Distributed Transaction

Contents

Flat & Nested Distributed Transaction

- What is Distributed Transaction
- Flat & Nested Transaction
- Coordinator & Participants of DistributedTransaction
- Examples

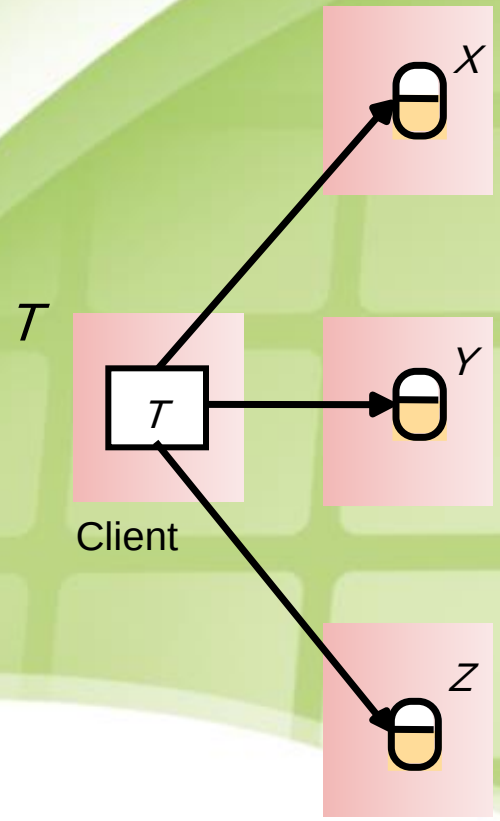
Distributed Transaction

- ✓ A client transaction becomes distributed if it invokes operations in several different Servers
- ✓ There are two different ways that distributed transactions can be structured:
  - flat transactions
  - nested transactions

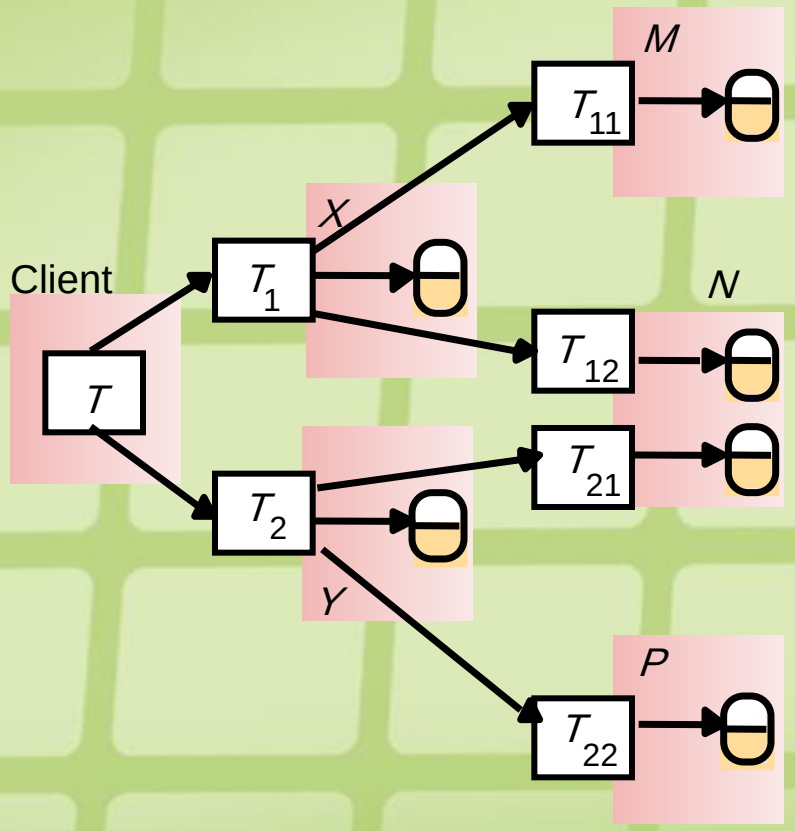
Flat & Nested Distributed Transaction

- ✓ ***In a flat transaction***
  - a client makes requests to more than one server
  - A flat transaction completes each of its requests before going on to the next one
- ✓ ***In a nested transaction***
  - the top-level transaction can open subtransactions, and each subtransaction can open further subtransactions
  - subtransactions at the same level can run concurrently

Flat & Nested Distributed Transaction



Flat transaction



Nested transactions

Flat & Nested Distributed Transaction

- ✓ The coordinator that is contacted, carries out the *openTransaction* and returns the resulting transaction identifier *(TID)* to the client.
- ✓ (TID) for distributed transactions must be unique within *a distributed system*.
- ✓ A simple way is for a TID to contain two parts
  - the identifier (for example, an IP address) of the server that created it
  - a number unique to the server

Nested banking transaction

T = openTransaction

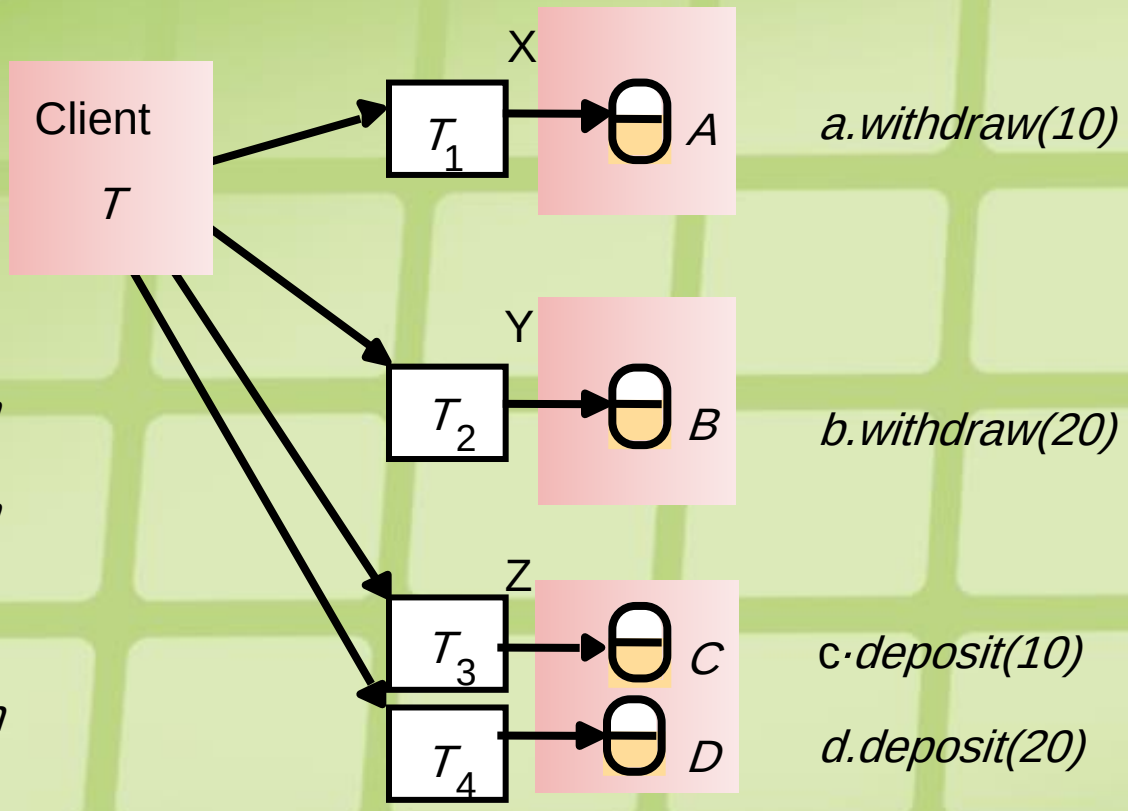
openSubTransaction
a.withdraw(10);

openSubTransaction
b.withdraw(20);

openSubTransaction
c.deposit(10);

openSubTransaction
d.deposit(20);

closeTransaction



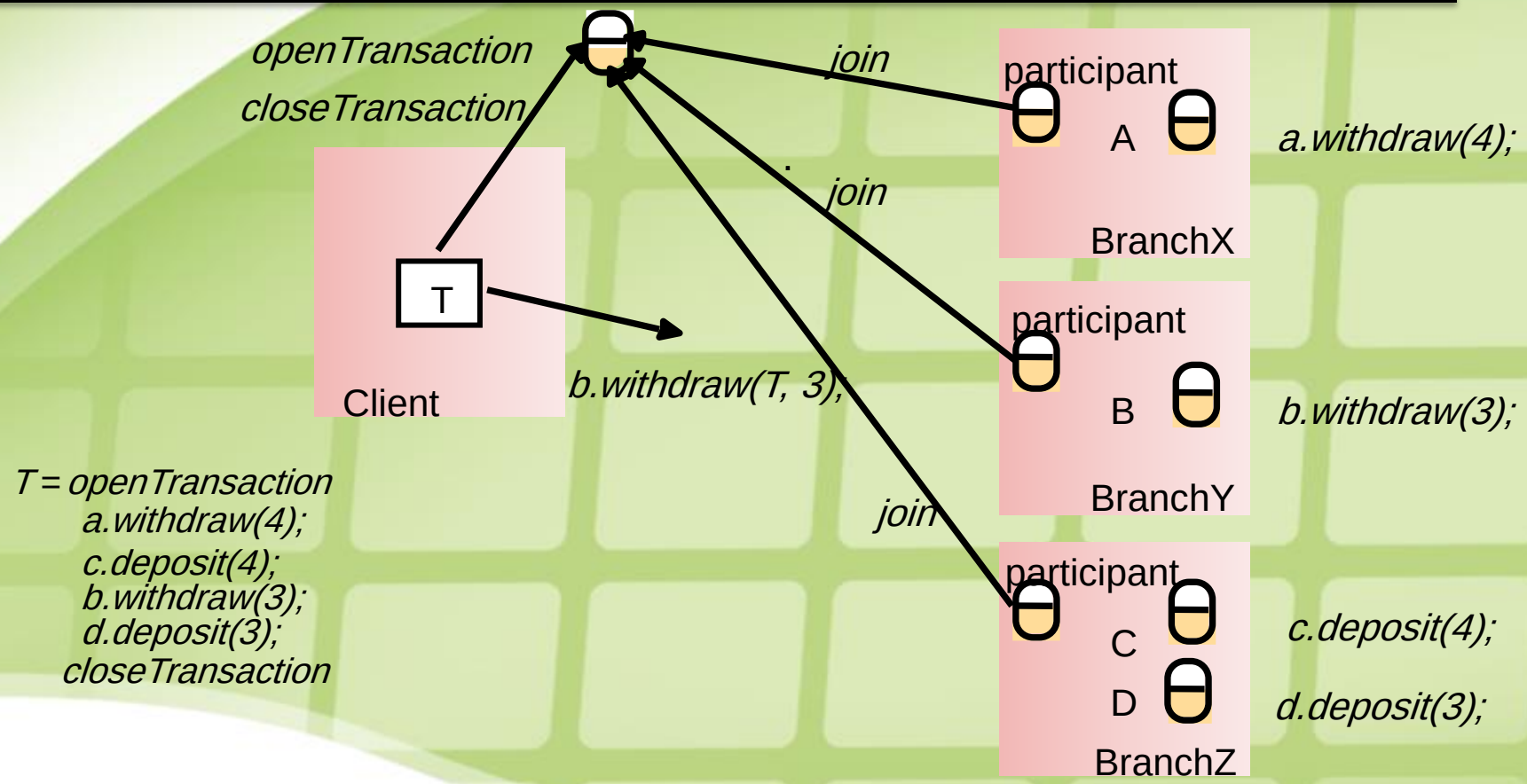
Coordinator & Participants

- ***The coordinator of Distributed Transaction***
  - The coordinator that opened the transaction becomes the coordinator for the distributed transaction
  - It starts commit protocol and It's responsible for committing or aborting of That Transaction
  - Record a list of references to the participants

Coordinator & Participants

- ✓ ***The participant***
  - servers that manages an object accessed by a transaction is a participant
  - It's responsible for keeping track of all of the recoverable objects at that server that are involved, in the transaction
  - cooperate with the coordinator in carrying out the commit protocol
  - Record a reference to the coordinator

Coordinator & Participants



join(Trans, reference to participant)

Informs a coordinator that a new participant has joined the transaction *Trans.*

Atomic Commit Protocol

- Why do we need Atomic Commit Protocol
- One Phase Commit protocol & Problems
- two-phase commit(2pc) Protocol
- Timeout actions & performance in the two-phase commit protocol
- Two-phase commit protocol for nested transactions
- Hierarchic & Flat two-phase commit protocol

Atomic Commit Protocol

- ✓ 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
- ✓ A simple way is *one-phase atomic commit protocol*

One-phase 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 unilaterally. Transaction is committed by consensus.
- ✓ ***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
- ✓ It works correctly when error happens

Operations for two-phase commit protocol

- ***methods in the interface of the participant***
  - *canCommit?(trans)-> Yes / No*
  - *doCommit(trans)*
  - *doAbort(trans)*

- ***methods in the interface of the coordinator***
  - *haveCommitted(trans, participant)*
  - *getDecision(trans) -> Yes / No*

The two-phase commit(2pc) protocol

- ✓ If the client requests *abortTransaction*, the coordinator informs all participants immediately
- ✓ When the client asks the coordinator to commit the transaction that the two-phase commit protocol comes into use.

The two-phase commit(2pc) 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(*prepared*). If the vote is *No* the participant aborts immediately.

Recall that server may crash

The two-phase commit(2pc) protocol

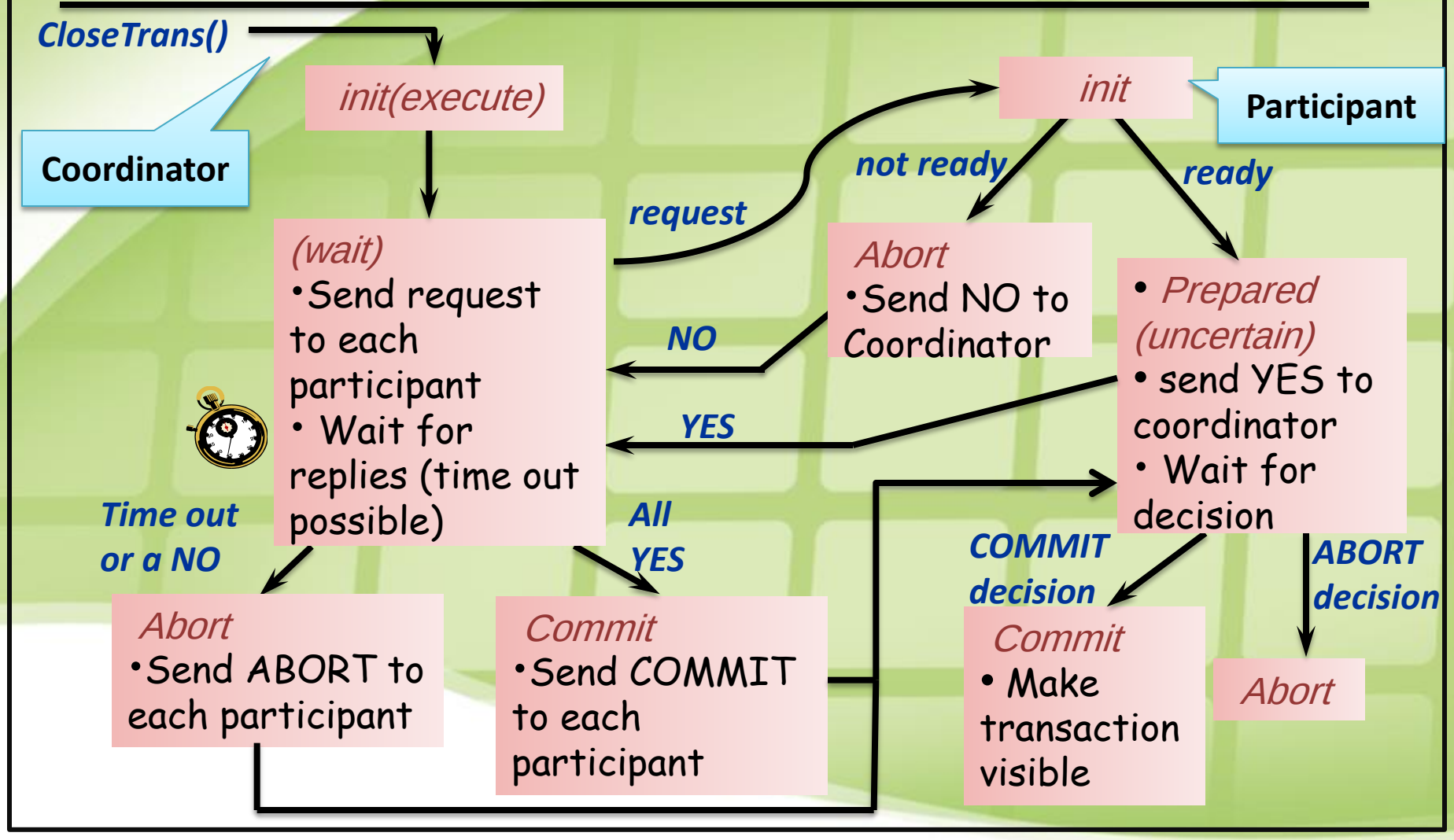
- ***Phase 2 (completion according to outcome of vote):***

  3. The coordinator collects the votes
     - (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*.

The two-phase commit(2pc) protocol

- ***Phase 2 (completion according to outcome of vote):***
  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.

The two-phase commit protocol



Failure model for the commit protocols

- ✓ It is assumed that
  - an underlying request-reply protocol removes corrupt and duplicated messages
- ✓ Servers may crash and messages may be lost

The two-phase commit(2pc) protocol

- To deal with the possibility of crashing
  - each server saves information relating to the two-phase commit protocol in permanent storage
  - Crashed process of coordinator and participant will be replaced by new processes and the information can be retrieved by a new process

Timeout actions in 2pc protocol

✓ ***Time out for the participant***

1. Timeout of waiting for canCommit(by timeout period on a lock): abort
2. Timeout of waiting for doCommit :
   - Participant is in uncertain status
   - send getDecision request to the coordinator and if coordinator fails,alternative strategies are available for the participants to obtain a decision cooperatively,but sometimes it doesn't work!(all in uncertain status)

Timeout actions in 2pc protocol

- ✓ ***Time out for the coordinator***
  1. Timeout of waiting for vote result: abort
     - 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 previous page(2)
  2. Timeout of waiting for haveCommited: do nothing,The protocol can work correctly without the confirmation

Performance of the 2pc protocol

- ✓ the cost in messages with *N participants* is *3N*, and the cost in time is three rounds of messages
- ✓ *Protocol is guaranteed to complete eventually, although it is not possible to specify a time limit within which it will be completed*
- ✓ It can cause considerable delays to participants in the *uncertain* state when coordinator fails

2pc protocol for nested transactions

- ✓ ID of subtransaction must be an extension of its parent's TID
- ✓ Subtransaction status can be:
  - Commit provisionally : updates are not saved in the permanent storage
  - Abort: it will abort all of its child.

2pc protocol for nested transactions

- ✓ *Each subtransaction*
  - If commit provisionally report the status of it and its descendants to its parent
  - If abort just report abort to its parent without any information about its descendants

- ✓ ***Operations in coordinator for nested transactions***
  - *Open subtransaction(trans) →subTrans*
  - *getStatus(trans) →commited, aborted, provisional*

2pc protocol for nested transactions

2pc protocol for nested transactions

Coordinator of transaction	Child transaction	Participant	Provisional commit list	Abort list
T	T1,T2	Yes	T1,T12	T11,T2
T1	T11,T12	Yes	T1,T12	T11
T2	T21,T22	No(aborted)		T2
T11		No(aborted)		T11
T12,T21		T12 but not T21	T21,T12	
T22		No(parent aborted)	T22	

✓We have *hierarchic* or *flat* two-phase commit protocol .In both,phase two is same as for flat transaction

Hierarchic two-phase commit protocol

- ✓ Messages are transferred according to the hierarchic relationship between successful participants
  - *canCommit?(trans, subTrans) Yes / No*
- ✓ Each participant collects the replies from its descendants before replying to its parent

Flat two-phase commit protocol...

- ✓ the coordinator of the top-level transaction sends canCommit? Messages to the coordinators of all of the subtransactions in the provisional commit list
  - *canCommit?(trans, abortList) Yes / No*
- ✓ If the participant has any provisionally committed *it* checks that they do not have aborted ancestors in the *abortList, then prepares* to commit
- aborts those with aborted ancestors;
- sends a *Yes vote to the coordinator.*

comparison of the two approaches

- ✓ *hierarchic protocol* has the advantage that at each stage, the participant only need look for subtransactions of its immediate parent
- ✓ *flat protocol* has the advantage that allows the coordinator of the top-level transaction to communicate directly with all of the participants,less messages

Concurrency control in Distributed Transaction

Concurrency control in distributed transactions

locking

timestamp ordering

optimistic concurrency control

Concurrency control in Distributed Transaction

- ✓ Each server is responsible for applying concurrency control to its own objects
- ✓ But we need Serial equivalence on all involved servers means:
  - 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

Locking

- ✓ Locks are held locally, and cannot be released until all servers involved in a transaction have committed or aborted.
- ✓ Locks are released after 2PC protocol unless transaction abort in phase 1
- ✓ Since lock managers work independently, deadlocks are very likely.

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*

Timestamp ordering concurrency control

- ✓ *Globally* unique transaction timestamp
  - timestamp consists of <localtimestamp, server-id> pair
  - it's passed to the coordinator of servers involved in the transaction
  - for efficiency it is required that the timestamps issued by one coordinator be roughly synchronized with other coordinators (synchronized local physical clocks)

Timestamp ordering concurrency control

- ✓ Each server accesses shared objects according to the timestamp
- ✓ conflicts are resolved using the rules given in Section 16.6.
- ✓ If the resolution of conflict requires a transaction to be aborted, it will abort the transaction at all the participants

Optimistic concurrency control

- ***The validation***
  - takes place during the *first phase* of two phase commit protocol
  - Serial validation is not suitable

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, to solve it each server validates according to a globally unique transaction number of each transaction

Distributed deadlocks

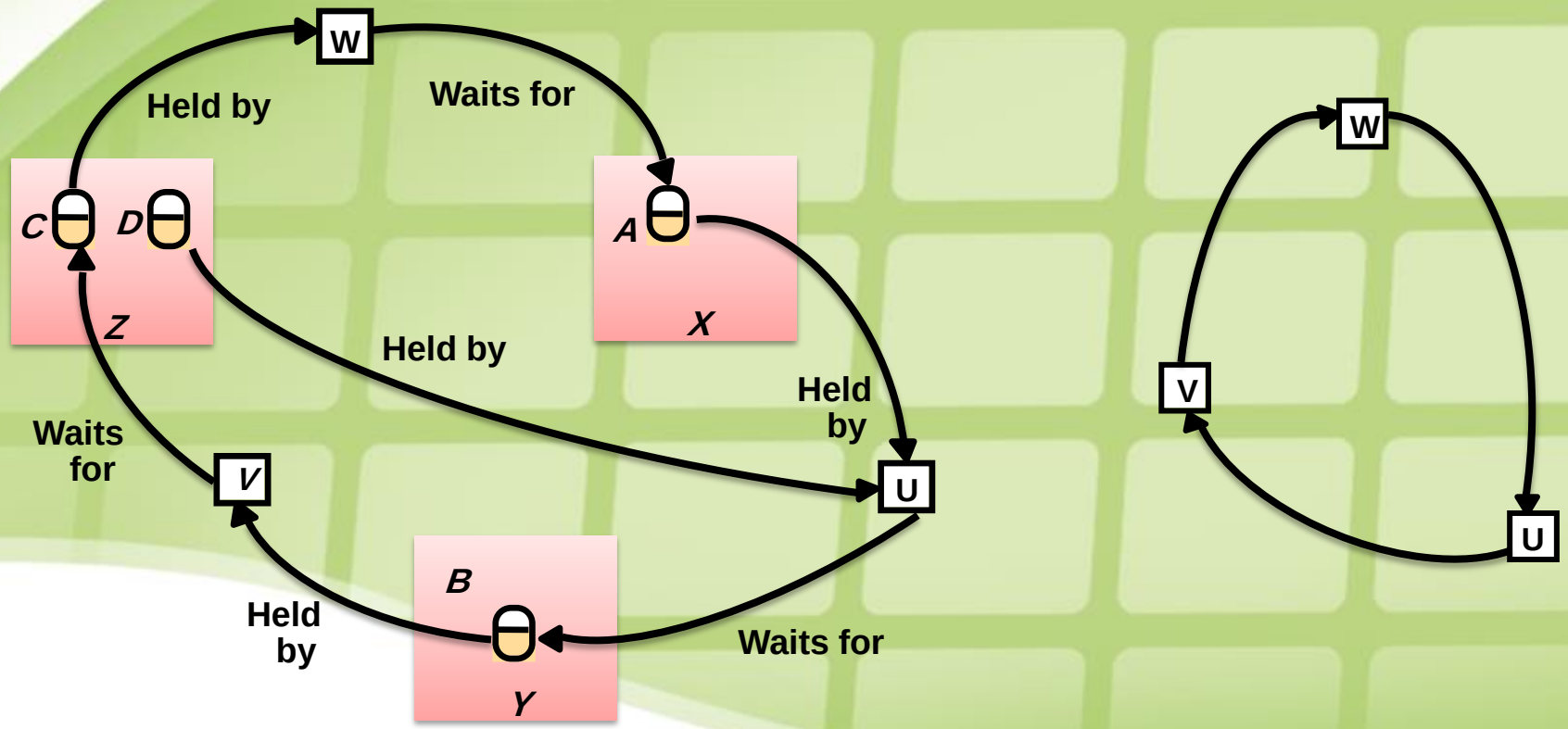
- What's distributed deadlock
- Centralized deadlock detection
- Distributed deadlock detection
- Phantom deadlocks
- Solve Distributed deadlocks
- Transaction Priority

Distributed deadlocks

U	V	W
d.deposit(10) lock D at Z		
	b.deposit(10) lock B at Y	
a.deposit(20) lock A at X		
		c.deposit(30) lock C at Z
b.withdraw(30) wait at Y		
	c.withdraw(20) wait at Z	
		a.withdraw(20) wait at X

Distributed deadlocks

- ✓ A cycle in the *global* wait-for graph



Centralized deadlock detector

- ***Simple Approach : centralized deadlock detection***
  - one server takes the role of global deadlock detector
  - each server sends the latest copy of its local wait-for graph to the global deadlock detector
  - The coordinator constructs a global graph and checks for cycles and makes a decision on how to resolve the deadlock

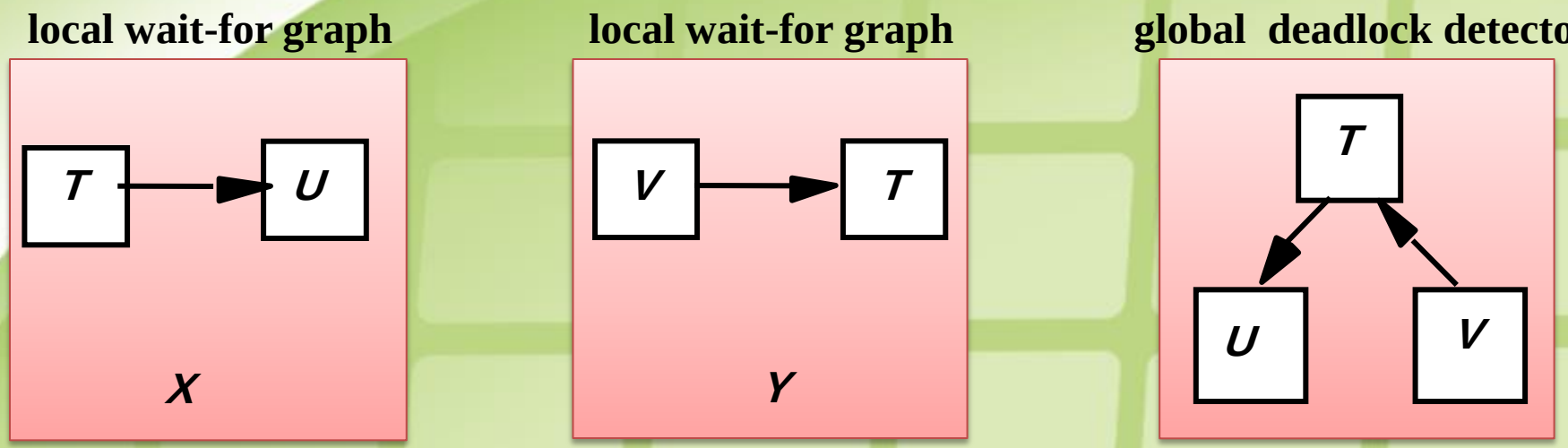
Centralized deadlock detector

- ✓ **The Problems**
  - usual problems associated with centralized solutions in distributed systems:
    poor availability
    lack of fault tolerance
    poor scalability
  - cost of collecting information is high
  - Phantom deadlock

Phantom deadlock

- ✓ A deadlock that is ‘detected’ but is not really a deadlock is called a *phantom deadlock*
- ✓ It may occur when some deadlocked transactions *abort* or *release locks*

Phantom deadlock



- at server Y: U request lock V
- at server X: U release lock for T
- at global deadlock detector: message from server Y arrives earlier than message from server X, then phantom deadlock happens

Edge chasing

- ***A distributed Aproach : Edge Chasing***
  - Each server involved in the dead-lock forwards the partial knowledge of wait-for edge which is called *probes* to other servers to construct the wait-for graph
  - But when to send a probe?

Edge chasing

Edge-chasing algorithms have three steps:

- ✓ **Initiation**
  - When a server finds that a transaction *T* starts waiting for another transaction *U*, where *U* is waiting to access an object at another server, it initiates detection by sending a *probe* containing the edge <$T \rightarrow U$> to the server of the object at which transaction *U* is blocked
- ✓ **Detection**
  - Receive probes

Edge chasing

- **Detection...**
  - Detect whether deadlock has occurred
    - Merge the local wait-for knowledge and that of the probes, find cycle
  - Decide whether to forward the probes
    - If there is a new transaction *V* is waiting for another object elsewhere, the probe is forwarded
- **Resolution**
  - When a cycle is detected, a transaction in the cycle is aborted

Edge chasing