Distributed Systems

Distributed Transactions

Alberto Montresor
Università di Trento

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-sa/2.5/ or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.
Introduction

- A possible scenario: transfer money from my “checking account” to a “saving” account (e.g. Conto Arancio)
  - Withdraw 200€ from account: IT16 01102 10502 0000 1054 7023 795
  - Deposit 200€ to account: IT16 01220 42440 0000 1234 795 7942

- What happens if one of the operations is not performed?
Introduction

- A transaction is 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.
Summary

- **Transactions**
  - The most important reliability technology for client-server systems

- **Recap about the transactional model**
  - ACID properties
  - How to implement a transactional system
    - Locking
    - Write-ahead log

- **From single-server to multiple server: distributed transactions**
  - Distributed locking
  - Distributed commit *(atomic commitment)*
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 effects are permanent
Applications are coded in a stylized way:

- begin transaction
- perform a series of read, write operations
- Terminate by commit or abort

begin transaction T;
    x = read("x", ....);
    y = read("y", ....);
    z = x+y;
    write("z", z, ....);
commit transaction T;
Types of transaction

- **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
Types of transaction

- **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:
Types of transaction

• Example of nested transaction (sketch)

begin transaction;
  start transaction "book flight"
  start transaction "book hotel"
  start transaction "book car"
  if (car aborted) then
    start transaction "book bus"
  if (flight, hotel are ok and (car or bus) is ok) then
    commit
  else
    abort
end transaction
Types of transaction

- **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
## Types of transaction

- **Example of distributed transaction (sketch)**
  Money transfer from A.x to B.y

**Site A, account x**

```plaintext
begin transaction T
  x = read("x")
  write("x", x+amount)
end transaction T
```

**Site B, account y**

```plaintext
begin transaction T
  y = read("y")
  write("y", y-amount)
end transaction T
```
Differences between nested and distributed

**Nested transaction**

- Sub-transaction
- **Airline database**
- **Hotel database**

- Two independent databases, hosted on the same machine

**Distributed transaction**

- **Net**
- **A**
- **B**

- Two physically separated parts of the same database
The diagram illustrates the components of a transactional system:

- **Application**: Handles user requests and application-specific code, ensuring consistency.
- **Transaction manager**: Manages transactions, ensuring atomicity.
- **Scheduler**: Controls the ordering of actions, ensuring isolation.
- **Data manager**: Performs read/writes, ensuring durability.

The system ensures:
- **Consistency**: Through application-specific code.
- **Atomicity**: By beginning, committing, or aborting transactions.
- **Isolation**: By managing locks and ordering actions.
- **Durability**: By executing read/writes.
Implementing transactions

- **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

- 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
Write-ahead logs

\[
x = 0; \\
y = 0; \\
\text{BEGIN\_TRANSACTION;} \\
\quad x = x + 1; \\
\quad y = y + 2; \\
\quad x = y \ast y; \\
\text{END\_TRANSACTION;}
\]

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

- **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)
Operation example

- Alice withdraws 250€ from X
  1) local = read("X")
  2) local := local-250
  3) write("X", local)

- Bob withdraws 250€ from X
  4) local = read("X")
  5) local := local-250
  6) write("X", local)

- What happens with the following sequences?
  - 1-2-3-4-5-6
  - 4-5-6-1-2-3
  - 1-4-2-5-3-6
  - 1-2-4-5-6-3
**Operation example**

- **Alice withdraws 250€ from X**
  1) local = read(“X”)
  2) local := local-250
  3) write(“X”, local)

- **Bob withdraws 250€ from X**
  4) local = read(“X”)
  5) local := local-250
  6) write(“X”, local)

- **What happens with the following sequences?**
  - 1-2-3-4-5-6
  - 4-5-6-1-2-3
  - 1-4-2-5-3-6
  - 1-2-4-5-6-3

  **correct**
Operation example

- Alice withdraws 250€ from X
  1) local = read("X")
  2) local := local-250
  3) write("X", local)

- Bob withdraws 250€ from X
  4) local = read("X")
  5) local := local-250
  6) write("X", local)

- What happens with the following sequences?
  - 1-2-3-4-5-6: correct
  - 4-5-6-1-2-3: correct
  - 1-4-2-5-3-6
  - 1-2-4-5-6-3
Operation example

- **Alice withdraws 250€ from X**
  1) local = read("X")
  2) local := local-250
  3) write("X", local)

- **Bob withdraws 250€ from X**
  4) local = read("X")
  5) local := local-250
  6) write("X", local)

- What happens with the following sequences?
  - 1-2-3-4-5-6: correct
  - 4-5-6-1-2-3: correct
  - 1-4-2-5-3-6: lost update
  - 1-2-4-5-6-3
**Operation example**

- **Alice withdraws 250€ from X**
  1) local = read(“X”)
  2) local := local-250
  3) write(“X”, local)

- **Bob withdraws 250€ from X**
  4) local = read(“X”)
  5) local := local-250
  6) write(“X”, local)

- **What happens with the following sequences?**
  - 1-2-3-4-5-6 **correct**
  - 4-5-6-1-2-3 **correct**
  - 1-4-2-5-3-6 lost update
  - 1-2-4-5-6-3 lost update
Data manager interleaves operations to improve concurrency
(Isolation) Serializability

Problem: transactions may “interfere”.

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

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

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

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

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.
Locks

- Unlike other kinds of distributed systems, transactional systems typically lock the data they access.

- **Lock coverage**
  - Suppose that transaction $T$ will access object $x$.
  - We must be sure that first, $T$ gets 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 use the same rules!
Strict 2-Phase Locking (2-PL)

- **1\textsuperscript{st} phase ("growing")**
  - Whenever the scheduler receives an operation $\text{oper}(T,x)$
    - *If $x$ is already owned by a conflicting lock:*
      the operation (and the transaction) is delayed
    - Otherwise:
      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
Strict 2-Phase Locking implies Serializability

- Suppose that T’ performs an operation that conflicts with an operation that T has done
  - T’ will update data item x that T read or updated
  - 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-Phase Locking may cause deadlocks:
  - Usual techniques apply
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
**Durability**

- **Lampson's stable storage**
  - Maintain two copies of object \( A (A_0 \text{ and } A_1) \) plus two timestamps and checksums (different disks)

- **Update(\( A, x \))**
  1) \( A_0 := x \)
  2) \( T_0 := \text{now()} \)
  3) \( S_0 := \text{checksum}(x, T_0) \)
  4) \( A_1 := x \)
  5) \( T_1 := \text{now()} \)
  6) \( S_1 := \text{checksum}(x, T_1) \)

- **Failure**
  - Anytime between 1 and 6

- **Recovery**
  - \( S_0 = \text{checksum}(A_0, T_0) \) and \( S_1 = \text{checksum}(A_1, T_1) \) and \( T_0 > T_1 \rightarrow \text{accept } A_0 \)
  - \( S_0 = \text{checksum}(A_0, T_0) \) and \( S_1 = \text{checksum}(A_1, T_1) \) and \( T_0 < T_1 \rightarrow \text{accept } A_1 \)
  - \( S_0 \neq \text{checksum}(A_0, T_0) \) and \( S_1 \neq \text{checksum}(A_1, T_1) \rightarrow \text{accept } A_0 \)
  - \( S_0 \neq \text{checksum}(A_0, T_0) \) and \( S_1 = \text{checksum}(A_1, T_1) \rightarrow \text{accept } A_1 \)
Distributed transactions

- **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**

- **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 exists
    - they are not required to be synchronized
    - they may drift from real-time

- **Which systems:**
  - local area networks
Generic participant

% Some participant (the invoker) executes:
1    send [T\_START: transaction, \(\Delta_c\), participants] to all participants

% All participants (including the invoker) execute:
2    upon (receipt of [T\_START: transaction, \(\Delta_c\), participants])
3        \(C_{kn.ow}\) := local_clock
4    % Perform operations requested by transaction
5    if (willing and able to make updates permanent) then
6        vote := YES
7    else vote := NO
8    % Decide COMMIT or ABORT according to atomic commitment protocol
9    atomic\_commitment(transaction, participants)
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 constant $\Delta_c$ such that no participant assumes the role of coordinator more than $\Delta_c$ time units after the beginning of the transaction

- Example implementation
  - The participant with the smallest id
Atomic Commitment Specification

- **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 (that is, a decision is irreversible).
Atomic Commitment

- We show a generic algorithm – ACP, or Atomic Commitment Protocol
  - Based on a generic broadcast primitive
  - By “plugging in” different versions of broadcast we obtain different versions of ACP

- 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
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 participants)
4          if (all votes are YES) then
5                broadcast (COMMIT, participants)
6          else broadcast (ABORT, participants)
7    on-timeout
8    broadcast (ABORT, participants)
A generic Atomic Commitment Protocol (ACP)

9 % Task 2: Executed by all participants (including the coordinator)
10     set-timeout-to $C_{know} + \Delta_c + \delta$
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 $C_{know} + \Delta_c + 2\delta + \Delta_b$
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 according to termination_protocol()
23         on-timeout
24             decide ABORT
25 coend
end
Terminating Best-Effort Broadcast (TBEB)

- **BEB1 - Validity**
  - By the Validity property of Perfect Links and the very facts that
    - (1) the sender sends the message to all
    - (2) every correct process that receives a message B-delivers it

- **BEB2 – Integrity**
  - By the Integrity property of Perfect Links

- **Bonus: $\Delta_b$-Timeliness**
  - All messages arrive in $\Delta_b$ time units since the time they were sent
ACP - BEB

ACP-BEB:

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

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

See next page (too complex to fit in this box!)

From the structure of the program (the coordinator must have received YES from all participants)

Given reliable communication, no failure, synchrony, all messages arrives before deadlines

From the structure of the algorithm (decide operations are mut. exclusive)
Proof of AC1, by contradiction

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

2) $p$ must have received a broadcast from a coordinator $c$

   2.1) By \textbf{AC2}, $c$ must have received votes \textsc{yes} from all, including $q$

3) a process decide \textsc{abort} in lines 14,19, 24

   3.1) 14 is excluded by 2.1
   3.2) 24 is excluded by 2.

4) so $q$ must have delivered a message \textsc{abort} in line 19

5) But this message must have been sent a coordinator different from $c$; but this is a contradiction with \textbf{AX1} (unique coordinator)
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-BEB is blocking in this scenario
Blocking vs non-blocking

- Non-blocking atomic commitment: \{ AC1-AC4 \} + AC5
- AC5
  - Every correct participant that executes the atomic commitment protocol eventually decides
- ACP-BEB = 2PC is blocking
Uniform Terminating Reliable Broadcast

- **Uniform Agreement**: If any participant (correct or not) delivers a message m, then all correct participants eventually deliver m.

- **BEB1, BEB2, $\Delta_b$ - Timeliness**

- If we use UTRB instead of TBEB, we obtain UTRB-APC, which is equivalent to 3PC.
% Task 2: Executed by all participants (including the coordinator)

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

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

send [VOTE: vote] to coordinator

if (vote = NO) then
    decide ABORT
else
    set-timeout-to $C_{know} + \Delta_c + 2\delta + \Delta_b$
    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

coend
end
Non-blocking ACP

- The termination protocol is not needed any more
- Proofs
  - 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
Performance

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

- **ACP-UTRB, flooding version**
  - $3n + n^2$ total messages
  - $n$ invoker-to-all
  - $n$ coordinator-to-all
  - $n$ all-to-coordinator
  - $n^2$ all-to-all

- **ACP-UTRB, with FD**
  - Perfect FD obtainable in a synchronous system
  - So we can use our FD-based URB protocol

- **ACP-UTRB, with FD**
  - $4n$ total messages
Dealing with 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
procedure recovery_protocol(p)
% Executed by recovering participant p
1   R := set of DT-log records regarding transaction
2   case R of
3       {}: skip
4       {start}: decide ABORT
5       {start,no}: decide ABORT
6       {start,vote,decision}: skip
7       {start,yes}:
8           while (undecided) do
9               send [HELP, transaction] to all participants
10              set-timeout-to 2δ
11              wait-for receipt of [REPLY: transaction, reply] message
12                  if (reply ≠?) then
13                      decide reply
14                  else
15                      if (received ? replies from all participants) then
16                          decide ABORT
17                      on-timeout
18                          skip
19               od
20       esac
21 end
Recovery protocol

1. \textbf{upon} (receipt of $[\text{HELP, transaction}]$ message from $p$)
2. \hspace{0.5cm} $R := \text{set of DT-log records regarding transaction}$
3. \hspace{0.5cm} \textbf{case} $R$ \textbf{of}
4. \hspace{1.5cm} \{\}\hspace{1cm} \text{decide ABORT; send} $[\text{REPLY: transaction, ABORT}]$ \text{to} $p$
5. \hspace{1.5cm} \{\text{start}\}\hspace{1cm} \text{decide ABORT; send} $[\text{REPLY: transaction, ABORT}]$ \text{to} $p$
6. \hspace{1.5cm} \{\text{start, no}\}\hspace{1cm} \text{send} $[\text{REPLY: transaction, ABORT}]$ \text{to} $p$
7. \hspace{1.5cm} \{\text{start, vote, decision}\}\hspace{1cm} \text{send} $[\text{REPLY: transaction, decision}]$ \text{to} $p$
8. \hspace{1.5cm} \{\text{start, yes}\}\hspace{1cm} \text{send} $[\text{REPLY: transaction, ?}]$ \text{to} $p$

\textbf{esac}