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)

```
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 $T$

\[
\begin{align*}
  & v_x \leftarrow \text{read(”} x \text{”) } \\
  & \text{write(”} x \text{”, } v_x + C \text{)} \\
  & \text{commit}
\end{align*}
\]

Site $B$, account $y$

transaction $T$

\[
\begin{align*}
  & v_y \leftarrow \text{read(”} y \text{”) } \\
  & \text{if } v_y \geq C \text{ then} \\
  & \quad \text{write(”} y \text{”, } v_y - C \text{)} \\
  & \quad \text{commit} \\
  & \text{abort}
\end{align*}
\]
Types of transactions

- **Nested transaction**
  - Sub-transaction
  - Sub-transaction
  - Airline database
  - Hotel database
- **Distributed transaction**
  - Net
  - A
  - B
  - Two physically separated parts of the same database

Two independent databases, hosted on the same machine
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.
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
Atomicity

\[\begin{align*}
x &= 0; \\
y &= 0;
\end{align*}\]

BEGIN TRANSACTION;
\[\begin{align*}
x &= x + 1; \\
y &= y + 2; \\
x &= y \times y;
\end{align*}\]

END TRANSACTION;

(a) \hspace{1cm} (b) \hspace{1cm} (c) \hspace{1cm} (d)

Log \hspace{1cm} Log \hspace{1cm} Log
\[\begin{align*}
x &= 0/1 \\
y &= 0/2 \\
x &= 1/4
\end{align*}\]
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)
Implementing transactions

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 \hspace{1cm} \text{Correct}
- 4-5-6-1-2-3 \hspace{1cm} \text{Correct}
- 1-4-2-5-3-6 \hspace{1cm} \text{Lost update}
- 1-2-4-5-6-3 \hspace{1cm} \text{Lost update}
Implementing transactions

Isolation

DB: \( R_1(X) \) \( R_2(X) \) \( W_2(X) \) \( R_1(Y) \) \( W_1(X) \) \( W_2(Y) \) \( \text{commit}_1 \) \( \text{commit}_2 \)

\( T_1: \) \( R_1(X) \) \( R_1(Y) \) \( W_1(X) \) \( \text{commit}_1 \)

\( T_2: \) \( R_2(X) \) \( W_2(X) \) \( W_2(Y) \) \( \text{commit}_2 \)

DB: \( R_1(X) \) \( R_2(X) \) \( W_2(X) \) \( R_1(Y) \) \( W_1(X) \) \( W_2(Y) \) \( \text{commit}_1 \) \( \text{commit}_2 \)
Implementing transactions

Isolation

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).

DB: $R_1(X) \ R_2(X) \ W_2(X) \ W_2(Y) \ R_1(Y) \ W_1(X) \ W_2(Y) \ commit_1 \ commit_2$

Unsafe! Not serializable
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.
Locking

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!**
Strict 2-Phase locking (2-PL)

1\textsuperscript{st} phase (”growing”)
- Whenever the scheduler receives an operation $operation(T, x)$
- 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

\[
\begin{align*}
\text{UPDATE}(A, x) & \quad \text{READ}(A) \\
A_0 & \leftarrow x & \text{if } S_0 = \text{checksum}(A_0, T_0) \land S_1 = \text{checksum}(A_1, T_1) \text{ then} \\
T_0 & \leftarrow \text{now}() \quad \text{if } T_0 > T_1 \text{ then return } A_0 \leftarrow x \quad \text{else return } A_1 \\
S_0 & \leftarrow \text{checksum}(x, t_0) \quad \text{if } S_0 \neq \text{checksum}(A_0, T_0) \land S_1 \neq \text{checksum}(A_1, T_1) \text{ then} \\
A_1 & \leftarrow x \quad \text{return } A_0 \leftarrow \text{checksum}(x, t_1) \quad \text{if } S_0 \neq \text{checksum}(A_0, T_0) \land S_1 = \text{checksum}(A_1, T_1) \text{ then} \\
T_1 & \leftarrow \text{now}() \quad \text{return } A_1 \left. \right\}
\end{align*}
\]
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

\[
\text{send } \langle \text{tstart}, \text{transaction}, \text{participants} \rangle \text{ to participants}
\]

Executed by all participants (including the invoker)

\[
\begin{align*}
\text{upon receipt of } \langle \text{tstart}, \text{transaction}, \text{participants} \rangle \text{ do} \\
& C_{\text{now}} \leftarrow \text{now()} \\
& \{ \text{ Perform operations requested by } \text{transaction } \} \\
& \text{if willing and able to make updates permanent then} \\
& \quad \text{vote } = \text{YES} \\
& \text{else} \\
& \quad \text{vote } = \text{NO} \\
& \text{AtomicCommitment}(\text{transaction}, \text{participants})
\end{align*}
\]
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
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

```plaintext
procedure atomic-commitment(transaction, participants)
    cobegin
        % Task 1: Executed by the coordinator
        1      send [VOTE_REQUEST] to all participants
        2      set-timeout-to local_clock + 2δ
        3      wait-for (receipt of [VOTE: vote] messages from all
                if (all votes are YES) then
                    5      broadcast (COMMIT, participants)
                else broadcast (ABORT, participants)
                7      on-timeout
                8      broadcast (ABORT, participants)
    coend
```

Alberto Montresor (UniTN)  DS - Distributed Transactions  2016/05/11  29 / 42
Atomic Commitment Protocol

9 % Task 2: Executed by all participants (including the coordinator)
10 set-timeout-to $C_{kno} + \Delta_c + \delta$
11 wait-for (receipt of [VOTE_REQUEST] from coordinator)
12 send [VOTE: vote] to coordinator
13 if (vote = NO) then
14 decide ABORT
15 else
16 set-timeout-to $C_{kno} + \Delta_c + 2\delta + \Delta_t$
17 wait-for (delivery of decision message)
18 if (decision message is ABORT) then
19 decide ABORT
20 else decide COMMIT
21 on-timeout
22 decide according to termination_protocol()
23 on-timeout
24 decide ABORT
coend
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.
ACP - TBEB

- The ACP algorithm with a best-effort broadcast implementation
- It happens to be equivalent to 2PC

<table>
<thead>
<tr>
<th>AC1:</th>
<th>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:</td>
<td>If any participant decides <strong>commit</strong>, then all participants must have voted <strong>YES</strong></td>
<td>From the structure of the program (the coordinator must have received yes from all participants)</td>
</tr>
<tr>
<td>AC3:</td>
<td>If all participants vote <strong>YES</strong> and no failures occur, then all participants decide <strong>commit</strong></td>
<td>Given reliable communication, no failure, synchrony, all messages arrives before deadlines</td>
</tr>
<tr>
<td>AC4:</td>
<td>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 \textbf{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-TBEB** 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

<table>
<thead>
<tr>
<th>Definition (URB1 - Validity)</th>
</tr>
</thead>
<tbody>
<tr>
<td>If $p$ and $q$ are correct, then every message B-broadcast by $p$ is eventually delivered by $q$.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Definition (URB2 - Uniform Agreement)</th>
</tr>
</thead>
<tbody>
<tr>
<td>If a correct process delivers $m$, then all correct processes eventually deliver $m$.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Definition (URB3 - Uniform Integrity)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$m$ is delivered by a process at most once, and only if it was previously broadcast.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Definition (URB4 - $\Delta_b$-Timeliness)</th>
</tr>
</thead>
<tbody>
<tr>
<td>All messages arrive in $\Delta_b$ time units since the time they were sent.</td>
</tr>
</tbody>
</table>
The Non-Blocking Atomic Commitment Protocol: ACP-UTRB

9  % Task 2: Executed by all participants (including the coordinator)
10   set-timeout-to $C_{kn cw} + \Delta_c + \delta$
11   wait-for (receipt of [VOTE_REQUEST] from coordinator)
12   send [VOTE: vote] to coordinator
13   if (vote = NO) then
14       decide ABORT
15   else
16       set-timeout-to $C_{kn cw} + \Delta_c + 2\delta + \Delta_b$
17       wait-for (delivery of decision message)
18       if (decision message is ABORT) then
19           decide ABORT
20       else decide COMMIT
21       on-timeout
22           decide ABORT
23       on-timeout
24           decide ABORT
25   coend
26 coend

Figure 4. ACP-UTRB: A Non-Blocking Atomic Commitment Protocol Based on UTRB
ACP - UTRB

If we use UTRB instead of TBEB, 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 \texttt{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
Recovery

- 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 := set of DT-log records regarding transaction
2 case R of
3 {}:
4 {start}:
5 {start,no}:
6 {start,vote,decision}:
7 {start,yes}:
8 \begin{align*}
9 & \text{while (undecided) do} \\
10 & \text{send [HELP, transaction] to all participants} \\
11 & \text{set-timeout-to } 2\delta \\
12 & \text{wait-for receipt of [REPLY: transaction, reply] message} \\
13 & \quad \text{if (reply \neq ?) then} \\
14 & \quad \quad \text{decide reply} \\
15 & \quad \text{else} \\
16 & \quad \quad \text{if (received ? replies from all participants) then} \\
17 & \quad \quad \quad \text{decide ABORT} \\
18 & \text{on-timeout} \\
19 & \quad \text{skip} \\
20 \end{align*}

Recovery protocol

1. upon (receipt of \([\text{HELP, transaction}]\) message from \(p\))
2. \(R := \text{set of DT-log records regarding transaction}\)
3. case \(R\) of
4. \(
\{
\\}
\)
5. \(
\{\text{start}\}\)
6. \(
\{\text{start, no}\}\)
7. \(
\{\text{start, vote, decision}\}\)
8. \(
\{\text{start, yes}\}\)
endcase

\text{decide ABORT; send [REPLY: transaction, ABORT] to } p
\text{decide ABORT; send [REPLY: transaction, ABORT] to } p
\text{send [REPLY: transaction, ABORT] to } p
\text{send [REPLY: transaction, decision] to } p
\text{send [REPLY: transaction, ?] to } p