# COMP90020: Distributed Algorithms

## 9. Transactions: Design Patterns for Concurrency Control

Serialising Access to Data

Miquel Ramirez

THE UNIVERSITY OF MELBOURNE

Semester 1, 2019

# Agenda

# Agenda

# What are Transactions?

## The Problem

Ensure all shared objects managed by DS are consistent

- objects stored in either volatile or persistent memory,

## Transactions are a *design pattern* for DS

A *transaction* is a sequence of operations on shared objects guaranteed to be atomic

- DS guarantees operations to be completed and effects recorded if no failures
- otherwise changes on objects are discarded.

Algorithms implementing transactions can be expected to handle

- Crash failures, Omission failures, and some Byzantine failures.

# 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

Transactions are *all about* synchronization!

# Illustrating Race Conditions

Concurrent processes $p_i$, $p_j$ can interfere with each other

Init both $p_i$ and $p_j$ have references to same $Account$ remote object,

$t = 1$ $p_i$ **send**$(deposit, amount)$

$t = 2$ server **receive**$(p_i, deposit, amount)$,

$t = 3$ $p_j$ **send**$(withdraw, amount)$,

$t = 4$ server **receive**$(p_i, deposit, amount)$

$t = 4$ server **receive**$(p_j, withdraw, amount)$

Operations in messages not commutative, as overdraft results in extra fees.

Server lacks context to interpret received operations.

# Illustrating Race Conditions

Concurrent processes $p_i$, $p_j$ can interfere with each other

Init both $p_i$ and $p_j$ have references to same $Account$ remote object,

- $t = 1$ $p_i$ **send**$(deposit, amount)$
- $t = 2$ server **receive**$(p_i, deposit, amount)$,
- $t = 3$ $p_j$ **send**$(withdraw, amount)$,
- $t = 4$ server **receive**$(p_i, deposit, amount)$
- $t = 4$ server **receive**$(p_j, withdraw, amount)$

Operations in messages not commutative, as overdraft results in extra fees.

Server lacks context to interpret received operations.

Operations can arrive in an unfortunate order...

# Concurrent Programming Techniques

Basic primitive: use `synchronized` blocks (C++, C#, JAVA)

- Compiler generates code that guarantees sequential execution,
- uses built–in, off–the–shelf mutual exclusion (mutex) mechanisms (semaphores, locks, etc.)
- processes (threads) accessing `synchronized` not allowed to execute concurrently.

Design system to ensure atomic and isolated operations

- Via `synchronized` or clever design avoid interference,

# Concurrent Programming on a Good Day

Concurrent processes $p_i$, $p_j$ correctness depends on order of operations

Lucky case, no network congestion

Init both $p_i$ and $p_j$ have references to same $Account$ remote object,

- $t = 1$ $p_i$ **send**$(deposit, amount)$, $p_i$ notifies $p_j$ deposit made
- $t = 3$ server **receive**$(p_i, deposit, amount)$,
- $t = 4$ $p_j$ **send**$(withdraw, amount)$,
- $t = 5$ server **receive**$(p_j, withdraw, amount)$

Happy customer

# Concurrent Programming on a Bad Day

Concurrent processes $p_i$, $p_j$ correctness depends on order of operations

Server network is congested

Init both $p_i$ and $p_j$ have references to same $Account$ remote object,

- $t = 1$ $p_i$ **send**$(server, deposit, amount)$,
- $t = 2$ $p_i$ **send**$(p_j, transferred, amount)$
- $t = 3$ $p_j$ **receive**$(p_j, transferred, amount)$
- $t = 4$ $p_j$ **send**$(withdraw, amount)$,
- $t = 6$ server **receive**$(p_j, withdraw, amount)$, fails because no funds
- $t = 7$ server **receive**$(p_i, deposit, amount)$

Angry customer

# Computational Dependencies and Fair Sharing of Resources

## Metaphor: Producers & Consumers

Processes can adopt two roles: producers ($p_i$) or consumers ($p_j$)

- $p_j$ (consumer) needs to wait for $p_i$ (producer) operations on shared objects to complete.

Serialization of *concurrent* operations via locks is classical approach

- $p_i$ sets up lock on $Account$ obj, $p_j$ waits for $p_i$ to release,
- *somebody* notifies $p_j$ when $p_i$ releases the lock

Serialization is complex but is more efficient than polling

$\rightarrow$ which is potentially unfair to *unlucky* processes

# Failure Model

Special case of Byzantine setting due to Lampson (1981)

### Write Failures

- Nothing gets written or wrong value is changed
- Dodgy writes can be detected checking checksums

### Process Crashes

- When rebooted process starts with blank slate.
- Crashing a faulty process better than letting it go feral.

### Arbitrary Delays in Message Delivery

- Messages lost, duplicated or corrupted
- Corrupt messages can be detected checking checksums

# Question: Centralized vs Decentralized Concurrency Control

## Question!

**We got $4$ processes doing some distributed computation, over several shared objects. We need to decide whether it will be better to manage concurrency control in a distributed fashion, under the failure model we just discussed, or a centralised one. We decide for a distributed solution, that relies on the Interactive Consistency algorithm to keep track of the states of locks throughout the DS. What are the possible issues with this decision?**

(A): Network congestion due to high volume of message exchanges

(B): Single point of failure

(C): Vulnerable to faulty processes

(D): Difficult for humans to fathom

# Question: Centralized vs Decentralized Concurrency Control

## Question!

**We got $4$ processes doing some distributed computation, over several shared objects. We need to decide whether it will be better to manage concurrency control in a distributed fashion, under the failure model we just discussed, or a centralised one. We decide for a distributed solution, that relies on the Interactive Consistency algorithm to keep track of the states of locks throughout the DS. What are the possible issues with this decision?**

(A): Network congestion due to high volume of message exchanges

(B): Single point of failure

(C): Vulnerable to faulty processes

(D): Difficult for humans to fathom

→ (A): the exponential bound on messaging is still there.

→ (B): no such a thing, stability of DS does not depend on a single process.

→ (C): IC is built on top of Byzantine Generals and/or Consensus, algorithms require bounds on faulty processes.

→ (D): interplay between communication delays, message passing and crashes makes very hard for humans to predict the evolution of DS over time.

# Question: Centralized vs Decentralized Concurrency Control

## Question!

**We got $4$ processes doing some distributed computation, over several shared objects. We need to decide whether it will be better to manage concurrency control in a distributed fashion, under the failure model we just discussed, or a centralised one. We decide to go for a centralized solution, relying on RTO multicast for the server to propagate the states of locks throughout the DS. What are the possible issues with this decision?**

- (A): Latency blowouts
- (B): Single point of failure
- (C): Vulnerable to faulty processes
- (D): Difficult for humans to fathom

# Question: Centralized vs Decentralized Concurrency Control

**Question!**

**We got $4$ processes doing some distributed computation, over several shared objects. We need to decide whether it will be better to manage concurrency control in a distributed fashion, under the failure model we just discussed, or a centralised one. We decide to go for a centralized solution, relying on RTO multicast for the server to propagate the states of locks throughout the DS. What are the possible issues with this decision?**

- (A): Latency blowouts
- (B): Single point of failure
- (C): Vulnerable to faulty processes
- (D): Difficult for humans to fathom

→ (A): the exponential bound on messaging is still there.
→ (B): the server is the single point of failure.
→ (C): as long as the server process does not crash, we will be fine.
→ (D): centralised systems are generally easier to debug and maintain.

# Agenda

# Running Example

## Keeping Your Accounts In Order

Processes $p_1$ hosting transaction $t$, $p_2$ hosting transaction $s$, with three instances $Account$ objects $a$, $b$, $c$

| $Time$ | $t$ | $s$ |
|---|---|---|
| 0 | $a.withdraw(100)$ | – |
| 1 | – | $c.deposit(100)$ |
| 2 | $c.withdraw(200)$ | – |
| 3 | – | $c.deposit(200)$ |
| 4 | $b.withdraw(50)$ | $a.deposit(50)$ |

Sequences of client requests must be

- Free from *interference*
- value of $balance$ is deterministic

# ACID Transactions

Forces on the *design pattern*

1. Atomicity, guarantee exactly two possible outcomes
   - All ops successful, value of $balance$ recorded on server,
   - otherwise, no effect on the value of $balance$ on server
2. Consistency
   - Value of $balance$ is deterministic,
   - additional ops may be required to return objects to initial states if transaction fails.
3. Isolation
   - Multiple transactions on server do do not interfere with each other,
4. Durability
   - If transaction succeeds, all effects saved in permanent storage

# Consequences of ACID Transactions

Storage requirements increase

- Need to store initial state of every object involved.
- To ensure isolation we need to keep multiple copies of an object.

Complexity of enforcing ACID is high

## *Coordinator* interface

# Consequences of ACID Transactions

Storage requirements increase

- Need to store initial state of every object involved.
- To ensure isolation we need to keep multiple copies of an object.

Complexity of enforcing ACID is high

## *Coordinator* interface

1. Start(), returns unique identifier $id$
   - Client proc uses $id$ with each request in the transaction
2. Close($id$), finalize transaction $id$, returns status (*commit* or *abort*)
   - Commit indicates transaction successful, coordinator guarantees shared objects saved,
   - Abort flags that it has not been completed.
3. Abort($id$), cancel transaction $id$
   - Coordinator guarantees *temporary effects* invisible to other transactions

# Implementing ACID Transactions

## Atomicity

- How to recover from aborts?

## Consistency

- How to deal with failures?
- How to deal with asynchronous message passing and processes with varying speeds?

## Isolation

- How to avoid interference?
- How we ensure the DS does not end deadlocked?

## Durability

- How to avoid writing *too early* results of transactions?

# Question: "Dirty Reads"

| Time | $t$ | | $s$ | |
|---|---|---|---|---|
| 0 | $start$ | – | – | – |
| 1 | $x \leftarrow b.getBalance()$ | 100 | $start$ | – |
| 2 | $b.setBalance(x + 10)$ | 110 | – | – |
| 3 | – | – | $x \leftarrow b.getBalance()$ | 110 |
| 4 | – | – | $b.setBalance(x + 20)$ | 130 |
| 5 | – | – | $commit$ | 130 |
| 6 | $abort$ | 100 | – | ?? |

## Question!

**How we can handle this situation?**

- (A): Delay $s$ commit
- (B): Delay commit and abort $s$
- (C): We cannot avoid this
- (D): Block $s$ until $t$ commits or aborts

# Question: "Dirty Reads"

| Time | $t$ | | $s$ | |
|---|---|---|---|---|
| 0 | $start$ | – | – | – |
| 1 | $x \leftarrow b.getBalance()$ | 100 | $start$ | – |
| 2 | $b.setBalance(x+10)$ | 110 | – | – |
| 3 | – | – | $x \leftarrow b.getBalance()$ | 110 |
| 4 | – | – | $b.setBalance(x+20)$ | 130 |
| 5 | – | – | $commit$ | 130 |
| 6 | $abort$ | 100 | – | ?? |

## Question!

**How we can handle this situation?**

(A): Delay $s$ commit

(B): Delay commit and abort $s$

(C): We cannot avoid this

(D): Block $s$ until $t$ commits or aborts

→ (A): Delaying the commits is not enough, we want $s$ to finish.

→ (B): Delaying and aborting $s$ introduces the possibility of cascading aborts, as we would need to abort other transactions reading from objects $s$ is writing to. This can be quite problematic.

→ (D): This is safe, but also pessimistic (reducing throughput).

# Question: "Premature Writes"

| t | $t$ | | $s$ | |
|---|---|---|---|---|
| 0 | $start$ | – | – | – |
| 1 | $x \leftarrow a.getBalance()$ | 100 | $start$ | – |
| 2 | $b.setBalance(x + 10)$ | 110 | – | – |
| 3 | – | – | $x \leftarrow b.getBalance()$ | 110 |
| 4 | – | – | $b.setBalance(x + 20)$ | 130 |
| 2 | – | – | $abort$ | ?? |
| 3 | $abort$ | 100 | – | ?? |

## Question!

**Both transactions abort. What should be the final value of $b.balance$ according to ACID?**

(A): $100$ (B): $110$

(C): $42$ (D): $130$

# Question: "Premature Writes"

| t | $t$ | | $s$ | |
|---|---|---|---|---|
| 0 | $start$ | – | – | – |
| 1 | $x \leftarrow a.getBalance()$ | 100 | $start$ | – |
| 2 | $b.setBalance(x+10)$ | 110 | – | – |
| 3 | – | – | $x \leftarrow b.getBalance()$ | 110 |
| 4 | – | – | $b.setBalance(x+20)$ | 130 |
| 2 | – | – | $abort$ | ?? |
| 3 | $abort$ | 100 | – | ?? |

## Question!

**Both transactions abort. What should be the final value of $b.balance$ according to ACID?**

(A): 100 (B): 110

(C): 42 (D): 130

→ (A): Transactions need to be all or nothing – we cannot leave side effects of aborted transactions lingering.

→In general we want to delay further reads and writes to shared objects and ongoing transaction is writing to or reading from.

# Review: Atomic Operations on Shared Objects

Internal events at processes $p_i$ read or write to *local var*

- Comm channels can be *abstracted* by having shared objects $x$
- Procs $p_i$ have all access to $x$, via atomic *read-modify-write* ops

## Atomic Read-Modify-Write Ops

Typical hardware enabled atomic read-modify-write ops:

- test-and-set: *writes* $\top$ to Boolean var, returns previous value
- get-and-increment: *increases* integer var by $1$, returns previous value
- get-and-set($new$): *writes* $new$ in var, returns previous value
- compare-and-set($old$,$new$): if var $= old$, then set var to $new$ and return $\top$

Note: single reads and writes are always atomic

# Agenda

# Lost Updates

| Time | $t$ | | $s$ | |
|---|---|---|---|---|
| 0 | $start$ | – | – | – |
| 1 | $x \leftarrow b.getBalance()$ | 200 | $start$ | – |
| 2 | – | – | $y \leftarrow b.getBalance()$ | 200 |
| 3 | – | – | $b.setBalance(1.1y)$ | 220 |
| 4 | $b.setBalance(1.1x)$ | 220 | – | – |
| 5 | $a.withdraw(0.1x)$ | – | – | – |
| 6 | – | – | $c.withdraw(0.1y)$ | 280 |

- Initially $a.balance = 100$, $b.balance = 200$ and $c.balance = 300$.
- Balance of $b$ is not correct, as $t$ overwrites the changes made by $s$

# Inconsistent Retrievals

$s$ calculates *total balance* in accounts $a$, $b$ and $c$, while $t$ transfers money from $a$ to $b$ (initially all balances are $200$)

| Time | $t$ | | $s$ | |
|---|---|---|---|---|
| 0 | $start$ | – | $u \leftarrow start$ | – |
| 1 | $a.withdraw(100)$ | $100$ | – | – |
| 2 | – | – | $total \leftarrow a.getBalance()$ | $100$ |
| 3 | – | – | $total \leftarrow total + b.getBalance()$ | $300$ |
| 4 | – | – | $total \leftarrow total + c.getBalance()$ | $500$ |
| 5 | $b.deposit(100)$ | $300$ | – | – |

- $s$ comes $100$ short since $t$ transaction was still ongoing...

# Serial Equivalence and Scheduling

## Serializing Transactions = Constrain Schedule of Operations Over Time

Let $s$ and $t$ be *transactions*, no a priori order between them,

- if there are no conflicting operations, these can be reordered arbitrarily e.g. $s$ then $t$, $t$ then $s$ or any interleaving of operations,
- otherwise all pairs of conflicting operations must be executed in the same order on all objects being accessed concurrently and for every possible schedule.

## Conflicting Operation

Let $s$ and $t$ be *transactions* with ops $\alpha(o) \in s$ and $\beta(o) \in t$ over objects $o$, let $\mathcal{R}(o)$ and $\mathcal{W}(o)$ be sets of ops that respectively read and write the value of an object $o$. Then for every pair $\langle \alpha(o),\ \beta(o) \rangle$

- if both in $\mathcal{R}$, then operations do not conflict,
- when at least one in $\mathcal{W}$ then operations are conflicting.

# Fixing Inconsistent Retrievals

$t$ and $u$ are conflicting over $withdraw$, $deposit$ and $balance$ operations

| Time | $t$ | | $s$ | |
|---|---|---|---|---|
| 0 | $start$ | – | $start$ | – |
| 1 | $a.withdraw(100)$ | 100 | – | – |
| 2 | – | – | $total \leftarrow a.getBalance()$ | 100 |
| 3 | – | – | $total \leftarrow total + b.getBalance()$ | 300 |
| 4 | – | – | $total \leftarrow total + c.getBalance()$ | 500 |
| 5 | $b.deposit(100)$ | 300 | – | – |

We have *two* serially equivalent schedules, that order operations as follows...

# Fixing Inconsistent Retrievals #1

$t \prec u$, *Coordinator* delays execution of $u$ until $t$ is finished

| Time | $t$ | | $u$ | |
|---|---|---|---|---|
| 0 | $start$ | – | – | – |
| 1 | $a.withdraw(100)$ | 100 | – | – |
| 2 | $b.deposit(100)$ | 300 | – | – |
| 3 | – | – | $start$ | – |
| 4 | – | – | $total \leftarrow a.getBalance()$ | 100 |
| 5 | – | – | $total \leftarrow total + b.getBalance()$ | 300 |
| 6 | – | – | $total \leftarrow total + c.getBalance()$ | 600 |

**Question!**

**Is there a similar, more efficient schedule than the above?**

(A): Yes, $u$ could start at $Time = 1$

(B): No

# Fixing Inconsistent Retrievals #2

Or the other way around, *Coordinator* enforces $u \prec t$, delaying $t$

| Time | $t$ | | $u$ | |
|---|---|---|---|---|
| 0 | – | – | $start$ | – |
| 1 | – | – | $total \leftarrow a.getBalance()$ | 100 |
| 2 | – | – | $total \leftarrow total + b.getBalance()$ | 300 |
| 3 | – | – | $total \leftarrow total + c.getBalance()$ | 600 |
| 4 | $start$ | – | – | – |
| 5 | $a.withdraw(100)$ | 100 | – | – |
| 6 | $b.deposit(100)$ | 300 | – | – |

**Question!**

**Is there a similar, more efficient schedule than the above?**

(A): Yes, $t$ could start $Time = 2$ (B): No

# Question: Fixing Lost Updates

| time | $t$ | | $u$ | |
|---|---|---|---|---|
| 0 | $start$ | – | – | – |
| 1 | $x \leftarrow b.getBalance()$ | 200 | $start$ | – |
| 2 | – | – | $y \leftarrow b.getBalance()$ | 200 |
| 3 | – | – | $b.setBalance(1.1y)$ | 220 |
| 4 | $b.setBalance(1.1x)$ | 220 | – | – |
| 5 | $a.withdraw(0.1x)$ | 80 | – | – |
| 6 | – | – | $c.withdraw(0.1y)$ | 280 |

## Question!

**Which of the following are valid possible ways of ensuring that $t$ and $u$ are serially equivalent?**

(A): enforce $t_3 \prec s_1$ for all schedules

(B): $t \prec u$

(C): $u \prec t$

(D): All of them

# Question: Fixing Lost Updates

| time | $t$ | | $u$ | |
|---|---|---|---|---|
| 0 | $start$ | – | – | – |
| 1 | $x \leftarrow b.getBalance()$ | 200 | $start$ | – |
| 2 | – | – | $y \leftarrow b.getBalance()$ | 200 |
| 3 | – | – | $b.setBalance(1.1y)$ | 220 |
| 4 | $b.setBalance(1.1x)$ | 220 | – | – |
| 5 | $a.withdraw(0.1x)$ | 80 | – | – |
| 6 | – | – | $c.withdraw(0.1y)$ | 280 |

## Question!

**Which of the following are valid possible ways of ensuring that $t$ and $u$ are serially equivalent?**

(A): enforce $t_3 \prec s_1$ for all schedules

(B): $t \prec u$

(C): $u \prec t$

(D): All of them

$\rightarrow$ (D): All are serially equivalent schedules, $a.withdraw(0.1x)$ and $c.withdraw(0.1y)$ can be executed in parallel as they do not interfere.

# Making Serial Equivalence Happen

Serial Equivalence is a crucial property, yet optimal scheduling hard to achieve

→ "real–time" ACID requires to trade–off throughput for latency

Naive solution

- *Coordinator* waits for processes to indicate end of transaction,
- FIFO: first transaction $t$ completed, is the first executed,
- objects written or read by $t$ covered with a `synchronized` block.

We can increase parallelism (hopefully, performance too) increasing complexity

- Fine–grained locks
  - ensure both fairness and efficiency
- Optimistic Concurrency Control
  - "easier to ask for forgiveness than for permission".

# Agenda

# Further Reading

Transaction Patterns: A Collection of Four Transaction-related Patterns
M. Grand, *Proceedings Conference on Pattern Languages of Programs*, 1999

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

- Chapter 16, Section 16.4, 16.5, 16.6