Course Project: Quorum-based Total Order Broadcast

Distributed Systems 1, 2019 – 2020

Team members: Davide Tessarolo, Devis Dal Moro

# Introduction

In this report, we’ll explain the architectural and implementation details regarding the development 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. Details on how the protocol works are not dealt within this document and are taken for granted, since they are well explained in the requirements document produced by the teaching instructors.

# Client

The client actor has just two simple needs which consist of reading and writing the value stored by the replicas. In both cases, there is a random selection of the replica that it’s going to query. Indeed, it knows from the beginning all the replicas that should be available, but it doesn’t know if they actually are. Thus, whenever it sends a read or a write request, it cannot be sure its request is going to be fulfilled.   
Specifically, the write request happens in a totally asynchronous mode, in the sense that it’s completely unaware of the fact that the request has actually been received and then processed by the distributed storage system. While, in the case of a failed read (which occurs because the random selection picked up a crashed replica) there is a timeout set up specifically for communicating the failure. Read requests and corresponding timeouts are associated to an internal (wrt client) incrementing id which is passed along in the request itself. If the response arrived, it’d be clear to which read request it’s related to, because the replica would have put the read id in the very same response.

# Replica

All replicas hold an integer value and a flag which to denote the fact that a specific replica should act as a coordinator. At the start, the coordinator is predetermined (simply the one with the greater id), but along the different epochs it changes as a result of the coordinator crash in the previous epoch. However, as result of the architecture and processes detailed below, it’s impossible to have simultaneously more than one coordinator. Furthermore, each non-faulty replica holds the reference to the current coordinator.

Since any replica could be in a future epoch a coordinator and will need to multicast/broadcast messages, when the system boots up a “JoinGroup” message is delivered to every member, notifying about all the replicas and the first coordinator.

Whether the replica is the coordinator or not, it does need an internal clock, which consists of a data structure created ad-hoc containing two integers: one for the epoch and the other for the sequence number. Its value corresponds to the number of delivered updates by the replica. Specifically, the coordinator will also need an hashmap in order to know the amount of UPDATEACK received for each UPDATE message. In the latter scenario, we could use the corresponding clock value as key. Further, all replicas retain the FIFO list of the pending updates, waiting for the WRITEOK broadcasted from the coordinator.

Specific flags are embedded in their structure, so they can leverage them to know if an election or a synchronization phase is in progress, as well as if they are crashed, because the state a replica finds itself in influences its reaction to the receipt of a message. Other data structures worth to mention are the hashmaps needed in order to handle the timeouts related to the waiting of WRITEOK messages (using the update as key) or of the ACK of election messages (using an incrementing counter valid for the replica during that election session as key). Finally, two other individual timeouts are needed to cover the following two cases: the non-arrival of the coordinator’s heartbeat or the complete failure of an election session.

### Dealing with requests

As per project’s requirements, all non-faulty replicas respond to read requests with their current stored value, without the need of further interaction. Write requests are instead more complex and require the forwarding of the request itself to the coordinator (assuming there is no election session going on). Whether the coordinator is the direct or indirect receiver of a request message, it doesn’t make any difference to how the latter is processed: the clock is updated to the following sequence number and the UPDATE with the new requested value can be packed and broadcasted to all the replicas[[1]](#footnote-1).

### Delivering updates

Upon the receiving of an UPDATE message, a non-faulty replica adds the update at the bottom of the pending updates queue, sends the UPDATEACK message back to the coordinator (containing the update itself) and sets up a timer which notifies the replica in case of missing WRITEOK message (i.e. coordinator fault detection).

When the coordinator receives an UPDATEACK, it needs to take in account the number of acks already received for that same update, using the update info contained in the UPDATEACK message itself and the ack hashmap mentioned before. When the quorum is reached, the WRITEOK message is broadcasted to all the replicas.

Upon the receiving of a WRITEOK message, a replica cancels the timeout set-up for that update when sending the corresponding UPDATEACK and deletes it from the relative hashmap, because now it can pickup the update from the pending updates list, remove it and deliver it (e.g. changing the value of the stored integer). Note that the delivery of an update is always logged to check the total order at the end of the simulation and triggers the update of the internal clock of the replicas which are not coordinator.

### Heartbeat

From the moment a replica becomes coordinator, it sets up a timer which, like all the timers mentioned in this report, is a message to itself and in this specific case triggers the coordinator to multicast an heartbeat message to ensure the other members of its availability status. Instead all the other non-faulty replicas, use the same scheduling mechanism (with a greater value in the interval due to accepted simulated delays in the message transmission) to notify themselves in case of a lack in the receiving of the heartbeat. This behaviour comes useful to detect a coordinator failure also outside the UPDATE-UPDATEACK-WRITEOK process. If the heartbeat is received as it should be in the normal scenario where the coordinator does not fail, the mentioned timer is simply refreshed.

### Crash and election

A crash is triggered by a special message, which can be customized allowing to specify the point in which it’s preferable that the destination replica will fail.

The crash of any replica which is not the coordinator has no direct implication on the system[[2]](#footnote-2), since we’re assuming a minimum amount corresponding to the quorum is always available. Furthermore, no recovery is contemplated in this implementation of the protocol, as specified in the requirements.

The failure detection of the coordinator, triggered by timers awaiting for a WRITEOK or a heartbeat message (as explained above), makes the replica initiate the election process. Note that many replicas could detect the failure of the coordinator at more or less the same time, thus initiate the election together, but this will not be a problem for the convergence of the algorithm. The initialization consists in the following steps:

* Cancel timer for heartbeat and potential timers for pending updates
* Initiate timer to detect the total failure of the election process[[3]](#footnote-3)
* Reset internal counter of ids for election message that will need to be sent
* Put itself in election mode (set up the value of the corresponding flag)
* Sends the first ELECTION message to the following replica in the overlay ring[[4]](#footnote-4) packed up with the potential pending updates and the reference to itself as candidate, setting up the timer ensuring the future receive of the ACK for this ELECTION message

All these first steps are performed also when any non-faulty replica receives an election message and it’s not been in election mode yet. Actually, in this case the forwarding of the ELECTION message requires an articulated and well thought set of checks to take into account all the different scenarios and packed it up accordingly. Let’s get more into the details:

* If the ELECTION message has already performed a round and the replica which is receiving is a plausible candidate and has the highest ID among those, the SYNCHRONIZATION process can begin, otherwise the message will be just passed to the next one and eventually will reach the candidate with the higher ID;
* Otherwise, the ELECTION message needs to be passed along and, in case, repacked accordingly:
  + If the replica has no pending updates to perform and the ELECTION message as well doesn’t present any last update OR if the replica’s most recent pending update is equal to the one in the ELECTION message, the replica will just add itself to the list of possible candidates;
  + If the replica has pending updates to perform, while in the ELECTION message there is no sign of the most recent one OR it is older than the most recent one within the replica’s pending update list, the replica will refresh the entire list of possible candidates by just putting itself and a reference to its most recent pending update;
  + If the replica has no pending updates to perform and the ELECTION message has OR if the most recent one within the replica’s list is older than the one in the ELECTION message, the replica will just pass along the message.

In any case, after sending an ELECTION message a specific timer will be set up and inserted in the hashmap using as key its ID which has a meaning wrt. the sender and can be cancelled when the latter receives the ELECTION\_ACK message for it (this can be indeed recognized through the ID). If the timeout is triggered, the following replica within the ring will be considered crashed and the same ELECTION message (different ID) has to be forwarded to the next one. IDs, timers and hashmap need to be set accordingly.

When the replica will recognize itself as the best candidate after a complete round of ELECTION, it’ll put itself to synchronization mode and broadcast the SYNCHRONIZATION message: containing all the pending updates that need to be performed before stepping into the next epoch in which the very same replica will be the coordinator. Upon the receiving of the synchronization message, all the replicas will adjust the heartbeat timer, updates to be delivered, coordinator flag and references, current value to be stored and current epoch accordingly. Election and synchronization session will be considered complete.

1. Note that the coordinator sends the UPDATE also to itself, so the handling of this message is symmetric and universal to all the replicas (e.g. responding back with the UPDATEACK and adding it to the pending updates) [↑](#footnote-ref-1)
2. Just the unfortunate client actor which randomly selects a crashed replica for the request will notice it [↑](#footnote-ref-2)
3. Note that this will have a time proportional to the number of replicas in order to be an appropriate value [↑](#footnote-ref-3)
4. Selection of the following is performed by considering their ids, skipping the coordinator for this epoch (because if we’re in an election, it is crashed for sure) [↑](#footnote-ref-4)