Department of Information
Engineering and Computer Science

University of Trento

Distributed Systems
A.Y. 2019/2020

Quorum-based Total Order Broadcast

Daniele Tapparelli **214235**
Francesco Penasa **215038**

1 Introduction

This report briefly describes the implementation of the project assigned in the distributed systems 1 course. The main goal was the creation of a protocol for the coordination of a group of replicas, sharing the same data, that tolerates multiple node failures thanks to a quorum-based approach.

Basically, the system is formed by two groups of actors: external clients that can issue read and write requests and replicas that handle them. When a particular client sends a read request, the specific replica will reply immediately with its local value. Instead, write requests are handled in a different way. There is a special type of replica called coordinator that guarantees that the system applies updates in a consistent way. The update will be forwarded from a replica to the coordinator that will use a quorum-based approach to decide if it will be applied or not. A second feature of this system is the tolerance of multiple node failures. Essentially, when a particular replica crashes, the system can continue its work, until there is a quorum that can answer. Otherwise, if the coordinator crashes, the remaining replicas hold an election, using a ring-based protocol.

2 Implementation

2.1 Two-phase Broadcast

An external client node is able to query a replica with a read or a write request. By design the replica should be able to answer the read request with the local value and to ask for an update when a write request is received. The update is performed by forwarding the client request to the replica's coordinator whom will ask for a vote on the new value. To maintain track of the update history the coordinator has to update the epoch (`onDecisionResponse`). The update is then confirmed for all nodes when at least the quorum of votes have been received by the coordinator. If an update is already been performed then the Coordinator's Vote Request is locked to new requests.

We used a stack of pair to mantain the history of the update which contain all the timestamps with the epoch and the sequence number of such epoch.

2.2 Timeouts and Crash Detection

The timeouts have been set at some fixed value through constants and all the messages sent have a fixed delay. In our design choices we opted for a set of boolean variables that activate the crashes at some specific actions of the code. By changing the values of such variables we can decide in which case simulate a crash. The timeout handlers that we implemented give the possibility to our replicas to understand if the coordinator or other replicas are crashed. For instance, if the coordinator crashes during the decision phase for a new value, the replicas can detect such crash through a timeout on the decision response, and then start a new coordinator election message.

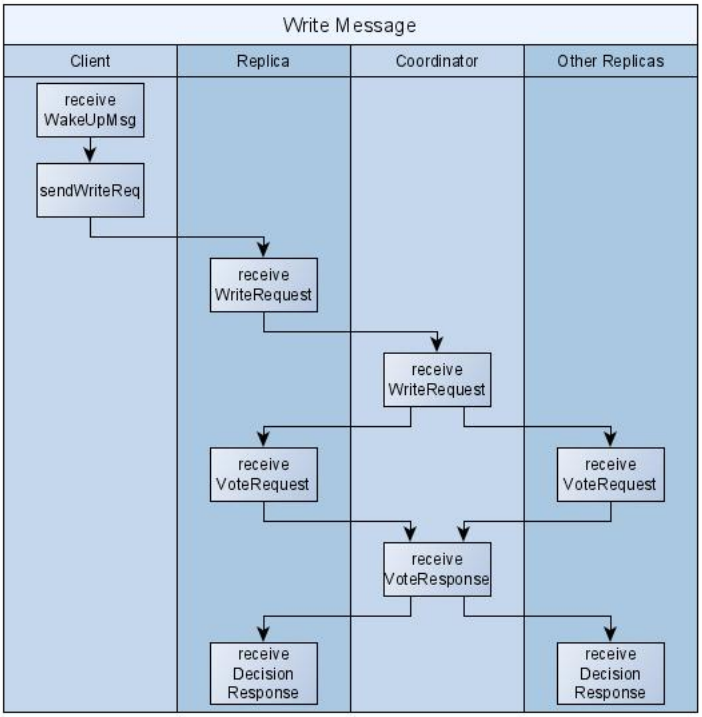

Figure 1: Exchange of message to preform a Write Request

2.3 Coordinator Election

In order to maintain the system running we also need a coordinator election. To deal with this problem, we decided to send a particular message when a replica detects some particular timeouts, like heartbeat.

To be more precise, when the coordinator crashes, a new message called `CoordinatorElectionMessage` is forwarded by the replica who detects it, to the next one, following a ring-based protocol. The message contains a list of `timestamp`(epoch and sequence number of a particular replica) in order to keep track of who has the latest update. In the first round every replica updates the message adding its own timestamp. In the second round, replicas check if they have the highest sequence number, using the list. If there are more than one with the last update, the chosen replica will be the one with the highest id. When the coordinator is found, replicas stop forwarding the election message. The elected coordinator sends a `SynchronizationMessage` to the remaining replicas, adding its own id and the last update value and every receiving specific actor checks the presence of its id inside `indexesOfReplicaWithoutUpdate` list. If there is a match the replica updates its variable with the last update, otherwise it skips it. Finally, all the replicas update the coordinator ID, using the specific value inside the `SynchronizationMessage`. To avoid blocks on election, we decided to insert an election timeout, to be able to resend the `CoordinatorElectionMessage`. Indeed, if a replica is no longer reachable by its neighbour, there can be an election block, and we need to find an active one following the ring.

In our design choices we decided that the coordinator can crash only once at each run, so the entire election will only appear when the first crash happens.

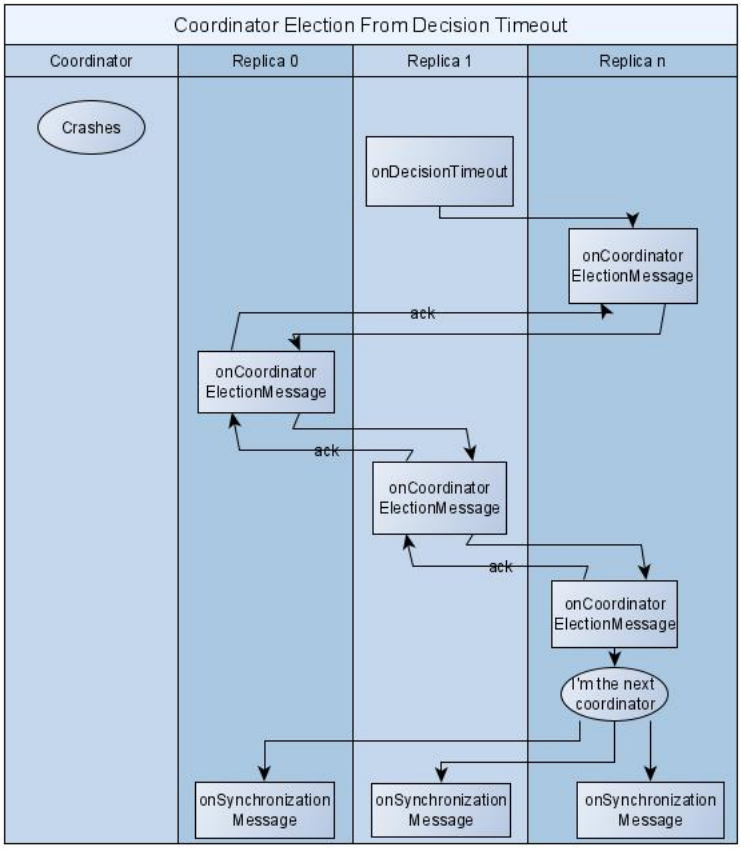

Figure 2: Coordinator election