Homework 4 Report: Groupy

Justas Dautaras

September 22, 2019

1 Introduction

In the fourth Distributed Systems task a group membership service is implemented. It uses atomic multicast to update all the nodes at the same time, i.e. they all perform the same sequence of state changes. In other words, all the nodes should receive changes at the same logical time.

This task is divided into couple main sections:

· **Leader** This node tags each message with a sequence number and multicast it to all nodes. The application layer is unaware of whether its group process is acting as a leader or not.

· **Slave** A slave will receive messages from its application layer process and forward them to the leader. It will also receive messages from the leader and forward them to the application layer.

· **Elections** If nodes would not die, this would not be necessary. But because nodes can die unexpectedly - all slaves have the same list of peers and they all elect the first node in the list as the new leader.

Completion of this task allows to test atomic multicasting properties and to understand how it works and where it can be used.

2 Main problems and solutions

One of the main problems in distributed systems that this task explains is total synchronization. Where the application layer not knowing about the inside processes, can be sure that information will be sent throughout all the nodes synchronously. This is done by picking one leader out of all the nodes who is in charge. It then keeps track of it's 'slave' nodes, where communication is done through the leader.

```
{mcast, Msg} ->
      bcast(Id, {msg, Msg}, Slaves),
      Master ! Msg,
      leader(Id, Master, Slaves, Group);
```

Of course, having just this would work, but another important issue is handling errors. And the most important one, is the crashing of the leader node. In this case, the slave nodes throw an election, where, randomly, the first node in the list of active nodes becomes the new leader.

```
election(Id, Master, Slaves, [_|Group]) ->
  Self = self(),
  case Slaves of
    [Self|Rest] ->
      bcast(Id, {view, Slaves, Group}, Rest),
      Master ! {view, Group},
      io:format("New leader: ~p ~n", [Self]),
      leader(Id, Master, Rest, Group);
    [Leader|Rest] ->
      erlang:monitor(process, Leader),
      slave(Id, Master, Leader, Rest, Group)
  end.
```

With these functionalities, we have a functioning distributed system, where all nodes are synchronized, without the fear of crashing when the leader node dies.

3 Evaluation

In this task we have tested three different variations of this system:

· **gms1** This is the simplest solution of all - it does not have error handling of a dying master node.

· **gms2** This solution solves the dying master node problem with an elections function, where a new leader is chosen.

· **gms3** This version is a slight improvement of gms2, where reliability question is solved by tracking the message's number – if the master node dies then at least the first node in the list should should have seen the message (FIFO). Thus, it retranslates the last gotten message and other nodes just discard it in case they've already read it.

Each one of these modules show magnificent differences to what minor changes have in distributed systems.

4 Conclusions

Solving this task has taught me to program on a more intermediate level of functional programming and helped to further extend my knowledge on

synchronization of multiple nodes and the fundamentals of distributed systems.