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

CS4/MSc

Distributed Systems

Jane Hillston

jeh@dcs.ed.ac.uk

Room 1604

(Lecture 13: Transactions and Concurrency Control,
18th February 2002)

Transactions

Transactions originate in the area of contract law. Parties can negotiate to make an agreement. During the negotiations they can back out and return to their initial state. However, once they have signed a contract they are bound by it.

In computer science transactions originate from database management systems, although transactions are now applied more widely than just to databases and can encompass more than file operations.

A transaction

- runs on shared entities such as databases, files or remote objects;
- is a unit of consistency, meaning that it transforms the entity from one consistent state to another consistent state; and
- has effects which are either made permanent (**committed**) or cancelled (**aborted**) so that the entity reverts back to its original state.

ACID properties

To cope with failures and concurrency issues transaction systems must guarantee the **ACID** properties:

Atomicity: a transaction must be all or nothing, i.e. to the rest of the system a transaction should appear as an indivisible action, even in the presence of failures.

Consistency: a transaction takes the system from one consistent state to another consistent state, i.e. the transaction does not violate system invariants;

Isolation: when several transactions are executed concurrently, the intermediate effects of a transaction are not visible to other transactions, i.e. transactions cannot interfere with one another;

Durability: once a transaction has committed its effects cannot be cancelled by the execution of an uncommitted transaction even in the presence of failures.

ACID Implications — Recoverability

The requirements of atomicity and durability mean that entities must be **recoverable**.

An entity is **recoverable** if when the server process crashes (due to hardware fault or software error) the changes due to completed transactions are not lost. Furthermore when the server is replaced by a new server (or, equivalently, restarted) the entity can be restored.

This implies that the state of each entity must be recorded in permanent storage, for example, written to disk. Moreover this must be done by the time that the server acknowledges the completion of a client's transaction.

ACID Implications — Synchronisation

The requirement of isolation means that the server must synchronise operations on entities in order to ensure that no interference occurs.

A simple solution is to impose a sequential order on transactions so that only one transaction is working at a time. This safe approach has large performance penalties.

To improve performance the server seeks to maximise concurrency, whilst maintaining isolation. In other words transactions are allowed to execute concurrently when it is certain that the effect is the same as if they executed serially. Such transactions are said to be **serially equivalent** or **serialisable**.

Transaction Coordinator

Each transaction is created and managed by a coordinator which should implement the following operations (client-side):

`openTransaction()` starts a new transaction and returns a unique **transaction ID (TID)** `trans`. This identifier is used in other operations in the transaction.

`closeTransaction(trans)` ends a transaction; the return value is used to indicate the success or failure. Instigates the saving of all recoverable objects accessed by the transaction. A return value of `commit` indicates that the transaction has been committed; a return value of `abort` indicate that it has been aborted.

`abortTransaction(trans)` is used when the client wants to abort the transaction. All recoverable objects accessed by the transaction will be **rolled back** to their state before the transaction started.

All the client's operations between the `openTransaction()` and the `closeTransaction(trans)`/`abortTransaction(trans)` calls are operations of the transaction.

Server Process Crashes

A transaction can also be aborted by the server process as the result of a process crash. We assume in this situation that the server will be restored or replaced after an unexpected failure.

- The new (restored) server process aborts any uncommitted transactions which were in progress when the crash occurred. It also starts a recovery procedure which reverts all objects to the values produced by the most recently committed transaction.
- The client will be made aware of the server crash when an operation returns an exception after a timeout. If the server is restored while the transaction is still in progress the client is again informed via an exception as the transaction will no longer be valid.

Servers can assign expiry times to transactions in order to detect client crashes. Any transaction which has not committed by its expiry time will be aborted by the server.

Transactions in CORBA

- In CORBA transactions are supported by the *Object Transaction Service*.
- IDL interfaces allow client transactions to include not just multiple accesses to a single object but also multiple objects at multiple servers.
- The client is provided with operations to specify the beginning and end of a transaction, as explained on the earlier slide.
- The TID is implicitly passed as an additional argument in each included remote method invocation, i.e. those made between calls to start the transaction and those to complete/abort it.
- The client ORB maintains a context for each transaction, which it propagates with each remote method invocation in that transaction.
- CORBA objects can be made transactional by making their interfaces extend an interface called `TransactionalObject`.

Concurrency Control

Concurrency control protocols are used to serialise transactions in their access to shared entities. The following are the main approaches to concurrency control:

Locking: the first transaction locks the entity; others cannot access it until the first transaction has committed or aborted.

Optimistic concurrency control: transactions proceed until they are ready to commit; at that point a check is performed. If there are conflicts the transaction is aborted and restarted.

Timestamps: transactions have timestamps, entities are timestamped when they are accessed. Timestamps are compared and transaction may proceed, delay or abort as a result.

The problem of concurrency control is to allow transactions to proceed concurrently and yet achieve the same **effect** (access operations return the same values and variables store the same values) as if the transactions had been performed sequentially or **serially**.

The lost update problem

Without proper concurrency control the operations of one transaction may mask the operations of another. For example in this banking example U's update of account B is lost because T overwrites it.

Transaction T:		**Transaction U:**	

`bal = b.getBalance();`
`b.setBalance(bal*1.1);`
`a.withdraw(bal/10);`
		`bal = b.getBalance();`
		`b.setBalance(bal*1.1);`
		`c.withdraw(bal/10);`	

`bal = b.getBalance();`	$200		
		`bal = b.getBalance();`	$200
		`b.setBalance(bal*1.1);`	$220
`b.setBalance(bal*1.1);`	$220		
`a.withdraw(bal/10);`	$80		
		`c.withdraw(bal/10);`	$280

Serially equivalent interleavings

A serially equivalent interleaving has the same effect as if two transactions were performed one after the other. If each individual transaction is correct then we can assume that performing one after the other is correct, and thus that the serial interleaving is correct.

Transaction T:		**Transaction U:**	

`bal = b.getBalance();`
`b.setBalance(bal*1.1);`
`a.withdraw(bal/10);`
		`bal = b.getBalance();`
		`b.setBalance(bal*1.1);`
		`c.withdraw(bal/10);`	

`bal = b.getBalance();`	$200		
`b.setBalance(bal*1.1);`	$220		
		`bal = b.getBalance();`	$220
		`b.setBalance(bal*1.1);`	$242
`a.withdraw(bal/10);`	$80		
		`c.withdraw(bal/10);`	$278

The inconsistent retrievals problem

The inconsistent retrieval problem can occur when a retrieval transaction is running concurrently with an update transaction. If operations of the update become interleaved with the retrieval operations, the sequence of retrievals no longer represent a consistent state.

Transaction V:
`a.withdraw(100);`
`b.deposit(100);`
		Transaction W:
		`aBranch.branchTotal()`	

`a.withdraw(100);`	$100		
		`total = a.getBalance();`	$100
		`total = total + b.getBalance();`	$300
		`total = total + c.getBalance();`	⋮
		⋮	
`b.deposit(100);`	$300		

A serially equivalent interleaving of V and W

Here the solution is to serialise the transactions: the retrieval should be completely wholly before or wholly after the update transaction.

Transaction V:
`a.withdraw(100);`
`b.deposit(100);`
		Transaction W:
		`aBranch.branchTotal()`	

`a.withdraw(100);`	$100		
`b.deposit(100);`	$300		
		`total = a.getBalance();`	$100
		`total = total + b.getBalance();`	$400
		`total = total + c.getBalance();`	⋮
		⋮	

Conflicting Operations

- A pair of operations are said to **conflict** if their combined effect is determined by the order in which they are executed.
- Examples of conflicting operations are `read` and `write`.
- Serial equivalence can be defined in terms of operation conflicts:
  - *For two transactions to be* **serially equivalent**, *it is necessary and sufficient that all pairs of conflicting operations of the two transactions be executed in the same order at all the objects they both access.*
- This means that if two transactions T_1 and T_2 have conflicting operations on two objects O_1 and O_2 then either
  - T_1 accesses O_1 before T_2 and T_1 accesses O_2 before T_2; or
  - T_2 accesses O_1 before T_1 and T_2 accesses O_2 before T_1.

Locks

- Locks are the most widely used form of concurrency control.
- The idea is that in order to gain access to an object a transaction must first acquire a lock on the object. As the transaction progresses through its operation it will acquire locks for each of the objects it needs (the **growing phase**).
- When the operations are complete the transaction releases locks (the **shrinking phase**).
- This approach is known as **two-phase locking**.
- To avoid problems such as **dirty reads** and **premature writes** the locking scheme may be **strict** in the sense that all acquired locks are held until the transaction commits or aborts.
- Different classes of locks may be used to distinguish types of access, e.g. **read** and **write** locks with only write locks being **exclusive**.

Transactions T and U with strict exclusive locks

Transaction T:		**Transaction U:**	
`bal = b.getBalance();`		`bal = b.getBalance();`	
`b.setBalance(bal*1.1);`		`b.setBalance(bal*1.1);`	
`a.withdraw(bal/10);`		`c.withdraw(bal/10);`	
Operations	Locks	Operations	Locks
`openTransaction();`			
`bal = b.getBalance();`	lock B		
`b.setBalance(bal*1.1);`		`openTransaction();`	
`a.withdraw(bal/10);`	lock A	`bal = b.getBalance();`	wait for B lock
`closeTransaction();`	unlock A, B		
			lock B
		`b.setBalance(bal*1.1);`	
		`c.withdraw(bal/10);`	lock C
		`closeTransaction();`	unlock B,C

Lock Implementation

- When a locking scheme is used each server will run a **lock manager** to grant locks for each of the objects on the server.
- The manager will maintain a mapping of locks and objects, for example, in a hash table.
- When a transaction requires a lock it sends a request to the lock manager giving details of the object and the type of lock required.
- Locking schemes are prone to **deadlock** when a set of transactions become blocked each waiting for a lock held by one of the others.
- One solution is for the lock manager to maintain a **waits-for** graph and periodically check it to detect cycles. When a cycle is detected one or more transactions must be aborted, releasing the locks it holds.
- An alternative approach is deadlock avoidance, which relies on each transaction acquiring all the locks it needs before it performs any operations.

Optimistic Concurrency Control

- Optimistic concurrency control is based on the assumption that the set of objects managed by a server is large and that most transactions will not run into conflict.
- Each transaction maintains its own **tentative version** of the object.
- During the **working phase** the transaction carries out operations on its tentative version of relevant objects, keeping track of which objects it has accessed in a **read set** and a **write set**.
- When the transaction is complete it enters a **validation phase**. Using the read set and write set the transaction is validated to establish whether or not its operations on objects conflict with operations of other transactions on the same objects.
- If the transaction is validated the **update phase** is entered and all changes recorded in the tentative version of objects are made permanent. The transaction then commits. Otherwise the transaction must abort.

Timestamp Ordering

- In the timestamp ordering approach to concurrency control each transaction is assigned a unique timestamp value when it starts.
- Requests from transactions can be totally ordered according to their timestamps.
- Each operation in a transaction is validated when it is carried out, against a simple timestamp ordering rule.
  - *A transaction's request to write an object is valid only if that object was last read and written by earlier transactions. A transaction's request to read an object is valid only if that object was last written by an earlier transaction.*
- The rule assumes that there is only one version of each object and restricts access to one transaction at a time; however, it can be adapted to systems in which each transaction has its own tentative version of each object it accesses.