Transactions

- *Transaction* = atomic action spanning several concurrent objects
- Classic example: Transfer between bank accounts
- Transactions may be *interactive*
- Transaction may *fail* or be *given up* (abort).
- Often associated with persistent data objects (data bases)

Transaction Properties

Atomicity. All-or-nothing property.

Consistency. Should preserve system invariants.

Isolation. No interference (= atomicity).

Durability. Effect should remain.

Serializability

- A set of concurrently executed transactions are *serializable* if the operations on the underlying objects can be reordered such that:
  1. The sequence of operations for each transaction is preserverd
  2. The transactions follow each other in some sequence
  3. The effect on the object states remains the same

Conflics

- Two operations are in *conflict* if they cannot always be swapped
- Operations on different objects do not conflict
- Single object conflicts may be characterized semantically, eg.

	Read	*Write*	*Incr*
Read		×	×
Write	×	×	×
Incr	×	×	

Transaction Management

- $tid = StartTransaction()$
  $\vdots$
  $obj.op(\ldots)_{tid}$
  $\vdots$
  $res = EndTransaction(tid)$ $\quad$ res : Commit, Abort
- If transaction may abort, operations must be *undone*

Pessimistic Solutions

- *Global lock* at transaction level
- Locking of *individual objects*, conflict locking, successive locking
- 2-phase locking: Lock until commit/abort decided, then unlock.

Optimistic Solutions

- Idea: Go ahead and validate transaction before commitment
- Many different techniques.

MySQL Transactions

Storage Engine MyISAM

- Efficient, but only supports explicit locking at table level
- Example:

```
LOCK TABLE A READ, B WRITE, C WRITE
   Use of tables A, B, and C
UNLOCK TABLES
```

Storage Engine InnoDB

- Supports transactions by row-level locking
- Example:

```
START TRANSACTION
    SELECT ...  FROM ...  WHERE ...
    ⋮
    UPDATE ...  SET ...
COMMIT
```

- Optionally *weaker isolation levels* may be used

Distributed Transaction Management

- Objects handled by several distributed servers
- Roles:
  - *Client*: Defines transaction extent and contents
  - *Coordinator*: Controls the outcome of transaction
  - *Participants*: Distributed objects being operated upon
- Client creates transaction and issues operations to participants
- Overall *decision* whether to commit or abort must be made
- Standard approach: *Two-phase commit*

Two-phase Commit Protocol

- Solution to distributed commit/abort decision [Gray 78].
- 1. The client requests the coordinator to *start* a new transaction
  2. Operations are executed in within a *transaction context*
  3. Participants *register* with the coordinator
  4. The client requests the coordinator to *end* the transaction
  5. The coordinator asks participants to *prepare* for commitment
  6. The participants respond with commit/abort *votes*
  7. The coordinator informs all about commit/about decision
  8. Partipants store changes persistently or discards them

Issues of Two-phase Commit Protocol

Robustness

- Decision protocol should tolerate common failures (network errors and node chrashes)
- Example: Use of *persistent logs* to record intermediate changes
- Consensus problem **unsolvable** for arbitrary failures [Fischer 85]

Global Deadlocks

- Participant may use *locking* for internal atomicity
- Each participant can detect only *local deadlocks*
- Solutions:
  - Use *timeouts* to *abort* blocked transitions
  - Apply *distritubed deadlock detection*

Two-phase Commit Protocol

Java

- *Java Transaction API (JTA)* and *Java Transaction Service (JTS)*
- Use mostly implicit within *J2EE application server*

CORBA

- *CORBA Transaction Service*

Web Services

- *WS Coordination* framework for general coordination
- *WS Atomic Transaction* for ACID-transactions
- *WS Business Activity* for "open" transactions
- Open transactions may use *compensation* to undo changes