The UNIVERSITY *of* EDINBURGH
SCHOOL *of* COMPUTER SCIENCE

CS4/MSc

Distributed Systems

Jane Hillston

jeh@dcs.ed.ac.uk

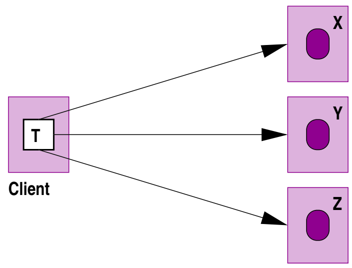
Room 1604

(Lecture 14: Distributed Transactions and Atomic Commit,
21st February 2002)

Distributed Transactions

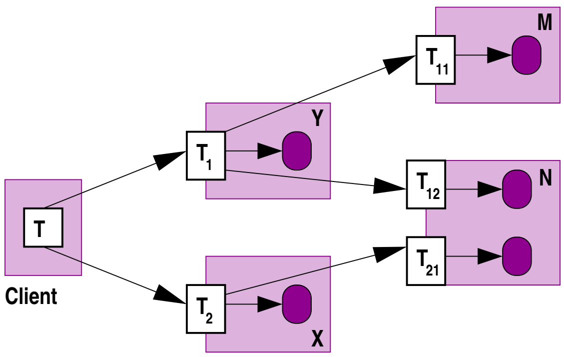
- A transaction is **distributed** if it invokes operations in several different servers.
- One of the servers takes on the role of **coordinator** to ensure that the atomicity property is kept and that either all the servers commit or they all abort.
- This process is managed by a **commit protocol**, the most common of which is the **two-phase commit protocol**.
- Each server applies concurrency control locally to its own objects to ensure that the local effects are serially equivalent.
- However additional procedures are needed to ensure that distributed transactions are globally serialised.
- To enhance concurrency it is often useful to organise transactions in terms of **nested transactions**: transactions within transactions, with subtransactions able to proceed in parallel.

Nested Transactions



In the previous lecture we considered transactions which were without internal structure, so-called **flat transactions**. Such transactions consist of a sequence of operations which are carried out serially.

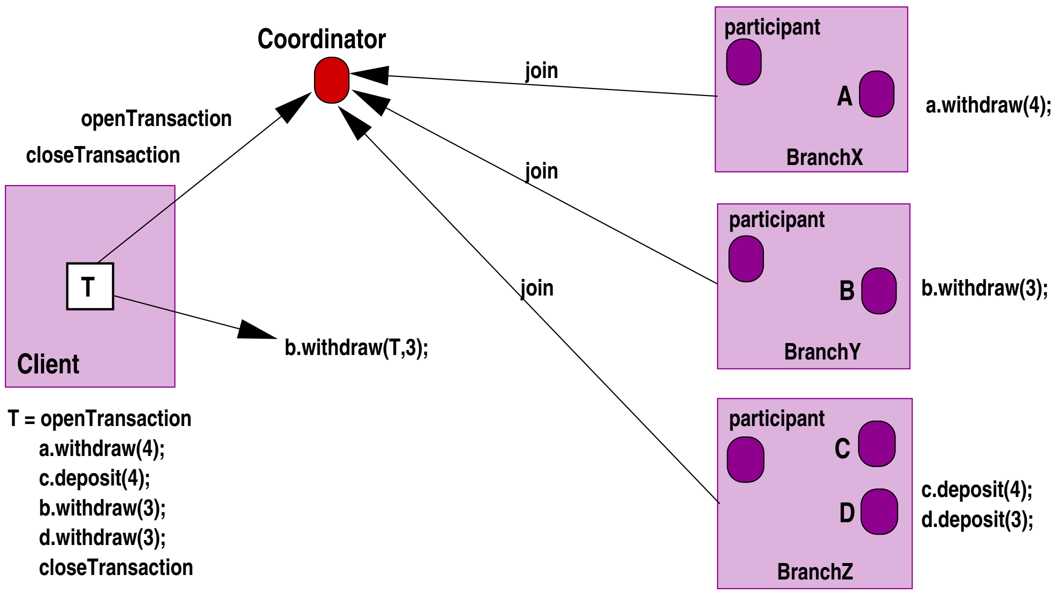
A **nested transaction** is made up of a number of **subtransactions**, which may in turn consist of subtransactions. Subtransactions at the same level can run in parallel if they are accessing objects in different servers.



Role of Coordinator

- A client starts a transaction by sending an `openTransaction` request to a coordinator in any server.
- The coordinator returns the transaction ID which now contains a server identifier (IP address) in addition to the transaction number assigned by the server.
- The coordinator will keep the role of coordinator with respect to that transaction throughout its lifetime, and in the end will be responsible for committing or aborting it.
- Other servers which manage objects accessed by the transaction are termed **participants** and each must be involved in the decision to commit or abort. Each has a reference to the coordinator.
- During the course of the transaction the coordinator records a list of references to the participants that have been involved, via the `join` method that each participant invokes on the coordinator whenever it becomes involved in a transaction.
- Each participant must manage its own recoverable objects.

Coordinator Example



Although separate for clarity here, the coordinator would be situated in one of the servers involved in the transaction (`BranchX` say).

Two-phase Commit Protocol

- When an distributed transaction comes to an end either all its operations are carried out or none of them (atomicity).
- Since events at each participant (which will not necessarily be known at the coordinator) may mean that a transaction has to be aborted, the commitment of a transaction proceeds in two steps.
- Thus each server can abort its part of the transaction due to local conditions, and then force the whole transaction to be aborted.
- In the first phase of the protocol each participant votes for the transaction to be committed or aborted.
- Each participant is allowed only one vote, so a vote to commit is binding even in the event of a subsequent failure at the server.
- A participant is said to be in a **prepared state** for a transaction if it will eventually be able to commit it.
- In the second phase of the protocol every participant in the transaction carries out the joint decision.

Two-phase Commit Protocol (2)

Phase 1

1. Coordinator sends a `canCommit?` request to each participant.
2. When a participant receives a `canCommit?` request it replies with its vote (`Yes` or `No`). If voting `Yes` it prepares to commit by saving objects in permanent storage. If voting `No` it aborts immediately.

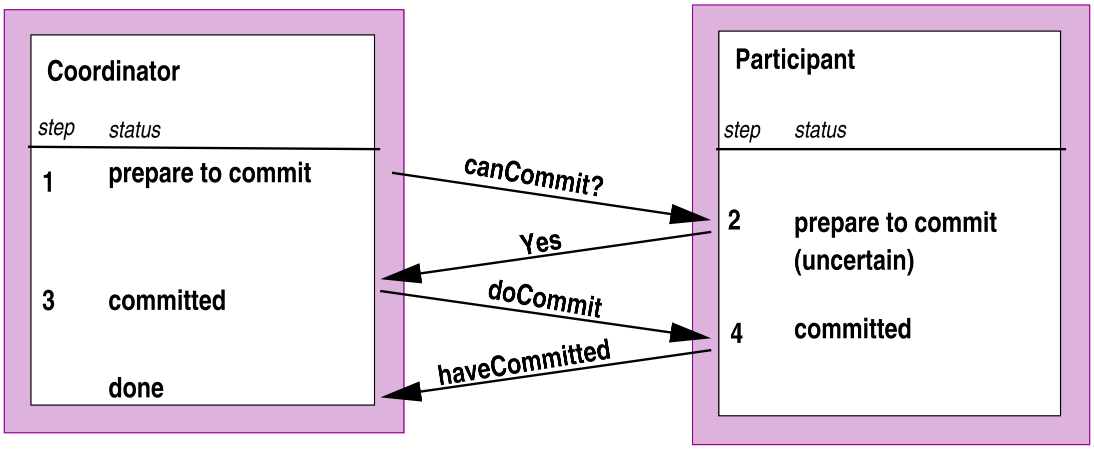
Phase 2

3 The coordinator collects votes (including its own).

  - If there are no failures and all votes are `Yes` coordinator decides to commit and sends everyone a `doCommit` request.
  - Otherwise the coordinator decides to abort and sends `doAbort` request to each participant that voted `Yes`.

4 Participants that voted `Yes` are waiting for a message from the coordinator. When a participant receives a `doCommit` or `doAbort` request it acts accordingly and in the case of commit sends a `haveCommitted` message to the coordinator.

Two-phase Commit Protocol (3)



Without errors this is quite straightforward but it must also work correctly when some of the servers fail, messages are lost or servers are temporarily unable to communicate. A participant can use the `getDecision` method of the coordinator if it is left at step 2.

Two-phase Commit for Nested Transactions

- A subtransaction can only commit if its parent transaction also commits but a parent transaction may decide to commit even if one or more of its subtransactions has aborted.
- If a subtransaction completes successfully it **provisionally commits** indicating its readiness to commit and reports its status to its parent together with the status of each of its descendants.
- Each subtransaction has a coordinator and the coordinator of the top-level transaction carries out a two-phase commit protocol.

  Hierarchic: `CanCommit?` messages are sent down the tree from the top-level coordinator to the coordinators of its subtransactions and so on. Each participant collects replies from its descendants before replying to its parent.

  Flat: The top-level coordinator sends `CanCommit?` messages to the coordinators of all subtransactions in the provisional commit list, together with a list of aborted subtransactions so that descendants of aborted subtransactions can vote accordingly.

Concurrency Control in Distributed Transactions

Each server is responsible for applying concurrency control to its own objects, and the set of servers is jointly responsible for ensuring that distributed transactions are performed in serially equivalent order.

This implies that if transaction T is before transaction U in their conflicting access to objects at one of the servers then they must be in that order at all the servers whose objects are accessed in a conflicting manner by both T and U. How this is achieved depends on the method of concurrency control used.

Optimistic Concurrency Control: If validation is only carried out locally conflicts may be resolved differently at different servers, for example T before U at server X but U before T at server Y. Various schemes have been proposed for making the validation phase global. One problem is that **commitment deadlock** can occur as servers wait to see if another transaction is validated to decide whether their current transaction can be validated.

Timestamp Ordering: To achieve the same ordering at all servers the coordinators must agree on the ordering of their timestamps. A timestamp consists of a pair *(local timestamp, server-id)*. The same ordering can be achieved at all servers even without clock synchronisation but greater efficiency is achieved with it.

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

Locks: Each server grants locks locally for each of its objects, with the local lock manager deciding whether each transaction gains the lock or must wait. Locks are held until the transaction has been committed or aborted at all servers involved. When two phase commit is used locks will be released at phase 1 for aborted transactions, but will remain held until the protocol is complete otherwise.

Since lock managers operate independently they may grant locks in different orders leading to cyclic dependencies and deadlocks.

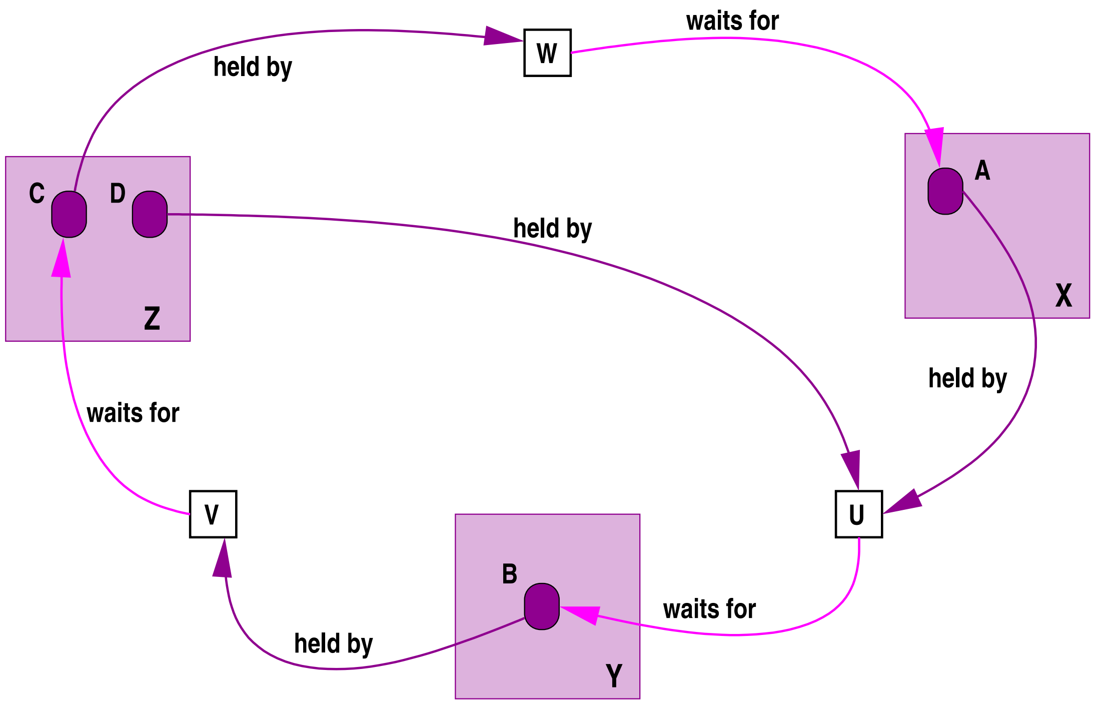
Distributed Deadlock

- Most deadlock detection algorithms operate by finding cycles in the transaction **waits-for** graph.
- A **waits-for** graph is a directed graph in which nodes represent transactions or objects and edges represent either that an object is held by a transaction or that a transaction is waiting for an object.
- There is a deadlock iff there is a cycle in the waits-for graph.
- There can be a cycle in the global waits-for graph that is not in any single local one.
- In theory at least, a global waits-for graph can be constructed by amalgamating the local waits-for graph from each server (lock manager) involved in the transactions. In this case one server would be nominated **global deadlock detector** and is responsible for amalgamating the graphs and detecting cycles.
- In practice cycles are found using the distributed graphs by communication between the servers.

Distributed Deadlock Example (1)

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

Distributed Deadlock Example (2)



Distributed Deadlock Detection – Edge Chasing

- The global waits-for graph is not constructed.
- Servers attempt to find cycles by forwarding messages called **probes** which follow the edges of the graph through the distributed system.
- A probe message consists of transaction waits-for relationships representing a path in the global waits-for graph, e.g. a message $\langle T \rightarrow U \rangle$ represents the edge of the graph showing that transaction T is blocked by U.
- A probe is sent by a server when a transaction is blocked at the server by a transaction which is itself blocked at another server. The probe is sent to the other server.
- The transaction's coordinator is responsible for recording whether a transaction is active or blocked and all participants can get this information from the coordinator.
- Lock managers inform coordinators when transactions start waiting for objects and when transactions acquire objects and become active again.

Edge-Chasing Algorithms

Initiation: When a server notes that a transaction T starts waiting for an object held by another transaction U and U is itself waiting to access an object at another server, the server will initiate the deadlock detection algorithm by sending a probe message to the server of the object at which U is blocked. The message will contain the edge $\langle T \rightarrow U \rangle$.

Detection: When a server receives a probe ending with transaction U say, it checks to see if U is waiting for an object; if it is the transaction it is waiting for is added to the probe, e.g. $\langle T \rightarrow U \rightarrow V \rangle$. If this transaction is waiting for another object elsewhere the probe is forwarded. First however the server checks to see if the transaction just added already appears in the probe. If so, a deadlock has been detected.

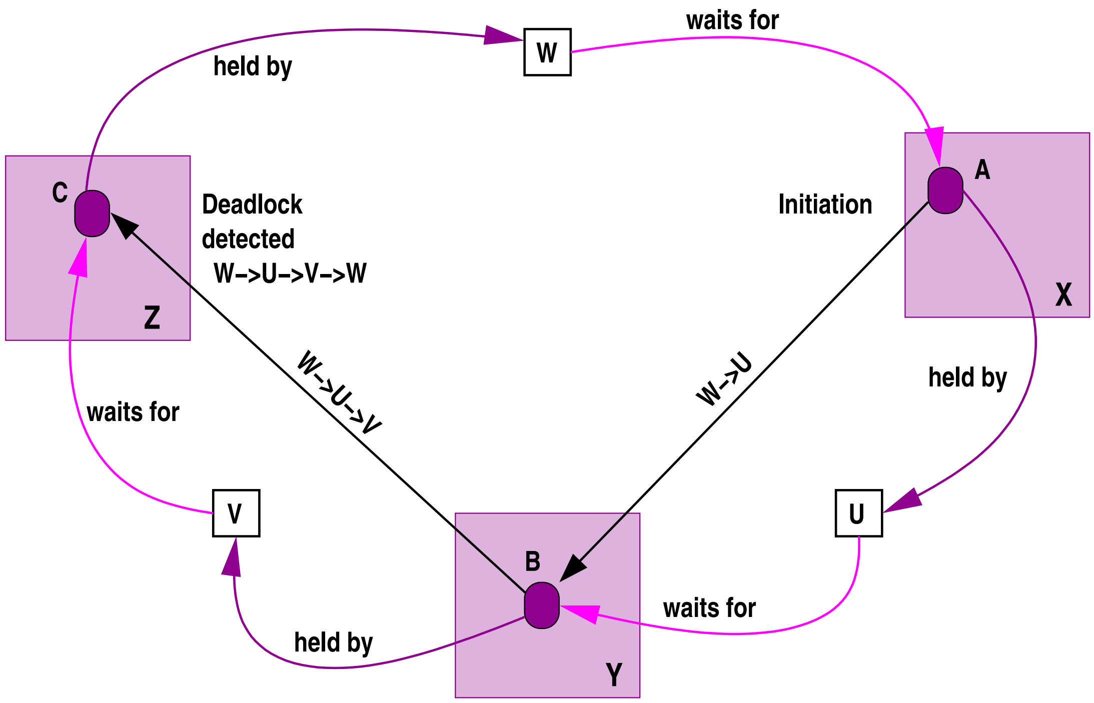
Edge-Chasing Algorithms (2)

Resolution: When a cycle is detected, e.g. T has just been added to the probe $\langle T \rightarrow U \rightarrow V \rightarrow T \rangle$, one transaction in the cycle is aborted to break the deadlock. Its coordinator will inform the participants and all its locks will be removed with the effect of all edges involving the transaction will be removed from local waits-for graphs.

In the following example we observe the following steps:

- Server X initiates detection by sending probe $\langle W \rightarrow U \rangle$ to the server of B (Server Y).
- Server Y receives probe $\langle W \rightarrow U \rangle$, notes that B is held by V and appends V to the probe ($\langle W \rightarrow U \rightarrow V \rangle$). It notes that V is waiting for C at Server Z; the probe is forwarded to Server Z.
- Server Z receives probe $\langle W \rightarrow U \rightarrow V \rangle$ and notes C is held by W and appends W to the probe to produce $\langle W \rightarrow U \rightarrow V \rightarrow W \rangle$.

Deadlock Detection Example



Distributed Deadlock Detection Practicalities

- In practice, in most edge-chasing algorithms, the server sends the probe to the coordinator of the last transaction on the path. If the transaction is waiting for an object the coordinator forwards the probe to the relevant server. Thus two messages are required for each time the probe is forwarded.
- A probe detecting a cycle involving N transactions will be forwarded by $(N-1)$ transaction coordinators via $(N-1)$ object servers, requiring $2(N-1)$ messages. However studies of databases have shown that the majority of deadlocks involve only two transactions.
- Every transaction involved in a deadlock may initiate deadlock detection concurrently which may result in several servers detecting deadlock and needlessly aborting more than one transaction. To avoid this problem totally ordered priorities are assigned to transactions (e.g. timestamps). Then, when a deadlock is detected, the transaction with the lowest priority is always the one to be aborted.