06-06798 Distributed Systems

Lecture 13:
Transactions
in a Distributed Environment

Overview

- Distributed transactions
  - multiple servers
  - atomicity
- Atomic commit protocols
  - 2-phase commit
- Concurrency control
  - locking
  - timestamping
  - optimistic concurrency control
- Other issues (deadlocks, recovery)

Transactions

- Definition
  - **sequence** of server operations
  - originate from databases (banking, airline reservation, etc)
  - atomic operations or sequences (free from interference by other clients and server crashes)
  - durable (when completed, saved in permanent storage)
- Issues in transaction processing
  - need to maximise concurrency while ensuring consistency
    - serial equivalence/serializability (= same effect as a serial execution)
  - must be recoverable from failures

Distributed transactions

- Definition
  - access objects which are managed by multiple servers
  - can be flat or nested
- Sources of difficulties
  - all servers must agree to commit or abort
    - two-phase commit protocol
  - concurrency control in a distributed environment
    - locking, timestamps
    - optimistic concurrency control
  - failures!
    - deadlocks, recovery from aborted transactions

Transaction handling

- Requires coordinator server, with *open/close/abort*
- Start new transaction (returns unique TID)

  openTransaction() -> trans;
- Then invoke operations on recoverable objects

  A.withdraw(100);

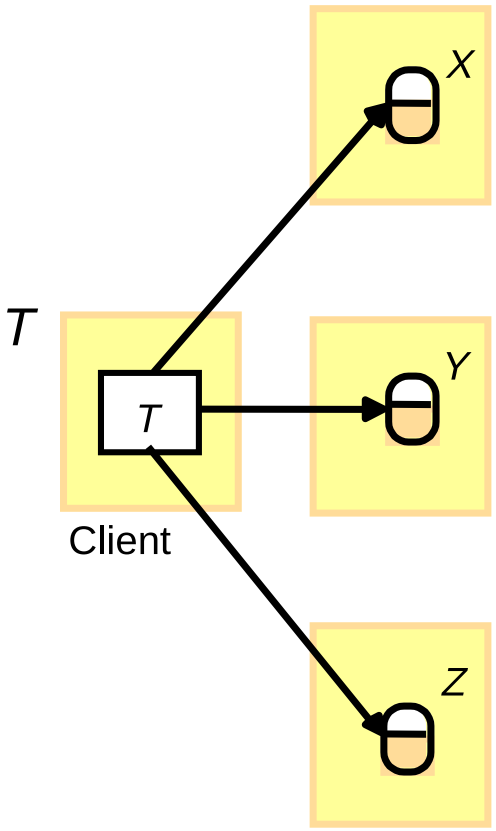
  B.deposit(300)
- If all goes well end transaction (*commit* or *abort)*
  - *closeTransaction(trans) -> (commit, abort);*
- Otherwise
  - *abortTransaction(trans);*

Distributed transactions

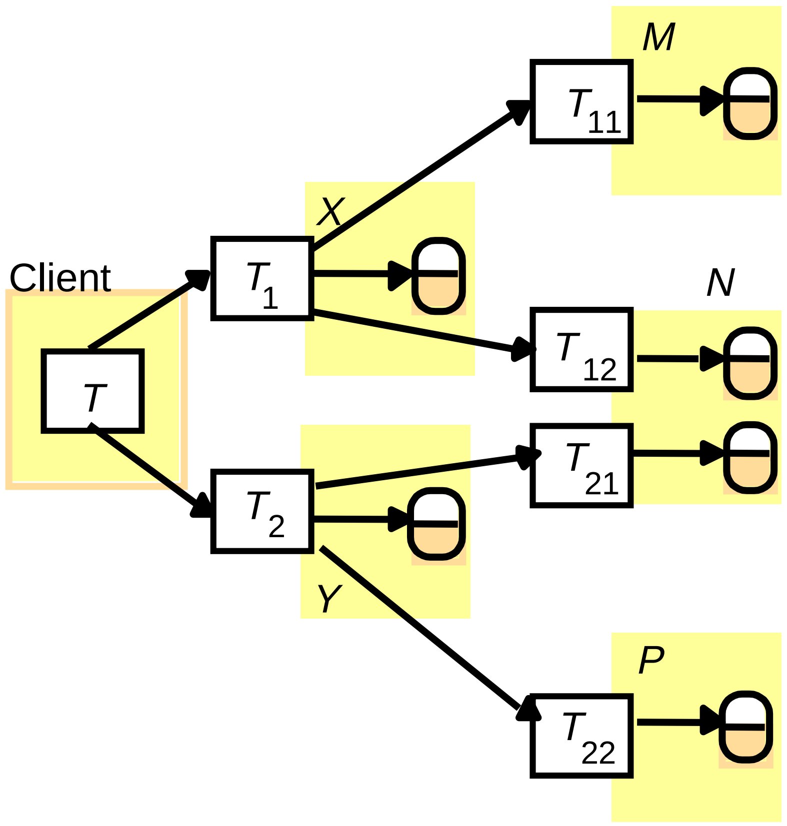
- Flat structure:
  - client makes requests to more than one server
  - request completed before going on to next
  - sequential access to objects
- Nested structure:
  - arranged in levels: top level can open sub-transactions
  - any depth of nesting
  - objects in different servers can be invoked in parallel
  - better performance

Distributed transactions

(a) Flat transaction



(b) Nested transactions



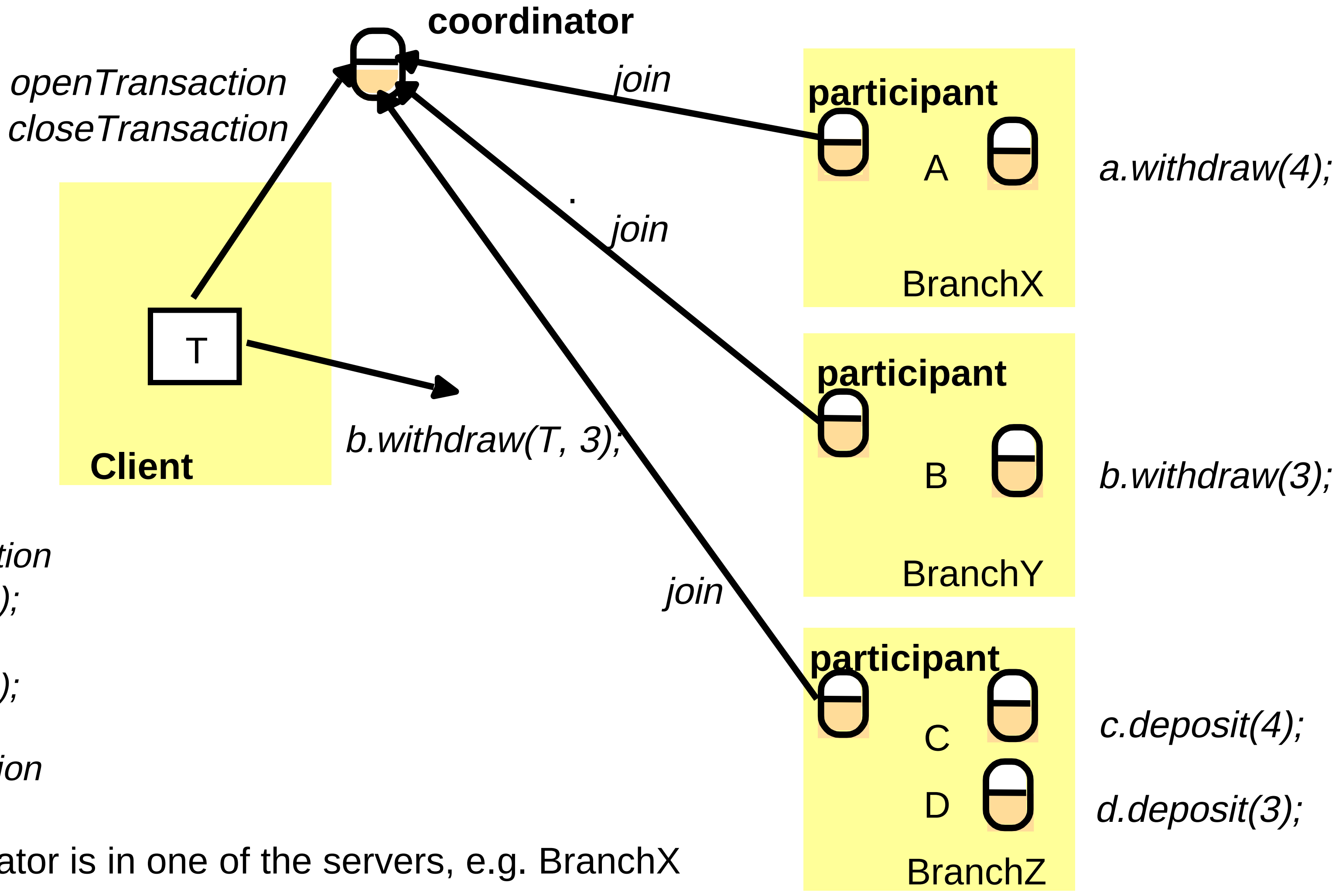
E.g. T_{11}, T_{12} can run in parallel

How it works...

- Client
  - issues *openTransaction()* to coordinator in any server
  - coordinator executes it and returns unique TID to client

    TID = server IP address + unique transaction ID
- Servers
  - communicate with each other
  - keep track of who is who
  - coordinator: responsible for commit/abort at the end
  - participant: can *join(Trans, RefToParticipant)*
    - manages object accessed in transaction
    - keeps track of recoverable objects
    - cooperates with coordinator

Distributed flat banking transaction



coordinator

openTransaction
closeTransaction

join

join

join

T

Client

b.withdraw(T, 3);

participant

A

BranchX

a.withdraw(4);

participant

B

BranchY

b.withdraw(3);

participant

C

D

BranchZ

c.deposit(4);

d.deposit(3);

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

One-phase commit

- Distributed transactions
  - multiple servers, must either be committed or aborted
- One-phase commit
  - coordinator communicates commit/abort to participants
  - keeps repeating the request until all acknowledged
- But… server cannot abort part of a transaction:
  - when the server crashed and has been replaced...
  - when deadlock has been detected and resolved…
- Problem
  - when part aborted, the whole transaction may have to be aborted

Two-phase commit

- Phase 1 (voting phase)
  - (1) coordinator sends *canCommit?* to participants
  - (2) participant replies with vote (*Yes* or *No*); before voting *Yes* prepares to commit by saving objects in permanent storage, and if *No* aborts
- Phase 2 (completion according to outcome of vote)
  - (3) coordinator collects votes (including own)
    - if no failures and all *Yes*, sends *doCommit* to participants
    - otherwise, sends *doAbort* to participants
  - (4) participants that voted Yes wait for *doCommit* or *doAbort* and act accordingly; confirm their action to coordinator by sending *haveCommitted*

Communication in 2-phase protocol

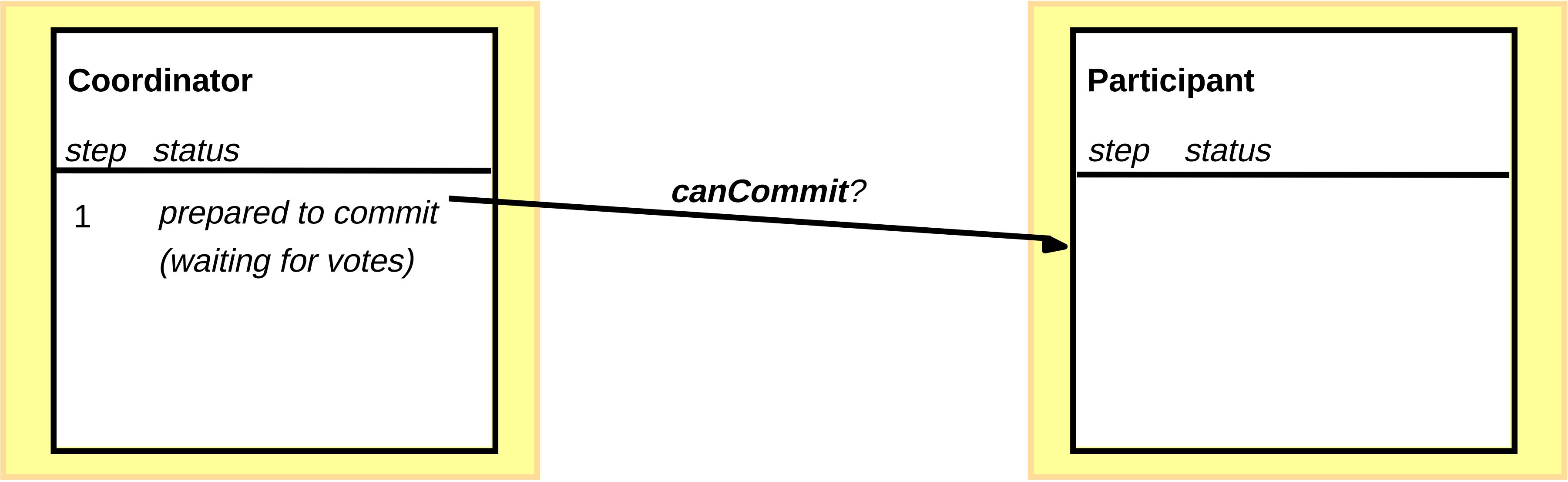
Coordinator

step	*status*

Participant

step	*status*

Communication in 2-phase protocol



Coordinator

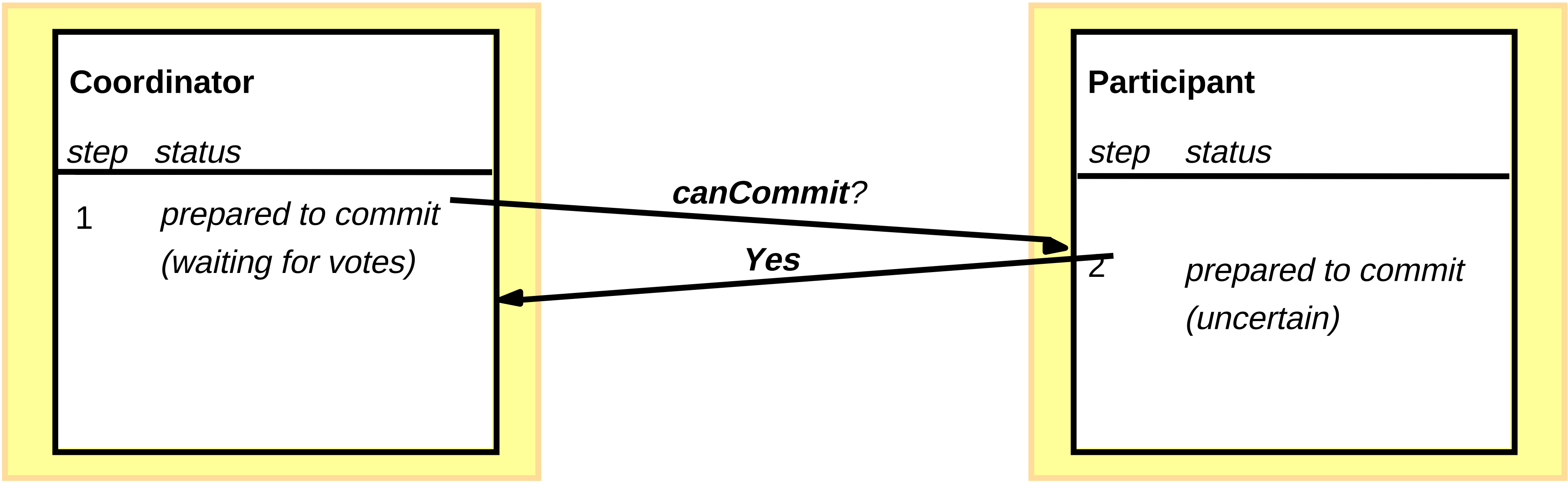
step	*status*
1	*prepared to commit* *(waiting for votes)*

canCommit*?*

Participant

step	*status*

Communication in 2-phase protocol



Coordinator

step *status*

1 *prepared to commit (waiting for votes)*

canCommit*?*

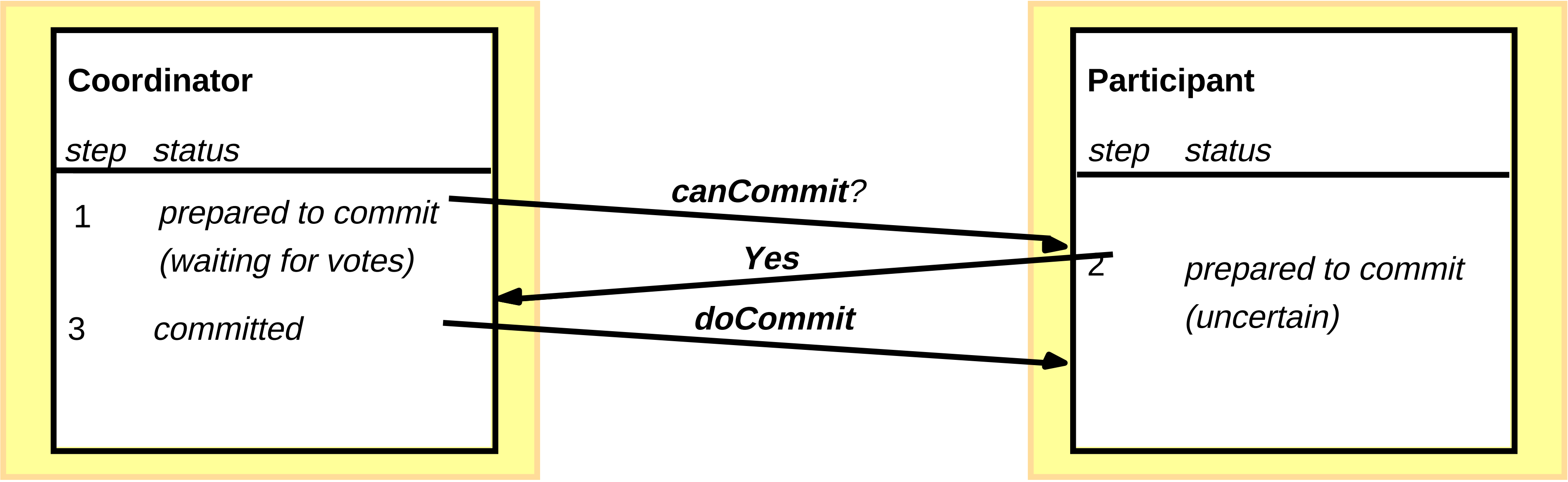
Yes

Participant

step *status*

2 *prepared to commit (uncertain)*

Communication in 2-phase protocol



Coordinator

step *status*

1 *prepared to commit*
(waiting for votes)

3 *committed*

canCommit*?*

Yes

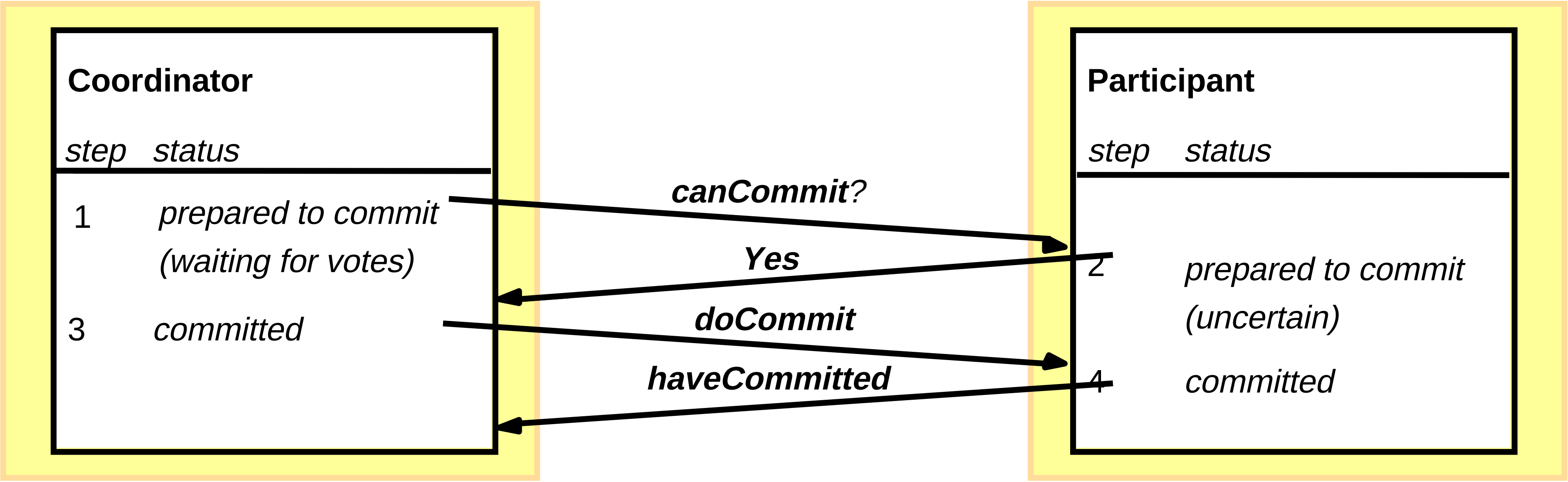
doCommit

Participant

step *status*

2 *prepared to commit*
(uncertain)

Communication in 2-phase protocol



Coordinator

step *status*

1 *prepared to commit*
(waiting for votes)

3 *committed*

canCommit*?*

Yes

doCommit

haveCommitted

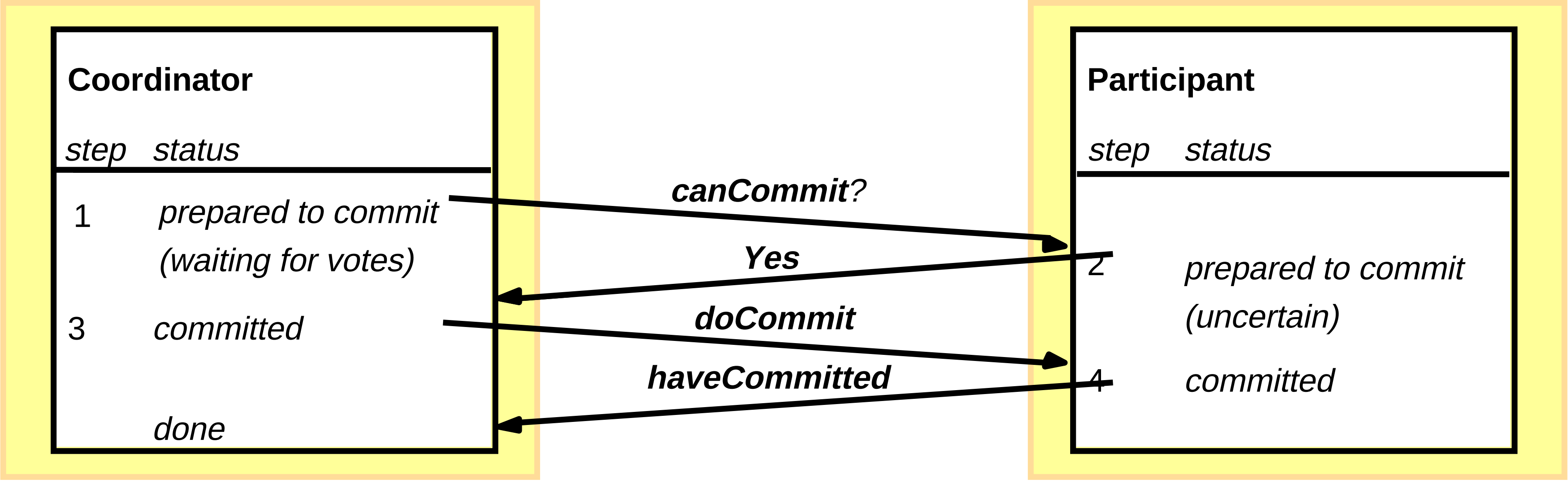
Participant

step *status*

2 *prepared to commit*
(uncertain)

4 *committed*

Communication in 2-phase protocol



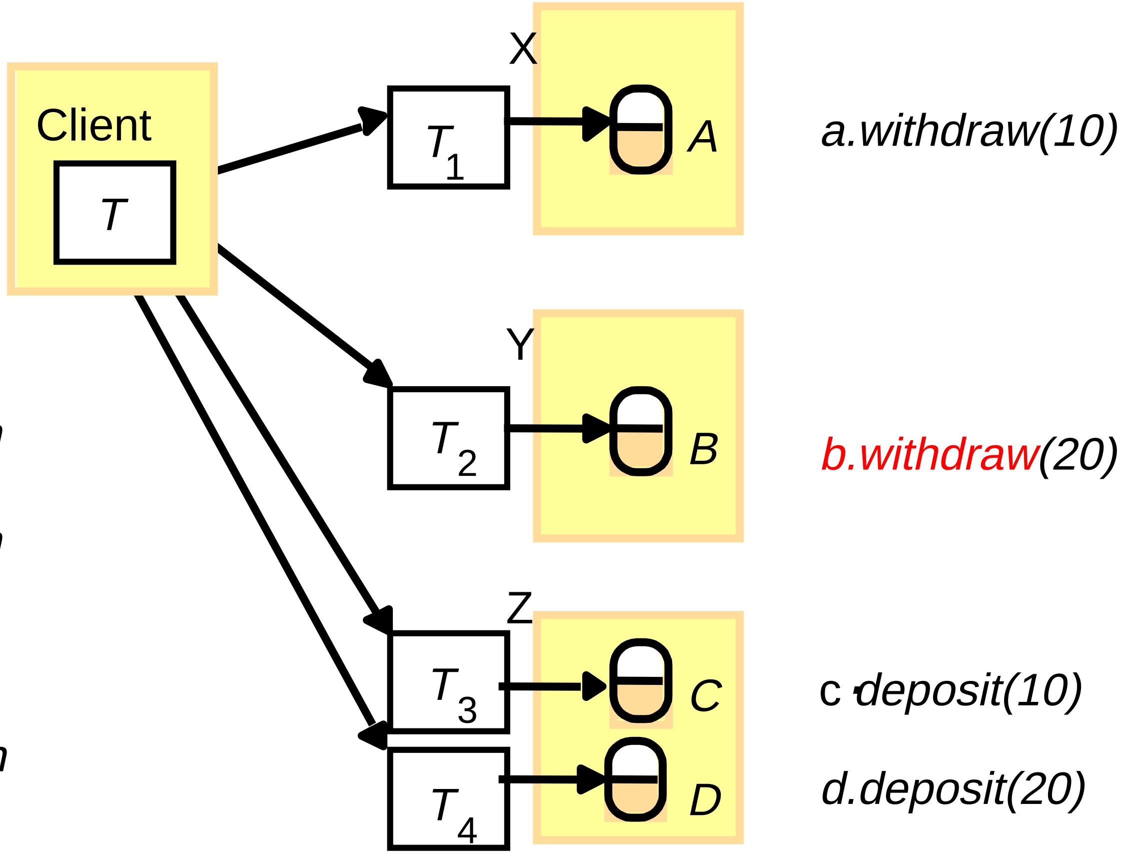
What can go wrong...

- In distributed systems
  - objects stored/managed at different servers
- Server crashes
  - participant: save in permanent storage when preparing to commit, retrieve data after crash
  - coordinator: delay till replaced, or cooperative approach
- Messages fail to arrive (server crash or link failure)
  - use timeout for each step that may block (but no reliable failure detector, asynchronous communication)
  - if uncertain, participant prompts coordinator by *getDecision*
  - if in doubt (e.g. initial *canCommit*? or votes missing), abort!

Nested transactions

- Top-level transaction
  - starts subtransactions with unique TID (extension of the parent TID)
  - subtransaction joins parent transaction
  - completes when all subtransactions have completed
  - can commit even if one of its subtransactions aborted...
- Subtransactions
  - can be independent (e.g. act on different bank accounts)
  - can execute in parallel, at different servers
  - can provisionally commit or abort
  - if parent aborts, must abort too

Nested banking transaction



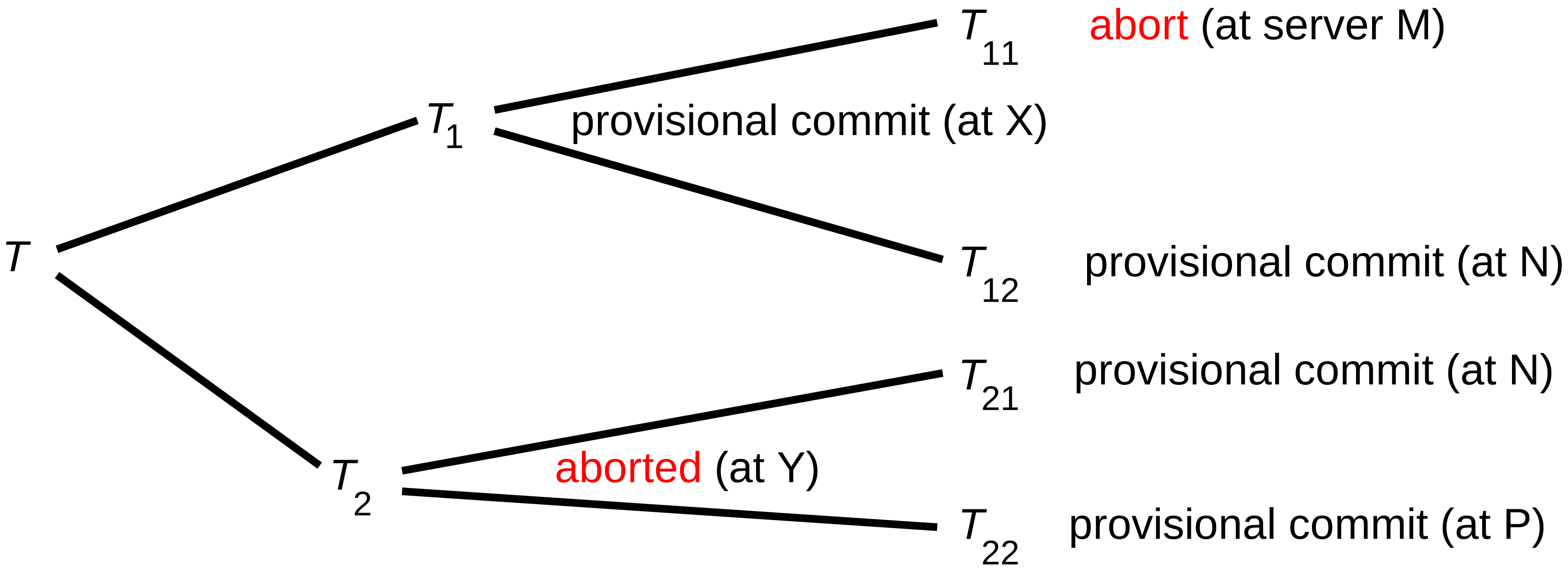
```
T = openTransaction
    openSubTransaction
        a.withdraw(10);
    openSubTransaction
        b.withdraw(20);
    openSubTransaction
        c.deposit(10);
    openSubTransaction
        d.deposit(20);
closeTransaction
```

If b.withdraw aborts due to insufficient funds,
no need to abort the whole transaction

Nested two-phase commit

- Used to decide when top-level transaction commits
- Top-level transaction
  - is coordinator in two-phase commit
  - knows all subtransactions that joined
  - keeps record of subtransaction info
- Subtransactions
  - report status back to parent
  - when abort: reports abort, ignoring children status (now orphans)
  - when provisionally commit: reports status of all child subtransactions

Transaction T decides to commit



T

T_1

T_{11} abort (at server M)

provisional commit (at X)

T_{12} provisional commit (at N)

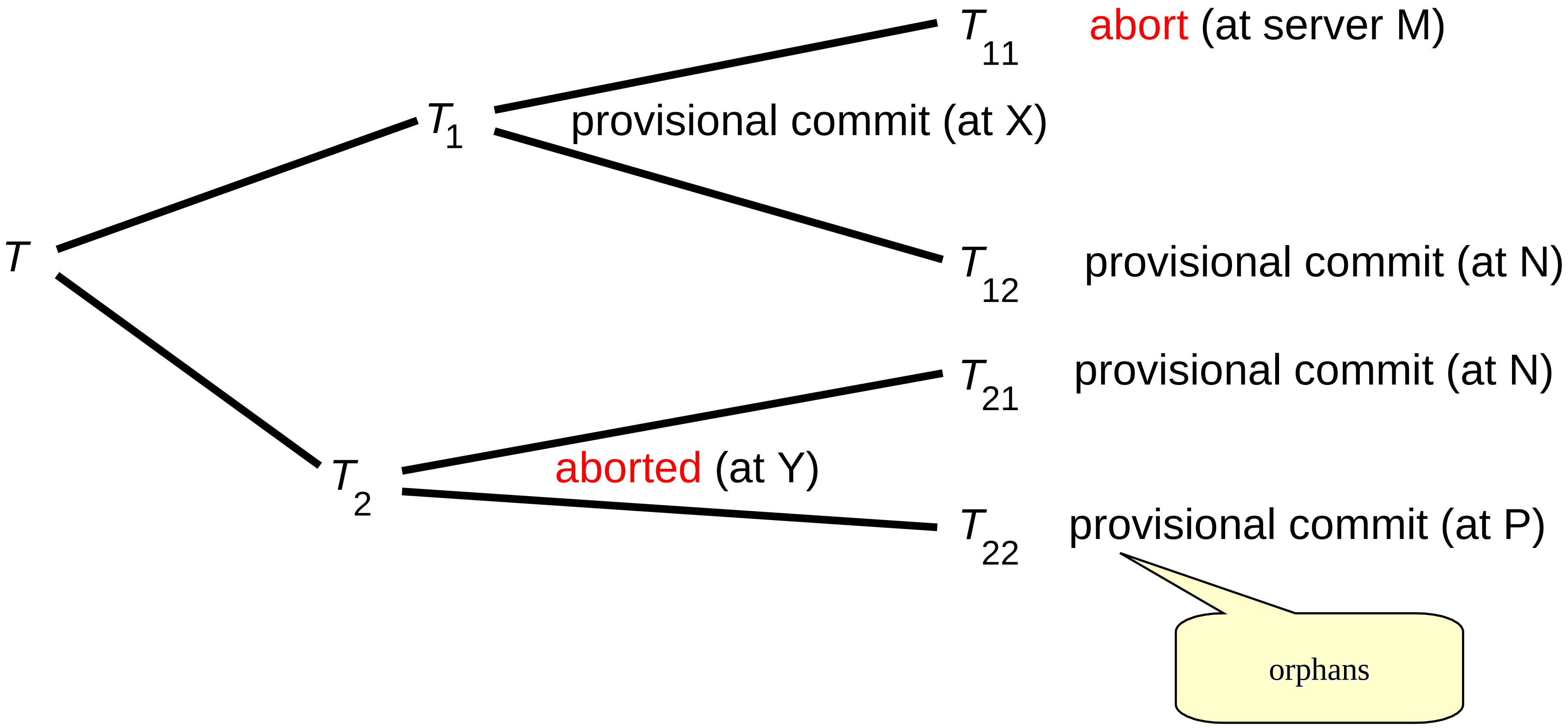
T_2

T_{21} provisional commit (at N)

aborted (at Y)

T_{22} provisional commit (at P)

Transaction T decides to commit



T_{11} abort (at server M)

T_1 provisional commit (at X)

T

T_{12} provisional commit (at N)

T_{21} provisional commit (at N)

T_2 aborted (at Y)

T_{22} provisional commit (at P)

orphans

Hierarchic two-phase commit

- Multi-level nested protocol
  - – coordinator of top-level transaction is coordinator
  - – coordinator sends *canCommit?* to coordinator of subtransactions one level down the tree
  - – propagate to next level down the tree, etc
  - – aborted subtransactions ignored
  - – participants collect replies from children before replying
    - if any provisionally committed subtransaction found, prepares the object and votes *Yes*
    - if none found, assume must have crashed and vote *No*
- Second phase (completion using *doCommit*)
  - – same as before

Concurrency control

- Needed at each server
  - to ensure consistency
- In distributed systems
  - consistency needed across multiple servers
- Methods
  - Locking
    - processes run at different servers can lock objects
  - Timestamping
    - global unique timestamps
  - Optimistic concurrency control
    - validate transaction at multiple servers before committing

Locking

- Locks
  - control availability of objects
  - lock manager held at the same server as objects
  - to acquire lock: contact server
  - to release: must delay until transactions commit/abort
- Issues
  - locks acquired independently
  - cyclic dependencies may arise

    T: locks A for writing; U: locks B for writing;

    T: wants to read B - must wait; U: wants to read A - must wait;
  - distributed deadlock detection and resolution needed

Timestamp ordering

- If a single server...
  - coordinator issues unique timestamp to each transaction
  - versions of objects committed in timestamp order
  - ensures serializability
- In distributed transactions
  - coordinator issues globally unique timestamps to the client opening transaction:

    <local timestamp, server ID>
  - synchronised clocks sometimes used for efficiency
  - objects committed in global timestamp order
  - conflicts resolved, or else abort

Optimistic concurrency control

- If a single server...
  - alternative to locking (avoids overhead and deadlocks)
  - transactions allowed to proceed but
  - validated before allowed to commit: if conflict arises may be aborted
    - transactions given numbers at the start of validation
    - serialised according to this order
- In distributed transactions
  - must be validated by multiple independent servers (in the first phase of two-phase commit protocol)
  - global validation needed (serialise across servers)
  - parallel also possible

Other issues

- Distributed deadlocks!
  - often unavoidable, since cannot predict dependencies and server crashes possible
  - use deadlock detection, priorities, etc
- Recovery
  - must ensure all of committed transactions and none of the aborted transactions recorded in permanent storage
  - use logging, recovery files, shadowing, etc
- See textbook for more info

Summary

- Transactions
  - crucial to the running of large distributed systems
  - atomic, durable, serializable
  - order of updates important
  - require two-phase commit protocol
- Distributed transactions
  - run on multiple servers
  - can be flat or nested
  - hierarchical two-phase commit
  - concurrency control adapted to distributed environment