Cloud computing and distributed systems

Distributed transactions

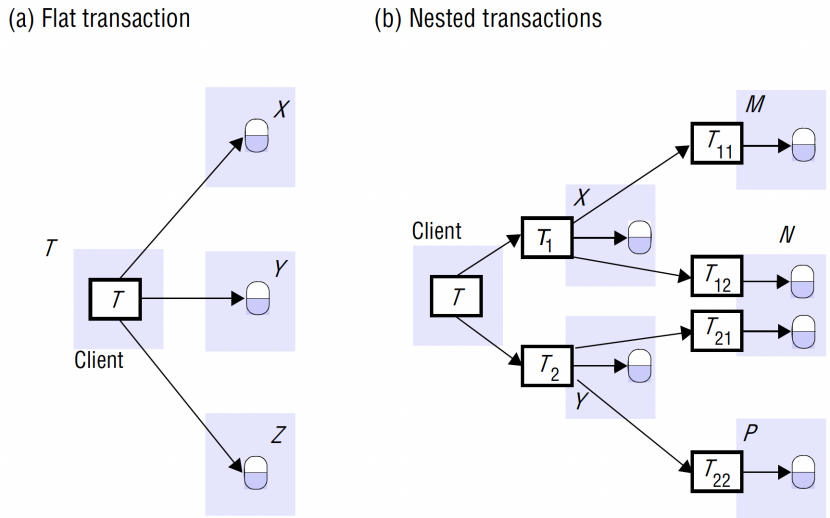
Zeynep Yücel

Ca' Foscari University of Venice
zeynep.yucel@unive.it
yucelzeynep.github.io

Introduction

- Distributed Transactions: accessing objects on multiple servers.
- Atomicity Requirement: all servers commit or abort together.
- Coordinator server.
  - A common decision protocol: "Two-Phase commit" protocol
- Concurrency Control:
  - Local concurrency control: Serialize transactions locally.
  - Global Serialization by locking, optimistic control, or timestamp ordering
    - Distributed Deadlock: Dependency cycles issue.
- Transaction Recovery: Reflect committed changes only.

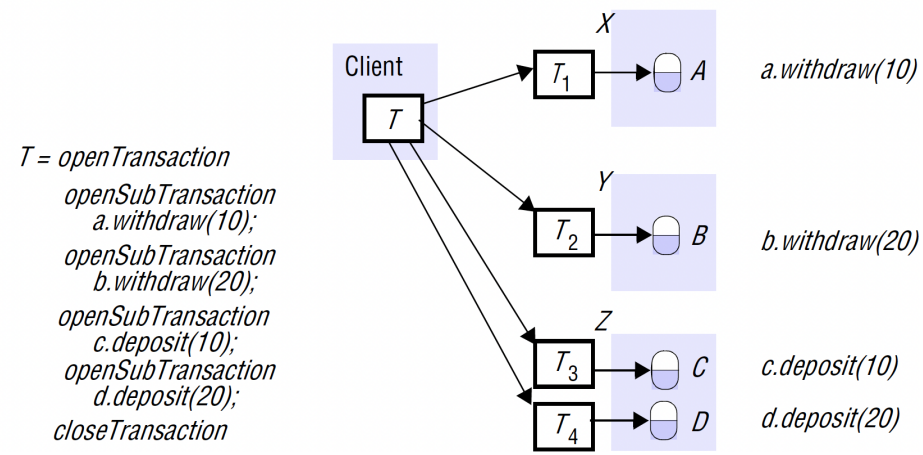
Flat and nested distributed transactions



- Distributed Transactions: Multiple server operations.
- Transaction Types:
  - Flat Transactions
  - Nested Transactions

- Flat Transactions:
  - Sequential requests to servers.
  - Completion before next request.
  - Locks for one object.
- Nested Transactions:
  - Transactions can create subtransactions.
  - Subtransactions can execute concurrently.

Flat and nested distributed transactions



- Client transfers \$10 from A to C, \$20 from B to D.
- Account A at server X; B at Y; C, D at Z.
- Four requests can run in parallel as a set of nested transactions.
- Improved performance compared to sequential execution.

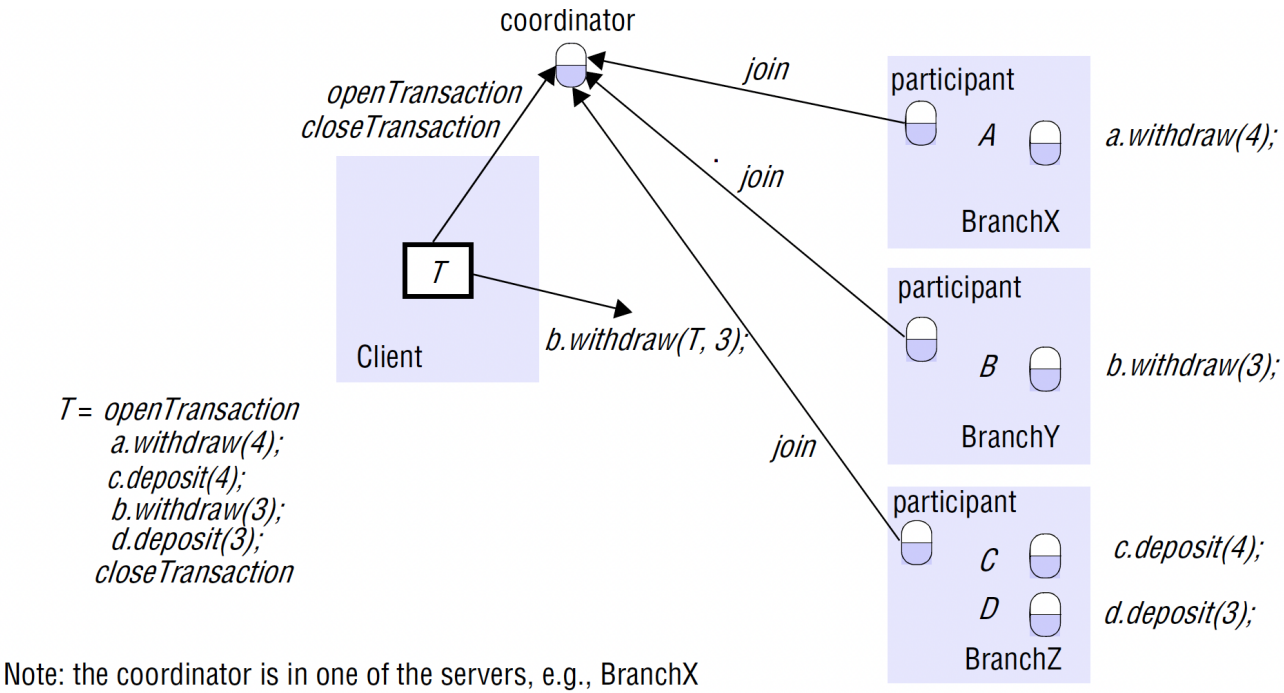
Flat and nested distributed transactions

The coordinator of a distributed transaction

- Coordinator manages transactions and commits (or aborts).
- openTransaction request starts a transactions.
- A unique transaction identifier TID is returned.
- Participants of a transaction are the servers managing the objects.
  - They track all recoverable objects.
  - They cooperate with the coordinator.
- The coordinator tracks participants.

Flat and nested distributed transactions

The coordinator of a distributed transaction



- Transaction involves moving \$4 from A to C and \$3 from B to D.
- Client invokes a method (say $b.withdraw(T, 3)$).
- The receiving object (B at Y) informs the participant (Y) that it belongs to the transaction (T) and calls *join*.
- Coordinator maintains references of participants until a *closeTransaction*.
- Participants can abort, if necessary.

Atomic commit protocols

- Atomicity ensures either all or none of the operations are carried out.
- One-phase commit protocol.
  - No unilateral aborts allowed.
- Two-phase commit features:
  - Phase 1: participants vote for commit or abort.
    - If they vote for commit, they go into a prepared state.
    - They store their objects and status.
  - Phase 2: decision execution.
    - An abort vote triggers total abort.
    - Commit requires unanimous consent.
- Challenge: vote consensus issues.

Atomic commit protocols

The two-phase commit protocol

- No communication between coordinator and participants during progress of T.
- abortTransaction request triggers immediate notification to all participants.
- If a client requests commit, two-phase commit protocol starts.
  - Phase 1: readiness confirmation.
  - Phase 2: commit or abort.

Atomic commit protocols

The two-phase commit protocol

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 when it has voted *Yes* but has still had no reply after some delay. Used to recover from server crash or delayed messages.

- Key methods include:
  - Participant methods: *canCommit*, *doCommit*, *doAbort*
  - Coordinator methods: *hasCommitted*, *getDecision*.

Atomic commit protocols

The two-phase commit protocol

The 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.

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.

- Phase-1: voting; Phase-2: completion.

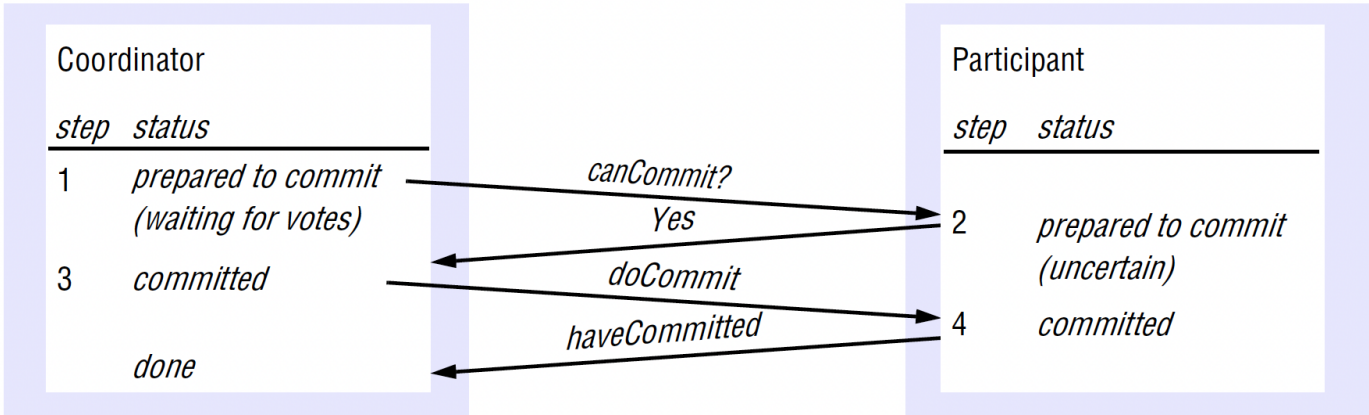
Atomic commit protocols

The two-phase commit protocol

- The protocol can fail due to:
  - Server crashes
    - Servers save protocol information for recovery.
  - Communication breakdowns
    - Timeouts are used to prevent indefinite blocking.

Atomic commit protocols

Timeout actions in the two-phase commit protocol



- 1. Participant votes "Yes" to a *canCommit*
  - It cannot proceed without the decision of the coordinator (uncertain state).
    - getDecision request to coordinator
    - Delays if coordinator fails

Atomic commit protocols

Timeout actions in the two-phase commit protocol

- 2. Participant executes all operations and waits for a *canCommit*
  - It can detect long wait times
  - Unilateral abort is possible
- 3. Coordinator waits for votes
  - It may decide to abort, if there are long waits
  - Announces *doAbort* to the participants that already voted
  - Late votes ignored and those participants go into uncertain states

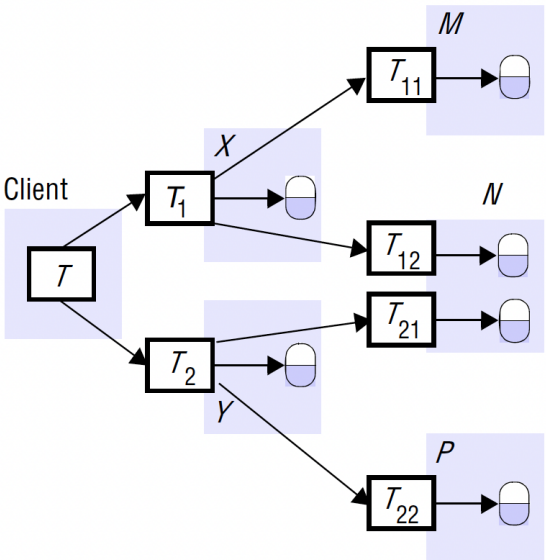
Atomic commit protocols

Performance of the two-phase commit

- Two-phase commit protocol requires a total of 3N messages: (canCommit, reply, doCommit)$\times$ N
  - Additional N haveCommitted messages
- Worst-case failure scenarios possible
- Delays may occur due to coordinator failure
- Three-phase is proposed to solve such problems

Atomic commit protocols

Two-phase commit protocol for nested transactions



- Top-level transaction: outer most one.
- Subtransactions under top-level.
- Subtransactions start after and finish before their parent.

- Subtransactions may commit provisionally or abort.
  - Provisional commit $\neq$ permanent storage.
- After all subtransactions finish, provisional committed ones enter in a two-phase protocol.

Atomic commit protocols

Two-phase commit protocol for nested transactions

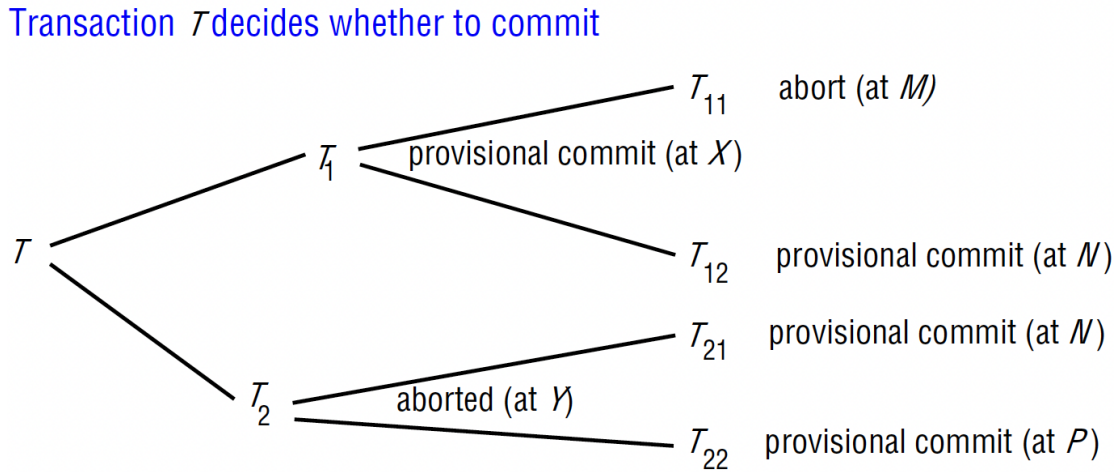
openSubTransaction(trans) → *subTrans*
Opens a new subtransaction whose parent is *trans* and returns a unique subtransaction identifier.

getStatus(trans)→ *committed, aborted, provisional*
Asks the coordinator to report on the status of the transaction *trans*. Returns values representing one of the following: *committed*, *aborted* or *provisional*.

- Coordinator provides *openSubTransaction* and *closeSubTransaction* operations to the client.
- Client starts a transaction with *openTransaction*.
  - Nested transaction opens at top-level.
  - It starts a subtransaction with *openSubTransaction*.
  - The subtransaction automatically *join*s the parent.
  - Subtransaction identifiers extend their parents' TID and are unique.
- Client calls closeTransaction or abortTransaction.

Atomic commit protocols

Two-phase commit protocol for nested transactions



- Nested transactions operate independently.
  - If parent aborts, its children must abort.
  - Parent actions vary by outcomes of children.
  - Parent can commit despite its child's abort.
- E.g. T12 status depends on its parent.
- E.g. T21, T22 provisionally committed, but T2 aborts so they need to abort.

Atomic commit protocols

Two-phase commit protocol for nested transactions

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}	

* T_{21}'s parent has aborted

- When top-level T completes, its coordinator starts two-phase commit.
- T12 and T21 share coordinator (N).
- T2 aborts, informing its parent, but not children.
- T2's children become orphans.
  - Orphans checked by coordinator via getStatus.
- Provisionally committed subtransactions of an aborted transaction are aborted.

Atomic commit protocols

Two-phase commit protocol for nested transactions

- Top-level transaction acts as coordinator.
- Participants include coordinators of subtransactions provisionally committed without aborted ancestors: T, T1, T12.
  - Participants vote on commit.
  - If they vote to commit, they save their state permanently.
- Two-phase commit protocol can be hierarchical or flat.
- In second phase, the coordinator collects votes, decides and informs participants.
- All participants commit or abort.

Atomic commit protocols

Hierarchic two-phase commit protocol

- Hierarchic case:
  - Top-level transaction sends canCommit messages to its immediate children.
  - Subtransactions forward these to their children.
- No aborted coordinators are involved.
- Participants collect replies from their descendant and send them to their parent.
- Akin to a multi-level nested protocol.

Atomic commit protocols

Hierarchic two-phase commit protocol

canCommit? for hierarchic two-phase commit protocol

canCommit?(trans, subTrans) → *Yes / No*

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

- *canCommit?* requires arguments: TIDs of top-level and parent.
- Participants check their transaction list.
  - E.g. N handles T12 ans T21.
  - N will receive a canCommit? only concerning T1, since T2 aborted.
  - So it will focus on T12.

Atomic commit protocols

Flat two-phase commit protocol

- In the flat case, top-level transaction sends a *canCommit*? to all its transactions in its provisional commit list.
  - Use only top-level TID.
  - Insufficient information (e.g. for N).
    - N is a Coordinator of T12, T21.
    - Checks T12, T21.
    - Both provisionally committed.
    - T2 aborted, so T21 should abort, but this is not clear from only top-level TID.

Atomic commit protocols

Flat two-phase commit protocol

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*.

- To solve this problem, *canCommit*? includes also an abort list.
- T's abort list: T11, T2.
- Check ancestors of provisionally committed transactions in abort list.
- Only not-orphaned descendants of the top-level transaction can be committed.

Concurrency control in distributed transactions

Locking

- Each server manages its own objects
- Serial equivalence of the transaction is ensured collectively by all servers.
- I.e. transaction order must be consistent in all servers.
  - Same access order to objects across servers

Concurrency control in distributed transactions

Introduction - Locking

- Locks on objects are managed locally at each server.
- Decision to make a transaction wait or grant a lock to it is made locally.
- Decisions are based on requests (i.e. locks cannot released arbitrarily).

Concurrency control in distributed transactions

Introduction - Locking

T			U		
write(A)	at X	locks A			
			write(B)	at Y	locks B
read(B)	at Y	waits for U			
			read(A)	at X	waits for T

- But such local management of locks can cause different transaction orderings at different servers.
- Also, cyclic dependencies are possible.
- These are called distributed deadlocks.
- A transaction can be selected and aborted to resolve the situation.
- Coordinator is notified to abort the transaction at all participants.

Concurrency control in distributed transactions

Optimistic concurrency control

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)*	

- Consider an example scenario:
  - Transactions T, U, objects A, B, servers X, Y.
  - Access order first T then U at X, first U then T at Y.
  - X waits for U.
  - Y waits for T.
  - Commitment deadlock.

Distributed deadlocks

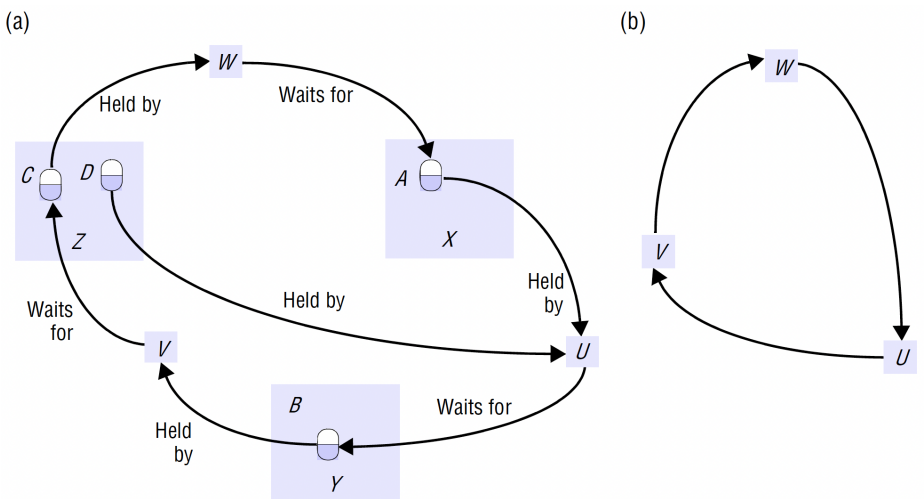
- Servers must prevent, detect, resolve deadlocks. But how?
  - Timeouts are not ideal for resolving deadlocks.
  - Thus, detection schemes often focus on cycles.
- Local wait-for graphs are maintained at each server.
- A global wait-for graph can be built from these local graphs.
- A deadlock exists, if a cycle is present in the global wait-for graph.
- Even if there is no cycle in any local graph, there may be a cycle in the global graph.
- Such a deadlock is called a *distributed* deadlock.

Distributed deadlocks

- Communication between servers is required for detecting distributed deadlocks.
- One server acts as a detector: Global deadlock detector server.
  - Drawbacks of Centralization: Poor availability, lack of fault tolerance, high communication costs etc.

Distributed deadlocks

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



- Transactions U, V, W; objects A, B, C, D; servers X, Y, Z
- Detect deadlock cycle in global graph
- Local wait-for graphs
  - Server Y: U $\rightarrow$ V
  - Server Z: V $\rightarrow$ W
  - Server X: W $\rightarrow$ U

Distributed deadlocks

Phantom deadlocks

- If a single server acts as detector
  - besides the usual issues of centralized systems (bottleneck, fault tolerance etc)
  - there is also a phantom deadlock issue.
- Phantom deadlock: False (not real) deadlock
  - Local wait-for relationships are shared
  - If a lock is released during information collection, global graph may misleadingly have a cycle.

Distributed deadlocks

Edge chasing

- Edge chasing (path pushing) is another method for distributed deadlock detection.
- No global graph
- Servers send messages (called probes) to check cycles
- Probe message contains local wait-for relationship
- Probe sending criteria:
  - The purpose is to detect potential cycles
  - So no probe sent, if the transaction is not waiting

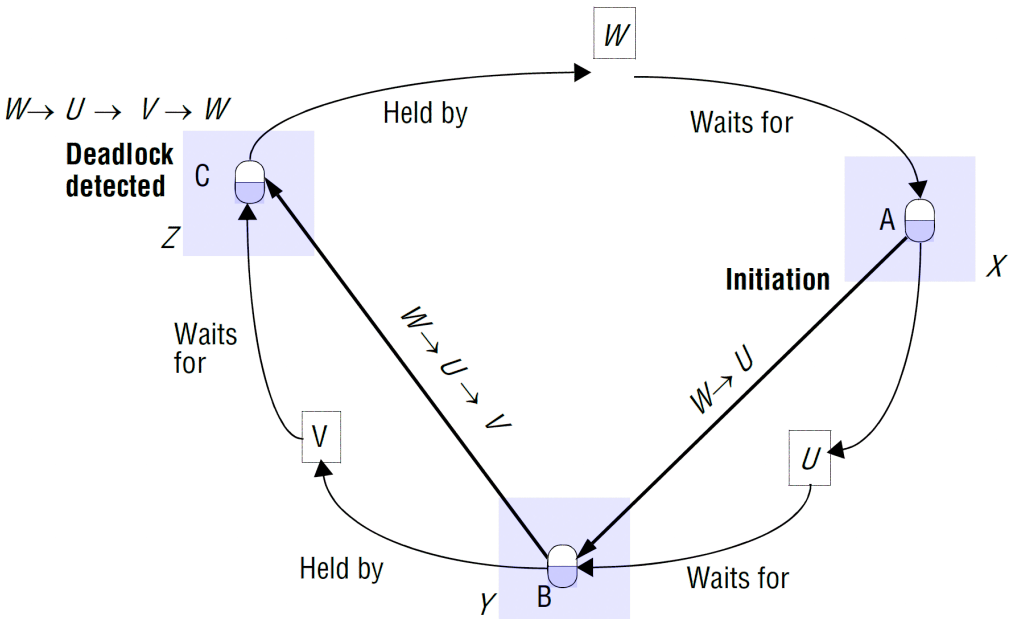
Distributed deadlocks

Edge chasing in 3 steps

- Initiation:
  - Server detects the waiting transaction (e.g. T waits for U)
  - Sends probe message ($\langle T \rightarrow U \rangle$) where U is blocked.
  - If U shares a lock, the probe is forwarded to all lock holders
- Detection:
  - Servers receiving the probe check wait-for relations
  - If U is waiting for another transaction V, it appends this to the probe ($\langle T \rightarrow U \rightarrow UV \rangle$)
  - It checks for a cycle on the appended probe
  - If no cycle, probe is forwarded to all lock holders of the servers which V is waiting for.
- Resolution:
  - Abort one transaction, if a cycle found.

Distributed deadlocks

Edge chasing in 3 steps - Example



- Server X sends $\langle W \rightarrow U \rangle$ to the server of object B (i.e. server Y).
- Y notices B is held b V, adds V to probe: $\langle W \rightarrow U \rightarrow V \rangle$. Y notes that V is waiting for C at Server Z, so forwards the new probe to Z.
- Server Z notes C is held by W, adds W to the probe, resulting in $\langle W \rightarrow U \rightarrow V \rightarrow W \rangle$.
- One transaction in the deadlock needs to be aborted.

Distributed deadlocks

Edge chasing in 3 steps - Example

- Algorithm detects deadlocks effectively.
- Detecting seem to involve many messages.
- But actually most deadlocks involve only few transactions.

Discussion topic

In a decentralized variant of the two-phase commit protocol the participants communicate directly with one another instead of indirectly via the coordinator.

In Phase 1, the coordinator sends its vote to all the participants. In Phase 2, if the coordinator's vote is No, the participants just abort the transaction; if it is Yes, each participant sends its vote to the coordinator and the other participants, each of which decides on the outcome according to the vote and carries it out.

Calculate the number of messages and the number of rounds it takes. What are its advantages or disadvantages in comparison with the centralized variant?