Report : Groupy - a group membership service

Vasileios Giannokostas , vasgia@kth.se

October 8, 2014

1 Introduction

*T*he aim in this seminar was the construction of a group membership service that provides multicast. All the nodes in this group have the same state , they update their states and also they must be synchronized.

After the seminar wil be able to :

- Understand the principles of the coordination in a distributed system.
- Explain the architecture that consists of a leader and slaves.
- Comprehend how we can detect the crashes.
- Understand the procedure of the election in case that the leader crashes.
- Realize the importance of reliability in multicasting.

We will implement a group membership with a leader and several slaves. Slaves and leader send messages to each other. A node that desires to change state must multicast the message to the leader and the leader will multicast the same message to the group , to inform all of them and to execute it.

2 Main problems and solutions

2.1 The first approach

Our first version creates several nodes and enter them to the group. The first node is by default the leader and the other nodes are the slaves. This version doesn't handle failures , for example if the leader crashes the remained slaves don't receive messages from the "died" leader so the slaves don't change state because they have not leader to inform them. This happens because there is no transition between being a slave and becoming a leader.

We don't care if a slave crashes (if the leader is "alive") , the program has a leader so it is up and running.

We introduced the given code for the gui module which creates frames (windows) for every node of our system. I introduced the command below to refresh the windows with the new state(color) , so we are able to notice if the first version of groupy works.

wxWindow:refresh(Window)

How we can detect crashes and make our system fault tolerance?
We will solve this problem by implementing the second version , gms2.

2.2 Improving our system

We improved the first version to solve some issues. In order to detect which node is alive or died , the nodes should monitor the other nodes of the group. We used the Erlang built to detect and report crashes of our nodes. As I inferred previously we care only for the leader , the leader is the major part of our system. This means that every slave must only monitor the leader and detect if it's died. If a slave detects that its leader is crashed then it will move to the election state.

This is the improvement from the first version. The remained nodes elect a new leader and our system has a head to coordinates all the nodes.The election is based on a very simple technique ,the next leader will be the first in the list of remained nodes.

What happens if some nodes don't receive the last message of a "died" leader?

2.3 The third version

In the third version we will create a more reliable multicast to avoid this situation. There is the case that a leader(L) must send messages to the slaves (S1,S2,S3). If the leader dies during the multicasting some nodes are updated with the new state when others are not. According to our implementation the first node of a group(S1) receives new messages from the leader before S2 and S3 receive the same messages. Also according to our election approach S1 will be the next leader(NL) , so NL is responsible to inform the other slaves about the last state.

If S2 had received the last message from L , now S2 will receive also the same message from NL. How can we solve it?
In order to avoid duplicate messages, we introduces a unique number of every message , so the slaves know if the received message is old or new.

3 Evaluation

Using the gms1 we observe that there is synchronization between the leader and the slaves. All the nodes change the same color in the same time. If you shut down a slave the procedure continues as previously but if we shut down the leader the procedure stops and the remained nodes represent the last state (the last color). This is predictable because our implementation doesn't include the election function.

If we follow the second version we notice that the system runs properly although the leader crashes. The election works and a new leader coordinates the slaves. The problem is that there is inconsistency when a node isn't informed about the last sending state of the leader. The result is that two frames change their colors without synchronization.

The upgraded third version seems to be the best. I started the program with several nodes and I noticed that the reliable multicasting works and the nodes are updated and synchronized during the running.

4 Conclusions

This seminar taught us the importance of synchronization among several application layers that want to have the same state. We get involved with new concepts in the distributed systems such as coordination , leader , election, crash , reliable multicast and we implemented a small group membership service that changes the states of several nodes simultaneously. Moreover , practicing in Erlang for one more seminar learnt us more about functional programming and distributed systems.