Overview

Distributed transactions
- multiple servers
- atomicity

Atomic commit protocols
- 2-phase commit

Concurrency control
- locking
- timestamping
- optimistic concurrency control

Other issues (deadlocks, recovery)

Transactions

Definition
- sequence of server operations
- originate from databases (banking, airline reservation, etc)
- atomic operations or sequences (free from interference by other clients and server crashes)
- durable (when completed, saved in permanent storage)

Issues in transaction processing
- need to maximise concurrency while ensuring consistency
  - serial equivalence/serializability (= same effect as a serial execution)
- must be recoverable from failures

Distributed transactions

Definition
- access objects which are managed by multiple servers
- can be flat or nested

Sources of difficulties
- all servers must agree to commit or abort
  - two-phase commit protocol
- concurrency control in a distributed environment
  - locking, timestamps
  - optimistic concurrency control
- failures!
  - deadlocks, recovery from aborted transactions

Transaction handling

Requires coordinator server, with open/close/abort
Start new transaction (returns unique TID)
  openTransaction() -> trans;
Then invoke operations on recoverable objects
  A.withdraw(100);
  B.deposit(300)
If all goes well end transaction (commit or abort)
  closeTransaction(trans) -> (commit, abort);
Otherwise
  abortTransaction(trans);

Distributed transactions

Flat structure:
- client makes requests to more than one server
- request completed before going on to next
- sequential access to objects

Nested structure:
- arranged in levels: top level can open sub-transactions
- any depth of nesting
- objects in different servers can be invoked in parallel
- better performance
**Distributed transactions**

(a) Flat transaction

Client

\[ \text{openTransaction()} \]

requirement

\[ T \]

(b) Nested transactions

Client

\[ \text{openTransaction()} \]

requirement

\[ T \]

\[ \text{commit} \]

\[ \text{abort} \]

1. \[ T_1 \]

2. \[ T_2 \]

E.g. \[ T_{11}, T_{12} \] can run in parallel

**How it works...**

Client

- issues \( \text{openTransaction()} \) to coordinator in any server
- coordinator executes it and returns unique TID to client
  - TID = server IP address + unique transaction ID

Servers

- communicate with each other
- keep track of who is who
- coordinator: responsible for commit/abort at the end
- participant: can \( \text{join} \) (Trans, RefToParticipant)
  - manages object accessed in transaction
  - keeps track of recoverable objects
  - cooperates with coordinator

**Distributed flat banking transaction**

**One-phase commit**

Distributed transactions

- multiple servers, must either be committed or aborted

One-phase commit

- coordinator communicates commit/abort to participants
- keeps repeating the request until all acknowledged

But… server cannot abort part of a transaction:

- when the server crashed and has been replaced...
- when deadlock has been detected and resolved...

Problem

- when part aborted, the whole transaction may have to be aborted

**Two-phase commit**

Phase 1 (voting phase)

1. coordinator sends \( \text{canCommit?} \) to participants
2. participant replies with vote (Yes or No); before voting Yes prepares to commit by saving objects in permanent storage, and if No aborts

Phase 2 (completion according to outcome of vote)

3. coordinator collects votes (including own)
   - if no failures and all Yes, sends \( \text{doCommit} \) to participants
   - otherwise, sends \( \text{doAbort} \) to participants
4. participants that voted Yes wait for \( \text{doCommit} \) or \( \text{doAbort} \) and act accordingly; confirm their action to coordinator by sending \( \text{haveCommitted} \)

**Communication in 2-phase protocol**

**Coordination and Commit**

Coordinator

<table>
<thead>
<tr>
<th>Event</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ready</td>
<td>ready</td>
</tr>
<tr>
<td>prepare</td>
<td>prepare</td>
</tr>
<tr>
<td>commit</td>
<td>commit</td>
</tr>
</tbody>
</table>

Participant

<table>
<thead>
<tr>
<th>Event</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>prepare</td>
<td>prepare</td>
</tr>
<tr>
<td>commit</td>
<td>commit</td>
</tr>
<tr>
<td>haveCommitted</td>
<td>haveCommitted</td>
</tr>
</tbody>
</table>
What can go wrong...

In distributed systems
- Objects stored/managed at different servers
- Server crashes
  - Participant: save in permanent storage when preparing to commit, retrieve data after crash
  - Coordinator: delay till replaced, or cooperative approach
- Messages fail to arrive (server crash or link failure)
  - Use timeout for each step that may block (but no reliable failure detector, asynchronous communication)
  - If uncertain, participant prompts coordinator by \texttt{getDecision}
  - If in doubt (e.g., initial \texttt{canCommit}? or votes missing), abort!

Nested transactions

Top-level transaction
- Starts subtransactions with unique TID (extension of the parent TID)
- Subtransaction joins parent transaction
- Completes when all subtransactions have completed
- Can commit even if one of its subtransactions aborted...

Subtransactions
- Can be independent (e.g., act on different bank accounts)
- Can execute in parallel, at different servers
- Can provisionally commit or abort
- If parent aborts, must abort too

Nested banking transaction

Transaction T decides to commit

Hierarchic two-phase commit

Multi-level nested protocol
- Coordinator of top-level transaction is coordinator
- Coordinator sends \texttt{canCommit?} to coordinator of subtransactions one level down the tree
- Propagate to next level down the tree, etc
- Aborted subtransactions ignored
- Participants collect replies from children before replying
  - If any provisionally committed subtransaction found, prepares the object and votes \texttt{Yes}
  - If none found, assume must have crashed and vote \texttt{No}
- Second phase (completion using \texttt{doCommit})
  - Same as before
Concurrency control

Needed at each server
- To ensure consistency

In distributed systems
- Consistency needed across multiple servers

Methods
- Locking
  - Processes run at different servers can lock objects
- Timestamping
  - Global unique timestamps
- Optimistic concurrency control
  - Validate transaction at multiple servers before committing

Locking

Locks
- Control availability of objects
- Lock manager held at the same server as objects
- To acquire lock: contact server
- To release: must delay until transactions commit/abort

Issues
- Locks acquired independently
- Cyclic dependencies may arise
  - T: Locks A for writing; U: Locks B for writing;
  - T: Wants to read B - must wait; U: Wants to read A - must wait;
  - Distributed deadlock detection and resolution needed

Timestamp ordering

If a single server...
- Coordinator issues unique timestamp to each transaction
- Versions of objects committed in timestamp order
- Ensures serializability

In distributed transactions
- Coordinator issues globally unique timestamps to the client opening transaction:
  - <local timestamp, server ID>
- Synchronous clocks sometimes used for efficiency
- Objects committed in global timestamp order
- Conflicts resolved, or else abort

Optimistic concurrency control

If a single server...
- Alternative to locking (avoids overhead and deadlocks)
- Transactions allowed to proceed but validated before allowed to commit: if conflict arises may be aborted
  - Transactions given numbers at the start of validation
  - Serialised according to this order

In distributed transactions
- Must be validated by multiple independent servers (in the first phase of two-phase commit protocol)
- Global validation needed (serialise across servers)
- Parallel also possible

Other issues

Distributed deadlocks!
- Often unavoidable, since cannot predict dependencies and server crashes possible
- Use deadlock detection, priorities, etc

Recovery
- Must ensure all of committed transactions and none of the aborted transactions recorded in permanent storage
- Use logging, recovery files, shadowing, etc

See textbook for more info

Summary

Transactions
- Crucial to the running of large distributed systems
- Atomic, durable, serializable
- Order of updates important
- Require two-phase commit protocol

Distributed transactions
- Run on multiple servers
- Can be flat or nested
- Hierarchical two-phase commit
- Concurrency control adapted to distributed environment