Distributed Systems - Groupy

Francis Gniady

September 27, 2022

1 Introduction

In this assignment we will be implementing a "group membership service", in the form of multiple workers communicating together in order to coordinate which color they should display. With the goal of having all workers display the same color at all times, and switch color together. We will achieve this by implementing atomic multicast in order to ensure messages get sent to all workers, even in the event of workers crashing in the middle of sending out messages.

2 Main problems and solutions

To make this work we will have two types of nodes, slaves and a single leader, if a slave wants to send a message to the group they will send it to the leader, which will then send it to the entire group.

The first issue that we encounter is the case of the worker acting as the leader going offline, by crashing or any other means. We solve this by having slaves monitor the leader process using erlangs monitor functionality, and for the purposes of the exercise we will assume monitors are fault-proof. Since all slaves monitor the leader process, when it dies they will all receive the DOWN message, we can then have them all go into an election state where they will pick a new leader for the group. Luckily since each worker has a list of all workers in the group ordered by when they joined the group, we can simply pick the first worker in the list and make it our new leader. All workers will go back to being slaves, now monitoring the new leader, except for the first worker in the list who will take on the role of the leader.

We have now solved the first issue of a leader crashing making the whole group inoperable. While our elections work, we now have a new problem. What if the leader crashes in the middle of sending out messages to the workers, with only some of the workers receiving the message. After electing a new leader the workers are now out of sync, with some of them having

received one additional message.
To combat this issue we will assume that a worker can only ever miss one message because of the leader crashing. We can also notice that since the leader sends out the messages to each worker in the order they were added to the group, if the first worker did not receive the message then none of the other workers did either. So in case of only a part of the group receiving a message, the new elected leader would have always received that message. Thus we have each worker keep track of the last message it received, and if it becomes the leader, the first thing it will do is resend that message to the group. To prevent workers who received the message from the previous reader from evaluating the message again, we will send a incrementing number with each message, and have the workers ignore any messages that have a number less than their internal counter, which is updated whenever we receive a message.
We have to remember that when we add workers to the group, we have to start their internal counters at the first message they receive from the group, rather than 0, otherwise they will be behind the other workers in the group.

Lastly, we have the issue of a message to a worker being lost, we can solve this by having each worker send an ack back to the leader whenever they receive a message. If the leader does not get an ack from each worker within a timeframe, it will resend all messages. Workers who already received the messages will simply ack them and ignore the contents, while workers who did not receive them will be brought back into sync.
While this solves the issue of messages being lost, it of course impacts performance, first of all workers will have to send an ack message which the leader has to handle. The implementation could also be improved to only resend messages to workers that did not reply with an ack, which would save on the amount of messages resent. Lastly, with the current implementation, if a message is lost the leader is completely frozen for the duration of waiting for the workers to reply. In time sensitive applications this may not be acceptable.

3 Conclusions

This assignment was a nice introduction to atomic multicast, and while we solved some of the issues that arise when we have a group of processes trying to stay in sync, a lot more problems pop up if we accept the fact that the built in features of erlang that we use are not perfect at all. For example in the assignment we assumed that the erlang monitor functionality is perfect and will never for example think a working node has crashed.
Nevertheless, this assignment was a nice introduction to the algorithm. Demonstrating the state of the system in a visual way was a nice addi-

tion which removed some ambiguity if faults lie with the way we display the state or with our implementation of the algorithm.