Quorum-based Total Order Broadcast

Distributed Systems 1 - 2019/2020

Quintino Francesco Lotito
215032
University of Trento
Trento, Italy
quintino.lotito@studenti.unitn.it

Davide Zanella
211463
University of Trento
Trento, Italy
davide.zanella-1@studenti.unitn.it

I. Introduction

This report aims at describing our main design and implementation choices behind the solution of the project of the Distributed Systems 1 (A.Y. 2019 / 2020) course.

We are requested to implement a system made up of two main actors: the *clients* and the *replicas*. All the replicas share the same data and allow external clients to access it through updating and reading requests. Updating requests are managed by a *coordinator*, which is a special replica that, with a quorum-based two-phase broadcast procedure, ensures consistency in the data stored by all the replicas.

The system needs also to handle multiple node failures, which should be detected based on timeouts. If in a given time the coordinator crashes, a new election takes place to guarantee the continuity of the service. If a replica crashes, the coordinator is still able to handle the updating process, as long as a quorum of the replicas is active (we assume this condition always holds). No new replicas can join and, after a crash, an actor does not recover.

II. Implementation

The project has been implemented in *Akka*. Each agent of the system is an Akka actor identified by a unique ID.

For simplicity, the data shared by the replicas is assumed to be a single variable.

There are two main classes: one representing Clients and one representing Replicas.

Each client follows a schedule in which it self-sends a special message every fixed amount of time (*scheduleWithFixedDelay* function). When receiving this message, the client is required to take action: it chooses a random Replica and sends it a request randomly choosing between writing or reading. Writing requests are issued attaching a randomly generated alphanumerical string.

On the other hand, each replica have an ArrayList to save the updates (i.e., every *WriteOK* message applied is stored in this array). *Timeouts* are managed by the class Timer, allowing actions to be performed after a fixed amount of time. Since the design and implementation choices behind replicas methods and algorithms are of particular interest, we will explain them with more details in the following section.

Other classes are used in our project: each of them represents a different type of message sent between actors.

III. Main design choices

In this section we enumerate a number of design and implementation choices, which, in our opinion, are of particular interest.

A. Elections blocked forever due to crashed replicas

A critical problem to solve is to avoid the elections to be blocked forever when a crash in two successive replicas of the ring occur. During the election phase, every replica forwards an election message and tries to reach the next neighbor of the ring. If the receiver does not answer after a fixed amount of time, the sender replica assumes a crash occurred in the receiver, and forwards the message to the replica following the crashed one in the ring. This algorithm repeats until one replica answers to the election message with an *election ACK*.

This technique avoids the system to be stuck forever due to some crashes during the election phase.

A similar problem occurs when the best candidate replica crashes during the election of the coordinator, after having proposed itself. To avoid the system to block, we proceed in the following way. Election messages are enriched with an array of boolean values, each replica has its own boolean value. If a replica receives an election message and it is not the best candidate, instead of just forwarding the message to another replica, it firstly checks if that message has already been seen. This can be decided by looking at the corresponding boolean value in the in the election message. If the message has done more than one cycle of the ring after everyone voted, it means that the best candidate is crashed before becoming the new coordinator, and a new election starts.

B. Writes are always applied in order

Due to delays in sending messages, two replicas may receive the same WriteOK message in a different order. To deal with this issue, every time a WriteOK message is received, it is pushed into a *PriorityBlockingQueue*. Every time an element is extracted from the queue, it will correspond to the write that has been requested before in terms of epoch and sequence number. After the insertion of the new WriteOK message in

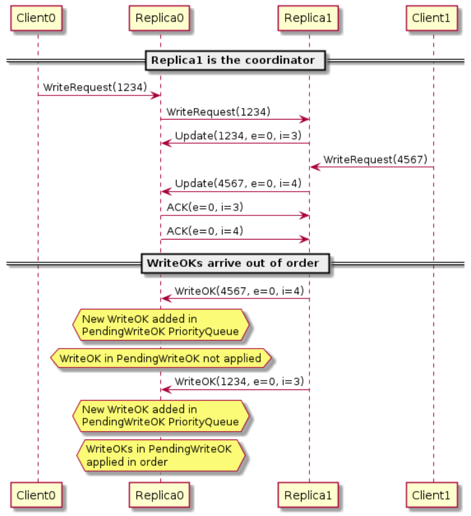
Client0
Replica0
Replica1
Client1
Replica1 is the coordinator
WriteRequest(1234)
WriteRequest(1234)
Update(1234, e=0, i=3)
WriteRequest(4567)
Update(4567, e=0, i=4)
ACK(e=0, i=3)
ACK(e=0, i=4)
WriteOKs arrive out of order
WriteOK(4567, e=0, i=4)
New WriteOK added in PendingWriteOK PriorityQueue
WriteOK in PendingWriteOK not applied
WriteOK(1234, e=0, i=3)
New WriteOK added in PendingWriteOK PriorityQueue
WriteOKs in PendingWriteOK applied in order

Fig. 1.

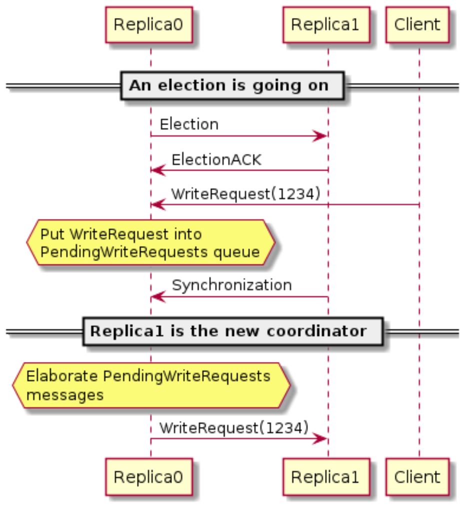
Replica0
Replica1
Client
An election is going on
Election
ElectionACK
WriteRequest(1234)
Put WriteRequest into PendingWriteRequests queue
Synchronization
Replica1 is the new coordinator
Elaborate PendingWriteRequests messages
WriteRequest(1234)

Fig. 2.

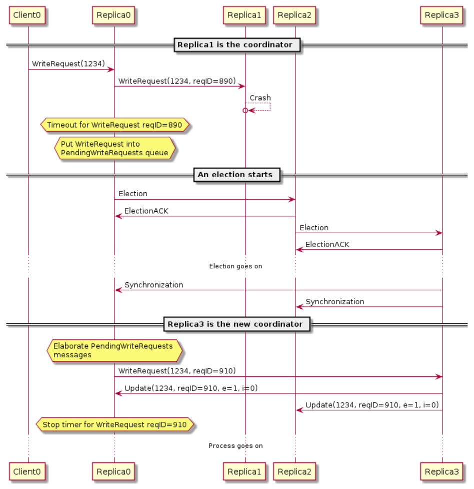
Client0
Replica0
Replica1
Replica2
Replica3
Replica1 is the coordinator
WriteRequest(1234)
WriteRequest(1234, reqID=890)
Crash
Timeout for WriteRequest reqID=890
Put WriteRequest into PendingWriteRequests queue
An election starts
Election
ElectionACK
Election
ElectionACK
Election goes on
Synchronization
Synchronization
Replica3 is the new coordinator
Elaborate PendingWriteRequests messages
WriteRequest(1234, reqID=910)
Update(1234, reqID=910, e=1, i=0)
Update(1234, reqID=910, e=1, i=0)
Stop timer for WriteRequest reqID=910
Process goes on

Fig. 3.

the priority queue, all the messages of the queue are iterated to check if some of them can be applied. To apply an update, the first update of the queue happens in the same epoch of the last update done and its sequence number is the immediately following one.

With this procedure, we are able to apply updates in the same order they were issued by the coordinator. Figure 1 shows how the protocol works.

C. *Avoid losing write requests*

Due to particular phases of the system, some messages containing a new value for the replicas may get lost.

We outline the principal circumstances where it can happen.

1) Write requests from clients while in election: During an election, some clients may send a write request to a replica. To manage this scenario, our system puts these messages in an ArrayDeque object called *PendingWriteRequests*, to preserve the arrival order (FIFO). The requests in this Queue will be managed by the replica only when the election is over, in particular, when a Synchronization message is received. Figure 2 shows how the protocol works.

2) The coordinator crashes between the Broadcast and the Update phase: When a replica forwards a WriteRequest to the coordinator, it sets a request ID property in the message and starts a specific timer for it. If this timer gets triggered, meaning that the timeout is over and no Update message is received containing the same request ID set before, the replica knows that the coordinator is crashed. A new election takes place and we put the original WriteRequest into the previously mentioned PendingWriteRequests ArrayDeque. This way of proceeding guarantees that the WriteRequest will be send again to the new coordinator when the election will end. To store the WriteRequests, we use a *ConcurrentHashMap*, using the request ID as the key. Figure 3 shows how the protocol works.

3) The coordinator crashes between the Update and the WriteOK phases: Similarly to the previous situation, an Update may arrive to a replica without being followed by a WriteOK message, due to a crash of the coordinator. To link an Update message to its WriteOK one, we use the epoch and the sequence number present in both the messages. We use a timer to check crashes of the coordinator. If the timer is triggered, a coordinator election starts. If the replica is the one that forwarded the original WriteRequest linked to the Update one, it will store the WriteRequest in its PendingWriteRequests ArrayDeque, in order to resend it later. A ConcurrentHashMap stores the original Update messages, because, otherwise, if the crashed coordinator was the replica that firstly managed the WriteRequest, the Update will get lost. When an election ends, the new coordinator manages all the Update messages present in the ConcurrentHashMap, which are the Updates started from the old coordinator but never applied. Figure 4 shows how the protocol works.

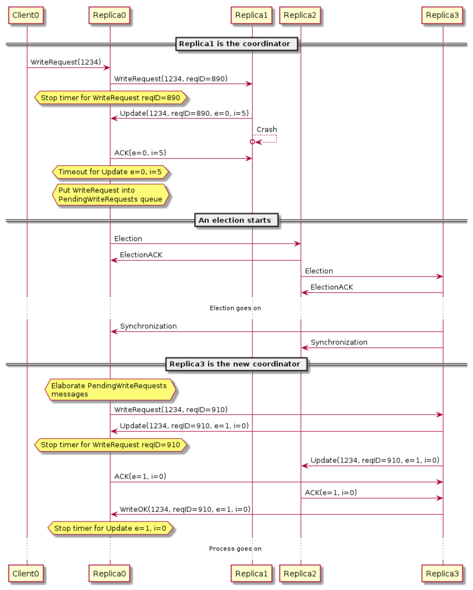
Client0
Replica0
Replica1
Replica2
Replica3
Replica1 is the coordinator
WriteRequest(1234)
WriteRequest(1234, reqID=890)
Stop timer for WriteRequest reqID=890
Update(1234, reqID=890, e=0, i=5)
Crash
ACK(e=0, i=5)
Timeout for Update e=0, i=5
Put WriteRequest into PendingWriteRequests queue
An election starts
Election
ElectionACK
Election
ElectionACK
Election goes on
Synchronization
Synchronization
Replica3 is the new coordinator
Elaborate PendingWriteRequests messages
WriteRequest(1234, reqID=910)
Update(1234, reqID=910, e=1, i=0)
Stop timer for WriteRequest reqID=910
Update(1234, reqID=910, e=1, i=0)
ACK(e=1, i=0)
ACK(e=1, i=0)
WriteOK(1234, reqID=910, e=1, i=0)
Stop timer for Update e=1, i=0
Process goes on
Client0
Replica0
Replica1
Replica2
Replica3

Fig. 4.

D. *Crash simulation*

To simulate crashes in specific phases of the system, we created a Crash message. Replicas are able to self-send *Crash messages* in some parts of the code. In particular, if in a given moment or situation, the ID of a replica is present in a target set (containing the IDs of the nodes required to crash), that node will crash. Thanks to this technique, we can schedule the crash of every node at the desired point of the protocol.

In particular, we can decide to make a crash happen for a specific replica while sending an Update message, or when receiving it, or when delivering an ACK message. Moreover, a replica can crash in a scheduled way, after transmitting a WriteOK message, or when receiving it. To test the election protocol, we can make a replica crash when receiving an Election message, or when the best candidate should send the Synchronization message to all the other replicas.