Middleware and Distributed Systems

Transactions

Martin v. Löwis
Terminology

• Financial Transaction (purchase, loan, mortgage, ...)

• Database Transaction: unit of interaction between a process and a relational database

• Atomic transaction: sequence of operations that should be atomic
  
  • not necessarily limited to databases - may involve regular files, or actions "in the real world"

  • all-or-nothing: should either completely succeed or completely fail

    • failure atomicity: should be atomic even in the presence of crashes

    • durability: changes should persist once transaction succeeds

  • isolation: concurrent transactions must not interfere
ACID

- Härder and Reuter, Principles of Transaction-Oriented Database Recovery, Computing Surveys, 1983

- Atomicity: updates are all-or-nothing

- Consistency: integrity is maintained across transactions

- Isolation: intermediate states are not observable to other processes

- Durability: changes are not undone after a transaction completes

- Recovery: system reverts to previous state in case of failure

- Concurrency: allow concurrent operations even though they possibly might have conflicting effects
  - server needs to verify that actions are *serializable*
Distributed Transactions

- client invokes operations on different servers
  - effects should be atomic across all servers
- flat vs. nested
  - flat: a client starts a transaction, then sequentially performs operations on multiple servers
  - nested: within a transaction, further transactions can be started; sub-transactions may run concurrently
Transaction Coordinator

• aka Transaction Manager aka Transaction Monitor

• allows identification of transaction, and keeps track of participants (resources) of a transaction

  • openTransaction: start a new transaction, returns transaction handle

  • closeTransaction: complete successfully

  • abortTransaction: discard all partial changes

  • join: include a reference to a participant (process) into the transaction

• client needs to communicate transaction handle to all participants

  • coordinator does not talk to participants during the transaction (only at the end)
\[ T = \text{openTransaction} \]
\[ a.\text{withdraw}(4); \]
\[ c.\text{deposit}(4); \]
\[ b.\text{withdraw}(3); \]
\[ d.\text{deposit}(3); \]
\[ \text{closeTransaction} \]

Note: the coordinator is in one of the servers, e.g. BranchX
Atomic Commit Protocols

• one-phase commit
  • server sends commit/abort messages to all participants
  • participant individually commits local changes
  • problem: what if a server fails to commit, e.g. when the server had to break a lock to resolve a deadlock with some other transaction

• two-phase commit (Gray 1978)
  • prepare phase: participants vote to commit or abort transactions
    • write prepared log entries, and enter uncertain (in-doubt) state
  • servers who voted to commit then must not change their minds
  • commit phase: participants all commit
Coordinator

<table>
<thead>
<tr>
<th>step</th>
<th>status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>prepared to commit (waiting for votes)</td>
</tr>
<tr>
<td>2</td>
<td>committed</td>
</tr>
<tr>
<td>3</td>
<td>committed</td>
</tr>
<tr>
<td>4</td>
<td>committed</td>
</tr>
</tbody>
</table>

Participant

<table>
<thead>
<tr>
<th>step</th>
<th>status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>canCommit?</td>
</tr>
<tr>
<td>2</td>
<td>prepared to commit (uncertain)</td>
</tr>
<tr>
<td>3</td>
<td>doCommit</td>
</tr>
<tr>
<td>4</td>
<td>haveCommitted</td>
</tr>
<tr>
<td>5</td>
<td>committed</td>
</tr>
</tbody>
</table>
Failures

- time-outs, server crashes, message loss
- server crash: server gets restarted from consistent state
  - information about ongoing transactions might have been lost, so prepare messages from coordinator result in aborts
  - in a crash after a "commit" vote, server needs to recover with prepare log
  - coordinator may abort transaction after participant timeout
Failures (2)

• coordinator should maintain log of ongoing transactions, redo log during recovery
  • for started transactions without completed prepare phase: abort
  • if no vote was recorded for some participant: ask again
  • if abort was logged: redo abort
  • if commit was logged: redo commit

• coordinator crash: participant needs to find out global state, by asking restarted coordinator
  • before prepare: can safely abort transaction
  • after prepare (*uncertainty period*): need to wait for coordinator, or try to find other participants
Correctness of 2PC

• Safety: if one process is in a final (committed/aborted) state, then either all processes are in the committed state, or all processes are in the aborted state.

• Liveness: for a finite number of failures, 2PC will reach a final global state after a finite sequence of state transitions (i.e. messages sent).
Nested Transactions

• Additional operations on coordinator:
  • `openSubTransaction(trans)`: nested transaction ID must include/refer to parent transaction

• Transaction status may be committed, aborted, or provisional
  • provisional commit is not durable, and visible only within the outer transaction (sub-transaction joins parent transaction)
  • server may lose information about provisional commits in a crash

• Parent transaction can be committed even if sub-transactions failed
  • Application needs to take appropriate corrective measures (e.g. retry)
Nested Transactions: 2PC

• Hierarchic model: prepare calls are made recursively through the tree
  • intermediate nodes act as coordinators for their sub-transactions
  • entire transaction will abort if one participant aborts

• Flat model: top-level coordinator asks all coordinators of provisionally-committed transactions
  • if a parent transaction has already aborted, the sub-transaction must vote "abort": coordinator should send list of aborted transactions in prepare message
**XA**

- X/Open specification for distributed transactions (1991)
- Application Program (AP)
- Resource Manager (RM)
- Transaction Manager (TM)
- not meant for communication (i.e. unspecified wire protocol)
  - suggests to use OSI DTP
  - XA is interface between RM and TM
- unit of work: global transaction
  - transaction branches on individual RMs, identified by XIDs
xa.h

- C API to be used by the RM
- XA implementation provided by the TM vendor
- routines to be called by the RM: ax_reg, ax_unreg
- routines to be called by the TM (implemented by the RM, as function pointers):
  - xa_open, xa_close: initialisation
  - xa_start: create a new branch for the current thread, and associate it with given XID (or join current thread if XID was already started)
  - xa_end: dissociate current thread with XID
  - xa_prepare, xa_commit, xa_rollback: 2PC
XA Implementations

• integrated into Java Transaction API (JTA), through javax.transaction.xa.XAResource

• TM Implementations
  • IBM Customer Information and Control Service (CICS)
  • Oracle (ehemals Bea) Tuxedo
  • Microsoft Transaction Server (also: OLE transactions)

• RM Implementations
  • Oracle, DB/2, MySQL, Berkeley DB, ...
TX

• X/Open API for APs
• tx_begin, tx_rollback, tx_commit
• tx_info: returns XID
CORBA Transaction Service

- OMG document formal/03-09-02: Transaction Service Specification, version 1.4
- both local API, and wire protocol
- IDL interfaces:
  - Current
  - Control
  - TransactionFactory
  - Terminator
  - Coordinator
  - Resource
  - Synchronization
Current Interface

• gives access to current transaction, simplifies programming

• available as initial reference ("TransactionCurrent")
  • needs to be thread-local

• void begin() raises(SubtransactionsUnavailable);

• void commit(in bool report_heuristics)
  raises(NoTransaction, HeuristicsMixed, HeuristicsHazard);

• void rollback()...

• Control get_control();

• Control suspend(); void resume(in Control which)…;
TransactionFactory Interface

- implemented by TP monitor

- Control create(in unsigned long time_out); // seconds

- Control recreate(in PropagationContext ctx);
Control Interface

- Terminator get_terminator()...
- Coordinator get_coordinator()...
Terminator Interface

• void commit(in boolean report_heuristics);

• void rollback();
Coordinator Interface

- responsible for a single transaction
- access to status, transaction hierarchy
- creation of sub-transactions

RecoveryCoordinator register_resource(in Resource r)...

void register_synchronization(in Synchronization sync)...

PropagationContext get_txcontext(...
Resource Interface

- Vote prepare() raises{HeuristicsMixed, HeuristicsHazard};
  - VoteReadOnly: no modifications made
  - VoteCommit, VoteRollback
- void rollback();
- void commit();
- void commit_one_phase();
- void forget();
  - only used after heuristic outcomes
Synchronization Interface

• used to integrate transient state

• void before_completion();
  • invoked before the prepare step
  • object may start copying transient state to some resource

• void after_completion(in Status s);
  • invoked after complete or rollback
Heuristic Decisions

• unilateral decisions, before consensus was achieved
  • typically in expectation of a likely outcome, and under some resource pressure (e.g. lock timeout)
  • only allowed/possible in the "uncertain" state

• reported as exceptions
  • HeuristicRollback
  • HeuristicCommit
  • HeuristicMixed
  • HeuristicHazard (not all outcomes known; the known ones are either all commit or all rollback)
Transaction Context

- Automatically transmitted together with operation invocations
  - Alternatively: explicitly pass Control object to remote operation

- Specific format for a single TP monitor unspecified; interoperable version encoded as a IOP::ServiceContext (Serviceld 0), as PropagationContext

```
struct TransIdentity{
    Coordinator coord;
    Terminator term;
    otid_t otid; // compatible with XA XID
};

struct PropagationContext{
    unsigned long timeout;
    TransIdentity current;
    sequence<TransIdentity> parents;
    any implementation_specific_data;
};
```
Policies

- objects need to express their ability to participate in a transaction
- OTS 1.0, 1.1: Inheritance from empty interface TransactionalObject
- OTS 1.2: IOR contains component indicating policy of object (OTSPolicy)
  - requires: object must be invoked in the context of a transaction
  - forbids: object must not be invoked in a transaction
  - adapts: can live with or without transaction
- CORBA messaging: communication may go through a broker breaks transaction boundary
  - InvocationPolicy specifies whether target object requires SHARED transactions, UNSHARED transactions, or either kind
- Server code sets policy on POA creation
OTS Implementations

- Java Mapping: JTS (Java Transaction Service) is based on OTS 1.2
  - BEA Jolt
    - VisiBroker ITS (Integrated Transaction Service)
  - BEA Tuxedo (for C++)
  - Orbix E2A Application Server Platform
  - Encina++ (IBM TXSeries)
  - OpenORB transaction service
  - ...

Related Technology

- Persistent State Service (PSS) (formal/02-09-06)
  - data definition in PSDL

- Additional Structuring Mechanisms for OTS (formal/05-01-01)
  - Activity Service
  - support for long-running transactions
    - ACID properties not necessary; resources are committed before end of activity
  - additional transaction concepts: activity, compensation