4. Distributed Transactions
Outline

- Transactions – Fundamental Concepts
- Remote Database Access
- Distributed Transactions
- Transaction Monitor
OLTP (Online Transaction Processing)

(Distributed) Transaction processing:

- Reliable data processing, also during system failures and multi-user access.

Application examples:
- Reservation systems (Booking)
- Bank transactions
- Accounting
- Generally: data access in write mode

Implementation via Transaction Processing Systems / Monitors, since 1960; System examples:
- JDBC (pure database access interface)
- TX Series (IBM)
- Tuxedo (Oracle)
Transactions implement the **ACID**-criteria:

**Atomicity**
- Execution complete or without impacts

**Consistency**
- Transformation between consistent states

**Isolation**
- No overlapping of concurrent transaction executions

**Durability**
- Survival of system failures due to persistent storage
Transactions: Remote Database Access

Two and Three Tier Architecture

Application

```
beginTransaction()  
Actions on Database  
commit()  
```

Database Driver for Remote Access

DBMS-specific Protocol

DBMS

Client

```
DBMS-independent Communication
```

Server

```
beginTransaction()  
Actions on Database  
commit()  
```

Database Driver for Remote Access

DBMS-specific Protocol

DBMS: Database Management System
Transactions: Remote Database Access

JDBC (Java Database Connectivity)

- Programming interface to access relational data bases
- Corresponds to ODBC (Open Database Connectivity)
- Database operations in Structured Query Language (SQL)
- Numerous drivers for different data bases (for instance, Oracle, Sybase, DB2, SQL Server etc.)
- Only realization of direct data base access
  → improved distributed transaction logic in heterogeneous systems
    requires Transaction Services / Transaction Monitors
try {
  Warehouse.con.setAutoCommit(false);  // Update the warehouse inventory
  PreparedStatement pstmt = Warehouse.con.prepareStatement("UPDATE
      Warehouse SET Count=Count-? WHERE ProductID = ?");
  pstmt.setInt(1, countOrdered);
pstmt.setInt(2, productNumber);
  int updated = pstmt.executeUpdate();
pstmt.close();  // Adding of the product to the dispatch list. Number of items =
  countOrdered.
...
  // Check product availability
  PreparedStatement pstmt = Warehouse.con.prepareStatement("SELECT Count FROM Warehouse
      WHERE ProductID = ?");
  pstmt.setInt(1, productNumber);
  ResultSet resultSet = pstmt.executeQuery();
  if(resultSet.first())  {
    int count = resultSet.getInt("Count");  // Available products
      if (count >= 0) Warehouse.con.commit();
    Warehouse.con.rollback();
  ... }
  catch (SQLException se) {  Warehouse.con.rollback();
}
Ensuring ACID properties (Atomicity, Consistency, Isolation and Durability):

- Either both Warehouse Management and the Dispatch list are updated or non at all
- Sum of products in the Dispatch List must be consistent with that in Warehouse Management
- Modifications to number of products are not allowed to become visible for other operations outside of the transaction until Warehouse Management and Dispatch are both updated and transaction successfully closed.
Two Phase Commit Protocol (2PC)

- Necessary for coordination between the various participants involved in a distributed transaction

- Assumptions
  - Participants save relevant data before transaction so that the original consistent state can be reset if required
  - Transaction started: All required operations carried out but only on temporary copies of the data.
  - Operations can be carried out through RPC and RMI as well as through various local mechanisms on the individual participants

- Initiation of the protocol
  - Coordinator (a client or an interconnected server) initiates the Two Phase Commit Protocol, after all temporary changes are made
Distributed Transactions: Coordination

Two Phase Commit Protocol (2PC)

Coordinator (for instance, Client)

(1a) prepare
(1b) ready / abort
(2a) commit
(2b) acknowledge

Participant (for instance, Server 1)

old Version: K
save Data: K'

K := K'

Participant (for instance, Server 2)

Old Version: L
save Data: L'

L := L'
Distributed Transactions: Coordination

Two Phase Commit Protocol

Input from the network / Output to the network

- / prepare
prepare / abort
prepare / ready
commit / acknowledge
abort / -

start
wait
aborted
committed
aborted
committed
ready

a) Coordinator
b) Participant

Prof. Dr. Alexander Schill
Distributed Systems – Lecture 4: Distributed Transactions
Two Phase Commit Protocol

- **1st Phase**
  - **Prepare** messages sent by coordinator to participants
  - Persistent copy of data with changes made and previous version is saved → Participants can use previous or new version in case of crash/restart
  - **Ready** message sent by participants after they have successfully saved the new version of the data
  - **Abort** sent, if saving the data was not successful

- **2nd Phase**
  - **Ready** received from all participants?
    - **Commit** sent by coordinator to all participants
    - Old version of data replaced by new version
    - Participants send confirmation to coordinator
    - **Transaction complete**
  - **Abort** received from one (or more) participant?
    - Coordinator encounters a failure
    - **Abort** sent to all participants
    - New version of the data is rejected and the old is reinstated
    - **Transaction aborted**
Special problem:
- One participant saves data successfully, returns “ready” message, but then crashes for some reason (power failure, out of memory, local error); all other participants are ok.
- Coordinator will assume everything is all right (“ready” messages from all participants) and will send “commit” to all participants.
- All participants except the crashed one will actually perform the commit operation, so a rollback is not possible any more.

Solution:
- Crashed participant will eventually restart and will find a persistent transaction log, together with the old and the new version of the affected data.
- Participant has to ask the former coordinator (or another participant, if information on that is available) how the outcome of the transaction was – “commit” or “abort”?
- It will finally take appropriate action (roll forward to new data or roll back to old data), so everything is consistent again; locks on data can only be released thereafter (-> so called blocking problem of distributed transactions).
Concurrency Control – isolation of transactions to ensure:

- Operations maintain consistent views of data
- Intermediate results withheld until conclusion of transaction

Pessimistic Approach (assumptions):
- Frequent conflicts
- High expense for undoing transactions

Optimistic Approach (assumptions):
- Conflicts are rare
- Undoing and retrying a transaction is less expensive than the blocking of transactions (through locking) and the analysis of Deadlocks
Pessimistic Approach:
- Locking → data is reserved exclusively for a particular transaction
- Other transactions must wait until release of locks to access same data
- Issues:
  - Avoiding Deadlocks – two transactions requiring access to each others locked data
  - Achieving higher concurrency
- Algorithms to ensure transactions executed sequentially – e.g. Two Phase Locking

a) **Simple** Two Phase Locking - lock released as soon as access finished – higher concurrency

b) **Strict** Two Phase Locking - locks only released at end of transaction - lower concurrency

- Extension required for distributed systems
  - Central lock manager - bottlenecks
  - Distributed lock management – local databases handle setting and releasing of local locks
Optimistic Approach:

- **Work Phase**
  - Data access without locks
  - Read-Set – data objects involved in transaction
  - Write-Set – changes made to the Read-Set

- **Validation Phase**
  - Check if Read-Set contains inconsistent data – i.e. if it has been altered by other transactions since the data was read in the Work Phase
  - Read-Set inconsistent? → current transaction is cancelled and its changes are revoked

- **Securing Phase**
  - Write-Set is committed to database
  - Takes place in the distributed case through a coordinator based on the Two Phase Commit Protocol
Problem of Simple Transactions:
- Success of many individual operations within the transaction are lost with the resetting of the whole transaction
- Lack of parallelism

Nested Transactions - Properties:
- Parallel partial transactions through separate threads (editing time reduced)
- Separate backtracking of partial transactions (Atomicity maintained through Transaction Contexts – record of modified data)
- Selective repeats
- Abort of all partial transactions during abort of the general transaction (⇒ Data versions to be kept until end)
- Inheritance of locks in both directions within the hierarchy
Distributed Transactions: Nested Transactions

Example

- Initial failures in Payment partial transactions do not affect Dispatch List Creation and Dispatch transactions
  - E.g. incorrect credit card details → retry with another payment method

- If Payment transaction continues to fail then all other partial transactions under Order Execution root transaction will have to be reset
Definition

- Middleware for supporting distributed transactions
- Realisation of 2PC protocols
- Driver software for linking different databases (resource managers)
- Administration tools

Implementations: e.g. IBM, Oracle, SAP, Microsoft

Possible Operations:

- Local transactions
- Remote object calls
- Parallelization of remote transactions through threads
- Call of other Transaction Monitors
- Nested transactions
Distributed Transaction Processing (DTP) Model

- Standardized (by Open Group) collaboration between transaction monitors, applications and different database systems.
- Software Architecture for the realization of distributed transaction systems.
- Definition of components and interfaces
Transaction Monitor: DTP Model

- **Resource Manager (RM)**
  - Represents resources involved within a distributed transaction e.g. databases, print servers etc.
  - Executes local transactions on these resources
  - Coordinates conclusion of such transactions

- **Application Programme (AP)**
  - Executes operations on various resources involved within a distributed transaction
  - Defines begin and end of transactions
  - Decides whether a transaction is concluded with a “commit” or “rollback”

- **Transaction Monitor (TM)**
  - Manages execution and coordination of transaction conclusion with various Resource Managers
  - Achieved on the basis of the 2 Phase-Commit-Protocol
  - Executes commit and reset operations

- **Communication Resource Manager (CRM)**
  - Represents services for the communication between components in different management domains of transaction monitors
  - Mediates calls and associated call data between application components
**TX interface**: between AP and TM

- **tx_open, tx_close**: connect to transaction manager
- **tx_begin, tx_rollback, tx_commit**: demarcate transaction

**XA interface**: between TM and RM

- **xa_open, xa_close**: initialization before any other call, termination after all other calls
- **xa_start, xa_end**: associate subsequent calls with a transaction branch
- **xa_prepare, xa_commit, xa_rollback**: classical two phase commit protocol
- **ax_reg**: Register an RM with a TM.
- **ax_unreg**: Unregister an RM with a TM.
Step A) Initialization:

1) Shop application is started
2) Shop initiates connection to Transaction Monitor: `tx_open(rm1, rm2)`
3) Transaction Monitor establishes connection to Resource Managers (of Warehouse & Shipping): `xa_open(…)`
4) Resource Managers respond with `xa_ok()`
5) Transaction Monitor informs shop about successful initiation: `ACK`
Step B) Transaction

1) Client starts transaction, e.g. buys a book
2) Shop starts transaction at TM: `tx_begin()` -> transaction id created
3) TM initializes transaction with RMs: `xa_start()`, RMs confirm: `ACK`
4) TM confirms successful initiation
5) Shop asks to remove book from stock: `requestProduct(pid) → return pid`
6) Shop adds book to the shipping list: `addProduct(pid) → return true`
7) Shop tells TM, whether transaction was successful -> `tx_commit, tx_rollback`
Step C) Commit (Two-Phase-Commit-Protocol)

1) TM prepares commit & sends \texttt{xa\_prepare()} to the RMs
2) RMs respond with \texttt{xa\_ready()}/\texttt{xa\_abort()}
3) TM sends \texttt{xa\_commit()}/\texttt{xa\_rollback()}, RMs answer with ACK
4) TM finishes transaction: \texttt{xa\_end()}, RMs answer with ACK
5) TM sends ACK to Shop
6) Shop sends confirmation of book order
Step D) Termination

1) Shop checks out from TM: `tx_close()`
2) TM closes connection to RMs: `xa_close()`, they answer with ACK
3) TM confirms check out: ACK
Example: Integration with IBM SNA legacy system

- **Idea**: Integration of old, but stable systems (e.g. IBM Systems Network Architecture (SNA)) into distributed transactions via gateways/protocol translation
- **Gateway** enables two-phase-commit interface across the different systems, bridging between modern installations (Internet protocols, Linux servers and languages like Java and C++) with older systems (SNA and legacy languages like Cobol)
Transaction Monitor

Component Based Transaction Control

- Separation of properties from application functionality
- Properties (i.e. specifics of how transactions should be carried out) defined at deployment time
- E.g. Transaction properties supported by Enterprise Java Beans:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOT_SUPPORTED</td>
<td>Execution of methods of a Bean within a transaction is not supported (if applicable – temporary suspension of transaction)</td>
</tr>
<tr>
<td>SUPPORTS</td>
<td>Use of a Bean is possible within and without transaction context</td>
</tr>
<tr>
<td>REQUIRED</td>
<td>Transaction mandatory; if applicable – implicit starting of a new transaction (in case a transaction is not active)</td>
</tr>
<tr>
<td>REQUIRES_NEW</td>
<td>Transaction mandatory, always started with the method call of a Bean (if applicable – temporary suspension of a presently running transaction)</td>
</tr>
<tr>
<td>MANDATORY</td>
<td>Transaction mandatory, must exist beforehand (otherwise exception message)</td>
</tr>
<tr>
<td>NEVER</td>
<td>Bean not allowed to be used within a transaction</td>
</tr>
</tbody>
</table>
Summary: Transactions

- Transaction Monitors are essential components for commercial applications
- Transaction Monitors provide greater flexibility than simple JDBC applications
- Consistent, distributed data management
- Optional nested transactions with partial rollback and more concurrency
- Integration with mainframe systems
- Broad product support

