Groupy — A group membership service

Anton Bothin

September 23, 2019

1 Introduction

The aim of this assignment was to implement atomic multicast to create a group membership service where several processes communicate with each other to keep a coordinated state. This means that they should all perform the same state changes in the same sequence, keeping all processes synchronized.

2 Main problems and solutions

The implementation was done in three steps.

2.1 The first implementation

The overall structure consists of one leader and all other nodes being slaves. The leader is the first process which was started. Each process started afterwards becomes a slave. We keep a list of these slaves ordered by when they joined the group, this is later used in the election process.

In this initial implementation we did not handle failures, meaning that for example: if the leader dies no new leader will be elected.

2.2 Handling failure

To handle the previously mentioned failure each slave now monitors the leader. If a slave notices that the leader has diead it continues to an election state. The implementation of the election state is very basic, we simply takes the first node in the list of slaves and it becomes the new leader. Each node will see the same first element since the list of slaves is ordered by when each slave joined. If the node sees itself as the first element it will change itself into a leader by broadcasting this to each other slave.

During this implementation step we also added a 1/100 risk of the leader crashing to more easily be able to test this failure handling. What we can see happening is that the nodes can become out of sync whenever a leader crashes. This happens since some nodes may receive a message from the leader while others do not.

2.3 Reliable multicast

During this step of the implementation we made an assumption, that messages are delivered reliably. A biproduct of this assumption is that if the leader sands a message to A and then sends that same message to B, and B receives the message, then A will also have received the message. With this assumption we can make the multicast reliable by having each slave store the last message it received. Now if the previous leader dies we can have the new leader resend the last message it received keeping all processes synchronized. This however introduces a new problem, and that is duplicate messages. Because of this we also have each slave keep track of the expected message number from the leader. If the slave now receives a message with a lower message number it means that this message is a duplicate. If not the slave increments the expected message number by one. With this final version processes no longer became out of sync no matter how many leaders died.

3 Evaluation

Erlang only guarantees that messages will arive in FIFO order, not that they will actually arrive. This means that our implementation still can fail since we don't handle this type of failure. One way to solve this would be by allowing the leader to keep a small history of messages. Since each slave knows which messages number to expect, it could request all messages it missed if it receives a higher message number than expected. This would probably work practiaclly since messages are not frequently lost, but theoretically it could still fail if the history no longer contains all messages that a slave has missed. Another solutions is using acknowledgements but this doubles the number of messages being sent.

It was difficult gathering information about what guarantees `monitor` gives. From my understanding `monitor` will send a `'DOWN'` message only when a process dies, if the process does not exist, or if the connection is lost. This leads me to the conclusion that a correct node could be perceived as dead if we have a congested network where there is delay. To handle this we could have the previous leader rejoin the group as a slave if it was perceived dead and a new leader was elected.

4 Conclusions

This assignment taught me some methods used to coordinate states between several processes. It also taught me what types of failures that can arise and how to deal with them. The assignment was good at showing how difficult it is to build a reliable and fault tolerant distributed system.