Distributed Systems
COMP 212

Lecture 21

Othon Michail

UNIVERSITY OF
LIVERPOOL

Transactions

Recap

- A distributed system is a collection of independent computers that appears to its users as a single coherent system
- To do so, they have to cooperate
- As part of this, they have to synchronise
  - Synchronise the clock
  - Synchronise concurrent actions

Synchronisation

- When a server uses multiple threads it can perform several client operations concurrently
- Clients share resources via the server
  - read/write at a database maintained at the server
  - modify files
- Synchronisation at server
- A Solution: Transactions

Transactions

1. Protect a shared resource against simultaneous access by concurrent processes
   - This can be also achieved by mutual exclusion algorithms
2. Allow a process to access and modify multiple data in a single atomic operation
   - Benefit: when half-success is not acceptable, everything can be restored as it never occurred

Example

- 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

Synchronisation at Server (1)

- When a server uses multiple threads it can perform several client operations concurrently
- If we allowed deposit and withdraw to run concurrently we could get inconsistent results
- Objects should be designed for safe concurrent access, e.g., in Java use synchronised methods, e.g.
  - *public synchronized void deposit(int amount) throws RemoteException*
- Atomic operations are free from interference with concurrent operations in other threads
- Use any available mutual exclusion mechanism

Synchronisation at Server (2)

- Clients share resources via a server
  - e.g. some clients update server objects and others access them
- In some applications, clients depend on one another to progress
  - e.g. one is a producer and another a consumer
  - e.g. one sets a lock and the other waits for it to be released
- There is a need to hide from other processes the internals of the process that has currently access to the shared resource
  - Each client is operating in this case on the shared resource as if it only performed a single indivisible step

More Problems: Failures

- *writes* to permanent storage may fail
  - e.g. by writing nothing or a wrong value (write to wrong block is a disaster)
- Servers may crash occasionally
  - when a crashed server is replaced by a new process its memory is cleared and then it carries out a recovery procedure to get its objects' state
- There may be an arbitrary delay before a message arrives
- A message may be lost, duplicated or corrupted
  - recipient can detect corrupt messages (by checksum)
  - forged messages and undetected corrupt messages are disasters

Transactions

- Some applications require a sequence of client requests to a server to be atomic in the sense that:
  - they are free from interference by operations being performed on behalf of other concurrent clients; and
  - either all of the operations must be completed successfully or they must have no effect at all in the presence of server crashes

- Such sequences are known as transactions

Banking Transaction

Transaction T:
a.withdraw(100);
b.deposit(100);
c.withdraw(200);
b.deposit(200);

- This transaction specifies a sequence of related operations involving bank accounts named *A*, *B* and *C* and referred to as *a, b* and *c* in the program
  - The first two operations transfer £100 from *A* to B
  - The second two operations transfer £200 from *C* to *B*

Transaction Life Histories

Successful	*Aborted by client*	*Aborted by server*
BEGIN_TRANSACTION	*BEGIN_TRANSACTION*	*BEGIN_TRANSACTION*
operation	*operation*	*operation*
operation	*operation*	*operation*
⋮	⋮	server aborts transaction → ⋮
operation	*operation*	*operation ERROR reported to client*
COMMIT_TRANSACTION	*ABORT_TRANSACTION*	

- A transaction is either successful (it commits)
  - all objects are saved in permanent storage
- …or it is aborted by the client or the server
  - make all temporary effects invisible to other transactions

The Transaction Model

Primitive	Description
BEGIN_TRANSACTION	Mark the start of a transaction
END_TRANSACTION	Terminate the transaction and try to commit
ABORT_TRANSACTION	Kill the transaction and restore the old values
READ	Read data from a file, a table, or otherwise
WRITE	Write data to a file, a table, or otherwise

- Example primitives for transactions
  - Provided by the underlying DS or by the language runtime system

Transactions: Not Only Databases!

```
BEGIN_TRANSACTION
  reserve Liverpool -> Paris;
  reserve Paris -> Athens;
  reserve Athens -> Rhodes;
END_TRANSACTION
```

(a)

```
BEGIN_TRANSACTION
  reserve Liverpool -> Paris;
  reserve Paris -> Athens;
   Athens -> Rhodes full =>
ABORT_TRANSACTION
```

(b)

a) Transaction to reserve three flights commits

b) Transaction aborts when the third flight is unavailable

ACID

- The four key transaction characteristics

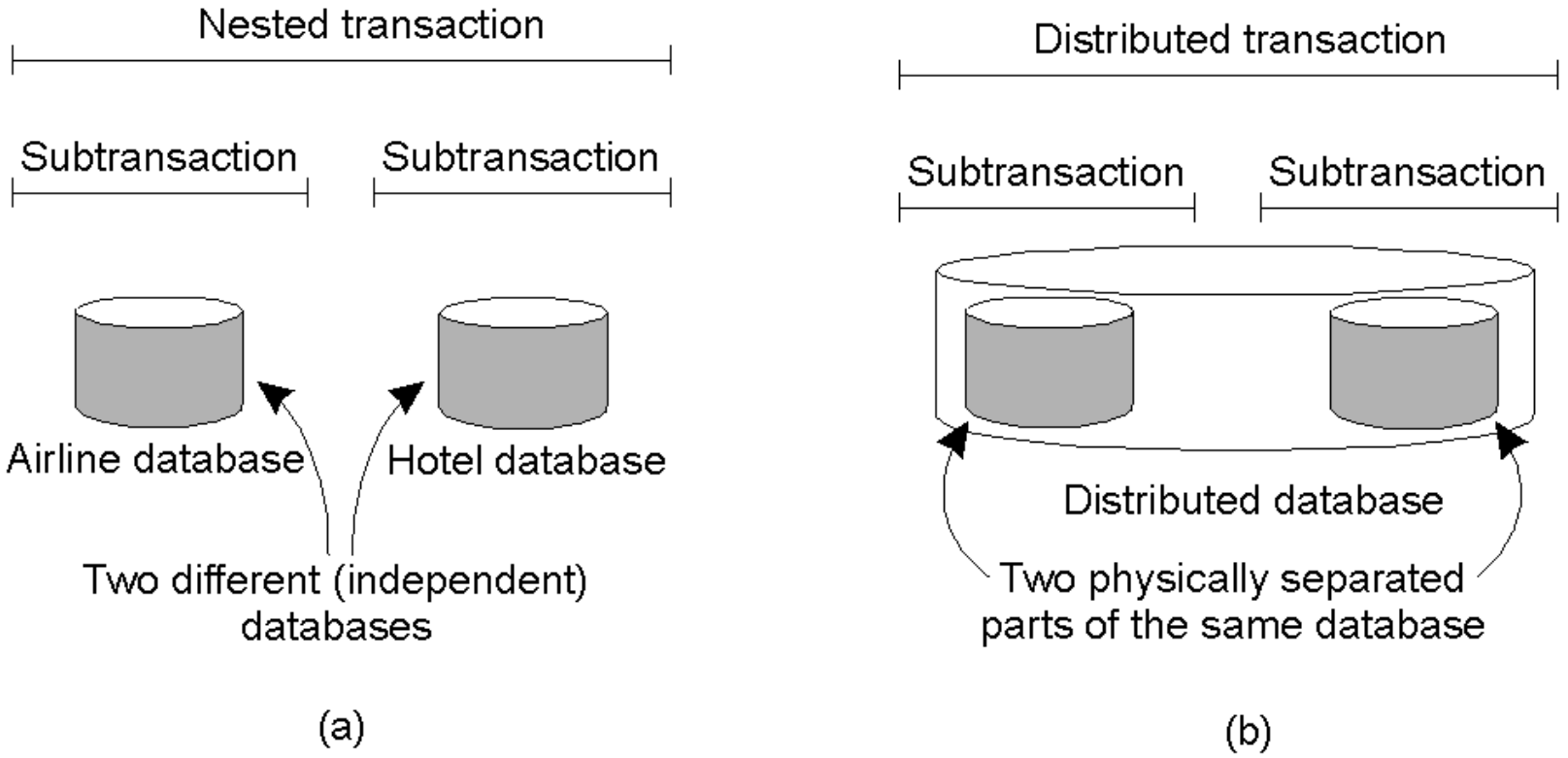
Transactions are:

- **Atomic**: The transaction is considered to be one thing, even though it may be made up of many different parts
- **Consistent**: "Invariants" that held before the transaction must also hold after its successful execution
- **Isolated**: If multiple transactions run at the same time, they must not interfere with each other. To the system, it should look like the two (or more) transactions are executed sequentially (i.e., that they are serializable).
- **Durable**: Once a transaction commits, any changes are permanent

Types of Transactions

- **Flat Transactions**: this is the model that we have looked at so far. Disadvantage: too rigid. Partial results cannot be committed. That is, the "atomic" nature of Flat Transactions can be a downside.
- **Nested Transactions**: a main, parent transaction spawns child sub-transactions to do the real work. Disadvantage: problems result when a sub-transaction commits and then the parent aborts the main transaction. Things get messy but still manageable.
- **Distributed Transactions**: these are sub-transactions operating on distributed data stores. Disadvantage: complex mechanisms required to lock the distributed data, as well as commit the entire transaction.

Nested vs. Distributed Transactions



a) A nested transaction – logically decomposed into a hierarchy of sub-transactions

b) A distributed transaction – logically a flat, indivisible transaction that operates on distributed data

Serialisability

```
BEGIN_TRANSACTION
  x = 0;
  x = x + 1;
END_TRANSACTION
```

(a)

```
BEGIN_TRANSACTION
  x = 0;
  x = x + 2;
END_TRANSACTION
```

(b)

```
BEGIN_TRANSACTION
  x = 0;
  x = x + 3;
END_TRANSACTION
```

(c)

e.g., if the three transactions are completed in the order (a) →(b) →(c), then x = 3

Schedule 1	x = 0;	x = x + 1;	x = 0;	x = x + 2;	x = 0;	x = x + 3	Legal
Schedule 2	x = 0;	x = 0;	x = x + 1;	x = x + 2;	x = 0;	x = x + 3;	Legal
Schedule 3	x = 0;	x = 0;	x = x + 1;	x = 0;	x = x + 2;	x = x + 3;	Illegal

(d)

a) – c) Three transactions T_1, T_2, and T_3

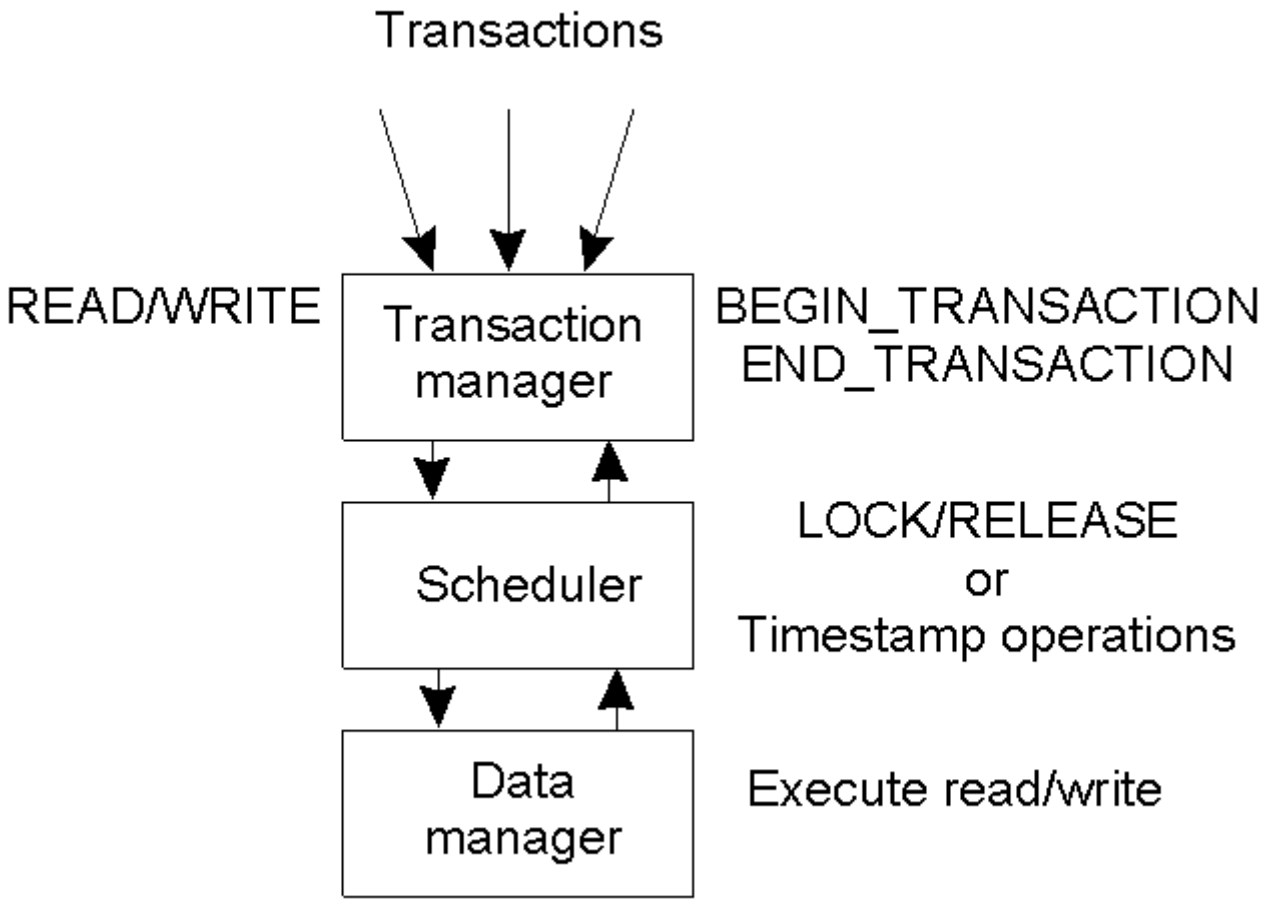
d) Possible schedules

Read and write conflicts

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

- Conflicting operations
  - A pair of operations conflicts if their combined effect depends on the order in which they were performed
  - Operate on the same data item and at least one of them is a *write*

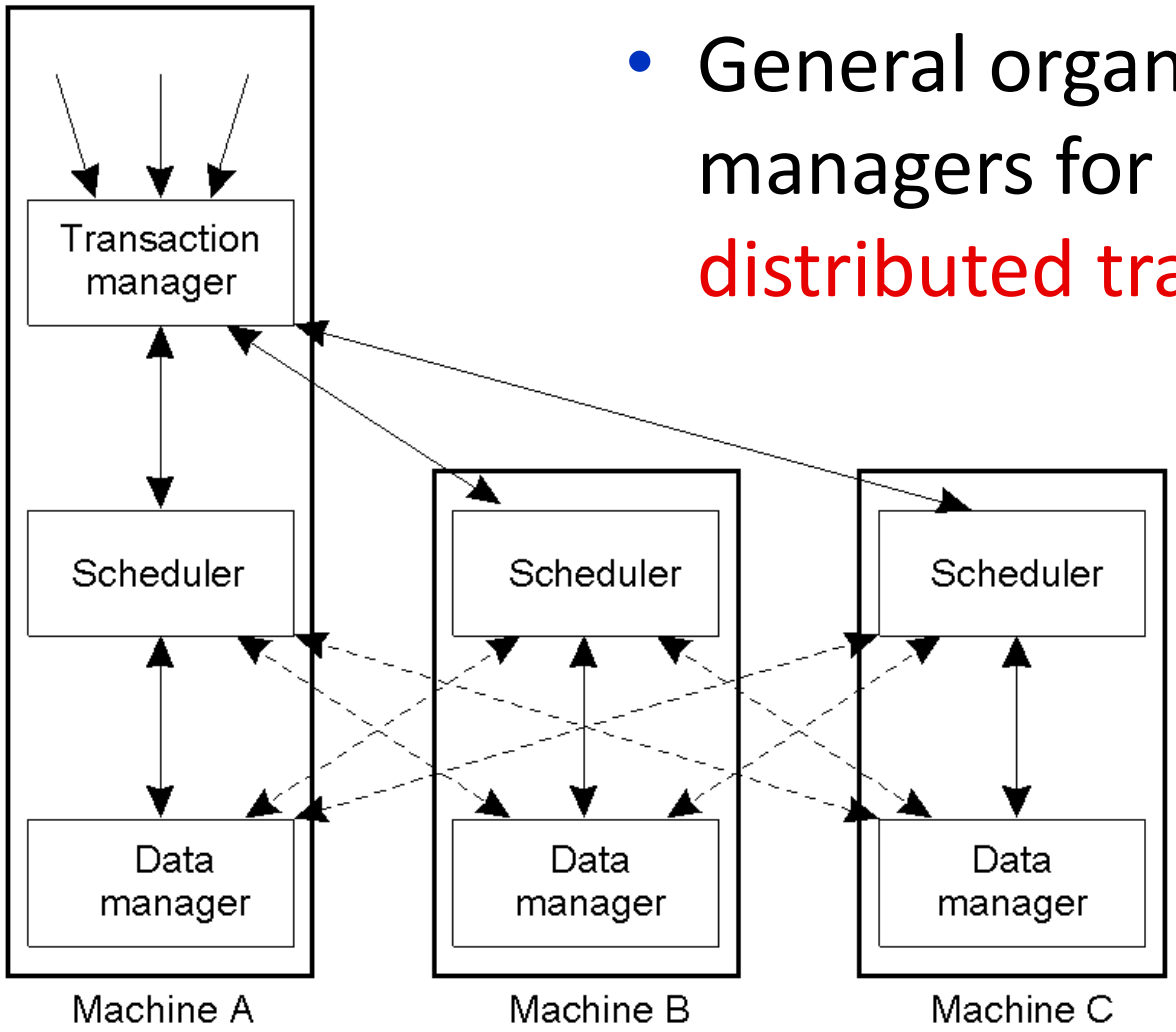
Concurrency Control (1)



- General organisation of managers for handling transactions

Concurrency Control (2)

- General organisation of managers for handling distributed transactions



How to undo actions

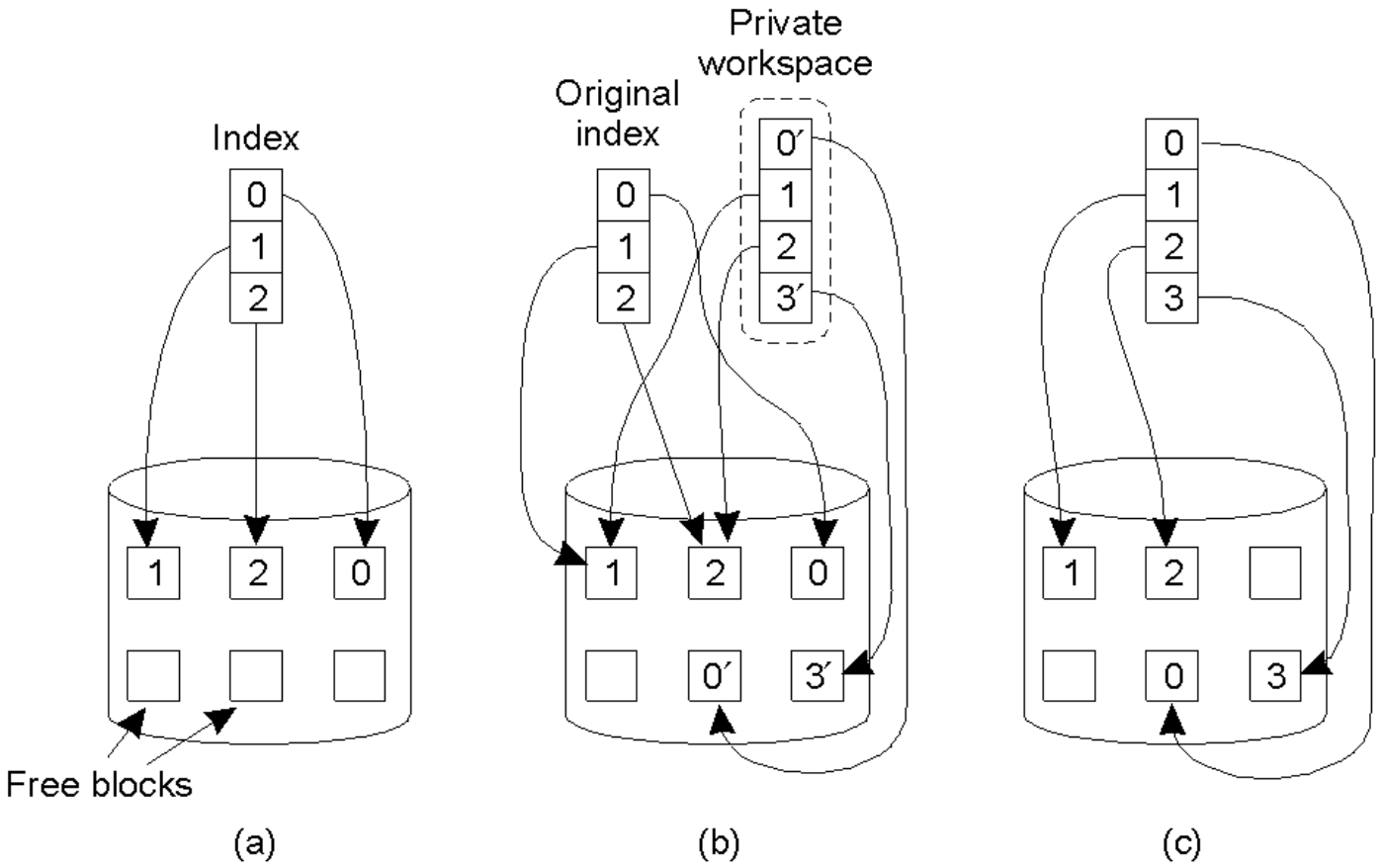
1. Private workspace
2. Writeahead log

Private Workspace (1)

- When a process starts a transaction it is given a private workspace containing all the files
- Until the transaction either commits or aborts, all of the reads and writes go to the private workspace

- Cost of copying everything!!!

Private Workspace (2)



a) The file index and disk blocks for a three-block file
b) The situation after a transaction has modified block 0 and appended block 3
c) After committing

Writeahead Log (1)

- Files are modified in place, but a record is written to a log prior to that
  - Only changes the file, after the log has been written successfully
- If the transaction aborts, the log can be used to rollback to the original state

- Problem with concurrent reads and writes

Writeahead Log (2)

x = 0;	Log	Log	Log
y = 0;			
BEGIN_TRANSACTION;			
x = x + 1;	[x = 0 / 1]	[x = 0 / 1]	[x = 0 / 1]
y = y + 2;		[y = 0 / 2]	[y = 0 / 2]
x = y * y;			[x = 1 / 4]
END_TRANSACTION;			
(a)	(b)	(c)	(d)

a) A transaction

b) – d) The log before each statement is executed