Report 4: Groupy - a group membership service

Yining Hou

October 3, 2023

1 Introduction

In this report, I present a group membership service called Groupy in 4 different ways.

In this seminar, the main topic is about how to keep a coordinate state in distributed systems. A node that wishes to perform a state change must first multicast the change to the group, then all nodes perform the same sequence of state changes and will be synchronized.

2 Main problems and solutions

The implementation goes from the simplest to the more complicated ones. Failure detector, reliable multicast and lost message handler will be added step by step.

2.1 Architecture

Group Layer process:

- Leader:
  - Receive messages from Master and Slaves and multicast them to all Slaves;
  - Determine when a new Slave is included and deliver a new view to Slaves and Master.
- Slaves:
  - Receive messages from Master and requests from a new node, and forward them to the Leader;
  - Receive messages and views from the Leader and deliver them to its Master;
  - Monitor the Leader, if it crashes, elect a new leader.

Application Layer process:

- Master:
  - Create a group process;
  - Request to join a group;
  - Obtain the state and wait for this message to be delivered to itself.

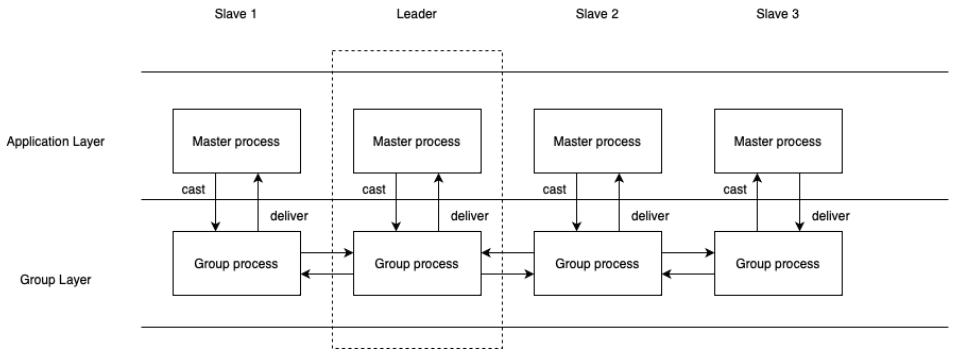


Figure 1: Groupy Architecture

2.2 No election

In the first implementation, I start a single node and add more nodes. Failures will not be handled: when the Leader crashes, the slaves will immediately lose connection with each other and they cannot keep consistant any longer.

2.3 Election with no sequence

Implement the election process when the leader crashes. Slaves can monitor the Leader and when they detect that a leader has died, they will move to an election state.

However, there is a possibility that during `bcast`, the leader crashes before it sends messages to all slaves. Some salves receive the messages but others do not. Then the nodes are in different states and become inconsistent.

```
leader 2 send msg:{change,13}
slave 3 recv msg:{change,13}
slave 4 recv msg:{change,13}
leader 2 send msg:{change,18}
leader 2: crash
```

```
slave 3 recv msg:{change,18}
leader 3 send msg:{change,12}
slave 4 recv msg:{change,12}
```

2.4 Election with sequence

To remedy the problem, we can keep a copy of the last message that Slaves have seen from the Leader. We assume that the leader sends messages to the peers in the order that they occur in the list of peers. If anyone receives a message, the first peer in the list receives the message. This means that only the next leader needs to resend the message.

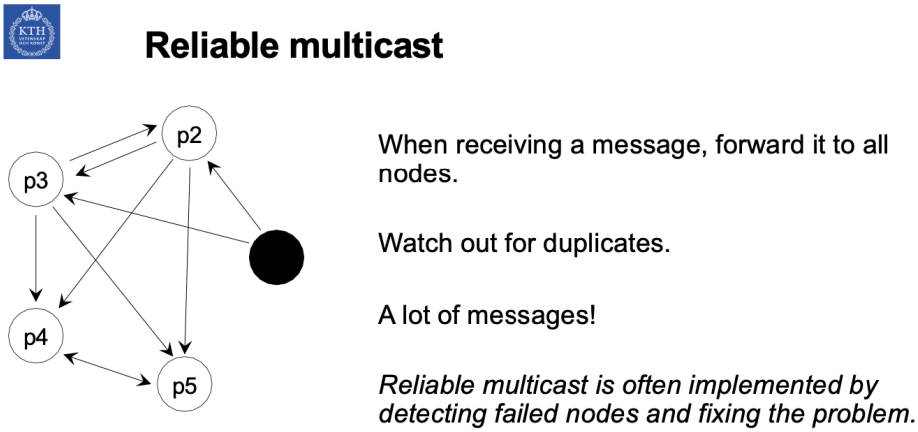
Add an additional parameter `N`, which is the expected sequence number of the next message, and `Last` is a copy of the last message. We discard messages that we have seen, i.e., messages with a sequence number less than `N`.

2.5 Bonus: Handle lost messages

We can't guarantee that messages are actually arrived. I add a `crashSend` method similar to `crash` which accidentally drops messages.

```
bcast(Id, Msg, Slaves) ->
    lists:foreach(fun(Slave) ->
        crashSend(Id, Slave, Msg),
        crash(Id) end, Slaves),
    acknowledgeMessage(Id, Msg, Slaves, length(Slaves))
```

2.5.1 Solution 1 - bcast



From what we learned from the class, I add `bcast` for every slave when it receives a new message. However, it also increases the chance of crash.

2.5.2 Solution 2 - tcp

Add another listener to keep track of the answers from Slaves. If answers are not all acknowledged within certain time, I assume that the message is lost and will resend the message. However, there will be small delay when recasting messages among all nodes.

```
acknowledgeMessage(_Id, Msg, _Slaves, 0) -> ok;
acknowledgeMessage(Id, Msg, Slaves, NumAcc) ->
  receive
    {acc, Msg} ->
      acknoledgeMessage(Id, Msg, Slaves, NumAcc - 1)
  after 1000 ->
    io:format("Message ~w recast~n", [Msg]),
    bcast(Id, Msg, Slaves)
  end.
```

Test `gms4` and print the logs.

```
11> test:more(3, gms4, 1000).
<0.168.0>
leader 1 dropped message: {msg,17,{change,14}}
Message {msg,17,{change,14}} recast
leader 1 dropped message: {msg,28,{change,17}}
Message {msg,28,{change,17}} recast
```

3 Conclusions

Groupy works well and certain methods have been implemented to guarantee reliable multicast.

The `worker` module has already been written and a lot of code are given from the document. However, it takes time to learn the responsibility of each member and it's not that easy to understand the message flow inside and outside different layers. The bonus part is also a bit hard to implement.