Report 4: Groupy

Mallu Mallu

October 06,2016

1.Introduction

In this homework I have implemented a group membership service in erlang which obeys atomic multicast in view synchrony. Atomic multicast preserves total ordering as well as reliable multicasting, total ordering can be defined as If a correct process delivers message C before it delivers D, then any other correct process that delivers D will deliver C before D. Main aim in this assignment is that all application layer processes should exist in a coordinated and synchronized state. For implementing gui WxWidget toolkit has been that creates worker processes in separate windows and when they multicast same colour is observed among them.

2. Main Problems & Solutions

a). Building Basic gms1

For the basic multicast group membership service gms1 is implemented, the architecture is implemented in a way that a group of nodes has one leader elected and the message flow is if a node wants to multicast a message to the network then it should first send it to the leader and leader will do a multicast. The application layer will send multicast messages to the group process and receive all multicasted messages from them. Also, FIFO. Order is followed in message delivery. In gms1 implementation starting a process and then adding more nodes in the network. Also the very first process started becomes the leader. **Leader** is responsible for handling messages coming from its Master as well from peers. For gms1 we initialized the first node using start(Id), init (id Master) and being the first node it is initialized as leader. Also, the same way slaves are initialized and then we run the test module which gives a very basic view synchrony among workers (peer processes).

b). Implementing fault Tolerance and monitoring for crashed nodes gms2

In gms1if a leader crashes there is no notification message or election for the new leader among alive processes. So this failure detection is implemented in gms2 and slave processes are selected to monitor using **erlang:monitor(process, Leader)** Also, if the leader crashes, election of a new leader is implemnetd using election function in gms2. Also a constant value **“arghh”** is introduced i.e. if the value 100 it means that the process will crash once in 100 attempts. One more modification that has been made in gms2 is seeding random generator to prevent crashing at same time for the processes.

seed() -> ran(). Seeds random number generation with default (fixed) values in the process dictionary and returns the old state.

c). Implementing reliable multicast

Taking gms2 as reference I moved in implementing more reliable multicast by printing messages that are multicasted. To keep the message records and also for handling duplicate messages I have added two arguments N and Last in slaves, leader and election where N is the expected sequence number of the next message and Last is a copy of the last message from leader. Messages and views should also keep sequence number {msg, N, Msg}, {view, N, Peers, Group}. Also I have modified slave to discard the, messages it already received to handle duplicity. Also the view synchrony is created through gui module in which WxWidget library has been used.

3. Evaluations

- In this assignment when I run the test for gms1 module for peers in a network to coordinate through multicast using **test:more(3,gms1,1000).** I saw 3 worker processes changing colours at the same time. Though there was no information about the messages getting multicasted and crashes.
- This what is covered in gms2 module, on testing **test:more(3,gms2,1000)** I see worker processes like coordinating messages like before but also when the leader crashes message is displayed “Leader N: Crash”. Also the nodes elect the next leader and multicast continues.
- While testing gms3 module **test:more(3,gms2,1000)** I saw more clear picture of the multicast because messages which are multicasted among processes are also getting printed on the

screen. Last messages are preserved from the leader in case it dies. Also election function elects and multicasts the new elected leader to all the nodes.

4. Conclusion

In conclusion I would say doing this assignment made me learn practical implementation of atomic multicast. Also leader, slave functionality implementation was made easy in erlang like monitoring the leader crash is done using erlang monitor built in function. We can still say that some more features can be implemented to increase the reliability like message acknowledgements from peers to leader after receiving the message from it.