HW4 - Groupy

Riccardo Fragale

October 1, 2025

1 Introduction

In this homework, we had to implement a group membership service providing atomic multicast. We are building several application layers that perform the same sequence of state changes. To prove the efficiency of our application we were given a simple GUI which associates a colored window to each created process. All the windows are synchronized and change color when the leader among the processes decides to change state. A node that wishes to perform a state change must first multicast the change to the group so that all nodes can execute it. Since the multicast layer provides total order, all nodes will be synchronized. The problem in this assignment is that all nodes need to be synchronized, even though failures might occur.

2 The implementation

2.1 Basic communication

The architecture consists of a set of nodes where one is the elected leader. All nodes that wish to multicast a message will send the message to the leader, who will do a basic multicast to all group members. First of all, inside the module **gms1**, we must set up communication between the nodes of the system. A new node that wishes to enter the group will contact any node in the group and request to join. Each node in the group should be able to multicast messages to the all members. The communication is divided into views, and messages will be said to be delivered in a view. A node will play the role of a leader or a slave. All slaves will forward messages to the leader, and the leader will tag each message with a sequence number and multicast it to all nodes.

The leader keeps the following information:

- Id: a unique name of the node, only used for debugging;
- Master: the process identifier of the application layer;

- Slaves: an ordered list of the process identifiers of all slaves in the group;
- Group: a list of all application layer processes in the group.

A slave has a much simpler job; it is simply forwarding messages from its master to the leader and vice versa.

After realizing some tests, I saw the leader can crash, but also any other node, so we definitely need to have a system that checks for failures and handles them.

2.2 Failures handling

We developed a second module called **gms2** where we look for node crashes and we setup the election process. To detect possible node crashes we decided to use the Erlang built-in failure detector.

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

We are assuming that the message that informs a process about a death of another process is the last message that it will see from the dead node. The message will thus be received as any other regular message. A slave that detects a leader has died will declare the election state.

```
election(Id, Master, Slaves, [_|Group]) ->
Self = self(),
case Slaves of
    [Self|Rest] ->
            bcast(Id, {view, Slaves, Group}, Rest),
            Master ! {view, Group},
            leader(Id, Master, Rest, Group);
    [Leader|Rest] ->
            erlang:monitor(process, Leader),
            slave(Id, Master, Leader, Rest, Group)
    end.
```

In the election state, the process will select the first node in its lists of peers and elect this one as the leader. Another issue we need to solve when the leader is crashing is that a node may want to join while the election process is on and no leader is active. The message could have been forwarded to a dead leader, and the joining node might never informed that its request was lost. We add a timeout when waiting for an invitation to join the group.

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

Somehow it still happens that some processes get out of sync when the leader crashes as it can be seen from the screenshot below. This means that we need to implement reliable multicast.

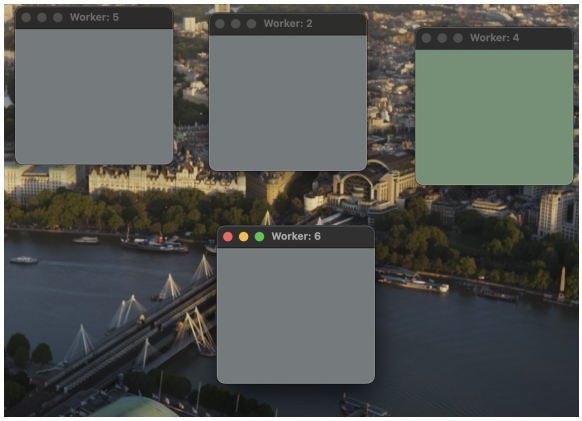

Figure 1: Sync lost

2.3 Reliable multicast

To remedy the problem, inside the module **gms3** we could replace the basic multicaster with a reliable multicaster. The third implementation was updated with a counter to give the sent messages a sequence, being incremented with each new message. To enable reliable multicasting and prevent the previously mentioned sync issue, each slave node also keeps a copy of the last received message. The reelection process was adjusted to re-multicast the last message to ensure the group is kept in sync. Since no record is kept of who already received this message, the sequence number is used to make sure the message is *new* in each s lave node.

2.4 Optional part

In a module called **gms4** I implemented further improvements that take into account the recovery of possibly lost messages due to unreliable links. I introduced a system of **Acknowledgements(ACKs)**. Each node confirms the receipt of a message from the leader. The acknowledgment state is tracked through a map containing the sequence numbers as keys and a map including the original message, a list of nodes it hasn't received acknowledgments from, and a map to track the number of remaining attempts for each nodes as values. The slave nodes were adjusted to randomly ignore received messages to simulate them being lost and acknowledge messages received, even with old sequence numbers to prevent unnecessary retransmission on leader reelection. After a delay, the leader iterates and retransmits messages to the nodes that haven't acknowledged.

This introduces a considerable amount of overhead. To make sure the state doesn't grow in size infinitely, it is continually cleaned to remove fully acknowledged messages and messages where the slave didn't respond within the set retry amount. The biggest logic issue encountered was about slaves

going out of sync after ignoring a message even though they received a retransmit. I traced the issue in the slave process incrementing it's own counter even upon ignored messages. This means acknowledgements were sent back correctly, but the worker didn't update the color as the message was considered *old* by the node.

3 Conclusion and further improvement

First of all, I would opt for a copy of the leader whenever a new one is elected. This imply a huge set of messages between the leader and its copy, that is happening everytime, but it guarantees a higher stability for the entire system since it guarantees a easy recovery. On the other hand it generates a single point of failure. An important point is that Erlang failure detectors may not be completely reliable. It relies on monitoring and timeouts, which means network delays, scheduling hiccups, or transient overloads can cause a correct node to be mistakenly suspected of failure (a false positive). In general, perfect failure detectors are not existing in asynchronous systems. We could introduce a couple of solutions such as:

- Allow rejoining: if a node was incorrectly suspected and excluded, it can rejoin via the standard join protocol.
- Implement protocols like Paxos or Raft, which only require eventual accuracy of the failure detector to guarantee progress.

A third unpredictable situation that may lead to unexpected errors is when one incorrect node delivers a message that any correct node will not deliver. This could happen even if we had reliable send operations and perfect failure detectors. A possible solution to this problems is to introduce a quorum agreement when sending messages or some protocols with replicas and other things to avoid those strange behaviours.