Report 4: groupy

Gabriele Morello

October 2022

1 Introduction

In this laboratory we implemented a Group Membership System, our program will provide an multicast. We will have different application layer processes and our goal is to coordinate them to have the always the same state. When a process wants to change his state he will it communicate to the other nodes so that everyone can execute it. The synchronization is achieved through total order at the multicast level.
A gui module will show the state of a process with a color. My goal is to make sure that the system keeps his synchronization when a node crash or when a new node is added

2 Main problems and solutions

Some modules were provided with the task: worker, gui and test.

Worker implements the application layer, its functionalities are starting and initializing workers, starting but joining an existing worker, listen and react to commands like: changing color, sending a state, handle joining requests, freeze, change the sleep time, stopping and send a multicast message.

Gui spawns some windows, one for each worker, and loops infinitely waiting for changing colors commands.

Test provides some useful functions to test the other modules. From here we can start a single node, add some nodes to an existing worker, we can also specify a number of worker and start them all at the same time with more, and we have commands like go, freeze, sleep and stop to try how the system works.

I operated only in the gms modules (four in total) they are the core of the group membership system and progressively add more features

2.1 gms1

This module is the most basic, it provides functionalities of multicasting but it's not fault tolerant. Every node is either a leader or a slave, nodes receive messages from the the worker module and forward them to the leader, the leader multicasts every message he gets to every node.

2.2 gms2

The new feature of this version of the gms module is to detect when a leader crash and elect a new one in case it happens. To detect a leader every slaves calls the erlang:monitor function during init, in case of crash he will receive a DOWN message and this message will trigger a new election, the election is very simple because every node only takes the first node in its Group list as new leader, if things are correct every node should have the same list so only one new leader will be elected.

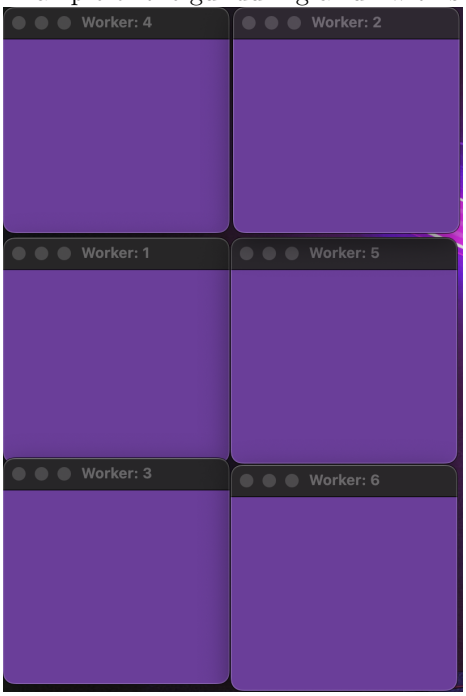
2.3 gms3

This version of the gms module introduce a reliable multicasting, this reliabilty is achieved thanks to a logical clock that allows nodes to keep track of what they have to receive and not act twice on the same message in case a leader crash before sending the message to all nodes, the leader will have a new task to increment the clock before sending a message. Slaves will also keep track of the last received message and broadcast it before taking the leader role in case of election.

2.4 gms4

This module was optional and part of the bonus point question, since Erlang guarantees the order of messages but not their deliverance we want to be sure that every message sent by the leader arrives, to obtain this with I implemented an acknowledgment system where instead of a simple send call during the broadcast iteration. I call a custom function that after sending a message it waits for an acknowledgment, if the acknowledgment does not arrive within an interval of time the leader sends the message again

Figure 1: Example of the gui during a run with six workers



3 Evaluation

During tests I noticed how gms1 doesn't guarantee stability in case of a crash, but it works as it should in normal conditions, gms2 tries to handle stability with an election after a crash but a synchronization is lost because of how multicast is unreliable. gms3 solves this last problem and I couldn't test any cases where I actually needed gms4. gms4 anyway should cover most of the problems, we still could came up with critical cases for example in case of a very slow network our timeout may become too short and we may end up send messages infinitely, we could solve this with a congestion control based on TCP New Reno but this is definitely out of the scope of this laboratory. We also have to consider how acknowledgments impact on performance because not only we will use more CPU time on each node but we will have at least the double amount of messages in our network