Groupy: a group membership service

Joakim Öberg

October 1, 2020

1 Introduction

In this assignment a group membership service (gms) was implemented. The goal was to have several application level processes with a coordinated state, i.e. they should all perform the same sequence of state changes. In this lab each process was a small window where the color of it was the state, i.e. a state change was a color change. The goal was to get all processes to be synchronized and keep on going even when some processes dies and other join. The files in this assignment are **gui.erl**, **worker.erl**, **test.erl**, **gms1.erl**, **gms2.erl** and **gms3.erl**.

2 Main problems and solutions

2.1 gms1

In the first implementation gms1 was all code except bcast/3 given from the assignment. My bcast/3 takes a message and broadcasts it to everyone in a list (slaves).

```
% written by me, sends the message to all its slaves(processes)
bcast(Id, Msg, Slaves) ->
    case Slaves of
        [] -> ok;
        [P1|Rest] ->
            P1 ! Msg,
            bcast(Id, Msg, Rest)
end.
```

2.2 gms2, handling faliure

In the second implementation gms2.erl did we add three functions crash/1, bcast/3 and election/4. The function bcast was here given and I exchanged it agains the one I wrote in gms1. The function crash/1 takes an Id and if a random value matches a defined value arghh the process for the executing Id

is closed. The code for crash/1 was given. The code for a function election/4 was also given.

For detecting failures we also added a call to **erlang:monitor(process,Leader)** in the initialization of the save. This means that if the leader of a slave crashes, the monitor function will send a **'DOWN'** message and then we know that we need to do an election of a new leader i.e. take the next node in line of the slaves as new leader. We also added a timeout that is used if a new node wants to join and the leader at that time dies, then the message will just be hanging. So if the timer goes out, i.e. we got no response to join a group then an error message: *error "no reply from leader"* is sent out.

2.3 gms3, reliable multi-cast

For gms3 we added ways to know if a received message is old or new and a copy of the last seen message. This was somewhat tricky because there was many places where this should be added, both in the arguments a function should take but also in function calls and when sending messages. One tricky thing was also when we should count up for the number of messages, both a leader and a slave needs to count up when recursively calling itself that it has received one more message. After my first try I missed to count up and no one could join because any new message was seen as old, after the fix the code worked.

3 Evaluation

3.1 worker

The worker is the lowest layer in this application. The worker has no idea about leaders and slaves. The worker handles if a node should freeze, wait, update its state, sleep etc. All code was given from the instruction.

3.2 gui

Gui is the code that delivers the graphical pop-up window with the color changes that we can see. All code was given from the instructions.

3.3 test

Test is a small program that can be used for testing the different modules that was implemented gms1, gms2 and gms3. Test has both a function first/3 that let us start the first leader and add/4 that let us add more nodes to this leader. Or we could use test:more/3 that takes the argument (N, Module, Sleep), where N is how many workers we want to start in one

go, Module is which module we want to test and Sleep i.e. for how long we should sleep between proposing state changes.

3.4 gms1

The first implementation of the group membership, it had no function that selected a new leader if the leader node crashed. To crash a specific node I closed the window for the process.

As described above the function **more** in test, is always assigning the first leader to have id = 1. I did a test by starting 5 nodes **test:more(5,gms1,1000)**, first a node with the highest number(5) was first closed, then node 3. Nothing happened because both node 5 and 3 are slaves under node 1. When node 1:s window was closed, the leader of the two slaves left was gone and no one was longer responsible to broadcast messages to everyone. Because there is no function in **gms1** that elects a new leader after the old one dies the two slaves freezes in the last known state.

3.5 gms2

In gms2 the nodes can select a new leader when an old leader crashes. Why a node(leader) crash can we see in how the bcast and crash is made.

```
bcast(Id, Msg, Nodes) ->
    lists:foreach(fun(Node) -> Node ! Msg, crash(Id) end, Nodes).

crash(Id) ->
    case random:uniform(?arghh) of
        ?arghh ->
            io:format("leader ~w: crash~n", [Id]),
            exit(no_luck);
        _ ->
            ok
    end.
```

In bcast the first node in the list Nodes is sent a message, then we call the function crash. If some random value matches our defined variable arghh then the process of the leader(which is executing this) is shut down with exit/1. This results in that the slaves receives a 'DOWN' message and elects the new leader. Because all slaves has the same copy of slaves, everyone will choose the same one as new leader.

A test was done by starting one node and then adding four slaves with peer set to the leader, and then adding two more but with peer set to one of the other slaves. Then the program was left to run until a node crashed.

When it happened five nodes was still in sync but one of them was out of sync. No more nodes become out of sync than this one during the whole test. This node that became out of sync was probably given an old message at the same time that the leader died.

3.6 gms3

To solve the problem with nodes out of sync we introduced a counter for messages and a place holder for old messages. The counter let us know that any messages with value less than the counter is old. If a leader dies, then because how broadcast(bcast) is built we know that at least the first node in list (the new leader) will have gotten the message, and the other maybe not. Then at least the new leader has a copy of the last sent message. I we look into the election function:

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

Then we see that the node that is to be chosen as new leader will directly broadcast the last message i.e. the one sent from the last leader. This will do that the nodes stay in sync as long as a sent message never gets lost.

A test as the one in gms2 was set up, and this time no node came out of sync when a leader died.

3.7 What we can improve?

The implementations above are relying on that a sent message will arrive to a node that is alive. That does not need to be true in the real world as network traffic can be lost. One way to solve this is to add some sort of acknowledgements to messages.

I'm thinking that where we actually want to have this extra feature is for the messages sent in the **bcast** function in **gms**. Then we have to think how to best implement it, one solution could be that for each node that we want to broadcast a message wait some time for an ACK to return, if it

receives an ACK then we send to the next node. If no ACK returns, should we consider it as dead or should we resend the message?

Another idea is that we broadcast the message to everyone directly but then we need a function that remember which nodes that we have and can check them off from a list when the ACK returns. The same problem with what we do if no ACK returns, then we also need to save a copy of the last sent message.

4 Conclusions

This assignment felt that it was closer to real life than the other ones that we have had because you cant be sure that all nodes stay alive or if new one comes. Code wise this assignment was not that hard as almost all code was given, instead one had to really understand the concept and what the code did to understand what happened. It was fun to have an assignment where the program was more or less automated and one just had to observe what happened and why it happened. I learned a lot about synchronization, gms3 and message counting feels really close to logical time where you also want nodes to stay in sync and not care about old ones.