HW4: Groupy - A Group Membership Service

Jian Wen

October 4, 2023

1 Introduction

In this assignment, I have implemented a group membership management service using Erlang. In this group system, we adopt a star-distributed system with the leader at the center connected to all the slaves in a single group. The slaves always send messages to its leader through unicast, while leader always use multicast to synchronize the states and update view.

I think one challenge in this assignment is to make sure that all the active nodes in our group still gets synchronized even though part of nodes have disconnected. If the leader who multicast states crushed, the rest of nodes should launch a new election to find out a new leader and carry on the multicast. All the transmission between the nodes will be using Master, which acts as the application layer. In this case, it's important to find out the structure of such system and the relationship between different layer.

2 Main problems and solutions

2.1 System Structure

As memtioned above, the group service discussed in this assignment is actually quite complex. We have a number of nodes in our system: one of them act as a leader, the rest act as slaves. For each node, there will be application layer, group layer and network layer, and each layer runs a process. The application layer is used to deal with state-changing messages from group layer; The group layer is used to communicate with the one at the other group members; The network layer is not implemented in this assignment because Erlang will deal with this.

2.2 Deal With Crush

Different from gms1 where we simply allow new node to join and get synchronized with the rest of group members, gms2 require the group being capable of dealing with possible failures. If a node is crushed, it is fine; but if the leader has crushed, we need to elect a new leader from the rest of

nodes. In our implementation, we just need to let the rest of nodes select the first node from their list as the leader, and problem solved!

```
slave(Id, Master, Leader, Slaves, Group) ->
  receive
    ...
    {'DOWN', _Ref, process, Leader, _Reason} ->
    % If leader crushed, start new election.
      election(Id, Master, Slaves, Group);
    ...
election(Id, Master, Slaves, [_|NewGroup]) ->
% [_|NewGroup]: New group without former leader.
  Self = self(),
  case Slaves of
    [Self|Rest] ->  % Leader is current node -> Broadcast.
      bcast(Id,{view, [Self|Rest], NewGroup}, Rest),
      Master ! {view, NewGroup},
      io:format("leader ~w: This is leader. ~n", [Id]),
      leader(Id, Master, Rest, NewGroup);
    [Leader|Rest] ->  % Leader is another node -> Wait for it.
      erlang:monitor(process, Leader),
      io:format("slave ~w: This is slave. ~n", [Id]),
      slave(Id, Master, Leader, Rest, NewGroup)
  end.
```

2.3 Reliable Multicast

From gms2 to gms3, we need to make sure the messages arrive at all of the connected nodes within our group. One possible situation is that, a leader may get crushed when it is doing a multicast. In this case, only part of nodes can successfully receive the multicast while the rest of nodes cannot because of interruption. This can be solved by letting the new elected leader to repeat the last message sent by the former leader. So how can the new elected leader know the last multicast message? This can be done because the new leader we picked is placed on the front of list on the former leader's multicast targets, and it will definitely receive the last message. All this new leader have to do is to, once it becomes a leader, get the last message it receive, and do the multicast. At the same time, we also attach a sequence number for the message and let nodes ignore the already-received message by comparing and updating the local tracker. This can eliminate the possible re-transmission and duplex messages.

```
election(Id, Master, N, Last, Slaves, [_|NewGroup])->
  Self = self(),
```

```
case Slaves of
  [Self|Rest]->
    bcast(Id, Last, Rest),
    bcast(Id, {view, N,[Self|Rest], NewGroup}, Rest),
    Master ! {view, NewGroup},
    io:format("~wleader ~w: This is leader.~nNew group: ~w~n", [self(), Id, NewGro
    leader(Id, Master, N+1, Rest, NewGroup);
  [Leader|Rest]->
    erlang:monitor(process, Leader),
    io:format("~wslave ~w: This is slave. ~n", [self(), Id]),
    slave(Id, Master, Leader, N, Last, Rest, NewGroup)
end.
```

3 Evaluation

We can see whether the nodes are synchronized in a group by looking at the GUI window. The window of those conncted nodes should display the same color.

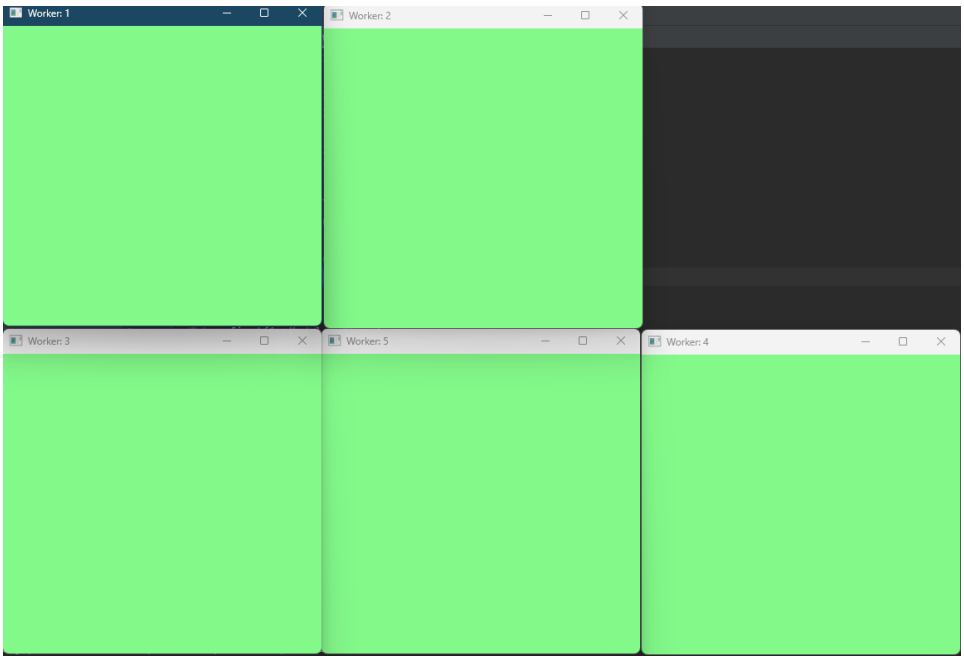

Figure 1: GUI from different connected nodes.

When a new node joins, it will send join request to one of group members, and then receive view from the leader. The leader will update the multicast list by adding the new node into it.

```
3> test:add(3, gms3, W1, 1000).
slave 3 send join to worker_pid:<0.87.0>
<0.99.0>
slave 3 wait for view
worker 1 receive join request from slave 3
leader 1 receive join request from its master_pid<0.87.0>
leader 1 announce: New member join!
NewGroup [<0.87.0>,<0.94.0>,<0.99.0>]
NewSlave[<0.95.0>,<0.100.0>]
slave 3: got view
slave 3: send view to its master
<0.88.0> leader:1 bcast message 37: {state_request,#Ref<0.3470104140.2606235650.257113>} to [<0.95.0>,<0.100.0>]
<0.88.0> leader:1 bcast message 38: {state,#Ref<0.3470104140.2606235650.257113>,{78,8,55}} to [<0.95.0>,<0.100.0>]
<0.88.0> leader:1 bcast message 39: {state,#Ref<0.3470104140.2606235650.257113>,{78,8,55}} to [<0.95.0>,<0.100.0>]
<0.88.0> leader:1 bcast message 40: {change,3} to [<0.95.0>,<0.100.0>]
```

Figure 2: New node join.

When a leader has crushed, the rest of the nodes, which are keep monitoring the leader, will get the DOWN message and start election.

```
leader 1: crash
<0.95.0>leader 2: This is leader.
New group: [<0.94.0>,<0.99.0>]
<0.100.0>slave 3: This is slave.
<0.95.0> leader:2 bcast message 122: {change,16} to [<0.100.0>]
<0.95.0> leader:2 bcast message 123: {change,24} to [<0.100.0>]
<0.95.0> leader:2 bcast message 124: {change,8} to [<0.100.0>]
```

Figure 3: Launch election when leader crushes.

4 Conclusion

In summary, this assignment involved managing a star-distributed group with a central leader and multiple slaves. Challenges included ensuring synchronization even during node disconnections and implementing leader election. Erlang's fault tolerance and concurrency were leveraged, and the Master application layer facilitated communication. This assignment deepened my understanding of distributed systems and system organization, preparing myself for real-world scenarios.