Final project - Distributed Systems 1

Claudio Facchinetti - 223814
<claudio.facchinetti@studenti.unitn.it>
Matteo Franzil - 221214
<matteo.franzil+github@gmail.com>

December 11, 2021

1 Introduction

The purpose of this paper is to report the content of the project assignment, how we interpreted it, the reasoning we made in order to solve the problem and the protocol we came up with.

1.1 The problem

The problem assigned was the following: given three actor types, namely Client, Coordinator and Datastore, we had to design a protocol which would allow multiple clients to perform transactions on the data items mantained by multiple datastores through a coordinator; this transactions had to be atomic and do not interfere with each other. The protocol proposed had also to be fault resistant, meaning that it had to survive the case of crashes by any of the involved actors, at least in the validation phase.

1.2 The client actor

The client is the starting point of the protocol: once it wants to start a new transaction it contacts one of the coordinators and signals them the intention to begin a new transaction.
Once it is notified by the coordinator that the transaction has been correctly accepted the client can start sending READ/WRITE requests on data items to the coordinator; in case of a READ request on a data item x he will also receive back the value associated to the data item x.
When the client has no more actions to perform and it wants to close the transaction it sends a "transaction end message" to the coordinator which will respond with a message containing the outcome of the transaction: either COMMIT or ABORT.
In our scenario we assumed that each client performs once transaction at a time, as specified in the project assignment; later in this paper it is discussed what would happen if a client performs multiple transactions at a time and how would it be possible to fix issues, if any.

1.3 The coordinator actor

The coordinator is the meeting point between clients and datastores; in particular it is in charge of handling the beginning/end of the transactions, forward the READ/WRITE requests from clients to datastores and possible results of such requests from datastores to the appropriate clients.
It is also responsible for handling the datastore outcomes and take the final decision about a transaction outcome following the rules of the protocol.
In our scenarios we assumed that the coordinator could only crash in the middle of a decision process, meaning that if there is only one client talking to it the coordinator will only crash in one of the following cases:

- the coordinator did receive the transaction end request and did not send anything to the datastores;
- the coordinator did send vote requests to the datastores but it has not taken a decision yet;
- the coordinator did take a decision but did not manage to inform any datastore;
- the coordinator did take a decision and informed only some of the datastores.

In case of crash the coordinator acts differently depending on when it crashed; we assumed that every coordinator is able to restore from a reliable storage, in our case the instance variables.

1.4 The datastore actor

The datastore is in charge of handling data items; in particular each datastore handles a configurable number of items, ten in our case, and it has to keep them consistent among transactions.
The datastore receives from coordinators the READ/WRITE requests for a specific transaction and performs them in a separate space, named *private workspace*; each data item is also associated with a version number used to check consistency when it needs to decide if a transaction can be committed or not.
When the datastore receives a vote request for a transaction it checks if the transaction leaves the store in a consistent state: if it is case then it votes *YES*, otherwise it votes *NO*.
In our scenarios we assumed that the datastore could only crash in the middle of a decision process, meaning that if there is only one transaction going on the single datastore will only crash in one of the following cases:

- the datastore did not vote for the transaction yet;
- the datastore did vote for the transaction but did not manage to send its vote;
- the datastore did manage to send its vote but it did not receive the final outcome from the coordinator.

Depending on when the datastore has crashed it will recover performing different actions; we assumed that the datastore can recover from a reliable storage which, as stated above, is represented by the instance variables.

2 The protocol

The protocol we designed is a variation of the Two-Phase Commit protocol, adapted to handle multiple transactions. The protocol we designed is fault tolerant without blocking: it blocks only in a specific case that will be discussed later in this chapter.

2.1 Beginning a transaction

When a client wants to start a new transaction it simply chooses a coordinator randomly and sends to it a `TxnBeginMsg` specifying its own clientID.
The coordinator will reply to the client with a `TxnAcceptedMsg` which signals the client that the transaction has been started successfully and that from now on the only possible outcomes will be either **COMMIT** or **ABORT**. Upon receiving the `TxnBeginMsg` the coordinator will assign to the transaction performed by the client a globally unique identifier, namely the transactionID.

2.2 Performing an operation on a data item

The client has the possibility to perform two different types of operation over a specific data item: it can perform either a **READ** or a **WRITE** operation; of course during a transaction a client can perform operations over an unlimited number of data items.
In order to read the value of a specific data item the client sends to the coordinator a `TxnReadRequestMsg`, specifying the data item of which it wants the value. Upon receiving this message the coordinator finds out the data store in charge of that specific key and will send to it a `DSSReadRequestMsg`, specifying the transactionID it refers to and the data item the client wants to read.
The data store retrieves the value of the specified data item and replies to the coordinator sending a `DSSRead ResponseMsg`, specifying the transactionID, the data item and the associated value the message refers to.
The coordinator finally translates this message into a `TxnReadResultMsg` and sends it back to the client.
The flow to perform a **WRITE** operation is almost identical: the only key difference is that there is no message returned neither from the data store to the coordinator nor from the coordinator to the client.

2.3 Ending a transaction

In order to end a transaction the client has two options:

- unilaterally abort the transaction;
- ask the coordinator to commit the transaction.

In both cases the client sends to the coordinator a `TxnEndMsg` specifying its own clientID. In the first case, it will signal that it does want to abort the whole transaction, while in the second case it will specify that it is willing to commit the transaction.
Upon receiving the message the coordinator will behave differently whether the client wants to commit the transaction or not:

- if the client wants to unilaterally abort the transaction the coordinator simply broadcasts to the data stores a `DSSDecisionResponseMsg` with the final decision **ABORT**;
- if the client wants to commit the transaction the coordinator then sends a `DSSVoteRequestMsg` to all the data stores, asking for their vote about the specific transaction. Upon receiving this message every data store will check if committing the transaction will leave the items it is responsible for in a consistent state and sends back a `DSSVoteResponseMsg` for the specified transaction to the coordinator setting the vote accordingly: if the items will be in a consistent state then the vote will be **YES**, otherwise the vote will be NO and it will also fix the decision to be **ABORT**. Upon receiving all the votes the coordinator will take the final decision and will send a `DSSDecisionResponseMsg` to all the data stores: if all data stores voted YES the decision will be **COMMIT**, otherwise it will be **ABORT**; it will also send back a `TxnResultMsg` to the client with the final outcome of the transaction. Upon receiving the `DSSDecisionResponseMsg` every data store will behave accordingly: if the decision is **ABORT** then all the changes made in that transaction will be discarded, otherwise they will become effective.

2.4 Coordinator crash

The actions taken by the data stores and by the coordinator upon restoring depend on then the coordinator crashed:

- if the coordinator crashed before sending any `DSSVoteRequestMsg` then nothing happens and once the coordinator will come back up it will start the vote request;
- if the coordinator crashed after sending some, but not all, the `DSSVoteRequestMsg`s then the data store(s) that received the message and sent back the vote will realize that the coordinator crashed and will perform the temination protocol. Upon restoring the coordinator will ask all the data stores if they managed to take a decision: if so the coordinator will agree, otherwise it will simply decide to abort;
- if the coordinator crashes before sending any `DSSDecisionResponseMsg` then the timeout of the data stores will expire and they will proceed as in the previous case. The only difference is that upon restoring the coordinator will send to all data stores its decision: if they did not manage to take a decision they will be informed of the decision while if they managed to take a decision without the coordinator they will simply ignore the message because it will be the same decision;
- if the coordinator crashes after sending some, but not all, `DSSDecisionResponseMsg`s then the timout of the data stores that did not receive the message will expire and they will run the termination protocol: they will contact one of the stores that got the decision message and they will be informed of the decision. Upon restore the coordinator sends again all the messages but they will be ignored by the stores since the decision of the coordinator will be the same as the previous one.

2.5 Data store crash

The actions taken by the other data stores, by the coordinator and by the data store upon recovery depend on when the crash happened:

- if the datastore crashes before receiving any vote request or before deciding its vote, then coordinator's timeout will expire and it will just decide **ABORT**, sending a `DSSDecisionResponseMsg` to all the data stores. Upon recovery the data store will realize that it has not voted and will simply **ABORT**. Note that this also works for the other transactions: while the data store was not alive he might have missed some operations, so it will just abort not to create inconsistencies;
- if the datastore crashes before sending back its vote, then the coordinator's timeout will expire and it will just decide **ABORT**, sending a `DSSDecision ResponseMsg` to all the data stores. Upon recovery the data store takes actions with respect to its vote: if it voted NO then it just aborts, otherwise it asks to the coordinator, and other data stores, for the final outcome of the decision;
- if the datastore crashes after sending its vote to the coordinator but before receiving the `DSSDecisionResponseMsg`, then the coordinator does nothing. Upon recovery the data store will take actions with respect to its vote: if the vote is NO than nothing happens since he already decided to abort, otherwise it will just ask other data stores and the coordinator for the final outcome of the transaction.

2.6 Termination protocol

When the coordinator crashes before sending all the `DSSDecisionResponseMsg`s, the data stores will run a simple protocol to determine whether they can take a decision without the coordinator; in particular when the timeout of a data store for the upper cited message expires it performs the following actions:

- it will send a `DSSDecisionRequestMsg` to all data stores;
- upon receiving this `DSSDecisionRequestMsg` if the data store already knows the final outcome of the transaction it replies with a `DSSDecisionResponseMsg`, setting the decision it know;
- upon receiving a `DSSDecisionResponseMsg` the data stores fixes its own decision so that it matches with the one in the message.
- if it receives no `DSSDecisionResponseMsg` then it means that the coordinator crashed and none of the other stores knows the final decision, therefore it is necessary to wait for the coordinator to come back up.

This is useful in case there one or more data stores that voted **NO**: in this case the final outcome of the transaction has already been decided because the only plausible final decision for the coordinator will be **ABORT**. On the other hand if all data stores voted **YES** then no assumptions can be made and they must wait the coordinator: it is possible that there is a data store that crashed after voting **NO** and they do not know about it.

3 Implementation

In this section it is described how we practically implemented all the aspects of the protocol; it will not be discussed how we implemented the Client actor, since the code has been provided by the instructor.
A common thing among the two actors we implemented is that in all the uni-cast communications we added a random delay in order to emulate network traffic

3.1 Coordinator: beginning a transaction

As said before, the transaction is initiated by the client using a `TxnBeginMsg`; upon receiving this message the coordinator generates a brand new random UUID and uses this as a unique identifier of the transaction. The transaction ID is then registered in a bidirectional map named `transactionMapping` and it is associated with the `ActorRef` of the client that sent the message. We adopted this solution because we need to efficiently find the transaction ID to which every client is associated and also efficiently find the `ActorRef` of a client associated to a given transaction ID in order to be able to send back to the client the result of possible **READ** operations on data items.
This map is also used by the coordinator upon recovery to determine which are the ongoing transactions that might have been already decided by the data stores using the termination protocol.

3.2 Coordinator: ending a transaction

Upon receiving a `TxnEndMsg` from the client the coordinator retrieves from the map the corresponding transaction ID and issues a new vote request for the transaction by sending a `DSSVoteRequest` to all the data stores, setting a timeout in order to be able to detect possible crashes.
When the coordinator collected all votes the coordinator fixes its decision for the current transaction, creating a new record in the `decision` map; the actual value stored in the map depends on whether all the stores voted **YES**: if this is the case then the decision will be **COMMIT**, otherwise it will be **ABORT**.
Once it fixed its decision it sends a new `DSSDecisionResponseMsg` to all data stores informing them of its decision for the current transaction; it also sends a new `TxnResultMsg` to the client to inform it of the final outcome by retrieving the correct `ActorRef` from the `transactionMapping` map.

3.3 Data store: data items

A single data store is in charge of ten data items which are assigned to it at creation time; every data item is a simple structure containing:

- `value`: an Integer indicating the value of the data item;
- `version`: an Integer indicating the version of the data item. It can be incremented only once during the item lifetime.;

- `touched`: a Boolean value indicating whether the version of this item has already been incremented.
- `locked`: an AtomicInteger indicating whether the data item is locked or not;
- `locker`: the ActorRef of the locker to ensure no locks from other actors, it has been used for debugging purposes.

The data items are stored in a map named `items`, having as key an integer representing their "name". The data item class also exposes three useful methods that are used for its management:

- `voidincrementVersion(ActorReflocker)`: it increments the version of the item by 1 if it was not yet been done. This employees a at-most-once semantic;
- `booleanacquireLock(ActorReflocker)`: it attempts to acquire a mutual exclusive lock on the item by using the CompareAndSet operation. The return value is the result of the locking attempt;
- `voidreleaseLock()`: it releases the lock from the item and also sets locker to null.

The `locker` parameter in the method is used to check that the one issuing the operation is the one actually having the lock; in case of the acquireLock operation if the lock is acquired than the locker property of the data item.

3.4 Data store: private workspaces

When the client performs **READ/WRITE** operations they do not reflect immediately on the actual data items: they are executed in a separate memory region and only at the end they are made effective, if they do not create inconsistencies.
When the first **READ/WRITE** request is received then a new private workspace is associated to the transaction ID using the `privateWorkspaces` map.
A private workspace is just a map associating integer keys to DataItems.
Upon receiving a request the private workspace associated to the transaction ID is retrieved, or a brand new one is created. The requested operation is performed on the requested item in the private workspace: if the item is not yet present in the private workspace then a copy of the original item is made and inserted into it.
The management of the data item inside the private workspace is pretty much the same whether the operation is a **READ** or a **WRITE**; the key difference is that if it is a **WRITE** operation than the incrementVersion() method on the data item in the private workspace is invoked. We decided to increment the version of the data item in the private workspace only for **WRITE** requests in order to achieve better liveness among transactions performing only **READ** operations.

3.5 Data store: checking for consistency

Upon receiving a `DSSVoteRequestMsg` the data store checks if it already decided for the specific transaction by looking into the `vote` and in the `decision` maps : if so it simply responds back with the vote, otherwise it checks whether committing the transaction would leave the store in a consistent state.
To do so it retrieves the correct private workspace for the current transaction and it performs the following steps for every item in the workspace:

- it tries to acquire the lock on the corresponding item in `items`: if it fails then it votes NO immediately;
- it checks if either the version in the private workspace is exactly one after the one of the original item or the versions are the same and also the values are the same. If none of these is true than it votes NO immediately.

The second condition guarantees that if the client only performed read operations on the item than the transaction does not get aborted because of other previous read operations by other transactions on the same item; it also guarantees that in case the client performed even a single WRITE operation than the current value in the store is the same as the one he could have read before performing the **WRITE** operations: this works because the version of the data item in the private workspace gets incremented as soon as the first **WRITE** operation is performed and it cannot be decreased anymore.
As we mentioned before the locking operation is performed by mean of the CompareAndSet operation: as soon as one of these operations the data store immediately votes NO. Taking into consideration that the CompareAndSet is a wait-free operation from the concurrency point of view is safe to say that the our implementation is indeed deadlock-free; in the worst case scenario there will be a lot of transactions aborting.
All the items that have been locked, regardless of the vote, get stored associated with the transaction ID of the current transaction in the `lockedItems` map: they need to be unlocked as soon as the node takes a final

decision.
With the given implementation of the client it is easy to check for inconsistency: at the end of all the transactions the sum of all data items must be the sum of all data items before the first transaction.
Please note that at the end of every "wave" of client transactions it is possible to interact with the system to make it print the current value of all data stores as well as temporarily stop the executions to check the actual consistency by following.

3.6 Data store: taking a decision

After the data store sent its vote it needs to take its final decision about the transaction(s) for which it sent a vote; this can happen with two different modalities depending on its vote:

- the data store voted **NO**: in this case it also fixes its decision to be **ABORT**, because the coordinator cannot decide **COMMIT** with even a single **NO** vote. This is done by sending itself a `DSSDecisionResponseMsg` immediately after voting, so that it can unlock the items as soon as possible;
- the data store voted YES: in this case it must wait for the `DSSDecisionResponseMsg` coming from the coordinator.

Upon receiving the `DSSDecisionResponseMsg` the data store checks the final decision and acts accordingly: if the decision is **COMMIT** then it applies the changes to the real items. In both cases the data store releases the locks it acquired over the items, fixes the decision and simply discards everything related to the transaction, except made for the vote and the final decision.

3.7 Crashing

The "crashed" state is actually not a real crash but it is a simulated one: a node that is crashed is in reality running but it is simply discarding every message it receives. We decided to act this way because it was, in our opinion, simpler than killing the node and it also allows us to make use of its properties as a safe storage.

4 Issues, fixes and improvements

In this final part possible issues with our implementation will be discussed, as well as possible fix. Other than this there will also be some proposals that could make the protocol we designed and the overall architecture of the various nodes more resilient to fail and possibly to eliminate the cases in which the protocol requires blocking.

4.1 Issues

4.1.1 Implementation issues

One first issue that comes straight from how the coordinators map the clients to transactions is that they use a bidirectional map: we adopted this method because in the project assignment is clearly stated that each client can make at most one transaction at a time. In case a single client decides to make more than one transaction clearly the implementation breaks, because the previous transaction ID gets discarded when a successive one begins. In order to solve this it would be, in our opinion, change a little the *transactionMapping* map in the coordinator and make assign a single transaction ID to a single ActorRef; in this way the new transaction will simply create a new assignment in the map. This is not enough: the client should also specify, using the transaction ID, in which transaction it is willing to perform a given operation. At the implementation level the change would be really small and also at the protocol level: it would be enough to add the transaction ID to every message sent from the client to the chosen coordinator. The data store would have no difference: they do not have the notion of client, but only the one of transaction ID.
Another possible issue regards the fact that the client will begin a transaction choosing a coordinator and will interact with that transaction only sending messages to that coordinator; if the client would have the possibility to perform operations contacting different coordinators the protocol does not work anymore. A proposal to possibly handle this situation could be adding a second decision message wave to all the coordinators: when a coordinator receives a `TxnEndMsg` containing the transaction ID *tID* it will issue a vote request to all the data stores for the transaction with identifier *tID*, take a decision for that transactions and then inform the data stores and the other coordinators about it. This also requires that the client sends the transaction ID of the transaction in which it wants to perform the operation: this is needed because in the current implementation a coordinator that was not the one which accepted the transaction does not know the mapping between the client and the corresponding transaction ID.

The third issue we would like to discuss is the fact that **WRITE** operations do not have a proper acknowledgment message: this is not a problem in the sense that it breaks the protocol, but it can be seen as a performance issue. Let's take, as a example, that a data store crashes while performing a write operation: the coordinator will never know it crashed. By inserting an acknowledgment message the coordinator would instead know that the data store crashed and it could set its decision to **ABORT** immediately: when the client tries to commit the transaction the coordinator will simply broadcast to the data stores the **ABORT** decision instead of performing the vote request, saving note. Please note that with the current implementation the transaction would also abort, because the data store will fix its vote to **NO** upon restoring for all transactions that are not yet complete: this is done on purpose to handle the fact that it might have missed some of the **READ/WRITE** requests.
One more issue that comes to our mind is the fact that two transaction that operate on two common items with WRITE **operations** do try to commit simultaneously: the first transaction gets the lock only on the first item and the second one gets the lock only on the second. In this case both the transaction will ABORT: the locking mechanism we adopted does not cause deadlock but it could lead to some liveness issues. A more sophisticated solution, in our opinion, would be to make the locking phase atomic in the sense that there can be only one transaction at a time trying to acquire the locks; this is possible by acquiring a temporary lock on the whole data store but would cause that all the other concurrent transactions would have to wait for it to finish acquiring locks. This solution would solve the problem just mentioned but as we explained would lead to unpredictable waiting periods therefore we did decide not to adopt it.
One last problem we would like to discuss is how the timeout is set: at the moment the timeout for every message is set as a constant, but we noticed that this approach is not really efficient or reliable: the larger the number of nodes in the system, the more messages need to arrive before a response is sent and this could lead to timeout expiration while the other party is not crashed; this does not really break the protocol but is acceptable for large-scale system. Our proposal would be to set an adaptive timeout, depending on the amount of traffic in the network and the number of nodes participating in the system. Additionally, the performance of the host system needs to be taken into account, as we noticed that during our stress tests (e.g. with 90+ clients, 1000+ datastores), the system would run out its allocated 2GB of RAM pretty quickly.

4.1.2 Performance issues

Another aspect we would like to discuss is the actual performance ratio we measured during transaction validation. In other words, we noticed that the performance of this project is subject to the constraints of the well-known *Birthday problem*[1]. In this case, we have n set as the number of total datastore items, while the number of people that draw from the pool of numbers are the set of writes directed by the clients. Assume we have the following hypothetical setting:

- 1000 datastore items (100 datastore instances)
- 5 competing clients running transactions concurrently
- each client performs one transaction
- each transaction is composed on average of 15 writes.

In this case, we would have $15 \cdot 5$ total writes. Each of this write may overlap with another one. Without adding pages of mathematical formulas, by checking the Wikipedia article we could derive that the probability that all writes involve different data items is:

$$\frac{{}_{1000}P_n}{1000^n} = \frac{{}_{1000}P_{75}}{1000^{75}} = \frac{\frac{1000!}{(1000-75)!}}{1000^{75}} = 0.058$$

Thus the probability that at least two writes are clashing is $1 - 0.058 = 0.942$. This is a very high probability: surely, at least one transaction will have to abort due to concurrent locking.

Of course, the real world case is much more unpredictable and complicated than this example. However, this goes to show that assuming randomly chosen writes as with the birthday problem, we would need a very high number of datastore items in order to minimize concurrent locking and maximize the number of committed transactions. We experimented with different setups of both datastore items and transaction numbers, and found that the default number (up to 40 writes per transaction!) would not fit well with a higher number of clients, and thus decided to lower it.

[1] `https://en.wikipedia.org/wiki/Birthday_problem`

4.2 Improvements

The protocol we designed does indeed require blocking for some specific cases which are the following:

- the datastores wait if the coordinator crashed before signalling the start of the validation phase of the transaction - this is necessary, as datastores don't know whether more R/W requests will arrive or the transaction will end;
- the datastores wait if the coordinator crashed before sending any `DSSDecisionResultMsg` and all data stores voted YES;
- the client waits in any of the cases in which the coordinator crashes.

To handle the second case it could be possible to change our reference protocol to the 3-Phase-Commit protocol: in this case the data stores would elect a new coordinator that would complete the decision process.
To handle the first case it could be possible to give the possibility to the client to contact different coordinators for performing operations within the context of the same transaction: the client sets a timeout and when it expires it simply tries to issue the same request to another coordinator. In this last case the only case in which the client is forced to wait is when all the coordinators crashed, but as long as one of them is correct then the client can proceed.