A Modified Cristian-Style Group Membership Protocol for Distributed Drone Swarm Coordination

Implementation Report and Progress Summary

Project Documentation

December 8, 2025

Abstract

We describe the design and implementation of a modified Cristian-style neighborhood-surveillance group membership protocol tailored for a distributed drone swarm simulation. The baseline Cristian protocol assumes a synchronous system, fixed neighbor relationships, no re-joining, and crash-stop failures. To adapt this algorithm to a dynamic drone environment with omission faults, timing jitter, crash–recovery behavior, multiple groups, and hierarchical coordination tasks, we introduce a series of principled modifications. This document outlines the original protocol, the full list of enhancements, their motivations, design choices, implementation status, and recent bug fixes.

Contents

1 Introduction

Cristian's 1991 group membership protocol assumes a synchronous distributed system in which each process monitors its successor using periodic "present" messages. Failure is detected through timeout, and a reconfiguration occurs when a process is declared failed. Although robust and simple, the original protocol does not directly support realistic drone-swarm conditions such as dynamic connectivity, omission faults, recovery after crash, or multiple overlapping coordination groups.

The goal of this project is to implement a variant of Cristian's protocol that remains *readable, modular, and faithful* at the membership layer while supporting the complexities needed for distributed drone coordination. All modifications to the protocol are isolated to a small subsystem to ensure clarity of reasoning and facilitate experimentation.

2 Baseline Cristian Algorithm

Cristian's original algorithm provides:

- A single group view shared by all correct processes.
- A fixed logical ring topology for failure detection.
- Unidirectional heartbeats (each process monitors only its successor).
- A synchronous timing assumption with a known message delay bound.
- A crash-stop failure model (failed processes do not recover).
- Reconfiguration via atomic broadcast of the new group view.

Let Δ denote the known upper bound on message delay. A single missed heartbeat from the successor implies failure, triggering a new membership view and a new group identifier.

3 Design Goals for Drone-Swarm Membership

A drone swarm differs substantially from Cristian's assumed environment:

- Connectivity between drones changes continuously.
- Message omissions and timing jitter are common.
- Nodes may crash temporarily and later recover.
- Multiple coordination groups may coexist.
- Atomic broadcasts may become expensive under network congestion.
- A hierarchical membership structure may be desirable.

Our design aims to satisfy the following:

G1: Preserve the conceptual clarity of Cristian's membership logic.

G2: Support realistic failures without altering the core protocol semantics.

G3: Cleanly separate physical drone state from logical membership state.

G4: Enable the introduction of SUSPECT states for omission tolerance.

G5: Allow nodes to rejoin after recovery with cooldown mechanisms.

G6: Support multiple independent membership groups and subgroups.

G7: Support optional group-to-group coordination (future work).

G8: Maintain performance under load via reduced broadcast frequency.

4 Internal vs. External State

We explicitly separate internal drone state from membership-visible state.

4.1 Internal Drone State (Not Visible to the Protocol)

RUNNING
Drone is operating normally.

TEMPORARILY_CRASHED
Drone is unresponsive but scheduled to recover.

PERMANENTLY_DEAD
Drone will never recover.

Each drone stores an internal time `recoveryTime` indicating when it will return from a temporary crash. This information is *not visible* to the membership protocol.

4.2 Public Drone Status (Visible to Membership Layer)

UP Drone is responsive (based on observed messages).

DOWN
Drone is unresponsive.

The membership layer treats all unresponsive drones identically, regardless of whether the underlying cause is temporary or permanent.

5 Membership Status and Failure Detection

Membership state is independent of physical drone state.

5.1 MembershipStatus

MEMBER
Drone is in the installed group view.

SUSPECT
Drone has missed one or more heartbeats but has not been removed from the group.

NOT_MEMBER
Drone is excluded from the current group view.

5.2 k-Timeout Failure Detector

Instead of declaring failure after a single timeout, we use a k-timeout failure detector:

$$\text{missed}(i) = \begin{cases} 0 & \Rightarrow \text{MEMBER}, \\ 1 & \Rightarrow \text{SUSPECT}, \\ k & \Rightarrow \text{NOT\_MEMBER (trigger reconfiguration)}. \end{cases} \tag{1}$$

This modification allows tolerance to omission and delay faults, which are expected in drone communication.

6 Protocol Extensions

We summarize all algorithmic modifications relative to Cristian's design.

6.1 Bidirectional and Multi-Neighborhood Surveillance

Instead of a single successor, each drone monitors both predecessor and successor or, more generally, all neighbors within a communication radius. This supports dynamic topologies.

6.2 SUSPECT State

The addition of SUSPECT enables resilience to omissions and jitter without frequent group reconfigurations.

6.3 Crash–Recovery Semantics

Drones may recover from crashes. Upon recovery, the drone becomes NOT_MEMBER and may issue a `JOIN_REQUEST` after a cooldown period r seconds.

6.4 Join Cooldown

To avoid rejoin storms caused by unstable drones, each drone enforces a cooldown interval before issuing membership join requests.

6.5 Group-Initiated Invites

Groups may send `INVITE` messages to drones that appear to be operational but unaligned with any membership view. A drone may respond with a `JOIN_REQUEST`.

6.6 Multiple Membership Groups

A drone can participate in multiple independent groups, each with its own:

- membership view,
- group epoch,
- surveillance relationships.

This enables hierarchical or role-based coordination.

6.7 Atomic Broadcast Cost Modeling

Reconfigurations incur simulated latency cost that increases with network load. This discourages overly frequent reconfigurations and motivates the use of SUSPECT state and multi-level group structures.

7 State Machine Summary

State Transitions:

$$\begin{aligned}
\text{MEMBER} &\xrightarrow{\text{missed}=1} \text{SUSPECT},\\
\text{SUSPECT} &\xrightarrow{\text{hb received}} \text{MEMBER},\\
\text{SUSPECT} &\xrightarrow{\text{missed}=k} \text{NOT_MEMBER},\\
\text{DOWN} &\xrightarrow{\text{recovery}} \text{NOT_MEMBER},\\
\text{NOT_MEMBER} &\xrightarrow{\text{JOIN_REQUEST}} \text{MEMBER}.
\end{aligned}$$

8 Code Organization

To maintain clarity, all membership logic resides in:

- `membership/MembershipState.h`
- `membership/MembershipState.cpp`
- `membership/MemberInfo.h`
- `membership/Types.h`

Simulation-layer concerns such as crash modeling, recovery distributions, and physical drone movement reside in:

- `core/Drone.h`, `core/Drone.cpp`
- `core/Simulation.h`, `core/Simulation.cpp`

Network behavior (delay, omission, overload) is implemented in:

- `network/NetworkSimulator.h`, `network/NetworkSimulator.cpp`
- `network/Message.h`

Visualization (heartbeats, membership coloring, packet animations) resides in:

- `visualization/PolyscopeRenderer.h`, `visualization/PolyscopeRenderer.cpp`

This separation ensures that the membership protocol remains simple and readable.

9 Three-Phase Reconfiguration Protocol

The implemented reconfiguration protocol follows a three-phase pattern where **members are discovered through ACKs**, not proposed upfront in the INIT. This design allows nodes to join a group dynamically without requiring the initiator to have prior knowledge of all participants.

9.1 Protocol Overview

1. **INIT Phase:** An initiator broadcasts a reconfiguration request
2. **ACK Phase:** All participating nodes respond with ACKs containing their positions
3. **COMMIT Phase:** The initiator finalizes membership based on received ACKs

9.2 Phase 1: INIT (Initiation)

When a node detects a failure or needs to form a group:

1. Generate a unique `reconfigId` based on timestamp (milliseconds)
2. Broadcast `RECONFIG_INIT` containing **only**:
   - `reconfigId` (4 bytes, 32-bit)
   - `initiatorId` (2 bytes, 16-bit)
3. Transition to `inReconfig` state
4. Immediately send own ACK (initiator participates too)

Key Design Choice: The INIT does *not* contain a proposed member list. Members are discovered through the ACK phase.

9.3 Phase 2: ACK (Member Discovery)

Upon receiving an INIT, each node:

1. Apply tie-breaking for concurrent INITs:
   - Higher `reconfigId` wins
   - If equal, **lower** `initiatorId` wins (first-come-first-served)
2. Enter `inReconfig` state tracking the accepted INIT
3. Broadcast `RECONFIG_ACK` containing:
   - `reconfigId` (4 bytes)
   - `senderId` (2 bytes)
   - `position` (12 bytes: 3 floats for x, y, z)

All nodes (including non-initiators) collect ACKs to learn about other participants and their positions. This enables distributed neighbor determination.

9.4 Phase 3: COMMIT (Neighbor Resolution and Finalization)

After waiting $2\Delta_2$ (stabilizing window), the initiator:

1. Collects all ACKs received for the pending `reconfigId`
2. Determines final membership: all nodes that sent ACKs
3. Optionally prunes members based on formation requirements
4. Broadcasts `RECONFIG_COMMIT` containing:
    - `reconfigId` (4 bytes)
    - `initiatorId` (2 bytes)
    - `numMembers` (2 bytes)
    - For each member: `id` (2 bytes) + `position` (12 bytes)

Upon receiving COMMIT, all nodes:

1. Install final membership with positions from the COMMIT payload
2. Set `groupId = reconfigId`
3. Update neighbor relationships using the received positions
4. Exit `inReconfig` state

9.5 Tie-Breaking Logic

Concurrent INITs are resolved as follows:

```
if (inReconfig) {
    // Higher reconfigId wins
    if (recId < pendingReconfigId) return; // reject

    // Same reconfigId: LOWER initiatorId wins
    if (recId == pendingReconfigId && initId >= initiatorId) return;

    // This INIT is better - switch to it
}
```

The tie-breaking ensures that when multiple nodes initiate simultaneously (with similar timestamps), the node with the *lowest* ID becomes the initiator, providing deterministic convergence.

9.6 Message Wire Formats

INIT:

```
[reconfigId: 4 bytes][initiatorId: 2 bytes]
Total: 6 bytes
```

ACK:

```
[reconfigId: 4 bytes][senderId: 2 bytes][pos.x: 4 bytes][pos.y: 4 bytes][pos.z: 4 bytes]
Total: 18 bytes
```

COMMIT:

```
[reconfigId: 4 bytes][initiatorId: 2 bytes][numMembers: 2 bytes]
[member1_id: 2 bytes][member1_pos: 12 bytes]
[member2_id: 2 bytes][member2_pos: 12 bytes]
...
Total: 8 + 14 * numMembers bytes
```

9.7 Timing Parameters

- Δ_1 (`deltaSmall`): Maximum datagram delay = 0.12s
- Δ_2 (`deltaLarge`): Stabilizing window = 0.50s
- Heartbeat interval: 0.2s
- Timeout delta: 0.6s
- Reconfiguration minimum interval: 1.0–2.0s

9.8 Protocol Properties

1. **Discovery-Based Membership:** Members are not proposed; they are discovered through ACK responses
2. **Position Propagation:** Positions are exchanged during ACK phase, enabling neighbor determination before COMMIT
3. **Deterministic Tie-Breaking:** Lower initiator ID wins ties, ensuring all nodes converge to the same initiator
4. **Self-Inclusion:** Initiator sends its own ACK, ensuring it is included in the final membership

10 Test Infrastructure

The project includes a test harness in `tests/MembershipReconfigHarness.cpp` with two primary test cases:

10.1 Test 1: Single Reconfig

1. Initialize 8 drones with deterministic network (no loss, no latency)
2. Allow 5 seconds for heartbeat propagation
3. Kill drone 0 to trigger reconfiguration
4. Verify all UP drones reach consensus on same groupId
5. Verify crashed drone is excluded from final membership

10.2 Test 2: Two Reconfigs with Monotone GroupId

1. Initialize 8 drones, allow warmup
2. First crash: drone 0, verify consensus reached
3. Wait 5 seconds for stabilization
4. Second crash: highest-id surviving drone
5. Verify new consensus with strictly larger groupId

10.3 Assertions Verified

- **Consistency:** All UP drones agree on same groupId
- **Completeness:** All UP drones have identical member sets
- **Safety:** No crashed drones in final membership
- **Liveness:** Reconfiguration completes within bounded time
- **Monotonicity:** GroupId strictly increases across reconfigs

11 Protocol Properties and Invariants

11.1 Safety Properties

1. **No Byzantine Members:** Dead nodes never appear in final membership
2. **Consensus:** All UP nodes eventually agree on same groupId and membership
3. **Monotonic GroupId:** GroupId strictly increases across sequential reconfigs
4. **ACK Completeness:** All ACKs arriving before commitDueTime are included

11.2 Liveness Properties

1. **Eventual Consensus:** After failure detected, system reaches consensus in bounded time
2. **Bounded Reconfig:** COMMIT sent after $2\Delta_2$ stabilizing window
3. **Rate Limiting:** Minimum gap between consecutive reconfigs prevents thrashing

11.3 Timing Guarantees

Total reconfiguration time (worst case):

$$T_{reconfig} \approx T_{detect} + \Delta_1 + 2\Delta_2 + \Delta_1 \approx 0.8 + 0.12 + 1.0 + 0.12 \approx 2.04\text{s}$$

12 Future Work

1. **Full SUSPECT State Implementation:** Currently defined but not fully utilized in failure detection
2. **k-Timeout Failure Detector:** Implement configurable missed heartbeat threshold
3. **Multiple Groups:** Enable hierarchical coordination with subgroups
4. **JOIN/INVITE Protocol:** Formal rejoin mechanism after recovery
5. **Performance Optimization:** Reduce broadcast overhead under load
6. **Visualization:** Enhanced membership state visualization in renderer

13 Conclusion

We have presented a systematic modernization of Cristian's membership protocol to support noisy, dynamic drone-swarm environments. By cleanly separating physical, membership, and network layers while preserving the conceptual simplicity of the original algorithm, the resulting system provides a robust foundation for fault tolerance, hierarchical coordination, and future scalability.

Recent debugging efforts have resolved critical issues with startup group formation, ensuring that all nodes correctly discover each other and form a consistent group view. The three-phase INIT-ACK-COMMIT protocol now correctly handles concurrent initiation attempts through tie-breaking, and ACK messages properly carry position information for neighbor determination.

Hypothetical future work could include extension to support more advanced features such as the SUSPECT state, multiple coordination groups, and hierarchical membership structures.