Advanced Computer Systems - Exam 2016/2017

Andrea Lekkas

January 19, 2016

Question 1: Data Processing

The algorithm to execute the repartition proceeds as follows:

0. Preliminary phase:

- *Assign to each node a NodeId*
  There are many ways to accomplish this part. For instance: we consider that each node contains the messages belonging to a range of `time` values.
  Each node broadcasts the `time` value of the first message it contains. After this phase, each node collects the values it has received in a dictionary (key = NodeFirstTimestamp, value = NodeAddress). Finally, the dictionary is sorted ascendingly on the key values, and the keys are reassigned to {1,2,...N }.

- *Choose a hash function*
  We must pick a hash function h : {user-id} $\rightarrow$ [1,N].
  Ideally, the hash function is uniform, and the number of user-ids associated to a single node is $\frac{\text{n. of users}}{\text{n. of nodes}}$.

Since there are no I/O operations performed in this phase, I/O cost = 0 and I/O time = 0.

Considerations over the main memory of each node:
Each node has M/N pages that contain the data of the MSG relation, plus 2N pages for communication with the other nodes.
These 2N pages are : N input buffers + N output buffers, each one of them is numbered, and thus associated to a specific node.

1. Sending the messages to the right nodes, according to the user-id:
In parallel, each node reads its main memory executing the following steps:

- read tuple $\#i$. Hash on its user-id, obtaining the NodeId n. $h(u_i) = n$
- write tuple $\#i$ in the output buffer $\#n$.
- When an output buffer is full: send its contents across the network to the node n.
- When the node receives a page in its input buffer: the node writes the page on its disk.

At the end of this phase, each node stores on disk the messages sent by the users associated with it.
The pages containing these messages are sorted internally according to the `time` attribute.
Each page was created by a single node that processed the tuples in its main memory sequentially, and those tuples were sorted by `time`.
Therefore, the pages on disk can be considered *sorted sublists*. We are in the same state as after the execution of Phase 1 of the External Merge-Sort algorithm.

Each node has processed its own main memory, sending out a number of pages ≤ M/N.
Since we assume there is no significant skew across users, we can state that each node has received and written to disk ≈ M/N pages.
Phase I/O cost: M I/Os.
Since the nodes work in parallel, Phase I/O time: M/N I/Os.

2. Finalization phase:
Since each node contains M/N pages, we have no need to replicate the Phase 2 of External Merge-Sort. We can simply load the messages in the main memory and sort them there.
The nodes work in parallel locally. For each node:

- load the M/N pages of messages into the main memory.
- sort the pages in the main memory, according to the `time` attribute.

Phase I/O cost: N(M/N)=M I/Os. Phase I/O time: M/N I/Os.

Total I/O cost = 2M I/Os. Total I/O time = 2(M/N) I/Os.

QUESTION 2: ARIES RECOVERY

2.1 The transaction table

After the first crash, the Analysis phase reads the log starting from the last checkpoint:

LSN 3 : Add T1 to the Transaction table, with status='In progress' and LastLSN = 3
LSN 4 : Add T2 to the Transaction table, with status='In progress' and LastLSN = 4
LSN 5 : Add T3 to the Transaction table, with status='In progress' and LastLSN = 5
LSN 6 : Change T3 LastLSN to 6
LSN 7 : Change T1 LastLSN to 7
LSN 8 : Change T2 LastLSN to 8
LSN 9 : Change T1 status to 'Commit' and LastLSN = 9
LSN 10 : Change T3 status to 'Abort' and LastLSN = 10
LSN 11: Remove T1 from the transaction table

Transaction table (after 1st crash)

tID	**tStatus**	**lastLSN**
T2	In progress	8
T3	Abort	10

After the second crash, the Analysis phase reads the log after the last checkpoint again: first it reproduces the situation we had after LSN 11, then it processes the information contained in the log records written after the first crash:
LSN 12 : Change T2 status to 'Abort' and LastLSN = 12
LSN 13 : Change T2 LastLSN to 13

Transaction table (after 2nd crash)

tID	**tStatus**	**lastLSN**
T2	Abort	13
T3	Abort	10

2.2 The dirty pages table

The dirty pages table stores the `pageID`, and the `recLSN` (the LSN of the first log record that caused the page to become dirty).

Dirty pages table (after 1st crash)

pageID	**recLSN**
P5	3
P2	4
P7	5
P3	8

After the 2nd crash, we have a modification caused by LSN 13 : It is a Compensation Log Record, that rolls back the modification made by T2 on P3 at LSN 8.

Dirty pages table (after 2nd crash)

pageID	recLSN
P5	3
P2	4
P7	5

2.3 Redo starting point

The Redo phase starts at the smallest recLSN in the dirty pages table. This phase brings the Database in the state it was at the time of the crash.
In both cases, smallest recLSN = LSN 3

2.4 Recovery from the 2nd crash: Redo and data pages

We can not say which Redo actions will cause actual writes to data pages.
We know the Redo phase executes all actions from the starting point (the min recLSN in the dirty pages table), unless one of the following 3 conditions holds:

- The affected page is not in the dirty pages table
- (recLSN of the page) > (LSN that is being checked)
- *(pageLSN)* $\geq$ *(LSN that is being checked)*

The pageLSN is the LSN of the last action for that page that was written in secondary storage. Since we do not know if or when some pages have been written to disk, we can not check the 3rd condition and we are unable to state which updates will cause writes on the data pages in the Redo phase.

2.5 Additional contents of the log

The Analysis phase and the Redo phase do not add any log entries. In the Undo phase:
toUndo = {13,10}; 13 is a CLR, we replace it with its `undoNextLsn`=4. No logging.
toUndo = {4,10}; 10 is an abort record. We replace it with its `prevLsn`=6. No logging.
toUndo = {4,6}; we proceed:

LSN	prevLSN	tID	type	pageID
14.	10	T3	CLR	P5 (undoNextLsn = 5)

toUndo = {4,5};

LSN	prevLSN	tID	type	pageID
15.	14	T3	CLR	P7 (undoNextLsn = null)
16.	15	T3	end	·

toUndo = {4};

LSN	prevLSN	tID	type	pageID
17.	13	T2	CLR	P2 (undoNextLsn = null)
18.	17	T2	end	P2 ·

QUESTION 3: REPLICATION

We have to define a distributed system that works with multiple clients and follows the Synchronous Replication model ReadAny-WriteAll.
Multiple copies of the database are stored at different sites. Since this is Synchronous Replication, the WRITE method does not return until all the sites have been updated successfully.
The algorithm I chose to solve this task resembles the PAXOS algorithm. The processes/machines are divided into 3 groups:

- *Proposer* : One Proposer receives the requests from the clients. It stores these requests in a queue, identified by a integer number N.
  The Proposer sets the values of some features of the Message class : the ID, the client reference and the Quorum.
  When a request arrives, The Proposer asks the *Acceptor* machines if they are ready to work on it. If a sufficient number (=quorum) of machines are ready, then it sends them the order to execute the READ/WRITE request.
  If there are not enough machines available, the Proposer keeps asking the Acceptors to solve the oldest request found in the queue until they are ready to execute it.
- *Acceptors* : each one of the M acceptors mantains a replica of the database. When needed, an Acceptor states if it is able to work on a request or if it is already busy.
- *Learner* : The Learner receives the answers the acceptors have elaborated. When it receives Q_r=1 (/Q_w=M) results of a reading (/writing) operation, it sends it back to the original client, that can thus complete the READ(/WRITE) function.

The Client:

```
value READ(key k):
    Message m = new Message(type="Reading", attributes=[k])
    rw_multicast(Proposer,m)
    //wait for the message:  {"answerToReading",v}
    return v


void WRITE(key k, value v):
    Message m = new Message(type="Writing", attributes=[k,v])
    rw_multicast(Proposer,m)
    //wait for the message:  {"answerToWriting"}
```

The Proposer:

```
void rw_deliver(Message m, Client c):

//msg received from a client:
    if (m.type == "Reading" or m.type == "Writing"):
```

```
        N++        m.setID(N)
        m.setClient(c)
        if m.type = "Reading" :  m.setQuorum(1)
        if m.type = "Writing" :  m.setQuorum(N_OFACCEPTORS)
        rw_multicast(Acceptors, {"askForPromise", m}
        queue.add({m})


//msg received from an acceptor:
    if (m.type == "IAmReady"):
        idOfRequest = m.attributes[0]
        readyAcceptors.add({idOfRequest,acceptor})
        if (readyAcceptors.getElems(idOfRequest).size ≥ quorum):
          msg = queue.getRequestById(idOfRequest)
          queue.removeElementById(idOfRequest)
          rw_multicast(ReadyAcceptors, {"ExecuteRequest", msg}
```

An Acceptor:

```
void rw_deliver(Message m, Client c):

    if (m.type = "askForPromise"):
        if (busy = True) :  send(c,"NotReady"})
        if (busy = False) :  send(c,{"IAmReady",idOfRequest})

    if (m.type = "ExecuteRequest"):
        execute the Read/Write request.
        answer = new Message(type="Reading/WritingAnswer",
                     attributes=[value,clientReference])
        rw_multicast(Learner, answer)
```

The Learner:

```
void rw_deliver(Message m, Client c) :

    originalClient = m.attributes[1]
    valueToSendBack = m.attributes[0]

    if m.type == ReadingAnswer:
        send (originalClient, {"answerToReading",valueToSendBack})

    if m.type == WritingAnswer:
        writingCounter++
        if writingCounter = N_OFACCEPTORS
            send (originalClient, {"answerToWriting"})
```

Observations:
Concurrency is allowed for reading operations: there may be as many as M=*# of acceptors* reading operations, each carried out by a different *Acceptor* machine.
The writing operations have to wait for any reading operations to finish, since they need to work on *all* the Acceptors.
If there are not enough available machines to carry out an operation, then the Proposer just waits, sending periodically a "askForPromise" request to the Acceptors for the operation that is at the head of the queue.
Note: the "queue" is not restricted to removing its head element. As it is written in the pseudocode, if the reading requests R1, R2, R3 arrive in sequence, and (for instance due to some connection issues) the next message comes from an Acceptor stating that it is ready to execute R3, then the Proposer executes `queue.removeElementById(3)`.

The Learner collects the responses of the Acceptors, and when the Quorum is reached it sends the answer back to the original client that sent the request to the system.

This model does not take failures into account. We can extend it to protect it from fail-stop failures in this way:
The *Proposer* and the *Learner*, that are single machines with a central role in the algorithm, have a backup machine that requests from them a heartbeat periodically. The Proposer's backup also retrieves the queue of requests (it does not add much overhead on the network, since the requests are lightweight objects). If the main machine fails, the backup steps in and the original machine, if it manages to restart itself, assumes the role of backup.
Periodically, the *Acceptor* machines send a heartbeat to the Proposer and to the Learner. If one of the Acceptors crashes, then it can not send it anymore; after K consecutive missed heartbeats (in order to allow for network hiccups), the Proposer and the Learner decrement the NumberOfAcceptors variable; the Writing requests that arrive after this adjustment can still be executed by the remaining machines. This helps to guarantee the availability of the service.

PROGRAMMING TASK

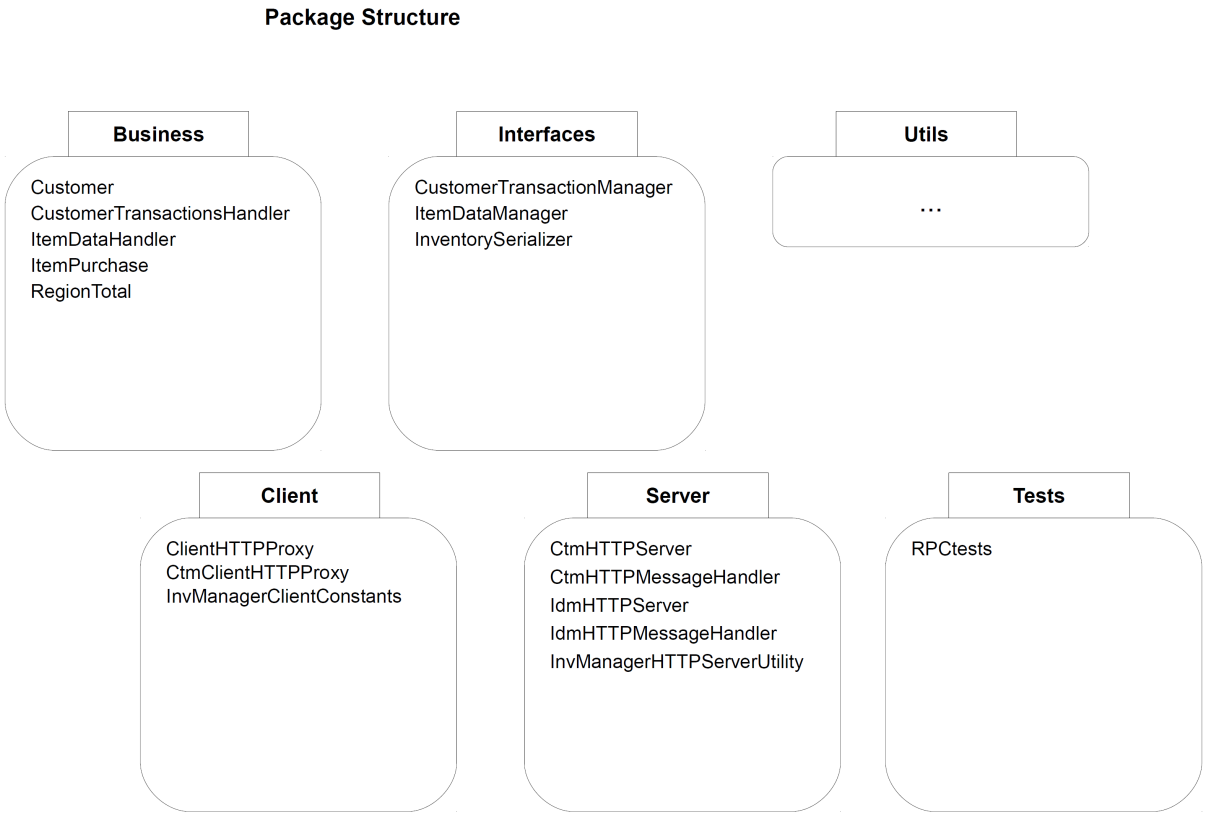
High-Level Design Decisions, Modularity, Fault-Tolerance

Question 1

Describe your overall implementation of the CustomerTransactionManager and the ItemDataManager interfaces.
Overview of your code:

The interface `CustomerTransactionManager` is implemented in the class `CustomerTransactionsHandler`. When we create an instance of the class, we fix the number of `ItemDataManagers` and we define the initial set of registered customers.
The customers are stored in a ConcurrentHashMap, where the key is the customer ID.
In the same way, another ConcurrentHashMap stores the `ItemDataManagers` associated with this `CustomerTransactionsHandler`.

The class `ItemDataHandler` implements the interface `ItemDataManager`.
A private `List<ItemPurchase>` contains the item purchases associated with this instance.



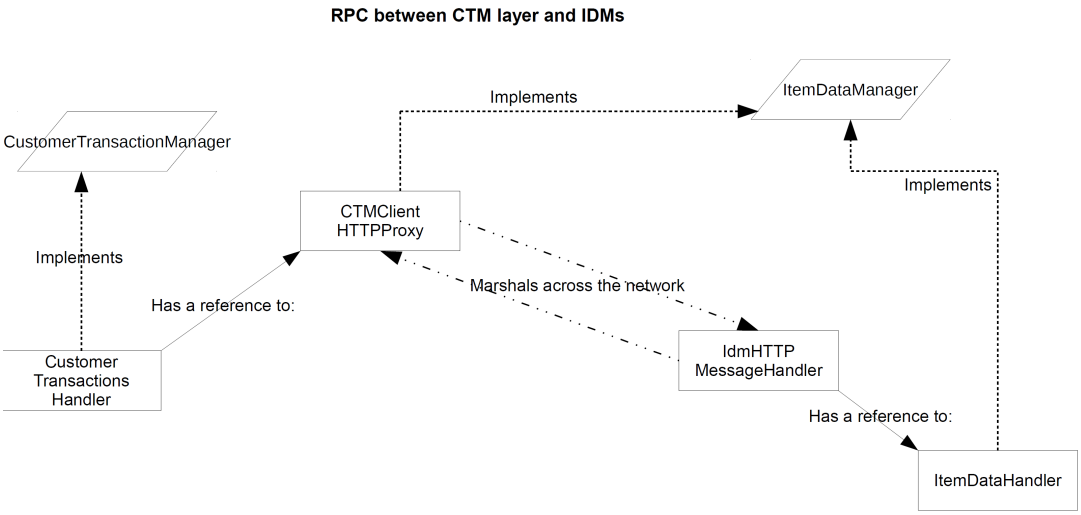
What RPC semantics are implemented between clients and the `CustomerTransactionManager`*? What about between the* `CustomerTransactionManager` *and the* `ItemDataManager`*? Explain.*

Regarding the RPC between clients and `CustomerTransactionManager`:
Instead of using `CustomerTransactionsHandler` as we would in a local setting, we use the class `ClientHTTPProxy`, that implements the same `CustomerTransactionManager` interface.
The `ClientHTTPProxy` contains the proxy invocations that serialize the client requests and marshal them across the network.

Server-side, the class `CtmHTTPMessageHandler` implements the naming service: it reads the keywords contained in the client messages (PROCESSORDERS, GETREGIONTOTALS, etc.) and it invokes the appropriate method to handle the request.
It passes the parameters to the API functions of `CustomerTransactionManager`.
The class `CtmHTTPServer` starts the Server for the CustomerTransactionManager layer. It listens on the DEFAULT_PORT=8081.

About the RPC between the `CustomerTransactionManager` layer and the `ItemDataManagers`:
In its constructor, the intermediate layer (`CustomerTransactionsHandler`) creates a number of `CtmClientHTTPProxy`, that implement the `ItemDataManager` interface. In the ConcurrentHashMap, each one of those has a key i. They listen on the port DEFAULT_PORT + i.

In both cases, we implemented at-most-once RPC semantics. While some operations are idempotent (such as getTotalsForRegions) and could be repeated, we attempt to invoke a service only once.



How are failures of ItemDataManager *instances contained at the* CustomerTransactionManager*? Why does your failure containment implementation turn out to guarantee a fail-soft design for the* CustomerTransactionManager *with respect to* ItemDataManager *instances?*

The failures we can detect and contain are fail-stop. We assume that when an ItemDataManager fails, the machine stops and any further attempt to communicate with that machine causes a timeout.
We catch the failure in the intermediate layer, in `CtmClientHTTPProxy`. In the remote procedure calls (addItemPurchase and removeItemPurchase), when we execute `performHTTPexchange(...)`, we catch any resulting exception:
if the message of the exception was the one defined in *InvManagerClientConstants.STR_ERR_CLIENT_REQUEST_TIMEOUT*, then we do nothing and the execution proceeds as always (apart from the fact that the modification that should have been carried out in that IDM has not been executed).
If the exception had another cause, and the error was something different from fail-stop, we throw the exception again.
This is in accord with the definition of fail-soft: the system continues to work, guaranteeing a subset of its initial specifications. In this case, those items that would be assigned to the failed IDM are lost.

You have implemented the services under the assumption that durability is not provided as a property. Consider what would happen if the staff of acertaininventorymanager.com decided to migrate the implementation of both the CustomerTransactionManager *and the* ItemDataManager *to utilize a two-level memory with durability provided by a recovery protocol such as ARIES at each component. Would there be a need then to implement a commit protocol among the several components? Explain why or why not.*

Yes, we would need a commit protocol among the IDMs.
We can consider a *processOrders* invocation as a distributed transaction, that acts upon several sites/IDMs. For this transaction, we must guarantee all-or-nothing atomicity. Either all of the itemPurchases are registered in the IDMs, or none is.
If an IDM fails, then the ARIES recovery protocol resumes the normal course of operations, but in doing so it rolls back the transactions (addItemPurchases) that were active and not yet committed.
As a commit protocol, we could use 2PhaseCommit: in Phase 1, the Supervisor asks the sites if the transaction *processOrders* can be committed, and they respond with yes/no. Depending on the outcome of the vote, the Supervisor orders the sites to doCommit/doAbort.

Before-or-After Atomicity

Question 2

Each instance of the ItemDataManager *interface executes concurrent addition and removal of item purchases. Describe how you ensured before-or-after atomicity of these operations. In particular, mention the following aspects in your answer:*
- Which method did you use for ensuring serializability at each ItemDataManager *instance? Describe your method at a high level.*
- Argue for the correctness of your method; to do so, show how your method is logically equivalent to a trivial solution based on a single global lock or to a well-known locking protocol, e.g., a variant of two-phase locking.
- Argue for the performance of your method; to do so, explain your assumptions about the workload the service is exposed to and why you believe other alternatives you considered, e.g., different granularities of locking, would be better or worse than your choice.

To ensure before-or-after atomicity in the ItemDataManager operations, I simply used a global lock: a shared lock for reading operations (in the utility function `findItemPurchase(...)`), an exclusive lock for writing operations (`addItemPurchase`, `removeItemPurchase`).
More accurately, the lock encloses the operations on the List<ItemPurchase>.
The lock is implemented in the class `GlobalReadWriteLock(...)`, and it is based on a counter.
With a shared lock active, `counter`>0. With an exclusive lock active, `counter`==-1. The threads are handled with wait() and notifyAll().

The actions performed by `addItemPurchase` and `removeItemPurchase` add or remove an element from the List.
In order to do this without creating inconsistencies, the List must be locked in the meanwhile. (If an operation "updateItemPurchase" existed, working on one element, I would have used an element-wise granularity).
The reading lock in `findItemPurchase` ensures that the list can not be modified while we are looking for an element.

Question 3

The CustomerTransactionManager *service executes operations originated at multiple clients. Describe how you ensured before-or-after atomicity of these operations. In particular, mention the following aspects in your answer:*
- Which method did you use for ensuring serializability at the CustomerTransactionManager *service (e.g., locking vs. optimistic approach)? Describe your method at a high level.*
- Argue for the correctness of your method; to do so, show how your method is logically equivalent to a trivial solution based on a single global lock or to a well-known locking protocol, e.g., a variant of two-phase locking.
- Argue for whether or not you need to consider the issue of reads on predicates vs. multigranularity locking in your implementation, and explain why.
- Argue for the performance of your method; to do so, explain your assumptions about the workload the service is exposed to and why you believe other alternatives you considered (e.g., locking vs. optimistic approach) would be better or worse than your choice.

To ensure before-or-after atomicity at the CustomerTransactionManager layer, I used a Strict 2PL protocol with deadlock detection. I implemented it using the classes Strict2PLManager, MyLock and WaitingGraphManager.
The customers and the IDMs are stored in 2 `HashMaps` in the CustomerTransactionHandler. When a transaction is executed (eg. when a thread invokes `processOrders`), I proceed this way:

- when I process each element (eg. each ItemPurchase), I try to acquire the appropriate lock on the customer, as well as on the IDM associated with the purchase. I am invoking, in `Strict2PLManager`, the function
  `tryToAcquireLock(Integer xactId, Integer objectId, LockType lockType)`
- The aforementioned method executes all appropriate evaluations (shared vs. exclusive lock, is this transaction already holding the lock?, etc.), and if there are no deadlocks the method eventually returns.
- after the transaction has executed its task over all the elements, we enter the Release phase: we release the locks over all the objects we were processing, and the method concludes.

In order to solve the deadlocks:
The transactions holding the lock on an object are stored in the `MyLock` class.
When T_i has to wait for the lock on an object, we add a number of edges to the waiting graph stored in `WaitingGraphManager`.
Periodically, another thread checks the graph for cycles. If some is found, one node/transaction is flagged as "to be aborted" in the Strict2PLManager. An `AbortedTransactionException` is thrown.
The CustomerTransactionManager has registered the orders that have been already processed. When a transaction has been aborted, it rolls back all the modifications that were made

on the IDMs and Customers, and finally it releases all the locks it holds (eliminating the deadlock).

Observations:

- Using Strict 2PL, different threads can access different ItemDataManagers, but 1 IDM can be accessed by a single thread at a time, and it stays locked and associated with that thread for the whole duration of the transaction. Given that the `addPurchase` and `removePurchase` operations have to lock the entire internal list to modify it, I initially deemed the loss in concurrency an acceptable trade-off, to be able to implement the Strict2PL protocol.
- Nevertheless, through extensive testing and logging I observed that this makes the transactions more prone to deadlocks and subsequent aborts. Since the number of IDMs is generally small (eg.5), if we have a considerable number of purchases (eg.10 or more) then each transaction ends up using all IDMs. If 2 transactions get hold of different IDMs, then it is likely that one of them will have to be aborted, and the surviving transaction is still likely to encounter conflicts with the other incoming transactions, until one manages to get hold of all IDMs. Therefore, I decided to simply use a ConcurrentHashMap for IDMs, and keep them out of the locking protocol.
- Optimistic concurrency control would have 2 drawbacks: first, when a single transaction is in the Validation phase, no other transaction is allowed to commit (otherwise we could have value conflicts). Moreover, we would have to write in a private working space and then transfer all the objects a transaction modifies. Depending on the number of ItemPurchases, we may have to operate with a great number of customers, and that also increases the probability of intersection of different Writng Sets.

Testing

Question 4

Describe your high-level strategy to test your implementation of the CustomerTransactionManager.
In particular, mention the following aspects in your answer:
- How did you test all-or-nothing atomicity of operations at the CustomerTransactionManager*?*
- How did you test before-or-after atomicity of operations at the CustomerTransactionManager*?*
Recall that you must document how your tests verify that anomalies do not occur (e.g., dirty reads or dirty writes).

I tested all-or-nothing atomicity in the test class `RPCtests`.
In the Initialization phase, I start the CTM and IDMs servers, and I add a random set of pre-registered customers to the CTM.
Test n.1, `testProcessOrders()`, simply checks the basic functionality of the CustomerTransactionManager:
- We pick a customer at random, and we create a purchase of nOfUnits * unitPrice.
- After the purchase, the totalValueBought for the region of the customer must have increased of nOfUnits * unitPrice.
Test n.2, `testProcessInvalidOrders()`, checks the property of all-or-nothing atomicity:
- We try to process a number of ItemPurchases, some of which contain invalid parameters.
- Afterwards, the totals for the regions must be unchanged.
(This is due to the fact that the transaction first validates the input, throwing an exception in case of error, and starts modifying the database only in the next phase).

I tested all-or-nothing atomicity in the test class `AtomicityTests`.
In the Initialization phase, we define 2 sets of ItemPurchases, with fixed unit prices and item quantities.
In `testConcurrentAdd1Add2()`:
- We start 2 client threads, that add to the database the 2 fixed sets of ItemPurchases concurrently.
- At the end, we may have had a number of aborted transactions. Therefore, we do not require equality between the total value added to the database and the value of the purchases. Nevertheless, we state that (newTotalValue - oldTotalValue) must be a multiple of the PurchaseValue. (We do not know how many transactions have aborted, but this way we know that the concurrent addition of purchases caused no errors and left the database in a consistent state).

Moreover, the reliability of detecting and solving deadlocks is verified in the test class `WaitingGraphTests`. (In `testFlagTransactions()` we lock objects in such a way to cause a cycle, $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_1$, and then we assert that the WaitingGraphManager must have flagged at least one of the 3 transactions for abortion.)