Distributed Transactions I

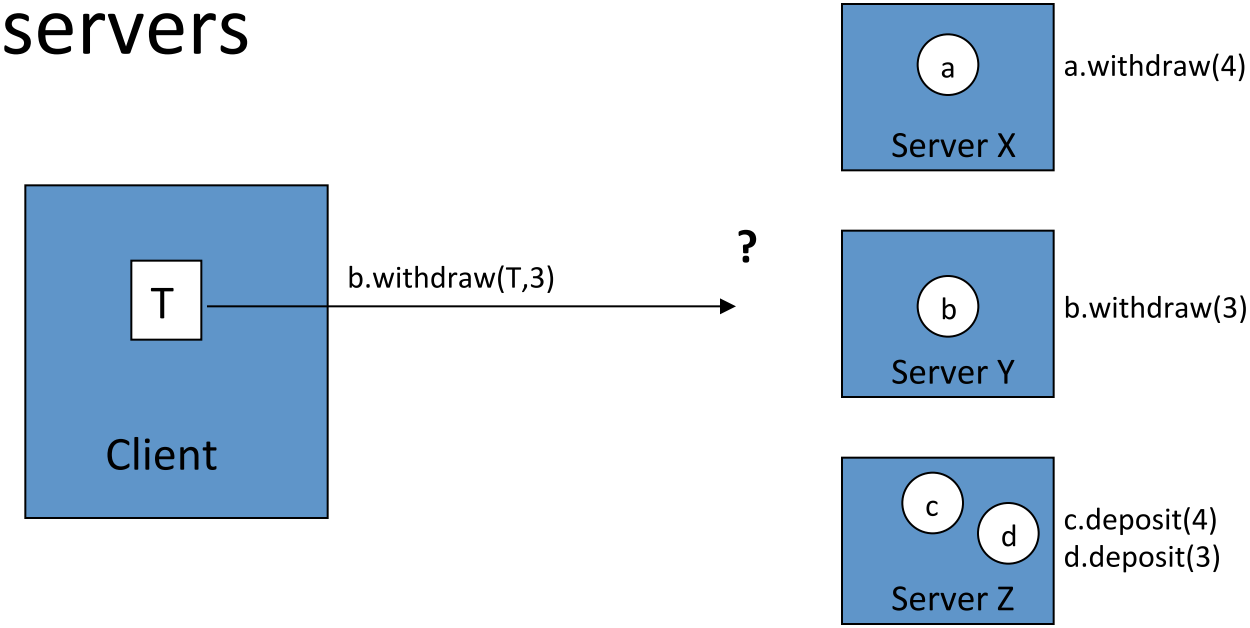
CS3524 Distributed Systems

Lecture 11

Distributed Transactions

- A transaction is distributed, if it invokes read() and write() operations on objects that reside on different servers

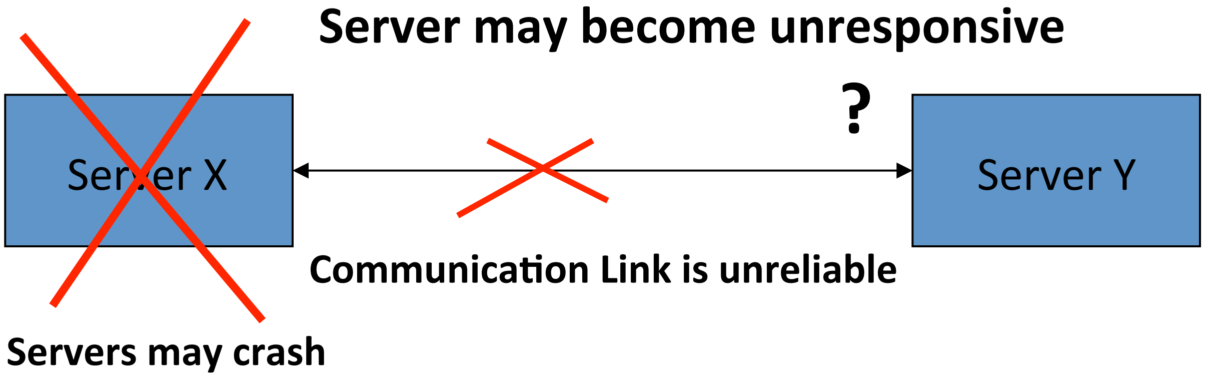
```
T = openTransaction
            a.withdraw(4)
            c.deposit(4)
            b.withdraw(3)
            d.deposit(3)
closeTransaction
```



Atomicity for Distributed Transactions

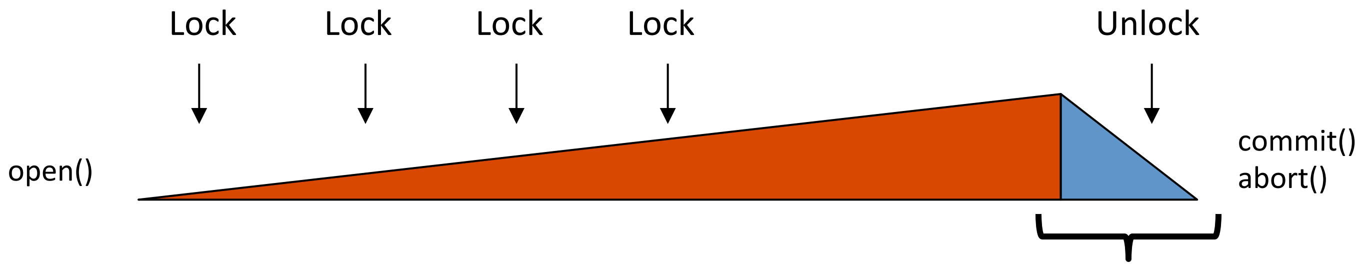
- **Atomicity** must be guaranteed:
  - Either **all servers** involved ***commit*** their part of distributed transaction or **all servers** involved ***abort*** their part of the distributed transaction
- This requires communication between servers
  - We need a **Commit Protocol**

Problem of Distribution


Server may become unresponsive
?
Server X
Server Y
Communication Link is unreliable
Servers may crash

- Communication links are unreliable
  - Servers can crash and become unreachable
  - Communication links may ***temporarily*** fail – the client does not receive any information about success / failure of (sub-) transactions
- Sub-transactions can be affected by server crashes
  - is it possible for the server to recover and to complete the transaction?
  - Is there a need to abort the complete distributed transaction?

Distributed Commit Protocol



At the end of the distributed
Transaction, we need a commitment protocol

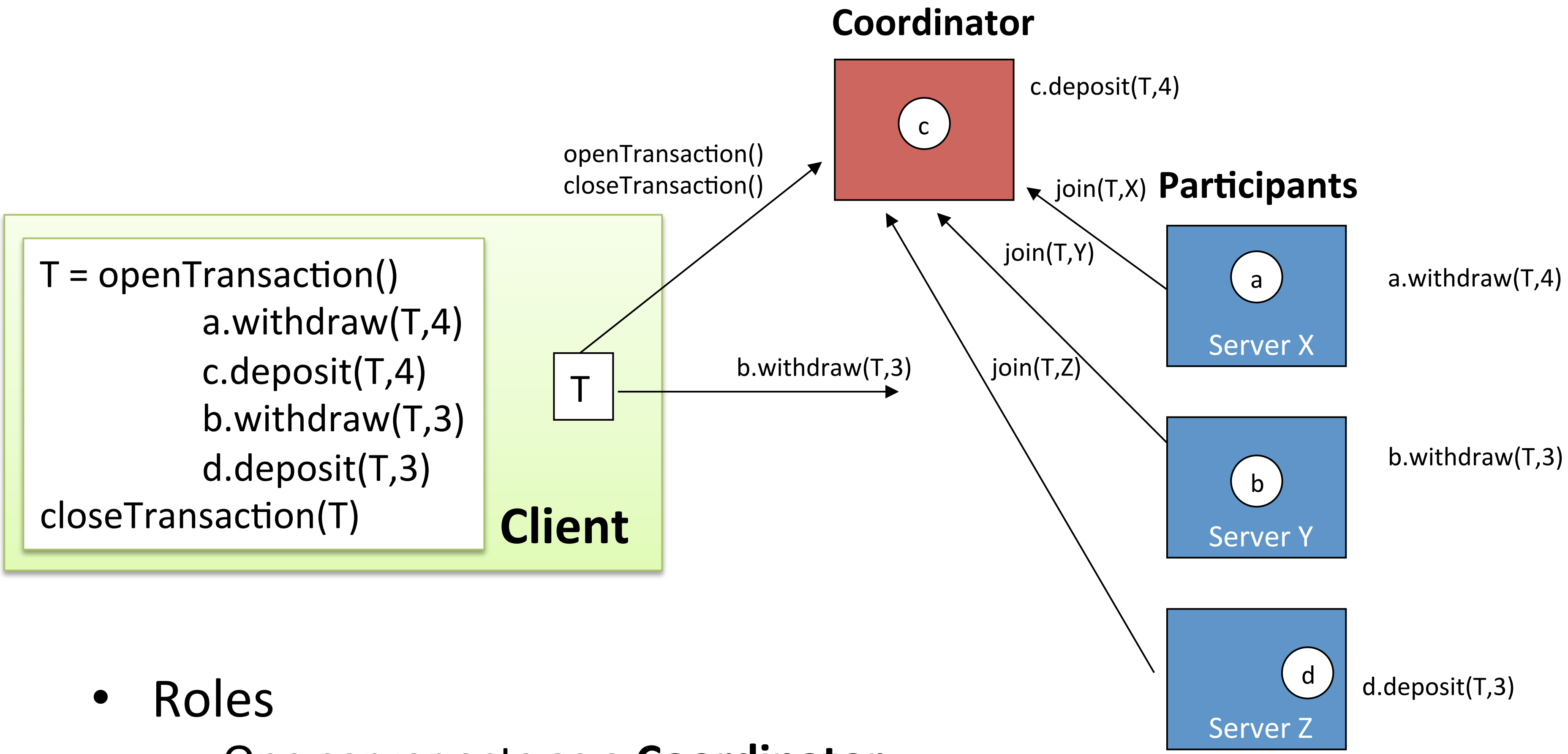
- Any lock can be released only after commit or abort
  - In distributed transactions, a set of servers may manipulate data
  - these manipulations have to be committed / aborted in an orderly fashion
- Commit protocol
  - Servers participating in such a transaction have to communicate and follow a particular protocol to complete their transaction

Distributed Commit Protocol

- Commit Protocol:
  - is a procedure that allows servers to coordinate their actions for committing a distributed transaction in an orderly fashion
- Has to guarantee Atomicity
  - Guarantee that either all of its operations (which are distributed across multiple servers) are carried out or none of them
- Has to guarantee Durability
  - has to be designed to account for possible failure situations
  - the commit phase of a transaction has to be finished (either committed or aborted) despite server crashes, lost messages etc.

Distributed Commit Protocol

Coordinator and Participant



```
T = openTransaction()
        a.withdraw(T,4)
        c.deposit(T,4)
        b.withdraw(T,3)
        d.deposit(T,3)
closeTransaction(T)
```

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

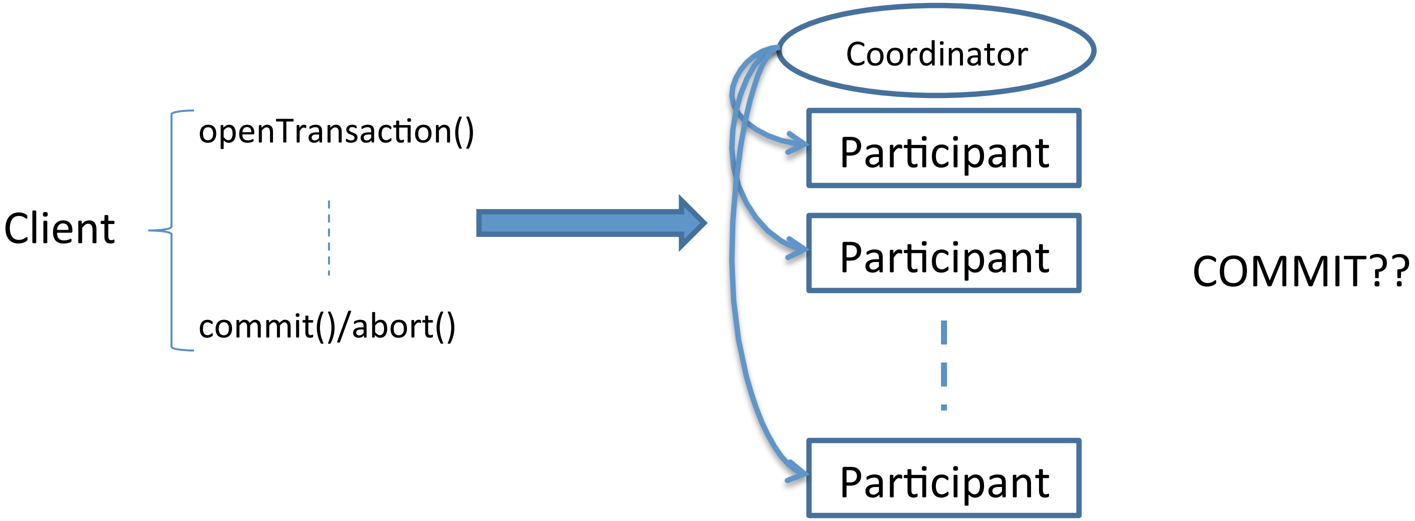
Distributed Commit Protocol

- The “Coordinator”:
  - One of the servers involved in a distributed transaction takes on the role of the “coordinator”
  - a client wishing to perform a transaction contacts a coordinator by sending an ***openTransaction()*** request
    - The coordinator server becomes the coordinator of the distributed transaction
    - The coordinator is responsible for committing or aborting the transaction
- The “Participant”:
  - Each server participating in a distributed transaction ***registers*** with the coordinator as a “participant”

Distributed Commit Protocol

- Objective
  - All participating servers have to coordinate their actions during the commit phase
  - All participating servers must be sure that the whole distributed transaction was either successfully committed or aborted

Simple Commit Protocol



Client

openTransaction()

commit()/abort()

Coordinator

Participant

Participant

Participant

COMMIT??

- Simple ("One-Phase") Commit Protocol:
  - The coordinator **keeps repeating a commit request**, until all of the participants have acknowledged the commit
- Problem – Is **not feasible**:
  - Coordinator may wait a long time or forever until all participants responded
  - What if one participant fails?
    - Only the coordinator can abort the transactions, servers cannot make a unilateral decision to abort a transaction
- One-phase commit protocol is inadequate !!
- Better solution: Two-phase commit protocol

Two-Phase Commit Protocol

Two-phase Commit Protocol

- Is based on a **voting scheme**:
  - All participants reach a **consensus** – participants vote *either to commit or to abort*
- Preserves **atomicity**:
  - if a part of a transaction is aborted, the complete transaction must be aborted
  - This requires informing all participants about such an abort and ***making sure*** that they get the message
- **Unilateral Abort possible**:
  - Servers (coordinator, participants) can unilaterally decide to abort their part of a transaction, **if their behaviour can be detected by coordinator**

Two-phase Commit Protocol Phases

- Consists of two phases

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

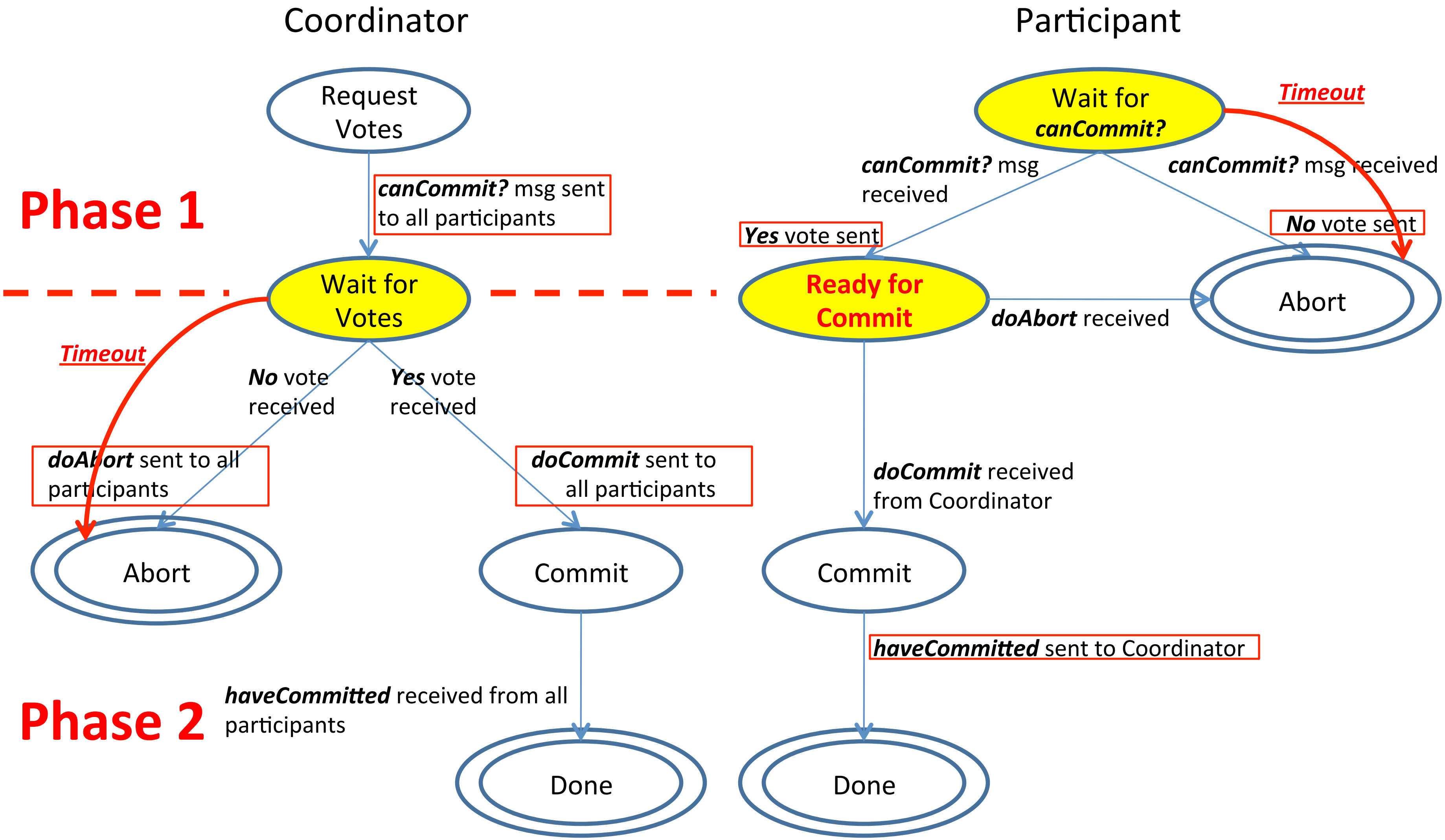
Coordinator asks participants whether they are ***prepared*** **to commit** or **abort**

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***

- If a participant votes to commit, it must make sure that it can commit eventually, even if it crashes – participant must be able to ***recover from system failure*** by storing intermediate state

2-Phase Commit Protocol



Coordinator

Participant

Phase 1

Phase 2

Request Votes

canCommit? msg sent to all participants

Wait for Votes

Timeout

No vote received

Yes vote received

doAbort sent to all participants

doCommit sent to all participants

Abort

Commit

haveCommitted received from all participants

Done

Wait for ***canCommit?***

Timeout

canCommit? msg received

canCommit? msg received

Yes vote sent

No vote sent

Ready for Commit

doAbort received

Abort

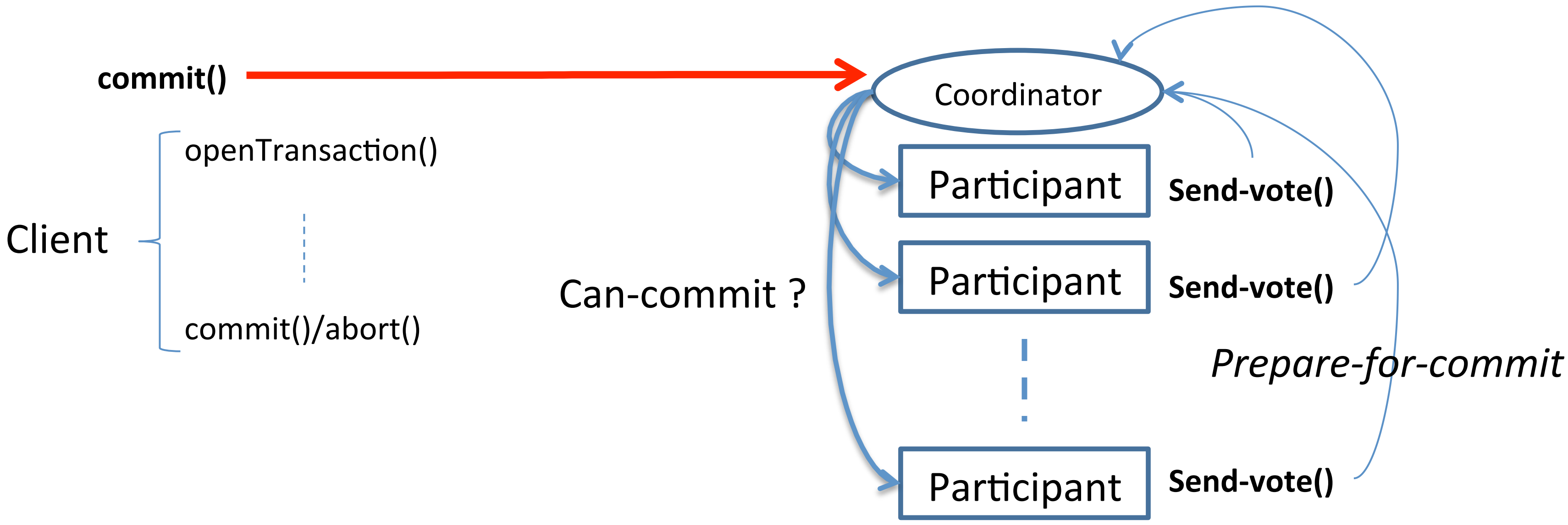
doCommit received from Coordinator

Commit

haveCommitted sent to Coordinator

Done

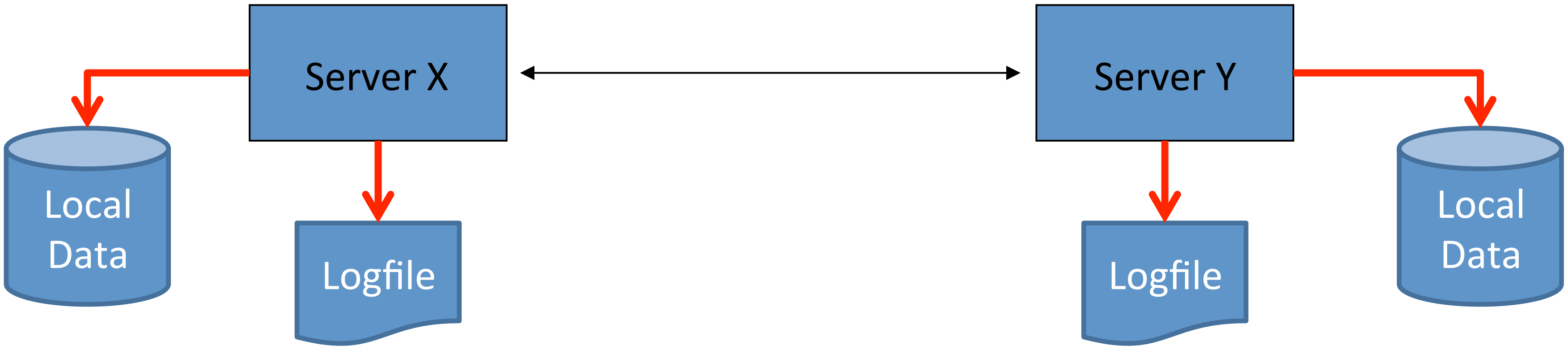
Two-phase Commit Protocol
Phase 1: Voting Phase



- Voting
  - Coordinator asks participatants whether they can commit
  - Participants send their vote: either "Yes" or "No"
- Participants have to prepare for commit:
  - Write log files to record which write operations to commit

Phase 1: Prepare to Commit

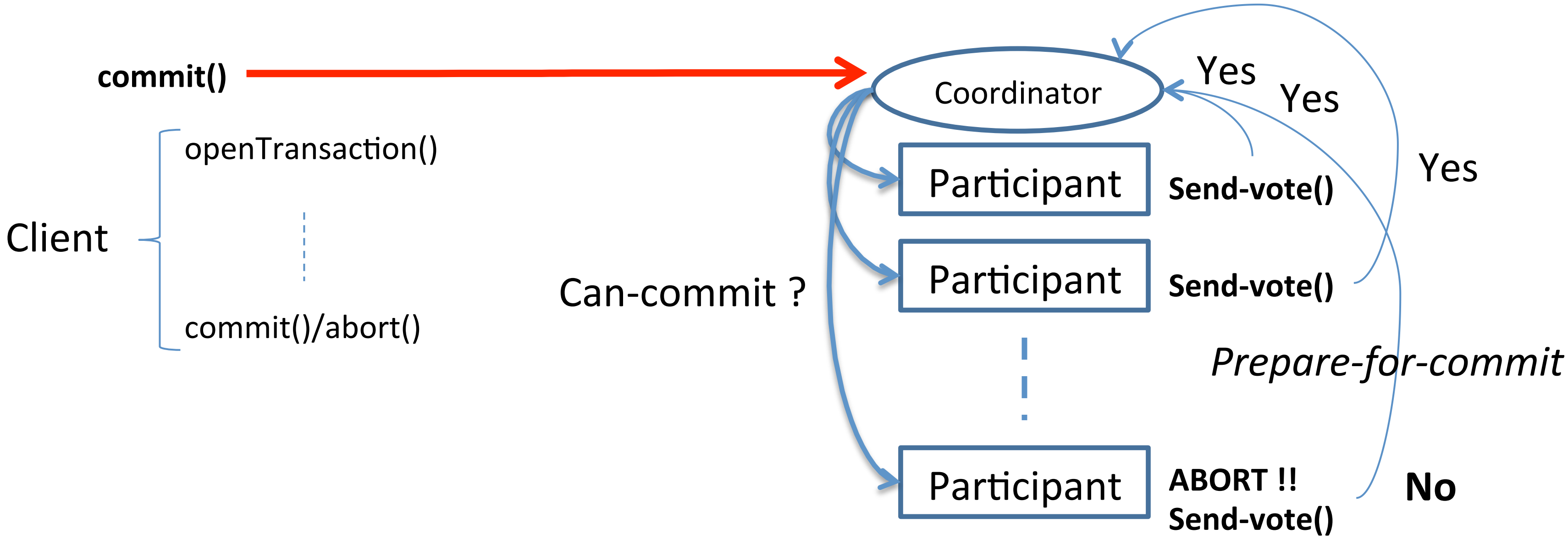
- Each server, coordinator and participant, must be able to ***recover from system failure*** by storing intermediate results
  - Intermediate results stored in persistent storage
  - Log files record what objects are manipulated , locked etc.
  - These log files are used when a server is restarted and recovers from failure
- There may be cases where all other servers have to wait until a crashed server recovers



ACID Principle: Durability

- Guaranteed Commit
  - *If a participant voted for a commit of the transaction then it must guarantee that it will eventually commit its part of the transaction*
- This must be guaranteed for system failure
  - If participant recovers from a system failure, it must be able to commit its transaction
- Principle of Durability !

Two-phase Commit Protocol
Phase 1: Voting Phase



commit()

Client

openTransaction()

commit()/abort()

Can-commit ?

Coordinator

Participant

Send-vote()

Yes

Participant

Send-vote()

Yes

Yes

Prepare-for-commit

Participant

ABORT !!

Send-vote()

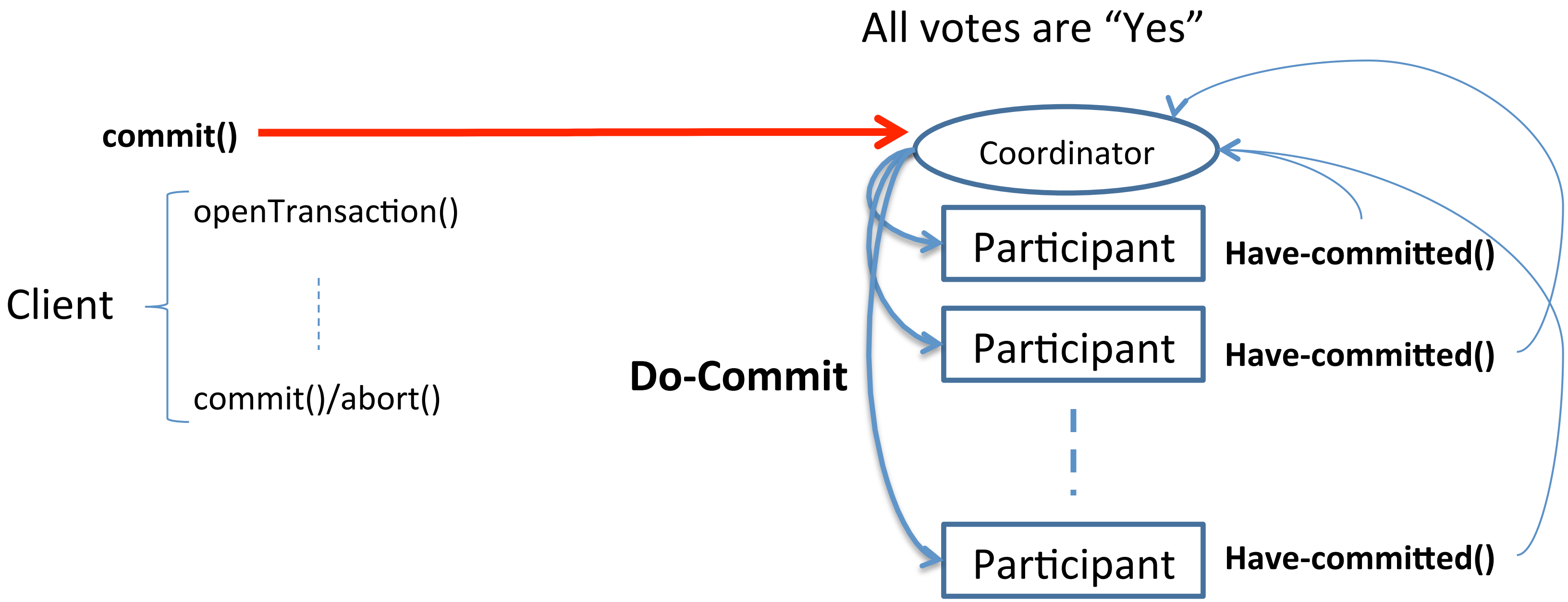
No

- Unilateral Abort in phase 1
  - A participant can **unilaterally** abort **any time** and send a “No” vote
  - Rollback of write operations, release of locks
- If there is at least one “No” vote than coordinator has to request abort from all other participants in phase 2

Two-phase Commit Protocol
Phase 1: Voting Phase

- ***Request for votes***:
  - Coordinator asks participants whether they are prepared to commit or abort
- ***Participant prepares for Commit***
  - If a participant sends a vote to commit its part of a transaction, it will prepare for commit by recording manipulations on data object in permanent storage and marks the transaction as *prepared*
    - A participant is, therefore, said to be in a *prepared* state for a transaction commit.
  - all participants vote whether to commit, once they voted to commit, they are not allowed to abort any more
  - When a participant is prepared for Commit, it has to wait for a Coordinator Decision – **blocked until coordinator decides**
- ***Participant aborts***
  - If a participant sends a vote to abort its part of a transaction, it will unilaterally abort (immediate abort, release locks)

Two-phase Commit Protocol
Phase 2: Completion Phase

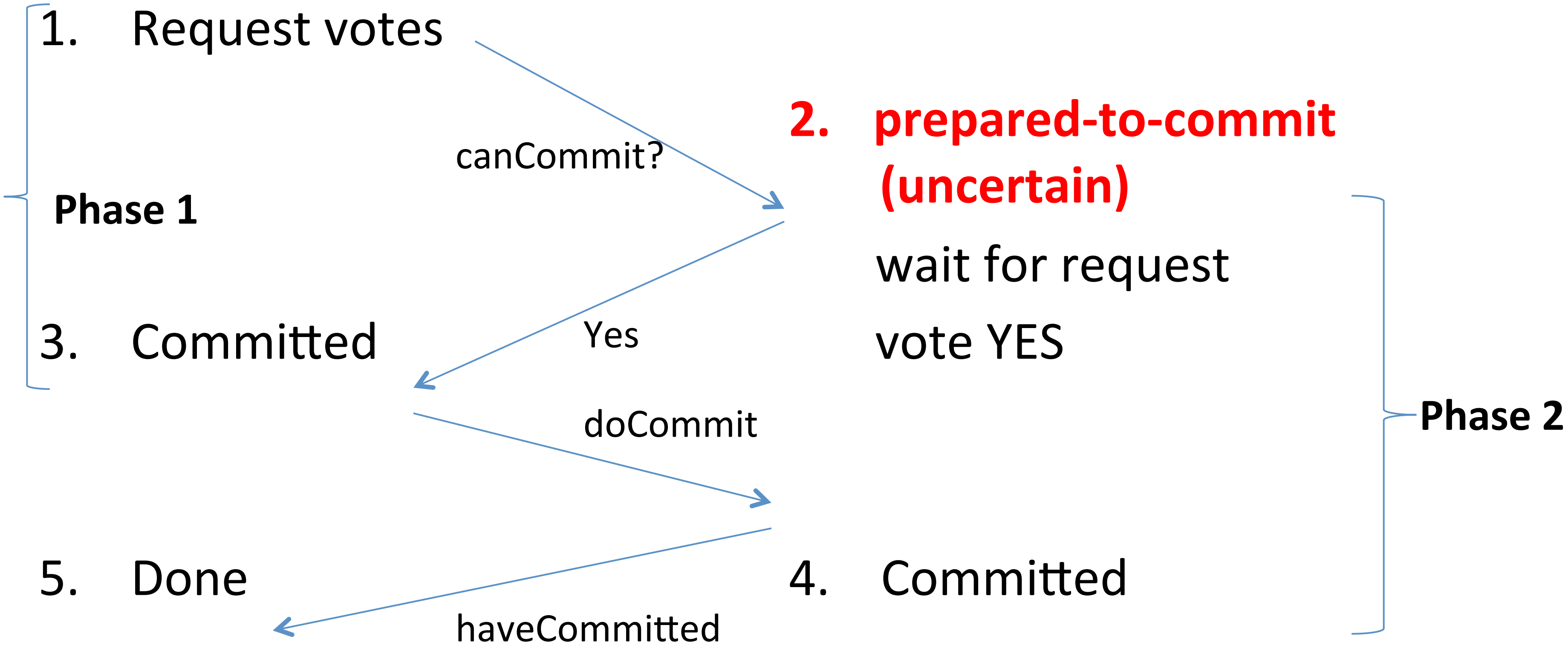


- Completion Phase – all votes "Yes"
  - Coordinator first asks participants to commit
  - Participants then send send message to coordinator about success of commit

2PC Protocol

All participants vote for commit, Coordinator commits

- Coordinator
- Participant



1. Request votes

canCommit?

2. prepared-to-commit (uncertain)

wait for request

vote YES

Yes

3. Committed

Phase 1

doCommit

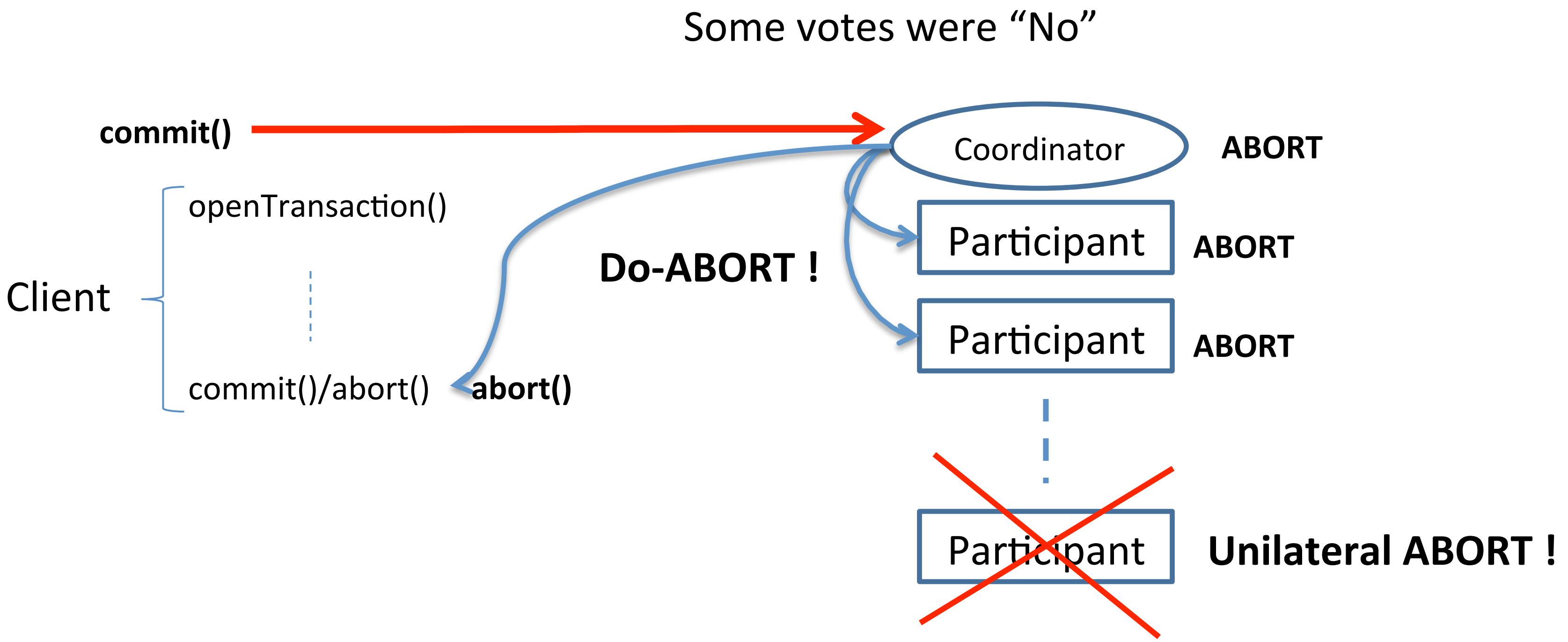
4. Committed

haveCommitted

5. Done

Phase 2

Two-phase Commit Protocol
Phase 2: Completion Phase

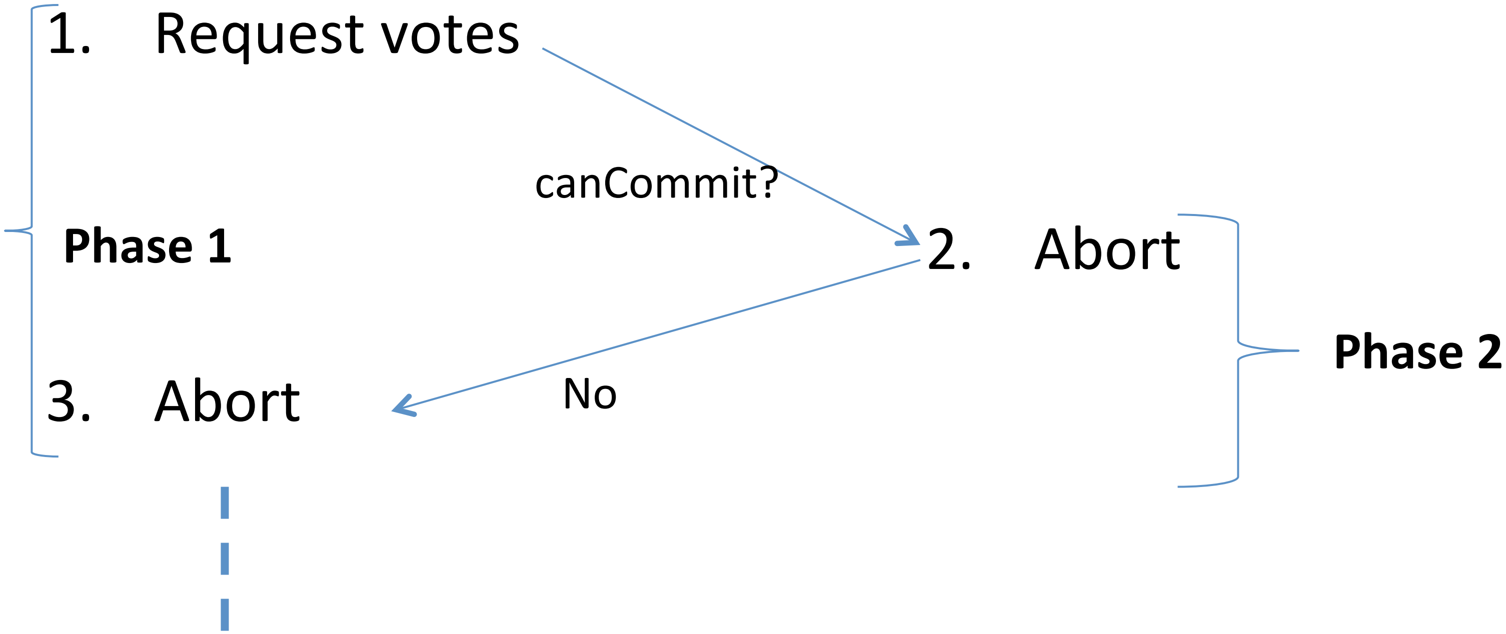


- Completion Phase – some votes are "No"
  - Coordinator asks participants to ABORT
  - No further communication between participants and coordinator
- Participants that voted "No" can unilaterally abort, without communicating with Coordinator

2PC Protocol

Participant votes Abort, unilaterally aborts

- Coordinator
- Participant



1. Request votes

canCommit?

2. Abort

Phase 1

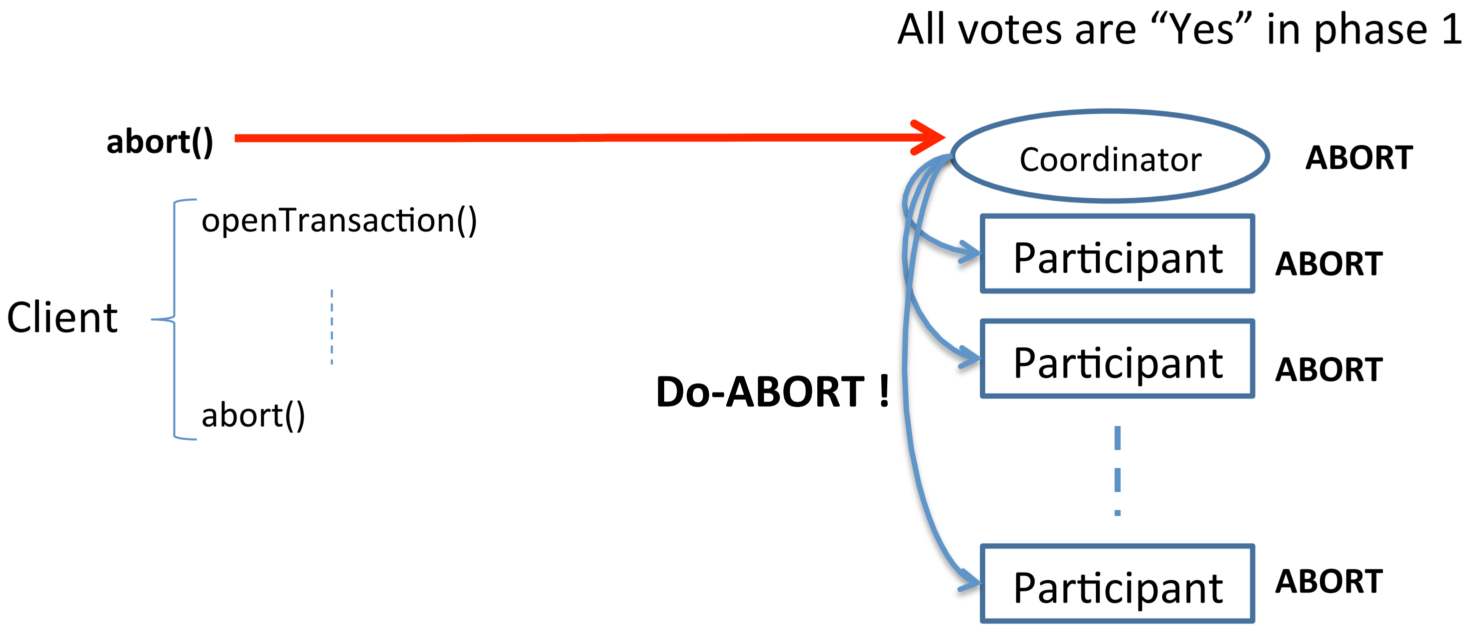
Phase 2

No

3. Abort

Coordinator has to abort all other participants.

Two-phase Commit Protocol
Phase 2: Completion Phase

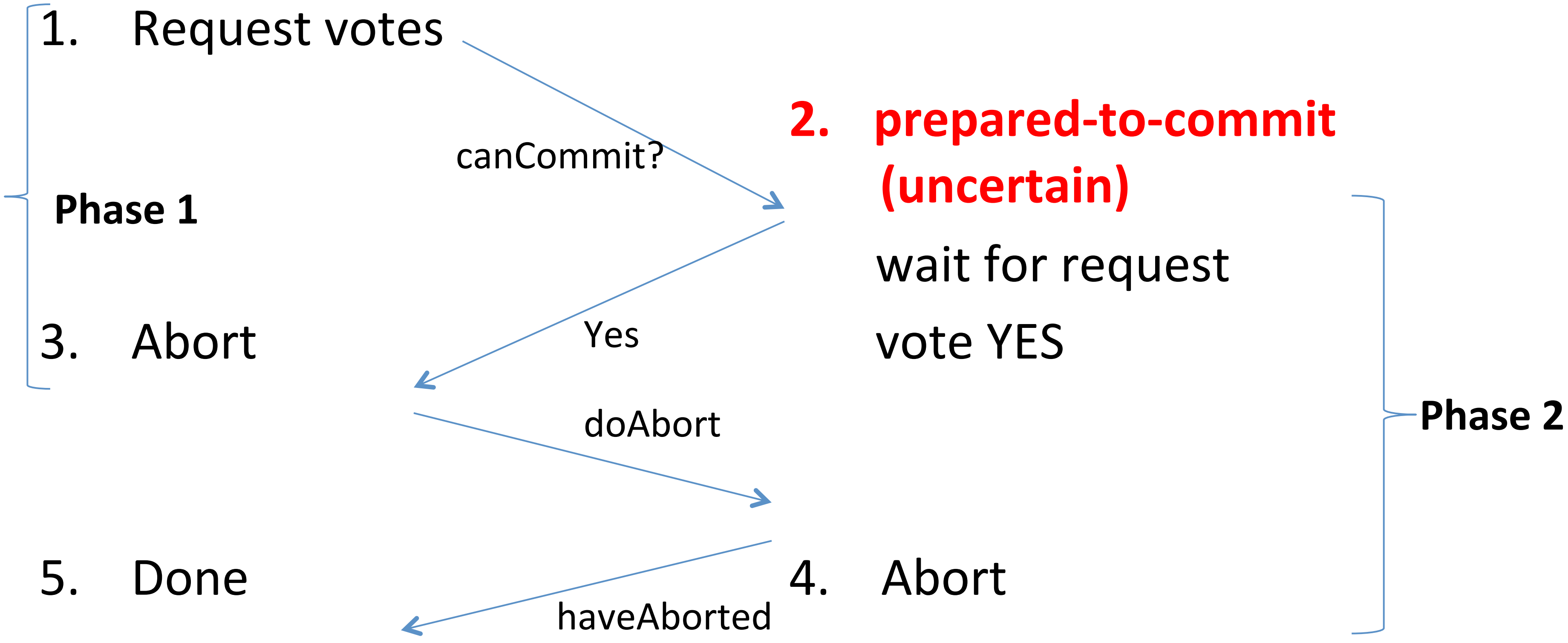


- Completion Phase – all votes "Yes"
  - Coordinator may still ask participants to abort
- Unilateral Abort:
  - In case of Abort, there is no communication between participants and coordinator

2PC Protocol

Participant votes for commit, Coordinator aborts

- Coordinator
- Participant



1. Request votes

canCommit?

2. **prepared-to-commit (uncertain)**

wait for request

vote YES

Yes

3. Abort

Phase 1

doAbort

4. Abort

haveAborted

5. Done

Phase 2

Coordinator received a ***No*** vote and sends a ***doAbort*** to all other participants.

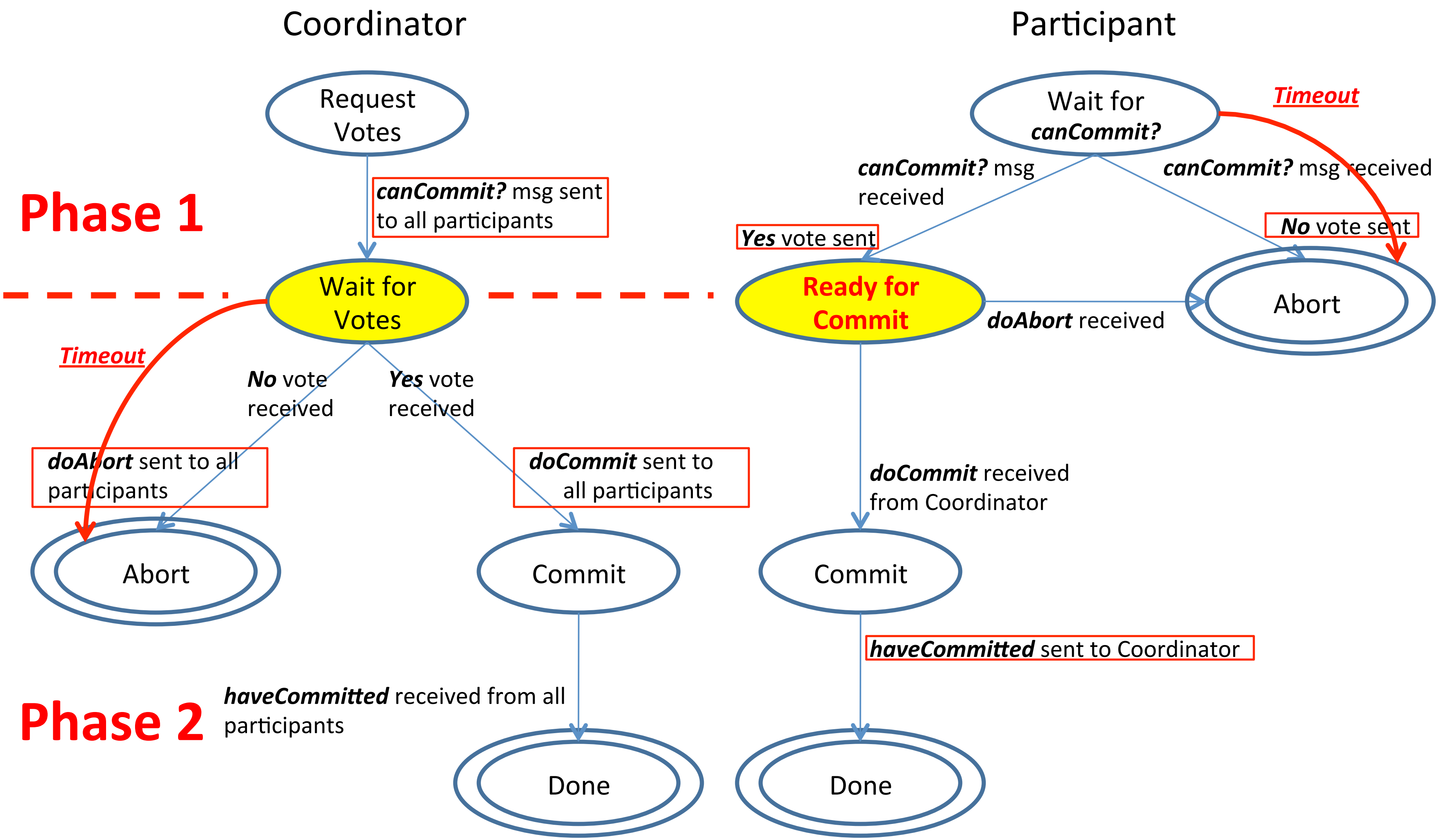
Two-phase Commit Protocol Phase 2: Completion Phase

- Coordinator collects votes from participants and informs participants about its decision
  - If all votes are commit-votes:
    - the coordinator informs all participants to commit their local transaction
  - If not all votes are commit-votes:

      (This may be because at least one participant sends abort **or is not responding within a time limit)**

    - the coordinator informs all those participants that voted for commit, to abort their local transaction
- Participants that voted for abort, immediately abort their local part of the distributed transaction, without waiting for coordinator

2-Phase Commit Protocol



Coordinator

Participant

Phase 1

Phase 2

Request Votes

canCommit? msg sent to all participants

Wait for Votes

Timeout

No vote received

Yes vote received

doAbort sent to all participants

doCommit sent to all participants

Abort

Commit

haveCommitted received from all participants

Done

Wait for ***canCommit?***

Timeout

canCommit? msg received

canCommit? msg received

Yes vote sent

No vote sent

Ready for Commit

doAbort received

Abort

doCommit received from Coordinator

Commit

haveCommitted sent to Coordinator

Done

2PC Protocol Implementation

- Phase 1: The Voting Phase
  - The coordinator sends a ***canCommit?*** Request to each participant in the transaction
  - When a participant receives a ***canCommit?*** request, it decides whether to vote *Yes* or *No*
    - If it decides to vote ***Yes***, the participant *prepares* for commit by saving data objects plus its current state in persistent storage (necessary for recovery)
    - If it decides to vote ***No***, it aborts immediately
  - The participant sends a vote: ***Yes*** or ***No***
- Phase 2: The Completion Phase
  - Coordinator collects the votes including its own
    - If there are no failures and all votes are ***Yes***, the coordinator decides to commit and sends a ***doCommit*** instruction to all participants
    - Otherwise the coordinator sends a ***doAbort***
  - Participants that voted ***Yes*** are waiting for a ***doCommit*** or ***doAbort***. If the instruction is ***doCommit*** they act accordingly and send a ***haveCommitted*** confirmation to the coordinator

Operations of the 2PC Commit Protocol

- canCommit? (Tid) --> *Yes* / *No*
  - Call from coordinator to participant
  - Asks whether it can commit a transaction
  - Participant replies with its vote
- doCommit (Tid)
  - Call from coordinator to participant
  - Tells participant to commit its part of a transaction
- doAbort (Tid)
  - Call from coordinator to participant to tell participant to abort its part of a transaction
- haveCommitted (Tid, Participant)
  - Call from participant to coordinator
  - Confirms that it has committed the transaction
- getDecision (Tid) --> *Commit* / *Abort*
  - Is used to recover from server crash or delayed messages
  - Call from participant to coordinator, if there is no reply from the coordinator after some delay
  - Participant asks for the decision on a transaction after it has voted *Yes*

Participant Interface

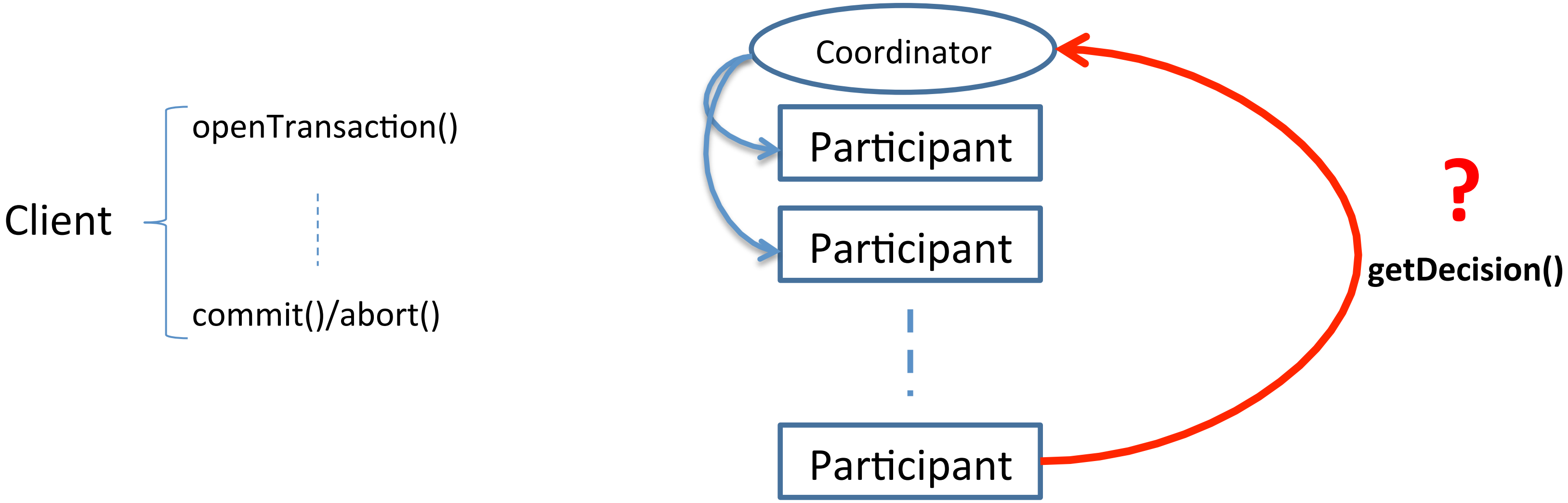
- Vote = canCommit (Tid)
  - Returns the vote of the participant to the coordinator
  - Asks whether it can commit a transaction
- doCommit (Tid)
  - Coordinator tells participant to commit its part of a transaction
- doAbort (Tid)
  - Call from coordinator to participant to tell participant to abort its part of a transaction

Handling Failure Situations

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 sends getDecision message to 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

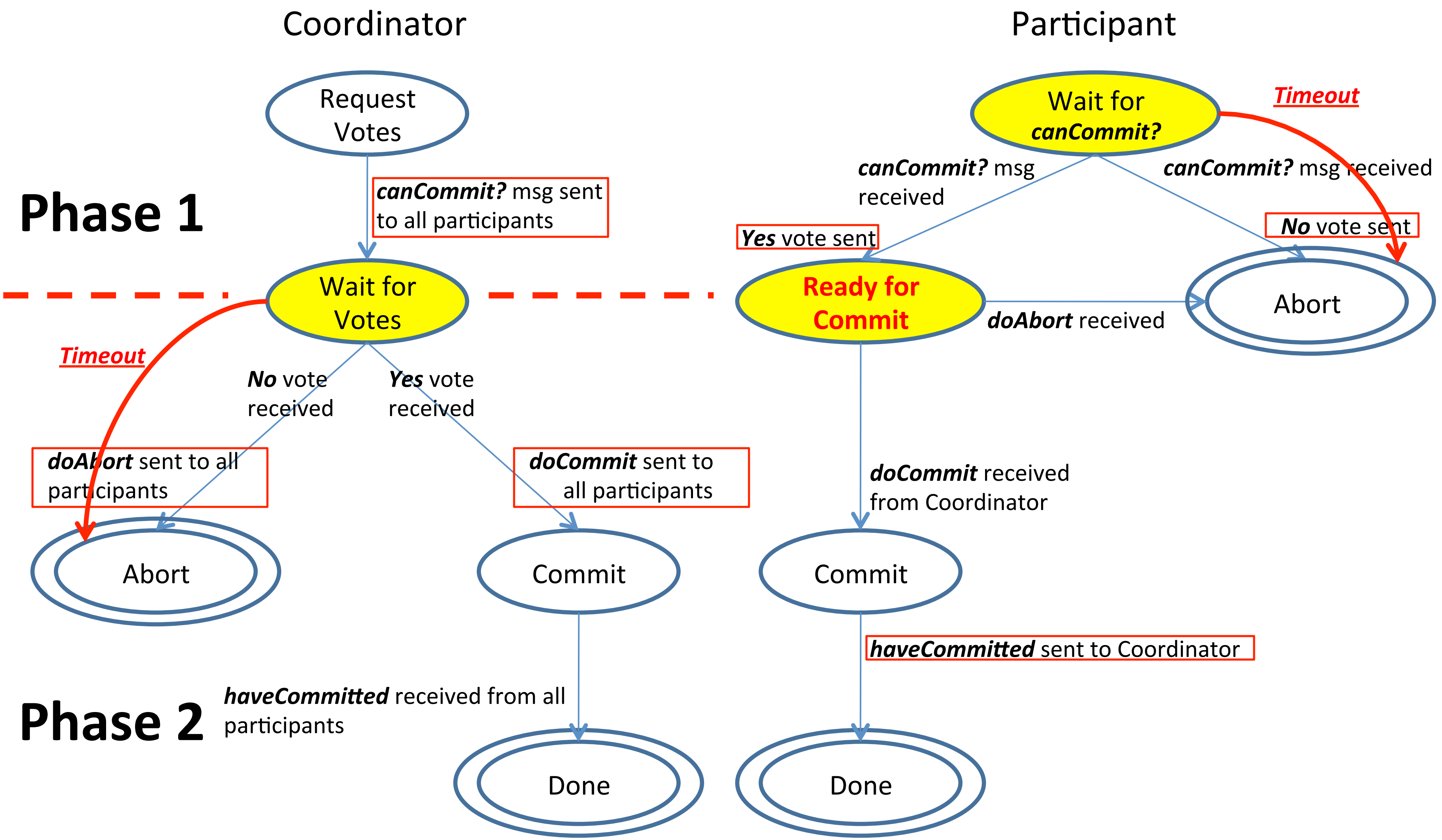
Two-phase Commit Protocol

- 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 ?**

Problem in System Failure Situations

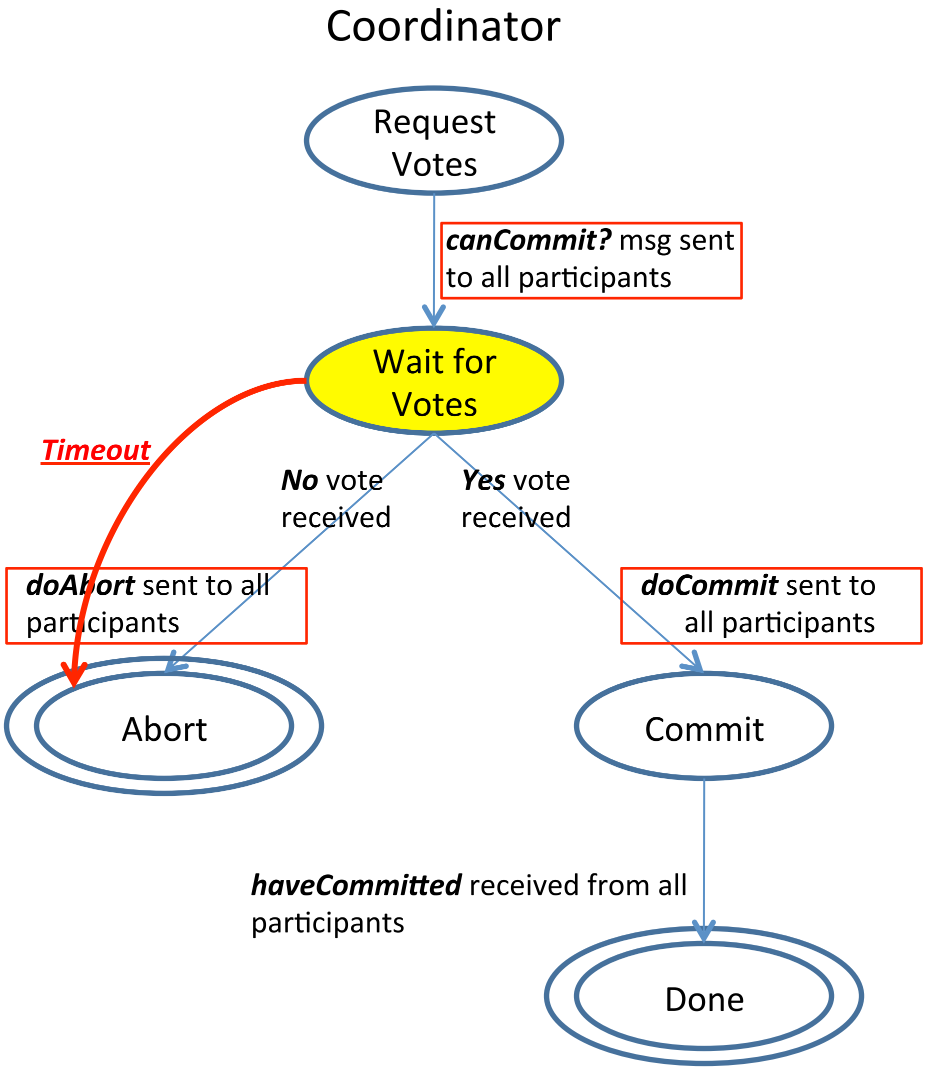
- 2PC can become a *blocking* protocol if coordinator fails in phase 2:
  - 2PC can cause considerable delays to participants in an *uncertain* state, this occurs when the coordinator fails and cannot reply to getDecision() requests
  - Participants have to wait, locks on data objects remain in place as the transaction cannot finish

Timeout 2-Phase Commit Protocol



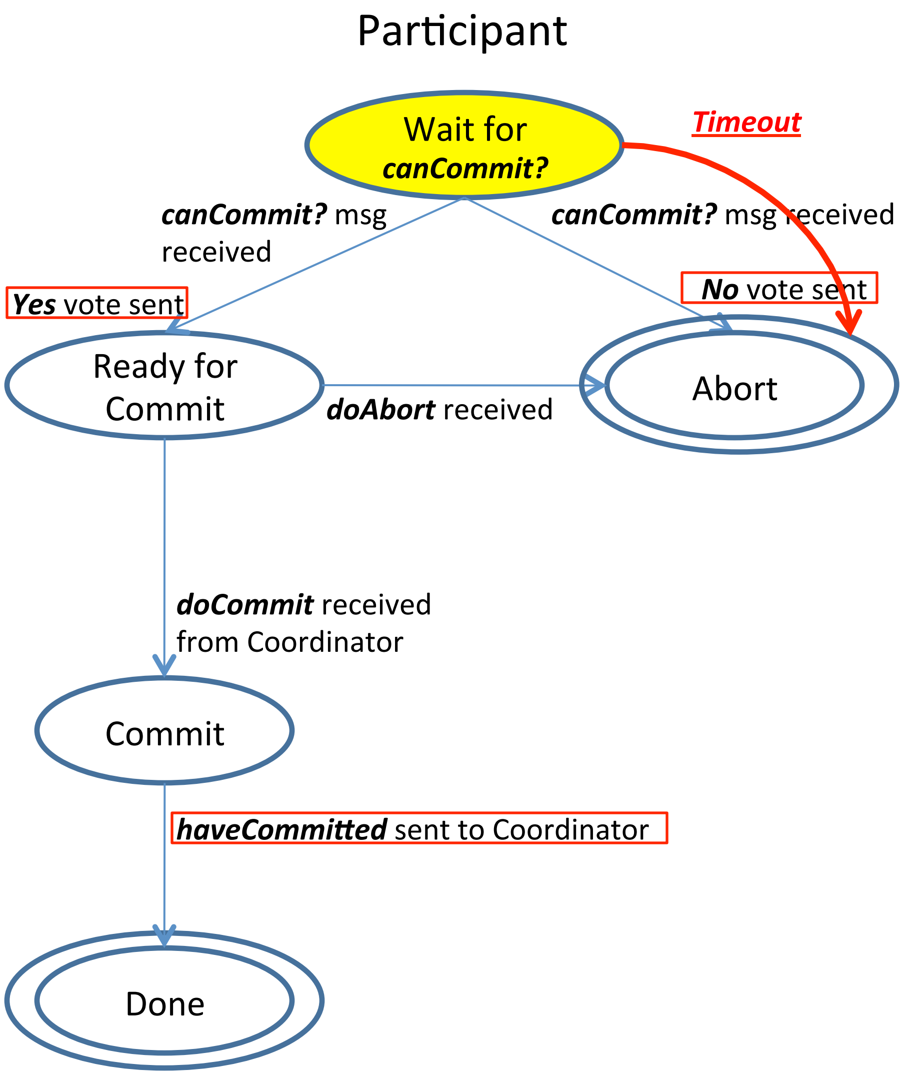
Timeout Actions in the 2PC Protocol

- 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


Coordinator
Request Votes
canCommit? msg sent to all participants
Wait for Votes
Timeout
No vote received
Yes vote received
doAbort sent to all participants
doCommit sent to all participants
Abort
Commit
haveCommitted received from all participants
Done

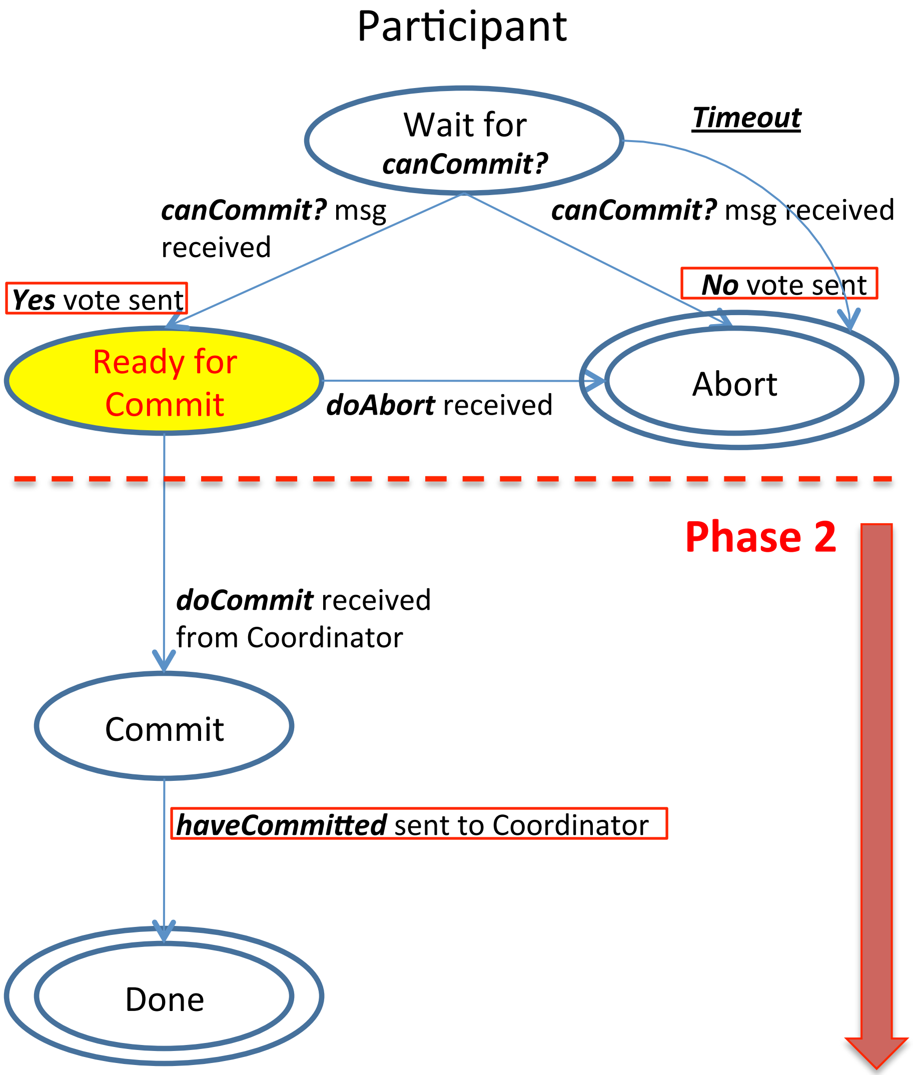
Timeout Actions in the 2PC Protocol

- 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


Participant
Wait for *canCommit?*
Timeout
canCommit? msg received
Yes vote sent
canCommit? msg received
No vote sent
Ready for Commit
doAbort received
Abort
doCommit received from Coordinator
Commit
haveCommitted sent to Coordinator
Done

Blocking Situation in Phase 2

- Participant waits for coordinator
  - Participant has sent vote to commit, is ready for commit
  - As it has voted for commit, no locks on shared objects can be released, the participant has to wait for coordinator decision
  - Coordinator is late in sending a doCommit or doAbort (phase 2)
- **Participant cannot abort unilaterally, it has to wait for the coordinator's decision !!**
- How to find out the coordinator's decision, if coordinator has crashed?



Strategy: Ask for Decision

- Situation: Coordinator is late in sending a doCommit or doAbort (phase 2)
  - 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 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 and call the coordinator's hasCommitted() method

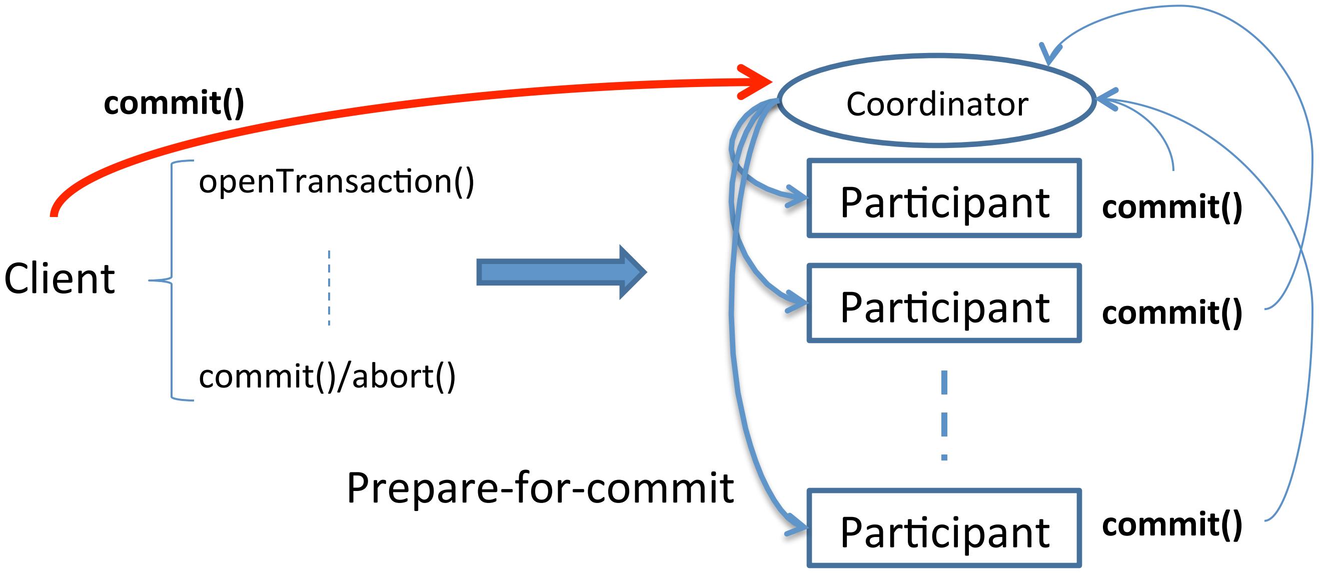
Strategy: Elect a new Coordinator

- Situation: Coordinator is late in sending a doCommit or doAbort (phase 2)
  - Participant is prepared for commit (in phase 2), but *uncertain* about the outcome of the voting – it cannot proceed, therefore objects remain locked
- What if the coordinator has failed after sending a ***request for vote*** message?
  - 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

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 and cannot reply to getDecision() requests
  - Participants have to wait, 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).

Commit Protocol, Requirements

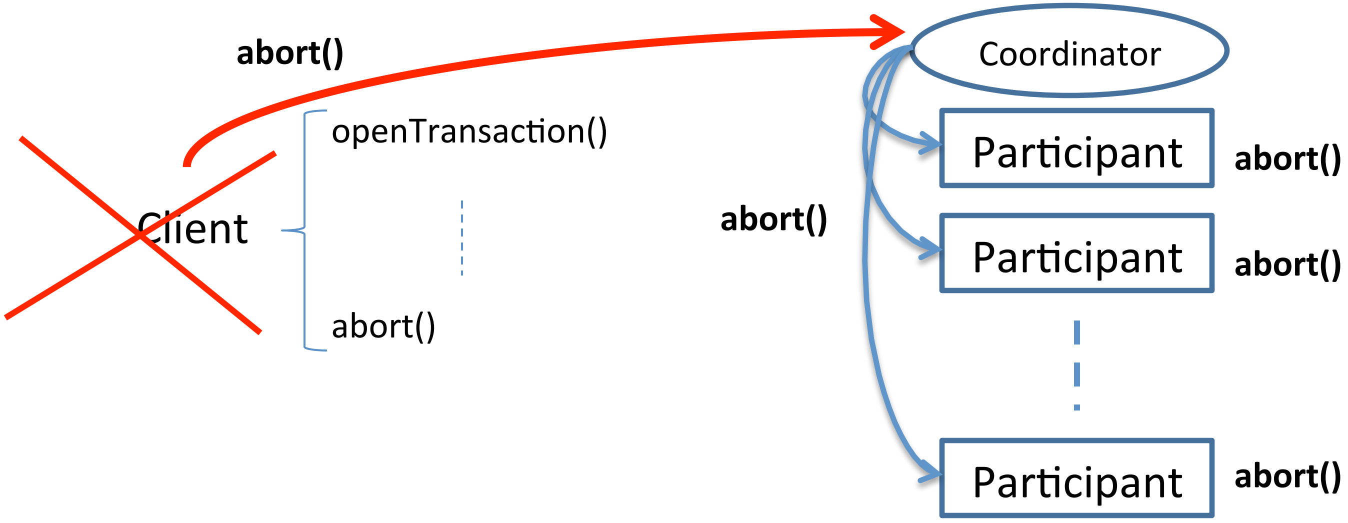


- If the client requests a **commit**:
  - The distributed transaction can only be committed, if all local sub-transactions executed on servers finished successfully
- **All participants have to be informed** that a **commit** is imminent, as they have to prepare for commit
  - Write log information to enable recovery from possible system crashes
- The coordinator can declare the transaction committed only if it gets a **reply from each participant** that their local sub-transactions have successfully finished

Commit Protocol, Requirements

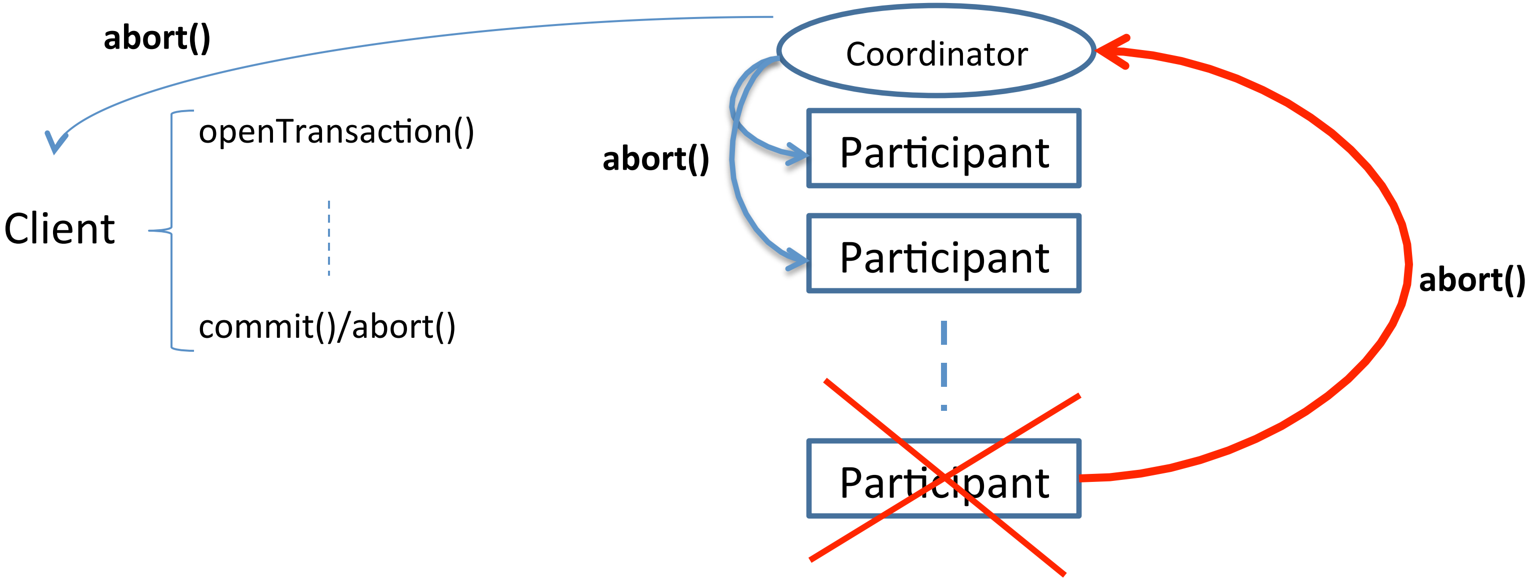
- If the client requests a **commit**:
  - The distributed transaction can only be committed, if all local sub-transactions executed on servers finished successfully
- Coordinator Action:
  - **All participants have to be informed** that a **commit** is imminent, as they have to prepare for commit
    - Write log information to enable recovery from possible system crashes
- Participant Action:
  - The coordinator can declare the transaction committed only if it gets a **reply from all participants** that their local sub-transactions have successfully finished
  - For this, they have to communicate the state of their local portion of the distributed transaction to the coordinator

Commit Protocol, Requirements



- If the client requests an **abort**:
  - All participants have to be informed that an abort is imminent
    - they have to release local locks and
    - undo/rollback any manipulation on data objects
  - Do they have to report back to Coordinator ?

Commit Protocol, Requirements



- Participant
- If a *participant* has to **abort** its portion of the distributed transaction
  - the coordinator has to abort the overall distributed transaction, all participants involved in that transaction are told to abort (rollback of complete transaction)
- How does Coordinator know that all transactions are aborted?