Distributed Systems 1

Concurrent 2-phase commit

Project Report

Gabriele Masina
220496

UNIVERSITY OF TRENTO

Giovanni Zotta
223898

1 Introduction

In this project we implemented a distributed system which handles transactions concurrently while guaranteeing strict serializability via a concurrent 2-phase commit protocol among multiple servers and coordinators.

The system is made by several clients which can start transactions by contacting a coordinator and sending to it a sequence of read/write operations on resources. Upon receiving a request, the coordinator forwards them to the servers holding the requested resource and forwards the answer back to the client if needed. The servers follow an optimistic concurrency control: they perform the operations on a private copy of the database, and only after the client has said that the transaction has ended, the updates can possibly be propagated to the official database. The latter phase is done through a validation process based on 2-PC, where servers agree on whether to commit or abort the transaction, in such a way that concurrent transactions do not conflict and that the number of committed transactions is maximized. In Figure 1 we can see an example of the typical workflow of a transaction.

2 Design choices

Following the instructions, we implemented an optimistic concurrency control on data items.

2.1 Data items

Our data items are stored in servers: each server has a portion of the whole database and only that server can access and manipulate it. The data stored in the server's database is in the form of a key-value item, which also has a version number associated to it. Every time a data item is *changed* by a committed transaction, the version gets updated.

2.2 Private Workspaces

Each transaction is identified by a pair $\langle$`client_id`, `client_attempted_transactions`$\rangle$. The first time a server sees a transaction, it creates an empty private workspace on which it can read and write resources at will. However, those changes are just fictional until the point on which that transaction is committed.

Each workspace contains the set of resources that were involved during the entire transaction. The resources are *copied* in the private workspace of a transaction T the first time they are accessed by it. This means that we store a *copy* of the `key`, `value` and `version` that the resource had *when it was first seen* by T. When a transaction wants to access a resource, it is read/written in its private workspace. Moreover, we keep a `changed` variable to check whether that item was updated or not during the transaction.

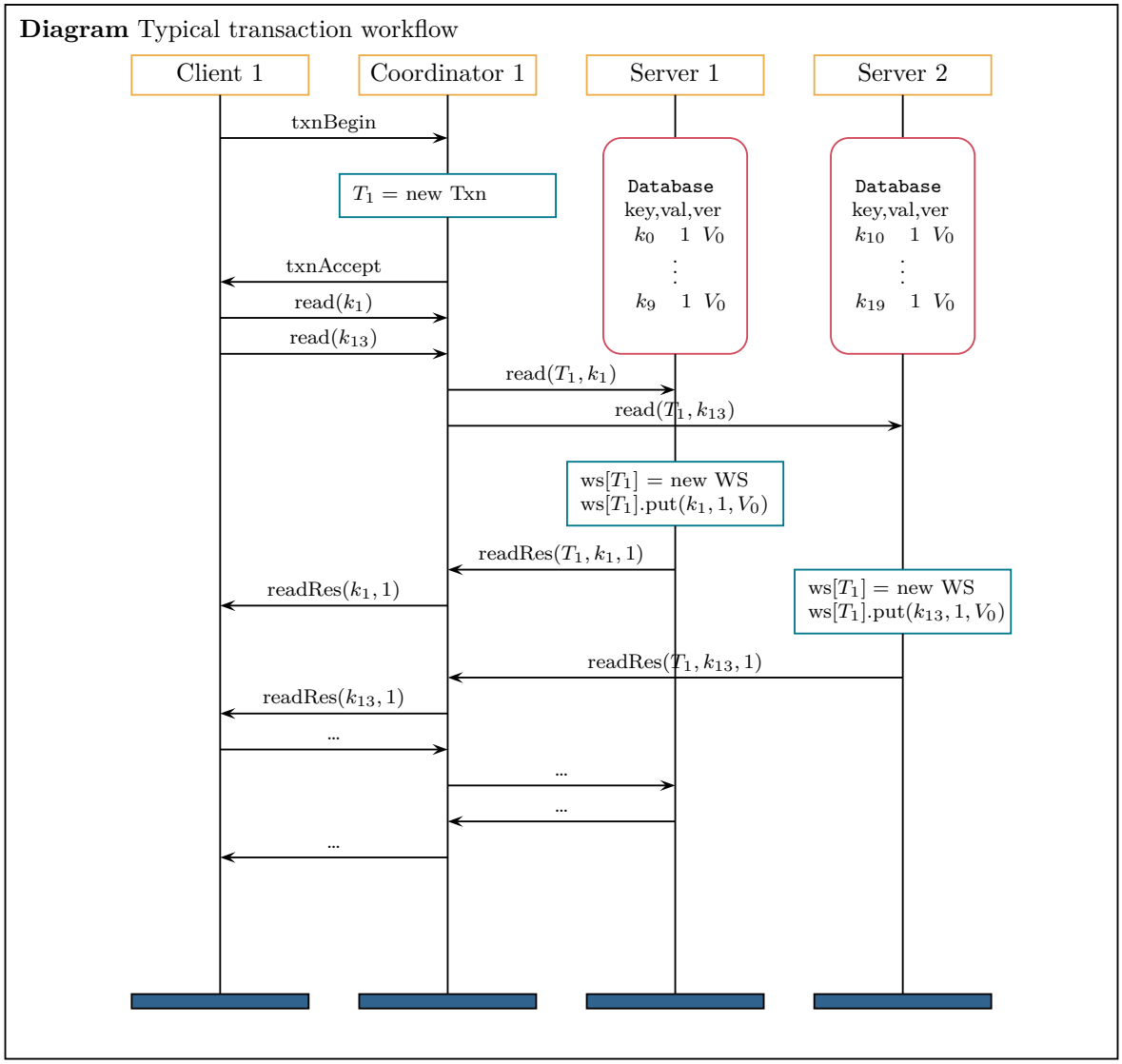

Figure 1: After the exchange of the txnBegin and txnAccept messages, the client can start sending read and write requests to the coordinator, which forwards them to the designated server. Each server will create a private workspace for the client transaction, where it reads and writes a copy of the requested resources.

2.3 Validation phase

Our protocol works just like a normal 2-PC protocol for what concerns the validation phase, with the addition of concurrent transactions. Once a client decides that it has no more operations to carry out during a transaction, it informs the coordinator about it. The coordinator will then proceed to ask each server that was involved in the transaction to vote on whether to commit or abort the changes.

Upon receiving a vote request, a server has to carry out a series of checks before voting:

- if any of the resources accessed by the transaction has a newer version in the official database with respect to the one in the private workspace, it aborts;
- if any of the resources contained in the workspace are *pending*, it aborts;
- otherwise, the server is free to vote commit on the transaction.

If the server votes to abort, it unilaterally aborts. If it votes to commit for a transaction, all the resources contained in the relative workspace are marked as *pending* until a decision is received from the coordinator. If the coordinator has decided to abort, the server needs to abort. If the decision was to commit, the server commits the transaction by updating the resources in the official database with the new values set by the transaction, and for each resource that was changed during the transaction, the corresponding version number is incremented.

2.4 Strict serializability

The private workspaces mechanism and the validation protocol we explained in Sections 2.2 and 2.3 guarantee strict serializability. In fact, when a transaction accesses a resource for the first time, it reads the last committed value.

Moreover, the fact that before voting to commit for a transaction we check if the versions of the resources it contains are up to date with the official database ensures that we commit only transactions that have handled the most recent item versions.

Lastly, in order to guarantee strict serializability we decide to vote abort for the transactions that involve resources accessed by other transactions for which we have voted to commit and not decided yet. This makes sure that once we have voted to commit for a transaction, no other transaction can modify the resources involved in the current *pending commit*.

2.5 Crash and recovery

Coordinators and servers can crash at significant points of the protocol and they recover after a while. We have implemented a system that allows us to somewhat control the moments in which we want coordinators and servers to crash with a non-zero probability. However, that moment matters only for one transaction, as all the other transactions managed by that host will crash in a different state. This is just a way to ensure that we can have hosts crashing at any point of interest of the protocol.

To ensure liveness, we implemented timeouts on every exchange of messages between actors. Let's outline the actions of servers and coordinators on timeout and recovery.

2.5.1 On timeout

Coordinator the coordinator sets a timeout for each interaction it has with a server about a transaction. If it meets a timeout it can safely abort, since the only timeouts that we set for coordinators are in order not to wait forever for server responses. This is safe to do, because we know that it is not possible for a server to commit without every server voting positively on a transaction.

Server if previously the server had voted commit on the transaction, start the termination protocol explained in Section 2.5.3. Otherwise, abort unilaterally.

2.5.2 On recovery

Coordinator abort every pending transaction for which no decision was taken.

Server for every pending transaction, if the server did not vote, abort unilaterally. Instead, if the server voted commit, start the termination protocol.

2.5.3 Termination protocol

A particular case of interest is the termination protocol for servers. Let's take into consideration the following scenario: for a certain transaction, the coordinator has decided to commit but it has crashed before telling any server about it. The servers involved in the transaction cannot unilaterally abort because this could cause inconsistencies, therefore they have to wait for the coordinator to recover.

Periodically, every server involved in the transaction asks the coordinator for the decision. It also asks to every other server involved in that transaction if they are aware of the decision, just in case the coordinator managed to send the decision only to a portion of the servers (or in case one of the servers unilaterally aborted). If any server is aware of the decision, it will spread to every other server eventually. Since by assumption we know that the coordinator recovers after a while, this protocol ensures liveness.

An example of validation phase where we need to start termination protocol due to the crash of a coordinator is shown in Figure 2.

3 Implementation

3.1 Assumptions

In order to be safe, our protocol does not need further assumptions from the ones stated in the requirements, namely reliable and FIFO links, clients that do not crash and servers and coordinators that recover after a while.

However, in order to ensure liveness, we have to assume that the timeouts that we have set are much greater than the network propagation delay of messages.

3.2 Network propagation delay

We simulate network propagation delay by introducing a `Thread.sleep()` line with a random delay bounded by a constant. This comes at a cost: genuine network delay would not make a machine sleep, it would be *asynchronous* delay. For example, if we have a fixed delay of 10ms and we have to multicast a message to 300 hosts, the machine would take 3 seconds to send the messages, while in a realistic scenario it would take much less time. This can potentially lead to problems with our timeouts system, and therefore we cannot have a large number of actors in the system.

We tried to schedule the messages instead of sleeping, but it led us to problems regarding non-FIFO communication of messages that violated our design assumptions. Therefore we settled for this alternative delay simulation.

3.3 Correctness

To make sure that our protocol is correct, we implemented the suggestion given in the presentation: since write operations are always carried out as a couple of operations which sum to zero, the total sum of all the data in our database should always be the same. Therefore, we have a `Checker` actor that asks all the servers for the sum of their values at the end of the execution of our program. Once it receives all the data, it computes the total sum to compare it to the expected result. In order to make sure all the transactions are finished when we perform this check, we first send a stop message to all the clients and ask the servers for the information only after a reasonably long time. This actor is also useful for retrieving logs from the actors of our system once the execution is over.

4 Possible improvements

As stated in Section 3.2, an improvement to our project would be to simulate network delay in an asynchronous way that does not tamper with the performance of the host machine. Furthermore, it would be interesting to try out different locking approaches that were not part of this project to try out different design trade-offs, like pessimistic timestamp ordering or 2-phase locking. Also, implementing the 3-phase commit protocol for transaction validation would be interesting to see how it performs under the same conditions when compared to our 2-phase commit protocol.

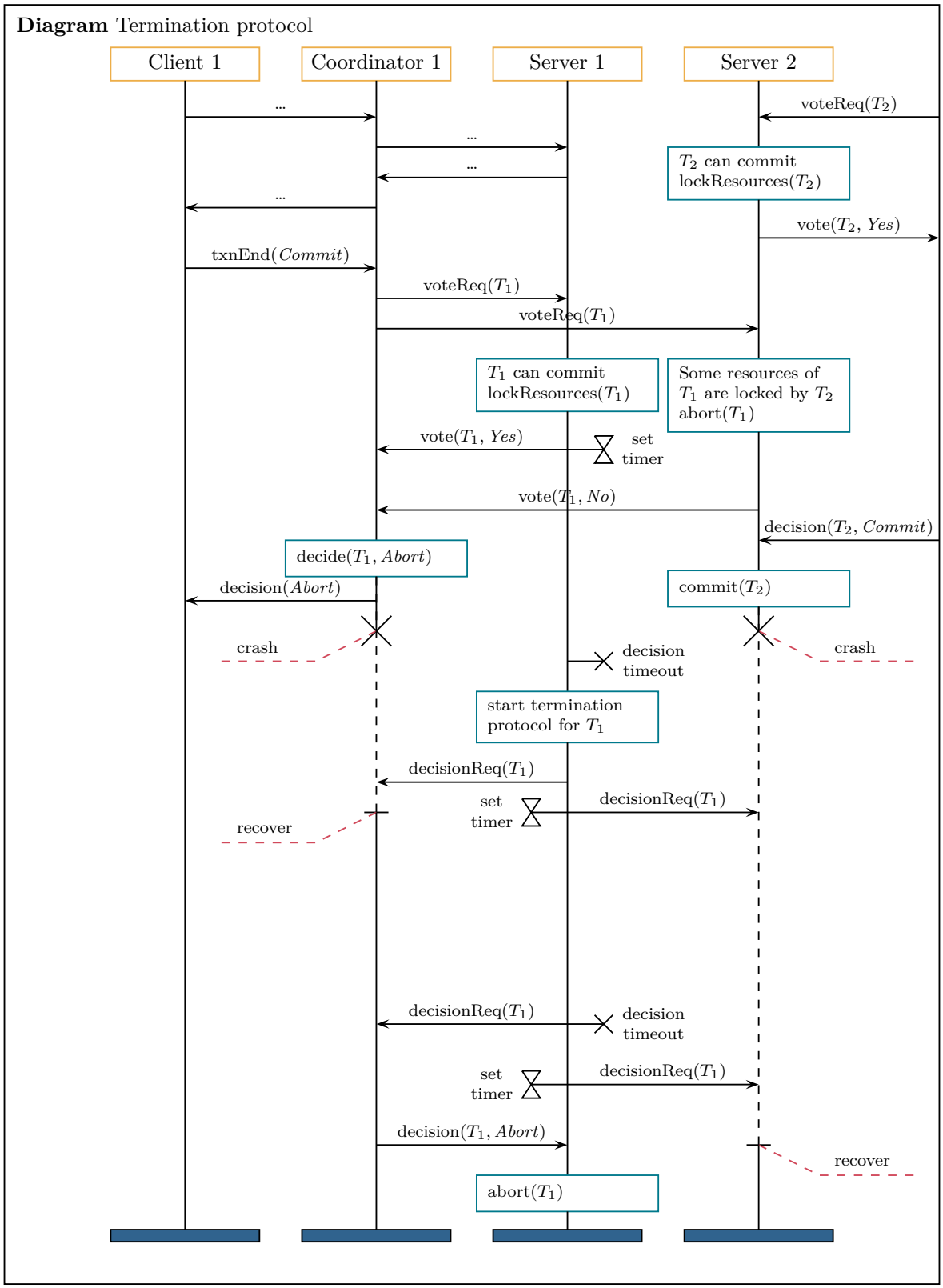

Figure 2: On receiving a vote request, Server 2 detects a conflict, aborts and sends the vote to the coordinator. Server 1 has no conflicts, so it votes to commit. However, if it does not receive the decision after some time, it asks for it to both the coordinator and Server 2. If nobody answers, because they are crashed or do not know the decision, after some time Server 1 asks again, until someone answers.