Report 4: Groupy: a group membership service

Nils Blomgren
October 4, 2023

1 Introduction

For this assignment the goal was to implement a group membership service where atomic multicast is implemented to sync processors with each other. The first leader will be chosen manually, but then if that specific leader crashes then the first slave in the list will be chosen as a new leader. The leader is the one who decides when a new node should join the group. All processor's are different application layers that will be connected to each other using a group process. Either the application layer or the slaves can request a new view. This will then be multicasted by the leader to all other nodes. This assignment has three different types of modules, where in the first one gms1 nodes can only be added. In gms2 a new leader can be selected but the group will not share the same state and therefore not be synced. In gms3 a new leader will be selected and then they will all share the same state (color). This is solved because the last message sent by the previous leader before terminating is being re-sent to all of the nodes.

2 Main problems and solution

Whenever a node wants to change the state they will first broadcast the change to all other nodes. This is done using a total order multicast, which will guarantee the fact that the nodes will have synchronized states because they all will receive the states in the same order.

The erlang:monitor keeps track of the nodes, whenever the leader is dead the others will be notified. To be able to decide on a new leader the group needs to share a list of nodes that are ordered. The first node in the list should be the leader, and whenever a node detects that they are first in the list they will be the new leader.

Whenever a node wants to connect and has been accepted by the leader the slave will receive a peer process in the group and connect to the group.

To be able to get gms3 to work we had to implement a module where the last message is always sent with any new message. This is because in gms2 the risk is that the leader can terminate before sending a message. The risk is then that only parts of the group actually receive the last message. This is being solved by also including the last message in any view. Therefore the next leader can then resend the last message and all nodes are then able to switch to the same state and continue. But because the previous leader's last message could have been received by one of the nodes already there is a risk of duplicate messages, therefore we also include a counter for the messages that iterates. So if a slave receives an old message by a new leader it will ignore it, since it has already updated its state to that one.

This assignment took a while to grasp. I had quite a hard time figuring out how to solve gms3. For a long time my nodes weren't receiving the last message sent by the leader before terminating. This was caused by me not incrementing my message counter in the correct places as well as not including N in one of the views in the slave function.

In short:
Application layer:

- Worker communicates with the group membership service to either join the group, send multicast messages, and receive messages from other group members.

Group layer:

- Synchronize the views.
  Leader:
    - Responsible for atomic multicasting and managing the group.
    - Forward messages to all slaves.
    - An election procedure selects the leader.

  Slaves:
    - Slaves forward messages between the application layer and the leader.
    - Slaves detect leader failures and participate in leader elections.

3 Evaluating

A test-program was given to us which included functionality to set up a leader and add nodes to the group. I used the test:more(N, module, sleep) function to test my implementation.

In the first iteration the test program was able to run and have slaves running, until the leader died they are all synchronized.

In the second iteration the test program was able to run synchronized and whenever a leader dies a new one is selected, but the other nodes are not aware of the new leader and stops in the current state. The leader on the other hand has changed its own state but does not broadcast correctly to the other’s stats.

In the third iteration the test program can correctly pick a new leader and resend the last message, in other words, the program can change leader and continue to work sending messages to each other whilst having a synchronized state.

4 Conclusion

Because we are not using any acknowledgement, sent messages can be lost because the leader can terminate. This could be solved with returning an acknowledgment so that the sender will be informed if the message was received.

This assignment helped me understand how to implement a group process which has a synchronized state. This was done using atomic multicast and a correct way of electing a new leader and making sure everyone is in the same state before the new leader starts broadcasting new states to the slaves. I would argue that we aren't really implementing an atomic multicast, because one node leader can multicast without having all the slaves receiving the message from them. But at the same time the actual message will arrive to all slaves, from the next leader. Therefore I’m not sure if it fully meets the requirements for an atomic multicast. This was a good lesson in how to coordinate nodes in a distributed system.