Distributed Systems Course

Distributed transactions

Commitment of distributed transactions - introduction

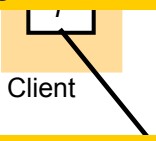
- a distributed transaction refers to a flat or nested transaction that accesses objects managed by multiple servers
- When a distributed transaction comes to an end
  - the either all of the servers commit the transaction
  - or all of them abort the transaction.
- one of the servers is coordinator, it must ensure the same outcome at all of the servers.
- the ‘two-phase commit protocol’ is the most commonly used protocol for achieving this

Flat vs nested transactions

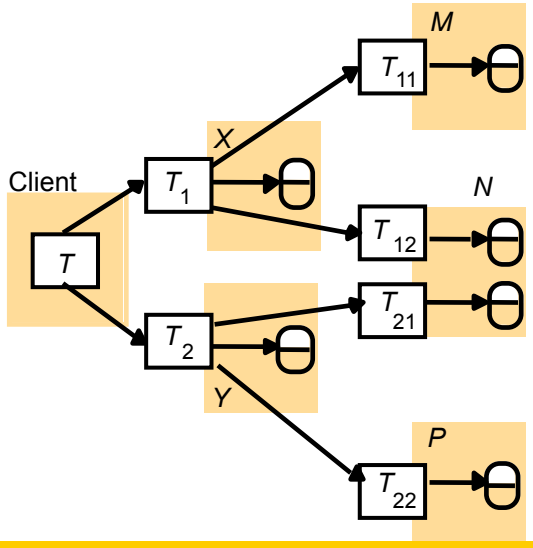
(a) Flat transaction

(b) Nested transactions

In a nested transaction, the top-level transaction can open subtransactions, and each subtransaction can open further subtransactions down to any depth of nesting

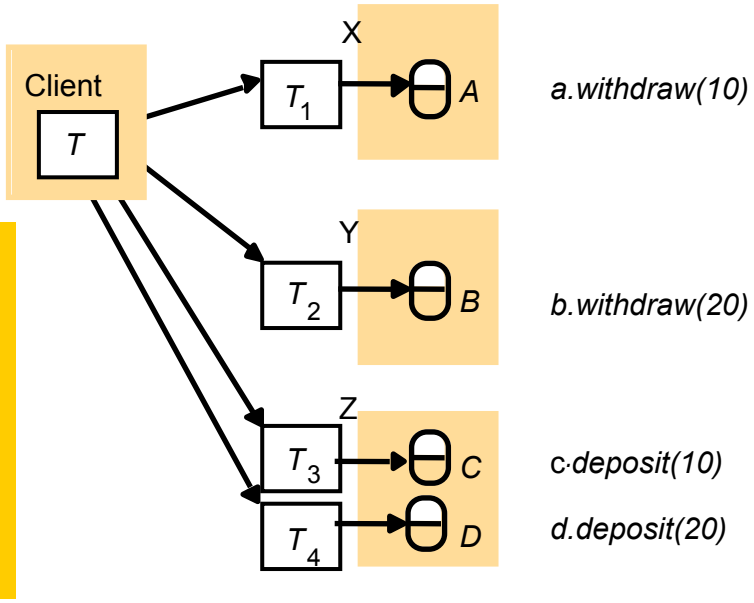


In the nested case, subtransactions at the same level can run concurrently, so *T*1 and *T*2 are concurrent, and as they invoke objects in different servers, they can run in parallel.



A flat client transaction completes each of its requests before going on to the next one. Therefore, each transaction accesses servers' objects sequentially

Nested banking transaction



requests can be run in parallel - with several servers, the nested transaction is more efficient

- client transfers $10 from A to C and then transfers $20 from B to D

Why might a participant abort a transaction?

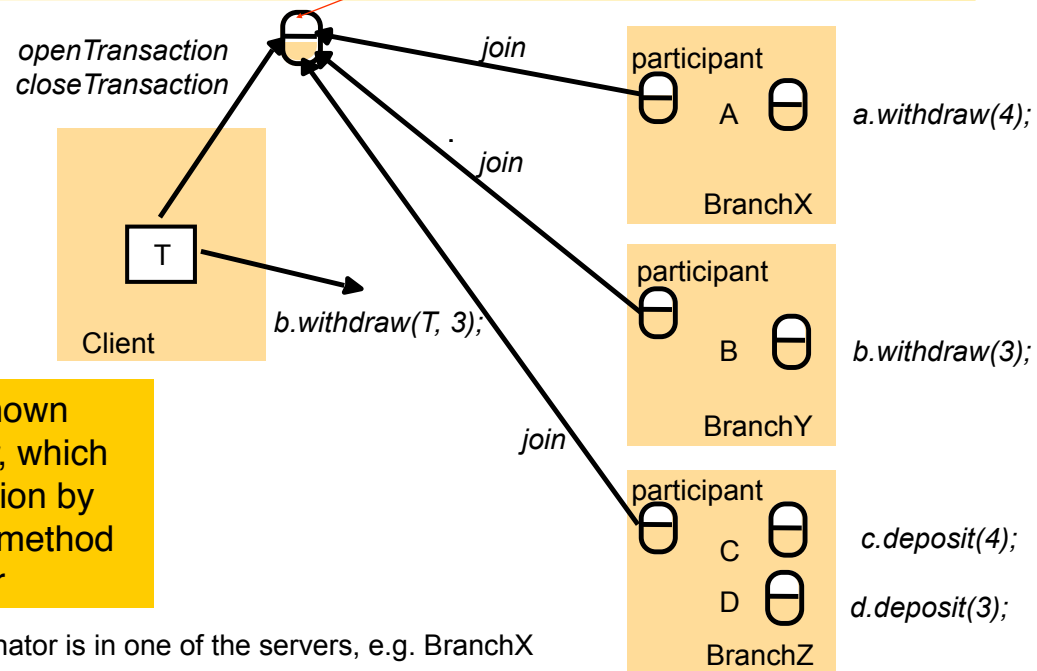
The coordinator of a flat distributed transaction

- Servers execute requests in a distributed transaction
  - when it commits they must communicate with one another to coordinate their actions
  - a client starts a transaction by sending an open*Transaction* request to a *coordinator* in any server (next slide)
    - it returns a TID unique in the distributed system(e.g. server ID + local transaction number)
    - at the end, it will be responsible for committing or aborting it
  - each server managing an object accessed by the transaction is a *participant* - it joins the transaction (next slide)
    - a participant keeps track of objects involved in the transaction
    - at the end it cooperates with the coordinator in carrying out the commit protocol
  - note that a participant can call *abortTransaction* in coordinator

A flat distributed banking transactio

openTransaction goes to the coordinator

a client's (flat) banking transaction involves accounts *A*, *B*, *C* and *D* at servers *BranchX*, *BranchY* and *BranchZ*



Each server is shown with a *participant*, which joins the transaction by invoking the *join* method in the coordinator

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

- Note that the TID (*T*) is passed with each request e.g. withdraw(T,3)

The join operation

- The interface for *Coordinator* is shown in the previous slide:
  - it has *openTransaction*, *closeTransaction* and *abortTransaction*
  - *openTransaction* returns a *TID* which is passed with each operation so that servers know which transaction is accessing its objects
- The *Coordinator* interface provides an additional method, *join*, which is used whenever a new participant joins the transaction:
  - *join*(*Trans, reference to participant*)
  - informs a coordinator that a new participant has joined the transaction *Trans*.
  - the coordinator records the new participant in its participant list.
  - the fact that the coordinator knows all the participants and each participant knows the coordinator will enable them to collect the information that will be needed at commit time.

Atomic commit protocols

- transaction atomicity requires that at the end,
  - either all of its operations are carried out or none of them.
- in a distributed transaction, the client has requested the operations at more than one server
- one-phase atomic commit protocol
  - the coordinator tells the participants whether to commit or abort
  - this does not allow one of the servers to decide to abort – it may have discovered a deadlock or it may have crashed and been restarted
- two-phase atomic commit protocol
  - is designed to allow any participant to choose to abort a transaction
  - *phase 1* - each participant votes. If it votes to commit, it is *prepared.* It cannot change its mind. In case it crashes, it must save updates in permanent store
  - phase 2 - the participants carry out the joint decision

The decision could be *commit* or *abort* - participants record it in permanent store

Failure model for the commit protocols

- Recall the failure model for transactions in Chapter 16
  - this applies to the two-phase commit protocol
- Commit protocols are designed to work in
  - asynchronous system (e.g. messages may take a very long time)
  - servers may crash
  - messages may be lost.
  - assume corrupt and duplicated messages are removed.
  - no byzantine faults – servers either crash or they obey their requests
- 2PC is an example of a protocol for reaching a consensus.
  - Chapter 15 says consensus cannot be reached in an asynchronous system if processes sometimes fail.
  - however, 2PC does reach consensus under those conditions.
  - because crash failures of processes are masked by replacing a crashed process with a new process whose state is set from information saved in permanent storage and information held by other processes.

The two-phase commit protocol

- During the progress of a transaction, the only communication between coordinator and participant is the *join* request
  - The client request to commit or abort goes to the coordinator
    - if client or participant request abort, the coordinator informs the participants immediately
    - if the client asks to commit, the 2PC comes into use
- 2PC
  - *voting phase*: coordinator asks all participants if they can commit
    - if yes, participant records updates in permanent storage and then votes
  - *completion phase*: coordinator tells all participants to commit or abort
  - the next slide shows the operations used in carrying out the protocol

Operations for two-phase commit protocol

canCommit?(trans)-> Yes / No
This is a request with a reply
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)
These are asynchronous requests to avoid delays
Call from coordinator to participant to tell participant to abort its part of a transaction.

haveCommitted(trans, participant)
Asynchronous request
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.

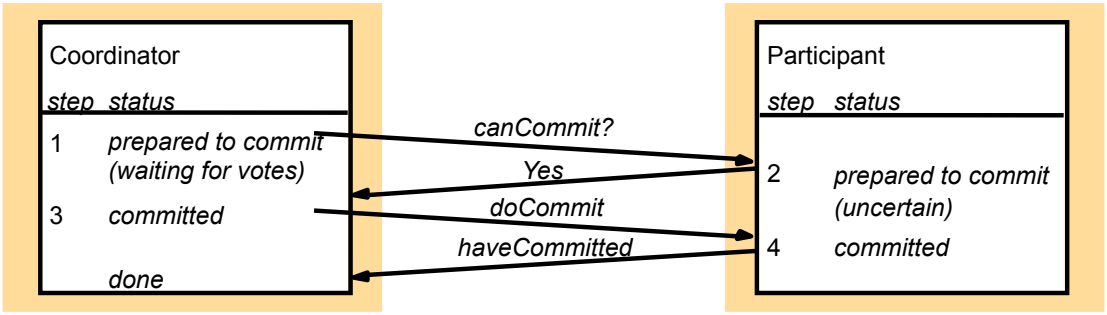
- participant interface- *canCommit?, doCommit, doAbort*
  coordinator interface- *haveCommitted, getDecision*

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.

Communication in two-phase commit protocol



- Time-out actions in the 2PC
  - to avoid blocking forever when a process crashes or a message is lost
  - *uncertain* participant (step 2) has voted yes. it can't decide on its own
    - it uses *getDecision* method to ask coordinator about outcome
  - participant has carried out client requests, but has not had a *Commit*? from the coordinator. It can abort unilaterally
  - coordinator delayed in waiting for votes (step 1). It can abort and send *doAbort* to participants.

Performance of the two-phase commit protocol

- if there are no failures, the 2PC involving *N* participants requires
  - *N canCommit?* messages and replies, followed by *N doCommit* messages.
    - the cost in messages is proportional to 3*N*, and the cost in time is three rounds of messages.
    - The *haveCommitted* messages are not counted
  - there may be arbitrarily many server and communication failures
  - 2PC is is guaranteed to complete eventually, but it is not possible to specify a time limit within which it will be completed
    - delays to participants in uncertain state
    - some 3PCs designed to alleviate such delays
      - they require more messages and more rounds for the normal case

Two-phase commit protocol for nested transactions

- Recall Fig 17.1b, top-level transaction T and subtransactions T_1, T_2, T_{11}, T_{12}, T_{21}, T_{22}
- A subtransaction starts after its parent and finishes before it
- When a subtransaction completes, it makes an independent decision either to *commit provisionally* or to abort.
  - A provisional commit is not the same as being prepared: it is a local decision and is not backed up on permanent storage.
  - If the server crashes subsequently, its replacement will not be able to carry out a provisional commit.
- A two-phase commit protocol is needed for nested transactions
  - it allows servers of provisionally committed transactions that have crashed to abort them when they recover.

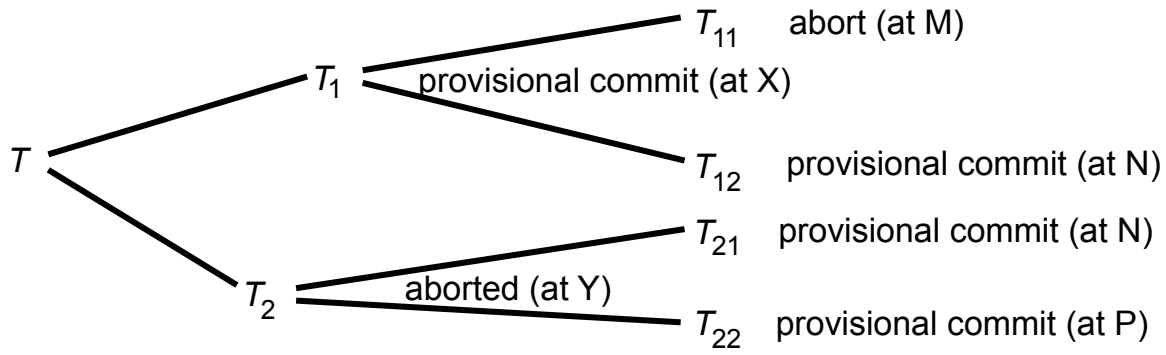
The client finishes a set of nested transactions by calling *closeTransaction* or *abortTransaction* in the top-level transaction.

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*, *provisional*.

- This is the interface of the coordinator of a subtransaction.
  - It allows it to open further subtransactions
  - It allows its subtransactions to enquire about its status
- Client starts by using *OpenTransaction* to open a top-level transaction.
  - This returns a TID for the top-level transaction
  - The TID can be used to open a subtransaction
    - The subtransaction automatically *joins* the parent and a TID is returned.

Suppose that T decides to commit although T_2 has aborted, also that T_1 decides to commit although T_{11} has aborted



- Recall that
  1. A parent can commit even if a subtransaction aborts
  2. If a parent aborts, then its subtransactions must abort
- In the figure, each subtransaction has either provisionally committed or aborted

Information held by coordinators of nested transactions

- When a top-level transaction commits it carries out a 2PC
- Each coordinator has a list of its subtransactions
- At provisional commit, a subtransaction reports its status and the status of its descendents to its parent
- If a subtransaction aborts, it tells its parent

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}	

an *orphan* uses *getStatus* to ask its parent about the outcome. It should abort if its parent has

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*.

- Top-level transaction is coordinator of 2PC.
- participant list:
  - the coordinators of all the subtransactions that have provisionally committed
  - but do not have an aborted ancestor
  - E.g. T, T1 and T12 in Figure 17.8
  - if they vote *yes*, they *prepare* to commit by saving state in permanent store
    - The state is marked as belonging to the top-level transaction
- The 2PC may be performed in a hierarchic or a flat manner

The *trans* argument is used when saving the objects in permanent storage

Compare the advantages and disadvantages of the flat and nested approaches

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*.

- Flat 2PC
  - the coordinator of the top-level transaction sends *canCommit*? messages to the coordinators of all of the subtransactions in the provisional commit list.
  - in our example, T sends to the coordinators of T_1 and T_{12}.
  - the *trans* argument is the TID of the top-level transaction
  - the *abortList* argument gives all aborted subtransactions
    - e.g. server N has T_{12} prov committed and T_{21} aborted
  - On receiving *canCommit*, participant
    - looks in list of transactions for any that match *trans* (e.g. T_{12} and T_{21} at N)
    - it *prepares* any that have provisionally committed and are not in *abortList* and votes *yes*
    - if it can't find any it votes *no*

Time-out actions in nested 2PC

- With nested transactions delays can occur in the same three places as before
  - when a *participant* is prepared to *commit*
  - when a participant has finished but has not yet received *canCommit*?
  - when a coordinator is waiting for votes
- Fourth place:
  - provisionally committed subtransactions of aborted subtransactions e.g. T22 whose parent T2 has aborted
  - use *getStatus* on parent, whose coordinator should remain active for a while
  - If parent does not reply, then abort

Summary of 2PC

- a distributed transaction involves several different servers.
  - A nested transaction structure allows
    - additional concurrency and
    - independent committing by the servers in a distributed transaction.
- atomicity requires that the servers participating in a distributed transaction either all commit it or all abort it.
- atomic commit protocols are designed to achieve this effect, even if servers crash during their execution.
- the 2PC protocol allows a server to abort unilaterally.
  - it includes timeout actions to deal with delays due to servers crashing.
  - 2PC protocol can take an unbounded amount of time to complete but is guaranteed to complete eventually.

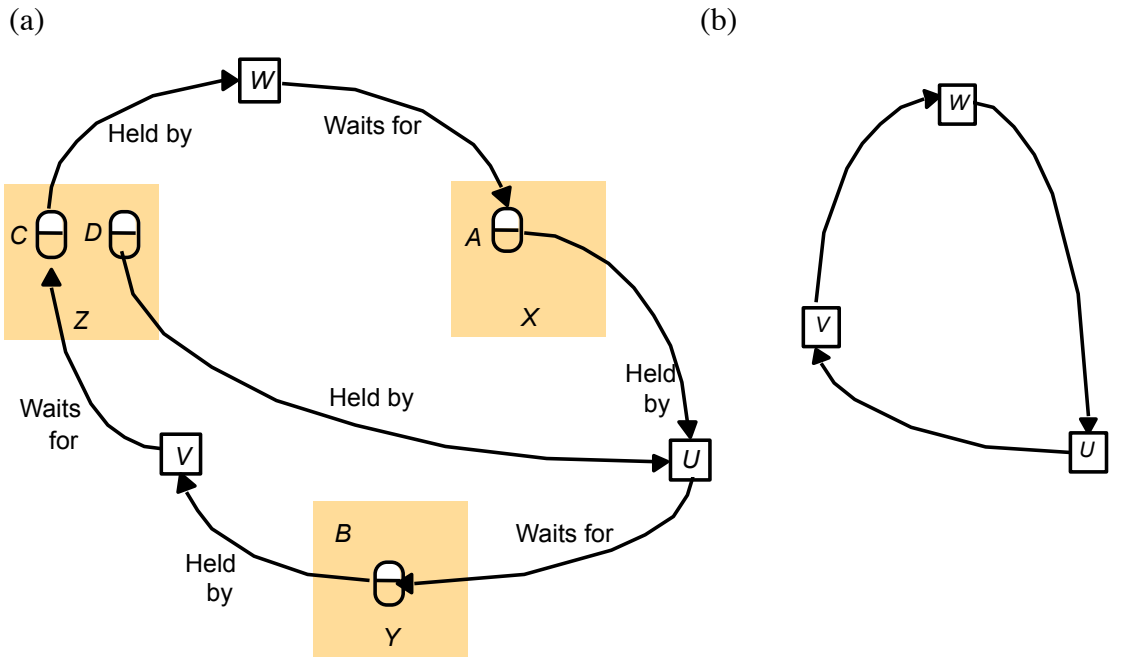
Distributed deadlocks

- Deadlocks can arise when locking is used for concurrency control
- Deadlocks can be either prevented or detected and resolved (or ignored)
- Timeouts are a clumsy approach to solve deadlocks
- Most deadlock detections schemes operate by finding cycles in the transaction wait-for graph
- in a distributed setting, a global graph can be built by merging local ones, kept by each server

Figure 17.12
Interleavings of transactions *U*, *V* and *W*

U		*V*		*W*	
d.deposit(10)	lock *D*				
		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*

Figure 17.13
Distributed deadlock



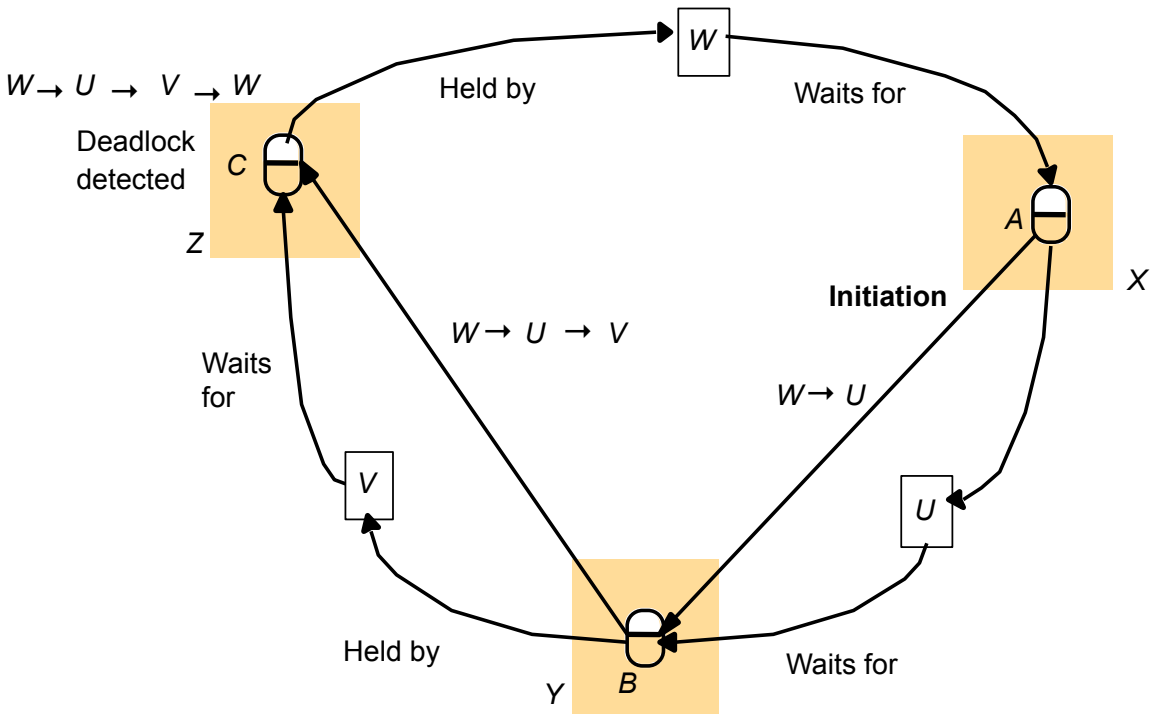
Distributed deadlock detection

- The global wait-for graph is held in part by each server => in order to find cycles, the servers must communicate
- Simple solution: centralized deadlock detection
  - each server sends its latest copy of the wait-for graph to a specific server, which constructs the global wait-for graph and runs usual loop detection algorithms
  - when a loop is found, the detector decides which transaction to abort, and tells the corresponding server to abort it
  - not scalable, SPOF, high traffic (for frequent transmissions of wait-for graphs)

Edge chasing

- Distributed approach to deadlock detection
- Servers try to find loops by forwarding messages called *probes* along the edges of the graph
- Probes are sent when a server holds an object and waits for another one

Figure 17.15
Probes transmitted to detect deadlock



Edge chasing algorithm

- **Initiation**: when a server notes a transaction T waits for another transaction U, where U is waiting to access an object at another server X, initiates detection by sending the probe T→U to the server X
- **Detection**: when a server receives a probe, decides if there is a deadlock, or if the probe has to be forwarded. Eg: when a server X receives a probe T→U, where U is a transaction holding an object local to X, it checks whether U is also waiting. If so, X adds the the transaction to the probe: T→U→V, and if V is waiting, it forwards the probe If the probe contains a loop (e.g. T→U→V→T), we have found the deadlock!
- Resolution: when a cycle is detected, a transaction in the cycle is aborted

Figure 17.16
Two probes initiated

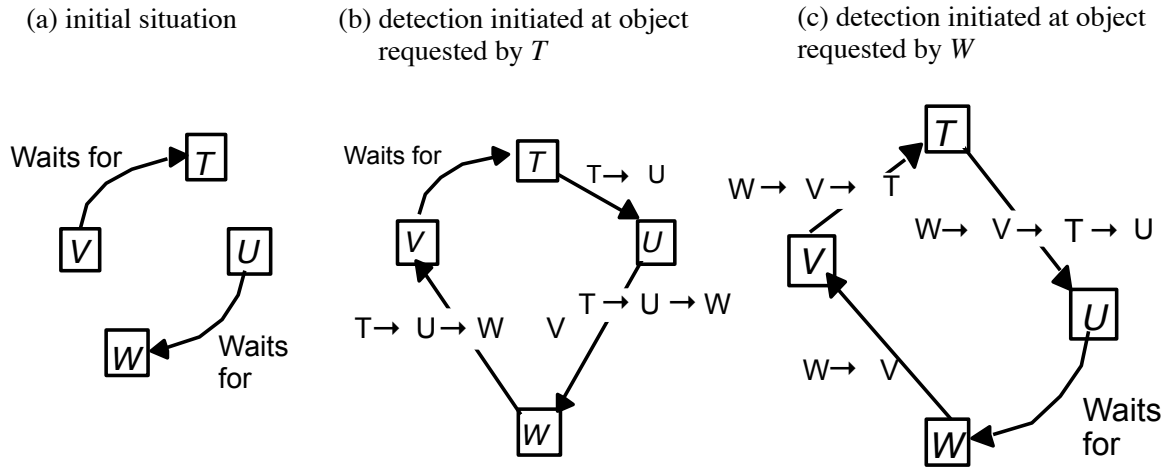
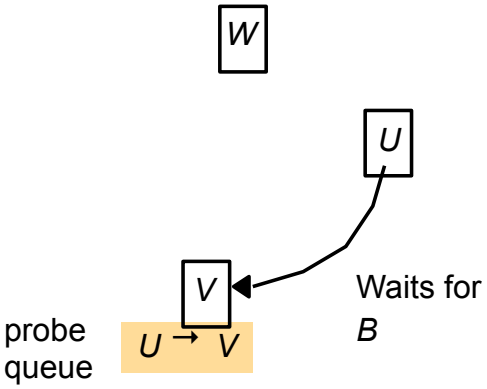


Figure 17.17
Probes travel downhill

(a) V stores probe when U starts waiting



(b) Probe is forwarded when V starts waiting