Groupy A Group Membership Service

jinjia zhang

September 19, 2023

1 Introduction

This assignment is to implement a group membership service. All nodes in the group have the same states. The basic idea is to have a leader node that takes care of all multicast messages and delivers a new view to the group when nodes come and go. Every application layer process has its group process and requests to join the group by providing the process identifier of its group process.

Four versions were implemented. In the first implementation, the basic requirement is that adding new nodes and broadcast is done but without handling failures. In the second version, handling failures is considered. When the leader dies, the new leader needs to be selected. But it may result in an out-of-sync state. The third version handles this problem. The last implementation handles that Erlang doesn't guarantee messages arrive.

Maintaining a consistent state in a distributed system is critical.

2 Main problems and solutions

Before I looked at the test and worker erlang file, I didn't understand why wee needed to distinguish between the application layer and the group layer when I looked through the assignment description. It is so confusing there are a few terminologies, such as master and group. And I didn't know which term represents the application process identifier or the group process identifier. But, after I ran the first version that was already provided in the project description, I had a clear idea of the architecture. Otherwise, it would be tough to implement the third and fourth version.

3 Evaluation

In the first implementation, the first worker is the leader and other workers are slaves that join the leader's group. After sleeping for a random time, every worker multicasts a change color message. The group state is always the same, which means that every GUI has the same color.

In the Second implementation, handling failure is considered. Every slave monitors the leader, if the leader is dead, all slaves move to the election process and the first slave on the slave list would become the new leader. However, there is one problem. When the leader is dead, the slaves' states are out of sync as the process of broadcast is not atomic, and the change messages cannot be forwarded to all other slaves. Figure 1 shows that the group state is not consistent.

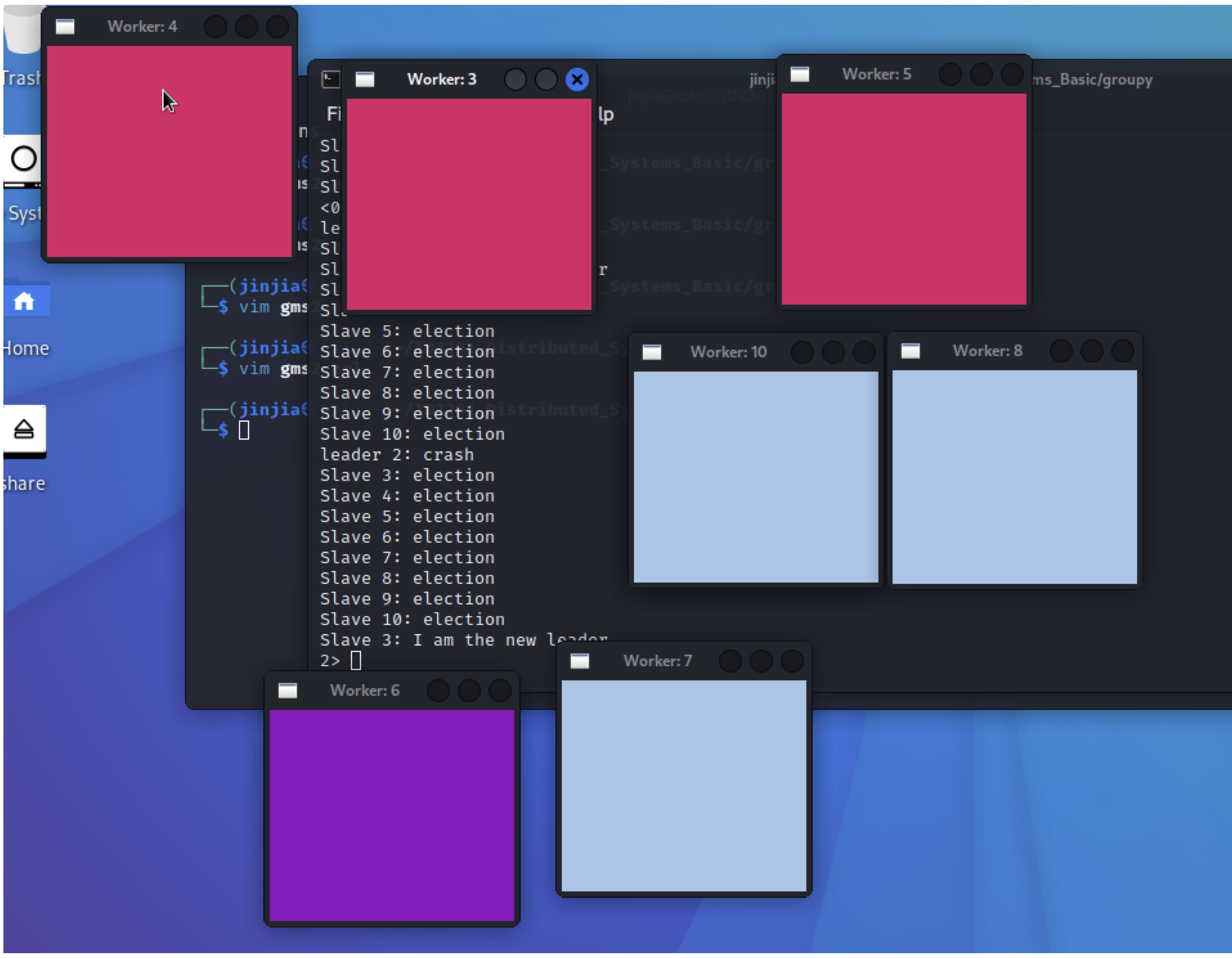
In the third implementation, resending the possible missing message is introduced with the help of restoring the last message and the next expected sequence number when the leader is dead, but it results in duplicated messages. It is safe to just ignore these messages. And we keep a group rolling by adding more nodes as existing nodes die.

The last problem is that Erlang doesn't guarantee messages actually do arrive. In the fourth implementation, the ack message is introduced. When a message is lost and it is resent again. In the foreach function, sequential executions of sending messages and receiving acks are utilized. This significantly slows down the process of broadcasting. We can measure the times of sending change messages in order to evaluate the performance.

On top of that, if one node is malicious and wants to send some bad info, we need a consensus mechanism that guarantees all the info in the group is correct.

4 Conclusions

In a distributed system, a consistent state during a group is crucial. During this assignment, I have learned that you shouldn't rely on some conditions that don't happen usually in a monolithic application, such as losing messages. Besides that, fault tolerance is one of the most important things when we design and implement a distributed system. The last thing I want to mention is that we need an easy way to monitor and debug distributed systems. Luckily, a lot of useful tools are widely adopted in the industry, such as Opentelemetry.



```
┌─(jinjia
└─$ vim gms

┌─(jinjia
└─$ vim gms

┌─(jinjia
└─$ ▯
```

```
Sl
Sl
Sl
<0
le
Sl
Sl
Sl
Sl
Slave 5: election
Slave 6: election
Slave 7: election
Slave 8: election
Slave 9: election
Slave 10: election
leader 2: crash
Slave 3: election
Slave 4: election
Slave 5: election
Slave 6: election
Slave 7: election
Slave 8: election
Slave 9: election
Slave 10: election
Slave 3: I am the new leader
2> ▯
```

Figure 1: gms2_crash