

# **ID2201 HT23 Distributed Systems, Basic Course (50929)**

## **Report Homework 4 - Groupy - a group membership service**

**by Eugen Lucchiari Hartz**

**October 04, 2023**

### **1 Introduction**

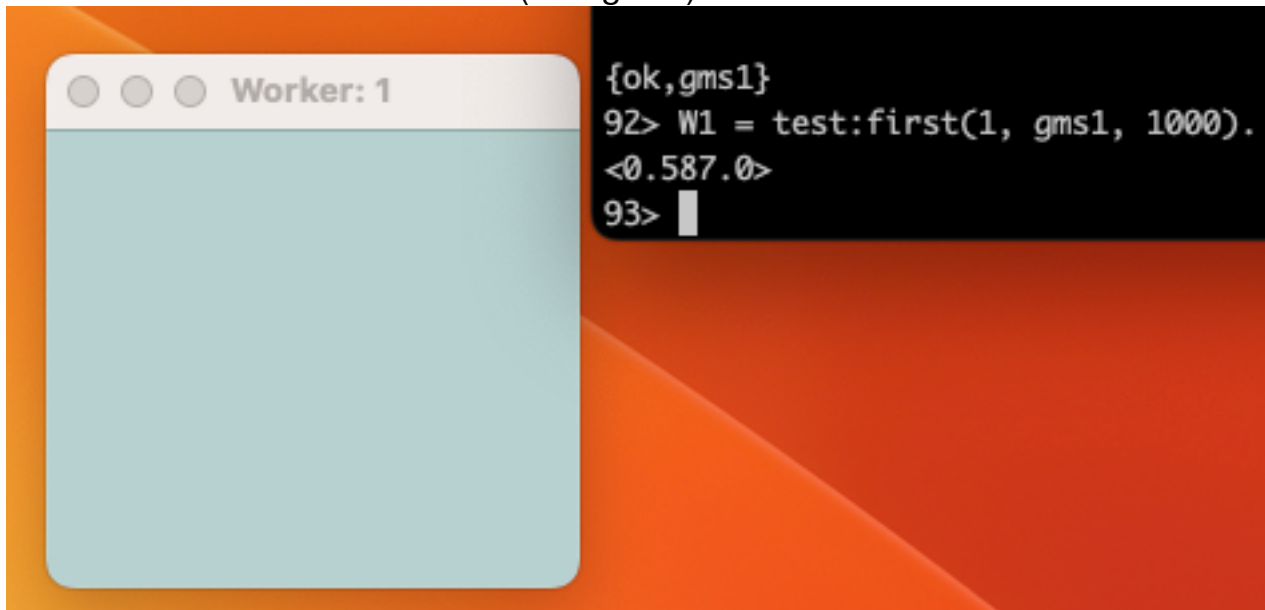
In this homework a group membership service that provides atomic multicast was developed using Erlang. This is about a node changing its state and multicasting this change so that the other nodes of the group can execute this change. The goal is that all nodes perform the same sequence of state changes. In this homework the state is a certain color, so the goal is that all nodes show the same color and accordingly are synchronised. The challenge is that this should still be the case even if possible crashes occur, such as a node that crashes. Such a group communication is an important part of distributed systems as it is essential for maintaining coordination and communication among nodes. An important aspect of this is the possibility to detect and handle node failures properly to increase robustness.

### **2 Main problems, solutions and evaluation**

#### **gms1**

The first implementation is still basic, so there is a leader and nodes can join the network. However there is no implemented failure handling so far. To do some experiments, we create a worker that uses a GUI to describe its state. Make sure that you can create a group and add some peers.

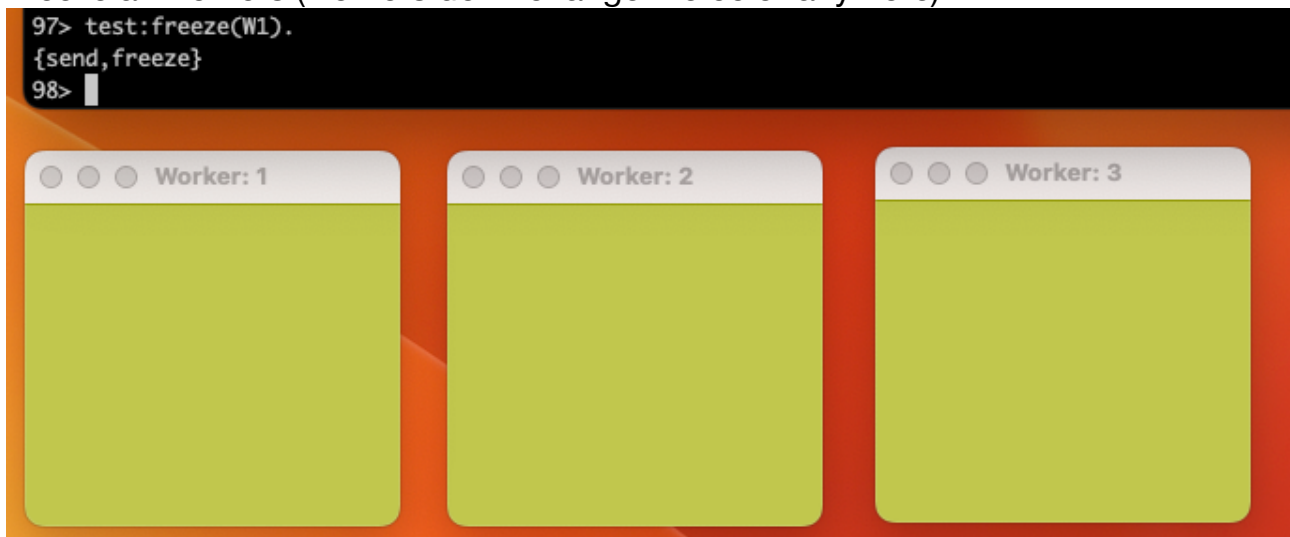
Create first worker and start GUI (with gms1)



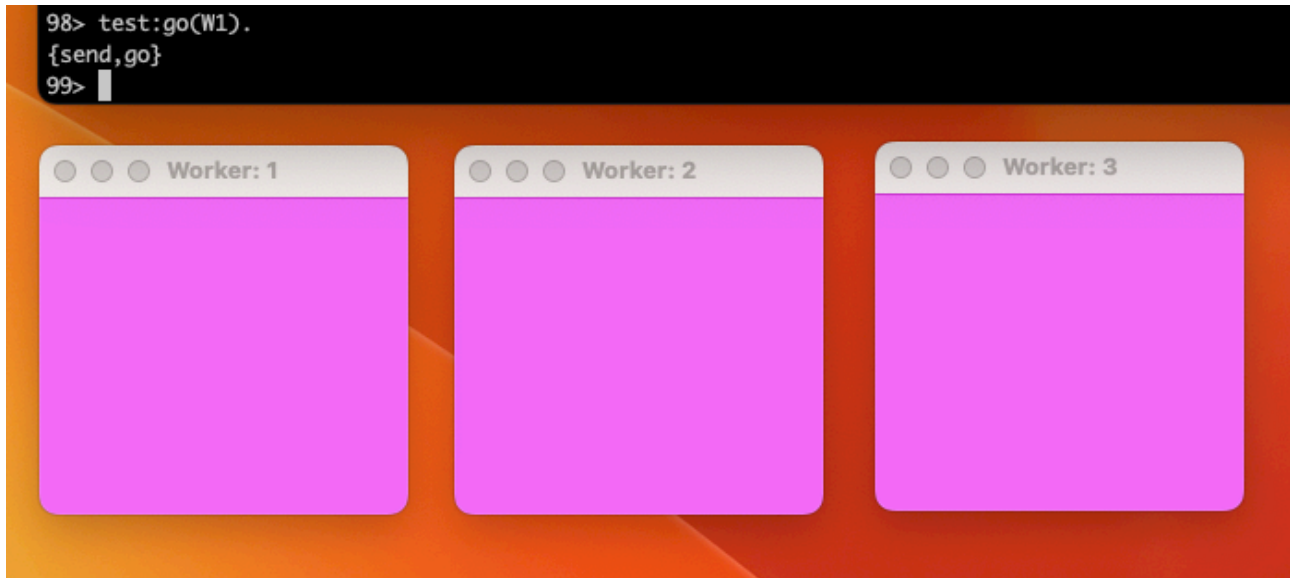
Add more workers to the group



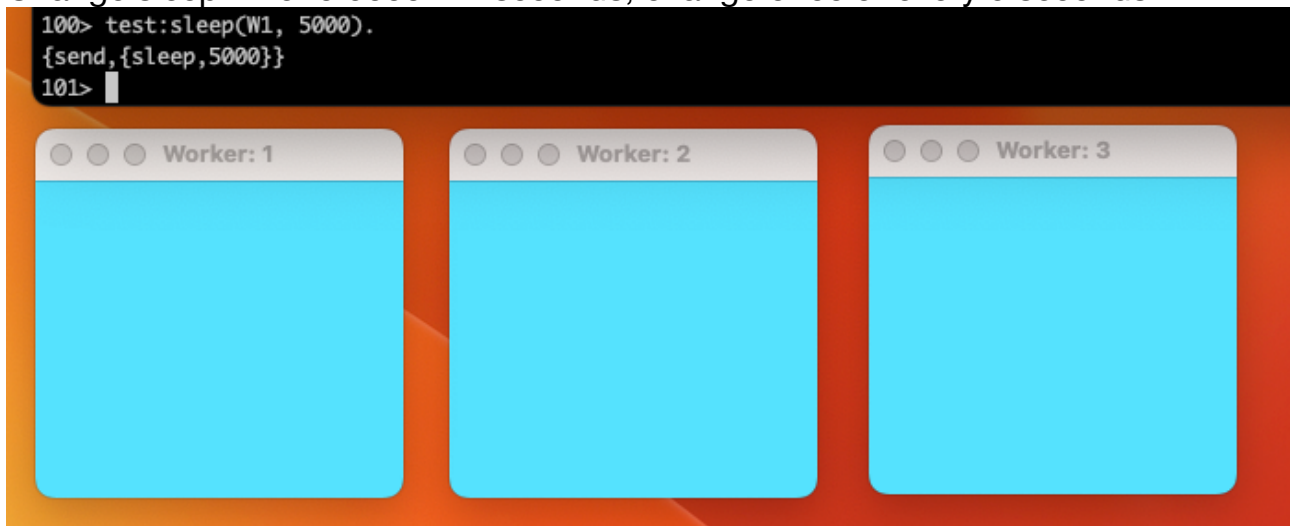
Freeze all workers (workers don't change the color anymore)



## Unfreeze workers



## Change sleep time to 5000 milliseconds, change of color every 5 seconds



## Stop a worker



Check what happens if a node that is not the leader dies



If a node that is not the leader dies the group members are still in sync, so the colours kept changing.

However, as gms1 doesn't handle failures, when the leader dies everything freezes.

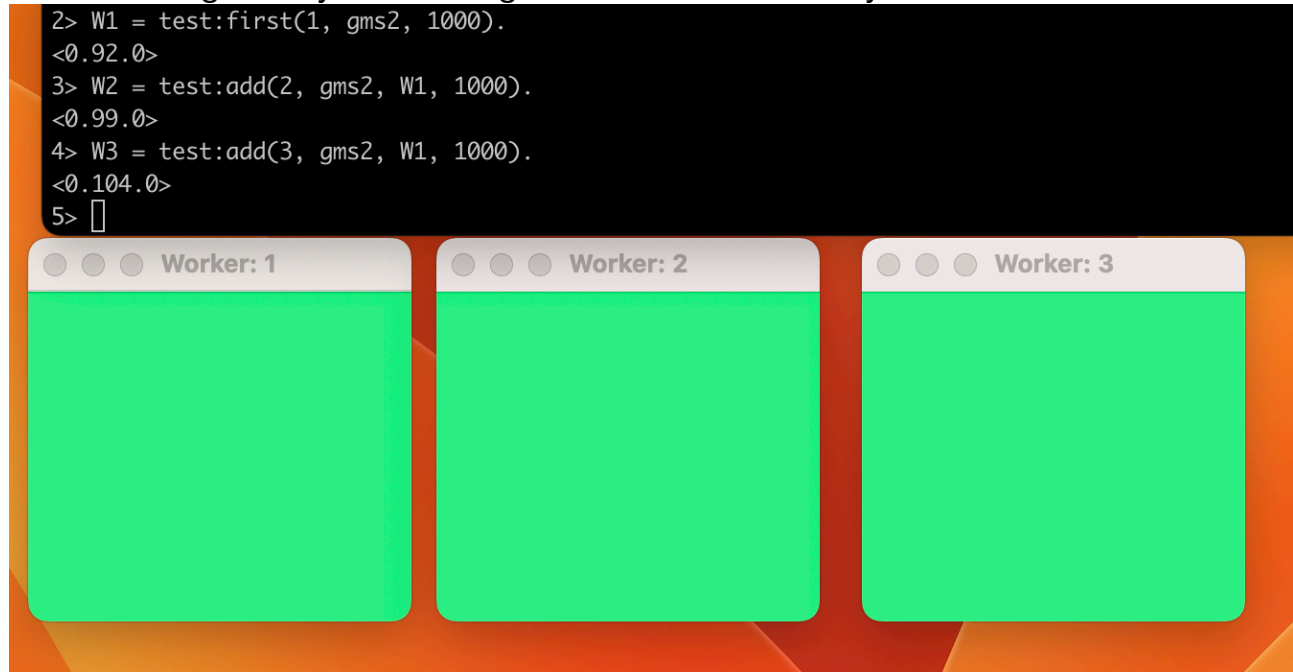


This is because they try to send changes to the leader to multicast them but when the leader is dead no one broadcasts the message to the group, so the colors stop changing.

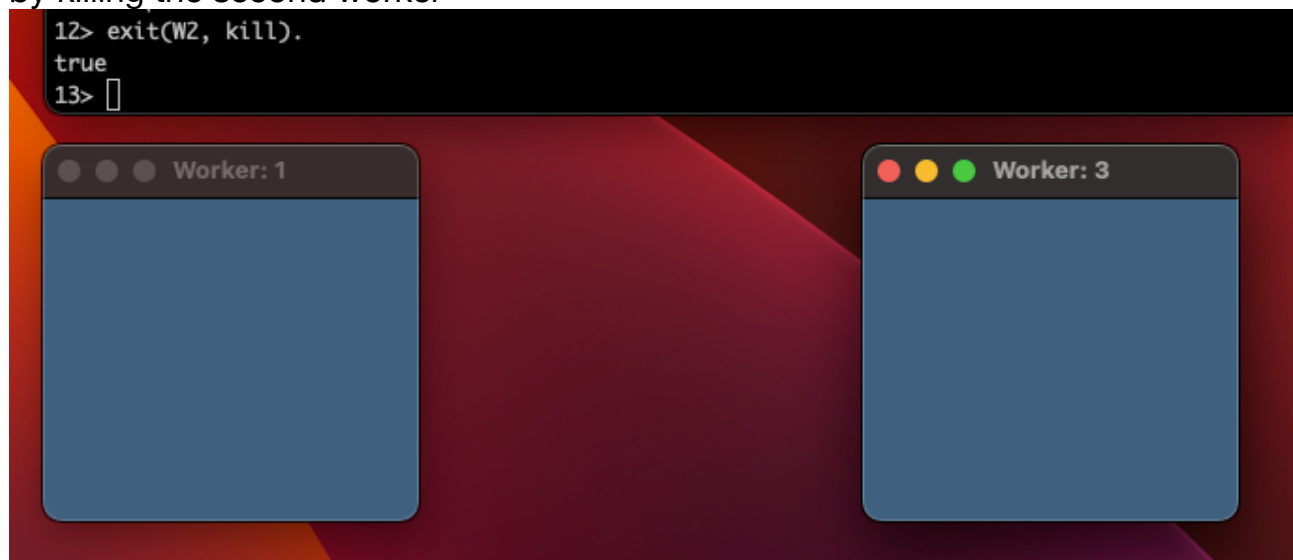
## gms2

The implementation of gms2 includes handling of crashes. When the leader dies another leader gets elected. Accordingly, we can now add new nodes to the system which should survive even if nodes crash.

Start test of gms2 by first adding three workers to the system



Now simulating node crashes to observe how the system reacts then, for example by killing the second worker



We can see that if a node that is not the leader gets killed the remaining nodes continue to work, so the system continues functioning without interruption.

When we kill the leader manually, the system now still continues working, so it continued to be consistent and the colors keep changing.

```
2> W1 = test:first(1, gms2, 1000).  
<0.92.0>  
3> W2 = test:add(2, gms2, W1, 1000).  
<0.99.0>  
4> W3 = test:add(3, gms2, W1, 1000).  
<0.104.0>  
5> exit(W1, kill).  
true  
6> []
```

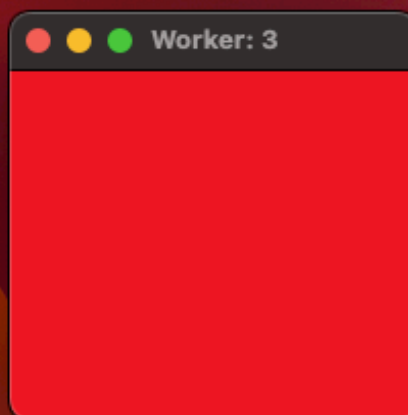


Now test with simulated random crashes during broadcasting.

First we can see that if the leader crashes its death gets detected and another leader gets elected.

Furthermore, we can see that if we add this random crash the leader dies before completing multicasting which leads to an inconsistent network.

```
{ok,gms2}  
2> W1 = test:first(1, gms2, 1000).  
<0.92.0>  
3> W2 = test:add(2, gms2, W1, 1000).  
<0.99.0>  
4> W3 = test:add(3, gms2, W1, 1000).  
<0.104.0>  
leader 1: crash  
5> █
```

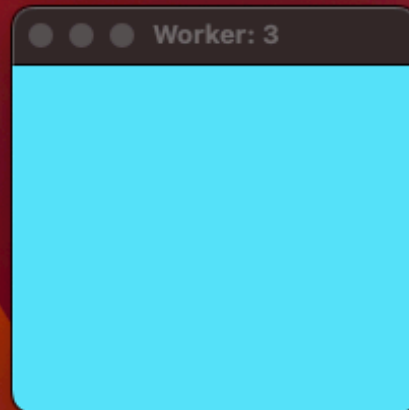


```
{ok,gms2}  
2> W1 = test:first(1, gms2, 1000).  
<0.92.0>  
3> W2 = test:add(2, gms2, W1, 1000).  
<0.99.0>  
4> W3 = test:add(3, gms2, W1, 1000).  
<0.104.0>  
leader 1: crash  
leader 2: crash  
5> █
```

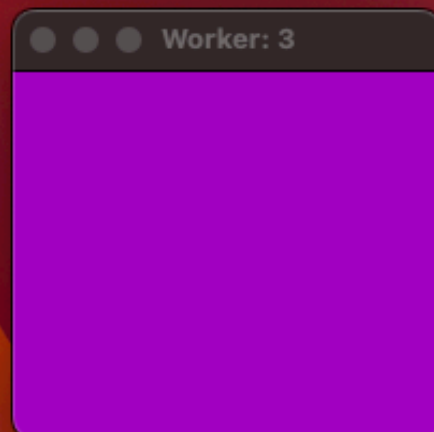




```
{ok,gms2}  
2> W1 = test:first(1, gms2, 1000).  
<0.92.0>  
3> W2 = test:add(2, gms2, W1, 1000).  
<0.99.0>  
4> W3 = test:add(3, gms2, W1, 1000).  
<0.104.0>  
leader 1: crash  
5> 
```



```
{ok,gms2}  
2> W1 = test:first(1, gms2, 1000).  
<0.92.0>  
3> W2 = test:add(2, gms2, W1, 1000).  
<0.99.0>  
4> W3 = test:add(3, gms2, W1, 1000).  
<0.104.0>  
leader 1: crash  
leader 2: crash  
5> 
```

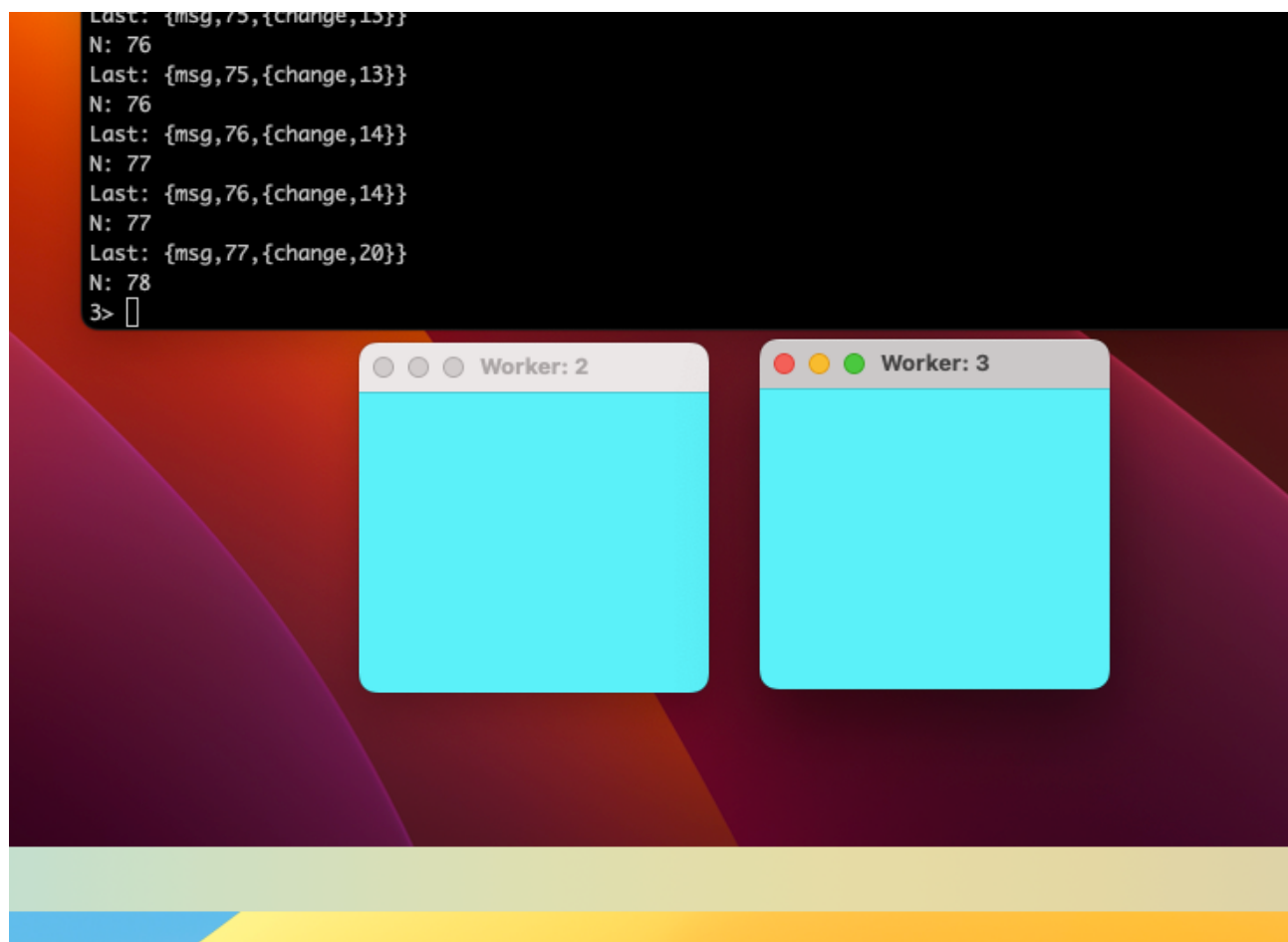




## gms3

Now in the third implementation we now handle the issue when a leader dies before completing multicasting to everyone in the group. A solution for that is that the newly elected leader resends the last message in case the previous didn't manage to send it.

However, then we have the problem that might receive duplicate messages, one from the old leader who just died and one from the new elected leader. To also handle this problem each message is tagged with a sequence number. After a leader multicasts or after a slave receives a message this sequence number should be incremented by one. Therefore, when a slave receives a message it first checks the sequence number. If the sequence number of the received message is less than the one that is expected that indicates that the message has already been sent, so the slave already saw the message before. Accordingly, this message can be ignored and discarded.



```
N: 39
N: 39
Last: {msg,38,{change,7}}
leader 1: crash
Last: {msg,39,{change,5}}
N: 39
N: 40
Last: {msg,39,{change,5}}
Last message: {msg,39,{change,5}}
N: 40
The new leader is <0.96.0>
Last message: {msg,39,{change,5}}
Last: {msg,39,{change,5}}
The new leader is: <0.96.0>
N: 40
N2: 39
Last: {msg,39,{change,5}}
N: 40
```

```
Last: {msg,219,{change,17}}
N: 220
Last: {msg,219,{change,17}}
leader 2: crash
N: 220
Last: {msg,220,{change,14}}
N: 221
Last message: {msg,220,{change,14}}
The new leader is: <0.97.0>
```


```
Last: {msg,220,{change,14}}
N: 221
Last message: {msg,220,{change,14}}
The new leader is: <0.97.0>
3> 
```



## Bonus

To handle the possibility of lost messages we can implement an acknowledgment mechanism. This can be achieved by a unique acknowledgment reference which is included to each message that gets sent from the leader to a slave. The slave waits for an acknowledgment from the leader within a specified timeout. If the acknowledgment is not received within the timeout period, the slave assumes the message may be lost and requests a retransmission.

```
N: 14
1: No acknowledgment received from <0.97.0>
2: No acknowledgment received from <0.95.0>
Last: {msg,13,{state,#Ref<0.2080029249.1605894145.163894>},{0,0,0}}}
N: 14
3> []
```



```
gms 2: Acknowledgment received
Last: {msg,23,{change,12}}
```

```
gms 2: Acknowledgment received
Last: {msg,27,{change,7}}
N: 28
1: No acknowledgment received from <0.96.0>
2: No acknowledgment received from <0.95.0>
Last: {msg,27,{change,7}}
N: 28
```

Acknowledgment Mechanism in the code:

- The `wait_for_ack/2` function is responsible for sending an acknowledgment request to a specified node and waiting for the acknowledgment.
- It generates a unique reference `Ref` and sends a `{ack, Ref}` message to the specified node.
- It then waits for a response with a matching reference using a `receive` block. If an acknowledgment is received within the specified timeout, it returns `{ok, Ref}` otherwise, it returns an error.

How this impacts the performance:

- Improved reliability because it ensures that messages are more likely to reach their destination.
- Increased overhead because these potential retransmissions add some overhead to the system. However, in scenarios where reliability is essential this overhead must then be accepted
- It also has an impact on the latency because of the waiting period for acknowledgments and potential retransmissions which comes along with this acknowledgment mechanism.
- The network traffic can also get increased due to the potential retransmissions

## **4 Conclusion**

In summary, the assignment was very useful to discover how a group membership service with atomic multicast work. It was very interesting to implement such a service where nodes receive the same message at the same time. Furthermore, it was very interesting to do investigations and implementations of failure handling so that the system continues to work even if a node crashes which makes it robust and reliable.

Atomic multicast is an essential part of distributed systems as it ensures that a group of computers receives the same message at the same time. If we think for example of an online game, where you do for example a movement, it is important that all players get this info, so this message, at the same time to ensure a fair gameplay. Accordingly, atomic multicast is crucial in distributed systems.