Report 4: Groupy: a group membership service

September 26, 2020

1 Introduction

The main goal of the Groupy assignment is to implement a group membership service that provides atomic multicast.The aim is to have several application layer processes with a coordinate state. The service is architect in the way that a node what wishes to perform a state change must first multicast the change to the group so all nodes can execute it. The tricky part of the assignment is that all nodes need to be synchronized even though new nodes may appear or crash.

2 Main problems and solutions

There several scenarious to be implemented to solve the main problem of the assignment: All nodes need to be synchronized even though nodes may come and go.

1)All nodes wishing to multicast a message do that by sending it to a leader only. Leader multicast the message to all group members nodes.

2) If a leader dies, a new leader is selected. All the slaves have the same list of peers and they all follow the same rule for election: The first node in the least will become new leader. The new leader will resend the last received message and the slaves will monitor the new leader.

3) A new node wishes to join the group can contact any other node which sends the request to the group leader. Leader decides when the node to be included and sends a new view to the group.

3 Evaluation

1) The first implementation version handles adding new nodes without dealing with failures and contains to scenarios: adding the first node (Leader node), and adding other nodes (Slaves). This version makes it possible to create a group of synchronised nodes. Yet if the leader (first node created) dies (closed manually by user) all other nodes frees since there is no new leader elected.

2) The second implementation builds fault tolerance. by detecting crashes, electing a new leader, making sure that the layer preserves the properties of the atomic multicast.

The crashes are detected now by monitoring the leader node by using Erlang build in support to detect and report (sends a message) that the process has crashed.

```
erlang:monitor(process, Leader)
```

Simple yet important design decision is to have a node new nodes added at the end of the peers list. During the election processes, the first node in the peers list will be selected as a leader.

```
Slaves2 = lists:append(Slaves, [Peer])
```

This version also introduces the timeout to handle the scenario where the new node send a request to join the group and the Leader crashes before replying.

```
after ?timeout ->
    Master ! {error, "no reply from leader"}
```

To test that our services The random crash is introduced by defining a constant "arghh" = 100, the defines that a process will crash in average once in a hundred attempts. It shows that the nodes become out of sync, which means the the list of the messages received by the Nodes are not the same. That is caused during the death of the Leader, when some of the nodes have received the last message from Leader, and some not.

3) The third version implements a reliable multicast, where process forwards all messages before delivering them to higher level. The issue async issue is resolved by keeping a copy of the last message from the leader. If the Leader dies, the message is resend by a new leader, and since the new leader is the firs node in the list, it has received the last message from the old leader (as leader sends message to peers in the order they appear in the list). To exclude the possibility of doublet messages, sequence number N is introduced and increased every time a new message is proceeded N+1. Thats helps the processes to identify the dubblets.

```
{msg, I, _} when I < N ->
    slave(Id, Master, Leader, N, Last, Slaves, Group)
```

4 Optional task: What could possibly go wrong

```
How would we have to change the implementation to handle
the possibly lost messages? How would this impact performance?
```

The solution for the handling of message lost could be to implement request/reply scenario where Leader stores message and waits for a reply from each Node upon receiving it and takes away the message from the list. The performance for this solution will decrease, since the amount of messages required for handling it will grow. And the extra storage of messages will be a disadvantage.

Other solution,could be to compare messages sequence numbers to see if any are missing so that the node can send a request to leader to resend a specific message. The lost message would be detected yet it will still require some kind of storage to be able to resend those. The performance will decrease, since the amount of messages required for handling both solutions will grow.