Information System Infrastructure II

Transaction Processing

Hans-Arno Jacobsen
jacobsen@eecg.toronto.edu
www.eecg.toronto.edu/~jacobsen

Transaction processing and transaction processing monitors

- A server or network crash in the middle of a sequence of operations!
- what should happen?
- how can a sequence of operations be grouped?
- what models exist?
- how is this done in networked environments, with multiple servers involved in one such sequence?
Transaction Examples

- Extracting money from ATM
- Booking a flight
- Booking a trip, flight, rental car, hotel
- Changing a flight - across multiple airlines
- Debiting / crediting accounts, e.g., double-booking systems
- Online shopping, e.g., buying books

A transaction

- What is transaction?
  - An execution of a unit-of-work that accesses one or more shared resources, usually database(s).
  - A unit of work is a set of activities that relate to each other and must be completed together.
- The following operations can be seen as one transaction.

| Transaction A | start:                     |
|              | operation 1: Get available flights |
|              | operation 2: Book a flight       |
|              | operation 3: Get available rooms |
|              | operation 4: Book a room         |

Transaction A end:
The ACID Transaction Model

- Atomicity
  - guarantees that many operations are bundled together and appear as one contiguous unit of work
  - all or nothing, i.e., if transaction is interrupted, all its work is undone
- Consistency
  - guarantees that a transaction will leave the system in a consistent state (i.e., all integrity constraints satisfied) after transaction completes
  - the transaction preserves all system properties
- Isolation:
  - protects concurrently executing transactions from seeing each other’s incomplete results
  - all transactions think they are executing serially, even if they are running concurrently
- Durability:
  - guarantees that updates to managed resources, such as database records, survive hardware and software failures.
  - The results of a completed transaction are permanent and will never be lost

Atomicity

- Atomicity prevents the following from happening
  - What if the flight cannot be booked?
  - What if the room booking system is down right after the flight is booked?
- Atomicity prevents a system from exposing only partially executed work to a user.
- The successful completion of a transaction is called commit.
- The failure of a transaction is called abort.
Consistency

- In database terms (informally speaking), consistency means that the data in the database is kept in a “correct” state (i.e., a consistent state.)
- That is all integrity constraints are fulfilled.
- For example:
  - Assume that a customer cancels the trip. The destination city should be removed from the list of cities in his trip plan.
  - An attached car rental in the destination city should be deleted.
- It is often a shared responsibility between application and the database to maintain consistency.

Isolation

- Suppose there is only one seat available for a certain flight.
- Two customers are trying to book the trip as described in Transaction A.
- In booking system:
  without isolation                      with isolation
  A: Get flights (one seat available)    A: Get flights (one seat available)
  B: Get flights (one seat available)    A: Book flights (issued a ticket)
  A: Book flights (issue ticket)         B: Get flights (no seat)
  B: Book flights (issue ticket)         B: Abort
  customer B is issued an                correct result
  non-existing ticket
Durability

- Durability handles the situation where system fails right after the execution of the transaction is completed and before the result is written to permanent storage.

Without durability
1. Transaction A start
2. Transaction B Commit
3. Disk fails
   (user already got the "success" message)
   (Result is not yet in the database)

With durability
1. Log every operation in a file
2. Transaction A start
3. Transaction B commit
4. Disk fails
5. check the transaction log
6. Recover from disk failure
7. Check log.
   Write the result to database
8. Transaction succeeds.

Example using Java Transaction API (JTA).

- A booking system provides methods for booking rooms and flights. It is accessed by a client object.
Implementing a Transaction in Java

public boolean BookTrip(){
    UserTransaction transaction = context.getUserTransaction();
    try{
        transaction.begin();
        if(bookingSystem.CheckFlight())
            bookingSystem.BookFlight();
        if(bookingSystem.CheckRoom())
            bookingSystem.BookRoom();
        transaction.end();
    } catch(Exception ex){
        try{
            transaction.rollback();
        } catch(SystemException s){
            throw new Exception('Rollback failed');
        }
    }
    return true; }

Distributed Transaction Processing

- In distributed environments, transactions often update data in more than one system.
- The processing of those transactions are thus distributed among involved systems.
- Processing a flight reservation with multiple legs serviced by different airlines is an example of a distributed transaction.
- While each system ensures durability of its share of the transaction processing, the atomicity of the entire operation needs to be ensured as well.
- That is done through the two-phase commit protocol.
The above entities are involved in the 2PC transaction model (X/Open).
The transaction manager processes Start, Commit and Abort.
It talks to resource managers and other transaction managers to run two-phase commit.
Two Phase Commit Protocol

- Two phase commit protocol is initiated by a special module, called transaction manager.
- Any participating database system can act as the transaction manager.
- In a "typical" system the two phase commit has two distinct processes that are accomplished in less than a fraction of a second: (http://www.sei.cmu.edu/str/descriptions/dtpc.html)

  - The Prepare Phase, where the global coordinator (initiating database) requests that all participants (distributed databases) will promise to commit or rollback the transaction. Note: Any database could serve as the global coordinator, depending on the transaction.

2. The Commit Phase, where all participants respond to the coordinator that they are prepared, then the coordinator asks all nodes to commit the transaction. If all participants cannot prepare or there is a system component failure, the coordinator asks all databases to roll back the transaction.
Why is 2pc (two phase commit) important?

- Suppose at the end of the transaction, we need to update the systems in both New York and London with the results.

New York System
- Update X
- Commit

London System
- Update Y
- System Fails
- Write X
- Commit
- System Recovers

The system lost the update to Y when it can’t commit the transaction after recovery.

New York System
- Update X
- System Fails
- Write Y
- Commit
- System Recovers

London System
- Update Y
- System Fails
- Write Y

Since the system saved the update to disk before failing, it can commit the transaction after it recovers.

a. Without two phased commit, the failure caused the update to Y to be lost.

b. With two-phase commit, the London system durably saved the update to Y, so it can commit after it recovers.

Transaction Manager
1. prepare

2. prepared

Resource Manager in New York

Resource Manager in London

Transaction Manager
1. prepare

2. prepared

Resource Manager in New York

Resource Manager in London

3. commit

4. ok

phase one

phase two
Object Transaction Services

The following slides have been adapted from slides by Ian Gorton, Paul Greenfield Software Arhitectures and Component Technologies, CSIRO, Sydney.

Role Definitions

- Transactional Client (TC)
- Transactional Objects (TO)
- Transaction Coordinator
- Non-transactional Objects
- Recoverable Objects and Resource Objects
- Transactional Servers
- Recoverable Servers
Coordinator

- Coordinator plays key role in managing transaction.
- Coordinator is the component that handles begin / commit / abort transaction calls.
- Coordinator allocates system-wide unique transaction identifier.
- Different transactions may have different coordinators.

Transactional Server

- Every component with a resource accessed or modified under transaction control.
- Transactional server has to know coordinator.
- Transactional server registers its participation in a transaction with the coordinator.
- Transactional server has to implement a transaction protocol (two-phase commit).
Transactional Client

- Only sees transactions through the transaction coordinator.
- Invokes services from the coordinator to begin, commit and abort transactions.
- Implementation of transactions are transparent for the client.
- Cannot tell difference between server and transactional server.

Transaction models

- Flat transactions
  - (transaction service must support these)
  - a top-level transaction and cannot have a child transaction
- Nested transactions
  - (transaction service need not implement)
  - provide for a finer granularity of recovery
  - transaction family
    - parent, child, sibling, ancestor, descendant
Flat Transactions

Begin Trans.  

Commit  

Flat Transaction

Begin Trans.  

Crash  

Begin Trans.  

Abort  

Flat Transaction

Nested Transactions

Begin Trans.  

Commit  

Main Transaction

Begin Trans.  

Commit  

Child Transaction

Begin Trans.  

Commit  

Child Transaction

Begin Trans.  

Commit  

Child Transaction

(c) 2002 Hans-Arne Jacobsen (jacobsen@eceg.toronto.edu)  MIE491f
Transaction Context

- part of the environment of each ORB-aware thread
- can be null or refers to a specific transaction
- can be implicitly or explicitly transmitted to transactional objects
- allow programmer to pass as an explicit parameter of a request
- contains unique transaction id (tid)

Basic architecture

![Diagram of transaction context and basic architecture](image-url)
OTS Interfaces (CosTransactions)

- Current Interface
- TransactionFactory Interface
- Control Interface
- Terminal Interface
- Coordinator Interface
- Recovery Coordinator Interface
- Resource Interface
- Synchronization Interface
- Subtransaction Aware Resource Interface
- TransactionalObject Interface
Current Interface

```cpp
interface Current : CORBA::Current
{
    void begin();
    raises (SubtransactionsUnavailable);
    void commit (in boolean report_heuristics)
        raises (NotTransaction, HeuristicMixed, HeuristicsHazard);
    void rollback()
        raises (NoTransaction);
    void rollback_only()
        raises (NoTransaction);
    Status get_status();
    string get_transaction_name();
    void set_timeout (in unsigned long seconds);
    Control get_control();
    Control suspend();
    void resume (in Control which)
        raises (InvalidControl);
};
```

Control Interface

```cpp
interface Control
{
    Terminator get_terminator()
        raises (Unavailable);
    Coordinator get_coordinator()
        raises (Unavailable);
};
```
Coordinator Interface

```java
interface Coordinator
{
    Status get_status();
    Status get_parent_status();
    Status get_top_level_status();
    boolean is_same_transaction (in Coordinator tc);
    boolean is_related_transaction (in Coordinator tc);
    boolean is_ancestor_transaction (in Coordinator tc);
    boolean is_descendant_transaction (in Coordinator tc);
    boolean is_top_level_transaction();
    unsigned long hash_transaction();
    unsigned long hash_top_level_transaction();
    RecoveryCoordinator register_resource (in Resource r)
        raises (Inactive);
    void register_synchronization (in Synchronization sync)
        raises (Inactive, SynchronizationUnavailable);
};
```

Terminator Interface

```java
interface Terminator
{
    void commit (in boolean report_heuristics)
        raises (HeuristicMixed, HeuristicHazard);
    void rollback ();
};
```
Resource Interface

```java
interface Resource {
    Vote prepare();
    raises (HeuristicMixed, HeuristicHazard);
    void rollback();
    raises (HeuristicCommit, HeuristicMixed, HeuristicHazard);
    void commit();
    raises (NotPrepared, HeuristicRollback, HeuristicMixed, HeuristicHazard);
    void commit_one_phase();
    raises (HeuristicHazard);
    void forget();
}
```

IDL sample

```cpp
//IDL sample

interface Stock: CosTransactions::TransactionObject {
    exception RejectOrder {
        string Reason;
    }
    exception DBNotAvailable {};
    attribute long inStock;
    readonly attribute string StockName;

    void Order ( in long amount ) raises (RejectOrder);
    void Query ( out long amount ) raises (DBNotAvailable);
}

interface Store: CosTransactions::TransactionObject {

    Stock GetStock (in long stock_id) raises (NoPart);
    void DeleteStock (in Stock p);
}
```

12/1/02
Code sample

```c++
int main (...) {
    Stock_var pStock;
    try {
        // bind to the Stock object
        pStock = Stock::_bind (Server, host);
    } catch (CORBA::SystemException ex) {
        cout << "Error Binding!" << ex << endl;
        Encina::Client::Exit(1);
    }
    // rest of client code
    ...
    return 0
}
```

12/1/02

Transaction Example

```c++
try {
    CosTransactions::Current::begin();
    Stock_var StockObj = pStore->GetStock(id);
    StockObj.Order(100);
    CosTransactions::Current::commit();
} catch (CORBA:: TRANSACTION_ROLLBACK) {
    CosTransactions::Current::rollback();
    cout << "TransactionRollback exception, transaction aborted" << endl;
    commit = FALSE;
} catch (CORBA:: UserException ex) {
    CosTransactions::Current::rollback();
    commit = FALSE;
    cout << "TransactionRollback by " << ex << endl;
} catch (...) {
    CoTransaction::Current::rollback();
    cout << "Unknown exception, transaction abortd" << endl;
    commit = FALSE
}
```

12/1/02
Recovery in Two-Phase Commit

- Failures prior to start of 2PC results in abort.
- Coordinator failure prior to transmitting commit messages results in abort.
- After this point, co-ordinator will retransmit all Commit messages on restart.
- If server fails prior to voting, it aborts.
- If it fails after voting, it sends GetDecision.
- If it fails after committing it (re)sends HaveCommitted message.

Committing Nested Transactions

- Cannot use same mechanism to commit nested transactions as:
  - sub-transactions can abort independently of parent.
  - sub-transactions must have made decision to commit or abort before parent transaction.
- top-level transaction needs to be able to communicate its decision down to all sub-transactions so they may react accordingly.
Provisional Commit

- sub-transactions vote either:
  - aborted or
  - provisionally committed.
- Abort is handled as normal.
- Provisional commit means that coordinator and transactional servers are willing to commit sub-transaction but have not yet done so.

Locking and Provisional Commits

- Locks cannot be released after provisional commit.
- Data items remain 'protected' until top-level transaction commits.
- This may reduce concurrency.
- Interactions between sibling sub-transactions:
  - should they be prevented (different)?
  - allowed (part of the same transaction)?
- Generally they are prevented.
Key Points

- A distributed object transaction is an atomic, consistency-preserving, isolated durable sequence of object requests
- Objects participating in transactions can be transactional clients, transactional servers and transaction coordinators
- Isolation is achieved by two-phase locking that can either be delegated to a database or be done explicitly by the server designer

Key Points

- Atomicity is achieved by two-phase commit, which consists of a voting and a completion phase
- Object-oriented middleware supports distributed transaction through transaction services
  - CORBA Transaction Service
  - Microsoft Transaction Server
  - Java Transaction Service