Seminar Report: Groupy

Daniel García, Aitor Ardila, Oscar Hernández
February 8th, 2016

1 - Introduction

We have created a group membership service that implements atomic multicast with a view synchrony, so all process must follow the same sequence of orders. We also have to deal with nodes crashes. There will be one leader and the rest of nodes will be slaves. All slaves will pass messages from its master process (called “worker”) to the leader. The leader will then multicast the messages to the rest of slaves. In the second implementation we will have to deal with the situation when the leader crashes and a new leader is elected. In the last implementation we will get a full synchronization.

2 - Work done

We have done 3 implementations:

1. Basic implementation: in the first version of the program we implement the atomic multicast but we don’t deal with process failures. If a slave fails, the program still runs as the leader can still send multicast messages to the other slaves. But if the leader goes down, all slaves die as well.

    ➔ Leader implementation:

```
leader(Name, Master, Slaves) ->
    receive
        {mcast, Msg} ->
            bcast(Name, {msg, Msg}, Slaves),  %% TODO: COMPLETE
            %% TODO: ADD SOME CODE
        Master ! {deliver, Msg},
            leader(Name, Master, Slaves);
        {join, Peer} ->
            NewSlaves = lists:append(Slaves, [Peer]),
            bcast(Name, {view, self(), NewSlaves}, NewSlaves),  %% TODO: COMPLETE ---MASTER OR SELF???
            leader(Name, Master, NewSlaves);  %% TODO: COMPLETE
        stop ->
            ok;
        Error ->
            io:format("leader ~s: strange message ~w~n", [Name, Error])
    end.

bcast(_, Msg, Nodes) ->
    lists:foreach(fun(Node) -> Node ! Msg end, Nodes).
```

    The leader can receive a **multicast** message from some slave. In this case it sends a broadcast to all the slaves and it delivers a message to its worker. When the leader

receives a **join** message (a new slave node wants to join the group), it updates the slaves list by adding this new node and sends a broadcast to the new list of slaves.

➔ Slave implementation:

```
slave(Name, Master, Leader, Slaves) ->
    receive
        {mcast, Msg} ->
            %% TODO: ADD SOME CODE
        Leader ! {mcast, Msg}, %%??
            slave(Name, Master, Leader, Slaves);
        {join, Peer} ->
            %% TODO: ADD SOME CODE
        Leader ! {join, Peer}, %%??
            slave(Name, Master, Leader, Slaves);
        {msg, Msg} ->
            %% TODO: ADD SOME CODE
        Master ! {deliver, Msg}, %%??
            slave(Name, Master, Leader, Slaves);
        {view, Leader, NewSlaves} ->
            slave(Name, Master, Leader, NewSlaves);  %% TODO: COMPLETE
        stop ->
            ok;
        Error ->
            io:format("slave ~s: strange message ~w~n", [Name, Error])
    end.
```

A slave can receive the following messages:

- **Mcast**: it happens when the worker sends a multicast message to its slave. The slave re-sends it to the leader.
- **Join**: when the worker wants to join the group. The slave also re-sends this message to the leader.
- **Msg**: it's a message received by the leader. The slave re-sends it to its worker.
- **View**: it's also received by the leader and it announces the new slaves list (the new view). The slave updates its own slaves list.

2. Failure detection: For this version we implement fault tolerance (not fully complete). We introduce a random crash that can happen to any node. Also, each slave monitors the leader node. When a slave detects that the leader has died, it starts an election process to find a new leader.

   First of all, in the init function we add a random seed and when a new slave is created it starts to monitorize the leader (we have to pass a new parameter to the slave with the reference number that returns the monitor call):

```
init(Name, Grp, Master) ->
    Self = self(),
    {A1,A2,A3} = now(),
    random:seed(A1,A2,A3),
    Grp ! {join, Self},
    receive
        {view, Leader, Slaves} ->
            Master ! joined,
            Ref = erlang:monitor(process, Leader), %%NEW!!!
            slave(Name, Master, Leader, Slaves, Ref)
    after 1000 -> %%NEW!!!
        Master ! {error, "no reply from leader"} %%NEW!!!
    end.
```

Slave function:

```
slave(Name, Master, Leader, Slaves, Ref) ->
    receive
        {mcast, Msg} ->
            %% TODO: ADD SOME CODE
        Leader ! {mcast, Msg}, %%??
            slave(Name, Master, Leader, Slaves, Ref);
        {join, Peer} ->
            %% TODO: ADD SOME CODE
        Leader ! {join, Peer}, %%??
            slave(Name, Master, Leader, Slaves, Ref);
        {msg, Msg} ->
            %% TODO: ADD SOME CODE
        Master ! {deliver, Msg}, %%??
            slave(Name, Master, Leader, Slaves, Ref);
        {view, NewLeader, NewSlaves} ->
            erlang:demonitor(Ref, [flush]),
            NewRef = erlang:monitor(process, NewLeader),
            slave(Name, Master, NewLeader, NewSlaves, NewRef);  %% TODO: COMPLETE
        {'DOWN', _Ref, process, Leader, _Reason} -> %%NEW!!!
            election(Name, Master, Slaves); %%NEW!!!
        stop ->
            ok;
        Error ->
            io:format("slave ~s: strange message ~w~n", [Name, Error])
    end.
```

When the slave receives a new view we stop monitoring and start to monitor the new leader. A message “DOWN” is sent and then the slave that has detected the crash calls election function.

In the election function the first process from the list of slaves is chosen to become the new leader. There are 2 cases: the first node from the list is the slave itself or it’s another one. In the first case it broadcasts the new view to the others and calls leader function. In the second case, it starts monitoring the new leader and calls slave.

```
election(Name, Master, Slaves) ->
    Self = self(),
    case Slaves of
        [Self|Rest] ->
            %%TODO: ADD SOME CODE HERE
            bcast(Name, {view, Self, Rest}, Rest),
            leader(Name, Master, Rest);  %%TODO: COMPLETE
        [NewLeader|Rest] ->
            %% TODO: ADD SOME CODE HERE
            NewRef = erlang:monitor(process, NewLeader), %%NEW!!!
            slave(Name, Master, NewLeader, Rest, NewRef) %%TODO: COMPLETE
    end.
```

3. Reliable multicast

Leader

We introduced parameter ‘N’ which is the message sequence ID.
Leader broadcast to its slaves next sequence ID.

```
leader(Name, Master, Slaves, N) ->
    receive
        {mcast, Msg} ->
            bcast(Name, {msg, Msg, N+1}, Slaves),  %% TODO: COMPLETE
            %% TODO: ADD SOME CODE
            Master ! {deliver, Msg},
            leader(Name, Master, Slaves, N+1);
        {join, Peer} ->
            NewSlaves = lists:append(Slaves, [Peer]),
            bcast(Name, {view, self(), NewSlaves, N+1}, NewSlaves),  %% TODO: COMPLETE
            leader(Name, Master, NewSlaves, N+1);  %% TODO: COMPLETE
        stop ->
            ok;
        Error ->
            io:format("leader ~s: strange message ~w~n", [Name, Error])
    end.
```

Slave

Now Slaves have two additional parameters : ‘N’ and ‘Last’ which is the last seen message from the leader.

If received message is greater than sequence number ‘N’ it will be forwarded else it will be discarded.

```
slave(Name, Master, Leader, Slaves, Ref, N, Last) ->
    receive
        {mcast, Msg} ->
            %% TODO: ADD SOME CODE
        Leader ! {mcast, Msg}, %%??
            slave(Name, Master, Leader, Slaves, Ref, N, Last);
        {join, Peer} ->
            %% TODO: ADD SOME CODE
        Leader ! {join, Peer}, %%??
            slave(Name, Master, Leader, Slaves, Ref, N, Last);
        {msg, Msg, Id} ->
            %% TODO: ADD SOME CODE
            if Id > N ->
               Master ! {deliver, Msg}, %%??
               slave(Name, Master, Leader, Slaves, Ref, Id, Msg);
            true ->
                slave(Name, Master, Leader, Slaves, Ref, Id, Msg)
            end;
        {view, NewLeader, NewSlaves, Id} ->
             if Id > N ->
               erlang:demonitor(Ref, [flush]),
               NewRef = erlang:monitor(process, NewLeader),
               slave(Name, Master, NewLeader, NewSlaves, NewRef, Id, Last);  %% TODO: COMPLETE
            true ->
                slave(Name, Master, NewLeader, NewSlaves, Ref, Id, Last)
            end;
        {'DOWN', _Ref, process, Leader, _Reason} -> %%NEW!!!
            election(Name, Master, Slaves, N, Last); %%NEW!!!
        stop ->
            ok;
        Error ->
            io:format("slave ~s: strange messageshit ~w~n ", [Name, Error])
    end.
```

Election

The new election takes into account that a possible loss of information can have been possible during the last message transmission before failure. So the new leader elected will broadcast the last message sent by the old leader. Like in the old version, the next step will be to notify all the slaves it is the new leader and also to provide the new view, but in this case it will add the sequence number increased by one.

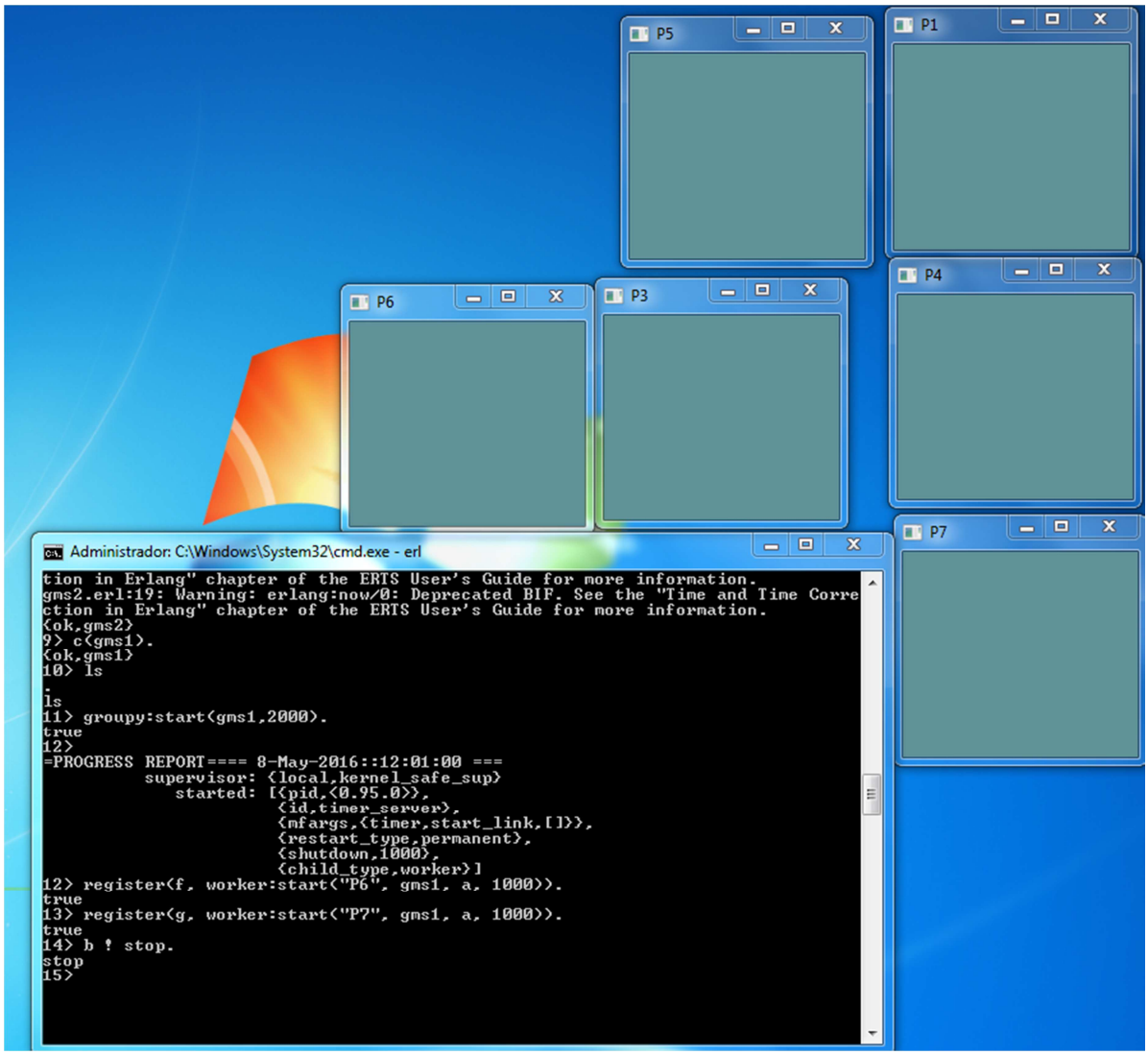
```
election(Name, Master, Slaves, N, Last) ->
    Self = self(),
    case Slaves of
        [Self|Rest] ->
            %%TODO: ADD SOME CODE HERE
            bcast(Name, {msg, Last, N}, Rest),
            bcast(Name, {view, Self, Rest, N+1}, Rest),
            leader(Name, Master, Rest, N+1);  %%TODO: COMPLETE
        [NewLeader|Rest] ->
            %% TODO: ADD SOME CODE HERE
           NewRef = erlang:monitor(process, NewLeader), %%NEW!!!
            slave(Name, Master, NewLeader, Rest, NewRef, N, Last) %%TODO: COMPLETE
    end.
```

3 - Experiments

BASIC IMPLEMENTATION

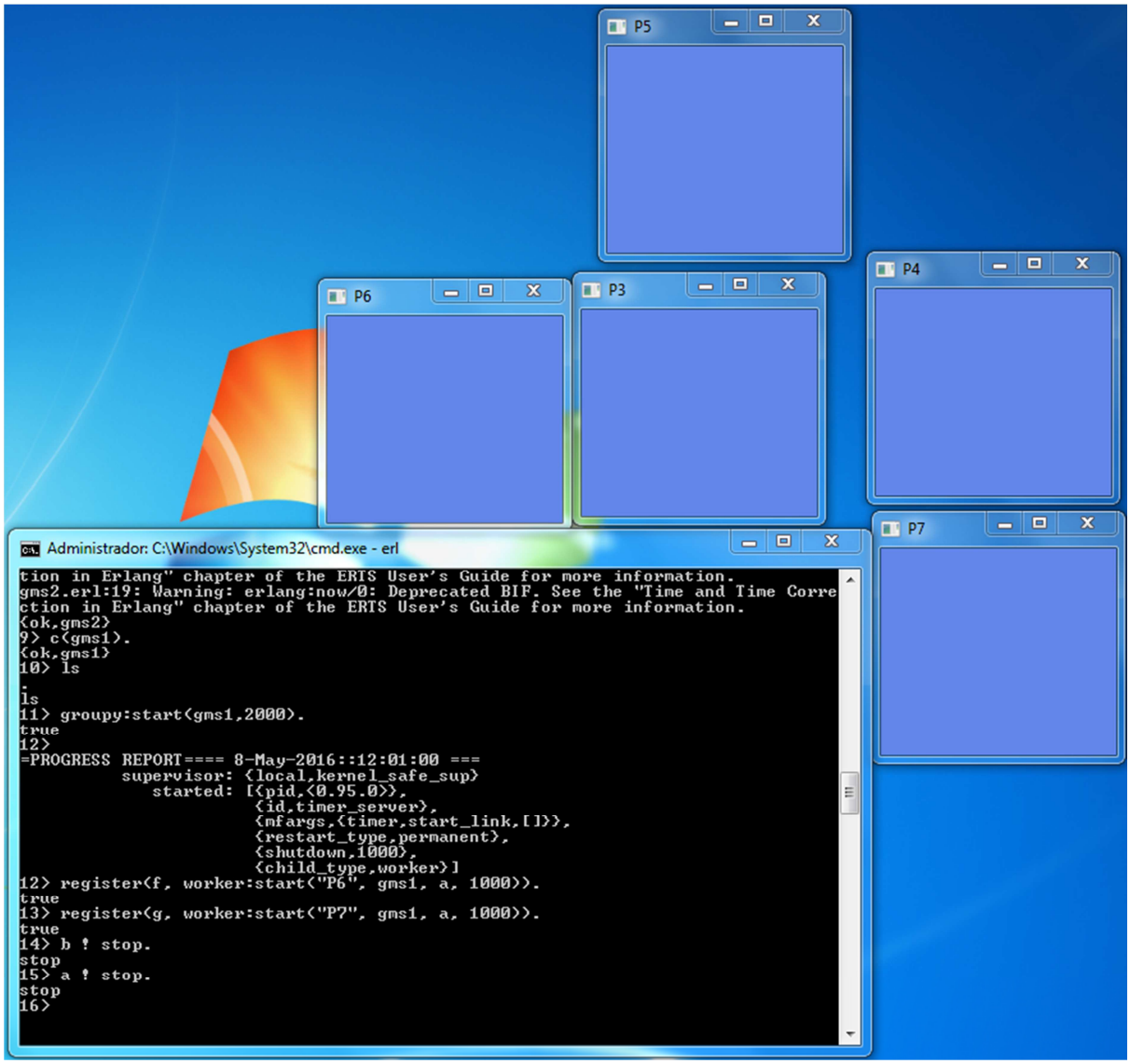
First of all, we implemented the first version which is a basic atomic multicast without failure detection. We have done the following tests:

- Create 2 new nodes and then kill 1 slave:



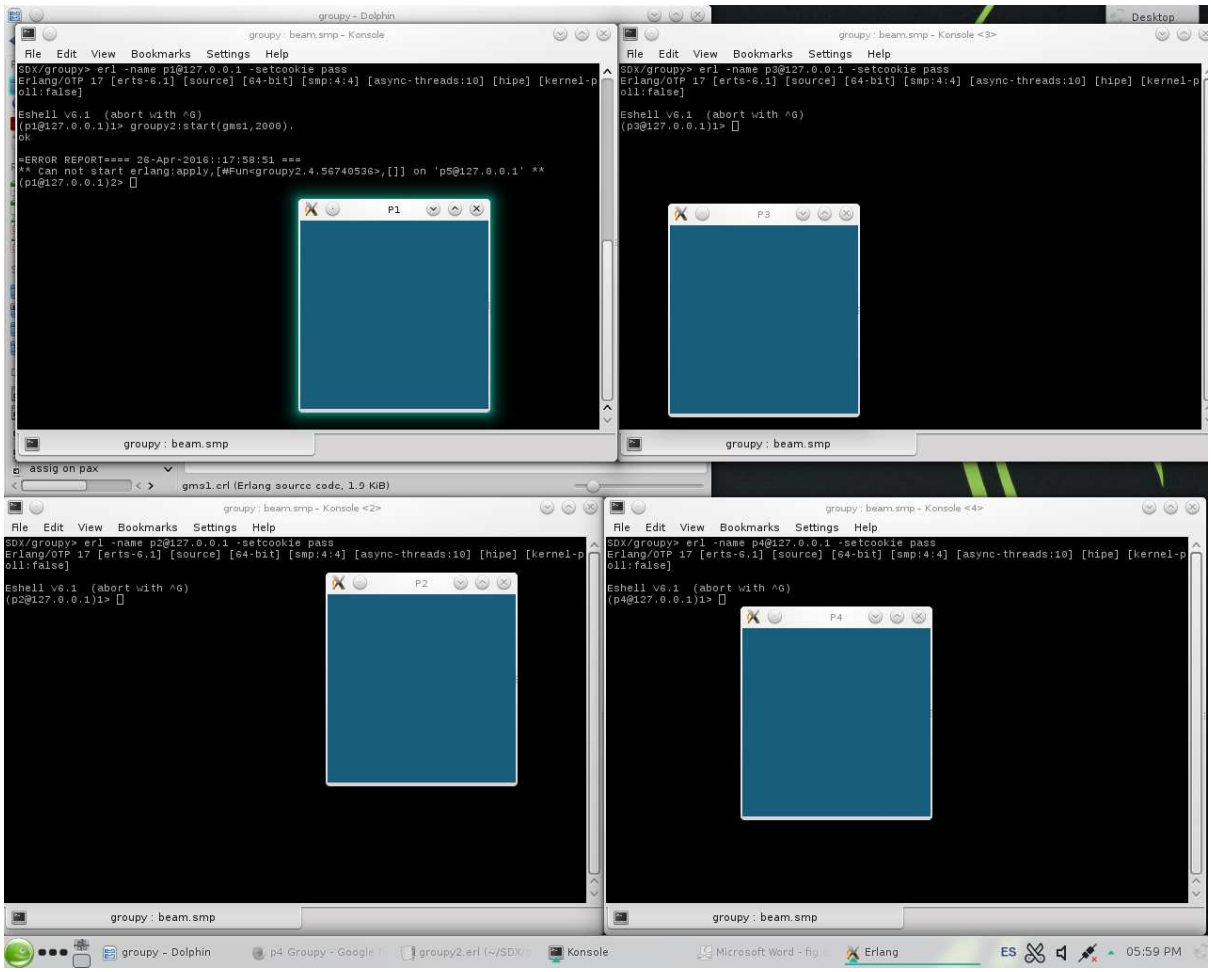
After registering the nodes (P6 and P7), they automatically join the group and receive the leader updates at the same time as the other nodes. After we kill one slave node (P2) the program is still running because the leader is still alive and can send multicast messages to the remaining slaves.

Now let’s see what happens when we kill the leader:



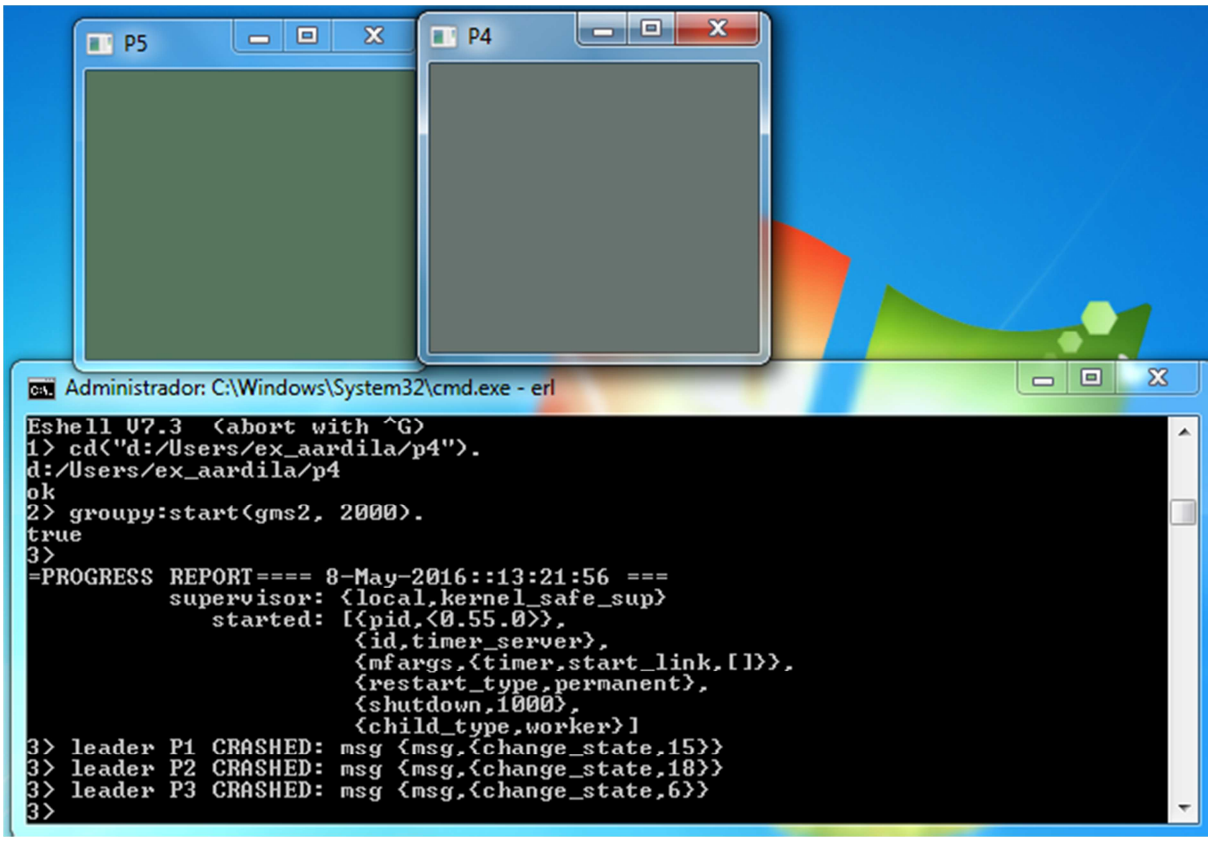
We send a stop message to the leader (“a ! stop.”). We cannot see it from the screenshot as it’s static image but the program actually stops and the colours are not changed anymore.

The second experiment that we have done consists in splitting the groupy module to enable each worker to run in different machines (to be used in a distributed architecture). As we can see from the screenshot we get 4 different machines running a worker each of them:



FAILURE DETECTION

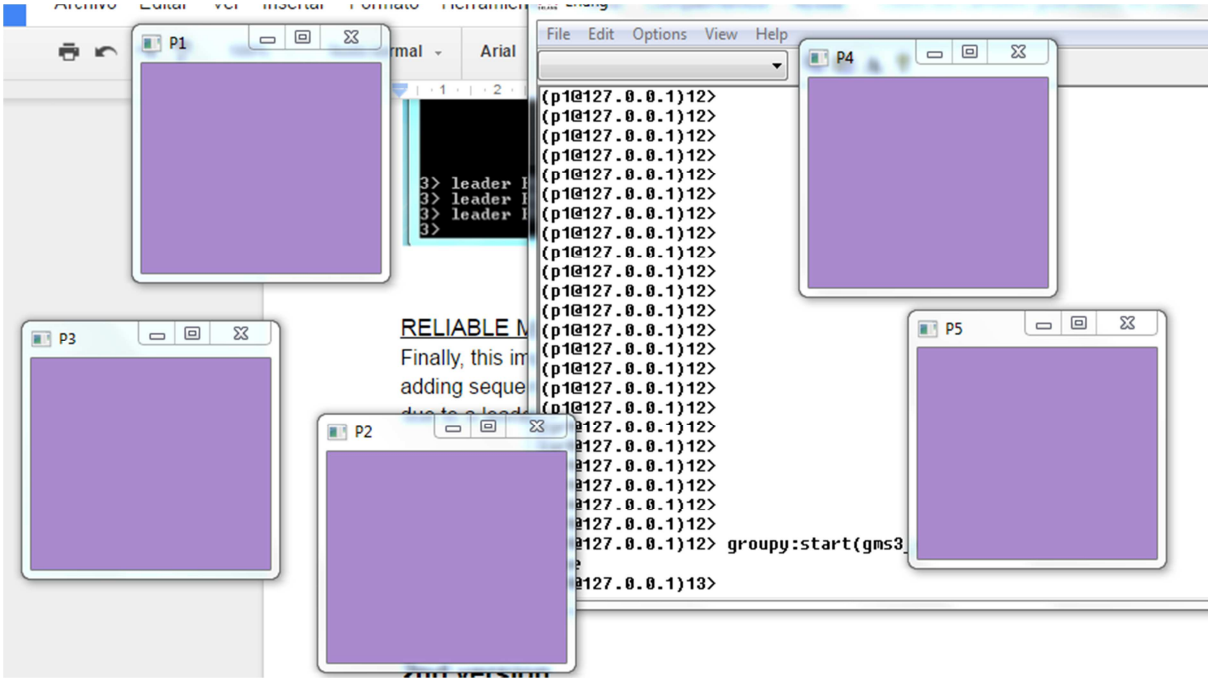
We run the program and see that the first leader crashes and then P2 becomes the new leader, which also crashes and then P3 becomes the new leader. We can see that the messages are not synchronized anymore because the leader might have crashed before sending the multicast.



RELIABLE MULTICAST

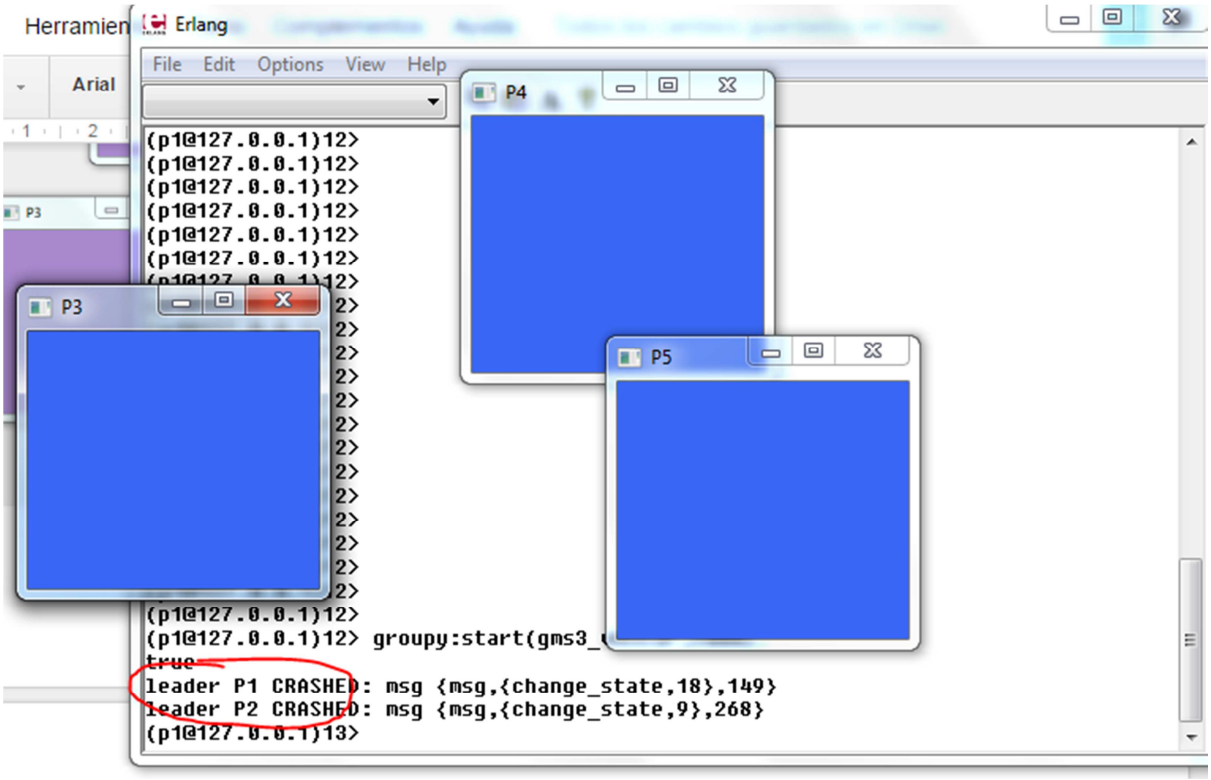
Finally, this implementation tries to avoid the previous failures. This can be achieved by adding sequential id messages and forwarding some messages that couldn't be delivered due to a leader crash during multicast.

As we can see in the screenshot nodes are started like in the previous experiments.



All processes start and they can fail with a 'p' probability.

See that P1 and P2 failed but system still working sync.



4 - Open questions

2nd version

Why do the workers become out of synch?

This happens when the leader crashes before sending the multicast.

3th version

How would we change the implementation to handle the possibly lost messages?

-We need to introduce sequenced messages and a layer that knows the last message sent by leader. This layer consists on the new leader forwarding the last message to all nodes in order to avoid the previous loss of information.
Nodes look for duplicated messages and discard them.

How would this impact performance?

-In the case that lots of failures affected the system, the network would be overloaded with replicated messages. But in general this solution doesn't imply too much overhead.

What would happen if we wrongly suspect the leader to have crashed?

- We would have two leaders sending broadcast messages. Fact that would make the system stop working correctly.

5 - Personal opinion

In our opinion, this seminar is a good approach to the atomic multicast with Erlang. Thanks to the colours of the nodes we can easily see if the synchronization is working or not. Each version of the code improves the previous one and at the end we get a perfect synchronization using atomic multicast.