Report 4: Groupy

Alexandru Matcov

October 2, 2024

1 Introduction

This seminar covers the implementation of a group membership service consisting of several group members that provides atomic multicast in view synchrony. Each member has a state which is represented by a color. Randomly, a node that wishes to perform a state change must multicast the change though a message to all the members of the group. The states of the group members are synchronized through the multicast's total order property. Accordingly, all the synchronized nodes represent the same color as their state.

2 Main problems and solutions

In order to provide synchronisation between group members the following architecture is proposed for a group of N nodes:

- 1 leader;
- N-1 slaves;

The leader is always assigned to the oldest node in the group. This is done based on the *Id* of the node and on the *process id (Pid)* (in case of nodes with the same *Id*). All nodes that want to multicast a message to the group will send a message to the leader where the following will do a basic multicast to all the group members. If the leader dies the next oldest (lowest *Id* and *Pid*) will be assigned as the new leader of the group. To note here that if a manual lower *Id* than the highest *Id* in the group is assigned to a newly added node, then all the nodes with lower *Pids* will become a leader before the newly added node with lower *Id*, but higher *Pid*.

2.1 Fault-intolerant Implementation

The first implementation is rather basic and simple. It supports the initialization of a leader and multiple slaves which communicate through the designated leader. Once a slave receive a message from the *Master Process* of the *Application Layer* it transmits the state to the leader which in its turn multicasts the message to all the member of the group, including itself. Therefore, considering no lost messages all the group members update their state. However, if the leader goes down all the slaves stop as their state change messages can't be delivered anymore.

2.2 Leader Re-election Implementation

In the second implementation the problem of the leader crash is solved through 2 things:

- monitoring the leader node;
- electing a new leader once the current leader crashed.

Monitoring the leader node is done through *erlang:monitor/2*. Once a slave node detected that the leader node died it get's immendiately in the election state. The implementation of the *election/4* function the process re-elects a new leader based on the first node in the peer list (oldest node). This way the slaves continue to have their state change messages multicasted, but there is a new problem - the nodes become unsynchronized. It is often the case that when a leader crashes it might have multicasted the state change messages to only a part of the group members. This way the nodes that didn't receive the message from the leader become out of sync.

2.3 Reliable Multicast Implementation

This implementation deals with the synchronization problem after leader re-election. The solution introduces with each message a sequence number of the next expected message and for each group member (except the first leader) a copy of the last message seen. With the re-election, the new leader makes sure that all the nodes have seen the last message multicasted by the last leader by multicasting the message one more time. The nodes that have seen the message previously discard the copy by comparing the sequence numbers of the last received message with the new (copy) one. Thus, the above described implementation extends the solution of reliably and synchronously multicasting to the group nodes after leader re-election

3 Evaluation & Extra Bonus

The specifications implemented so far only guarantee that the messages are delivered in FIFO order, but now acknowledgement that they arrive. A solution to solve the issue of lost messages is implementing a message queue.

When a node is elected as the leader, it will manage a message queue. Messages, already assigned sequence numbers in the **gms3.erl** module, will be added to the queue in a FIFO order. The leader will ensure that each message is "safe to multicast" by checking that its number is lower than the next message in the queue. A message will only be removed from the queue once it has been successfully multicasted to all nodes.

If a message is sent to one slave node but fails to reach another, it remains in the queue and will be multicasted until all nodes have received it. Once this happens, the message will be removed from the queue, ensuring that no messages are lost during the multicast.

However, there is a limitation: if the leader node crashes, the queue will not be transferred to the new leader. While the system handles leader re-election, there is no mechanism to move the message queue to the new leader. The solution is to replicate the queue on all nodes so the new leader has access to it immediately. However, this would require every node to update the queue and check if messages are safe to multicast, which could negatively affect performance. Therefore, while this approach may prevent message loss, it would come with a performance trade-off.

4 Conclusions

In conclusion, this assignment was great to understand how atomic multicasting works in Erlang in a reliable way, delivering the messages to the members of the group membership service.