Groupy - a group membership service

Abdullah
aabdull@kth.se

October 13, 2021

1 Introduction

The seminar consisted of implementing a group membership service that provides atomic multicast so that we can have several application layer processes with a coordinate state. When the process state is changed it first multicasts it to the rest of the group about its state change. The multicast layer provides total order which means all nodes will be synchronized. However reliable and ordering of messages is a major challenge in implementing a group membership service.

The objective of this assignment was to know about the importance of global state in distributed systems and how to achieve a shared global state in distributed systems.

2 Groupy Architecture and Implementation (Problems and Solutions)

2.1 Architecture

The assignment follows the pattern of Leader and Slave for the group membership service. The architecture for groupy is following.

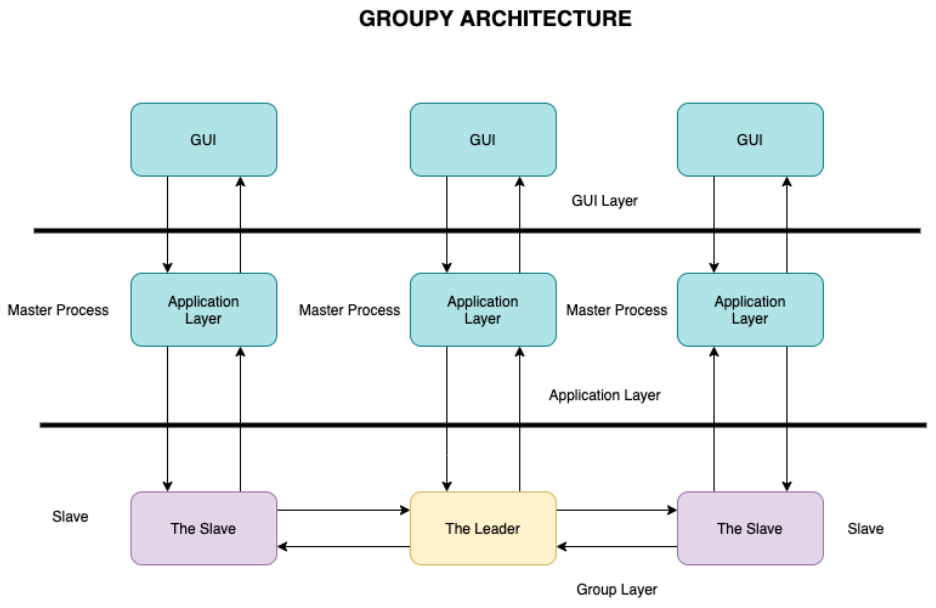

Figure 1: Groupy Architecture

2.2 Initial Implementation - gms1

The code for initial implementation is called gms1. In this implementation the first process or node in a group becomes the Leader of the group and the remaining processes becomes slaves. Leader node is responsible for multicasting the message to all other nodes and also accept join requests from nodes to join the group.

The problem with the initial code is that it had no failure detectors like if the leader node crashes, there is no code to make another node leader to handle the message multicasting to all other slaves. The solution to handle failure detectors is given in gms2 implementation.

2.3 Failure Handling - gms2

To introduce failure detector in gms2, erlang monitor is used for slaves to monitor leader process. Whenever a leader dies the slaves move to an election state where next leader is choosen. The next leader is the first slave in list. If a node finds itself at the first position in the list it will become a leader, otherwise it will continue being a slave.

A random crash is added in system which works for one request in every 100 requests. The leader crashes and a new leader is choosen from the slaves list.

After doing some test I found the problem with the implementation is that some nodes maybe dont receive messages because the leader dies just before broadcasting the message, so the worker processes are out of sync and we can see different colors appearing for some worker processes.

To remedy this problem reliable multicast in gms3 is implemented.

2.4 Reliable Multicast - gms3

For this part it is assumed that message delivery is reliable and that if two messages are sent after each other and the second one is delivered, then the first would also have been delivered. To implement this multicaster reliable sequence number is tagged with each message and also the last message received is stored so that it can be resent by the new leader choosen through election when the leader crashes. However this leads to duplication of message so to avoid or discard duplicate messages both the leader and slaves save a counter with the value expected of the next message. If a message number is lower than the saved number, the message is a duplicate and gets discarded.

The methods were given for gms3 but we had to manage the parameters for the nodes and messages received.

After doing some tests, it was seen that now the nodes were synchronized and crashing of leader had no problem with synchronization of nodes and output was same.

3 Optional - Handling Lost Messages

As it was assumed that our message delivery is multicast reliable but erlang only guarantess that messages are delivered in FIFO order if they are actually delivered but as said that the delivery was assumed to be guaranteed but that is not always the case. Messages could be lost and our implementation lacked to handle the lost messages.

A simple simulation for message loss was done where one message in 200 requests was simulated to be lossed and we could see that lost messages were never handled by the system.

```
lossMessage() ->
  case rand:uniform(200) of
    200 ->
      io:format("---message loss---~n"),
      true;
    _ ->
    false
  end.
```

3.1 Solution

To handle the lost messages a messages queue was added which adds each message and view from leader process to queue and we checked the next seqeuce number with current sequence number and if its greater then it means we have missed messages from queue equal to the difference of current sequence number - next sequence number. So to deliver the messagess I implemented a callQueue method which delivers the missed messages first and after that delivers current message.

```
{msg, Num, Msg, Queue} when Num > (N + 1) ->
    callQueue(Master, Msg, Queue, N + 1, Num - (N + 1)),
    slave(Id, Master, Leader, N, {msg, Num, Msg, []}, Slaves, Group);
```

```
callQueue(Node,Message,List,Start,Length) ->
  Msgs = lists:sublist(List,Start,Length),
  lists:foreach(fun(Msg) -> Node ! Msg end, Msgs),
  Node ! Message.
```

The missed messages were now handled in correct manner. However, The drawback to this approch was the missing message will be lost forever if the Leader crashes during the process because we are storing the queue only at leader process and not at slave process.

4 Conclusion

The assignment was related to the theory lecture Global State and the main aim was to know about the importance of global state in distributed systems and how it is managed through different algorithms. I also got to know about multicasting in group communication. I got a chance to visualize the nodes and how they got unsynchronized and then adding some solution to actually cope this problem was a good practice.