10 Shared Data and Transactions

- Stateful Servers
- Atomicity
- Transactions
- ACID
- Serial Equivalence

10.1 Servers and their state

- Servers manage a resource, such as a database or a printer
- Attempt to limit problems of distributed access by making server *stateless*, such that each request is independent of other requests.
  - Servers can crash in between servicing clients
  - Client requests cannot interfere with each other (assuming concurrency control in server
- But we can't always design stateless servers...

10.1.1 Stateful Servers

- Some applications better modelled as extended conversations, eg retrieving a list of records in a large database better modelled as getting batch of records at a time.
- If application requires state to be consistent across a number of machines, then each machine must recognise when it can update internal data. Needs to keep track of state of distributed conversation
- If long duration then, then need to be aware of state.
- If other conversations need to go on - eg modify records during retrieval, how do we allow concurrency?
- What happens if machine fails - need to recover.
- Should also aim to be fault tolerant

10.2 Atomicity

Stateful server have two requirements

1. Accesses from different clients shouldn't interfere with each other
2. Clients should get fast access to the server

10.2.1 Definition

We define atomicity as

All or Nothing A client's operation on a server's resource should complete successfully, and the results hold thereafter (yea, even unto a server crash), or it should fail and the resource should show no effect of the failed operation

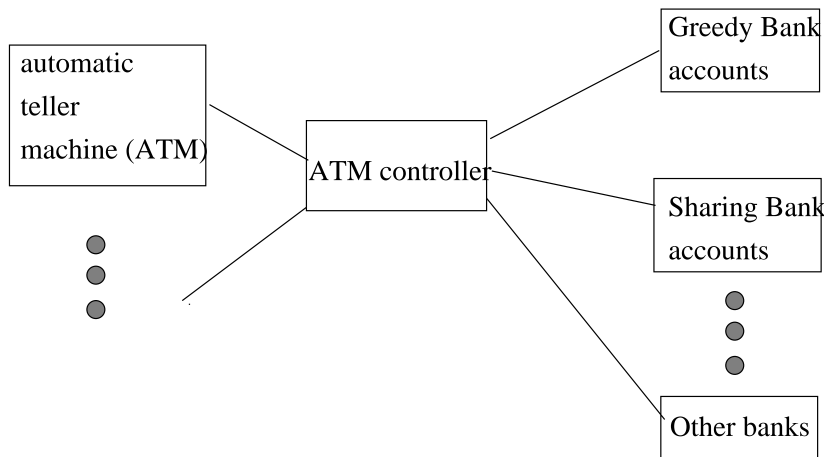
Isolation Each operation should proceed without interference from other clients' operations - intermediate effects should not be visible.

10.2.2 Example

Mutual Exclusion For a multi-threaded server, if two or more threads attempt to modify the same piece of data, then the updates should have mutual exclusion around the updates to provide isolation, using semaphores or monitors

Synchronisation In situations such as Producer Consumer, need to allow one operation to finish so second operation can use results, needing isolation.

10.3 Automatic Teller Machines and Bank accounts



- An ATM or cashmachine allows transfer of funds between accounts.
- Accounts are held at various machines belonging to different banks
- Accounts offer the following operations

  deposit Place an amount of money in an account

  withdraw Take an amount of money from an account

  balance Get the current value in an account

- Operations implemented as read() and write() of values, so withdraw x from A and deposit x in B implemented as

  1. A.write(A.read() - x)
  2. B.write(B.read() + x)

10.4 Transactions

Transactions are technique for grouping operations on data so that either all complete or none complete

Typically server offers transaction service, such as:

beginTransaction(transId) Record the start of a transaction and associate operations with this transId with this transaction.

commitTransaction(transId) Commit all the changes the operations in this transaction have made to permanent storage.

abortTransaction(transId) Abort all the changes the transaction operations have done, and roll back to previous state.

10.4.1 ACID

Transactions are described by the ACID mnemonic

Atomicity Either all or none of the Transaction's operations are performed. If a transaction is interrupted by failure, then partial changes are undone

Consistency System moves from one self-consistent state to another

Isolation An incomplete transaction never reveals partial state or changes before commiting

Durability After committing, the system never loses the results of the transaction, independent of any subsequent failure

10.4.2 Concurrency Problems

Transaction T A.withdraw(4,T) B.deposit(4,T)		Transaction U C.withdraw(3,U) B.deposit(3,U)	
balance = A.read()	£100		
A.write(balance - 4)	£96		
		balance = C.read()	£300
		C.write(balance - 3)	£297
balance = B.read()	£200		
		balance = B.read()	£200
		B.write(balance + 3)	£203
B.write(balance + 4)	£204		

Lost Update

Transaction T A.withdraw(100,T) B.deposit(100,T)		Transaction U Bank.total(U)	
balance = A.read()	£200		
A.write(balance - 100)	£100		
		balance = A.read()	£100
		balance = B.read() + balance	£300
		balance = C.read() + balance	£300+
balance = B.read()	£200		
B.write(balance + 100)	£300		

Inconsistent Retrievals

10.5 Serial Equivalence

Definition: Two transactions are *serial* if all the operations in one transaction precede the operations in the other.

eg the following actions are serial

$R_i(x)W_i(x)R_i(y)R_j(x)W_j(y)$

Definition: Two operations are *in conflict* if:

- At least one is a write
- They both act on the same data
- They are issued by different transactions

$R_i(x)R_j(x)W_i(x)W_j(y)R_i(y)$ has $R_j(x)W_i(x)$ in conflict
Definition: Two schedules are *computationally equivalent* if:

- The same operations are involved (possibly reordered)
- For every pair of operations in conflict, the same operation appears first in each schedule

So, a schedule is serialisable if the schedule is computationally equivalent to a serial schedule.

Question: Is the following schedule for these two transaction serially equivalent?
$R_i(x)R_i(y)R_j(y)W_j(y)R_i(x)W_j(x)W_i(y)$

10.5.1 Transaction Nesting

Transactions may themselves be composed of multiple transactions

eg *Transfer* is a composition of *withdraw* and *deposit* transactions, which are themselves composed of read and write transactions

Benefits:

- Nested transactions can run concurrently with other transactions at same level in hierarchy
- If lower levels abort, may not need to abort whole transaction. Can instead use other means of recovery.

10.6 Summary

- Transactions provide technique for managing stateful servers
- Need to worry about concurrency control
- Need to worry about aspects of distribution
- Need to worry about recovery from failure