COMP90020: Distributed Algorithms

12. Nested & Distributed Transactions

From Many to Many, Seeking Scalability and Robustness

Miquel Ramirez

THE UNIVERSITY OF MELBOURNE

Semester 1, 2019

Agenda

Agenda

Nested Transactions

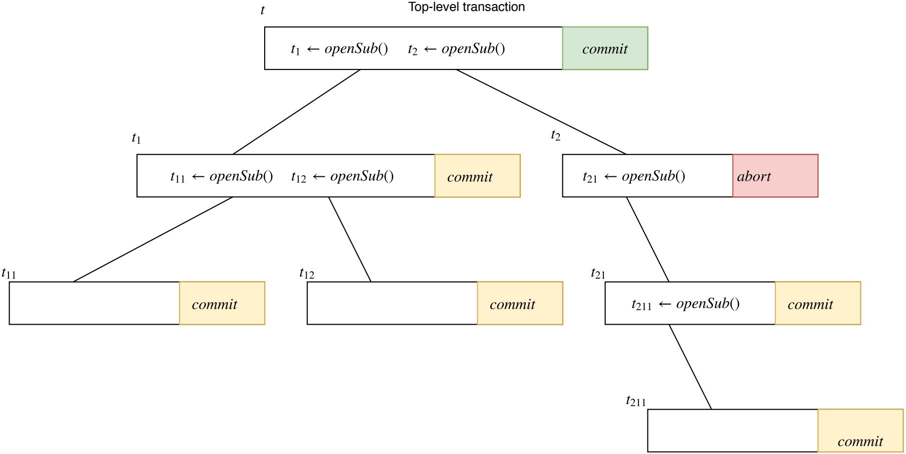
Transactions can be Composed

Transactions can be broken down into different independent components to increase throughput and reliability.

A transaction can start several others

- outermost transaction t is called top-level transaction,
- others are called subtransactions.

Illustration of Nested Transactions



Coordinator with Nested Transactions Support

Coordinator interface

1. start(), returns unique identifier id
2. close(id), finalize transaction id, returns status (*commit* or *abort*)
   - Cannot close subtransactions!
3. abort(id), cancel transaction id
4. openSub($trans$) $\rightarrow subTransId$
   - Opens a new subtransaction whose parent is $trans$ and returns a unique subtransaction identifier
5. getStatus($trans$)$\rightarrow committed$, $aborted$, $provisional$
   - Asks coordinator to report status of trans

Subtransactions get restricted interface

- Only openSub() and getStatus() available

Subtransactions Facts

- Are atomic to their parent
  - with respect to failures and,
  - concurrent access
- When at the same level can run concurrently
  - Need to be serialized if there are shared objects
- They can fail independently from other subtransactions
  - Parent transactions can choose to open alternative subtransactions,
  - if several subtransactions fail, parent could tolerate this and commit,
  - and then all successful child transactions would commit too.

Flat versus Nested Transactions

Flat Transactions

All work done at same level between open and a commit or abort result.

Example: deliver mail message to list of recipients

- openSub() transaction to deliver each message

Flat versus Nested Transactions

Flat Transactions

All work done at same level between open and a commit or abort result.

Example: deliver mail message to list of recipients

- openSub() transaction to deliver each message

Question!

Consider the case that there is a typo in exactly one of the addresses, right in the middle of the list. What of the following outcomes are possible when we use a nested transaction to implement the above?

(A): Only one of the recipients is sent a message

(B): All recipients receive their message

(C): Only half of the recipients are sent a message

(D): All but one of the recipients are sent a message

Flat versus Nested Transactions

Flat Transactions

All work done at same level between open and a commit or abort result.

Example: deliver mail message to list of recipients

- openSub() transaction to deliver each message

Question!

Consider the case that there is a typo in exactly one of the addresses, right in the middle of the list. What of the following outcomes are possible when we use a nested transaction to implement the above?

(A): Only one of the recipients is sent a message

(B): All recipients receive their message

(C): Only half of the recipients are sent a message

(D): All but one of the recipients are sent a message

→ (B & D): Depending on what the parent transaction does, both outcomes possible.

Advantages of Nested Transactions

With respect to flat transactions

Advantages of Nested Transactions

With respect to flat transactions

$\rightarrow$ higher degree of concurrency and operations can be run in parallel

Example Illustrating Impact of Concurrency

Consider $branchTotal()$ method in $Account$

- implemented invoking $getBalance()$ for every account in branch,
- above run in parallel when accounts distributed across network of servers,
- runtime $= \max$(runtime of subtransactions), rather than $\sum$

Sub-transactions can commit or abort independently

Advantages of Nested Transactions

With respect to flat transactions

→ higher degree of concurrency and operations can be run in parallel

Example Illustrating Impact of Concurrency

Consider $branchTotal()$ method in $Account$

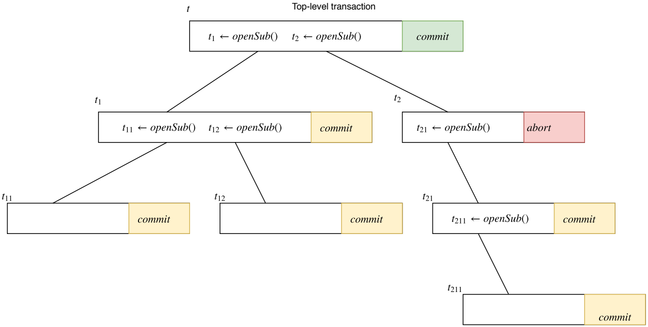
- implemented invoking $getBalance()$ for every account in branch,
- above run in parallel when accounts distributed across network of servers,
- runtime $= \max$(runtime of subtransactions), rather than $\sum$

Sub-transactions can commit or abort independently

→ more robust (potentially, depends on comms etc.),

→ failure handling is more flexible.

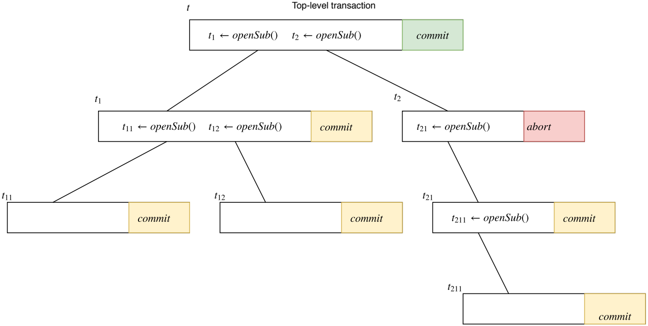
Commits for Nested Transactions are Tricky - I



1. Transactions can only commit or abort once children transactions completed.
2. When subtransaction completes, it commits provisionally or aborts. The later is final.

Commits for Nested Transactions are Tricky - II

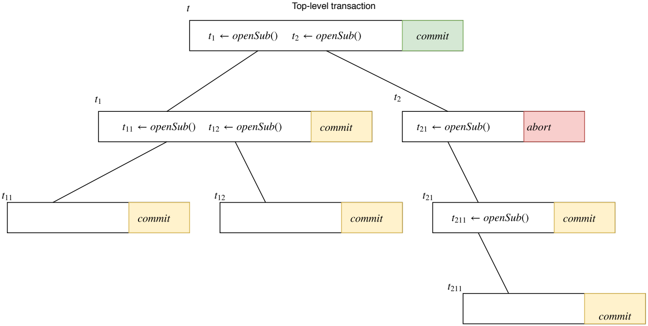
3. When parent aborts, all children are aborted.



If t_2 aborts, t_{21} and t_{211} must also abort, overruling their (provisional) commits.

Commits for Nested Transactions are Tricky - III

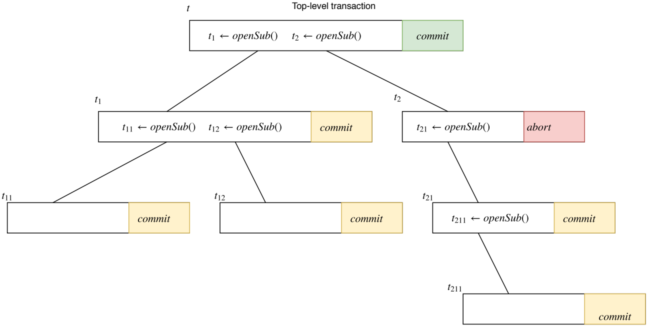
4. When a subtransaction aborts, the parent is free to abort or not.



Above, t has decided to commit, even if t_2 has aborted.

Commits for Nested Transactions are Tricky - IV

5. When top-level transaction commits , all subtransactions that have provisionally committed can commit
   - → If and only if no ancestor has aborted.



t's commit allows t_1, t_{11} and t_{12} to commit, but not t_{21} and t_{211} since t_2 aborted.

When to Use Nested Transactions?

In a single processor environment they are overkill... otherwise always

ACID is a very useful set of guarantees to have

- goes well beyond Database Management Systems (DBMS),
- guranteeing ACID ongoing research problem for Cyberphysical Systems (CPS) verification and control

Challenges in CPS (and other types of DS with similar requirements):

- Implementation for specific tasks provided by embedded systems that are sourced *off-the-shelf* and often proprietary,
- each embedded system has its own computing resources, but limited,
- failure modes can be very diverse, flexibility in error handling fundamental.

Agenda

Definitions

Server

A server is a process p_s that manages a number of shared objects, and is responsible for permanent storage and failure recovery.

Client

A client is a process p that executes a number of transactions over the shared objects hosted by server p_s.

Distributed Transaction

A *flat* or *nested* transaction is distributed when executed by a client over shared objects hosted by more than one server p_s.

Overview of Distributed Transactions

Atomicity

- All servers involved either commit or abort,
- one server adopts role of coordinator and ensures all other servers reach same decision,
- many protocols possible, so–called two–phase commit (2PC) is widely used.

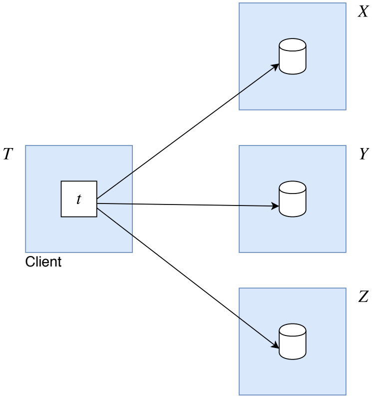
Concurrency Control

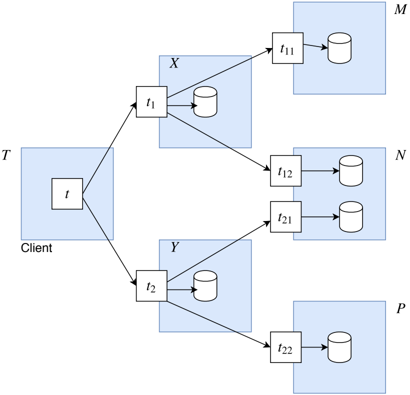
- Transactions must be globally serialized,
- even if serialization can be local to servers deadlocks are possible.

Recovery

- All shared objects involved must be recoverable,
- enforcing isolation and durability is challenging.

Illustration: Flat and Nested Distributed Transactions





Flat transaction:

- Accesses objects sequentally
- When locking used, at most waits for one object

Nested transaction

- Accesses objects concurrently
- t_1 and t_2 make remote invocations on different servers, can run in parallel

Running Example: Banking

Account interface

$deposit(amount)$

deposit $amount$ in account

$withdraw(amount)$

withdraws $amount$ from account

$getBalance() \rightarrow amount$

returns account $balance$

$setBalance(amount)$

sets $balance$ to $amount$

Branch interface

$create(name) \rightarrow account$

create $account$, labeled with $name$

$lookUp(name) \rightarrow account$

returns ref to $account$ with label $name$

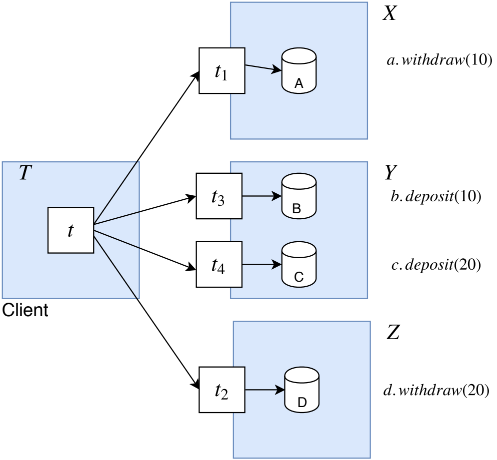
$branchTotal() \rightarrow amount$

returns $\sum_a a.balance$, a in set of accounts

- Two remote object interfaces, distributed amongst several processes
- A server holds every $Account$ and $Branch$ object
- All methods get extra argument t, identifying transaction (we omit this most of the time for brevity)

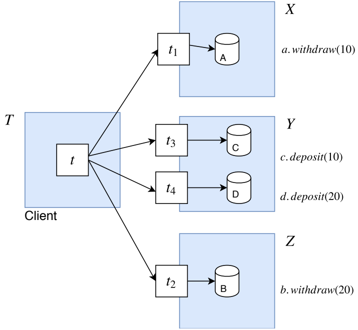
Banking Distributed

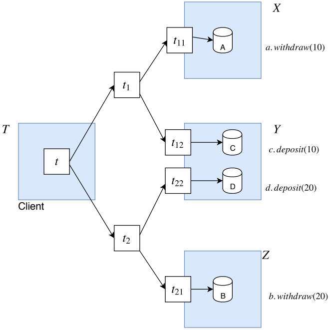
$t \leftarrow open()$
$t_1 \leftarrow openSub()$
$a.withdraw(10)$
$t_2 \leftarrow openSub()$
$d.withdraw(20)$
$t_3 \leftarrow openSub()$
$b.deposit(10)$
$t_4 \leftarrow openSub()$
$c.deposit(20)$
$close(t)$



Increased parallelism, but increased robustness?

Throughput and Robustness





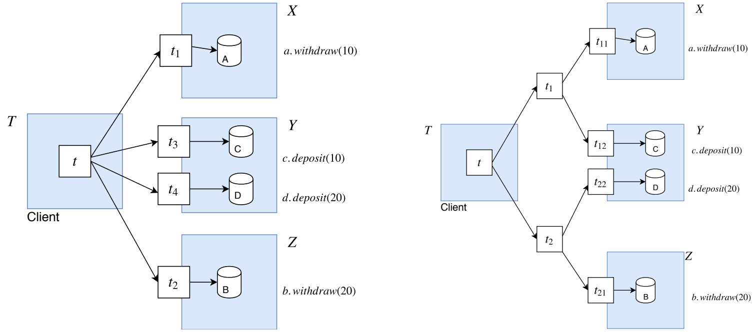
Question!

Which of the two nested transaction maximizes throughput and robustness?

(A): Left

(B): Right

Throughput and Robustness



Question!

Which of the two nested transaction maximizes throughput and robustness?

(A): Left (B): Right

→ (B): Because (1) if one of the subtransactions fail, still one of the goals of the top-level transaction is achieved, and (2) does not require locking and there is no overlap between subtransactions read and write sets.

Coordination in Distributed Transactions

Idea #1: Coordinators & Workers

One server becomes coordinator, all other processes become workers/participants.

Example protocols to elect coordinators

- If one server per transaction, first (if flat) or top–level (if nested),
- or use election DA, based on consensus (MongoDB 3.4+),

Too Cool for School

Blockchain protocols do away with the need to select a coordinator.

Idea #2: Transaction Identifiers

Transactions globally identifiable,identifiers made of two parts

$$(server\,global\,identifier,\ server\,sequence\,number)$$

Roles and Responsibilities in Distributed Transactions

Coordinator Responsibilities

1. Maintains list of participating servers
2. Collects info from workers needed for commit
3. Responsible to commit or abort the transaction.

Workers (Participants)

1. Knows the coordinator and keeps reference to it
2. Notifies coordinator to join transaction
3. Reports local result (provisional commit, abort) to the coordinator and follows its final decision
4. Notifies the coordinator it is aborting

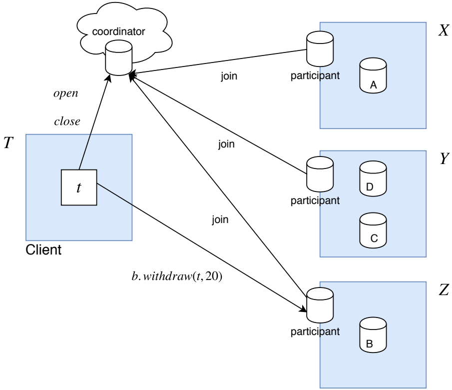
Coordinator Interface with Distributed Transactions Support

Coordinator interface

1. start(), returns unique identifier $id = (serverId, seqNumber)$ made of two parts
   - server unique identifier $serverId$ and,
   - local sequence number $seqNumber$.
2. close(id), finalize transaction id, returns status (*commit* or *abort*)
3. abort(id), cancel transaction id
4. openSub() returns unique identifier id
5. getStatus() returns status of transaction
6. join(id, ref), informs coordinator a new participant (ref) joined transaction id

Coordination of Distributed Banking Transaction - I

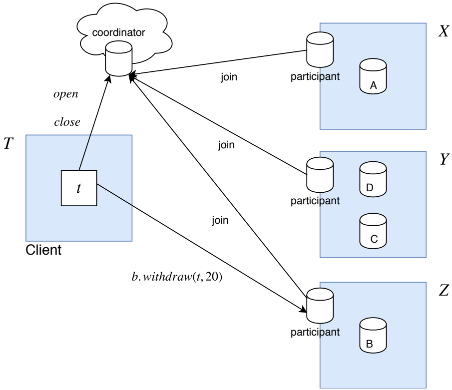
Five servers X, Y, Z, T and the cloud.



1. Servers X, Y and Z keep a participant object, which joins the transaction, when prompted to do so.

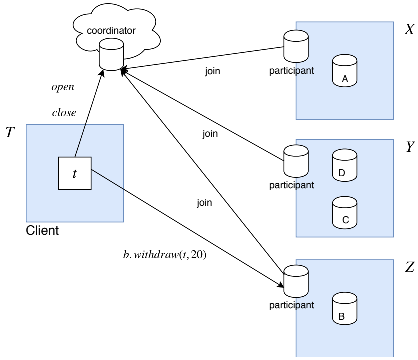
Coordination of Distributed Banking Transaction - II

Five servers X, Y, Z, T and the cloud.



2. When $b.withdraw(t, 20)$ is executed, the participant at server Z (that is hosting b) joins, knows coordinator thanks to transaction id t

Crash Failures



Question!

Server Y crashes after depositing money on account d (but did so correctly for c). What are we guaranteed with respect the state of shared objects?

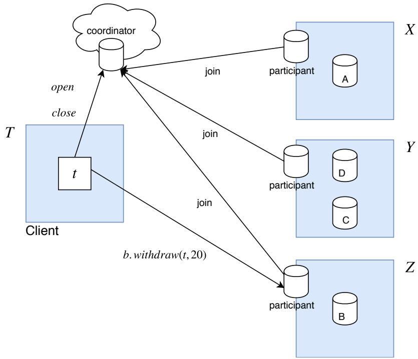
(A): Depends on transaction design.

(B): Some work will have been wasted.

(C): All objects revert to original states.

(D): Y will have notified *coordinator* of the crash.

Crash Failures



Question!

Server Y crashes after depositing money on account d (but did so correctly for c). What are we guaranteed with respect the state of shared objects?

(A): Depends on transaction design.

(B): Some work will have been wasted.

(C): All objects revert to original states.

(D): Y will have notified *coordinator* of the crash.

$\rightarrow$ (A & C): Robustness really depends on availability of servers and design. At a minimum, we have atomicity & recoverability guarantees.

Agenda

Atomic Commit is a Fundamental Problem in DS

Atomic commitment problem (ACP) [Babaoglu & Toueg]

- Ensure a globally consistent transaction despite failures
- Decision is based on agreement among all participants
  - Commit: all participants make the transaction's update permanent
  - Abort: none will

Properties

- All participants that decide reach the same decision
- If any participant decides commit, then all participants must have voted "yes"
- If all participants vote yes and no failure occur, then all participants decide commit
- Each participant decides at most once (i.e. decision is not reversible)

The Atomic Commit Protocol (ACP) Problem

Conditions

Validity

- If a coordinator broadcasts a message m, then all participants eventually receive m

Integrity

- For any message m, each participant receives m at most once and only if a coordinator actually broadcasts m

Timeliness (only for synchronous systems)

- There is a known constant d (delay) such that a broadcast of m initiated at time t, is received by every participant by $t + d$

One-Phase Commit Protocol

Reminder

Transactions come to an end when client requests commit or abort.

One-Phase Commit (1PC) Protocol

1. Coordinator receives request from client
2. Coordinator broadcasts message indicating commit or abort
3. When all participants have sent back acknowledgment exit, otherwise broadcast instruction again

One-Phase Commit Protocol

Reminder

Transactions come to an end when client requests commit or abort.

One-Phase Commit (1PC) Protocol

1. Coordinator receives request from client
2. Coordinator broadcasts message indicating commit or abort
3. When all participants have sent back acknowledgment exit, otherwise broadcast instruction again

Limitations of 1PC

Coordinator cannot decide to abort when client requests commit

- Concurrency control requires abort (deadlock, failed validation)
- Server may have crashed and been restarted (which conveys to abort the transaction)

Agenda

Further Reading

Coulouris et al. *Distributed Systems: Concepts & Design*

- Chapter 16.3 for Nested Transactions
- Chapter 17.1, 17.2 and 17.3 for Distributed Transactions and Atomic Commit Protocols