Distributed Systems Report

Blascovich Alessio, Cereser Lorenzo

11th August 2024

University of Trento
Department of Information Engineering and Computer Science - DISI
Via Sommarive, 9 I-38123
Povo (TN), Italy

1 Runtime engine

Mimicking a classical modular application structure (MVC Java Architectural Pattern), our runtime engine is divided into three main classes and one interface :

1.1 User interface

`it.unitn.disi.ds1.qtop.UserInterface.java`

The User interface class handles the menus that are shown to the user, one menu for the setting of the various runtime parameters (e.g., various timeouts, number of nodes, number of clients, etc.), and another one for handling events while the nodes are running.

The class inherits from the Simulation callback interface to allow the class to communicate with the Simulation class.

1.2 Simulation

`it.unitn.disi.ds1.qtop.Simulation.java`

The simulation class is responsible for interacting with the program while it runs, it provides initialization functionalities for the whole Actors group (including clients), as well as managing crash insertions.

1.3 Simulation callback

`it.unitn.disi.ds1.qtop.SimulationCallback.java`

The simulation callback interface represents a convenient way of managing communication between various classes at runtime while ensuring encapsulation and reliability of classes.

1.4 Controller

`it.unitn.disi.ds1.qtop.Controller.java`

The Controller class orchestrates communication between Simulation and User interface; both of those components in fact do not communicate between each other to encourage encapsulation, ease up the location and functionalities, as well as helping in keeping the code tidy during development.

2 Auxiliary data structures

2.1 Pairs History

`it.unitn.disi.ds1.qtop.PairsHistory.java`

To maintain a certain degree of consistency, the nodes had to keep track of the various updates proposed by the coordinator.

We decided to use a matrix to represent the history of updates. The epoch and sequence are used as indices to access and update the matrix. Every matrix's cell contains a tuple with the value proposed and the commitment state.

The commitment state can be either "PENDING", "WRITEOK" or "ABORT". The latter two flags an update as already decided by the coordinator, respectively, committed or discarded. The former indicates that an update is idle, waiting for a decision from the coordinator.

The use of a matrix has been motivated, by the intuitive use of the update pair elements as indices of the structure. We could have used a map, however, it seemed an over-abstracted and over-complicated data structure compared to a matrix.

2.2 Timeuots manager

`it.unitn.disi.ds1.qtop.TimeOutManager.java`

To manage many ACKs and to detect a coordinator failure the various nodes have to use a series of countdowns and timeouts.

We utilised a map optimised for the use of *enum* values as keys. Every value of the map is a tuples array, every tuple contains a *Cancellable* and a natural number.

The *Cancellable* is a scheduled message that a node sends to itself to mimic a countdown and decrease the natural number. The natural number has as its initial value the number set by the user before starting the simulation, once it reaches zero it is considered to be expired, hence, its respective countdown is cancelled and the node self sends a *TimeOut* message.

The timeouts manager is used for many *ACK*s, these messages invalidate the respective countdowns upon reception. The *HeartBeat*s countdown has a different behaviour, once an *HeartBeat* is received the countdown is reset to its initial value and starts again.

2.3 Voters map

`it.unitn.disi.ds1.qtop.VotersMap.java`

To make a final decision, the coordinator had to keep track of all the votes cast for every update proposed.

This structure is similar to the one used in 2.1. Except this time the matrix holds, for every cell, a tuple containing a map with all the voters and the final decision imposed by the coordinator.

Here too the epoch and the sequence are used as indices. In every cell beside the map, there is the final decision cast by the coordinator or a phoney value in case the node is still waiting for the decision to be cast. As in 2.1 the value of the decision can be either “PENDING”, “WRITEOK” or “ABORT”.

3 Crash system

Crashes can be induced at run-time, through the user interface, at specific points to observe how the system reacts and to ensure that it can recover or continue to operate despite failures.

The types of crashes that can be simulated include both node and coordinator crashes during different operational stages:

- Normal node - Before receiving a new write request from a client
- Normal node - After forwarding a new write request to a coordinator
- Normal node - After receiving a vote request from the coordinator
- Normal node - After casting the vote for a request
- Normal node - Before the acknowledgement of an election message
- Normal node - After an election message is sent or forwarded
- Coordinator node - During the multicast of a vote request
- Coordinator node During the multicast of a decision response

To mimic a real crush, this system picks a client which will send the designated crush request to a random node. Then, if the node does not match the recipient it will forward the message to a compatible node.

> **E.g.**
> If the user inserts a crush of type “normal node”, and the client sends it to the coordinator, then the coordinator will forward it to a random node within the network.

The node (or coordinator) that receives the crash message does not crash. Instead, it enters a virtual crash state where every message received is simply ignored; as by specification, a crashed node does not recover from its crashed state.

The last two crush types are peculiar, they are probabilistic crushes. To create more realistic scenarios, and generate inconsistency among the nodes, the coordinator can crush only during the multicast of some messages. Every message that it sends during a multicast has a 20% chance to make the coordinator crush.

4 Election process

4.1 A node starts the election process

When certain events happen the nodes enter the state "election state". During this state, they handle only a subset of messages, in particular, the *Election* and the *ElectionAKC* messages. The election state can be triggered by a heartbeat or write timeout. Multiple nodes can enter the election state at the same time, in this scenario nothing unpredictable happens because the nodes always check if they are already in an election state via a boolean variable.

4.2 A node receives an election message

If a node enters the election state by a timeout, it means it has not yet received any *Election*. The node crafts a new *Election* message and sends it to the next node. The next node is computed by sequentially scanning the nodes within the group.

If a node receives an *Election* message while it is in a non-election state the flow is more complex:

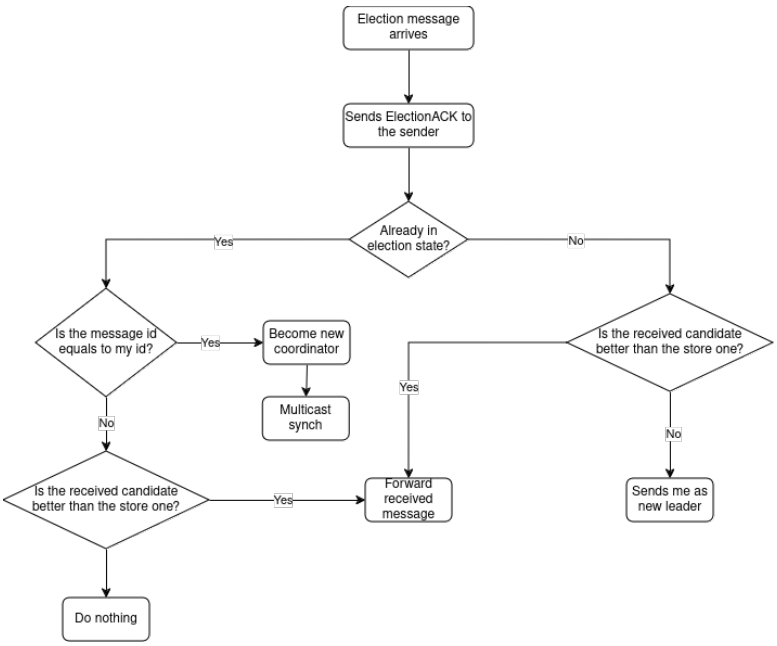

Figure 1: Election flow

In the case, a node does not receive, in time, an *ElectionACK* it re-sends the *Election* message to the following node. More in general a node sends the *Election* message to the node with index:

$$\text{index} = (\text{nodeId} + x) \mod |\text{nodes}|$$

With x the number of tries to send a specific *Election* message.

4.3 Global Timeout

Once a node enters into the election state, it starts a countdown called "global election countdown", Every node has an instance of this countdown and serves as a safeguard in case, during the election process, multiple consecutive nodes crush. Once this countdown terminates, it self-fires a timeout to make every node restart the election process.

In the worst-case scenario in every election round, this timeout is triggered. However, by restarting the election process we ensure an eventual election.

5 Synchronisation

Once the new coordinator is chosen and therefore elected, a new Synchronisation message is multicasted; the message contains the coordinator's pairs history (which is the most updated node in the whole network).

When a synchronisation message is received by a Node, the Node sets the reference to the new coordinator, it overwrites its history with the coordinator's history, and ends its election state. At the end of the synchronisation phase, the network undergoes a fresh restart that renews all the previous actor behaviours such as heartbeats, write requests, etc.