Mores Nicola, Roccon Marco

Distributed System Project

Introduction

In the assigned Project we were required to implement a replication protocol in order to share a common value among a ring-based network of nodes, handling the possibility of crashes. To do so we will distinguish two main roles among the nodes: a coordinator and many cohorts. An external client will be able to interact with our system by sending to one of the nodes a write or read request. In case of a read request, the node will simply respond with the current value shared among the replicas; if we are dealing with a write operation, the request will need to be processed by the coordinator (thus, if a cohort receives the request, it will forward it to the coordinator). After receiving a write request, the coordinator will use a Quorum-based Total Order Broadcast to update all the replicas. The protocol will also use an Election protocol to deal with coordinator's crashes: leveraging the ring topology, we will notify each node of the crash, requiring the election of a new coordinator. Among the active nodes, the most up-to-date one (break eventual ties) will be elected as the current coordinator: it will then propagate his knowledge to the active cohorts and return to its normal behavior.

Crashes to Handle

By reasoning on the system at hand, we identified many notable situations that needed particular attention, especially when talking about crashes. In our implementation, we will handle crashes by using timeouts, that will keep track of missing messages that should be received by a node after specific actions. In particular, we will deal with four main timeouts categories:

- (1) Heartbeat Timeout: In order to check if the coordinator is still active, each cohort will wait for a periodical message (Heartbeat) from it. We will keep track of this event using a timeout: we will reset it every time we receive this type of message and if it runs out, the coordinator will be considered as crashed and so the system will start an election phase.
- (2) Update Forwarding Timeout: If a cohort receives a write request from the client, it will start a timeout after forwarding it to the coordinator. We will wait for a feedback from the coordinator, in means of an update request, and if this doesn't happen in a limited amount of time, we will consider the coordinator as crashed, starting also in this case the election process.
- (3) WriteOk Timeout: Once a cohort expresses its approval for an update request, it will start a timeout, removed once it receives a corresponding update confirmation message (WriteOk), stating that the coordinator has proceeded with overwriting the common value. As usual, if the timeout runs out, we will start a new election.
- Election Timeouts: We might face node's crashes even during the election process. Being this the case, we will need to:
    - (4) Handle the possibility of crashed nodes in the ring topology: While we are propagating, step-by-step, the election request message through the ring, we will use an acknowledgement to confirm the reception of our election request

to the next node. To do so we will use once again a timeout that runs out if it doesn't receive the ACK, leading to the removal of the node from the list of active ones and to a retransmission of the request to the node after it.
    - (5) Guarantee an eventual end to the process: In order to have a certain termination of the process, each node involved in the election will start a timeout that runs out if it doesn't receive a synchronization message in a limited amount of time.

Code Implementation & Architectural Choices

During the implementation of the project, we divided the workflow in two main phases: one focusing on the Update process and one on the Election process.
First of all we setted up the system, defining the characteristic needed by a Node (such as its ID, its current value for the common variable and the list of the other participants), some utility functions (like multicast, print, crash…) and then focusing on the various messages that we will need to make the system works.
Starting from the update phase and in particular the write operation, the client will communicate with a replica using a IssueWrite message, containing the value to set. If a cohort receives it, it will forward it to the coordinator and set a new timeout in the forwardingTimeoutList (2). In this case we will use a list of timeouts so that we can keep track of multiple messages. Once there, the coordinator will create an UpdateRequest message, containing an Update object, and it will multicast (which, in our implementation, will send a message also to the sender itself) to all the nodes in the system. The Update object we defined is used to contain both the identifier <e, i> (stored in a UpdateIdentifier object and created only by the coordinator) and the new value. When a node receives an UpdateRequest message it will store its information in a data structure called pendingUpdates, used to keep track of updates not yet completed. This structure will contain the UpdateIdentifier and a PendingUpdateTuple, a structure present in every node, but used only by the current coordinator to handle the ACKs (UpdateResponse) corresponding to each UpdateRequest. To detect a possible crash, this time we will use writeOkTimeout (3), a map to handle a timeout for each update.
When the coordinator receives an UpdateResponse it updates this structure and checks if the quorum is reached. In this case it will create a WriteOk message, containing the UpdateIdentifier it refers to, and send it in multicast. Notice that this transmission will be done only the first time the quorum is achieved. When a node receives a WriteOk message, it will remove the corresponding Update from the pendingUpdates structure, add it to completedUpdates, a simple list of Update, and update both its i and v.

Talking now about the Election phase, we implemented two "modes" for the node involved in the system: CreateReceive and ElectionMode. This necessity arises because we don't want to handle any incoming updates while in the election phase. This property is guaranteed by switching from the normal mode (CreateReceive) to the ElectionMode. The election process will start whenever we detect a crashed coordinator through a timeout running out. While reasoning on the system we noticed that, if not properly managed, we might face the possibility of having multiple elections running simultaneously, in particular this is the case whenever the coordinator crashes due to the missing heartbeat, that will lead to a cascade of expired timeouts (notice that this is only the most frequent occurrence, not the only one).

To avoid this type of problem we decided to slightly change how the election process starts. Whenever a node detects a crashed coordinator, it will enter the ElectionMode and then it will send an ElectionInit message to the first node in the ring topology. This message is used to delegate the task of initiating the election process, so that we are sure that only one active node can start a new election. If a node receives an ElectionInit request, it will activate ElectionMode, send an ACK (ElctionInitAck) and start a new election sending an ElectionRequest to the next active node in the ring. However, if it receives an ElectionRequest while it is already in this mode, it won't start a new one, but it will just reply with an ACK. To handle a possible node crash in the ring topology, after sending a ElectionRequest we will start electionTimeout (4), a timeout (similarly for ElectionInit, with electionInitTimeout) that will run out if we don't receive a ElectionResponse (or an ElectionInitAck). Also, to guarantee that the election phase does in fact start, we will also setup electionStart, another timeout that will run out if we don't receive a ElectionRequest after receiving a ElectionInitAck (for example, this might happen if the first active node crashes before sending any ElectionRequest). In this case, the node in which the timeout expires will send another ElectionInit. Another special case that might occur is when the first node sends the ElectionRequest and then crashes. In such case, we can face two different situation: if the coordinator's crash is detected only by one node, so that only one ElectionInit will be sent to the first node, the election will proceed normally and we will simply elect a new coordinator or start a new election (since the electionCompleted (5) timeout will run out) if the most up-to-date node was the one that initiated the election (and, once again, that crashed);
However if the coordinator's crash was detected by other nodes, let's call one of them A, we might have multiple ElectionInit going through the network. If the first active node B, crashes early, this could lead to a transmission of another ElctionInit message from a different node towards the new first active node C (i.e, node A sends an ElectionInit to C), that might already have received an ElectionRequest. Since the whole reason for this digression is to not have multiple elections running at the same time, we obviously don't want to start another one right away. To do so, the node C, that will receive the ElectionInit after a ElectionRequest (this will be kept track of using the initStarted variable, that will be resetted after receiving a Synchronization message or when electionCompleted runs out), will simply reply with an ElectionInitAck. In both cases, every node that will receive for the second time an ElectionRequest, will check if it is the most up-to-date (breaking eventual ties by choosing the node with the lowest ID) and in this case it will proceed with the synchronization, otherwise it will just forward the ElectionRequest to the following node.
The elected node will set itself as the new coordinator and multicast a Synchronization message, containing the currently active nodes and the list of completedUpdates. Once a node receives a Synchronization message, it will update its <e, i> tuple, overwrite the current coordinator and the completedUpdates structure and, if it is a simple cohort, it will also clear the pendingUpdates one. We can safely do this because, being the elected node the most up-to-date one, we are sure that its pending updates won't have been completed by any node in the system. So, if the coordinator has any pending update, it will use the previously implemented UpdateRequest to initiate a proper amount of update procedures. Since these updates will be completed in the new epoch the i in the update identifiers won't start from 0 but from |previouslyPendingUpdates|.

GitHub code: https://gitfront.io/r/NikoMrs/NAiJjrr9fQk1/DistributedSystemProject/