Chapter 13: Distributed Transactions

- Introduction
- Flat and nested distributed transactions
- Atomic commit protocols
- Concurrency control in distributed transactions
- Summary

Introduction

- Distributed transaction
  - A flat or nested transaction that accesses objects managed by multiple servers
- Atomicity of transaction
  - All or nothing for all involved servers
  - Two phase commit
- Concurrency control
  - Serialize locally + serialize globally

Chapter 13: Distributed Transactions

- Introduction
- Flat and nested distributed transactions
- Atomic commit protocols
- Concurrency control in distributed transactions

Flat and nested distributed transactions

- Flat transaction
- Nested transaction
- Nested banking transaction
  - The four subtransactions run in parallel

The architecture of distributed transactions

- **The coordinator**
  - Accept client request
  - Coordinate behaviors on different servers
  - Send result to client
  - Record a list of references to the participants
- **The participant**
  - One participant per server
  - Keep track of all recoverable objects at each server
  - Cooperate with the coordinator
  - Record a reference to the coordinator
- **Example**

Chapter 13: Distributed Transactions

- Introduction
- Flat and nested distributed transactions
- Atomic commit protocols
- Concurrency control in distributed transactions

One-phase atomic commit protocol

- The protocol
  - Client request to end a transaction
  - The coordinator communicates the commit or abort request to all of the participants and to keep on repeating the request until all of them have acknowledged that they had carried it out
- The problem
  - some servers commit, some servers abort
    - How to deal with the situation that some servers decide to abort?

Introduction to two-phase commit protocol

- **Allow for any participant to abort**
- **First phase**
  - Each participant votes to commit or abort
- **The second phase**
  - All participants reach the same decision
    - If any one participant votes to abort, then all abort
    - If all participants votes to commit, then all commit
  - The challenge
    - work correctly when error happens
- **Failure model**
  - Server crash, message may be lost, no arbitrary fails

The two-phase commit protocol

- When the client request to abort
  - The coordinator informs all participants to abort
- When the client request to commit
  - First phase
    - The coordinator ask all participants if they prepare to commit
    - If a participant prepare to commit, it saves in the permanent storage all of the objects that it has altered in the transaction and reply *yes*. Otherwise, reply *no*
  - Second phase
    - The coordinator tell all participants to commit (or abort)

The two-phase commit protocol … *continued*

- Operations for two-phase commit protocol
- The two-phase commit protocol
  - Record updates that are prepared to commit in the permanent storage
    - When the server crash, the information can be retrieved by a new process
    - If the coordinator decide to commit, all participants will commit eventually

Timeout actions in the two-phase commit protocol

- Communication in two-phase commit protocol
- **New processes to mask crash failure**
  - Crashed process of coordinator and participant will be replaced by new processes
- **Time out for the participant**
  - Timeout of waiting for *canCommit*: abort
  - Timeout of waiting for *doCommit*
    - Uncertain status: Keep updates in the permanent storage
    - *getDecision* request to the coordinator
- **Time out for the coordinator**
  - Timeout of waiting for vote result: abort
  - Timeout of waiting for *haveCommited*: do nothing
    - The protocol can work correctly without the confirmation

Two-phase commit protocol for nested transactions

- Nested transaction semantics
  - Subtransaction
    - Commit provisionally
    - abort
  - Parent transaction
    - Abort: all subtransactions abort
    - Commit: exclude aborting subtransactions
- Distributed nested transaction
  - When a subtransaction completes
    - provisionally committed updates are not saved in the permanent storage

Distributed nested transactions commit protocol

- **A coordinator for a subtransaction will provide an operation to open a sunbtransaction**
- **Open subtransaction(*trans*)→*subTrans***
  - Open a subtransaction whose parents is *trans* and returns a unique subtransaction identifier.
- **getStatus(*trans*)→*commited, aborted, provisional***
  - Asks the coordinator to report on the status of the transactions *trans*. Return values representing one of the following:committed, aborted, provisional

Distributed nested transactions commit protocol

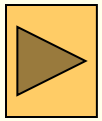
- Each subtransaction
  - If commit provisionally
    - Report the status of it and its descendants to its parent
  - If abort
    - Report abort to its parent
- Top level transaction
  - Receive a list of status of all subtransactions
  - Start two-phase commit protocol on all subtransactions that have committed provisionally

Example of a distributed nested transactions

- The execution process
- The information held by each coordinator
  - Top level coordinator
    - The participant list: the coordinators of all the subtransactions in the tree that have provisionally committed but do not have aborted parent
  - Two-phase commit protocol
    - Conducted on the participant of T, T_1 and T_{12}

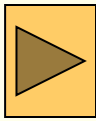
Different two-phase commit protocol

- Hierarchic two-phase commit protocol
  - Messages are transferred according to the hierarchic relationship between successful participants
  - The interface



Different two-phase commit protocol

- **Flat two-phase commit protocol(The interface)**
  - Messages are transferred from top-level coordinator to all successful participants directly
- **If the participant has any provisionally committed transactions that are descendents of the top-level transaction, trans**
  - Check that they do not have aborted ancestors in the *abortList*,then prepare to commit
  - Those with aborted ancestors are aborted;
  - Send a *Yes* vote to coordinator
- **If the participant doesn't have a provisionally committed descendent ,send *No* to coordinator**



Chapter 13: Distributed Transactions

- Introduction
- Flat and nested distributed transactions
- Atomic commit protocols
- Concurrency control in distributed transactions

Serial equivalence on all servers

- Objective
  - Serial equivalence on all involved servers
    - If transaction T is before transaction U in their conflicting access to objects at one of the server then they must be in that order at all of the servers whose objects are accessed in a conflicting manner by both T and U
- Approach
  - Each server apply concurrency control it its own objects
  - All servers coordinate together to reach the objective

Lock

- Each participant locks on objects locally
  - strict two phase locking scheme
- Atomic commit protocol
  - a server can not release any locks until it knows that the transaction has been committed or aborted at all
- Distributed deadlock
- either deadlock or serial equivalence

Timestamp ordering concurrency control

- **Globally unique transaction timestamp**
  - Be issued to the client by the first coordinator accessed by a transaction
  - The transaction timestamp is passed to the coordinator at each server
  - Each server accesses shared objects according to the timestamp
- **Resolution of a conflict**
  - Abort a transaction from all servers

Optimistic concurrency control

- The validation
  - takes place during the first phase of two phase commit protocol
  - Commitment deadlock

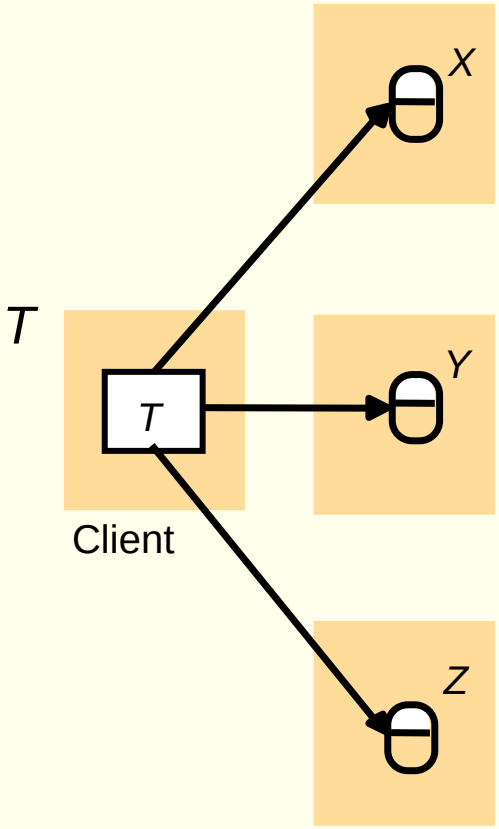
T	U
Read (A) At X	Read (B) At Y
Write (A)	Write (B)
Read(B) At Y	Read(A) At X
Write (B)	Write (A)

Optimistic concurrency control

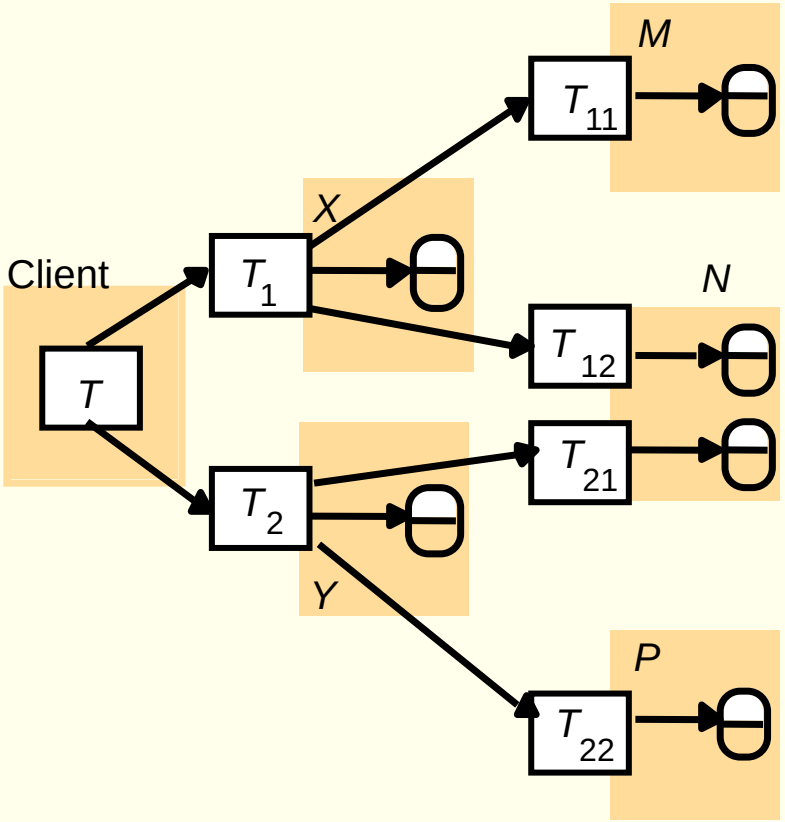
- Parallel validation
  - Suitable for distributed transaction
  - write-write conflict must be checked as well as write-read for backward validation
  - Possibly different validation order on different server
    - Measure1:global validation check after individual server is serializable.
    - measure2: each server validates according to a globally unique transaction number of each transaction

Distributed transactions

(a) Flat transaction



(b) Nested transactions



Nested banking transaction

T = openTransaction

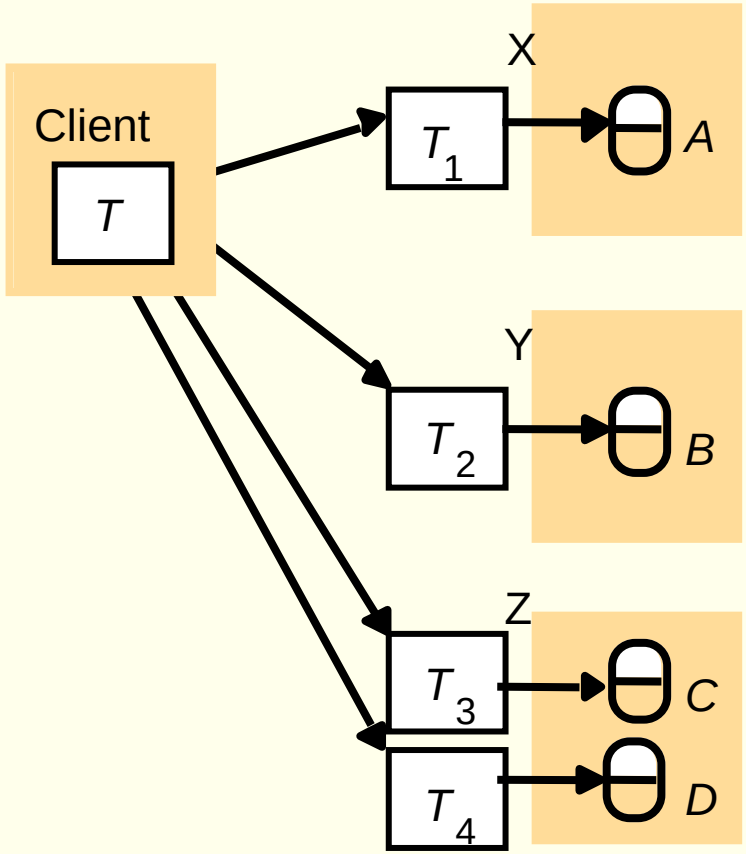
openSubTransaction
a.withdraw(10);

openSubTransaction
b.withdraw(20);

openSubTransaction
c.deposit(10);

openSubTransaction
d.deposit(20);

closeTransaction



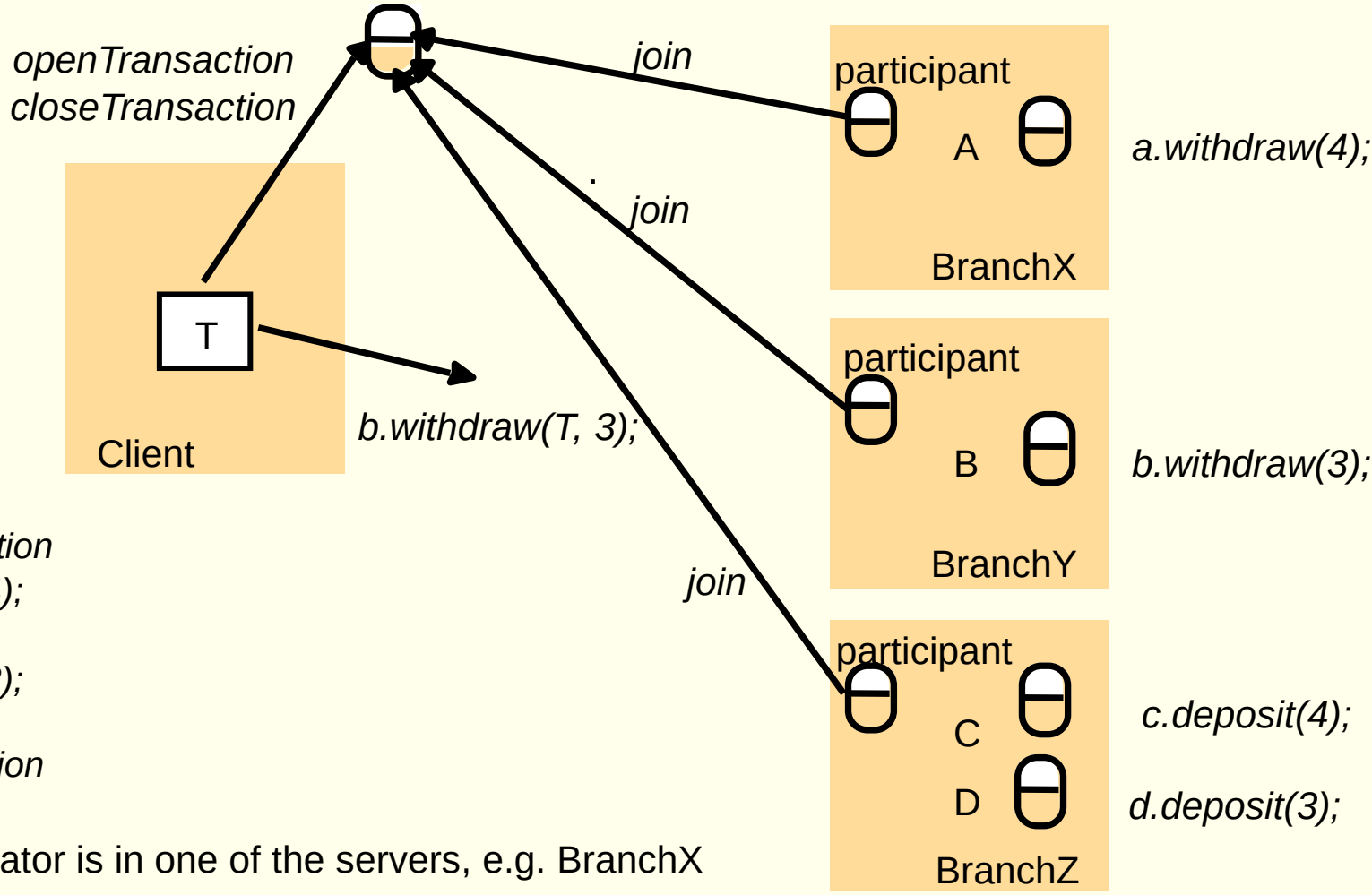
a.withdraw(10)

b.withdraw(20)

c·deposit(10)

d.deposit(20)

A distributed banking transaction



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

Note: the coordinator is in one of the servers, e.g. BranchX

Operations for two-phase commit protocol

canCommit?(trans)-> Yes / No

Call from coordinator to participant to ask whether it can commit a transaction. Participant replies with its vote.

doCommit(trans)

Call from coordinator to participant to tell participant to commit its part of a transaction.

doAbort(trans)

Call from coordinator to participant to tell participant to abort its part of a transaction.

haveCommitted(trans, participant)

Call from participant to coordinator to confirm that it has committed the transaction.

getDecision(trans) -> Yes / No

Call from participant to coordinator to ask for the decision on a transaction after it has voted *Yes* but has still had no reply after some delay. Used to recover from server crash or delayed messages.

Operations for two-phase commit protocol

Phase 1 (voting phase):

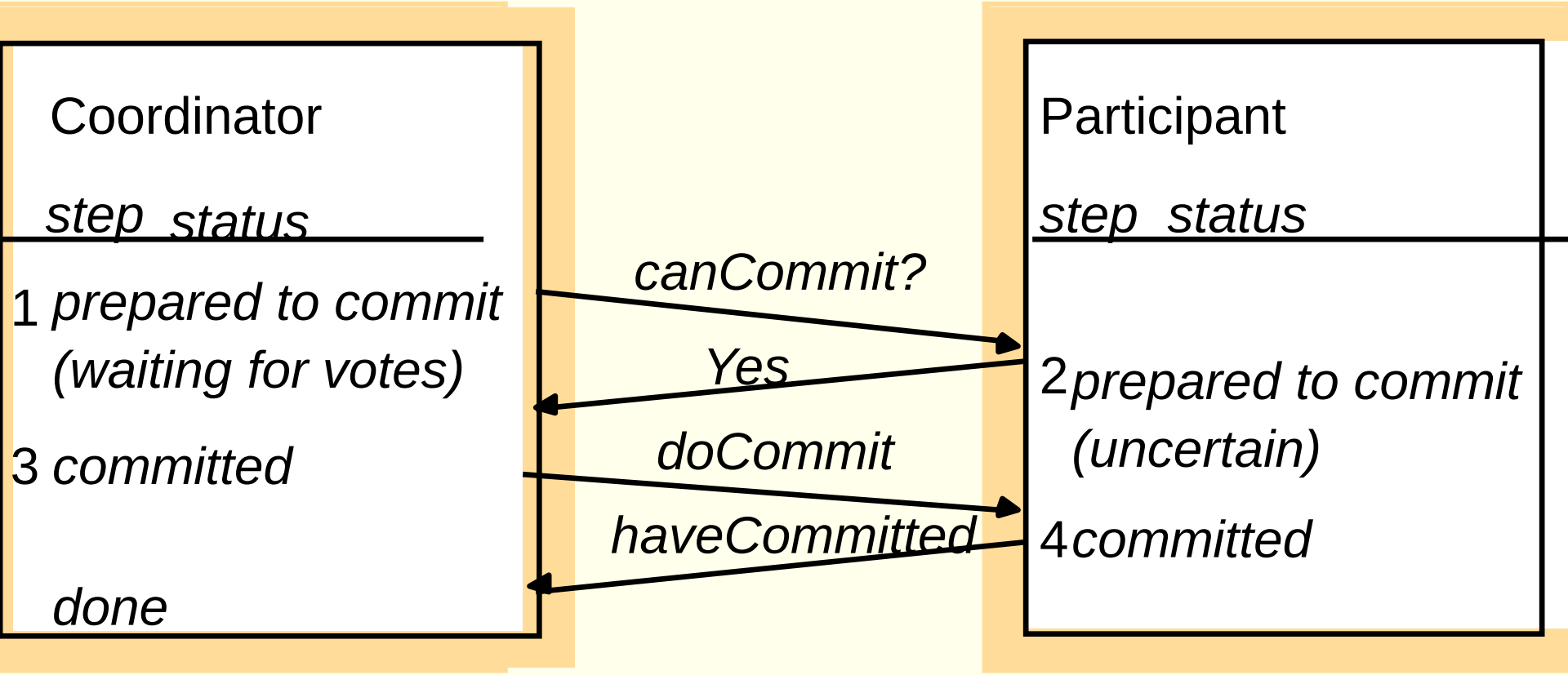
1. The coordinator sends a *canCommit*? request to each of the participants in the transaction.
2. When a participant receives a *canCommit*? request it replies with its vote (*Yes* or *No*) to the coordinator. Before voting *Yes*, it prepares to commit by saving objects in permanent storage. If the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

Operations for two-phase commit protocol

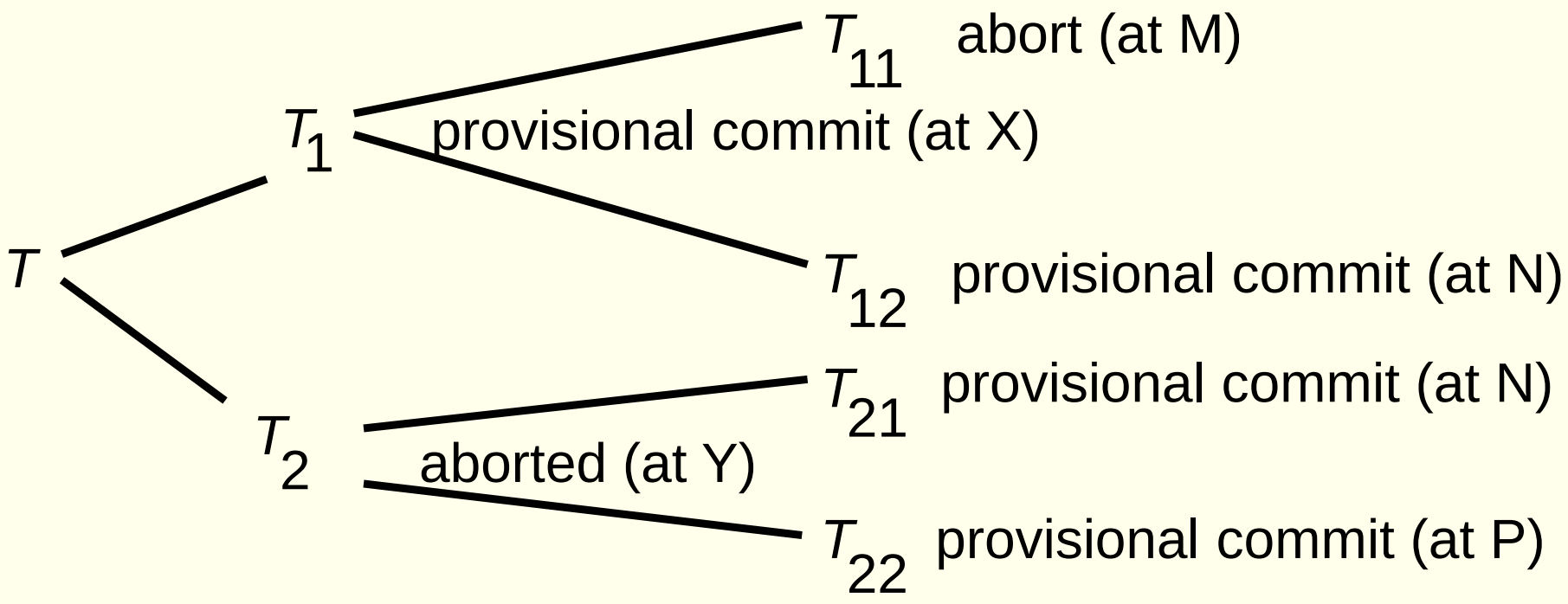
Phase 2 (completion according to outcome of vote):

3. The coordinator collects the votes (including its own).
   (a) 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.
   (b) Otherwise the coordinator decides to abort the transaction and sends *doAbort* requests to all participants that voted *Yes*.
4. 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.

Communication in two-phase commit protocol



Transaction *T* decides whether to commit



Transaction *T* decides whether to commit

Coordinator of transaction	Child transactions	Participant	Provisional commit list	Abort list
T	T_1, T_2	yes	T_1, T_{12}	T_{11}, T_2
T_1	T_{11}, T_{12}	yes	T_1, T_{12}	$T_{11},$
T_2	$T_{21} T_{22}$	no (aborted)		$, T_2$
T_{11}		no (aborted)		$T_{11},$
$T_{12} T_{21}$		T_{12} but not T_{21}	T_{21}, T_{12}	
T_{22}		no (parent aborted)	T_{22}	

canCommit? for hierarchic two-phase commit protocol

canCommit?(trans, subTrans) -> Yes / No

Call a coordinator to ask coordinator of child subtransaction whether it can commit a subtransaction *subTrans*. The first argument *trans* is the transaction identifier of top-level transaction. Participant replies with its vote *Yes* / *No*.

canCommit? for flat two-phase commit protocol

canCommit?(trans, abortList) -> Yes / No

Call from coordinator to participant to ask whether it can commit a transaction. Participant replies with its vote *Yes* / *No*.

The *abortList* is used by the coordinator of the participants to filter aborted subtransactions if multiple participants share a same coordinator

Example of a distributed deadlock

T			*U*		
Write(A)	*At X*	Locks A			
			Write(B)	*At Y*	Locks *B*
Read(B)	*At Y*	Wait for *U*			
			Read(A)	*At X*	Wait for *T*