Distributed Systems

Distributed Transactions

Tenshi Hara
tenshi.hara@tu-dresden.de
https://www.rn.inf.tu-dresden.de/hara
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;
  System examples:
  - JDBC (pure database access interface)
  - CICS - Customer Information Control System (IBM)
  - Encina (Transarc / IBM)
  - Tuxedo (BEA Systems)
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
Two and Three Tier Architecture

Application

beginTransaction() →
Actions on Database
commit() →

Database Driver for Remote Access

DBMS-specific Protocol

DBMS

Client

DBMS-independent Communication

beginTransaction() →
Actions on Database
commit() →

Database Driver for Remote Access

DBMS-specific Protocol

DBMS

Server
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 heterogenic systems requires Transaction Services/ Monitors
try {
    // Update the warehouse inventory
    Warehouse.con.setAutoCommit(false);
    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
pstmt = Warehouse.con.prepareStatement("SELECT Count
    FROM Warehouse
    WHERE ProductID = ?
");
pstmt.setInt(1, productNumber);
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

- 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 Protocol, after all temporary changes are made
Two Phase Commit Protocol

Coordinator (e.g., Client)

(1a) prepare

(1b) ready / abort

(2) commit

Participant (e.g., Server 1)

old Version: K

save Data: K'

K := K'

Participant (e.g., Server 2)

Old Version: L

save Data: L'

L := L'
Two Phase Commit Protocol

Notation: *Input from / Output to* network

- Start
- Wait
- Aborted
- Committed
- Prepare
- Ready

a) Coordinator
b) Participant
Two Phase Commit Protocol

1st Phase (PREPARE Phase)

- **Prepare** messages sent by coordinator to all participants

  - all participants create persistent copies of data expected to change and simulate all instructed operations on the copies
  - previous version for rollbacks
  - new version for commit or restart

- all participants must reply to the coordinator
  - **Ready** message sent by participants after they have successfully created the data copy
  - **Abort** message sent, if saving the data was not successful
    - due to active locks on data (ACID criteria!)
    - due to unfulfillable dependencies
  - no reply covered by timeout mechanism
Two Phase Commit Protocol

2nd Phase (COMMIT Phase)

- **Ready** messages received from all participants?
  - **Commit** messages sent by coordinator to all participants
  - previously copied new version of data becomes active data set (by simply redirecting an active pointer)
  - participants send Done confirmation to coordinator
  - transaction complete

- **Abort** received from one (or more) participant or timeout occurred?
  - coordinator encounters a failure
  - **Abort** messages sent to all participants
  - new version of the data is rejected and the old is reinstated (by unlinking the copy’s pointer; active pointer remains untouched)
  - Transaction aborted
### Two Phase Commit Protocol

<table>
<thead>
<tr>
<th>Message</th>
<th>sent by</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Phase 1</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BEGIN</td>
<td>coordinator</td>
<td>request readiness to receive instructions</td>
</tr>
<tr>
<td>PREPARE</td>
<td>coordinator</td>
<td>includes the operation(s) to be conducted as well as necessary variables</td>
</tr>
<tr>
<td>READY</td>
<td>participant</td>
<td>confirmation of successful operation(s) simulation</td>
</tr>
<tr>
<td>ABORT (also: FAILED)</td>
<td>participant</td>
<td>notification on errors that occurred while simulation</td>
</tr>
<tr>
<td><strong>Phase 2</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>COMMIT</td>
<td>coordinator</td>
<td>instruction to make changes persistent (redirect active pointer onto modified data copy)</td>
</tr>
<tr>
<td>COMMIT-ACK (also: DONE)</td>
<td>participant</td>
<td>confirmation of successful persistation</td>
</tr>
<tr>
<td>ABORT (also: ROLLBACK)</td>
<td>coordinator</td>
<td>instruction to discard the modified copies and retain the original data set</td>
</tr>
<tr>
<td>ABORT-ACK (also: ROLLBACK-ACK)</td>
<td>participant</td>
<td>confirmation of the abort/rollback instruction</td>
</tr>
<tr>
<td>RESTART</td>
<td>either</td>
<td>notification on crashes and other local errors</td>
</tr>
</tbody>
</table>
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 await release of locks to access same data
- issues:
  - avoiding deadlocks (a.k.a. mutex situations)
    (two transactions requiring access to each other's locked data)
  - achieving higher concurrency
- algorithms to ensure transactions executed sequentially;
  e.g., Two Phase Locking
- extensions required for distributed systems
  - central lock manager - bottlenecks
  - distributed lock management – local databases handle setting and releasing of local locks
**Simple** Two Phase Locking:
lock released as soon as access finished – higher concurrency

**Strict** Two Phase Locking
locks only released at conclusion of transaction - lower concurrency
## 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? \(\rightarrow\) 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 on the basis of the 2 Phase Commit Protocol

<table>
<thead>
<tr>
<th>Transaction Start</th>
<th>Transaction End</th>
</tr>
</thead>
<tbody>
<tr>
<td>Work Phase</td>
<td>Validation Phase</td>
</tr>
<tr>
<td></td>
<td>Securing Phase</td>
</tr>
</tbody>
</table>

Distributed Systems – Distributed Transactions
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 through transaction contexts; requires record of modified data)
- selective repeats
- abort of all partial transactions during abort of the general transaction (data versions must be kept until end)
- inheritance of locks in both directions within the hierarchy
initial failures in Payment (i.e., partial transactions) neither affect Dispatch List Creation nor Dispatch transactions (e.g., incorrect credit card details → try with other payment method)

if Payment transaction continues to fail then all other partial transactions under Order Execution 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

exemplary implementations: IBM, Oracle, SAP, Microsoft

Possible Operations:
- remote object calls
- parallelisation of remote transactions through threads
- Call of other transaction monitors
- nested transactions
- local transactions
**Distributed Transaction Processing (DTP) Model**

- standardised (by Open Group) collaboration between transaction monitors, applications and different database systems
- software architecture for the realisation of distributed transaction systems
- definition of components and interfaces

---

**Diagram:**
- **Application Programme (AP)**
  - **Resources Manager (RM)**
  - **Transaction Monitor (TM)**
  - **Communication Resources Manager (CRM)**
  - **OSI TP**
  - **CPI-C, XATM, TxRPC**
  - **TX**
  - **XA**
  - **XA+**
  - **XAP-TP**

---

**OSI-TP:**
*Open Systems Interconnection Transaction Processing (Protocol)*
Resource Manager (RM)

- represents resources involved within a distributed transaction; e.g., databases, print servers
- 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 Two 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 TM
- `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`: deregister an RM with a TM
Step A) Initialization:

1) Shop application is started
2) Shop initiates connection to TM: `tx_open(rm1, rm2)`
3) TM establishes connection to RMs (of Warehouse and Shipping): `xa_open(...)`
4) RM respond with `xa_ok()`
5) TM 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` or `tx_rollback`
Step C) Commit (2-Phase-Commit-Protocol)

1) TM prepares commit and sends `xa_prepare()` to the RMs
2) RMs respond with `xa_ready()`/`xa_abort()`
3) TM sends `xa_commit()`/`xa_rollback()`, RMs answer with ACK
4) TM finishes transaction: `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 CICS legacy system

- idea: integration of old stable systems (e.g., CICS with SNA) into distributed transactions via gateways/protocol translation
- transaction frontend as CICS region (from point of view of the mainframe)
- access to mainframe data base possible

Basis:
- SNA-gateway; but today also TCP/IP in mainframe environment available
- COBOL-interface to transactional backend data systems
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 through EJB 3.0:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOT_SUPPORTED</td>
<td>Execution of a Bean’s methods 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 a Bean’s method call (if applicable – temporary suspend 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
- trusted, consistent, distributed data management
- optional nested transactions, loading balancing, security aspects
- integration with mainframe systems (for instance, CICS-Monitor)
- broad product support
References