Groupy: Group Membership Service

Weherage, Pradeep Peiris

March 4, 2016

1 Introduction

Groupy is a group membership service implemented to examin the behavior of process coordination in a distributed system. As the use-case a simple Erlang application is used, which comunicate its internal state change with all the peer applications.

The group communication system is consisted one or more of these applications, and it's each application's task to properly communicate its internal state (Color) change with peer applications.

2 Main problems and solutions

The main task for each application is to properly synchornized its internal state with its peer applications.

This is achieved with having each application an Application Layer process. The application layer process has its own group process, which is responsible for the Group Communication. The application layer send multicast messages to the group process and receive all multi cast messages from it.

Therefore, system architecture consider here contains a collection of process nodes. All nodes that wish to multicast a message first send the message to a special node, called leader. The leader does a basic multicast to all members of the group.

In our implementation, the leader process signature contains parameter of, Id: Unique identifier, Master: Application Layer process, Slaves: slaves processes and Group: A list of Application layer processes in the group. When the leader receive a message it multicasts to all peer applications.

The slaves process nodes simply forwarding messages from its master to the leader and vice verse. In addition to that slaves keep explicit track of the leader.

2.1 Failure Detection

As stated above the slave nodes keep explicit track of the leader for any failures. In real scenario the failure detection is quite problematics. And they are not necessarily accurate.

There are two categories of failure detectors, Unreliable and Reliable failure detectors. Most failure detectors are unreliable which indicates a process is Unsuspected or Suspected to be failed with no guarantee. But reliable detectors has a guarantee when it reports Failed.

In this Erlang implementation of process, we use its inbuilt support of failure monitoring for processes. The slaves processes let monitor the leader process via *erlang:monitor* function. At failure the slave process is called for an election.

2.2 Election

In real scenario Election is an algorithm for choosing unique process, the leader to play specific tasks. In our implementation, the slave nodes select the first node in the peer list as the leader.

3 Evaluation

It is evaluated that without failure detection, the application stop functioning, that is shearing its internal state changes (Broadcast of the state change message) with peer appications once the leader node is crashed or stopped. It is overcome with introduction of failure detection and electing a new leader node from the peer node list.

3.1 Missing Messages

It is experiment that letting the broadcast procedure to lose some messages causes groups nodes become out of sync.

```
bcast(_, Msg, Nodes) ->
   lists:foreach(fun(Node) ->
    case random:uniform(?arghh) of
     ?arghh ->
       io:format("missing message~n", []);
     _ ->
       Node ! Msg
   end
 end, Nodes).
```

3.2 Reliable Multicast

In reliable multicast it guarantee that all correct processes in the group must receive a message if any of them does. In our basic multicast model we evaluated that some nodes may deliver a message while others do not with random crash. And this causes application layer to be out of synced with other peer applications. We improve our multicast process keeping a copy of the last message received from leader node. At Leader node failure, now slave node could forward the last message which may have not seen from other nodes.

```
election(Id, Master, N, Last, Slaves, [_|Group]) ->
  Self = self(),
  case Slaves of
   [Self|Rest] ->
     bcast(Id, Last, Rest),
     bcast(Id, {view, N, Slaves, Group}, Rest),
     Master ! {view, Group},
     leader(Id, Master, N+1, Rest, Group);
   [Leader|Rest] ->
    erlang:monitor(process, Leader),
    slave(Id, Master, Leader, N, Last, Rest, Group)
 end.
```

4 Conclusions

In this experiment, the process groups were implemented on Erlang system, which guarantee order of message delivery and and reliable message delivery. To achieve this in real scenario the group multicast should be implemented on top of a protocol which guarantees those feature. But an additional layer may cause performance impact.