Report : Groupy, an Erlang implementation of a Group Membership Service

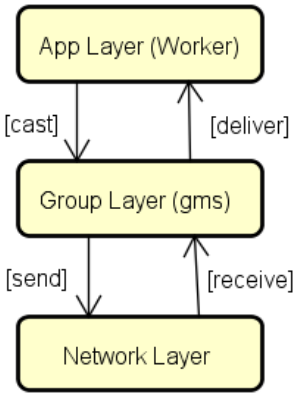
Avneesh Vyas

avneesh@kth.se

October 05, 2016

1 Introduction

The report describes the various steps and procedures followed during the implementation of an erlang based Group Membership Service (GMS). The author started with the bare minimal skeletal implementation of GMS based atomic multicast and gradually enhanced it to handle problems typically faced by a multicast system in a distributed environment.
To understand multicast strategy and protocol used in the implementation, the following visualization of a typical node in a multicast group is helpful:



Here an application layer modeled as a worker process uses a low lying group membership service, modeled as GMS process for casting messages to a group. GMS process hides all details related to actual multicasting and fault-tolerance from the Worker process. Also, whenever a message is received on the network interface, GMS delivers it to the Worker. New workers can join a particular group by requesting any worker process which in turn informs its own GMS about the new peer worker. GMS through broadcast messages ensures that all GMS processes maintain an identical list of peer nodes or view synchrony. Here 'view' refers to the network view of a GMS process.

2 Main Requirements for atomic multicast

A basic multicast service is one which guarantees that a correct process will eventually deliver the message. But such a service is not very fault tolerant. For example, what if the multicaster crashes after sending the message to only a subset of nodes belonging to a group. Most practical problems expect 'all or nothing' approach. And this is where atomic multicast is useful. An atomic multicast satisfies the following main requirements

1. Integrity: a message is delivered to an application layer at most once.
2. Validity: if a correct node multicasts a message, it will eventually be delivered.
3. Agreement: If one node delivers a message, then all nodes deliver it.
4. Total ordering: All nodes see the same sequence of messages.

3 Solution

Election of leader: To support atomic multicast, gms layer processes choose a leader among themselves. To multicast a message, the particular gms process will send the message to leader process which will eventually broadcast it across all peer gms processes. Individual processes will then deliver the message to their respective worker process.

Fault tolerance (leader crashes): To handle the scenario where leader process dies, the gms processes will have a mechanism to choose a new leader. As soon as new leader process takes charge, it will re-send the 'last' message it received from the previous leader to meet the 'agreement' requirement. All gms processes will ignore the message which they have already received by checking against the expected sequence number. A lower incoming sequence number than expected indicates duplicate message.

Fault Tolerance (Non-leader crashes): For simplicity, such crash was not monitored. Ideally every node crash should trigger removal of that node from the view.

New nodes joining in: Any new node may send a request to any member node. On receiving such a request from the worker, the gms layer will forward it to the leader process which in turn will broadcast the new view to all peer nodes. This way, all nodes maintain the same view of the network.

Possibly lost messages: Each gms layer will keep track of the next expected sequence number. If a process notices that the incoming sequence number is higher than the expected value, then it knows that it has missed some messages. To allow recovery of such messages, leader

process can maintain a list of all broadcasted messages mapped against its sequence number. This way, nodes can query lost messages from the leader and then deliver to higher worker process. Although this may work but it will not be trivial for leader to maintain a history of all broadcasted messages, especially in systems with large volume of messages.

Process Failure Detection: Our implementation relies on Erlang's process failure detection which is not perfect and it can indicate a failure for a leader process even though it is alive. In such cases, the process can use a heartbeat message towards the leader message and wait for a response. If it does not receive a response within a timeout period, it can assume that the leader process is dead.

4 Verification

The implementation was tested for scenarios where leader crashed. It was observed that through leader election, the rest of the group was still able to maintain synchrony.

5 Conclusion

Through this assignment, author was introduced to techniques involved in building a reliable multicast service which is fault tolerant to nodes leaving/joining the group. Also, it gave an opportunity to understand design goals of a reliable multicast system.