Distributed Systems

Monsoon 2024

Lecture 16

International Institute of Information Technology

Hyderabad, India

Resources:

1. The book Distributed Systems: Concepts and Design, Kindberg et al.
2. The paper Efficient Commit Protocols for the Tree of Processes Model of Distributed Transactions, C. Mohan, B. Lindsay, 1983, ACM
3. Lecture slides by Kathleen Durant, NEU.

Logistics

- HW4 on gRPC – to be issued today
- Project choices – to be released tonight, document with list of projects and brief description, form for expressing preferences, find your teammate, each team to fill the form.
- Two special lectures
  - Distributing trust and blockchain: October 10, 2024
  - Federated Learning: October 24, 2024

Transactions

- Some situations, clients require a sequence of requests to a server to be atomic.
- Atomic understood as free from interruption by operations being performed on behalf of other concurrent clients.
- Either all of the operations must be completed successfully or they must have no effect at all in the presence of server crashes.

Transactions

- The goal of transactions is to ensure that all of the objects managed by a server remain in a consistent state when they are accessed by multiple transactions even in the presence of server crashes.
- Transactions deal with crash failures of processes and omission failures in communication, but not any type of arbitrary (or Byzantine) behaviour.

Fault Models

- Lampson proposed a fault model for distributed transactions that accounts for failures of disks, servers and communication.

ACID

- The properties of transactions are:
- **A**tomicity

ACID

- The desirable properties of transactions are:
- **A**tomicity: a transaction must be all or nothing;

ACID

- The desirable properties of transactions are:
- **A**tomicity: a transaction must be all or nothing;
- **C**onsistency:

ACID

- The desirable properties of transactions are:
- **A**tomicity: a transaction must be all or nothing;
- **C**onsistency: a transaction takes the system from one consistent state to another consistent state;

ACID

- The desirable properties of transactions are:
- **A**tomicity: a transaction must be all or nothing;
- **C**onsistency: a transaction takes the system from one consistent state to another consistent state;
- **I**solation:

ACID

- The desirable properties of transactions are:
- **A**tomicity: a transaction must be all or nothing;
- **C**onsistency: a transaction takes the system from one consistent state to another consistent state;
- **I**solation: Each transaction must be performed without interference from other transactions;

ACID

- The desirable properties of transactions are:
- **A**tomicity: a transaction must be all or nothing;
- **C**onsistency: a transaction takes the system from one consistent state to another consistent state;
- **I**solation: Each transaction must be performed without interference from other transactions;
- **D**urability:

ACID

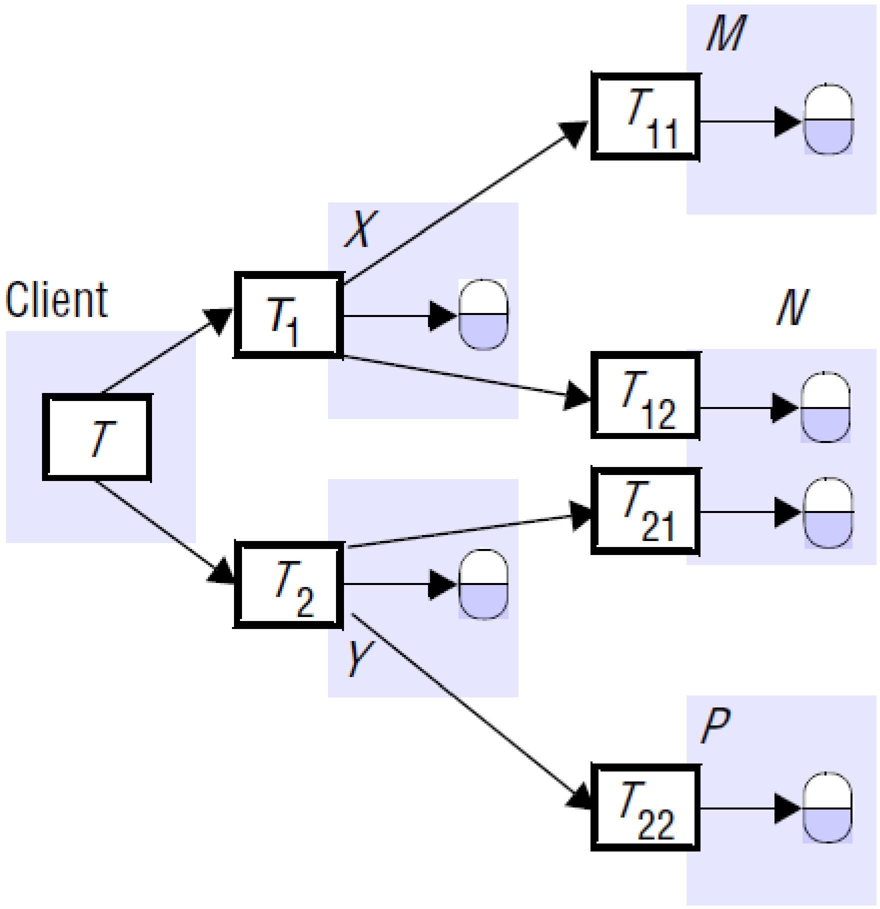
- The desirable properties of transactions are:
- **A**tomicity: a transaction must be all or nothing;
- **C**onsistency: a transaction takes the system from one consistent state to another consistent state;
- **I**solation: Each transaction must be performed without interference from other transactions;
- **D**urability: After a transaction has completed successfully, all its effects are saved in permanent storage
- Härder and Reuter suggested the mnemonic ‘ACID’ to remember these properties.

ACID

- A transaction is either successful or is aborted in one of two ways – the client aborts it or the server aborts it.

Nested Transactions

- Transactions can be nested too.
- Figure shows an example.
- Semantics can be extended to nested transactions also.



Nested Transactions

- An example setting for nested transactions is as follows.
- Consider booking a holiday on a website such as makemytrip.com.
- The holiday may involve multiple entities: flights, ground transport, meals, adventure activities, hotel, and so on.
- Each is typically handled by different agencies and may involve further hierarchical steps.

Nested Transactions

- The rules for committing of nested transactions are:
- A transaction may commit or abort only after its child transactions have completed.
- When a subtransaction completes, it makes an independent decision either to commit provisionally or to abort. Its decision to abort is final.
- When a parent aborts, all of its subtransactions are aborted.
- When a subtransaction aborts, the parent can decide whether to abort or not.
- If the top-level transaction commits, then all of the subtransactions that have provisionally committed can commit too, provided that none of their ancestors has aborted.

Means of Handling Concurrency

- Locks
- Time stamp ordering
- Optimistic Concurrency control

- Read more offline.
  - Distributed Systems: Concepts and Design, Kindberg et al. has a chapter on database systems and transactions

On to Distributed Transactions

- *A distributed transaction* refers to a flat or nested transaction that accesses objects managed by multiple servers.
- When a distributed transaction comes to an end, the atomicity property of transactions requires that either all of the servers involved in the transaction commit the transaction or all of them abort the transaction.
- To achieve this, one of the servers acts as a coordinator to ensure the same outcome at all of the servers.

On to Distributed Transactions

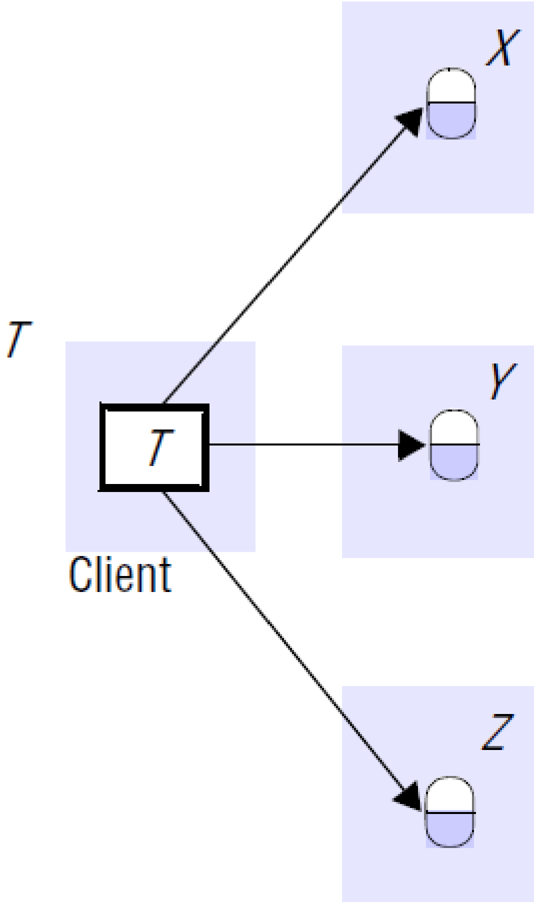
- A quick example of a distributed transaction is as follows.
- Imagine that you want to pay for you cup of tea at the canteen using an UPI application.
- The canteen vendor may have his phone number linked with his bank with Union Bank.
- Your phone number is linked with your account held at SBI.
- The transaction has to debit your account and credit to the canteen vendor account.

Distributed Transactions

- The actions of the coordinator are guided by the protocol chosen.
- A protocol known as the ‘two-phase commit protocol’ is the most commonly used.
- This protocol allows the servers to communicate with one another to reach a joint decision as to whether to commit or abort.

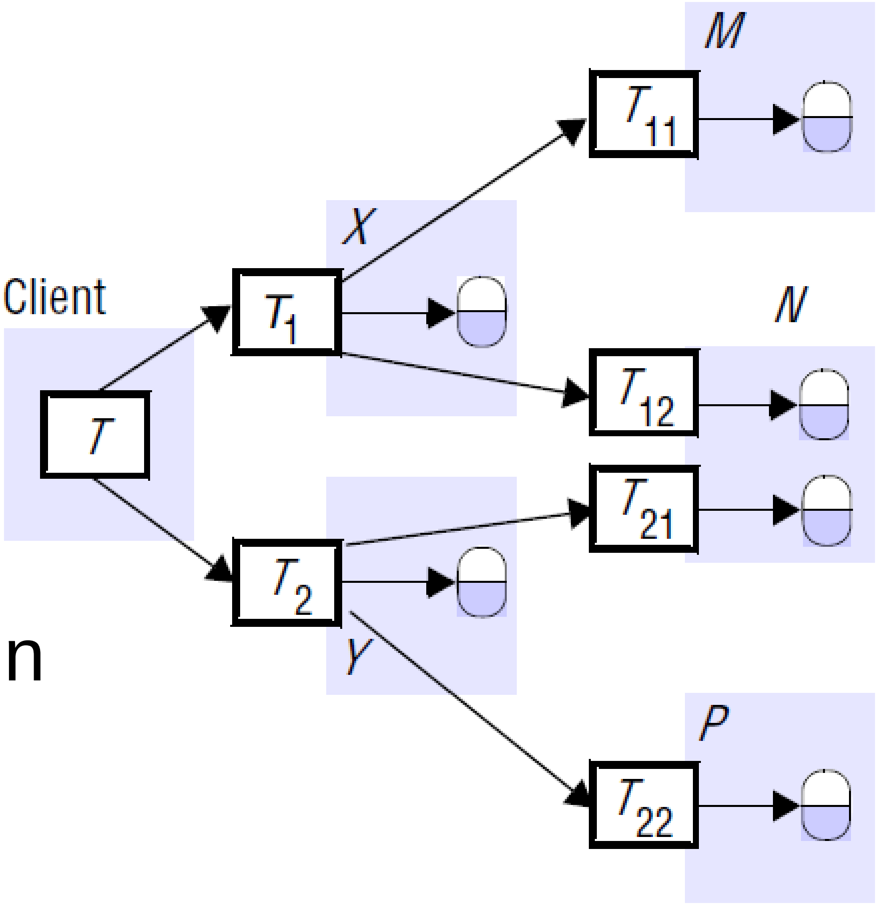
Flat vs Nested Transactions

- In a flat transaction, a client makes requests to more than one server.
- A flat client transaction completes each of its requests before going on to the next one.
- Therefore, each transaction accesses servers' objects sequentially.
- When servers use locking, a transaction can only be waiting for one object at a time.



Flat vs Nested Transactions

- In a nested transaction, the top-level transaction can initiate subtransactions, and each subtransaction can initiate further subtransactions.
- Subtransactions at the same level can run concurrently.
- So T_1 and T_2 are concurrent, and as they invoke objects in different servers, they can run in parallel.
- The four subtransactions T_{11}, T_{12}, T_{21} and T_{22} also run concurrently.



The Coordinator and the Participant

- Servers that execute requests as part of a distributed transaction need to be able to communicate with one another to coordinate their actions when the transaction commits.
- A client starts a transaction by sending an *openTransaction* request to a coordinator.
- The coordinator carries out the *openTransaction* and returns the resulting transaction identifier (TID) to the client.
- Transaction identifiers for distributed transactions must be unique within a distributed system.

The Coordinator and the Participant

- The coordinator that opened the transaction becomes the coordinator for the distributed transaction and at the end is responsible for committing or aborting it.
- Each of the servers that manages an object accessed by a transaction is a participant in the transaction and provides an object to the *participant*.
- The participants are responsible for cooperating with the coordinator in carrying out the commit protocol.

Atomic commit protocols

- Transaction commit protocols are from the early 1970s
- The atomicity property 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 the case of a distributed transaction, the client has requested operations at more than one server.

Atomic commit protocols

- A transaction comes to an end when the client requests that it be committed or aborted.
- A simple way to complete the transaction in an atomic manner is for the coordinator to communicate the commit or abort request to all of the participants in the transaction and to keep on repeating the request until all of them have acknowledged that they have carried it out.
- This is called as *one phase atomic commit protocol*.

Atomic Commit Protocols

- This simple one-phase atomic commit protocol is inadequate.
- It does not allow a server to make a unilateral decision to abort a transaction when the client requests a commit.
- Reasons that prevent a server from being able to commit its part of a transaction generally relate to issues of concurrency control.
  - For example, the resolution of a deadlock can lead to the aborting of a transaction without the client being aware
  - optimistic concurrency control is in use, the failure of validation at a server would cause it to decide to abort the transaction.
  - Finally, the coordinator may not know if a server has crashed and been replaced during the progress of a distributed transaction

Atomic Commit Protocols

- The *two-phase commit protocol* is designed to allow any participant to abort its part of a transaction.

Atomic Commit Protocols

- In the first phase of the protocol, each participant votes for the transaction to be committed or aborted.
- Once a participant has voted to commit a transaction, it is not allowed to abort it.
- Therefore, before a participant votes to commit a transaction, it must ensure that it will eventually be able to carry out its part of the commit protocol, even if it fails and is replaced in the interim.
- A participant in a transaction is said to be in a *prepared* state for a transaction if it will eventually be able to commit it.
- To make sure of this, each participant saves in permanent storage all of the objects that it has altered in the transaction, together with its status – prepared.

Atomic Commit Protocols

- In the second phase of the protocol, every participant in the transaction carries out the joint decision.
- If any one participant votes to abort, then the decision must be to abort the transaction.
- If all the participants vote to commit, then the decision is to commit the transaction.

Atomic Commit Protocols

- The problem is to ensure that all of the participants vote and that they all reach the same decision.
- This is fairly simple if no errors occur, but the protocol must work correctly even when some of the servers fail, messages are lost or servers are temporarily unable to communicate with one another.

Atomic Commit Protocols

- Commit protocols are designed to work in an asynchronous system in which
  - servers may crash and
  - messages may be lost.
  - It is assumed that an underlying request-reply protocol removes corrupt and duplicated messages.
- There are no Byzantine faults: servers either crash or obey the messages they are sent.

Atomic Commit Protocols

- The two-phase commit protocol is an example of a protocol for reaching a consensus.
- Recall that consensus cannot be reached in an asynchronous system if processes sometimes fail.
- However, the two-phase commit protocol does reach consensus under those conditions.
- This is because processes that crash are assumed to be replaced with a new process.
- The *new* process has its state information set from the state saved in permanent storage and information held by other processes.

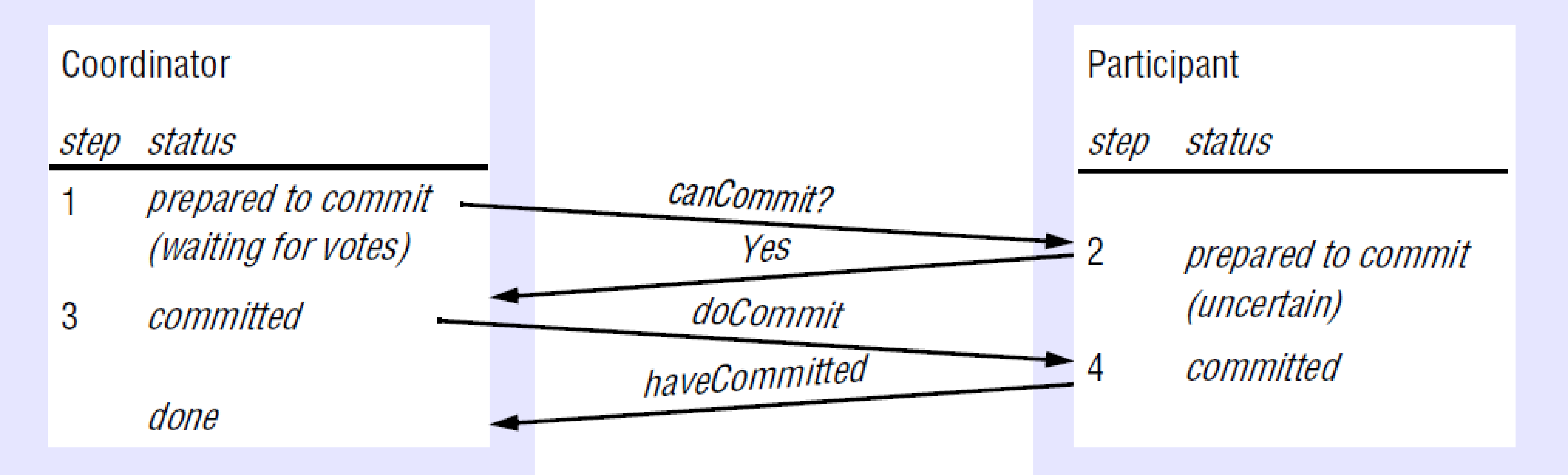
Two-Phase Commit Protocol

- *Phase 1 (voting phase):*
  1. The coordinator sends a *canCommit?* request to each of the participants in the transaction.
  2. When a participant receives a *canCommit?* request it replies with its vote (*Yes* or *No*) to the coordinator.
  3. Before voting *Yes*, it prepares to commit by saving objects in permanent storage.
  4. If the vote is *No*, the participant aborts immediately.

Two-Phase Commit Protocol

- *Phase 2 (completion according to outcome of vote):*
- The coordinator collects the votes (including its own).
  - If there are no failures and all the votes are *Yes*, the coordinator decides to commit the transaction and sends a *doCommit* request to each of the participants.
  - Otherwise, the coordinator decides to abort the transaction and sends *doAbort* requests to all participants that voted *Yes*.
- Participants that voted *Yes* are waiting for a *doCommit* or *doAbort* request from the coordinator.
- When a participant receives one of these messages it acts accordingly and, in the case of commit, makes a *haveCommitted* call as confirmation to the coordinator.

Two-Phase Commit Protocol



Coordinator

step	*status*
1	*prepared to commit* *(waiting for votes)*
3	*committed*
	done

canCommit?

Yes

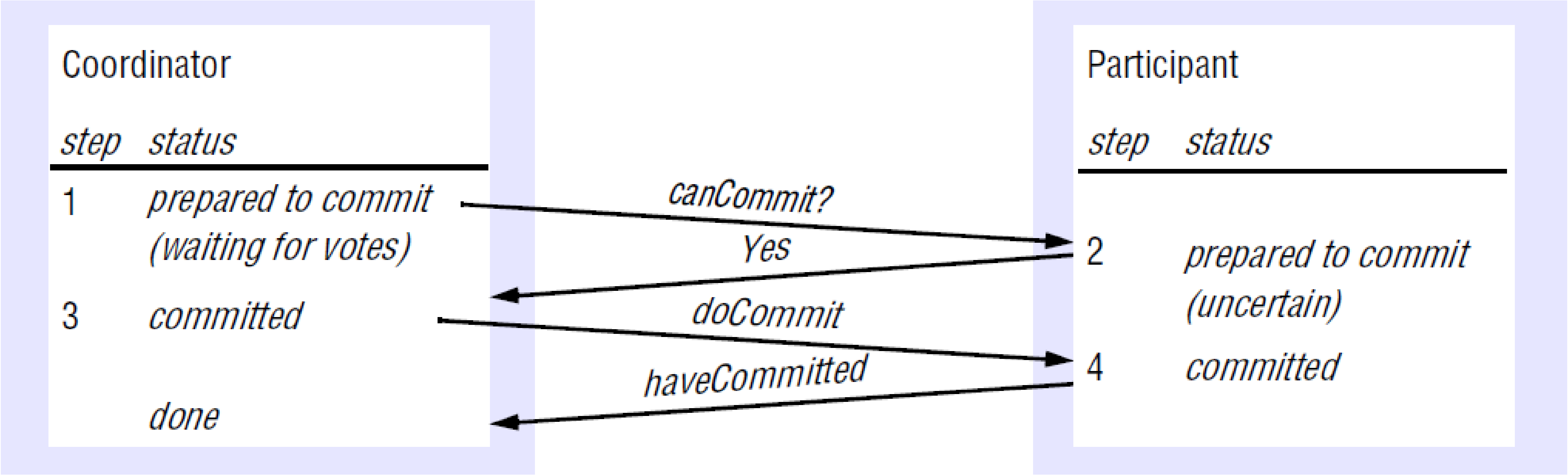
doCommit

haveCommitted

Participant

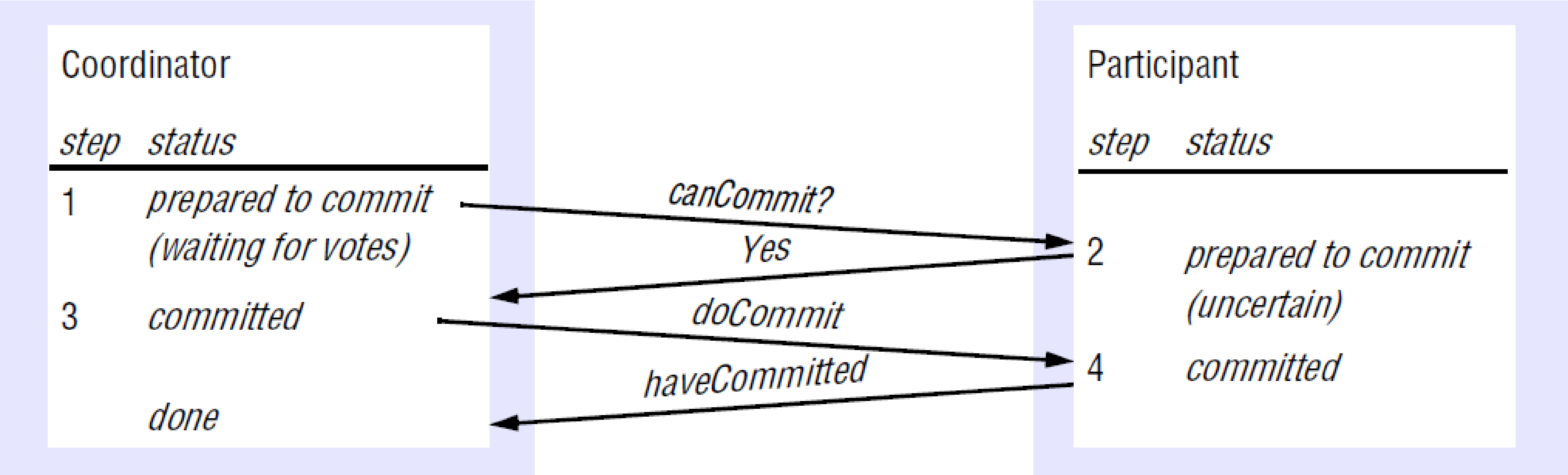
step	*status*
2	*prepared to commit* *(uncertain)*
4	*committed*

Two-Phase Commit Protocol



- Timeout situations
- Consider first the situation where a participant has voted *Yes* and is waiting for the coordinator to report on the outcome of the vote by telling it to commit or abort the transaction
- Such a participant is *uncertain* of the outcome and cannot proceed any further until it gets the outcome of the vote from the coordinator.

Two-Phase Commit Protocol



Coordinator

step	status
1	*prepared to commit (waiting for votes)*
3	*committed*
	done

Participant

step	status
2	*prepared to commit (uncertain)*
4	*committed*

- The participant can make a *getDecision* request to the coordinator to determine the outcome of the transaction.
- When it gets the reply, it continues the protocol at step 4. If the coordinator has failed, the participant will not be able to get the decision until the coordinator is replaced, which can result in extensive delays for participants in the *uncertain* state.

Timeout Scenarios

- Another point at which a participant may be delayed is when it has carried out all its client requests in the transaction but has not yet received a *canCommit?* call from the coordinator.
- As the client sends the *closeTransaction* to the coordinator, a participant can only detect such a situation if it notices that it has not had a request in a particular transaction for a long time
- As no decision has been made at this stage, the participant can decide to *abort* unilaterally

Timeout Scenarios

- The coordinator may be delayed when it is waiting for votes from the participants.
- As it has not yet decided the fate of the transaction it may decide to abort the transaction after some period of time.
- It must then announce *doAbort* to the participants who have already sent their votes.
- Some tardy participants may try to vote *Yes* after this, but their votes will be ignored and they will enter the *uncertain* state as described above.

Performance

- The two-phase commit protocol involving *N* participants can be completed with *N canCommit?* messages and replies, followed by *N doCommit* messages.
- The cost in messages is proportional to 4*N*, and the cost in time is four rounds of messages.
- In the worst case, there may be arbitrarily many server and communication failures during the two-phase commit protocol.
- However, the protocol is designed to tolerate a succession of failures.
- The protocol is guaranteed to complete eventually, although it is not possible to specify a time limit within which it will be completed.