Distributed Algorithms
Distributed Transactions

Alberto Montresor
University of Trento, Italy

2016/05/11

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.
Contents

1. Introduction and motivation
2. Transactional models
   - ACID Properties
   - Types of transactions
3. Implementing transactions
   - Atomicity
   - Isolation
   - Durability
4. From centralized to distributed
   - Isolation
   - Distributed commitment
Introduction

Example (Transfer money between two accounts)
- Withdraw 200€ from account 1
- Deposit 200€ to account 2

What happens if one of the operations is not performed?

Definition (Transaction)
The execution of a sequence of actions on a server that must be either entirely completed or aborted, independently of other transactions

Our goals:
- We want to allow the execution of concurrent transactions, yet to maintain consistency
- We want to deal with the failure of either the server or the client
ACID Properties

- **Atomicity**
  - Either all operations are completed or none of them is executed

- **Consistency**
  - Application invariants must hold before and after a transaction;
  - during the transaction, invariants may be violated but this is not visible outside

- **Isolation (Serializability)**
  - Execution of concurrent transactions should be equivalent (in effect) to a serialized execution

- **Durability**
  - Once a transaction commits, its effect are permanent
Transactional syntax

- Applications are coded in a stylized way:
  - begin transaction
  - perform a series of read, write operations
  - terminate by commit or abort

Example of transaction (sketch)

```plaintext
transaction T
  v_x ← read("x")
  v_y ← read("y")
  v_z ← v_x + v_y
  write("z", z)
  commit
```
Flat transactions

- Simplest, relatively easy to implement
- Their greatest strength (atomicity) is also their weakness (lack of flexibility)
- Technical issues:
  - How to maintain isolation
  - How to maintain atomicity + consistency
Nested transactions

- Constructed from a number of sub-transactions
- Sub-transactions may run in parallel or in sequence
- The subdivision is logical
- Flexibility
  - When a transaction fails, all its sub-transactions must fail too
  - When a sub-transaction fails:
    - The parent transaction could fail
    - Or, alternative actions could be taken
- Example next slide
Nested transactions

Example of nested transaction (sketch)

```
transaction "book travel"

    start transaction "book flight"
    start transaction "book hotel"
    start transaction "book car"
    if "book car" aborted then
        start transaction "book bus"
    if "book flight" and "book hotel" and ("book car" or "book bus") committed then
        commit
    else
        abort
```
Distributed transactions

- Can be either flat or nested
- Operates on distributed data (multiple servers)
- Technical issues
  - Separate distributed algorithms are needed to handle
    - locking of data in multiple distributed systems
    - committing data in an atomic way
### Distributed transactions

**Example of distributed transaction (sketch)**

---

**Site A, account x**

**transaction T**

\[ v_x \leftarrow \text{read}("x") \]

\[ \text{write}("x", v_x + C) \]

\[ \text{commit} \]

---

**Site B, account y**

**transaction T**

\[ v_y \leftarrow \text{read}("y") \]

\[ \text{if } v_y \geq C \text{ then} \]

\[ \text{write}("y", v_y - C) \]

\[ \text{commit} \]

\[ \text{abort} \]
Types of transactions

Nested transaction

Sub-transaction  Sub-transaction

Airline database  Hotel database

Two independent databases, hosted on the same machine

Distributed transaction

Two physically separated parts of the same database
Implementing transactions

Transactional system

- Application
- Transaction manager
- Scheduler
- Data manager
- Application-specific code (consistency)
- Begin/commit/abort (atomicity)
- Lock management, ordering of actions (isolation)
- Executes read/writes (durability)
Implementing transactions

Atomicity

- **Private workspaces**
  - At the beginning, give the transaction a private workspace and copy all required objects
  - Read/write/perform operations on the private workspace
  - If all operations are successful, commit by writing the updates in the permanent record; otherwise abort

- **How to extend this to a distributed system?**
  - Each copy of the transaction on different server is given a private workspace
  - Perform a distributed atomic commitment protocol
Implementing transactions

Atomicity

- Write-ahead log
  - Write operation / initial state / final state on a log
  - Modify "real" data
- In case of commit
  - Mark operation as committed on the log
- In case of abort
  - Mark operation as aborted on the log
  - Revert "real" data to the initial state
- How to extend this to a distributed system?
  - Distributed rollback recovery
Implementing transactions

Atomicity

x = 0;
y = 0;
BEGIN_TRANSACTION;
  x = x + 1;
  y = y + 2;
  x = y * y;
END_TRANSACTION;

(a) Log x = 0/1
(b) Log y = 0/2
(c) Log x = 0/1
(d) Log x = 1/4
Isolation

- Isolation
  - Means that effect of the interleaved execution is indistinguishable from some possible serial execution of the committed transactions

- Example:
  - $T_1$ and $T_2$ are interleaved but it ”looks like” $T_2$ ran before $T_1$

- The idea:
  - Transactions can be coded to be correct if run in isolation, and yet will run correctly when executed concurrently (hence gain a speedup)
Isolation

- Alice withdraws 250€ from \( X \)
  - 1) \( \text{local} \leftarrow \text{read}("x") \)
  - 2) \( \text{local} \leftarrow \text{local} - 250 \)
  - 3) \( \text{write}("x", \text{local}) \)

- Bob withdraws 250€ from \( X \)
  - 4) \( \text{local} \leftarrow \text{read}("x") \)
  - 5) \( \text{local} \leftarrow \text{local} - 250 \)
  - 6) \( \text{write}("x", \text{local}) \)

What happens with the following sequences?

- 1-2-3-4-5-6\textcolor{red}{- Correct}
- 4-5-6-1-2-3\textcolor{red}{- Correct}
- 1-4-2-5-3-6\textcolor{red}{- Lost update}
- 1-2-4-5-6-3\textcolor{red}{- Lost update}
Isolation

\[ \begin{align*}
T_1: & \quad R_1(X) \ R_1(Y) \ W_1(X) \ \text{commit}_1 \\
T_2: & \quad R_2(X) \ W_2(X) \ W_2(Y) \ \text{commit}_2 \\
DB: & \quad R_1(X) \ R_2(X) \ W_2(X) \ R_1(Y) \ W_1(X) \ W_2(Y) \ \text{commit}_1 \ \text{commit}_2
\end{align*} \]
Problem: transactions may “interfere”.

Here, $T_2$ changes $x$, hence $T_1$ should have either run first (read and write) or after (reading the changed value).

\[ T_1: \quad R_1(X) \quad R_1(Y) \quad W_1(X) \quad \text{commit}_1 \]
\[ T_2: \quad R_2(X) \quad W_2(X) \quad W_2(Y) \quad \text{commit}_2 \]
\[ \text{DB:} \quad R_1(X) \quad R_2(X) \quad W_2(X) \quad R_1(Y) \quad W_1(X) \quad W_2(Y) \quad \text{commit}_1 \quad \text{commit}_2 \]

Unsafe! Not serializable
Implementing transactions
Isolation

Data manager interleaves operations to improve concurrency but schedules them so that it looks as if one transaction ran at a time.

This schedule “looks” like $T_2$ ran first.
Unlike other kinds of distributed systems, transactional systems typically lock the data they access.

- **Lock coverage**
  - Suppose that transaction $T$ will access object $x$.
  - $T$ must get a lock that "covers" $x$.

- **We could have one lock**
  - ... per object
  - ... for the whole database
  - ... for a category of objects
  - ... per table
  - ... per row

- All transactions must obey the same rules!
Implementing transactions

Isolation

Strict 2-Phase locking (2-PL)

1\textsuperscript{st} phase ("growing")
- Whenever the scheduler receives an operation \(\text{operation}(T, x)\)
- If \(x\) is already owned by a conflicting lock:
  - the operation (and the transaction) is delayed
  - the lock is granted
- Obtain all the locks it needs while it runs and hold onto them even if no longer needed!

2\textsuperscript{nd} phase ("shrinking")
- release locks only after making commit/abort decision and only after updates are persistent
2-PL implies Serializability

- Suppose that $T'$ performs an operation that conflicts with an operation that $T$ has done
  - e.g., $T'$ will update data item $x$ that $T$ read or updated
  - e.g., $T$ updated item $y$ and $T'$ will read it
- $T$ must have had a lock on $x/y$ that conflicts with the lock that $T'$ wants
  - $T$ won’t release it until it commits or aborts
  - So $T'$ will wait until $T$ commits or aborts
- Note: 2-PL may cause deadlocks; usual techniques apply
Durability: Lampson’s stable storage

- Maintain two copies of object $A$ ($A_0$ and $A_1$) plus two timestamps and checksums (different disks)
- Failure may happen at anytime between the six operations

### Writing

```
UPDATE(A, x)
```

- $A_0 \leftarrow x$
- $T_0 \leftarrow \text{now}()$
- $S_0 \leftarrow \text{checksum}(x, t_0)$
- $A_1 \leftarrow x$
- $T_1 \leftarrow \text{now}()$
- $S_1 \leftarrow \text{checksum}(x, t_1)$

### Recovery

```
READ(A)
```

- if $S_0 = \text{checksum}(A_0, T_0)$ \&\& $S_1 = \text{checksum}(A_1, T_1)$ then
  - if $T_0 > T_1$ then return $A_0$
  - else return $A_1$
- if $S_0 = \text{checksum}(A_0, T_0)$ \&\& $S_1 \neq \text{checksum}(A_1, T_1)$ then
  - return $A_0$
- if $S_0 \neq \text{checksum}(A_0, T_0)$ \&\& $S_1 = \text{checksum}(A_1, T_1)$ then
  - return $A_1$
Distributed locking

- **Centralized 2-PL**
  - A single site is responsible for granting and releasing locks
  - Each scheduler communicates with this centralized scheduler
  - Issues: bottleneck, central point of failure

- **Primary 2-PL**
  - Each data item is assigned to a "primary" server
  - Each scheduler communicates with the scheduler of the primary
  - Issues: central points of failure

- **Distributed 2-PL**
  - Locking is done in a decentralized way; messages are exchanged through reliable multicast
Failures on centralized system

- If application crashes:
  - Treat as an abort
- If transactional system crashes:
  - Abort non-committed transactions, but committed state is durable
- Aborted transactions:
  - Leave no effect, either in database itself or in terms of indirect side-effects
  - Only need to consider committed operations in determining serializability
Distributed commit problem

- Atomic commitment
- The distributed commit problem involves having an operation being performed by a distributed set of participants
- Two protocols
  - Two-phase commit (2PC)
    - blocking, efficient
    - Jim Gray (1978)
  - Three-phase commit (3PC)
    - non-blocking
    - Dale Skeen (1981)
- Implementations based on a coordinator
System model

- Fixed set of participants, known to all
- No message losses
- Synchronous communication: all messages arrive in $\delta$ time units
- $\Delta_b$-timeliness: it is possible to build a broadcast primitive such that all messages arrive in $\Delta_b$ time units
- Clocks exist
  - they are not required to be synchronized
  - they may drift from real-time

- Which systems:
  - local area networks
Generic Participant

Executed by the invoker

\textbf{send} \langle \text{TSTART}, \text{transaction}, \text{participants} \rangle \text{ to participants}

Executed by all participants (including the invoker)

\textbf{upon} receipt of \langle \text{TSTART}, \text{transaction}, \text{participants} \rangle \text{ do}

\begin{align*}
C_{\text{now}} & \leftarrow \text{now}() \\
\{ \text{ Perform operations requested by transaction } \} \\
\text{if} \ &\text{ willing and able to make updates permanent} \ \text{then} \\
& \text{vote} = \text{YES} \\
\text{else} \ \\
& \text{vote} = \text{NO} \\
\text{AtomicCommitment(} &\text{transaction, participants})
\end{align*}
Coordinator selection

Coordinator axioms

AX1 At most one participant will assume the role of coordinator

AX2 If no failures occur, one participant will assume the role of coordinator

AX3 There exists a known constant $\Delta_c$ such that no participant assumes the role of coordinator more than $\Delta_c$ time units after the beginning of the transaction
From centralized to distributed

Distributed commitment

Specification

Atomic commitment

AC1 All participants that decide reach the same decision
AC2 If any participant decides commit, then all participants must have voted yes
AC3 If all participants vote yes and no failures occur, then all participants decide commit
AC4 Each participant decides at most once
Atomic commitment

A generic algorithm – *Atomic Commitment Protocol* (ACP)
- Based on a generic broadcast primitive
- By ”plugging in” different versions of broadcast we obtain different versions of ACP

Three phases:
- **Phase 1**: The coordinator asks for votes yes/no from participants and take a commit/abort decision
- **Phase 2**: The coordinator disseminates the decision
- **Phase 3**: Termination protocol; we’ll see
Atomic Commitment Protocol

procedure atomic_commitment(transaction, participants)
    cobegin
        % Task 1: Executed by the coordinator
        send [VOTE_REQUEST] to all participants
        set-timeout-to local_clock + 2\delta
        wait-for (receipt of [VOTE: vote] messages from all
        if (all votes are YES) then
            broadcast (COMMIT, participants)
        else broadcast (ABORT, participants)
        on-timeout
        broadcast (ABORT, participants)
    coend
Atomic Commitment Protocol

9 % Task 2: Executed by all participants (including the coordinator)
10 \textbf{set-timeout-to} \( C_{\text{knou}} + \Delta_c + \delta \)
11 \textbf{wait-for} (receipt of \texttt{[VOTE REQUEST]} from coordinator)
12 \textbf{send} \texttt{[VOTE: vote]} to coordinator
13 \textbf{if} (vote = \texttt{NO}) \textbf{then}
14 \hspace{1em} decide \texttt{ABORT}
15 \textbf{else}
16 \hspace{1em} \textbf{set-timeout-to} \( C_{\text{knou}} + \Delta_c + 2\delta + \Delta_t \)
17 \hspace{1em} \textbf{wait-for} (delivery of decision message)
18 \hspace{1em} \textbf{if} (decision message is \texttt{ABORT}) \textbf{then}
19 \hspace{2em} decide \texttt{ABORT}
20 \hspace{1em} \textbf{else} decide \texttt{COMMIT}
21 \hspace{1em} \textbf{on-timeout}
22 \hspace{2em} decide according to termination\_protocol()
23 \hspace{1em} \textbf{on-timeout}
24 \hspace{2em} decide \texttt{ABORT}
25 \textbf{coend}
26 \textbf{end}
Terminating Best-Effort Broadcast

Definition (TBEB1 - Validity)
If $p$ and $q$ are correct, then every message B-broadcast by $p$ is eventually delivered by $q$

Definition (TBEB2 - Uniform Integrity)
$m$ is delivered by a process at most once, and only if it was previously broadcast

Definition (TBEB3 - $\Delta_b$-Timeliness)
All messages arrive in $\Delta_b$ time units since the time they were sent
The ACP algorithm with a best-effort broadcast implementation

- AC1: All participants that decide reach the same decision
- AC2: If any participant decides commit, then all participants must have voted yes
- AC3: If all participants vote yes and no failures occur, then all participants decide commit
- AC4: Each participant decides at most once

---

<table>
<thead>
<tr>
<th>AC1: All participants that decide reach the same decision</th>
<th>See next page (too complex to fit in this box!)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AC2: If any participant decides commit, then all participants must have voted yes</td>
<td>From the structure of the program (the coordinator must have received yes from all participants)</td>
</tr>
<tr>
<td>AC3: If all participants vote yes and no failures occur, then all participants decide commit</td>
<td>Given reliable communication, no failure, synchrony, all messages arrives before deadlines</td>
</tr>
<tr>
<td>AC4: Each participant decides at most once</td>
<td>From the algorithm structure (decide op. are mutually exclusive)</td>
</tr>
</tbody>
</table>
Proof of AC1, by contradiction

1. Let $p$ decide commit, let $q$ decide \textbf{abort}. By AC4, $p \neq q$

2. $p$ must have received a broadcast from a coordinator $c$
   
   2.1 By AC2, $c$ must have received votes \textit{YES} from all, including $q$

3. A process decide \textbf{abort} in lines 14, 19, 24
   
   3.1 Line 14 is excluded by 2.1
   3.2 Line 24 is excluded by 2.

4. So $q$ must have delivered a message \textbf{abort} in line 19

5. But this message must have been sent by a coordinator different from $c$; but this is a contradiction with AX1 (unique coordinator)
Blocking vs non-blocking

- In some cases, a termination protocol is invoked
- Informally, tries to contact other participants to learn a decision
  - For example:
    - if a process has already decided, copy the decision
    - if a process has not voted, decide **abort**
- But consider this scenario:
  - the coordinator crashes during the broadcast of a decision
  - all faulty participants decide and then crash
  - all correct participants have previously voted YES, and they do not deliver a decision
- **ACP-TBEF** is blocking in this scenario
Blocking vs non-blocking

Non-blocking atomic commitment

\{ AC1–AC4 \}

AC5 Every correct participant that executes the atomic commitment protocol eventually decides
Uniform Terminating Reliable Broadcast

**Definition (URB1 - Validity)**

If $p$ and $q$ are correct, then every message B-broadcast by $p$ is eventually delivered by $q$.

**Definition (URB2 - Uniform Agreement)**

If a correct process delivers $m$, then all correct processes eventually deliver $m$.

**Definition (URB3 - Uniform Integrity)**

$m$ is delivered by a process at most once, and only if it was previously broadcast.

**Definition (URB4 - $\Delta_b$-Timeliness)**

All messages arrive in $\Delta_b$ time units since the time they were sent.
ACP - UTRB

% Task 2: Executed by all participants (including the coordinator)

set-timeout-to $C_{knco} + \Delta_c + \delta$

wait-for (receipt of [VOTE_REQUEST] from coordinator)

send [VOTE: vote] to coordinator

if (vote = NO) then
    decide ABORT

else

set-timeout-to $C_{knco} + \Delta_c + 2\delta + \Delta_b$

wait-for (delivery of decision message)

if (decision message is ABORT) then
    decide ABORT

else decide COMMIT

on-timeout

decide ABORT

on-timeout

decide ABORT

doend
### ACP - UTRB

If we use UTRB instead of TBE, we obtain ACP-UTRB, which is equivalent to 3PC.

### Correctness

The termination protocol is not needed any more.

Proof:

- **AC1-AC4**: only AC1 is changed from before, we need to prove that $q$ cannot decide `abort` in line 22
- **AC5**: By the structure of the protocol, each line we have a decide
ACP - UTRB – Performance

- **ACP- BEB**
  - 4n total messages
  - n invoker-to-all
  - n coordinator-to-all
  - n all-to-coordinator
  - n coordinator-to-all

- **ACP-UTRB**
  - 3n + n^2 total messages
  - n invoker-to-all
  - n coordinator-to-all
  - n all-to-coordinator
  - n^2 all-to-all
To conclude, we need to consider the possibility of a participant that was down becoming operational after being repaired.

**Recovery protocol**

- During normal execution, log all “transactional events” in a distributed transaction log (dt-log)
  - T-START, VOTE YES, VOTE NO, commit, abort
- At recovery, try to conclude all transactions that were in progress at the participant at the time of crash
- If recovery is not possible by simply looking at the log, try to get help from other participants
Recovery protocol

procedure recovery\_protocol(p)
% Executed by recovering participant p
1 \( R := \text{set of DT-log records regarding transaction} \)
2 \[\text{case } R \text{ of} \]
3 \{\} : \text{skip} \\
4 \{\text{start}\} : \text{decide ABORT} \\
5 \{\text{start,no}\} : \text{decide ABORT} \\
6 \{\text{start,vote,decision}\} : \text{skip} \\
7 \{\text{start,yes}\} : \\
8 \text{while (undecided) do} \\
9 \text{send [HELP, transaction] to all participants} \\
10 \text{set-timeout-to } 2\delta \\
11 \text{wait-for receipt of [REPLY: transaction, reply] message} \\
12 \text{if (reply \neq ?) then} \\
13 \text{decide reply} \\
14 \text{else} \\
15 \text{if (received ? replies from all participants) then} \\
16 \text{decide ABORT} \\
17 \text{on-timeout} \\
18 \text{skip} \]
Recovery protocol

upon (receipt of [HELP, transaction] message from p)

R := set of DT-log records regarding transaction

case R of

{}: decide ABORT; send [REPLY: transaction, ABORT] to p

{start}: decide ABORT; send [REPLY: transaction, ABORT] to p

{start,no}: send [REPLY: transaction, ABORT] to p

{start,vote,decision}: send [REPLY: transaction, decision] to p

{start,yes}: send [REPLY: transaction, ?] to p

esac
Reading Material