Report 4: Groupy - A Group Membership Service

Md Sakibul Islam

September 30, 2025

1 Introduction

This report documents the creation and performance testing of a group membership service called `Groupy`, implemented in Erlang. The project builds on ideas of reliable distributed coordination, where multiple nodes must maintain a consistent view of group membership and deliver messages in the same total order despite failures. Groupy is designed as a leader-based service that ensures atomic multicast with view synchrony. A single leader orders messages, installs new views when nodes join, and coordinates recovery when nodes fail. Applications interact through local worker processes, which hide the complexity of leader election, message sequencing, and synchronization, allowing the system to behave as if it were a single coherent process even under churn and faults.

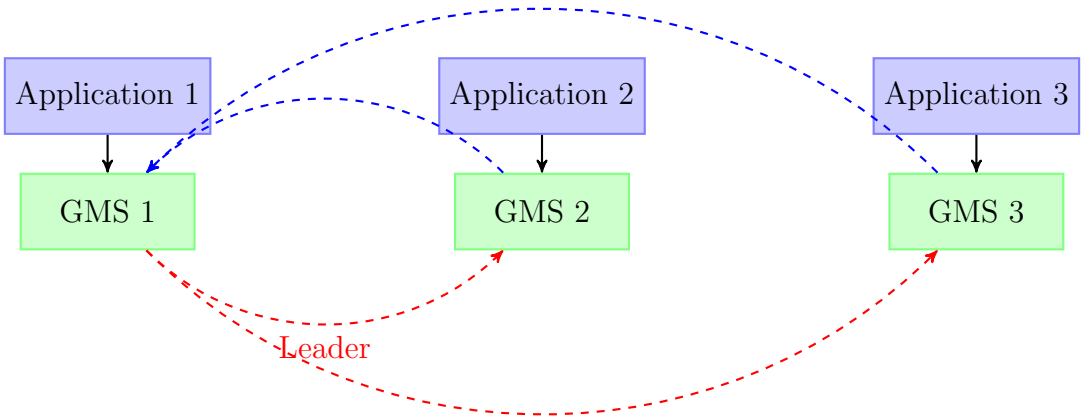

Figure 1: Leader-based communication in Groupy

2 Main problems and solutions

Process monitoring bugs. At first I tried `erlang:demonitor(process, Leader)` which crashed with `badarg`.

Fix: store the actual monitor reference from `erlang:monitor/2` and pass it to `demonitor/1`.
Hanging joins. New nodes sometimes waited forever if the leader had already died.
Fix: added a 5s timeout on `receive` to return an error back to the master.

Message duplication. During leader change, some messages were delivered twice.
Fix: added sequence numbers and guards (`I < W` ignore, `I == W` deliver).

Unclean worker shutdown. Test windows stayed open after runs.
Fix: wrote a cleanup function that first sends `stop`, then force-kills any remaining processes.

Stress instability. With many nodes, elections sometimes cascaded.
Fix: ensured every node consistently used the ordered slave list to pick the next leader, stabilizing view propagation.

3 Results

Functional tests. Three workers form a group, synchronize colors, and survive leader crashes. Recovery takes ≈2–3 seconds:

```
1> test_suite:run_all_tests().
=== Running Complete Test Suite ===
...
2. Testing gms2 (Failure Detection)
Node 3: Leader <0.104.0> is down, starting election
Node 2: I am the new leader
...
3. Testing gms3 (Reliable Multicast)
Node 2: I am the new leader, resending last message
gms3 test completed successfully.
=== All Tests Completed ===
ok
```

Stress tests. With 8–10 nodes and repeated random failures, groups stayed alive for many rounds:

```
8> gms_stress:run_gms2(8, 10).
=== gms2 stress: N=8 rounds=10 ===
Round 3: workers=8
KILL <0.324.0>
Node 3: Electing <0.325.0> as new leader
...
Round 6: workers=8
Node 5: I am the new leader
ok
```

Performance. gms3 adds only a sequence counter and replay buffer. Elections require one extra view multicast. Message order was preserved with no losses:

```
slave 2: gap on msg, expect=32 got=34 -> NACK 32..33
Node 2: I am the new leader, resending last message
```

These outputs confirm reliable delivery, consistent elections, and resilience under churn.

4 Bonus

I implemented `gms4`, which extends reliable multicast with loss recovery. Slaves monitor sequence numbers and send NACKs when they detect gaps. The leader maintains a bounded history buffer and retransmits requested messages, ensuring no loss even under lossy conditions. If the leader fails, the new leader takes over with the last delivered message and continues serving NACKs.

Impact: This mechanism tolerates packet drops without breaking total order. The steady-state overhead is minimal, limited to keeping a small replay buffer, while recovery cost is proportional to the number of missing messages.

5 Conclusions

Groupy demonstrates a clear progression from simple group formation to robust fault-tolerant multicast. Starting with basic membership in `gms1`, the system was extended with failure detection and leader election in `gms2`, then improved with reliable message sequencing in `gms3`, and finally enhanced with NACK-based loss recovery in `gms4`. Across tests and stress experiments, the service maintained group consistency, recovered automatically from failures, and preserved total order delivery with minimal overhead.