HW Report 4: Groupy – Group Membership Service

Ayushman Khazanchi

October 6th, 2021

1 Introduction

The goal of this assignment was to implement a group membership service among multiple nodes where the entire cluster remains in a synchronized state throughout based on atomic multicasting. The cluster is composed of an elected "leader" node and the rest identified as "slave" nodes. The state is identified by a "color" that is selected at random by each node and passed as a signal to the leader node. The leader node then broadcasts this "state" to all the other nodes of the cluster. As part of the assignment we examine different scenarios involved in setting up the cluster as well as those when a leader or message is dropped.

2 Main Problems and Solutions

2.1 Architecture

Figure 1 below describes the architecture of our four-node cluster with the first node being elected as the leader node and the rest of the nodes joining the cluster as slave nodes.

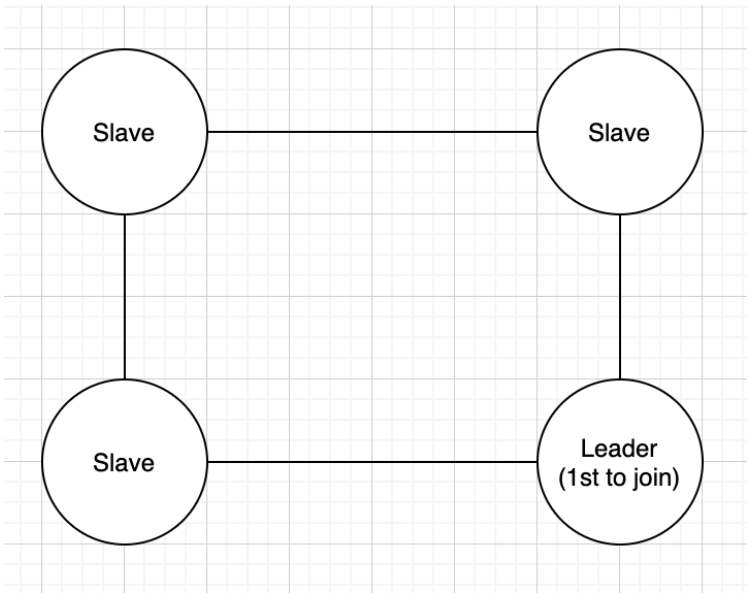

Figure 1

Figure 2 descibes the various layers of the nodes in more detail and how the processes map across the GUI, Application, and Group Layers.

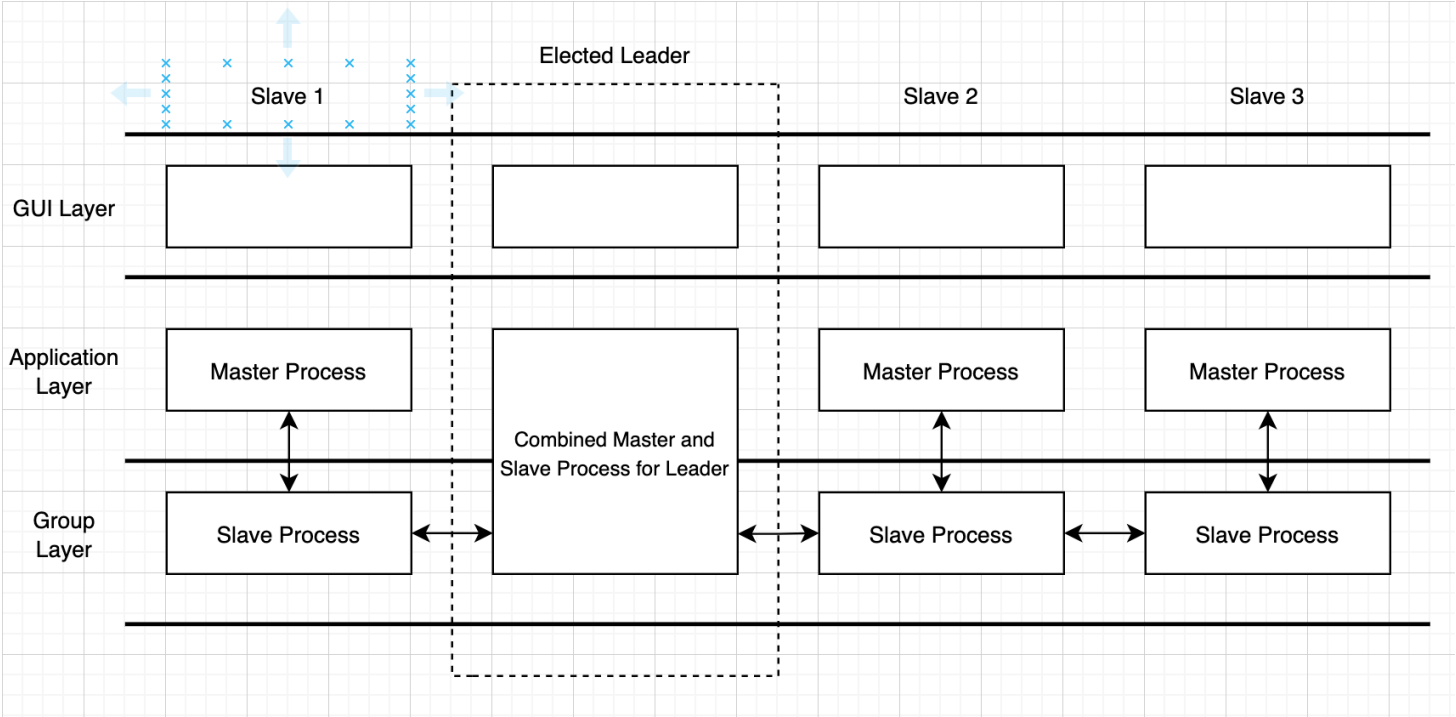

Figure 2

2.2 Implementation

Our initial implementation is fairly basic. We setup a leader and a few slaves and start passing messages between the slaves and the leader. When a message is received by the leader it is broadcast to the cluster and a new view (new color) appears on all the nodes. In the first implementation we only deal with adding nodes so if a leader fails or a node is somehow lost the state of the cluster is lost as well. At this point, there is no re-election.

After the above test is successful, we implement a leader re-election whenever a leader is lost or crashes. To test this, we also introduce a randomized crash function that causes a leader to crash and send a "DOWN" message to the slaves.

```
{'DOWN', _Ref, process, Leader, _Reason} ->
        election(Id, Master, N, Last, Slaves, Group);
```

We know the slaves will receive a "DOWN" message because we have implemented *erlang:monitor(process, Leader)* that monitors the pid of the leader process. When the slaves

lose the leader, they can start a re-election process via the *election* function. However, even though a new leader is elected the cluster goes out of sync as the messages from the crashed leader are not always multicast to all the nodes before crashing.

Now that we have fault tolerance through re-election, we can fix the synchronization problem by assigning each message a number that we increment. Each node also now checks whether the message number they've received is higher than the previous message number. This is shown in the code snippet below. In the case where the new message is of a higher number, the message is broadcast. Otherwise, it is dropped.

```
{msg, I, _} when I < N->
        slave(Id, Master, Leader, N, Last, Slaves, Group);
```

This way we're able to maintain synchronization in the cluster even when a leader goes down and a new leader is elected.

3 Evaluation and Extension

Although our cluster is stable enough to undergo a leader crash and re-election and remain synchronized after that, we're still facing some limitations. There is a possibility that a message may reach the leader but get dropped before it is broadcast. Though we simulate this in our environment, this can still happen in the real-world due to various factors like a network issue for example. One possible way to not lose messages before they're broadcast to all slaves is to implement a message queue on the leader node.

Whenever a node is elected as the leader it can begin to hold a message queue. Since we're already passing the messages in gms3.erl along with a message number we can keep adding the messages in the message queue in the same FIFO order. On the leader we can implement a check to ensure that a message is "safe to broadcast". Every message that is sent out should be of a lower number than the next message in the queue. The message is removed from the queue only when the message is checked as "safe to broadcast" **and** it is broadcast successfully to all nodes.
In the implementation above, if a message is broadcast to slave 1 but is dropped before it reaches slave 2, it will still be present in our leader message queue. Since it is not succesfuly removed from the queue, the same message will be broadcast again and this time slave 2 will

be able to receive it. Now, the message having been broadcast successfully will be removed from the message queue. This, I feel, is one way we can implement a solution to ensuring that messages are not dropped in the middle.

However, there are some limitations with this approach as well. We can hold messages in a queue on the leader but what happens when our leader itself crashes in this process? We have implemented re-election in our code for a new leader, but we don't have anything for saving and transferring our message queue to the new leader. One way to do it is to maintain a leader queue on every node but that is updated only by the leader node so when a leader and a new leader is elected, the queue is already available to it. But this way, on each node we will have to update the queue and perform a "safe to broadcast" check when we do a broadcast. We can see now how this is starting to adversely affect our performance. As a result, this is probably not a feasible real-world solution, but it does provide one way to save our "lost" messages.

4 Conclusions

In this assignment we're able to understand how atomic multicasting works through Erlang message passing. We're able to get an idea of a rudimentary election process and how to bring multiple nodes together into a distributed cluster. I found this part quite interesting. We learn how to keep the cluster active despite introducing a simulated leader crash. While this was a tricky assignment to understand in terms of the code and the flow of events through code I found it quite interesting to learn a small-scale implementation of a real-world scenario. I also found it quite useful to think through possible solutions to the bonus question of trying to save lost messages.