7. Transaction Management for distributed databases

Distributed Transactions
7. Transaction Management for distributed databases

28 Distributed Transactions

29 Distributed Commit
7. Transaction Management for distributed databases

28 Distributed Transactions

29 Distributed Commit

30 Distributed Synchronization
7. Transaction Management for distributed databases

28 Distributed Transactions

29 Distributed Commit

30 Distributed Synchronization

31 Distributed Deadlocks
7. Transaction Management for distributed databases

Distributed Transactions

Distributed Commit

Distributed Synchronization

Distributed Deadlocks

Transaction Monitors
Distributed Transactions

- In distributed DBS, transactions across several nodes
- Commit as an atomic event → Simultaneous in distributed nodes
- Distributed synchronization in order to guarantee consistency in interleaved executions
- Deadlock detection
Requirements for distributed commit

- Commit protocol: Guarantee for atomicity and durability
- Requirements for distributed cases
  - All nodes make a decision (**Commit**, **Abort**); globally, all nodes make the same decision
  - **Commit** only if all nodes vote “yes“
  - If no failure occurs and all nodes vote “yes“ \(\implies\) global decision is **commit**
  - All processes terminate
Two-phase commit protocol

- Roles: 1 coordinator, several participants
- Execution:
  1. **Voting phase**
     1. Coordinator asks participants whether \texttt{commit} can be performed
     2. Participants signal their decision to coordinator
  2. **Decision phase**
     1. Coordinator makes a decision based on participants’ signals (all \texttt{commit} $\rightarrow$ \texttt{Global-Commit}; one \texttt{Abort} $\rightarrow$ \texttt{Global-Abort}
     2. Participants that voted “yes“ wait for decision
2PC: Execution scheme

- Coordinator:
  - Initial
  - Write `begin_commit in Log`
  - Wait
  - Write `commit in Log`
  - Commit
  - Write `EOT in Log`

- Participants:
  - Initial
  - Write `commit in log`
  - Ready
  - Write `ready in log`
  - Vote-Commit
  - Write `commit in Log`
  - Commit
  - Write `EOT in Log`
  - Vote-Abort
  - Write `abort in Log`
  - Abort
  - Global-Abort
  - Write `abort in Log`
  - Unilateral Abort
  - Write `commit in log`
  - Commit
  - Write `EOT in Log`

- Decisions:
  - All committed?
  - Ready to commit?
2PC: State transition

(a) coordinator

1. **INITIAL**
2. **WAIT**
   - Wait for vote-commit
   - Vote-Abort
   - Global-Abort
3. **ABORT**
4. **COMMIT**

(b) participants

1. **INITIAL**
2. **READY**
   - Prepare
   - Vote-Abort
   - Global-Abort
   - ACK
3. **ABORT**
4. **COMMIT**

**Commit command**
- Prepare
- Vote-Abort
- Vote-Commit
- Global-Abort
- ACK

**Global-commit**
- Global-Commit
- ACK
2PC: Problems I

1st Phase

(1) (2)

2nd Phase

v-c: Vote-Commit
p: Prepare-To_Commit
g-c: Global-Commit
2PC: Problems II

- Participants signaled **Vote-Commit** but coordinator fails (1)
  - **Abort** of participants after timeout
  - But: Undo of a made decision!

- After sending **Global-Commit** (to an unknown number of participants) has been sent, coordinator and participant 1 fail (2)
  - Who sends **Global-Commit**? Or **Abort**?
Variants of 2PC I

- **Linear 2PC: Coordinator as initiator**
  - Coordinator sends `Prepare-To-Commit` to participant 1
  - Participant 1 makes a decision and sends it to the next participant
  - `Vote-Abort` signal also to predecessor
  - Last participant receives `Vote-Commit` and votes “Yes“ → `Global-Commit` to predecessor

Disadvantage: Slow because of sequential processing
Linear 2PC: Execution schema
Variants of 2PC II

- Distributed 2PC: Local voting process
  - Coordinator sends *Prepare-To-Commit* to all participants
  - Write decision into Log and forward to all participants
  - Every participant receives all results and makes a local decision
  - Disadvantage: A lot of communication is required
  - Advantage: Quick answers because of missing phase 2

- Hierarchical 2PC: Coordinators and sub-coordinators
Distributed 2PC: Execution schema

- Coordinator
- Participants

- Prepare
- Vote-Abort
- Vote-Commit
- Global-Commit
- Global-Abort
Hierarchical 2PC: Execution schema

P = Prepare  R = Ready  C = Commit
3-Phase Commit Protocol

- Problems of 2PC: Failure of coordinators before participants receive \texttt{Global-Commit} / \texttt{Global-Abort}
- Solution: 3PC with additional \texttt{PRE-COMMIT}-phase
  - Participants that receive \texttt{Prepare-To-Commit} know that \texttt{Commit} will arrive only if coordinator does not fail
  - Coordinator sends \texttt{Commit}, only after $k$ participants confirm the \texttt{Prepare-To-Commit} with a \texttt{Ready-To-Commit}
3PC: Phases I

1. **Voting phase**
   1. Coordinator sends *Prepare* signal
   2. Every participant answers and signals its decision (*Vote-Commit* or *Vote-Abort*)
   3. In case of *Vote-Abort*, directly into state *ABORT*

2. **Decision preparation phase**
   1. Coordinator collects decisions; in case of *Vote-Commit*, a *Prepare-To-Commit* is sent to all; otherwise *Global-Abort*
   2. Every participant with *Vote-Commit* waits for *Prepare-To-Commit* and confirms with *Ready-To-Commit*; otherwise *Global-Abort*
3PC Phases II

3  Decision Phase

1  Coordinator collects all confirmations and makes a decision
2  Participants wait for decision
3PC: Execution schema

- Coordinator:
  - INITIAL
  - WAIT
  - PRE-COMMIT
  - COMMIT
  - ABORT

- Participants:
  - INITIAL
  - READY
  - PRE-COMMIT
  - COMMIT

- Flow:
  - Write begin_commit in log
  - Write abort in log
  - Write precommit in log
  - Write commit in log
  - Write EOT in log
  - Vote-Commit
  - Vote-Abort
  - Unilateral Abort
  - Global-Abort
  - Global-Commit
  - Ready-to-commit
  - ReadyKto-commit
  - All committed?
  - Prepare-To-Commit
  - Prepare
  - Commit in log
  - Ready-to-commit
  - ACK
  - Precommit in log
  - Global decision?
3PC: State Transition

(a) coordinator

(b) participants
3PC: Errors

Failure of coordinator and up to $k - 1$ further participants

1. All further participants in state READY
   - Failed participants can only be in states READY, ABORT or PRE-COMMIT $\rightarrow$ Abort of transaction

2. One participant in state PRE-COMMIT or COMMIT
   - Becomes new coordinator and continues protocol
   - decision was already made for commit
Distributed Synchronization

- Local synchronization not sufficient
- Example: $T_1$, $T_2$

<table>
<thead>
<tr>
<th>Node 1</th>
<th>Node 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$</td>
<td>$T_1$</td>
</tr>
<tr>
<td>$T_2$</td>
<td>$T_2$</td>
</tr>
<tr>
<td>$r_1(x)$</td>
<td>$r_2(y)$</td>
</tr>
<tr>
<td>$w_1(x)$</td>
<td>$w_2(y)$</td>
</tr>
<tr>
<td>$r_2(x)$</td>
<td></td>
</tr>
<tr>
<td>$w_2(x)$</td>
<td></td>
</tr>
<tr>
<td>$r_1(y)$</td>
<td></td>
</tr>
<tr>
<td>$w_1(y)$</td>
<td></td>
</tr>
</tbody>
</table>
Distributed Synchronization: II

- Resulting schedule not conflict serializable; however no local synchronization conflict

Solution:
- Distributed timestamp-ordering method
  - Total order on timestamps \((\text{Time, Node-ID})\)
  - Requires global clock and distributed clock synchronization
- Central or distributed locking methods
- ...
Distributed timestamp-ordering method

Global timestamp $ts$ as two-tuple
(local timestamp $ts_l$, $hid$ is Host-ID):

$$ts = (ts_l, hid)$$

Order determination:
- Order according to $ts_l$ value
- In case of same $ts_l$ value, decision according to $hid$
Synchronization of local clocks

- **Usage of global clock:**
  Requires regular synchronization of local clocks → often not acceptable

- **Usage of radio-controlled clock**

- **Distributed clock synchronization**
  Synchronization during communication, later time is adopted → (unique time)
## Distributed timestamp allocation (via counter)

<table>
<thead>
<tr>
<th>Point in time</th>
<th>Node 1</th>
<th>Node 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Local TA timestamps</td>
<td>local TA timestamps</td>
</tr>
<tr>
<td>1</td>
<td>$T_1$</td>
<td>$T_1$</td>
</tr>
<tr>
<td>2</td>
<td>$T_2$</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>$T_3$</td>
<td>$T_2$</td>
</tr>
<tr>
<td>5</td>
<td>$T_4$</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>$T_5$</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>$T_3$</td>
</tr>
<tr>
<td>9</td>
<td>$T_6$</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>$T_4$</td>
</tr>
<tr>
<td>11</td>
<td>$T_7$</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td>$T_5$</td>
</tr>
<tr>
<td>13</td>
<td>$T_8$</td>
<td></td>
</tr>
</tbody>
</table>
Transactions on replicas I

A schedule $s$ on a replicated database is 1-copy serializable if there is a serial schedule on a non-replicable database that has the same effect as $s$ on a replicable data set.
Transactions on replicas II

Replication protocol

- **ROWA-Method** (Read One, Write All): Local read and synchronized updates of all replicas
  → extremely high complexity; some computer nodes might be unavailable

- **ROWAA-Method** (Read One, Write All Available)

- **Voting procedure**: voting procedure or quorum procedure
  - Statistical number of “eligible voters“
  - Dynamical number of “eligible voters“ is depending on environmental influences such as lost connections and access behavior

(Weighting of votes is possible)
Transactions on replicas III

Replication protocol

- *Absolutistic approaches*: e.g. primary copy method: A certain node updates a replica in any case. Choice is static or the node has a *token*
Distributed locking methods

- **Centralized 2PL (C2PL):** Central management of locks on a node → requires a lot of communication, heavy load for central lock manager (replication protocol needs to be observed)

- **Primary copy 2PL (PC2PL):** Several lock managers on different nodes; each DB-object has exactly one lock manager → Distribution of lock managing load

- **Distributed 2PL (D2PL):** Lock manager on every DBMS; lock manager is responsible for its own DB-objects (no replication → PC2PL, otherwise ROWA)
Distributed Deadlocks

Classes of deadlock handling:

- **Deadlock-free**
  Preclaiming (C2PL) – atomic requirement can hardly be fulfilled in distributed cases

- **Deadlock prevention**
  Total order of objects and their occupancy

- **Deadlock detection**
  Detection of distributed deadlocks is problematic
Deadlock detection

- **Time-Out-Mechanism**
- **Global deadlock graph**
  Central coordinator manages conflict graph (coordinator could fail!)

![Diagram of distributed transactions and deadlock detection](image-url)
Deadlock handling

Practical methods

- **Conservative locking**
  C2PL-method: Problems with atomicity

- **Timestamps as requirement order**
  Timestamps for handling lock conflicts

- **Deadlock detection**
  - *Centralized*: Central vertex manages complete wait graph
  - *Hierarchical deadlock detection*: Many deadlocks can be identified locally; difficult implementation though
  - *Distributed deadlock detection*
Distributed detection of global deadlocks I

Deadlocks do not exist locally in any computer node. Computers send each other messages of the form $[m, n, k]$.

1. $m \equiv$ Number of blocked process
2. $n \equiv$ Number of transaction that sent the message (sender)
3. $k \equiv$ Number of transaction to which the message is directed (receiver)
Distributed identification of global deadlocks: II

- Message transmission starts with $[0, 0, 1]$.
- Message $[0, 2, 3]$ means that the blocked transaction is transaction 0, sender is transaction 2 and receiver is transaction 3.
- A deadlock occurs if and only if the message arrives at the blocked process.
Distributed identification of global deadlocks: Example

The transaction \( m \) waits for transaction \( n \) or the transaction \( m \) is blocked by transaction \( n \)
Transaction monitors

- Presentation Server
- Presentation Server
- Workflow Controller
- Transaction Server
- Transaction Server
Transaction monitors: Architecture

- Presentation server, acts as client and realizes communication with user (command language or menu-driven interfaces for sending transactions etc.)
- Workflow controller forces routing of transaction requirements of different DBMS and realizes, for instance, two-phase commit protocol
- Transaction server realizes connection of local DBMS with transaction monitor
Advantages of a transaction monitor

- Offers *one* standardized interface for programming transactions on different DBMS
- In distributed processing, it manages routing of transactions and forces commit protocols
- Offers systems functions such as load balancing, error control, and system configuration
- Is able to fulfill functions such as writing log files or monitoring communication
- Transaction server of a TP-monitor can also encapsulate data that is not managed by a DBMS with full transaction functionality