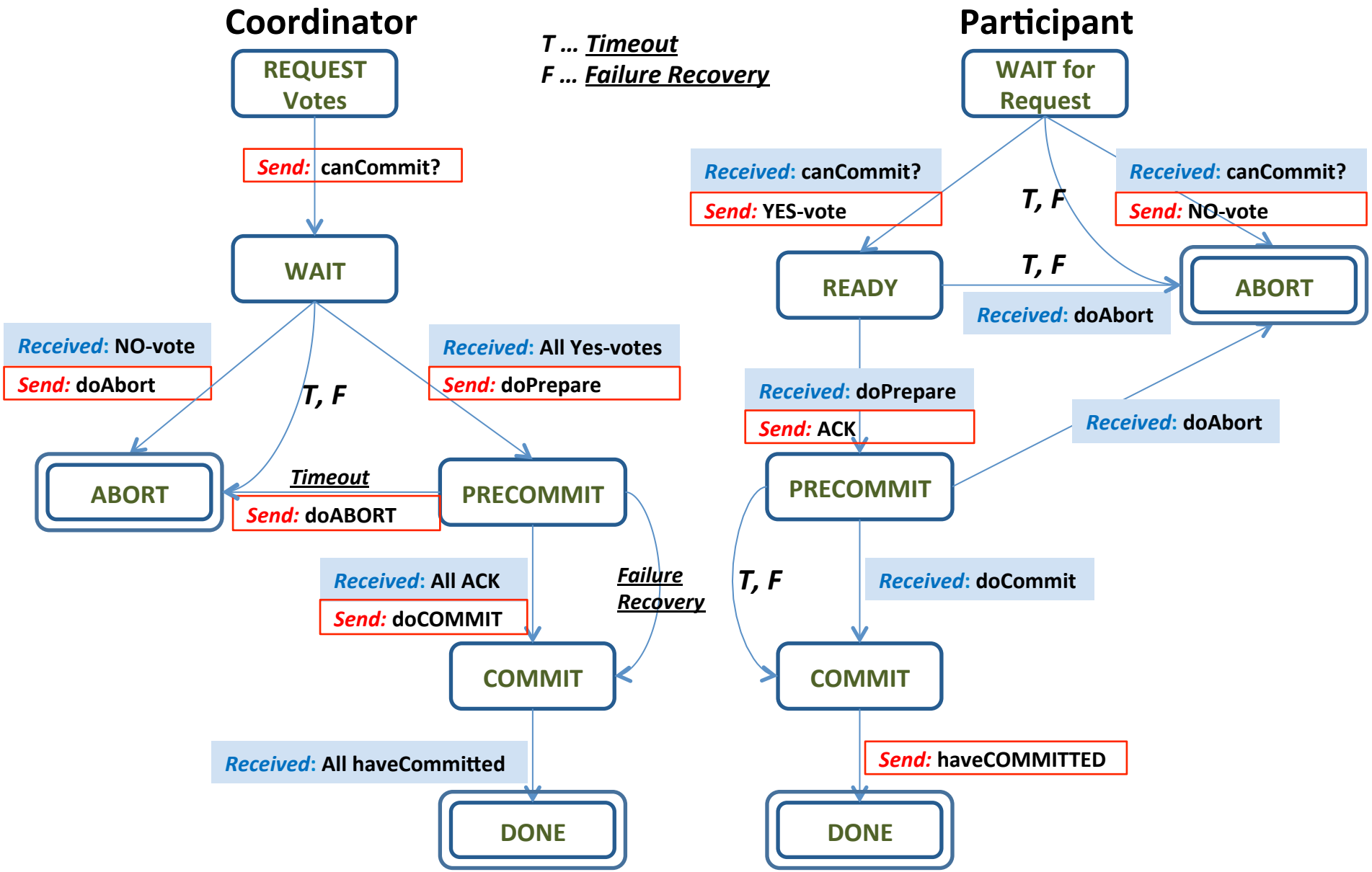
Completion phase

If all participants **acknowledge** that they are ***prepared***, coordinator asks participants to ***commit***
If at least one participant fails to send **ACK** message that they are prepared,
coordinator asks all participants to ***abort***
If coordinator and participants are ***prepared***, **they can commit**, even in case of failure or timeout

3-Phase Commit

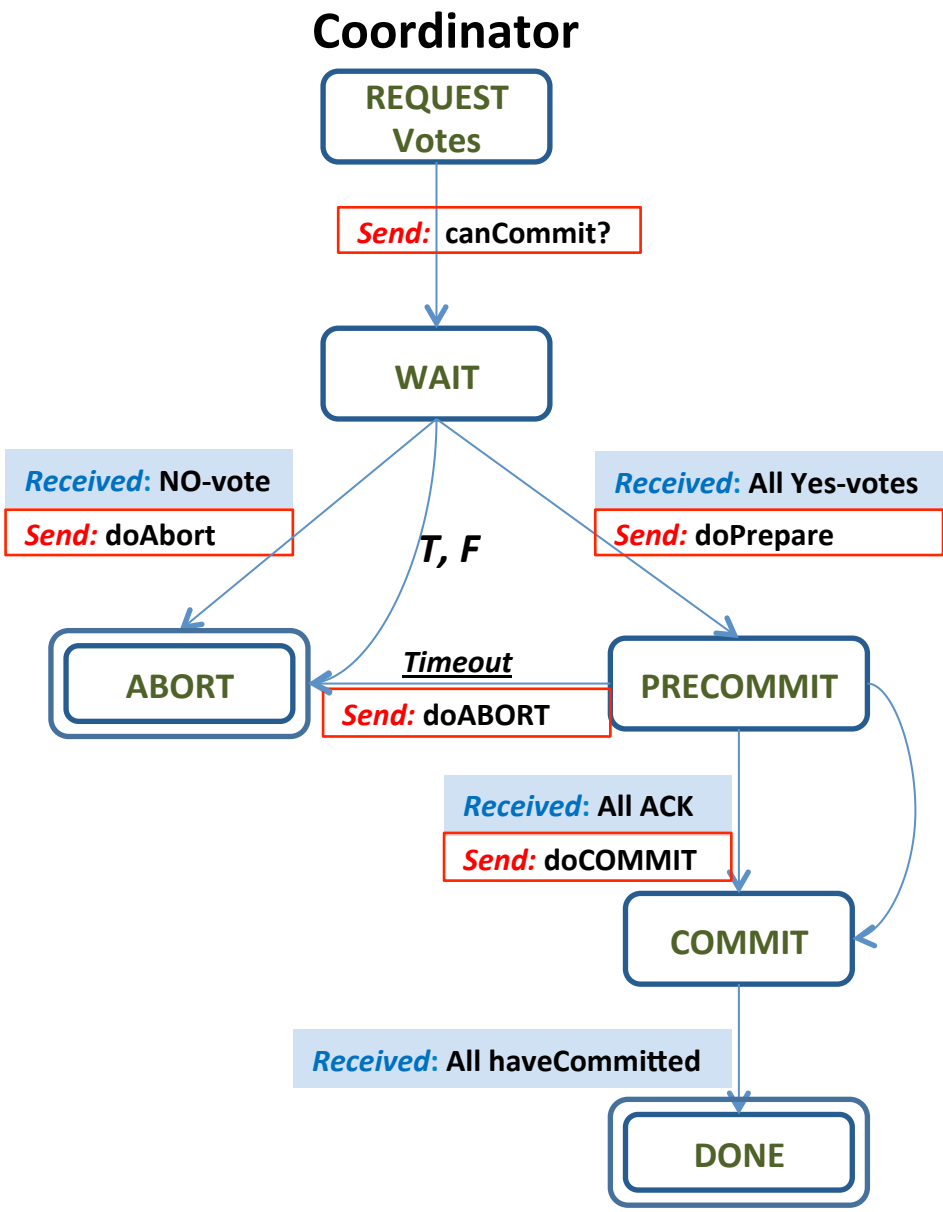


3-Phase Commit



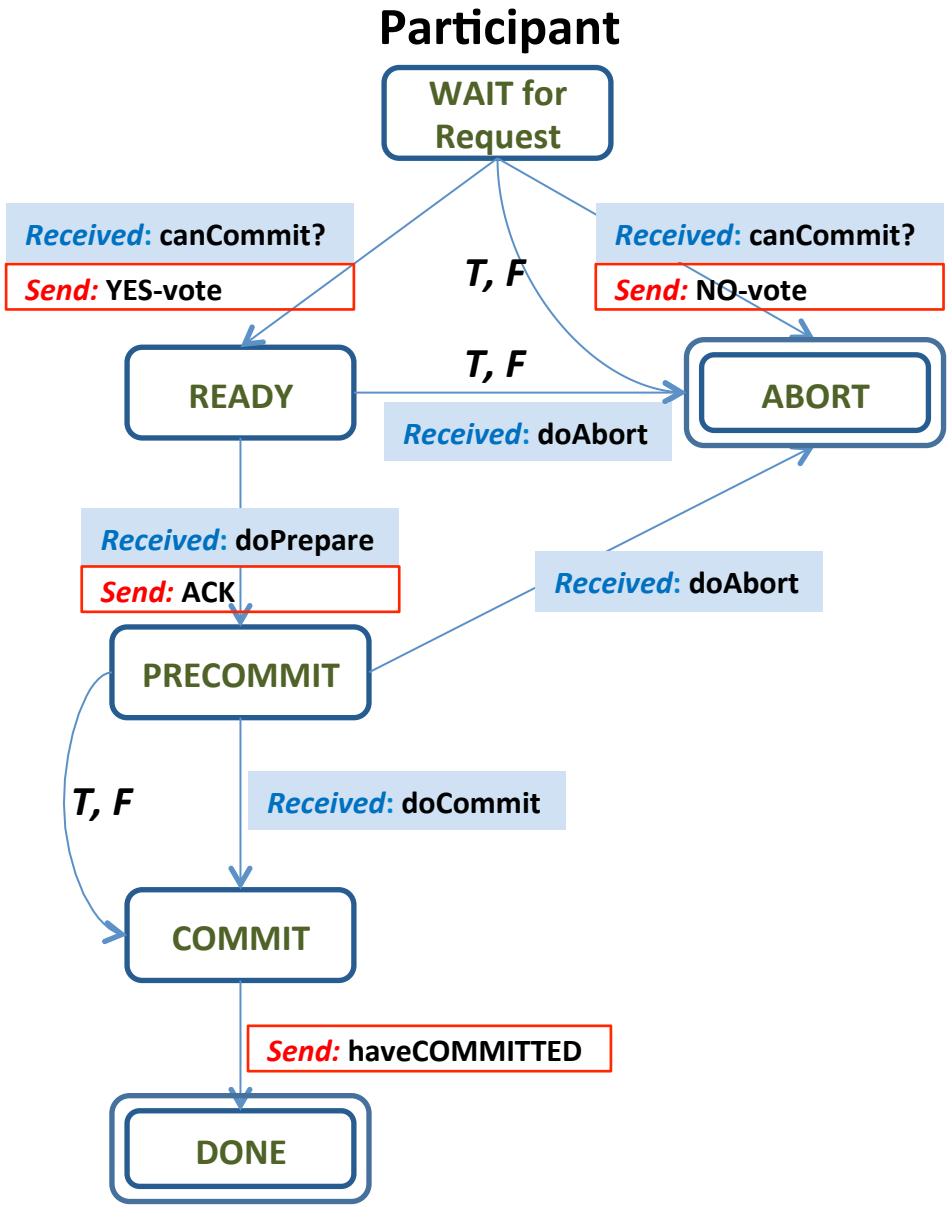
3-Phase Commit Protocol

- Coordinator
  - Phase 1
    - Coordinator sends requests for votes to participants
    - Coordinator waits for votes
  - Phase 2
    - If coordinator receives at least one vote for abort, it will abort transaction
    - If coordinator reaches timeout, it will abort transaction
    - If all participants vote for commit, coordinator will send a doPrepare message and transition into the PRECOMMIT state
  - Phase 3
    - If coordinator receives ACK messages from all participants (acknowledging that they are all in the prepared state), it will send out the doCommit message



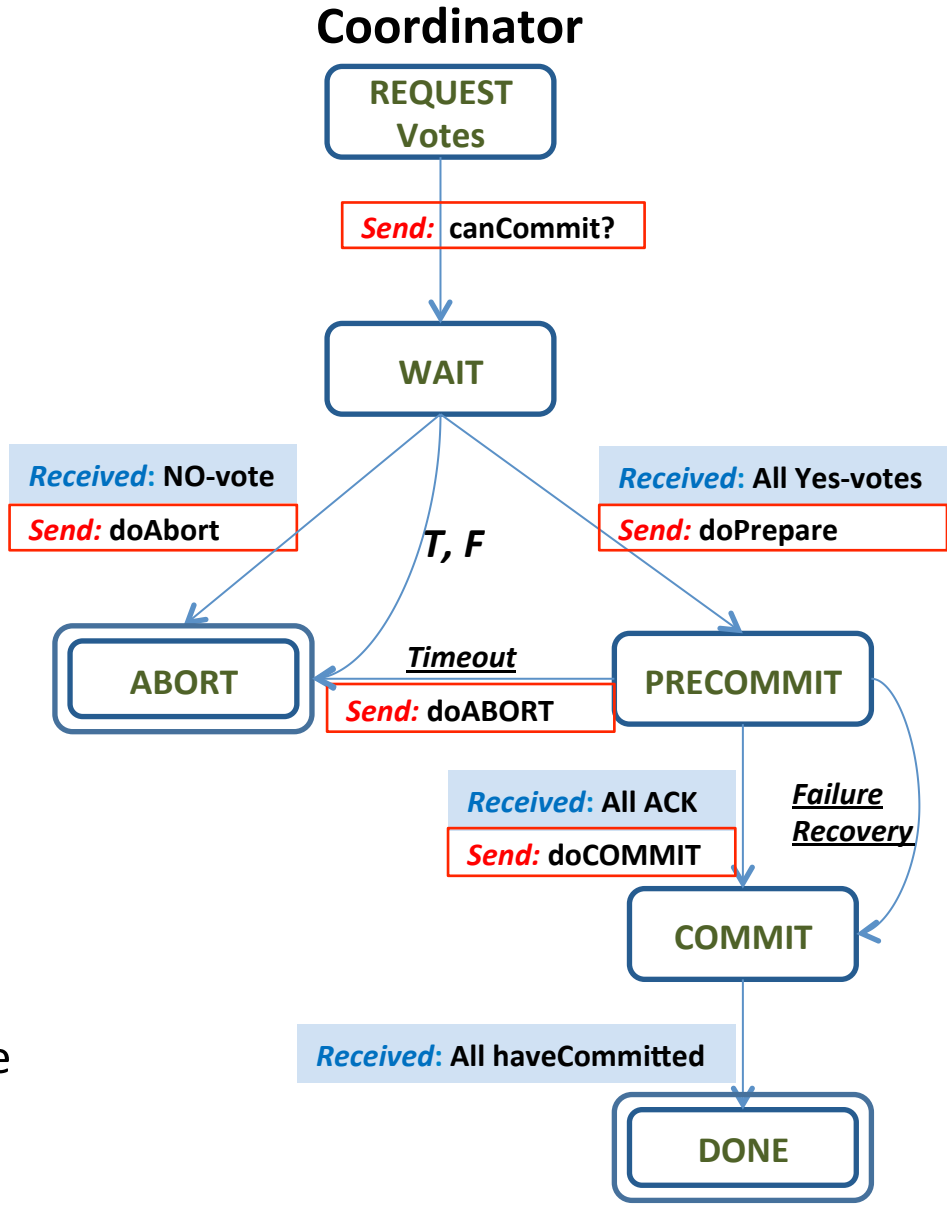
3-Phase Commit Protocol

- Participants
  - Phase 1:
    - Participant receives a request for vote
    - If participant decides to commit, it will move to the READY state and send a commit vote
    - If participant decides to abort or reaches a timeout, it will abort
  - Phase 2:
    - Participant is in READY state, waits for preCommit message
    - If participant receives a preCommit message, it will move to prepared state and send an ACK message back to coordinator
  - Phase 3:
    - Participant is in PRECOMMIT state, waits for doCommit message
    - If participant receives doCommit message, it will commit its local transaction and send haveCommitted message back to coordinator



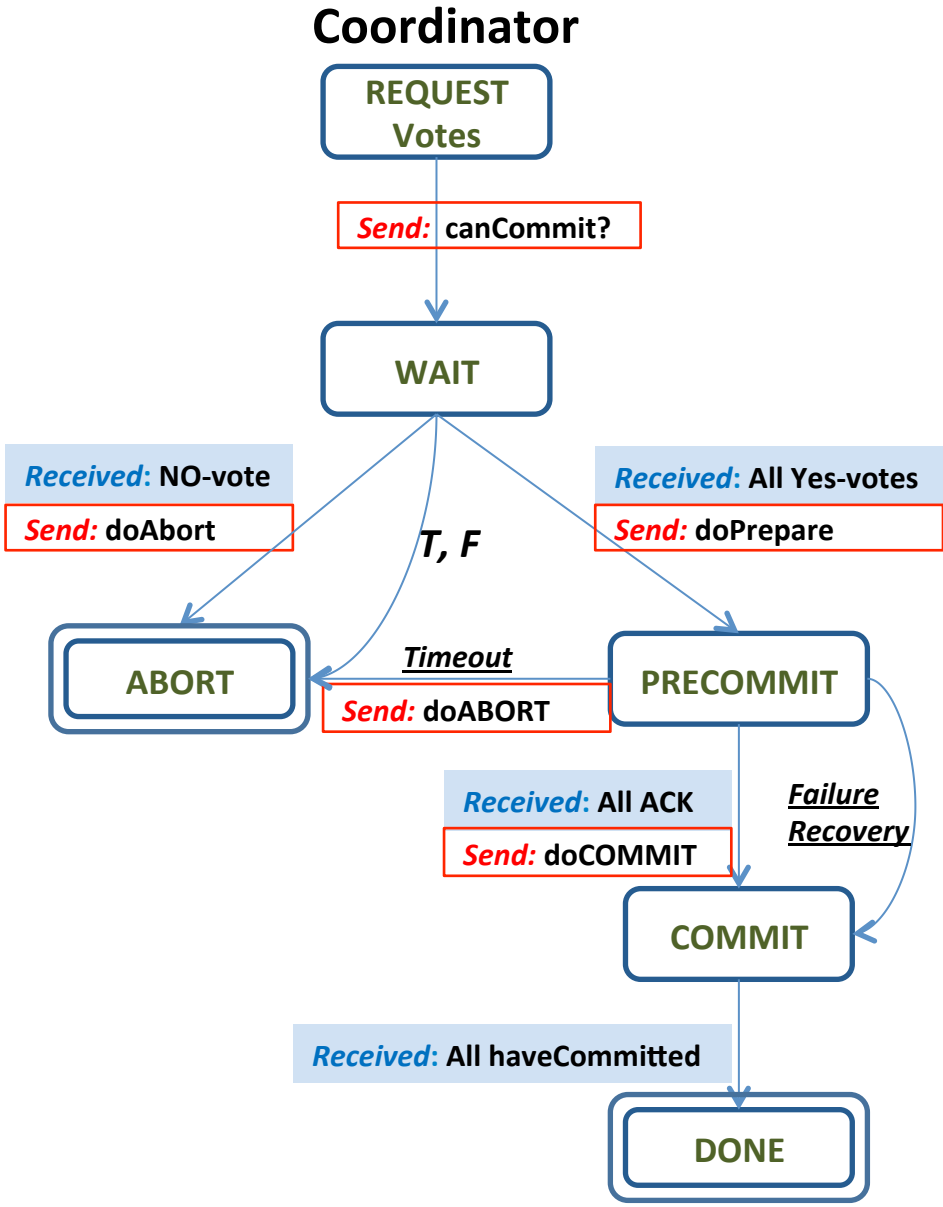
Timeout and Failure Recovery

- Coordinator Phase 1
  - State "WAIT"
    - Timeout: Participants are late with votes
      - Coordinator will send doAbort messages to all participants
      - Coordinator aborts
- Coordinator Phase 2
  - State "Wait": Coordinator received all YES-votes
    - Coordinator fails before sending doPrepare message
      - After Recovery: Coordinator aborts local transaction (no message send), participants time out waiting for doPrepare message and abort their local transaction as well



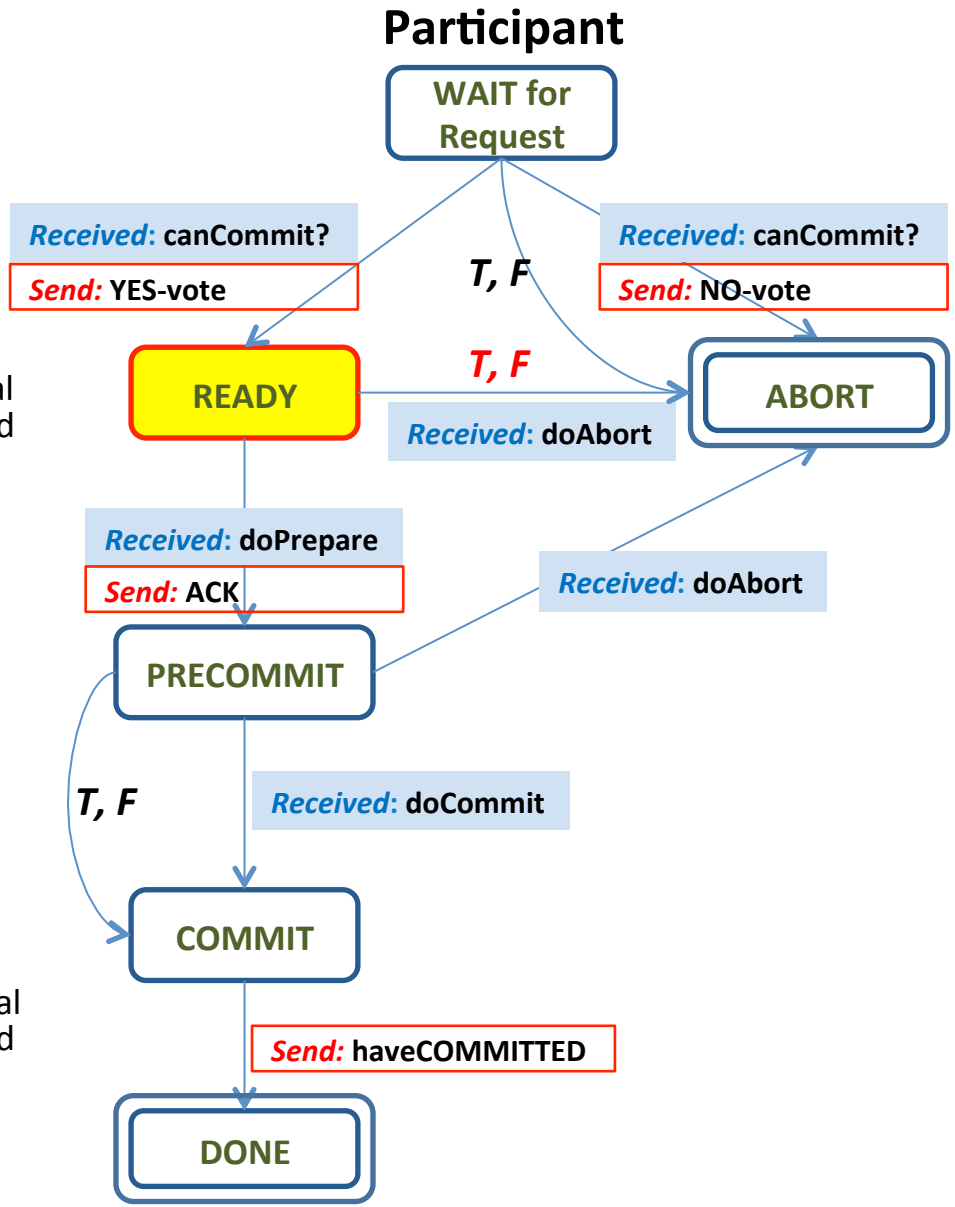
Timeout and Failure Recovery

- Coordinator Phase 3
  - State "PRECOMMIT": Coordinator sent doPrepare messages to participants
    - Timeout: Participants are late with ACK messages
      - Coordinator will send doAbort messages to all participants
      - Coordinator aborts
    - Coordinator fails before sending doCommit message
      - After Recovery: **Coordinator commits local transaction** (no messages sent to participants), participants time out waiting for the doCommit message and **commit** their local transaction as well



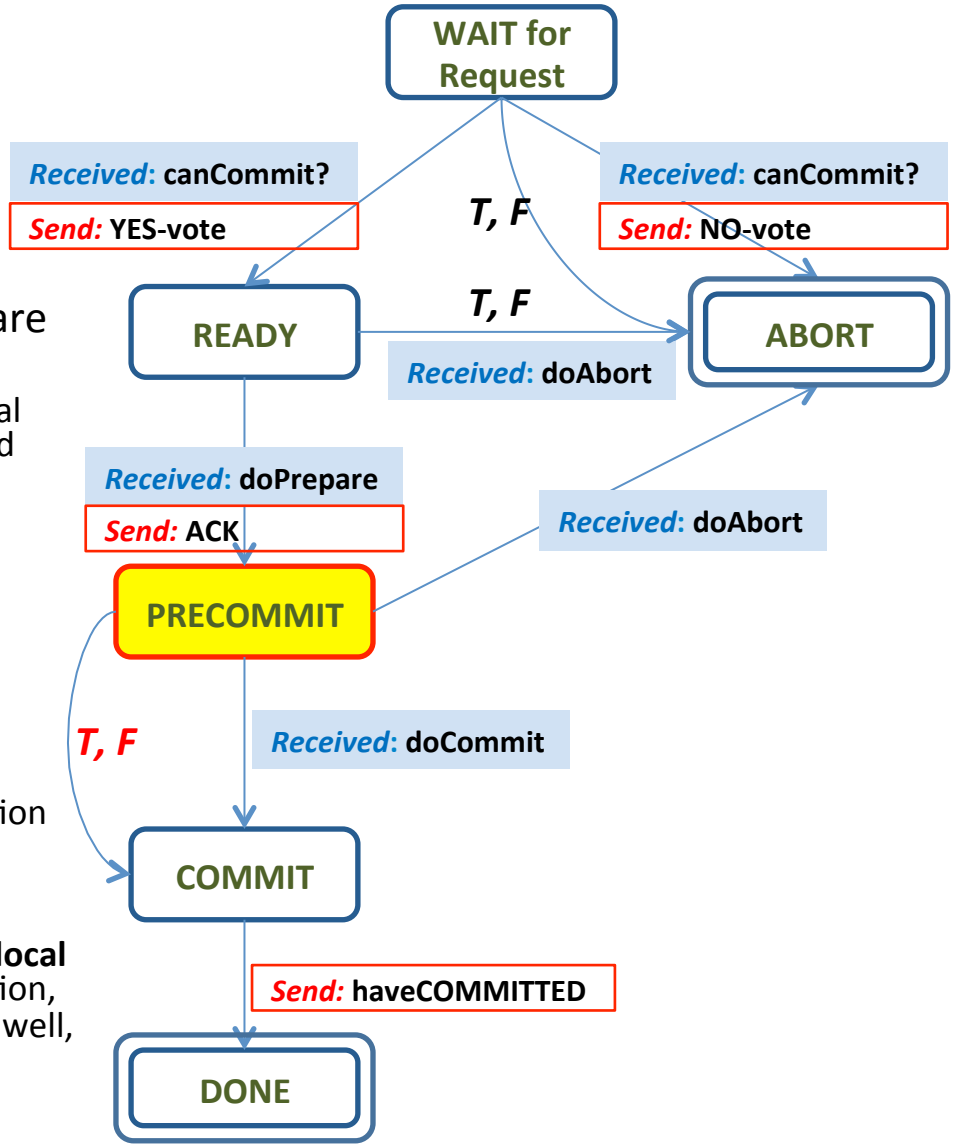
Timeout and Failure Recovery

- Participant Phase 1
  - State "WAIT for Request"
    - Timeout: Coordinator is late with request for vote
      - Participant will abort local transaction
    - Participant fails before receiving request
      - After recovery: participant will abort local transaction, coordinator will time out and send doAbort to remaining participants
- Participant Phase 2
  - State "READY": Participant has sent Yes-vote
    - Timeout: Coordinator is late with doPrepare message
      - Participant will time out waiting for doPrepare message and abort local transaction
    - Participant fails before receiving doPrepare message
      - After Recovery: participant will abort local transaction, coordinator will time out and send doAbort to remaining participants



Timeout and Failure Recovery

- Participant Phase 3
  - State "READY"
    - Participant fails before sending ACK message back to acknowledge doPrepare message
      - After Recovery: participant will abort local transaction, coordinator will time out and send doAbort to remaining participants
  - State "PRECOMMIT": Participant has sent ACK message
    - Timeout: Coordinator is late with doCommit message
      - Participant will time out waiting for doCommit message and **commit local transaction** without further communication
    - Participant fails before receiving doCommit message
      - After Recovery: **participant will commit local transaction** without further communication, coordinator will time out and **commit** as well, all other participants have received doCommit and will **commit** as well

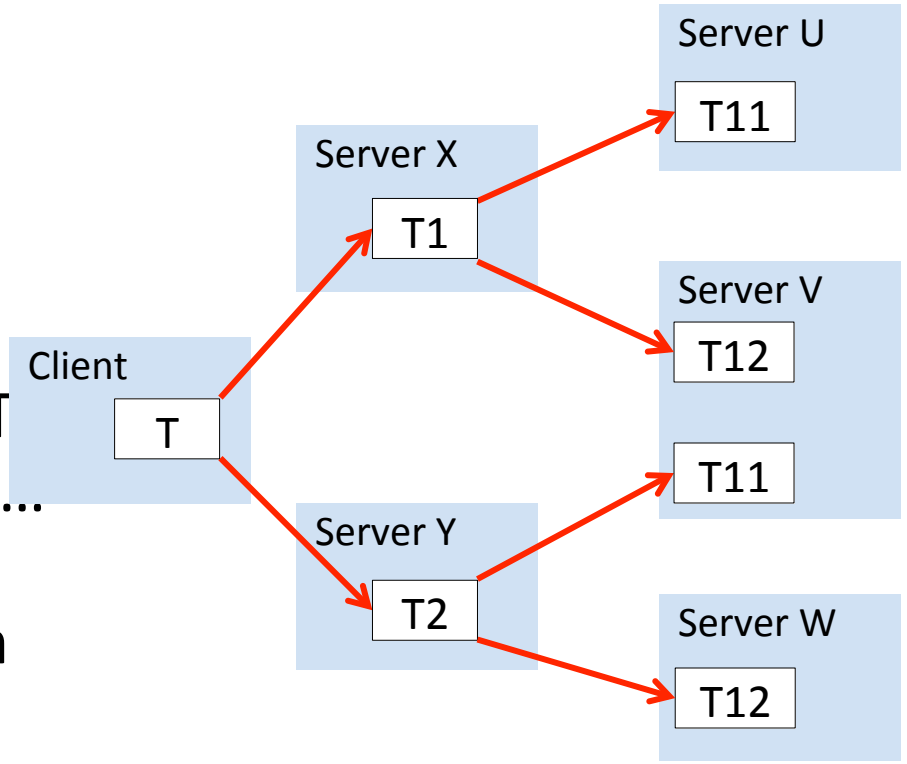


3-Phase Commit

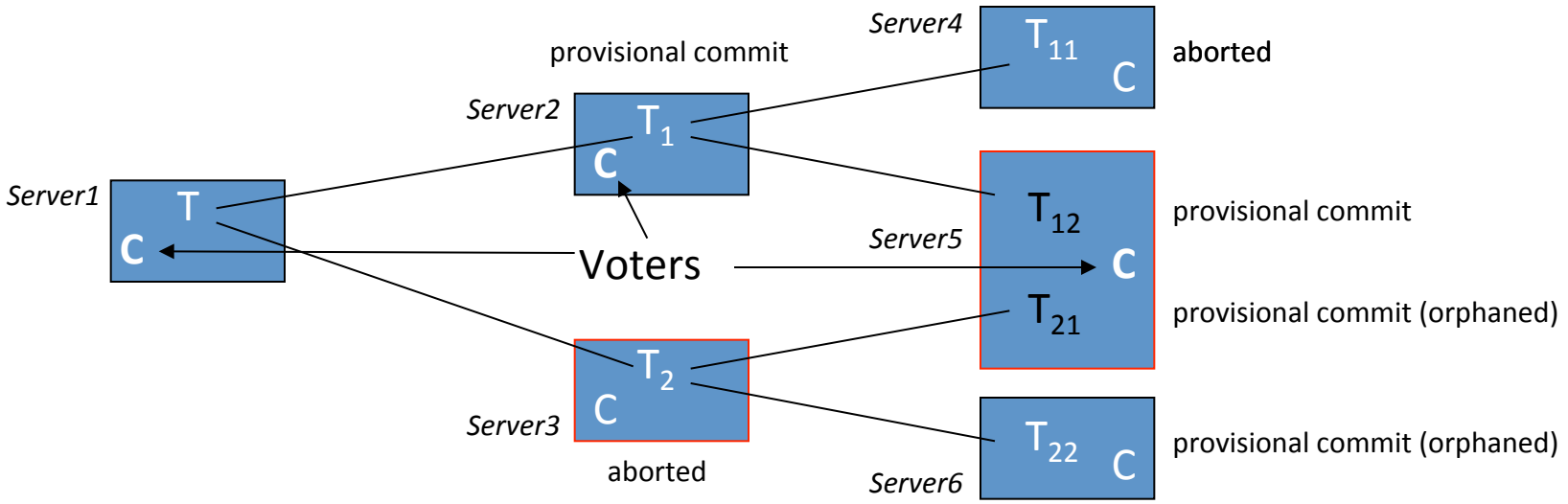
- Three-phase commit protocol is a non-blocking protocol
- Introduction of a third phase:
  - Coordinator and participants first transfer in a “pre-commit” state to make sure that
    - Coordinator received all YES votes for a commit
    - Coordinator received ACKnowledgement that all participants are ready for commit
- With that in place, coordinator as well as participants may independently commit their local transactions in case of failure
- No wait for recovery of coordinator necessary

2PC in Nested Transactions

- Nested transactions are particularly suitable for implementing distributed transaction schemes
  - The top-level transaction T opens sub-transactions T1… Tn that each handle data retrieval and manipulation operations on remote servers
  - Each of these remote manipulations is isolated with a sub-transaction



Commit for Nested Transactions

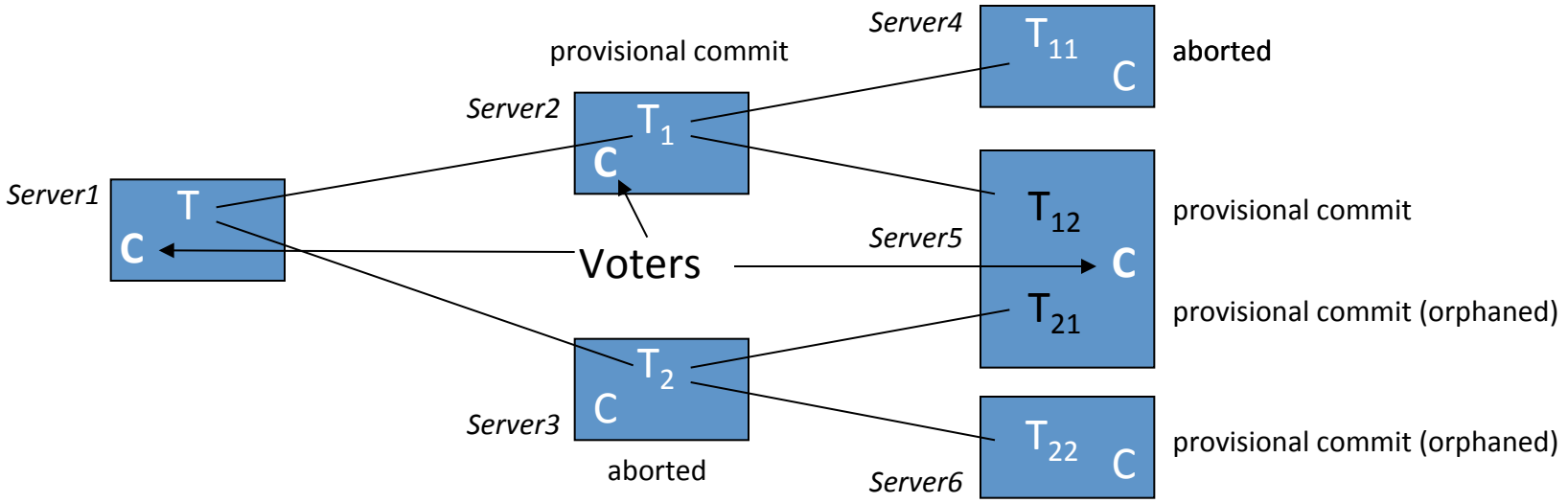


- Behaviour of nested transactions
  - When a sub-transaction comes to an end, it ***provisionally commits*** or it aborts
  - If a parent-transaction aborts, all its sub-transactions must abort too:
    - E.g.: if transaction T2 is aborted, also sub-transactions T21 and T22 must be aborted
  - But: a parent transaction may commit despite sub-transactions aborting

Commitment in Nested Transactions

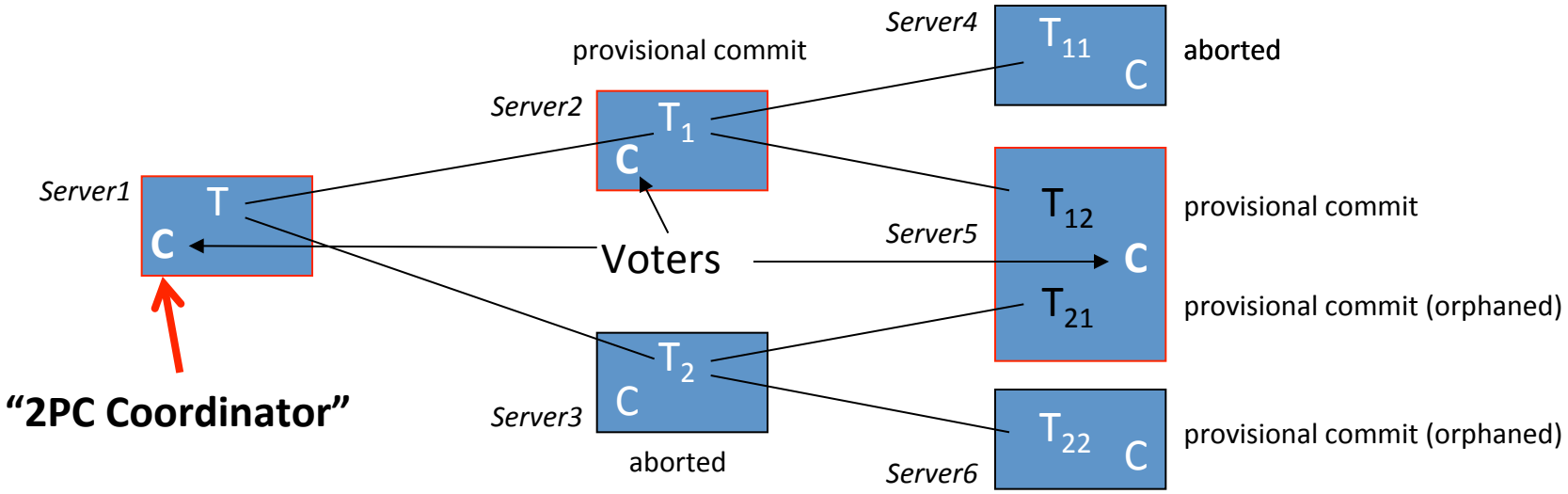
- Concepts in nested Transactions:
  - **Provisional Commitment**: when a sub-transaction completes, it makes an *independent* decision whether to commit *provisionally* or to abort
  - The final commit/abort is done by the top-level transaction, independently from the decisions of the sub-transaction
- A *provisional commitment* cannot be used in a distributed situation
  - No recovery: a provisional commit is not accompanied by measures for recovery from system crash – their intermediate state (manipulations on locked data objects) before the final commit of the complete transaction is not saved
  - If a server, executing such a sub-transaction, crashes, it cannot continue and finish the provisional commitment of the sub-transaction after recovery
- Concept of the **2PC protocol**:
  - Prepared for Commit: a particular server participating in a distributed transaction is "**Ready to Commit**" in phase 2
- A *provisional commit* is not the same as being prepared for recovery from system failure in the **2PC protocol** for flat transactions
- Therefore: 2PC for nested transactions is required

Provisional Commits in Nested Transactions



- When a sub-transaction completes, it will decide whether to abort or to ***commit provisionally***
- If a parent transaction aborts, sub-transactions that are committed provisionally, become orphaned – they have to be aborted as well
- But: parents can commit despite sub-transactions aborting – we have to account for this situation when we use 2PC in nested transactions

2-Phase Commit for Nested Transactions



- There is a coordinator process at each server node
- If sub-transactions execute on the same server, they will share such a coordinator process
- The coordinators at each node play now the role of "2PC coordinator" and "2PC participant" in the 2PC protocol:
  - The coordinator process for the top-level transaction is the "2PC coordinator" for the 2PC protocol
  - All those coordinators of server nodes that have sub-transactions in state "provisional commit", are "2PC participants" and will vote

Preparing Nested Transactions for 2-Phase Commit

- The coordinator process for the top-level transaction acts as the overall “2PC Coordinator” for the 2PC protocol and initiates the 2-phase commit
- Determining the “2PC Participants”
  - The coordinator process of each parent transaction maintains a list of connected sub-transactions
  - When a nested transaction provisionally commits, it reports its own status and that of its descendents to its parent transaction
  - When a nested transaction aborts, it only reports its own Abort to the parent transaction
  - Eventually, the top-level transaction receives a list of all sub-transactions throughout the transaction hierarchy that are provisionally committed
  - Only those coordinator processes that are responsible for these provisionally committed sub-transactions, will become “2PC participants” in the 2PC protocol
- In nested transactions, therefore, the “2PC Coordinator” will try to commit whatever sub-transaction remained in the provisionally committed state

Hierarchic 2PC Protocol

- The 2PC protocol becomes a multi-level nested protocol
  - The coordinator process of the top-level transaction, acting as the "2PC coordinator" for the 2PC protocol, will send a canCommit? message to all coordinator processes of those connected sub-transactions that are in the provisional commit state
  - These contacted coordinator processes themselves will send canCommit? messages to further connected sub-transactions – these canCommit? requests, therefore, propagate through the whole hierarchy of nested transactions.
  - Each coordinator process will also collect the Yes/No votes from the contacted coordinator processes of sub-transactions, before replying to its parent – votes will be handed back to the "2PC coordinator", which is the coordinator process of the top-level transaction.

Hierarchic 2PC Protocol

- Adapting canCommit? for nested transactions:
  - ***canCommit? (topTid, subTransTid) --> Yes / No***
    - Call from a coordinator process to a coordinator process of a sub-transaction
    - Asks whether it can commit a sub-transaction with id subTransTid
    - The first argument topTid is the transaction ID of the top-level transaction
  - A participant receiving a ***canCommit?*** message will lookup ***subTransTid*** in its transaction list of provisionally committed sub-transactions
  - If it finds such a transaction, it will
    - store the intermediate state of this sub-transaction in persistent storage to "prepare" it for possible system failures and recovery
    - Reply with a ***Yes*** vote
  - If it cannot find such a sub-transaction then this is an indicator that the coordinator process of the sub-transaction has crashed since it executed the sub-transaction and it will reply with a ***No*** vote