Report 4: Groupy - A group membership service

Emil Ståhl

October 1, 2019

1 Introduction

This report covers the implementation of a group membership service (GMS) which is a form of coordination that is needed by distributed systems in order to handle concurrent events, crashing of nodes and joining of new nodes. This concept was demonstrated by implementing a GMS with atomic multicast which should be able to handle multiple application layer processes with a coordinated state. The state of each node is visualized by an arbitrary color. When one node wishes to change its state it then must multicast it to all other nodes in the group. When all the nodes in the group are synchronized they will be able to display the same color. The main topic covered in this work is to get a deeper understanding of the problems that coordination results in and how they can be solved.

2 Main problems and solutions

2.1 Initial modules

gms1.erl

The GMS acts as a group layer that provides transparent multicast for an above application layer.

worker.erl

The application layer that handles all of the sending of state updates, joining of workers and deliveries of views.

test.erl

Test includes a number of functions that start a bunch of workers and the ability to send them messages such as freeze, go and sleep.

2.2 Development phases

To implement the desired improvements to the primitive gsm1 module the work was split up into four phases that are described below:

- **gms1.erl** was tested and analysed together with the two other initial modules.
- **gms2.erl** was implemented with the added functionality of detecting the crash of a leader by forcing each slave to monitor the leader and after a detected crash going into a leader election state to elect a new leader node.
- **gms3.erl** the basic multicaster was replaced with a reliable multicaster. A process that forwards all messages before delivering them to the higher layer. All slaves also keeps a copy of the last message received from the leader.
- **gms4.erl** assures that messages arrive by implementing reliable delivery with acknowledgements and resending of messages after a specific timeout.

2.2.1 gms1

The first implementation of the group membership lacks the support for fault-tolerance. A group consists of several nodes that is either a leader or a slave. Slaves receives messages from the application layer and forwards them to the leader who in turn multicasts the message to the rest of the group. The nodes will then receive the message and update their color accordingly.

2.2.2 gms2 - Failure handling

As stated in the section of 2.2 this phase covered the implementation of crash detection and leader election. To implement the monitoring the built in Erlang function of erlang:monitor/2 was added on all of the slaves.

```
receive
    {view, [Leader|Slaves], Group} ->
        erlang:monitor(process, Leader),
```

If a slave detects the crash of the monitored leader it will receive the message with the folowing format: `{'DOWN', _Ref, process, Leader, _Reason}`

After this message is received by the slave, it immediately goes into the election phase. The election function gives the leadership to the first in a list with the slaves PIDs in the group layer, all group members have the same list, and the rest starts to monitor the new leader.

A join request to a group timeout was also added.

2.2.3 gms3 - Reliable multicast

To solve the problem of a leader who dies before multicasting a message to the group, reliable multicasting was implemented. This is done by using a simple logical clock with a sequence number on all broadcasts from the leader. The leader is responsible for keeping track of the number and increases it before every broadcast while the slaves keep track of what sequence that should be received next and also stores the last received message from the leader. This is needed to cover the cases where a previous leader crashed and did not manage to send the message to all nodes in the group. However, this can cause duplicate messages to be delivered but is easily taken care of with the help of sequence numbers.

2.3 Bonus task - gms4

In Erlang the only guarantee is that messages are delivered in FIFO order and no guarantee that they actually do arrive. This can be solved with an ACK sent from receiver (slave) to broadcaster (leader) and if the message has not arrived before a timeout it will be sent again by the leader.

The ack is implemented as follows;

```
Leader ! {ack, Id},

receive
     {ack, Id} ->
     io:format("Node ~w received message ~w with Id = ~w\n", [Node,
     Msg,Id])

after 100 ->
     resend(Node, Msg)
end.
```

3 Evaluation

The evaluation of this work is that with gms1 all the nodes changes their state accordingly but in the case of a leader crash it won't be able to handle this. In gms2 failure handling was introduced and after a slave detects a crash of the leader it will go into an election state to elect a new leader. However, the gms2 will not be able to continue after a crash. Therefore, a reliable multicast was implemented in gms3. The downside with gms3 is that there is no reliable delivery. This is fixed in gms4 with acknowledgements. Gms4 shows printouts of messages that has been lost but the nodes are still synchronized because the leader resends the messages. The nodes would otherwise become out of synchronization if there was no acknowledgement message. The disadvantage with using acknowledgement messages is the effect it has on performance since the number of messages now has increased. If there is a congested network the monitor might think that a leader has died when in reality it is still alive, which results in that a new leader is elected and then it will be one extra leader.

4 Conclusions

This assignment was not that hard to get up and working with gms1-2 as almost all the code was given in the manual. However, the gms3 and gms4 required some thinking but not as much as the two previous assignements. The main topics learned was understanding the importance of node synchronization and the things that might happen during crashes. As well as learning how Erlang delivers messages in FIFO order and that they are not always delivered correctly.