Report - Homework 4

Emil Karlsson

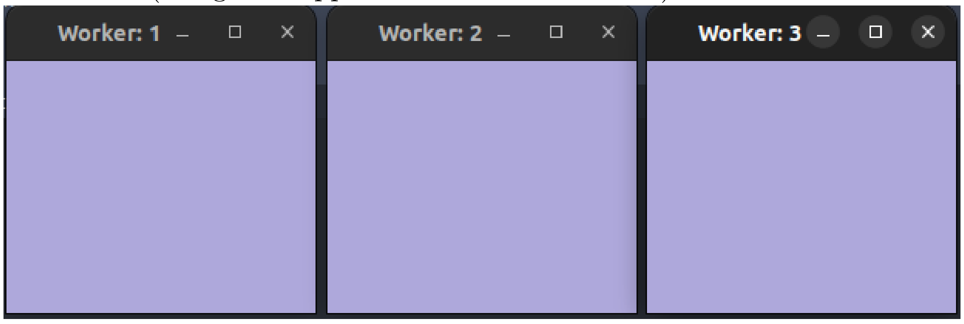
September 29, 2022

1 Introduction

This homework presents challenges regarding managing group membership between nodes in a network. Initially something that appeared as a simple problem, but issues with error and failure handling quickly arose. This report presents the problems encountered along the way, and the methods use to solve them.

2 gms1

The first implementation proved relatively simple and straight forward. No error or failure handling was added, only sending messages between peer. The result of this implementation was as expected, and the result can be seen below (using the supplied *test.erl* from Canvas).



3 gms2

The second implementation added failure detectors. Using Erlang builtin *monitor* function, it was possible to detect if the *leader* process crashed. This function was already used in previous homeworks, and was therefore implemented with ease. Obviously, only the slave nodes needed to listen for these message.

Once a leader crashed, the election process begins to determine a new leader.

Since every node shared the same view, i.e. a list of nodes in the same order, it is as simple as choosing the first in the list, if a node itself is the first in the list, it takes the role as leader.

Finally, if a node tries to connect to a group, but the leader crashes when sending a message to the new node, a timeout was added to prevent the crashing of the new node too.

To verify that it can handle crashes, a fake crash was implemented as chance of happening every time a message is sent. This was set to once 100 messages. The result can be shown below, where 3 nodes were notified of their leaders crash:

```
leader 1: crash
Received DOWN from Master, electing new one...
Received DOWN from Master, electing new one...
Received DOWN from Master, electing new one...
```

4 gms3

The third implementation aimed to fix the issues regarding potentially lost messages when a leader crashes. The idea of the earlier implemented multicast within the group was an iterative unicast over the list of nodes. Since the list of nodes are in a common order, some logic can be extracted. Namely, if a node crashes when sending a messages to the Nth node, Node 1 to N - 1 have received the message, and N + 1 to end have not (whether the Nth node has received the message is not certain in the real world, however because of the way the fake crashing was implemented in this homework it not received it message).

To remedy the problem, each slave saves the previous message received. Since every slave could potentially become the new leader, it will resend the last message to every node. This obviously introduces other problems, primarily since messages now could be duplicates. Therefore, a number was added to every message indicated the oldness of it, and slaves could then filter messages based on this number.

A small code snippet of the election process is shown below:

```
election(Id, Master, N, Last, Slaves, [_ | Group]) ->
  Self = self(),
  case Slaves of
    [Self | Rest] ->
      % Send our last message in case it was not received by the others when we cras
      bcast(Id, Last, Rest),
```

5 gmsbonus

The fourth implementation aimed to solve a more subtle issue, namely lost messages. As mentioned in the PDF, Erlang does not guarantee that messages are received, no acknowledgement is made when sending a message. This could be compared to how UDP and TCP works, and it was exactly what the solution to the issue was based on.

Implementing a TCP-like sending mechanism can be done in multiple ways, however it boils down to sending acknowledgements for every message. For this homework, a simple implementation was chosen: At every message sent from the leader to the slaves, wait for a *okrec* before continuing to the next node, and if the reply does not come within a timeout, resend the message. Further, similar to gms2, a fake message lost was implemented as a change to happen every time a message is sent (not sent is identical as a lost message from the receivers perspective)

A code snipper of the implementation can be seen below:

```
bcast(Id, Msg, List) ->
  lists:foreach(fun(Item) -> ucast(Item, Msg), crash(Id) end, List).

% Unicast function to wrap behavior of lost messages
ucast(To, Msg) ->
  send(To, Msg),
  Result = wait_for_okrec(),
  case Result of
    ok -> ok;
    % Resend if we had a timeout
    timeout ->
      io:format("message timeout, resending~n"),
      ucast(To, Msg)
  end.

send(To, Msg) ->
  Failed = random:uniform(?arghh_msg) == ?arghh_msg,
  if
  % Opsie we failed to send the message, don't do anything
    Failed -> ok;
  % Just a normal send
    true -> To ! Msg
  end.

wait_for_okrec() ->
```

```
receive
  {okrec} ->
    ok
after ?okrec_timeout ->
  timeout
end.
```

Performance

While the solution for lost messages works, it adds a lot of waiting when waiting for replies from the nodes, especially if the nodes would have been far away. This performance lost is negligible in the local environment, and is therefore acceptable. However, in the real world this would most likely cause huge performance issues. Another way of implemented the same feature would be to broadcast the message to every node, then wait for replies. Another entirely different approach would be to periodically sync a form of state describing which message that was received or not. This is comparable to methods used in flow control for TCP.