Groupy: A GMS implementation

Tobias Johansson

October 8, 2015

1 Introduction

A group membership service (GMS) have been implemented in Erlang that provides reliable multicast. The main problem to be solved is to keep the application layer processes coordinated, in the same state. To achieve this the interaction between the group layer and application layer needed to be understood, leader election to be implemented for if the leader crashes and sequencing of broadcasted messages handled.

2 Main problems and solutions

An Erlang module for a group membership service with reliable multicast have been implemented. The GMS acts as a group layer that provides transparent multicast for an above application layer. Three modules where given initially; **gms1** the primitive GMS module, **worker** the application layer, **gui** its graphical user interface and **test** that provided tests that showed the systems functionality.

The development of the final GMS was made in three steps, with three versions of the GMS, each following more advanced, as following:

1. **gms1** and the other given modules were tested and analysed.
2. **gms2** was implemented with a leader election in the circumstance of that leader crashed, and in the third part the module.
3. **gms3** introduced reliable multicast.

To the second step in the development code was given for election functionality and to the third part guidelines was given for a solution.

2.1 gms1

The initial GMS, that is **gms1** relied on a predetermined leader node, if this node crashed the provided multicast did go down. Much time were put

into be able to grasp the system because of it was confusing from first for the reports author with multiple layers. A summary of the analyse of the system is therefore stated below.

The the application layers API **worker** has two start functions; for either start as a group leader or slave to join an existing group. The GMS also haves a start function for either case. Back to the application layer, both direct leaders and slaves spawns a **gms1** instance with its corresponding start functions and saves their PID. If started as a slave, the application layer waits for get a response that it could join and then get the groups current state, while a leader just chooses a state. After the start then the runtime for slaves and leaders are the same in the application layer; receiving messages for change of state, messages that just should be cast further to the group layer like multicast, and also multicast a change of state (so that the systems functionality could be observed). The only the application layer now knows about the group layer now is the PID to its own GMS instance.

To get a grasp of how the two layers worked together, functions were implemented to print when a **worker** instance created a **gms1** instance and to show the leaders status. This implementation gives the terminal output below when starting a group with 2 nodes with the **test** module as following:

```
> test:more(2, gms1, 1000).
worker 1 <0.35.0>: spawned leader <0.37.0>
worker 2 <0.36.0>: spawned slave  <0.38.0> to join <0.35.0>
leader 1: group =[<0.35.0>]
          slaves=[]
leader 1: group =[<0.35.0>,<0.36.0>]
          slaves=[<0.38.0>]
```

Were PID is shown for both **worker** and GMS instances on start. The output after "group" and "slaves" show what PID:s the leader keeps track of in the application layer and group layer. Each slave also keeps track of the same "group" and "slaves", this data gets updated in all nodes when a node joins the group through a broadcast from the leader.

When a application layer instance multicasts a message it casts the multicast to its group layer instance. If the receiver in the group layer is the leader it broadcasts the message to all it slaves and if the receiver is a slave it the casts the request further to its leader.

2.2 gms2

In this part of the development an leader election was added and a timeout on the join request to a group with given code. The timeout on join was tested to get an insight of the communication between the group and application layer. To monitor if the leader crashed the built in function

`erlang:monitor/2` was used on all slaves, they then receives a message on the format:

```
{'DOWN', _MonitorRef, process, Leader, _Info}
```

When the leader crashes and calls the election function. The election function gives the leadership to the first in a list over the slaves PID in the group layer, all group members have the same list, and the rest starts to monitor the new leader. An observation is that if the new elected leader already have crashed then `'DOWN'` is sent immediately, starting a re-election.

2.3 gms3

At last reliable multicast was about to be implemented. To achieve this a logical clock, a counter was introduced to give all broadcasts a sequence number (only the leader broadcasts). The leader keeps track of the sequence number and increase it before broadcast while the slaves keep track of the expected sequence number of the next message and a copy of the last received message.

With the above implemented it was added that the new leader after an election resend the last message that it received and the slaves will have to monitor the new leader. Duplicate messages for the receivers was filtered out and ignored with help of the sequence number.

Experiments where done on the solution, also over multiple Erlang shells, one setup setup was the following:

```
> % node1@127.0.0.1
> register(w5, test:add(5, gms3, test:more(4,gms3,5000), 5000)).

> % node2@127.0.0.1
> register(w6, test:add(6, gms3, {w5,'node1@127.0.0.1'}, 5000)).

> % node3@127.0.0.1
> test:add(7, gms3, {w6,'node2@127.0.0.1'}, 5000).
```

3 Evaluation

The problem with **gms2** occurs when the leader crashes during a broadcast, the *agreement requirement* (if a node delivers a message then all nodes will) is violated. If a state change is multicasted this leads to that nodes in the group get set to different state. This could be seen through test there the nodes became out of sync.

With **gms3** this issue was attempted to be solved. When experiments where done on the solution the nodes never became out of sync.

4 Conclusions

The final implementation works fine in the test environment, but it does not handle lost messages since Erlang only guarantee that messages are delivered in FIFO order, not that they actually do arrive. To handle this would be costly but since we don't use vanilla reliable multicaster, instead a much more economical solution, the net result may still be good performance. This is proposed to be solved with an ACK sent from receiver (slave) to broadcaster (leader) and the message sent again if the leader does not receive the ACK, after a timeout, the message is sent again. To preserve total order the following messages, with higher sequence number, must be put in a hold-back queue at either the leaders or slaves.

Another issue is the Erlang failure detectors is not perfect. The monitors is supposed to be false-positive and `'DOWN'` even if the leader have not crashed due to inactivity. This is thought to be solved with dummy operations made by the leader to show that they are not dead.

It is said by Johan Montelius that a situation could occur there an incorrect node delivers a message that will not be delivered by any correct node even though reliable send operations and perfect failure detectors. This could not be found. Since a node never updates its own state directly and instead sends a multicast to the leader, that broadcast to all nodes with a reliable communication. And if the multicasting node is the leader then it broadcasts before updating its own state, so how could this happened?