MODULE-5
Transactions and Concurrency Control: Introduction, Transactions, Nested Transactions, Locks, Optimistic concurrency control, Timestamp ordering, Comparison of methods for concurrency control.
Distributed Transactions: Introduction, Flat and Nested Distributed Transactions, Atomic commit protocols, Concurrency control in distributed transactions, Distributed deadlocks, Transaction recovery.

Transactions and Concurrency Control: Introduction

Banking transaction for a customer (e.g., at ATM or browser)

Transfer $100 from saving to checking account;

Transfer $200 from money-market to checking account;

Withdraw $400 from checking account.

Transaction (invoked at client): /* Every step is an RPC */

1. savings.withdraw(100) /* includes verification */
2. checking.deposit(100) /* depends on success of 1 */
3. mnymkt.withdraw(200) /* includes verification */
4. checking. deposit(200) /* depends on success of 3 */
5. checking.withdraw(400) /* includes verification */
6. dispense(400)
7. commit

- All the following are RPCs from a client to the server
- Transaction calls that can be made at a client, and return values from the server:

Transactions

openTransaction() -> trans;

starts a new transaction and delivers a unique transaction identifier (TID) trans. This TID will be used in the other operations in the transaction.

closeTransaction(trans) -> (commit, abort);

ends a transaction: a commit return value indicates that the transaction has committed; an abort return value indicates that it has aborted.

abortTransaction(trans);

BCS515D,SKIT

aborts the transaction.

- TID can be passed implicitly (for other operations between open and close) with CORBA
- deposit(amount)
- deposit amount in the account
- withdraw(amount)
- withdraw amount from the account
- getBalance() -> amount
- return the balance of the account
- setBalance(amount)
- set the balance of the account to amount
- Sequence of operations that forms a single step, transforming the server data from one consistent state to another.
  - All or nothing principle: a transaction either completes successfully, and the effects are recorded in the objects, or it has no effect at all. (even with multiple clients, or crashes)
- A transactions is indivisible (atomic) from the point of view of other transactions
  - No access to intermediate results/states of other transactions
  - Free from interference by operations of other transactions

But…

- Transactions could run concurrently, i.e., with multiple clients
- Transactions may be distributed, i.e., across multiple servers
- Atomicity: All or nothing

BCS515D,SKIT

- Consistency: if the server starts in a consistent state, the transaction ends the server in a consistent state.
- Isolation: Each transaction must be performed without interference from other transactions, i.e., the non-final effects of a transaction must not be visible to other transactions.
- Durability: After a transaction has completed successfully, all its effects are saved in permanent storage.
- Atomicity: store tentative object updates (for later undo/redo) – many different ways of doing this
- Durability: store entire results of transactions (all updated objects) to recover from permanent server crashes.
- The effect of an operation refers to
- The value of an object set by a write operation
- The result returned by a read operation.
- Two operations are said to be conflicting operations, if their combined effect depends on the order they are executed, e.g., read-write, write-read, write-write (all on same variables). NOT read-read, NOT on different variables.
- Two transactions are serially equivalent if and only if all pairs of conflicting operations (pair containing one operation from each transaction) are executed in the same order (transaction order) for all objects (data) they both access.
- Why? Can start from original operation sequence and swap the order of non-conflicting operations to obtain a series of operations where one transaction finishes completely before the second transaction starts
- Why is the above result important? Because: Serial equivalence is the basis for concurrency control protocols for transactions.

Operations of different transactions		*Conflict*	*Reason*
read	*read*	No	Because the effect of a pair of *read* operations does not depend on the order in which they are executed
read	*write*	Yes	Because the effect of a *read* and a *write* operation depends on the order of their execution
write	*write*	Yes	Because the effect of a pair of *write* operations depends on the order of their execution

BCS515D.SKIT

Concurrency control

- Lost update
  - 3 accounts (A, B, C)
    - » with balances 100, 200, 300
  - T1 transfers from A to B, for 10% increase
  - T2 transfers from C to B, for 10% increase
  - Both T1, T2 read balance of B (200)
  - T1 overwrites the update by T2
    - » Without seeing it
- Inconsistent retrievals
  - T1: transfers 10% of account A to account B
  - T2: computes sum of account balances
  - T2 computes sum before T1 updates B

Recoverability from aborts

- Servers must prevent a aborting Tx from affecting other concurrent Tx's.
  - Dirty reads:
    - » T2 sees result update by T1 on account A
    - » T2 performs its own update on A & then commits.
    - » T1 aborts -> T2 has seen a "transient" value
      - T2 is not recoverable
    - » If T2 delays its commit until T1's outcome is resolved:
      - Abort(T1) -> Abort(T2)
      - However, if T3 has seen results of T2:
        - Abort(T2) -> Abort(T3) !
- Premature writes:
  - Assume server implements abort by maintaining the "before" image of all update operations
    - » T1 & T2 both updates account A

BCS515D,SKIT

» T1 completes its work before T2

» If T1 commits & T2 aborts, the balance of A is correct

» If T1 aborts & T2 commits, the "before" image that is restored corresponds to the balance of A before T2

» If both T1 & T2 abort, the "before" image that is restored corresponds to the balance of A as set by T1

- Tx's should delay both their reads & updates in order to avoid interference
  - Strict execution -> enforce isolation
- Servers should maintain tentative versions of objects in volatile memory
- Tx's should delay both their reads & updates in order to avoid interference
  - Strict execution -> enforce isolation
- Servers should maintain tentative versions of objects in volatile

memory Concurrency Control: Locks

- Transactions:
  - Must be scheduled so that their effect on shared data is serially equivalent
  - Two types of approach

» Pessimistic If something can go wrong, it will

Operations are synchronized before they are carried out

» Optimistic In general, nothing will go wrong

Operations are carried out, synchronization at the end of the transaction

- Locks (pessimistic)

» can be used to ensuring serializability

» lock(x), unlock(x)

- Oldest and most widely used CC algorithm
- A process before read/write requests the scheduler to grant a lock

BCS515D,SKIT

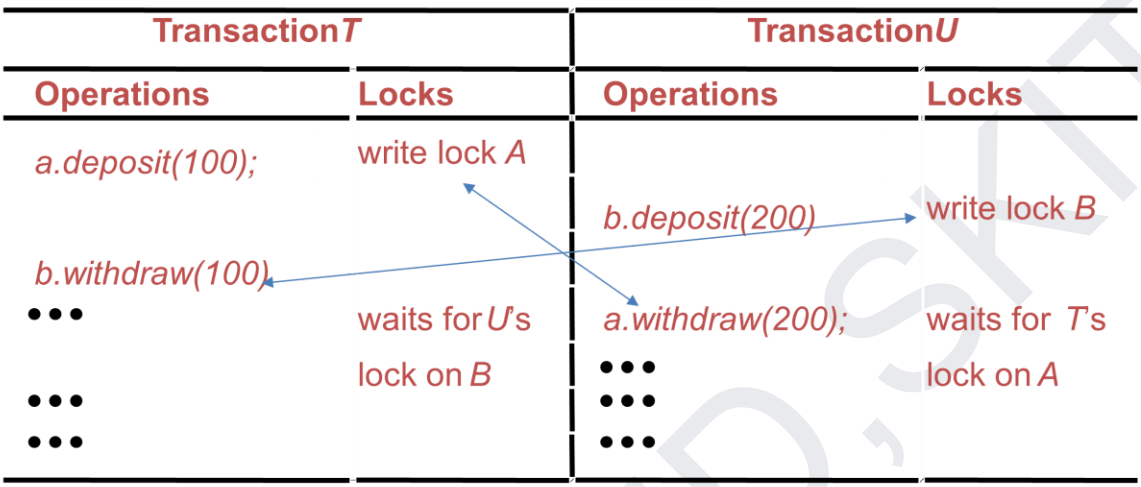
- Upon finishing read/write the lock is released
- In order to ensure serialized transaction Two Phase Locking (2PL) is used
- How Locks prevent consistency problems
  - Lost update and inconsistent retrieval:
  - Causes:
    - » are caused by the conflict between $r_i(x)$ and $w_j(x)$
    - » two transactions read a value and use it to compute new value
  - Prevention:
    - » delay the reads of later transactions until the earlier ones have completed
    - » Disadvantage of Locking
  - Deadlocks
- Strict 2PL avoids Cascading Aborts
  - A situation where a committed transaction has to be undone because it saw a file it shouldn't have seen.
- Problems of Locking
  - Deadlocks
  - Livelocks
    - » A transaction can't proceed for an indefinite amount of time while other transactions continue normally. It happens due to unfair locking.
  - Lock overhead
    - » If the system doesn't allow shared access--wastage of resources
  - Avoidance of Cascading Aborts may be costly
    - » Strict 2PL in fact, reduces the effect of concurrency
- Transaction managers (on server side) set locks on objects they need. A concurrent trans. cannot access locked objects.
- Two phase locking:
  - In the first (growing) phase of the transaction, new locks are only acquired, and in the second (shrinking) phase, locks are only released.

BCS515D.SKIT

– A transaction is not allowed acquire *any* new locks, once it has released any one lock.

Deadlock with write locks

Deadlock with write locks



T locks A and waits for U to release the lock on B, U on the other hand locks B and waits for T to release the lock on A
➔ Circular hold and wait ➔ Deadlock

❖ Necessary conditions for deadlocks

- ❑ Non-shareable resources (exclusive lock modes)
- ❑ No preemption on locks
- ❑ Hold & Wait or Circular Wait

BCS515D,SKIT

Naïve Deadlock Resolution Using Timeout

Transaction T		Transaction U	
Operations	**Locks**	**Operations**	**Locks**
a.deposit(100);	write lock *A*		
		b.deposit(200)	write lock *B*
b.withdraw(100)			
•••	waits for *U*'s lock on *B*	*a.withdraw(200);*	waits for T's lock on *A*
	(timeout elapses)	•••	
	T's lock on *A* becomes vulnerable, unlock *A*, abort T	•••	
		a.withdraw(200);	write locks *A*
			unlock *A*, *B*

Disadvantages?

Strategies to Fight Deadlock

- Lock timeout (costly and open to false positives)
- Deadlock Prevention: violate one of the necessary conditions for deadlock (from 2 slides ago), e.g., lock all objects before transaction starts, aborting entire transaction if any fails
- Deadlock Avoidance: Have transactions declare max resources they will request, but allow them to lock at any time (Banker's algorithm)
- Deadlock Detection: detect cycles in the wait-for graph, and then abort one or more of the transactions in cycle
- We have seen locking has some problems
- OCC based on the following simple idea:

Distributed Transactions

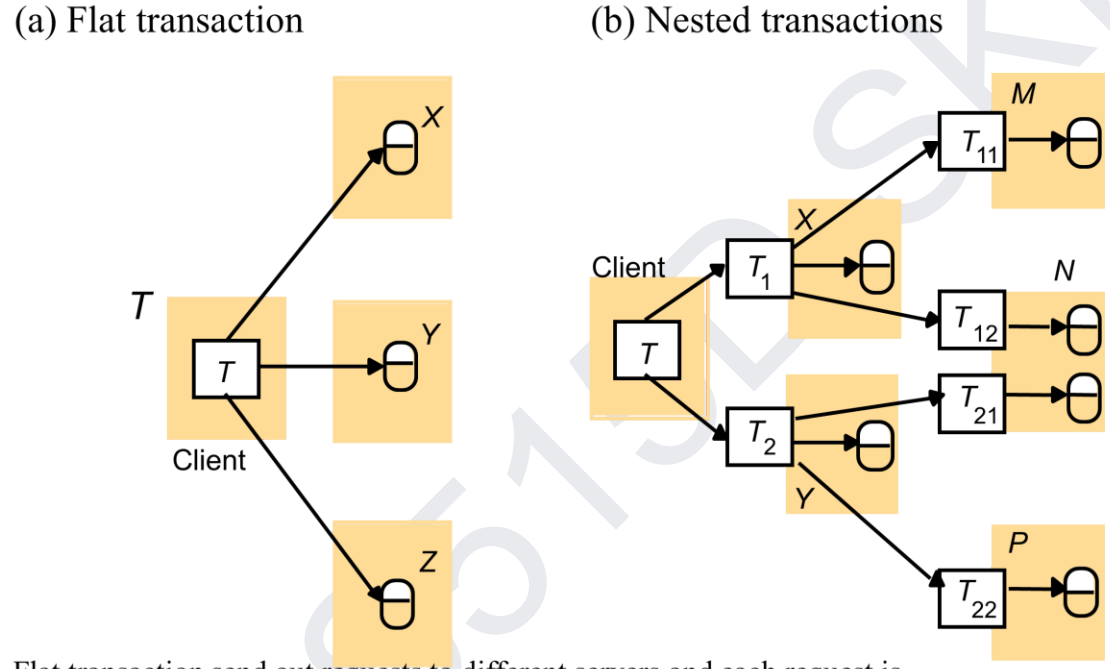
BCS515D,SKIT

In previous chapter, we discussed transactions accessed objects at a single server. In the general case, a transaction will access objects located in different computers. Distributed transaction accesses objects managed by multiple servers.

The atomicity property requires that either all of the servers involved in the same transaction commit the transaction or all of them abort. Agreement among servers are necessary.

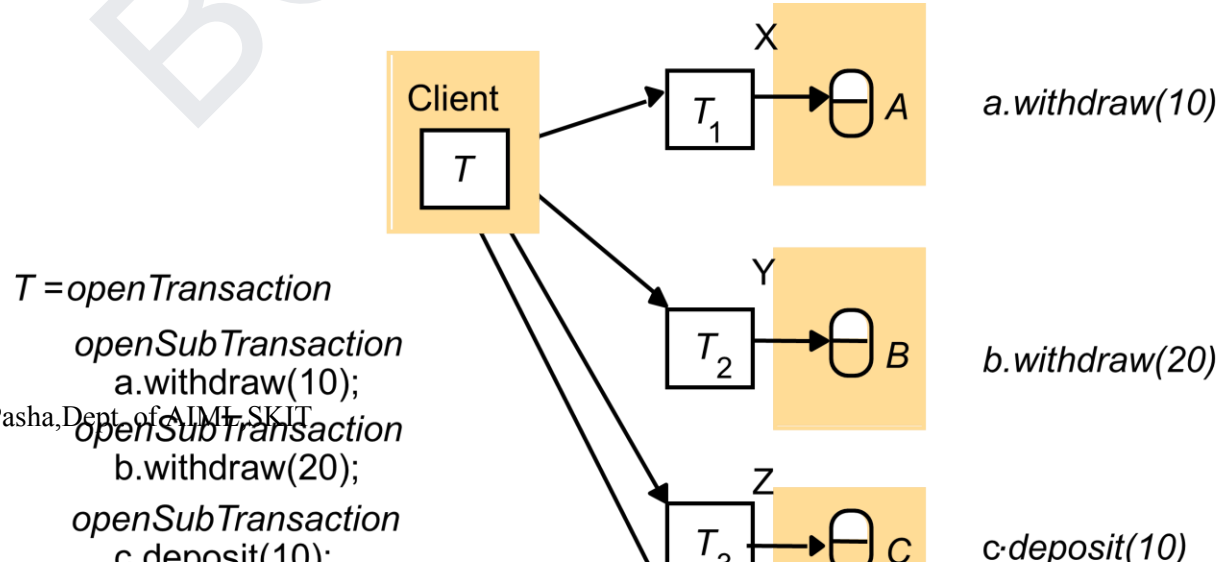
Transaction recovery is to ensure that all objects are recoverable. The values of the objects reflect all changes made by committed transactions and none of those made by aborted ones.

Figure 14.1
Distributed transactions



Flat transaction send out requests to different servers and each request is completed before client goes to the next one. Nested transaction allows sub-transactions at the same level to execute concurrently.

Instructor's Guide for Coulouris, Dollimore and Kindberg Distributed Systems: Concepts and Design Edn.4
© Pearson Education 2005



A transaction comes to an end when the client requests that a transaction be committed or aborted.

Simple way is: coordinator to communicate the commit or abort request to all of the participants in the transaction and to keep on repeating the request until all of them have acknowledged that they had carried it out.

Inadequate because when the client requests a commit, it does not allow a server to make a unilateral decision to abort a transaction. E.g. deadlock avoidance may force a transaction to abort at a server when locking is used. So any server may fail or abort and client is not aware.

Allow any participant to abort its part of a transaction. Due to atomicity, the whole transaction must also be aborted.

In the first phase, each participant votes for the transaction to be committed or aborted. Once voted to commit, not allowed to abort it. So before votes to commit, it must ensure that it will eventually be able to carry out its part, even if it fails and is replaced.

A participant is said to be in a **prepared** state if it will eventually be able to commit it. So each participant needs to save the altered objects in the permanent storage device together with its status-prepared.

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.

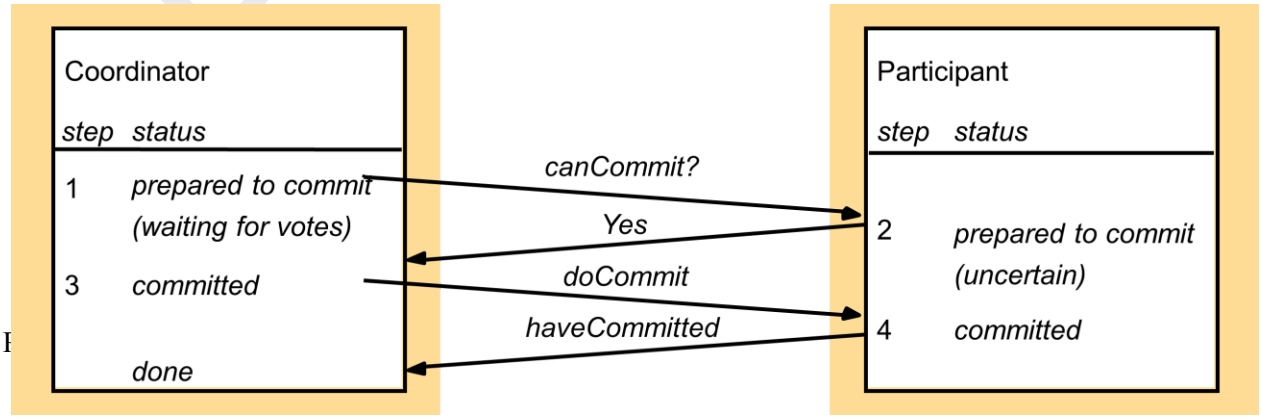
Phase 2 (completion according to outcome of vote):

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.



BCS515D,SKT

Consider when a participant has voted Yes and is waiting for the coordinator to report on the outcome of the vote by telling it to commit or abort.

Such a participant is uncertain and cannot proceed any further. The objects used by its transaction cannot be released for use by other transactions.

Participant makes a getDecision request to the coordinator to determine the outcome. If the coordinator has failed, the participant will not get the decision until the coordinator is replaced resulting in extensive delay for participant in uncertain state.

Timeout are used since exchange of information can fail when one of the servers crashes, or when messages are lost So process will not block forever.

Provided that all servers and communication channels do not fail, with N participants

N number of canCommit? Messages and replies

Followed by N doCommit messages

The cost in messages is proportional to 3N

The cost in time is three rounds of message.

The cost of haveCommitted messages are not counted, which can function correctly without them- their role is to enable server to delete stale coordinator information.

Performance of two-phase commit protocol

Provided that all servers and communication channels do not fail, with N participants

N number of *canCommit?* Messages and replies

Followed by N *doCommit* messages

The cost in messages is proportional to 3N

The cost in time is three rounds of message.

The cost of *haveCommitted* messages are not counted, which can function correctly without them- their role is to enable server to delete stale coordinator information.

When a participant has voted *Yes* and is waiting for the coordinator to report on the outcome of the vote, such participant is in uncertain stage. If the coordinator has failed, the participant will not be able to get the decision until the coordinator is replaced, which can result in extensive delays for participants in the uncertain state.

Concurrency Control in Distributed Transactions

Concurrency control for distributed transactions: each server applies local concurrency control to its

BCS515D,SKIT

own objects, which ensure transactions serializability locally.

However, the members of a collection of servers of distributed transactions are jointly responsible for ensuring that they are performed in a serially equivalent manner. Thus global serializability is required.

Locks

Lock manager at each server decide whether to grant a lock or make the requesting transaction wait.

However, it cannot release any locks until it knows that the transaction has been committed or aborted at all the servers involved in the transaction.

A lock managers in different servers set their locks independently of one another. It is possible that different servers may impose different orderings on transactions.

Timestamp ordering concurrency control

In a single server transaction, the coordinator issues a unique timestamp to each transaction when it starts. Serial equivalence is enforced by committing the versions of objects in the order of the timestamps of transactions that accessed them.

In distributed transactions, we require that each coordinator issue globally unique time stamps. The coordinators must agree as to the ordering of their timestamps. <local timestamp, server-id>, the agreed ordering of pairs of timestamps is based on a comparison in which the server-id is less significant.

The timestamp is passed to each server whose objects perform an operation in the transaction.

To achieve the same ordering at all the servers, The servers of distributed transactions are jointly responsible for ensuring that they are performed in a serially equivalent manner. E.g. If T commits after U at server X, T must commits after U at server Y.

Conflicts are resolved as each operation is performed. If the resolution of a conflict requires a transaction to be aborted, the coordinator will be informed and it will abort the transaction at all the participants.

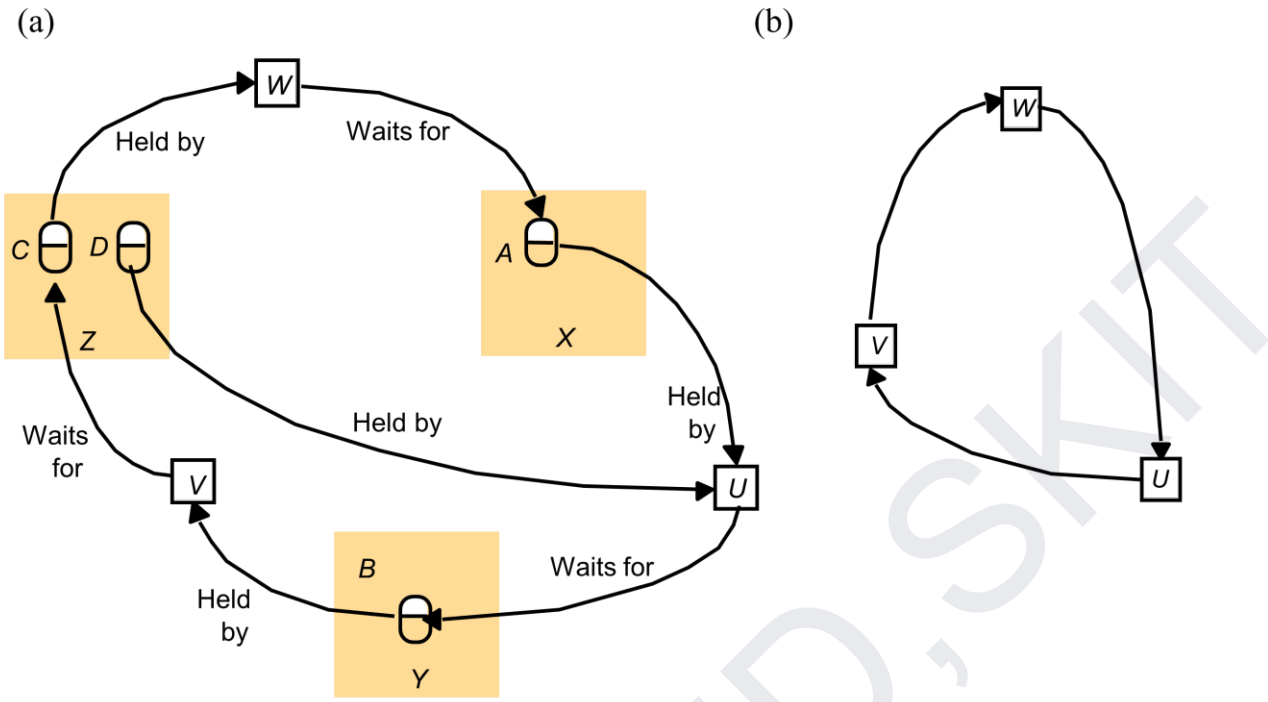
Distributed Deadlock

Deadlocks can arise within a single server when locking is used for concurrency control. Servers must either prevent or detect and resolve deadlocks.

Using timeout to resolve deadlock is a clumsy approach. Why? Another way is to detect deadlock by detecting cycles in a wait for graph

BCS515D,SKIT

Figure 14.14
Distributed deadlock

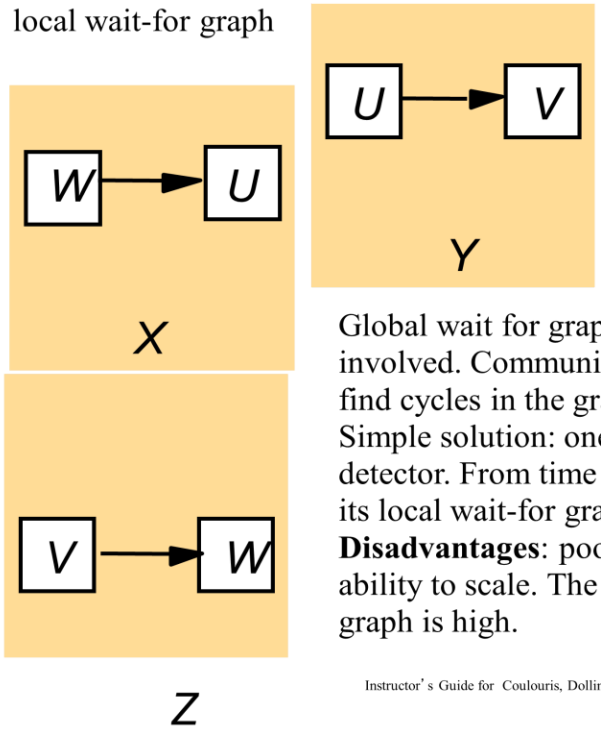


Instructor's Guide for Coulouris, Dollimore and Kindberg Distributed Systems: Concepts and Design Edn.4
© Pearson Education 2005

- A deadlock that is detected but is not really a deadlock is called a phantom deadlock.
- As the procedure of sending local wait-for graph to one place will take some time, there is a chance that one of the transactions that holds a lock will meanwhile have released it, in which case the deadlock will no longer exist.

BCS515D,SKIT

Figure 14.14
Local and global wait-for graphs



Global wait for graph is held in part by each of the several servers involved. Communication between these servers is required to find cycles in the graph.
Simple solution: one server takes on the role of global deadlock detector. From time to time, each server sends the latest copy of its local wait-for graph.
Disadvantages: poor availability, lack of fault tolerance and no ability to scale. The cost of frequent transmission of local wait-for graph is high.

Instructor's Guide for Coulouris, Dollimore and Kindberg Distributed Systems: Concepts and Design Edn.4
© Pearson Education 2005

At about the same time, T waits for U (T->U) and W waits for V (W->V). Two probes occur, two deadlocks detected by different servers.

We want to ensure that only one transaction is aborted in the same deadlock since different servers may choose different transaction to abort leading to unnecessary abort of transactions.

So using priorities to determine which transaction to abort will result in the same transaction to abort even if the cycles are detected by different servers.

Using priority can also reduce the number of probes. For example, we only initiate probe when higher priority transaction starts to wait for lower priority transaction.

If we say the priority order from high to low is: T, U, V and W. Then only the probe of T->U will be sent and not the probe of W->V.

Transaction recovery

Atomic property of transactions can be described in two aspects:

Durability: objects are saved in permanent storage and will be available indefinitely thereafter. Acknowledgement of a client's commit request implies that all the effects of the transaction have been recorded in permanent storage as well as in the server's volatile object.

Failure atomicity: the effects of transactions are atomic even when the server crashes.

Both can be realized by recovery manager.

Recovery manager

Tasks of a recovery manager:

Save objects in permanent storage (in a recovery file) for committed transactions;

To restore the server's objects after a crash;

To reorganize the recovery file to improve the performance of recovery;

To reclaim storage space in the recovery file.

Log with entries relating to two-phase commit protocol

In phase 1, when the coordinator is prepared to commit and has already added a prepared status entry, its recovery manager adds a coordinator entry. Before a participant can vote Yes, it must have already prepared to commit and must have already added a prepared status entry. When it votes Yes, its recovery manager records a participant entry and adds an uncertain status. When a participant votes No, it adds an abort status to recovery file.

In phase 2, the recovery manager of the coordinator adds either a committed or an aborted, according to the

Figure 14.18
Types of entry in a recovery file

Type of entry	*Description of contents of entry*
Object	A value of an object.
Transaction status	Transaction identifier, transaction status (*prepared*, *committed* *aborted*) and other status values used for the two-phase commit protocol.
Intentions list	Transaction identifier and a sequence of intentions, each of which consists of <identifier of object>, < Position of value of object>.

Intention list records all of its currently active transactions. A list of a particular transaction contains a list of the references and the values of all the objects that are altered. When committed, the committed version of each object is replaced by the tentative version made by that transaction. When a transaction aborts, the server uses the intention list to delete all the tentative versions of objects.

When a participant says it is prepared to commit, its recovery manager must have saved both its intention list for that transaction and the objects in that intention list in its recovery file, so it will be able to carry out the commitment later on, even if it crashes in the interim.

Instructor's Guide for Coulouris, Dollimore and Kindberg Distributed Systems: Concepts and Design Edn.4
© Pearson Education 2005

decision. Recovery manager of participants add a commit or abort status to their recovery files according to

BCS515D-SKIT

message received from coordinator. When a coordinator has received a confirmation from all its participants, its recovery manager adds a done status.

When a server is replaced after a crash, the recovery manager has to deal with the two-phase commit protocol in addition to restore the objects.

For any transaction where the server has played the coordinator role, it should find a coordinator entry and a set of transaction status entries. For any transaction where the server has played the participant role, it should find a participant entry and a set of set of transaction status entries. In both cases, the most recent transaction status entry, that is the one nearest the end of log determine the status at the time of failure.

The action of the recovery manage with respect to the two-phase commit protocol for any transaction depends on whether the server was the coordinator or a participant and on its status at the time of failure as shown in the following table.

Figure 14.22
Recovery of the two-phase commit protocol

Role	*Status*	*Action of recovery manager*
Coordinator	*prepared*	No decision had been reached before the server failed. It sends *abortTransaction* to all the servers in the participant list and adds the transaction status *aborted* in its recovery file. Same action for state *aborted*. If there is no participant list, the participants will eventually timeout and abort the transaction.
Coordinator	*committed*	A decision to commit had been reached before the server failed. It sends a *doCommit* to all the participants in its participant list (in case it had not done so before) and resumes the two-phase protocol at step 4 (Fig 13.5).
Participant	*committed*	The participant sends a *haveCommitted* message to the coordinator (in case this was not done before it failed). This will allow the coordinator to discard information about this transaction at the next checkpoint.
Participant	*uncertain*	The participant failed before it knew the outcome of the transaction. It cannot determine the status of the transaction until the coordinator informs it of the decision. It will send a *getDecision* to the coordinator to determine the status of the transaction. When it receives the reply it will commit or abort accordingly.
Participant	*prepared*	The participant has not yet voted and can abort the transaction.
Coordinator	*done*	No action is required.

Instructor's Guide for Coulouris, Dollimore and Kindberg Distributed Systems: Concepts and Design Edn.4
© Pearson Education 2005