Groupy – a group membership service

Óttar Guðmundsson
28 September, 2017

1 Introduction

This week's assignment was to test out a group membership service with several different methods to see how nodes that crash before or after sending a message can still be sent properly to all nodes with atomic multicasting. By providing an application layer that keeps the nodes synchronized and in a total order, a process can send a message about something or let them know it has changed its state to it so the layer will notify the rest of the nodes.

The assignment provides a lot of skeletal code to test out and lean how a synchronized state can be achieved in a distributed system.

2 Main problems and solutions

Gsm1

For the first implementation of our membership service which was taken from the pdf, the first node that would be created was called the leader (master). After initializing it, other nodes could be created and added to the group at the same time would be called slaves. With a small *test* file provided, we were instructed on how to run the workers. I had problems running the file for the first time but figured out later that I had written *mcst* in a receiving message tuple rather than *mcast*. Unfortunately the application layer had a *try/catch* function to grab errors so it took my quite a while to figure out the errors. After adding an Error command and printing it out in the *receive* function of the *gsm1* I finally found my error.

Gsm2

The next step was to handle failures of nodes, that is, if the leader node died the whole system wouldn't stop (what would happen if the leader crashed). The first solution to this problem was to let all nodes use the *erlang:monitor* inbuilt function to monitor the leader. If some node found out it has died, it would initiate an election. The election method was pretty primitive, if a node would notice that it was next in line after the crashed leader in the process list it would become the leader and broadcast it to the other nodes, otherwise it would stay as a slave and monitor the first node in the list. To see if this was working as intended, a small random crash was implemented each time the leader sent a message to a node.

The interesting part was that after the timeout (1/1337 to begin with) a new leader was elected but some nodes got out of sync. This happened when the leader had already sent a message to few nodes while others were still to receive theirs e.g. if I was running 5 and the leader would crash after sending the message to the first two, after the election the latter two were out of sync.

Gsm3

To solve this, each message sent would include a counter to know the "timestamp" of the message, which was simply a counter that represented the expected value of the next message. Also, each node would keep track of the last message it had sent or received. So everyone would receive the same message and keep in sync, the newly elected leader would then broadcast the last message it had sent to make sure everyone would receive it and following it up with the new view sent. To get rid of duplicates, the code would simply check the timestamp of the message and discard the ones that were

older than the expected timestamp of the node. After changing the code with these features the nodes stayed in sync.

Gsm4 and Gsm6 (Optional part / extra point)

It was worth noting that the implementation did not really work, since we are relaying on an external monitor provided by Erlang. We do not have any guarantees that the messages sent will actually arrive but we know at least that they are delivered in FIFO order. After a little bit of counseling with the professor he said that implementing a custom monitor would suffice to get the extra point but I was unsure how to do that. According to the book, we could both implement async and sync monitor by using timeout and *suspect* nodes but this sounded like too ambitious addon for this project. There also was an option to use something called negative acknowledgement for the reliable message passing but that would affect performance heavily, since the number of messages sent would be doubled. Instead, I tried two different methods.

Gsm4

The first one was to implement it through this reliable algorithm. It was pretty simple, every node kept a history of messages. When it received a new one, it added it to its history and broadcasted the same message to all other nodes in the network.

On initialization
  Received := {};

For process p to R-multicast message m to group g
  B-multicast(*g*, *m*); // $p \in g$ is included as a destination

On B-deliver(*m*) *at process q with g = group*(*m*)
  if ($m \notin$ *Received*)
  then
    Received := *Received* $\cup$ {*m*};
    if ($q \neq p$) *then B-multicast*(*g*, *m*); *end if*
    R-deliver m;
  end if

Image 2: The algorithm

This surely worked but really did decrease the performance of the network. But as the professor said, there is always a tradeoff in qualities that you are looking for. I didn't implement the q=/=p part though, since we already had the timestamp and I did not want to modify the message layer.

```
{msg, N, Msg} ->
  Master ! Msg,
  ReceivedAdded = manageReceived(Received, Leader, N, Msg, Id),
  slave(Id, Master, Leader, N+1, {msg, N, Msg}, Slaves, Group, ReceivedAdded);
```

```
manageReceived(Received, Leader, N, Msg, SenderId) ->
  case lists:keyfind(N,1,Received) of
    {FoundN, FoundMsg } ->
        Received;
    false ->
      UpdatedMessages = [{N,Msg} | Received],
      Leader ! {mcast, Msg, N},
      UpdatedMessages
  end.
```

Image 2 and 3: We update the list in every message we receive, and look for it in our history. If it isn't found, ask leader to broadcast it again. I could look for the timestamp or the message itself.

Gsm5

Note that gsm5 was a failure and did not survive the final publish.

Gsm6

Instead of keeping history of all messages for every node, I simply let the leader keep the history of the last 10 messages so I wouldn't overflow the history and loose performance. In the slave function I added a new type of receive pattern matching. That every time we received a message with a timestamp that was higher than the one we expected, we calculated the missing amount of messages and sent a message to the master. It would then loop through its current history and send the messages back to the node.

```
% WHEN WE HAVE LOST A MESSAGE
{msg, I, _} when I > N ->
  io:format("happens ~w > ~w~n",[I, N]),
  Leader ! {lostmessage, I-N, self()},
  slave(Id, Master, Leader, N, Last, Slaves, Group);
```

```
manageHistory(Message, List) ->
  UpdatedList = [Message | List],
  if length(UpdatedList) > 10 ->
      lists:droplast(UpdatedList);
    true ->
      UpdatedList
  end.
```

```
    {lostmessage, N, SenderId} ->
      sendLostMessage(N,History, SenderId);
    stop ->
      ok
    end.

sendLostMessage(0,_List, _Sender) ->
   ok;
sendLostMessage(N,List, SenderId) when N > 0 ->
  [Young | Old] = List,
  [SenderId ! Young | sendLostMessage(N-1,Old, SenderId)].
```

Image 4, 5 and 6: The receiving pattern match of the slave, the history management and how the master handles it.

Doing it this way, I was bypassing the message layer but this can be done properly with better architecture. This method also excluded the possibility of providing the next leader with the history of the crashed leader. I didn't think about implementing this during my coding session but this could be done with an external server where the last history is always saved or something similar.

3 Evaluation

There was no evaluation to take based on the results. Here is a picture of the console printing the leader multicasting and crashing. After that the next process is elected and broadcasts the last message, then sending the view again with the timestamps intact and in correct order. Neat.

```
BCAST 1{msg,319,{change,6}}

leader 1:crash
BCAST 2{msg,319,{change,6}}

BCAST 2{view,320,[<0.73.0>,<0.74.0>,<0.75.0>,<0.76.0>],[<0.68.0>,<0.69.0>,<0.70.
0>,<0.71.0>]}

BCAST 2{msg,321,{change,1}}
```

Image 7: Console printing of the improved election.

4 Conclusions

I thought the techniques used in this assignment were interesting as they demonstrated how complex synchronization between nodes can be. The assignment did not include a lot of coding problems to figure out, rather to understand the problem and how it was tackled to prevent getting the nodes out of sync.

The book did mention that synchronizing closed groups was of course way easier than managing overlapping groups, but that might be figured out later.