Groupy: a group membership service

Antonios Kouzoupis <antkou@kth.se>

October 8, 2014

1 Introduction

In this seminar we have implemented *Groupy*, a group membership service that provides atomic multicast. We assume a network of nodes and each one changes state. The goal is for the other nodes to keep up with that state. In our assignment the "state" is represented by a certain colour, so every node in the network should visualize the same colour (aka state).

Group membership is of vital importance in distributed systems especially when our goal is to tackle with node failures. Fault-tolerant services should be able to keep a consistent state by group communication and keep track of the current *View* of the system. Members of the group should be updated as soon as possible of the active nodes. Unfortunately this depends on fault detection mechanisms and it is difficult to build a perfect one.

2 Main problems and solutions

Each node is separated in two layers. The first one is the *application layer* or *Master*. It is responsible for printing the correct state in screen, receiving/sending messages from/to the *group process* and contacted when a new node wants to join the network.

The second layer of the node is the *group process* or *slave*. Slave's main responsibility is to forward messages from its Master to the group service (leader) and vice versa, from the group service to its Master. Also, slaves monitor the Leader if he/she is alive and in case of the inevitable happen will conduct elections to elect new Leader for the service. The new leader will always be the first one in the list of the joined nodes. Anyone who wishes to multicast to the group should first send the message to Leader and then the Leader will forward it to the members of the group and to its Master of course.

The consistency of our group is based on views delivery. Each node knows what are the active nodes of the network by receiving a list of them. The tricky part was the last one where we should keep track of the last message received and the sequence number of the expected message.

The first implementation of *Groupy* (gms1) is a basic. There is a leader and nodes can join the network. Of course the state of the group is consistent but it can not deal with leader failures. The sole issue I had was that initially windows did not update with the new colour. I fixed it by calling the `wxWindow:refresh(Window)` function after setting the new background colour in *gui.erl*.

The second implementation (gms2) goes one step further and introduces fault detection mechanism and election. We make use of Erlang's `erlang:monitor/2` for every slave process to monitor the Leader. As soon as the Leader is dead we initiate election. Leader is the first one in the list of joined nodes in the network. After that we broadcast the new view of the group to the members.

Finally, in the third version of the group process (gms3) we address the issue of a Leader who dies before multicasting to everyone in the group a message but to some of the members. A solution to this problem is for the newly elected Leader to resend the last message, just in case the previous one did non manage to forward it to all members of the group. So, our process keeps track of the messages and stores always the last one. After the election procedure it multicasts that last message. This solution however, creates another problem, nodes might receive duplicate messages. They have received the message from the previous Leader, just before he died, but also have received it from the new Leader. The final adjustment to the process is to introduce a sequence number with whom each message will be tagged. When a new node bootstraps from another node in the network, it receives the view along with the sequence number. The group process should store what is the expected sequence number. When a slave receives a messages, he/she checks the sequence number. If it is less than the one that is expected, that means he/she has received an already seen message so it is discarded. On the contrary, if the sequence number is equal to the expected one, it means that it is a new message and slave processes it as it should. Of course the Leader should increment by one the expected sequence number after he/she multicasts and slave should also increment by one after he/she has received a message of type *msg* or *view*.

3 Evaluation

For the evaluation procedure I wrote a very simple test case which spawn a leader and 6 other nodes. Each node bootstraps from the previous one. During testing of the first version of *gms* module we can see that the group is in a consistent state since all windows change to the same colour "simultaneously". If a node, which is not the leader, dies everything is fine, the group members are still in sync. The problem arises when the leader dies. Our first implementation is very basic without failure–detection or election.

So when the leader dies everything freezes, nodes try to send new changes to leader in order to multicast them but since the leader is dead there is nobody to broadcast the message to the group, hence colours do not change.

For the second version of group process I used the same test bed. I run some tests before writing the random crash "feature" and after. In both cases there was detection of the leader death and election of new one. The only difference is that when I manually killed the Leader the network continued to be consistent. Colours kept changing the same way. Which mean that the leader died after he has forwarded the last message to all of the members of the group. On the contrary when I introduce the random assassination of the Leader, he/she had not completed yet the multicasting leading to an inconsistent network.

Which lead us to the final implementation of *gms* module. After the death of the leader, even though he/she has not yet delivered the message to all slaves, nodes keep changing their colour in the same way. This happened due to the last message re-sending of the newly elected leader. I have also seen old messages been dropped which shows that the node in question has received a duplicate message but decided to keep that latest.

4 Conclusions

In this assignment we have implemented *Groupy*, a group membership service with atomic multicast. Nodes subscribed to the service are able to receive the same message. Failure detection mechanisms prevent the system collapse even though nodes fail.

Nevertheless there are some issues that might still provoke malfunction to our system. In terms of simplicity we assume that messages do arrive at nodes and not get lost in the middle. To address this issue we could introduce acknowledge messages from slaves back to leader but this would have a great impact to the performance of our system. Also another issue that we have blindfolded is that an erroneous node might not be detected in time and send messages to the network. A possible solution to this problem would be to gather a handful of responses and pick the majority of them as the correct one. Following this way we can exclude an incorrect message from being processed by the nodes of the network.