HW4 Report, Groupy: A Group Membership Service

David Fischer

September 30, 2025

1 Introduction

This assignments goal was the implementation of *Groupy* a group membership service supporting reliable multicast, leader election when an existing leader fails, message sequencing, and message acknowledgment. Each member of the group, referred to as workers, is a multi-layered process containing the GUI window and gms process, referred to as nodes, either being a leader or worker. The processes cycle through a set of colors maintained as the state by the worker, updated only through shifting the red color value, making order a critical element of the system. The exercise as a whole covers many core principles of distributed systems such as failure detection, totally ordered multicast, and reliable delivery.

2 Implementation

2.1 Basic Multicast

The initial implementation features basic multitasking, but still lacks failure detection and reconciliation. A *correct* node in this case will only be one correctly receiving all messages in order, adhering to the protocol, and not crashing.

2.2 Leader Election

The second implementation features a mechanism to elect a new leader should the initial leader crash, which is set up happen randomly on a parameterized frequency when sending messages. Each node participates in the reelection by going through the list of its neighbors, each selecting the first node in the list as the new leader, including the first node which is then promoted by entering the `leader/4` function in the process. While the system now recovers from the leader crashing, it can still frequently go out of sync. This happens when a leader crashes in the middle of broadcasting a

set of messages, meaning only a part of the nodes in the group have received the update and are therefore still in sync.

2.3 Reliable Multicast and Message Sequencing

The third implementation was updated with a counter to give the sent messages a sequence, being incremented with each new message. To enable reliable multicasting and prevent the previously mentioned sync issue, each slave node also keeps a copy of the last received message. The reelection process was adjusted to re-multicast the last message to ensure the group is kept in sync. Since no record is kept of who already received this message, the sequence number is used to make sure the message is *new* in each slave node.

2.4 Message Acknowledgment

The fourth and final implementation provides handing for possibly lost messages through message acknowledgment. To achieve this, the leader needs to receive *ack* messages including the sequence number for each message it sends out. The acknowledgment state is tracked through a map containing the sequence numbers as keys and a map including the original message, a list of nodes it hasn't received acknowledgments from, and a map to track the number of remaining attempts for each nodes as values. The slave nodes were adjusted to randomly ignore received messages to simulate them being lost and acknowledge messages received, even with old sequence numbers to prevent unnecessary retransmission on leader reelection. After a parameterizable delay, the leader iterates through the acknowledgment state and retransmits messages to the nodes that haven't acknowledged.

This introduces a considerable amount of overhead primarily by introducing a new variable to process and keep track of on each iteration of the new `leader/6` function. To make sure the state doesn't grow in size infinitely, it is continually cleaned to remove fully acknowledged messages and messages where the slave didn't respond within the set retry amount.

3 Main problems and solutions

3.1 Given Modules

As most of the code including the entirety of the worker was given without needing changes the enable functionality, understanding the entire architecture presented a problem at first. This was resolved through close study of the assignment text as well as the given modules and the sequential implementation of the distributed system principles in each generation of *gms*.

3.2 Incrementing

The biggest logic issue encountered during development was on `gms4`, with slaves going out of sync after ignoring a message even though they received a retransmit. After thoroughly examining the retransmission mechanism and acknowledgment state data structure, the issue was traced to the slave process incrementing it's own counter even upon ignored messages. This means acknowledgements were sent back correctly, but the worker didn't update the color as the message was considered *old* by the node.

3.3 What could possibly go wrong

While Erlang will eventually detect all crashes through the built in failure detector, but it can be wrong.[1] Networking and delays could cause a process to be incorrectly identified as crashed. This can be solved in one of two ways, ensuring safety through a more elaborate election process but introducing delays, or ensuring liveness to make sure the system keeps going at the cost of consistency through optimistic leaders.

In a situation where a slave, next in line for election, delivers a message to its worker and the leader, both of them crashing once the message has been delivered thus making the node incorrect. The new leader will have no record of this message and retransmits it's last message. While not affecting this system directly and being an unlikely case, this loss could be solved through persistent storage and message recovery on restarts.

4 Conclusions

Through the many generations of the underlying group membership service it became clear that more redundancy and reliability comes at a cost both in implementation complexity and operational efficiency.

The use of a tiling window manager granted a visual indication of the election algorithms implementation, with the leftmost (first) worker in the list always being the leader. Crashing causes the next window to take it's position, and therefore also leadership.

Future improvements could include extending the message acknowledgment mechanism to support leader elections and other messages passing through the system, native window arrangement functionality to improve insight and visualization, and a more graceful crash recovery to not fully loose crashed workers.

[1] https://www.erlang.org/doc/apps/erts/erlang#monitor/2