Groupy - a group membership service

Xin Ren

September 26, 2017

1 Introduction

In this assignment, a group membership service that provides atomic multicast is implemented. Several application layer processes have a coordinated state i.e. they should all perform the same sequence of state changes. A node that wishes to perform a state change must first multicast the change to the group so that all nodes can execute it. Since the multicast layer provides total order, all nodes will be synchronized.

2 Detailed Implementation

Following modules are implemented in this homework.

gms1.erl	The first version only handles starting of a single node and the adding of more nodes. When a node joins the group, every node in the group receives view update from the leader.
gms2.erl	The 2nd version which handles failure: leader crash. Every slave node starts to monitor leader node after it joins the group with Erlang build in support. After the leader crashes, the first node in the slave nodes list claims to be the leader, and the rest nodes start to communicate with new leader.
gms3.erl	The 3rd version that can handle missing messages caused by leader crash. Leader includes a sequence number in each message it sends to slaves, and slaves save the last message it receives from leader and its sequence number. The new leader sends the last message from the previous leader to all the slaves after taking over the leader role and before broadcasting the new view. Slaves that already received the last message from the previous leader before it crashes ignore the resending from the new leader.
gms4.erl	The 4th version which can handle the possibly lost messages.

3 Problems and solutions

1. When the leader node crashes, Erlang build in support will send each slave node a “DOWN” message and the new leader will broadcast the new view. It is possible that slaves receive the new view from new leader before the “DOWN” message for the previous leader.
   To handles this following is added to slave’s loop.

```
%view from new leader comes before down from old leader
{view, [NewLeader|Slaves2], Group2} ->
  erlang:monitor(process, NewLeader),
  Master ! {view, Group2},
  slave(Id, Master, NewLeader, Slaves2, Group2);
{'DOWN', _Ref, process, _OldLeader, _Reason} ->
  slave(Id, Master, Leader, Slaves, Group);
```

It means that slave node starts communicating to new leader and monitoring new leader after it receives new view from new leader, and "DOWN" message for the previous leader comes after the new view is ignored.

2. The speculations only guarantee that messages are delivered in FIFO order, not that they actually do arrive. So, it could happen that messages sent from leader to slaves get lost. (Messages from slave to leader that get lost can be resend by application layer.)
   I have two solutions for this problem.
   a. Once a slave receives a message from leader, it replies an acknowledgement and the leader waits for the acknowledgement after sending the message. If no acknowledgement received by the leader within a timer, it resends the message to the slave and waits for an acknowledgement again. We can set a limit on the time that the message is resent.
   b. Since there is a sequence number in the messages sent from leader to slaves and slaves save the last message from the leader with its sequence number, a slave knows if there has been messages that sent by the leader and it has not received yet when it receives a new message from leader by checking if the sequence number in the new message is exactly bigger than the saved sequence number by 1. When it is bigger by more than 1, the slave sends the saved sequence number to the leader and request resending of the messages after the sequence number. And of course, the leader saves the last N messages it sent to slaves as history.

   Comparison of two solutions:
   1) Both solutions can not completely solve the problem. Solution a can not handle the case that all the resent messages are also lost, while solution b can not handle the case that the number of the lost messages is bigger than the size of the history, which means that some lost messages are no longer in the history.
   2) Before the solutions are applied, the sending of messages to different slaves from leader almost happens at the same time. However, with solution a applied, the minimum time difference between sending a message to the first slave node and the last one is (K-1)*RTT where K is the number of slave node. This extends the period that slave nodes have inconsistent states even if the all the slave nodes receive messages in the same order and increases the time the system uses to handle one message. But this will not happen with solution b. With solution b, only the time spent on the resending of the lost messages will be added to the system, which is insignificant compare to the RTT of a message. A bit memory is used by the history of the leader in solution b.

Giving that the memory is not a bottleneck here, I think solution b is better than solution a as it has less impact on the performance, so solution b is implemented in this assignment.

4 Conclusions

Through this assignment, I have got the idea of how to implement a group membership service, how to handle failures and how to handle message loss.