Transactions

Denis Miginsky

Basic transaction definition

A transaction is a sequence of actions with side-effects that turns the system from one **valid state** to another one and that must be either **completed entirely or not completed at all**.

Unlike a critical section, a transaction could be reverted and restarted when necessary.

Extended transaction definition: ACI(D)

ACID:

- Atomicity
- Consistency
- Isolation
- Durability

It is a requirements set to any transactional system and it could be considered as an extended definition for a transaction.

ACID: Atomicity

Atomicity: a transaction must be either completed entirely or not completed at all under any circumstances. It includes any kind of failures: exceptions, errors in code, power supply failures, or supernova explosions.

Example: a transaction is going to transfer a sum of money from one account to another. At the point where this sum is already withdrawn from the first account, it was found that the second one is unavailable. The sum has to be returned back to the first account even if the transaction's code doesn’t handle it explicitly.

ACID: Consistency

Consistency: a transaction has to bring the system from one **valid/consistent** state to another. It includes both technical validity and domain constraints. This requirement is not applicable to the intermediate state inside the transaction.
If the transaction's code tries to change a state to invalid (e.g. due to error in code), a transaction must be rolled back.

Example: a transaction tries to change a user's name to one that already exists within the system. This case is prohibited by the problem statement. This transaction must be rolled back despite that there are no technical problems (e.g. exceptions).

ACID: Isolation

Isolation: for multiple transactions executed within the system (concurrently possibly) the result must be equivalent to the **sequential** execution in **some order** of the same set of transaction.

In other words, a transaction can behave as it is the only one within the whole world and can't interfere with any other transaction.

Violation example: see lost update, more problems to go.

ACID: Durability

Durability: a transaction that is committed already must remain committed even in case of system failure or shutdown. It means that the changes must be applied to non-volatile memory (e.g. HDD).

Durability works together with other properties (ACI), i.e. if the transaction is not committed yet, but some changes are already applied at the point where a failure occurred, the transaction must either rolled back or continued after the system's restart. Other transaction can't see these intermediate (and potentially inconsistent) changes.

Durability is not applicable to in-memory transactions.

Example: an application is rewriting a large file to a file system and a power supply failure occurred. After an OS and a FS driver restarted and completed the recovery process, a file must be either of new or previous version, but not the combination of both.

Typical isolation problems

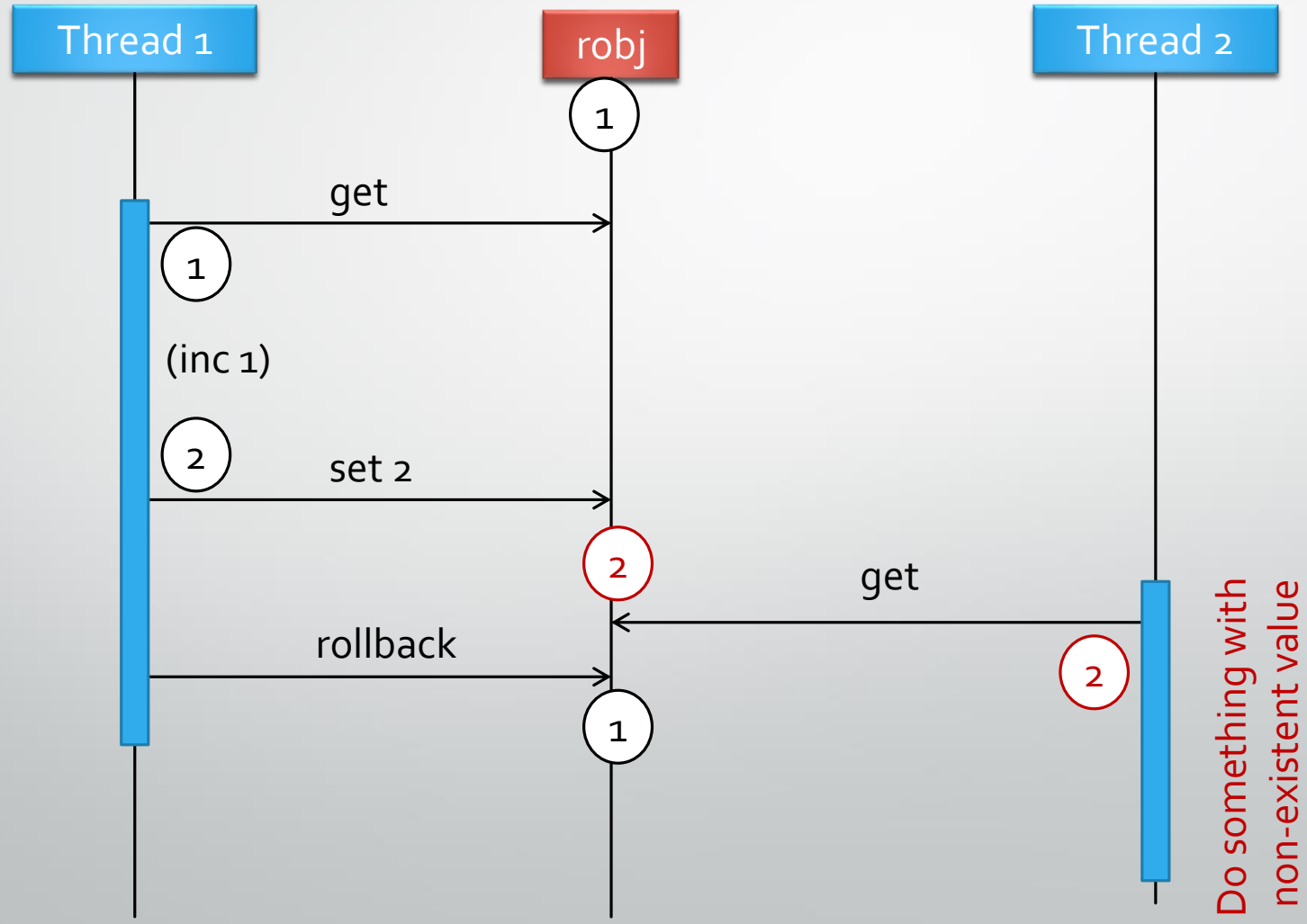
Transactions using the same resources can be completely isolated only at cost of almost total serialization (and lost of parallelism). This is how pessimistic transactions work (i.e. transactions based on critical sections).

Race condition – a result could depend on total order of operations in all transactions.
Optimistic transactions allow race conditions (and improve the level of parallelism by this way), but try to handle possible problems with them.

Typical problems:

- Lost update
- Dirty read
- Non-repeatable read

Dirty read: illustration



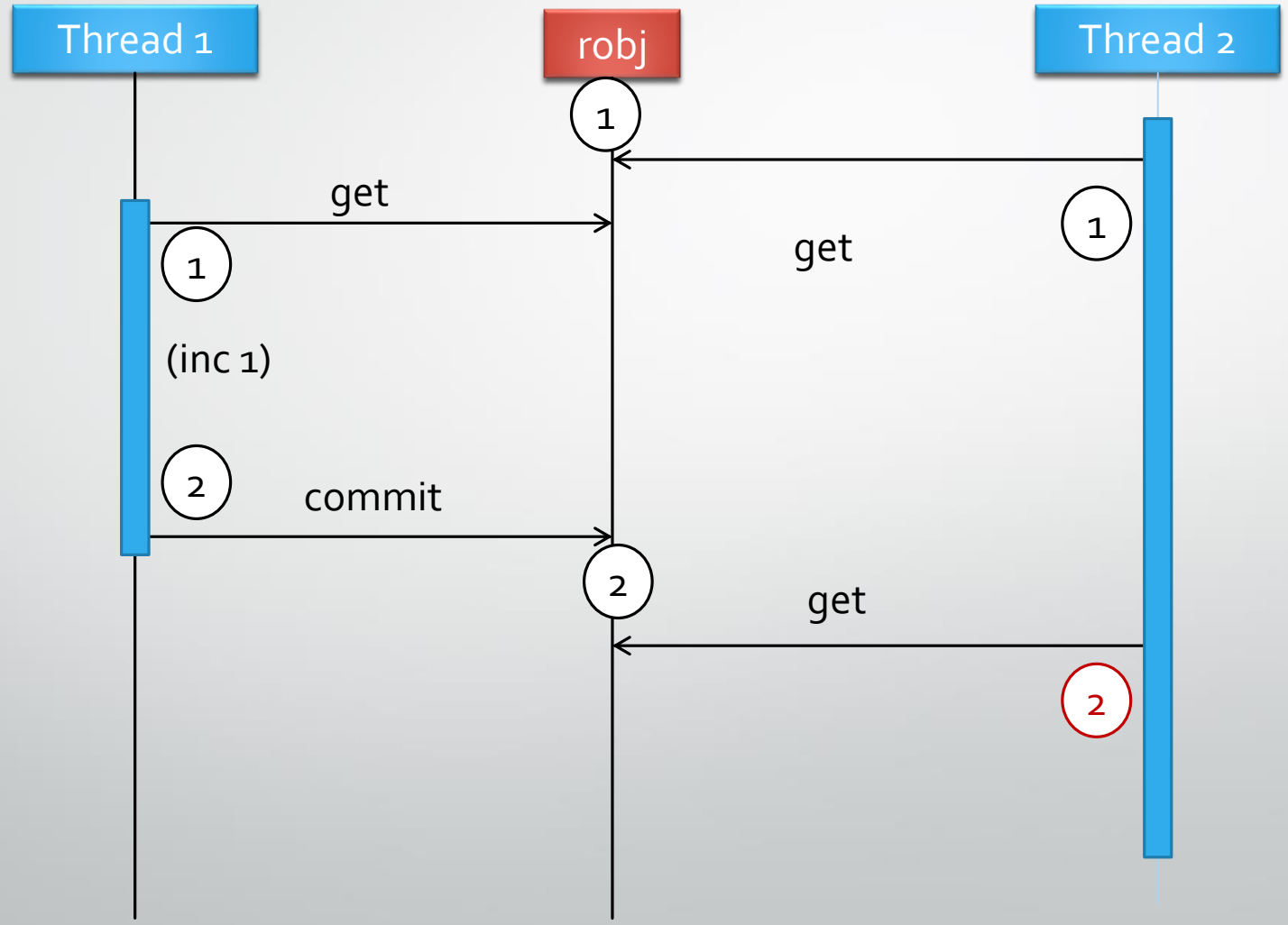
Dirty read

Problem: a transaction reads a resource that is changing simultaneously in another transaction and either has never been committed or will be changed further. The value is considered to be non-existent and can even violate consistency.

Solution (pessimistic transaction): total locking (including read locking) solves a problem

Solution (optimistic transaction): keep in-transaction changes isolated from other transactions.
How to achieve it?

Non-repeatable read: illustration



Non-repeatable read

Problem: a transaction reads a resource that subsequently changed and committed in another transaction. After that the initial transaction reads it once more and finds it changed. It violates isolation.

Solution (pessimistic transaction): total locking (including read locking) solves a problem

Solution (optimistic transaction): keep in-transaction read operations memoized. The transaction in such scenario can be committed (and logically occurs before interfered transaction) unless it changes the same resource.

Towards transactional engine implementation

A transaction itself is somewhere in-between imperative and functional programming.

~~Dark~~ Imperative side of the transaction:

- Mutable state

Functional side:

- All the mutable resources must support **undo** mechanism and must be under control
- All other side-effects are prohibited

Transaction vs critical section

Critical section	Transaction
Explicit locking, bad for SoC	Implicit locking if any, good for SoC
Unlimited side effects	Side effects are limited to resources managed by this transaction
Supported operations: run, try-run	Supported operations: run, commit, rollback
Doesn’t care about atomicity and consistency. Only isolation matters.	Cares about ACI.

Transactional Memory

A concurrency control is a mechanism that handles concurrency problems (e.g. critical sections, futures, agents, etc.)

A transactional memory is a concurrency control that operates in terms of transactions with manageable resources (represented as transactional references for example).

Software transactional memory (STM) could be implemented either as a framework or a part of programming language. First appearance in Haskell in the middle of 90th

Hardware Transactional Memory is supported by a hardware. First appearance in IBM BlueGene/Q processor in 2011.

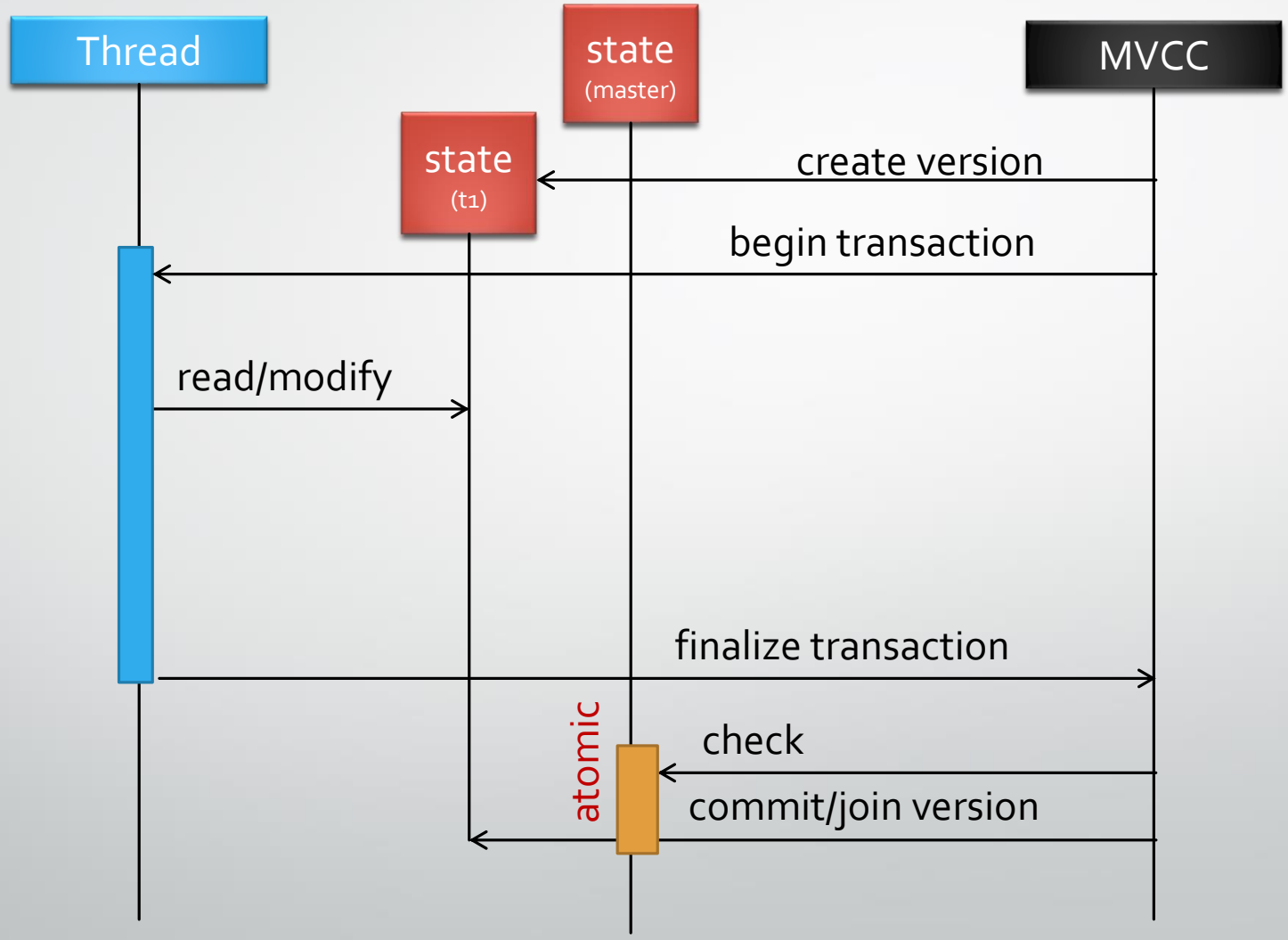
Multi-version concurrency control

A multi-version concurrency control (MVCC) is a transactional memory implementation using the optimistic approach.

In MVCC each transaction operates with its own version that is branched form the master state. The master state would be changed only on commit phase.
Conflicts must be detected on this phase and fixed somehow. The simplest way is just to restart a transaction.

MVCC could be considered as an implementation of copy-modify-merge concept.

STM/MVCC illustration



Transactional reference in Clojure

```
;;transactional references
(def mobj1 (ref 1))
(def mobj2 (ref 2))

(let [swap-fn
      (fn []
;;transaction
        (dosync
          (let [v1 @mobj1
                v2 @mobj2]
;;similar to reset! for atom
            (ref-set mobj1 v2)
            (ref-set mobj2 v1))))
;;the rest is on the right
```

```
;;continue with let definitions
      t1 (new Thread swap-fn)
      t2 (new Thread swap-fn)]
  (.start t1)
  (.start t2)
  (.join t1)
  (.join t2))
(println @mobj1 @mobj2)
;;>> 1 2
;;the result will always be the same
```

ref: API

```
;;create and initialise a transactional reference
;;validator is also allowed similar to atom
;;it will be checked on the transaction commit stage
(ref 1)
;;get the current value
(deref a)
@a
;;a transaction. refs can be modified inside the transactions only.
;;other side effects are not welcome (due to restarts, etc.)
(dosync & forms)
;;replace a value with a new one (similar to atom's reset!)
(ref-set a 2)
;;replace a value by passing a mutator function
;;similar to atom's swap!
(alter a (fn [x] (* x 2)))
;;limited and more efficient version of alter
;;will be discussed later
(commute a inc)
```

Task C5: dining philosophers

Solve the dining philosophers problem using Clojure's STM (see Java Concurrency task 9). Each fork must be represented by a **ref** with counter inside that shows how many times the given fork was in use.
A number of philosophers, length of thinking and dining periods and their number must be configurable.
Measure the efficiency of the solution in terms of overall execution time and number of transaction restarts (**atom** counter could be used for this).
Experiment with even and odd numbers of philosophers. Try to achieve live-lock scenario.

ref: dirty read attempt

```
(def cnt1 (ref 1))
(def cnt2 (ref 1))

(dosync
  (Thread/sleep 80)
  (println "t1(before): " @cnt2)
  (alter cnt1 inc)


  (Thread/sleep 100)
  (println "t1(after): " @cnt2)

  (Thread/sleep 100))

;;>>t1(before):  1
;;>>t2(before):  1
;;>>t1(after):  1
;;>>t2(after):  1
```

```
(dosync
 (Thread/sleep 100)


 (println "t2(before): " @cnt1)
 (alter cnt2 inc)
 (Thread/sleep 100)

 (println "t2(after): " @cnt1)
 (Thread/sleep 100))
```

ref: non-repeatable read attempt

```
(def cnt1 (ref 1))
(def cnt2 (ref 1))

(dosync
  (Thread/sleep 80)
  (println "t1(before): " @cnt2)
  (alter cnt1 inc)


  (Thread/sleep 100)
  (println "t1(after): " @cnt2)

  #_(Thread/sleep 100))

;;>>t1(before):  1
;;>>t2(before):  1
;;>>t1(after):  1
;;>>t2(before):  2
;;>>t2(after):  2
```

```
(dosync
 (Thread/sleep 100)


 (println "t2(before): " @cnt1)
 (alter cnt2 inc)
 (Thread/sleep 100)

 (println "t2(after): " @cnt1)
 #_(Thread/sleep 100))
```

Concurrent changes

```
(def cnt1 (ref 1))

(dosync
  (println "t1 started")
  (alter cnt1 inc)
  (Thread/sleep 100))

;;possible output
;;>>t1 started
;;>>t2 started
;;>>t1 started
;;>>t1 started
;;>>t1 started
;;the result in cnt1 will be
;;correct, but at what cost?
```

```
(dosync
  (println "t2 started")
  (alter cnt1 inc)
  (Thread/sleep 100))
```

Concurrent changes: commute

```
(def cnt1 (ref 1))

(dosync
  (println "t1 started")
  (commute cnt1 inc)
  (Thread/sleep 100))

;;possible output
;;>>t1 started
;;>>t2 started
;;there will be no restarts!
```

```
(dosync
  (println "t2 started")
  (commute cnt1 inc)
  (Thread/sleep 100))
```

commute disclosure

```
(def cnt1 (ref 1))

(defn debug-inc [x]
  (println "debug: inc")
  (inc x))

(dosync
  (println "t1 started")
  (commute cnt1 debug-inc)
  (Thread/sleep 100)
  (println "t1 finished"))

;;possible output
;;>>t1 started
;;>>debug: inc
;;>>t2 started
;;>>debug: inc
;;>>t1 finished
;;>>debug: inc
;;>>t2 finished
;;>>debug: inc
```

```
(dosync
  (println "t2 started")
  (commute cnt1 debug-inc)
  (Thread/sleep 100)
  (println "t2 finished"))
```

commute: algorithm

1. In transaction's body **commute** applies an operation to an in-transaction state similar to **alter**. Besides that the operation itself is stored until the commit.
2. On commit, the state merging is applied by the following (using a critical section):
   i. The current master state is accepted as an initial point (even if it changed since the transaction start)
   ii. All the operations provided with **commute** are applied once again to the master state
3. No restarts under any circumstances

Rules to use **commute**:

- All the operations with the same (even from different transactions) **ref** must be commutative to each other
- All the operations must be lightweight
- A transaction must not rely on the reference's value where commute

commute: rules violation

Goal: acquire unique identifiers from multiple threads. The obvious solution is to use counter and each subsequent value is considered as unique.

```
(def id-seq (ref 0))


(dosync
   (let [id (commute id-seq inc)]
     (println "t1: uniq id is" id)
     (Thread/sleep 100)))

;;>>t2: uniq id is 1
;;>>t1: uniq id is 1
```

```
(dosync
   (let [id (commute id-seq inc)]
     (println "t2: uniq id is" id)
     (Thread/sleep 100)))
```

Essence of violation: both threads rely on the result of the commute

How to fix: replace **commute** with **alter**, that will cause restarts when necessary

Task c6

Given a map of air routes, limited number of tickets and particular ticket price for each route, implement a program that simulate concurrent booking for tickets.
Typically, there are no direct routes to reach the destination desired by a customer, so transfers are necessary. Each customer would like to either book the tickets atomically for all the necessary routes or be notified that it is impossible. Use transactions (STM) to achieve the atomicity.
Lower price is preferred (even at cost of more transfers).
The skeleton code with some missing parts is provided. Complete it, measure the efficiency in terms of transaction (re-)starts and try to tune timeouts to satisfy all the customers.