Optimistic Concurrency Control for Distributed Transactions

Distributed Systems 1 Course Project

Davide Eccher - 223810
University of Trento
davide.eccher-1@studenti.unitn.it

Riccardo Scilla - 213723
University of Trento
riccardo.scilla@studenti.unitn.it

1 INTRODUCTION

The aim of this work is to implement an optimistic concurrency control method using *Akka* framework in *Java*.

OCC assumes that multiple transactions can frequently complete without interfering with each other. It is applied for distributed transactions, which means that a single transaction started by a client involves data stored in multiple servers. Transactions use data resources without acquiring lock, thus they do not block each other, leading to an higher throughput. Before committing, each transaction verify that no other transaction has modified the data he has read: if the check reveals conflict modification, the transaction aborts and the changes are discarded.

In this report, the main implementation choices are described and discussed to evaluate the properties of the system. Crashes are also implemented and the protocol is designed with specific handling. Many tests are performed changing the number of nodes and random crashes to assess the correctness of the system.

2 DESIGN CHOICES

The system is composed by a TxnSystem and 3 main Actors: client, coordinator and server. An example with 2 Servers and 2 Clients that concurrently modify an item is shown in figure 3.

2.1 TxnSystem

In this file the ActorSystem is created together with the the specified number of Clients, Coordinators and Server. The WelcomeMsg are sent to clients and coordinators to make them known the presence of the coordinators and servers respectively. A crash scheduler is implemented to test the correctness of the general system in faulty scenarios.

2.2 TxnCoordinator

- **coordinatorId**: unique identifier of the coordinator.
- **servers**: list of all the servers present in the system.
- **globID**: number of accepted transactions.
- **txnBind**: binds client with the transaction.
- **OngoingTxn**: binds the transaction with all the servers contacted .
- **ServerDecisions**: store the commit votes of the servers.
- **txnHistory**: store the final commit decision.
- **readTimeout**: contain a timeout for every transaction waiting for server reads.
- **voteTimeout**: contain a timeout for every transaction waiting for server votes.
- **txnState**: follow the steps of a transaction.

TxnId. This class is used to associate operations to their transactions. A TxnId is composed by the ActorRef of client and coordinator and id that represents the globID of the coordinator. These information are used to get the clientId and coordId from the actor's names and put together in a unique name for logging:

$$TxnId = coordId.id/clientId$$

Function *equals* is overrided to compare the client, coordinator and id of two TxnId objects. Function *hashCode* is also overrided to map the coordId and id values into a single integer using the Szudzik's function.

How it works: once the coordinator gets a txn request, it creates a TxnId using the sender, itself and the current globID. This TxnId is stored in *txnBind* that allow to retrieve the TxnId given the sender; TxnId is used as key of others maps, like *OngoingTxn*, *ServerDecisions* and *txnState*.
On client messages, the coordinator retrieves the TxnId from *txnBind*. The TxnId is put into every message forwarded to the servers involved in the transaction, so that they can associate their operations without computing a binding: this is because a coordinator can have multiple transactions with the same server at the same time, while a client can have only one transaction at the same time with a chosen coordinator.

2.3 TxnServer

- **serverId**: unique identifier of the server.
- **dataStore**: map the item integer key with a triple <version, value, lock>.
- **workSpace**: map every transaction with a list of changes. Each change contains <key, version ,value, r/w flag>.
- **txnParticipants**: map every transaction with the servers and coordinator involved.
- **txnHistory**: store the final commit decision.
- **decisionTimeout**: contain a timeout for every transaction waiting for final decision.
- **txnState**: follow the steps of a transaction.

Private workspace. Each server keeps a workspace for every transaction, in which store the values and versions of the items read and modify them with the writes. Given a TxnId, the workspace contains a set of changes that are applied to the specified item. In general the structure is like:

```
<TxnId: [[key, version, value, r/w], ...]>
```

The key is given by the request message, the version is retrieved from the datastore calling *getVersionFromKey*, the value is retrieved calling *getValueFromKey* from the datastore if it has not been modified previously in that transaction, otherwise from the value in the workspace itself: this is done to avoid overwrites of previous operations. Last, the r/w flag is just a 0/1 that tells if the change is read or a write.

How it works: once a server receives a read, it first inserts a new set of changes for that TxnId if not present in the workspace; then it retrieve version and value as described before and calls *addWorkspace* that adds the read `[key, version, value, 0]` if that key has not been changed before in this transaction.
Once a server receives a write, it calls *updateWorkspace* that searches the change that has the same key of the received write, it increments the version if the change was a read and finally update the value with the new one and switch the r/w flag to 1.
Once a server is asked if it can commit, it calls *checkIfCanChange*

that loops every write change in the corresponding txn: if the lock in the dataStore for a given key is already acquired or the version in the dataStore is not previous one in the change, then the server can't change and the changes of the transaction are removed from the workspace; otherwise it calls *LockChanges* that acquires all the lock of the keys in the change.
Once a server receives the final decision, if it is a commit, then calls *ApplyChanges* that replaces the datastore keys with the version and value of the changes in the workspace. If it is an abort, then calls *FreeLocks* that remove the acquired locks without changing the datastore. Last the changes of the transaction are removed from the workspace.

Validation/Commit Phase. The validation phase starts when the coordinator receives *TxnEndMsg* from its client. At this point, if the client wants to commit, the coordinator starts the validation procedure by contacting all the servers involved in the transaction with *CanCommitMsg* and setting a timeout for the votes.
The server calls *checkIfCanChange* as described before, and if so, set a timeout for the final decision and send its vote with *ServerDecisionMsg*.
The coordinator stores all the votes in *ServerDecisions* and if arrives an abort or it receives all the votes from the servers, then decide by calling *getfinalDecision*. The final decision is then sent back to the servers and the client.

2.4 Crashes

To test the fault tolerance of the protocol, coordinators and servers are crashed at key points of the algorithm, that is in the validation/commit phase: BeforeDecide or AfterDecide for the coordinators and BeforeVote or AfterVote for the servers. These crashes are scheduled periodically in *TxnSystem*.

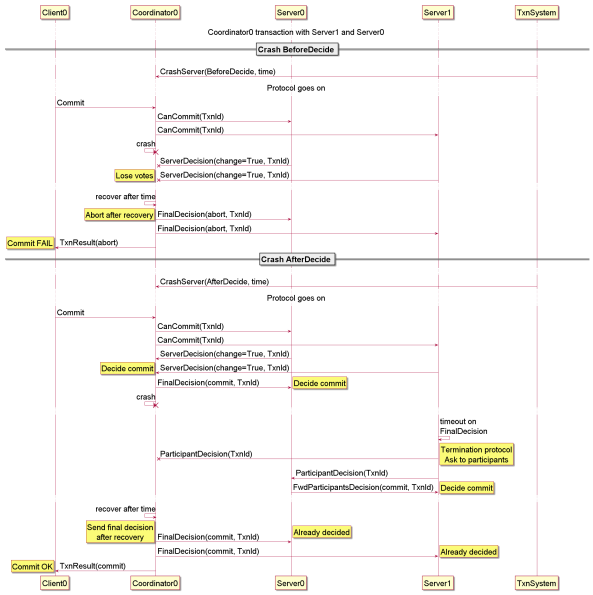

Figure 1: Example Crash Coordinator

Coordinators and servers have different crash behaviours:

The **coordinator** receives a *CrashCoordMsg* containing the crashType and the time when it has to recover. The BeforeDecide crash is performed in case it receives a *TxnEndMsg* from the client with commit decision, and after having asked to all the servers if they can commit. The AfterDecide crash is performed after having received the final decision from the servers: to see the behaviour in a different context with respect to the one of BeforeDecide, the final decision is sent to just one server before crashing.

The **server** receives a *CrashServerMsg* containing the crashType and the time when it has to recover. The BeforeVote crash is performed after having received the *CanCommitMsg* from the coordinator. The AfterVote crash is performed after having sent the vote to the coordinator but before having received the *FinalDecisionMsg*.

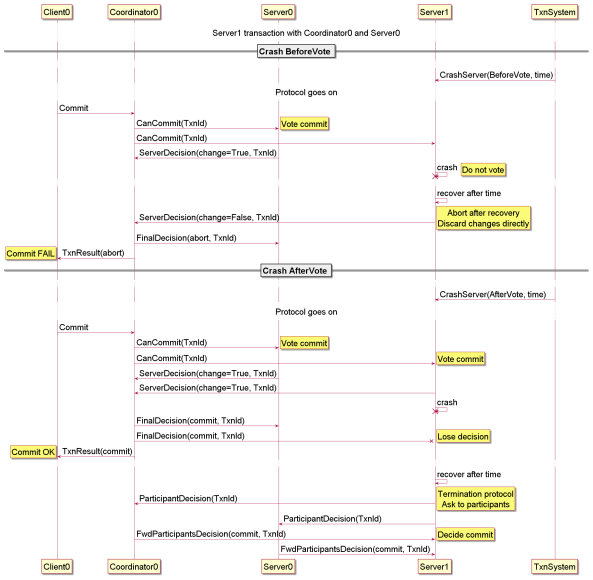

Figure 2: Example Crash Server

Once an actor receives the crash message and it arrives in the correct point in on the transaction, it schedules the *RecoveryMsg* after the specified time and switches the context to crashed: in this way all the messages are ignored except the recovery one.
Given that actors may be involved in many transactions concurrently and these have different lengths, once an actor is crashed in one of the key point of one txn, this same actor may not be in validation/commit phase in another concurrent transaction. For this reason, each actor keeps a state of where it is arrived in the txn using *txnState*, so that once it is crashed, it can perform the correct recovery actions for that specific transaction.

Coordinators and servers have different recovery behaviours:

The **coordinator** switches context to receive the messages, then loops for all *OngoingTxn* to see in which state the transaction was when it crashed. If it was BeforeDecide, then it sends abort to all servers in the transaction and to the client. If it was AfterDecide, it will retrieve the final decision from the *txnHistory* and sends it to all servers and to the client.

The **server** switches context to receive the messages, then loops for all *workSpace* to see in which state the transaction was when it crashed. If it was BeforeVote, then it sends abort to the coordinator. If it was AfterVote, it starts the termination protocol of 2PC. This requires that the server knows the participants of the transaction: these are sent by the coordinator with the *CanCommitMsg* and it includes also the coordinator.

clients	coords	servers	init_txn	na_txn	end_txn	commit_ok	commit_fail	abort	coordinator crash	server crash	sum	result
5	9	43	62	13	47	13 (27.66%)	27 (57.45%)	7 (14.89%)	1	1	43000	OK
8	7	75	79	24	52	12 (23.08%)	27 (51.92%)	13 (25.0%)	1	1	75000	OK
5	4	48	63	26	33	9 (27.27%)	21 (63.64%)	3 (9.09%)	1	2	48000	OK
3	8	25	38	2	33	11 (33.33%)	9 (27.27%)	13 (39.39%)	1	0	25000	OK
7	6	60	59	18	37	10 (27.03%)	20 (54.05%)	7 (18.92%)	1	1	60000	OK
7	9	57	49	9	39	9 (23.08%)	25 (64.1%)	5 (12.82%)	1	2	57000	OK
5	5	41	33	2	28	8 (28.57%)	15 (53.57%)	5 (17.86%)	2	0	41000	OK
3	9	24	39	1	35	13 (37.14%)	11 (31.43%)	11 (31.43%)	2	0	24000	OK
3	7	29	60	27	35	8 (22.86%)	23 (65.71%)	4 (11.43%)	2	1	29000	OK
2	8	17	34	0	32	14 (43.75%)	13 (40.62%)	5 (15.62%)	0	1	17000	OK

Table 1: Some test results

3 IMPLEMENTATION DISCUSSION

The 2 phase commit protocol guarantees strict *serializability*, in fact every transaction can modify concurrently a different private workspace. *Safety* is then achieved during the voting/commit phase: before allowing to commit the **coordinator** asks every involved server to check for conflicts, they control if the transaction has used or modified the last version of the database and no other transactions have modified that values in the meantime. Only after all the servers give a positive response the **coordinator** orders them to commit. This allow to maintain consistency inside the distributed database even with concurrent transactions under some assumptions:

- The system doesn't lose messages between the nodes
- Nodes store their decision consistently and permanently (the log storage never fails)
- Nodes can use their log to recover the last state after a crash

Node failures are taken into consideration by the protocol with the exception of some edge cases.

Server crash. If it happens during the *voting phase* it triggers a timeout in the coordinator that abort the transaction to avoid blocking waiting for the server to coming back alive. If it happens when the *decision is taken* by the coordinator, when the server recovers from the crash it asks the other server and the coordinator the result of the voting phase through the **termination protocol**.

Coordinator crash. When it comes back alive it simply forward the decision that it had taken before crashing, if no decision was taken it can safely abort to avoid problems and still maintain consistency. This is a simple and straightforward solution but can cause unnecessary abort, so it can be improved by repeating the voting phase
Coordinator crashes can cause *blocking issues to the servers*, in fact if servers are waiting for a decision from the coordinator, and they can't recover it using the termination protocol, they can't safely abort (the coordinator may have already taken the decision and the servers that have received it may have crashed or may have been isolated) so they are forced to **block and wait** for the coordinator to come back online. This corner case is not covered by the 2 phase commit algorithm, it can be solved by adding a *precommit* phase before the commit one (*3 phase commit*): if the coordinator crashes the servers can communicate through the *termination protocol*, if some of them are in precommit they can safely commit (all voted to commit), if not they can safely abort (no one has already done the commit).

4 EXPERIMENTAL EVALUATION

Some simulations are performed to assess the correctness of the system. Each simulation lasts 90 seconds; in each one, a random number of clients and coordinators are chosen between $2-10$, while the number of servers is taken between $8 \cdot \#clients - 10 \cdot \#clients$. The number of servers should be large enough with respect to the clients to make sure that the conflicts are rare.
Timeouts for reads, votes and decisions are adapted to the number of servers, to make sure that every node receives and responds correctly without timeouts prematurely.
A crash is scheduled every 20 seconds and every time an actor to crash, a type of crash and time before recovery are set. The time is randomly sampled depending on the number of server in the system, to test the behaviour in case the *RecoveryMsg* arrives before of after the timeout ($70\%timeout\_time - 150\%timeout\_time$).
When the time for each simulation expires, the system stops each client so that the transactions do not continue; however the systems proceeds for 10 seconds more before stopping, to allow all the servers to terminate correctly the commit phases and print all the logs needed for the check.

A *Check.java* is executed after each simulation to parse the logs and assess correctness of the system. First, it retrieves the order in which the clients begins the transaction; depending on this order, the TxnId is matched in the logs and the changes on the datastore values done by each transaction are stored in a map structure. Before processing the next TxnId, all the changes of the current TxnId are copied in the next, so that at the end we get the changes of all the transactions stored at the last TxnId. To assess the correctness, the differences of the datastore values in the changes of each TxnId must be 0 and the sum of all the datastore values in the last TxnId must be a number divisible by 1000.

The results of some tests are shown in table 1. We can see that the number of correctly committed transactions is in general quite low with respect to the failed one: this is because the number of operations in each transaction is high and thus each client contacts many servers. When a crash occur in one server or coordinator, there is a significant probability that this actor sends an abort decision, even though the result would be commit.

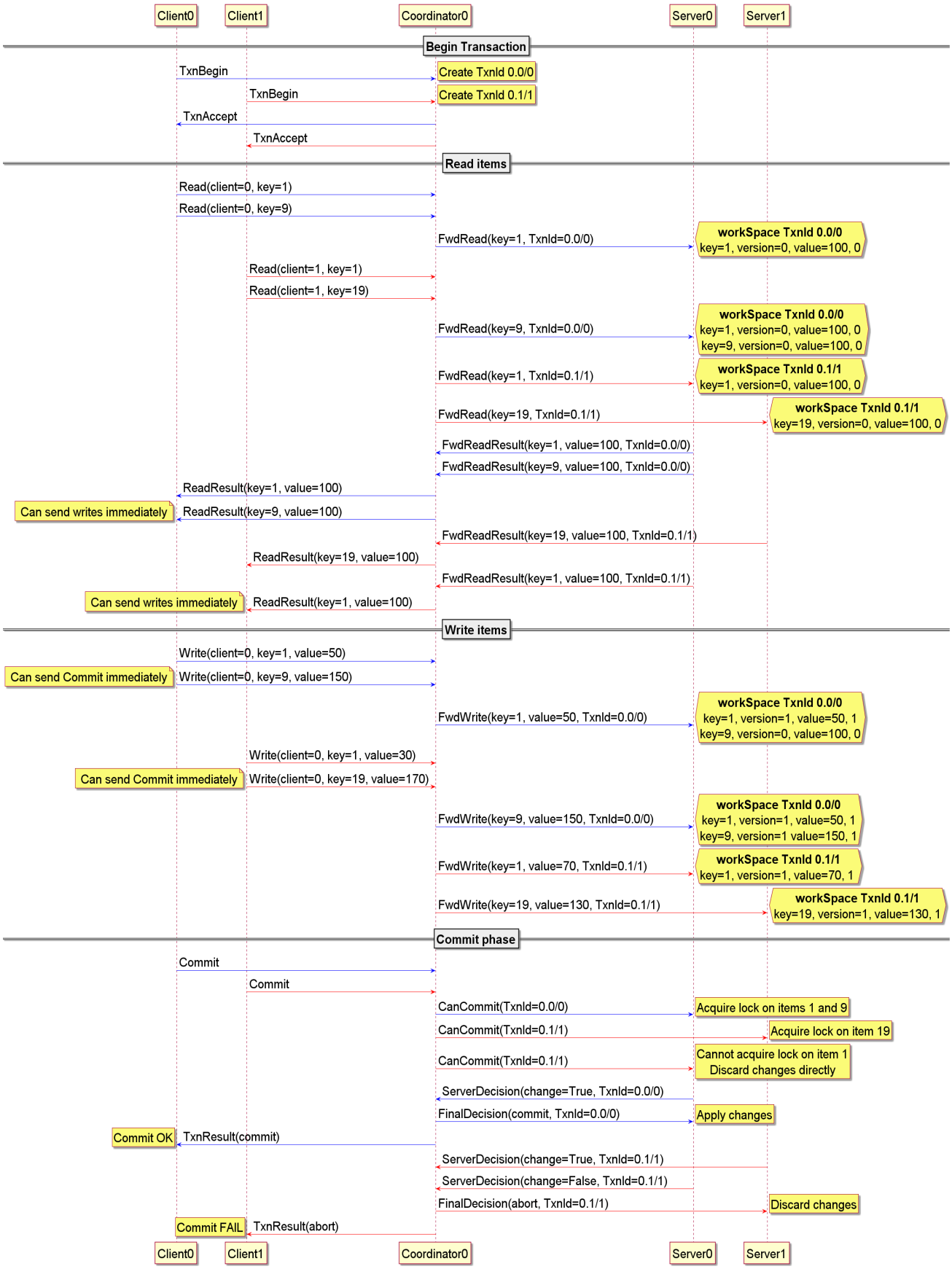

Figure 3: Example Concurrent R/W - 2 Clients 2 Servers