Annales Univ. Sci. Budapest., Sect. Comp. 58 (2025) 231–250

ENERGY-EFFICIENT LAZY GROUP MEMBERSHIP PROTOCOL IMPLEMENTATION IN HASKELL

Jianhao Li and Viktória Zsók (Budapest, Hungary)

Communicated by Zoltán Horváth
(Received 3 June 2025; accepted 2 August 2025)

Abstract. In distributed systems, maintaining an up-to-date view of active nodes is essential for ensuring reliable communication, failure detection, and system reconfiguration. Traditional group membership protocols, such as SWIM [3], rely on periodic gossip exchanges, incurring communication and energy overhead. In this work, we implement a "lazy" group membership protocol in Haskell that eliminates periodic messages. We describe the complete implementation and introduce an energy-efficient benchmark. An experiment based on the new benchmark compares our lazy protocol to a standard SWIM implementation. Results show that, despite higher absolute power caused by the secure UDP library and the Haskell runtime, our lazy protocol achieves a lower normalized energy index than SWIM.

1. Introduction

Distributed systems are essential for a variety of modern applications, including cloud computing, the Internet of Things (IoT), and distributed databases. These systems typically comprise multiple nodes that must collaborate and communicate effectively. For such systems to operate reliably, they must maintain a consistent view of the active participants. All nodes should have the same understanding of who is part of the system at any given time. Group membership is crucial for tasks such as reliable communication, where a sender must know which nodes to address, and failure detection, where nodes need to agree when another node has failed or left the group.

Key words and phrases: Distributed systems, functional programming group membership protocol, distributed algorithms.
2010 Mathematics Subject Classification: 68M14, 68N18, 68Q85, 68W15.
https://doi.org/10.71352/ac.58.020825
First published online: 8 August 2025

Group membership protocols ensure that all nodes in a distributed system agree on the set of active participants. These protocols ensure that all nodes in the system maintain an up-to-date list of group members, which is crucial for tasks such as reliable group communication and failure detection.

Classic membership protocols typically use periodic heartbeat or gossip messages. In these protocols, nodes periodically exchange messages with a few other nodes to propagate membership changes and detect failures. For example, SWIM [3] uses frequent gossip to communicate membership changes and detect failures. The term *gossip* refers to this process of passing information between nodes in a probabilistic manner. However, the periodic messages can cause unnecessary overhead, especially in systems with stable membership.

In contrast, the *lazy* membership protocol we proposed in this paper avoids periodic messages to reduce energy consumption. It only sends messages when a node joins or leaves the group. In quiet periods, there is no background communication. The main contributions of this paper are:

- We implement the lazy membership protocol in Haskell, utilizing the secure UDP (User Datagram Protocol) library for communication, and provide a clear description of its design. Haskell offers strong typing and concise concurrency abstractions, which help write clear, correct code.
- We design an energy-focused benchmark to evaluate membership protocols. This benchmark includes two modes: baseline message exchange (for pure communication costs) and full-protocol operation (with membership management logic).
- We conduct an evaluation comparing the lazy protocol to SWIM in an energy-focused benchmark. We test the protocols on separate machines to ensure accurate results. We measure both the base energy consumption of sending raw messages and the energy consumption of the protocol.

The paper is organized as follows: Section 2 provides background on group membership protocols and compares our approach with the SWIM protocol. Section 3 describes the implementation details of our lazy protocol, including message formats, data structures, and concurrency mechanisms. Section 4 illustrates the runtime behavior through example executions. Section 5 outlines our energy benchmark and how it simulates realistic workloads. Section 6 details the evaluation setup for both lazy and SWIM protocols. Section 7 presents and analyzes the experimental results. Finally, Section 8 concludes the paper and discusses possible directions for future work.

2. Background and comparison with SWIM

Group membership is critical in Wireless Sensor Networks (WSNs), as it enables nodes to coordinate tasks like group communication and group mem-

bership verification. For instance, the causal-order protocol combines gossip protocols and virtual synchronous group membership to ensure causal ordering in WSNs [6].

In the WSN membership authentication and group key establishment protocol, each member knows exactly the memberships of users participating in the secure group communication after membership authentication [2].

For the flat group key management schemas, all sensors have the same capabilities, collect data, and forward the data to other sensors in the network [7].

The membership authentication and key management scheme [7] enhanced the key update mechanism of group key management schemes by requiring all sensors in the WSN to broadcast heartbeat feedback to the control center periodically. If the control center does not receive the heartbeat feedback from any sensor within a reasonable time interval, it will notify all sensors to revoke the current session key.

Moreover, group membership plays an essential role in data center orchestration systems. For example, the Serf failure detection and orchestration tool [5] is based on the SWIM group membership protocol [3, 4]. SWIM uses periodic gossip messages to communicate membership changes and detect failures in a distributed environment.

However, to our knowledge, only a very few works explicitly optimize energy consumption in group membership. Instead, our lazy group membership protocol emphasizes sustainability. It eliminates periodic messages, sending updates only when there is a change in membership.

Compared to SWIM, our lazy protocol focuses on reducing the overhead of constant message exchanges. The SWIM protocol, while providing fast failure detection and scalability, sends periodic heartbeats even when there are no changes in group membership. In contrast, the lazy protocol operates only when necessary, significantly reducing unnecessary energy consumption.

3. Protocol implementation

The goal of our implementation is to create a lazy group membership protocol in Haskell. This protocol eliminates periodic ping messages and instead uses event-driven messages and timeouts to maintain group membership.

The protocol is implemented based on SecureUDP [1], which is a Haskell library that provides reliable UDP packet delivery through acknowledgments and retransmissions. In addition, several other libraries are imported to facilitate concurrency, data handling, and network communication.

The `GHC.Generics` library supports generic programming, enabling easier serialization and data manipulation.

`Data.Word` defines fixed-width integer types such as `Word64` and `Word8`, which are used for handling data and identifiers. `Data.Data` allows for working with data types in a generic way, enhancing flexibility and abstraction.

`Data.Set` implements sets, which are used to manage collections of unique elements, such as active group members.

`Data.Serialize` facilitates the serialization and deserialization of data for communication between nodes.

`Network.Socket` provides low-level network functionality for socket programming, enabling communication over UDP.

`Data.List.Split` offers utility functions to split lists, useful for processing received messages.

`Control.Monad` is used to control monadic flow, which is essential for concurrency and iterative operations.

`Control.Concurrent` provides concurrency support, including thread management and synchronization.

`Data.ByteString` is a more efficient representation of byte sequences, useful for network transmission.

`Control.Exception` helps handle exceptions and ensures proper resource management during concurrent operations.

Lastly, `Data.ByteString.Char8` facilitates working with `ByteString` in a character-based format, commonly used in network communication.

The following `Data structures and state` subsection explains how each node maintains the membership list and protocol state. The `Join procedure` subsection details the steps involved when a new node joins the group. The `Leave and failure handling` section describes how nodes handle leaving the group or detecting failures. The `Concurrency` subsection discusses the use of concurrent threads for timeout management. Finally, the `Protocol messages` section outlines the various message types used in the protocol.

3.1. Data structures and state

Each node has a unique identifier (`NodeId`) consisting of its IP address and port, and maintains local configuration and state. The configuration (`NodeConfig`) includes the node's ID and a secure UDP channel for communication. Shared mutable state (`LocalInfo`) contains fields such as the current membership list, the node's protocol state, pending join/leave information, and timers. For example, a node state can be `Idle` (a member with no ongoing group membership changing event), `Joining` (in the process of joining), and `Introducer` (a node introducing new members).

The data and state definitions form the basis of the following implementation, and MVar-based updates ensure safe concurrent access.

3.2. Join procedure

Joining a group is an essential process that allows new nodes to obtain the current group view and join without compromising consistency.

The general procedure for a new node to join a group and to become a member of it consists of the following steps. A new node starts the joining

process by sending a `JoinRequest` to any known member and waits for either a `GroupStructure` reply or a timeout. The group member who received the `Join-Request` and became the `Introducer` is informing all the other group members consistently. The other group members who received information from the `Introducer` will update the local group list.

When a new node wishes to join, it starts in the `NotInGroup` state and sends a `JoinReq` to the designated introducer node. The introducer receives this request. In the `Idle` state, upon receiving `JoinReq`, the introducer adds the requester to a pending list (`toAdd`), updates a local join counter, and broadcasts an `Inform` message to all current group members (except those whose removal is pending). Then the introducer changes to `WaitingInformAck` state and waits for acknowledgments.

Upon receiving `Inform`, other group members send back an `InformAck` and change to the `Informed` state temporarily. If all active members acknowledge the `Inform` message, the introducer proceeds: it sends a final `Operation` message to existing members and a `GroupStruct` to the new member, thus completing the joining process.

Suppose acknowledgments are insufficient (e.g., less than half respond in time) and there is another `Inform` message in the mailbox. In that case, the introducer aborts the join by sending `Finish` to other group members and `JoinFail` to the joining node to cancel the attempt.

As illustrated in Listing 1, the `recvInformAck` function handles the processing of `InformAck` messages in the lazy group membership protocol. When a node receives an `InformAck` message, it checks whether the `informId` of the incoming message matches the node's `localInformId`. If the incoming `informId` matches `localInformId`, it logs and inserts the `sender` into `ack`. Otherwise, it re-enqueues. After updating the local information, it calls `recvAllInformAck`.

```
recvInformAck :: C.MVar MsgList -> Message -> NodeConfig ->
C.MVar LocalInfo -> IO ()
recvInformAck msgListMVar msg@(Informack { informId = inID,
sender = ackSender }) cfg localInfoMVar = do
  C.modifyMVar_ localInfoMVar $ \li ->
    case (inID == localInformId li) of
      True -> do
        putStrLn $ "[recvInformAck] executing"
        return li { ack = Set.insert ackSender (ack li) }
      False -> do
        reenqueue msg msgListMVar
        return li
  recvAllInformAck cfg localInfoMVar
recvInformAck _ _ _ _ = error "[recvInformAck] pattern match error"
```

Listing 1. The `recvInformAck` function processes incoming `InformAck` messages, updating the acknowledgment set or re-enqueuing unrecognized messages.

As detailed in Listing 2, the `recvOp` function handles the `Operation` message in the lazy group membership protocol. When the node is in the `Informed` state and the incoming `informId` matches the node's `localInformId`, it cancels any active timers and updates the group membership list. The new list is computed by taking the union of the existing group list with the new members and removing the nodes that are no longer part of the group. The function also updates the `deleteInform` list by removing the deleted nodes. A `Gsinfo` message is then sent to each newly added node. After the update, the node transitions to the `Idle` state, resets its `localInformId`, and logs the updated group membership. If the `informId` does not match or the node is not in the `Informed` state, the message is re-enqueued for later processing.

```
recvOp :: C.MVar MsgList -> Message -> NodeConfig ->
C.MVar LocalInfo -> IO ()
recvOp msgListMVar msg@(Operation { informId = opInID, newMembers = newMs,
removeMembers = remMs }) cfg localInfoMVar = do
  C.modifyMVar_ localInfoMVar $ \li ->
    case (nodeState li, opInID == localInformId li) of
      (Informed, True) -> do
        cancelTimer cfg
        let newGL = Set.difference (Set.union (groupList li) newMs) remMs
            newDI = Set.difference (deleteInform li) remMs
            gsInfoMsg = Ser.encode (Gsinfo { informId = opInID,
            sender = selfNodeID cfg })
            msgs = [(nodeIdToSockAddr e, gsInfoMsg) | e <- Set.toList newMs]
        _ <- Sec.sendMessages (secureUDP cfg) msgs
        putStrLn $ "[recvOp] executing, groupList" ++ show newGL
        return li { nodeState = Idle, localInformId = zeroInformID,
          groupList = newGL, deleteInform = newDI }
      _ -> do
        reenqueue msg msgListMVar
        return li
recvOp _ _ _ _ = error "[recvOp] pattern match error"
```

Listing 2. The `recvOp` function processes the `Operation` message to update the group membership and notify new members.

The join protocol utilizes an inform/ack mechanism to add new nodes without compromising group consistency, ensuring a safe and clear process.

3.3. Leave and failure handling

The general process of leave and failure handling has the following procedure. Each member maintains a small "deletion list" of nodes to remove, either because they have failed or have chosen to leave. When a member detects a failure or decides to leave, it adds the node ID to its deletion list and triggers an inform round. In this round, the member sends an `Inform` message to all group members except those on its deletion list, then waits for `InformAck`

replies within a timeout. If all alive members acknowledge, the informer becomes the `Introducer` and broadcasts the final update.

A leaving node can add itself to a removal set (`toDelete`) and broadcast an `Inform`, triggering the same acknowledgment and update procedure. Failure detection leads nodes to mark lost members in a pending removal set.

As shown in Listing 3, the `idleDetectFail` function is responsible for detecting node failures when the node is in the `Idle` state. It takes a set of `lostIds`, representing nodes that have failed or are unreachable, and checks whether any of these nodes are already marked for deletion in `deleteInform`. If there are any new lost nodes, they are added to the `deleteInform` set. This function ensures that the group membership list remains updated by tracking failed nodes. If no new failures are detected, the function does not make any changes to the state.

```
idleDetectFail :: C.MVar LocalInfo -> Set.Set NodeId -> IO ()
idleDetectFail localInfoMVar lostIds = C.modifyMVar_ localInfoMVar $ \li -> do
    let newLost = lostIds `Set.difference` deleteInform li
        newDeleteInform = Set.union (deleteInform li) lostIds
    case (nodeState li, Set.null(newLost)) of
      (Idle, False) -> do
        putStrLn $ "[idleDetectFail] executing"
        return li { deleteInform = newDeleteInform }
      _ -> return li
```

Listing 3. The `idleDetectFail` function detects node failures in the `Idle` state and updates the `deleteInform` list accordingly.

As demonstrated in Listing 4, the `introDetectFail` function is used to detect node failures when the node is in the `Introducer` state. It receives a set of `lostIds`, which represent the nodes that have failed or become unreachable. If any of the lost nodes are not already marked for deletion in `toDelete`, the function logs the failure and adds them to the `toDelete` set. This ensures that the node acting as an introducer keeps track of failed members and updates the group membership accordingly. If no new failures are detected, the state remains unchanged.

```
introDetectFail :: C.MVar LocalInfo -> Set.Set NodeId -> IO ()
introDetectFail localInfoMVar lostIds = C.modifyMVar_ localInfoMVar $ \li-> do
    let newLost = Set.difference lostIds (toDelete li)
    case (nodeState li, Set.null newLost) of
      (Introducer, False) -> do
        putStrLn "[introDetectFail] executing"
        return li { toDelete = Set.union (toDelete li) lostIds }
      _ -> return li
```

Listing 4. The `introDetectFail` function detects node failures in the `Introducer` state and updates the `toDelete` list accordingly.

As presented in Listing 5, the `recvFinish` function implements the `RecvFinish` action, which handles the reception of the `Finish` message in the lazy group membership protocol. When the node is in the `Informed` state and the incoming `informId` matches the node's `localInformId`, the function cancels any active timer, logs the action, and transitions the node back to the `Idle` state. It also resets the `localInformId` to signify the finish of the group membership update. If the conditions are not met, the message is re-enqueued for later processing.

```
recvFinish :: C.MVar MsgList -> Message -> NodeConfig ->
C.MVar LocalInfo -> IO ()
recvFinish msgListMVar msg@(Finish { informId = inID }) cfg localInfoMVar =
  C.modifyMVar_ localInfoMVar $ \li ->
    case (nodeState li, inID == localInformId li) of
      (Informed, True) -> do
        cancelTimer cfg
        putStrLn "[recvFinish] executing"
        return li { nodeState = Idle, localInformId = zeroInformID }
      _ -> do
        reenqueue msg msgListMVar
        return li
recvFinish _ _ _ _ = error "[recvFinish] pattern match error"
```

Listing 5. The `recvFinish` function processes the `Finish` message to finalize the group membership update and return to the `Idle` state.

The failure and leave events are handled via the deletion list and the inform phase, ensuring that all live members are aware of the removals.

3.4. Concurrency

The Haskell code utilizes concurrent threads for networking and timer operations. Each node spawns a receiver thread that listens on the secure UDP channel, deserializes incoming messages, and dispatches them to the corresponding handlers. Timers (using `forkIO` and delays) enforce timeouts. Shared state is protected by `MVars` (atomic mutable variables in Haskell) to ensure thread-safe updates of the membership list and state.

As shown in Listing 6, the timer management functions are responsible for handling the timeout mechanism in the protocol. The `cancelTimer` function stops any active timeout thread by checking the thread ID stored in `timerThread` and killing the thread if it exists. The `startTimer` function first cancels any existing timer, then forks a new thread that waits for a specified delay (in microseconds) before executing the provided `action`.

The thread ID is stored in `timerThread` to allow for proper management and cancellation. The `defaultTimeout` function defines the default timeout period as 16 times the message-fetch interval, ensuring that the protocol has a reasonable waiting period for each operation. These functions are crucial

for managing timeouts and ensuring that operations are executed within the specified time limits.

```
cancelTimer :: NodeConfig -> IO ()
cancelTimer cfg = mask_ $ do
  mOld <- C.modifyMVar (timerThread cfg) $ \old -> return (Nothing, old)
  case mOld of
    Just tid -> C.killThread tid
    Nothing -> return ()

startTimer :: Int -> IO () -> NodeConfig -> IO ()
startTimer delay action cfg = mask_ $ do
  cancelTimer cfg
  tid <- C.forkIO $ do
    C.threadDelay delay
    uninterruptibleMask_ action
  C.modifyMVar_ (timerThread cfg) $ \_ -> return (Just tid)

defaultTimeout :: NodeConfig -> Int
defaultTimeout cfg = 16 * msgFetchInterval cfg
```

Listing 6. Timer management functions for controlling the timeout mechanism in the protocol.

Concurrency in this implementation involves a receiver thread and timer threads, with all shared state stored in MVars, which ensures reliable timeouts and message handling.

3.5. Protocol messages

The protocol uses several message types (modeled as a Haskell data type `Message`), including `JoinReq` (join requests), `Inform` (introducer broadcasts that there are membership changes), `InformAck` (acknowledgments of informs), `GroupStruct` and `Operation` (final group update and data), `Finish` (cancel join), and `JoinFail` (inform the joining node that the joining is failed). Message handling functions dispatch on message type and current state to implement the protocol logic.

The `processGroupMessage` matches on the incoming `Message` constructor and calls the corresponding handler function: `recvInform`, `recvJoinRequest`, `forwardJoinReq`, `introRecvJoinRequest`, `recvInformAck`, `recvFinish`, `recvHandlerUpdate`, `recvGroupStruct`, `recvGsInfo`, `recvJoinFail`, `recvAllInformAck`, `updateOp`, `recvOp`.

Each handler implements a part of the protocol logic: Once the `Joinreq` is received, if the node is `Idle`, it changes to `introCounter`, assigns a new `localInformId`, adds the requester to `toAdd`, broadcasts an `Inform` to current members, starts a join timeout, and sets the state to `WaitingInformAck`. When receiving a `GroupStruct`, the joining node cancels the join timer, adds the introducer to the group list, sets `numInMemory`, and changes to `GsCollecting`,

then calls `recvAllGsInfo` to check if all group info has arrived. When receiving a `Gsinfo` (group structure info), if in `GsCollecting` and the inform ID matches, we add the sender to `groupList` and again check if all group info has arrived.

The `recvAllGsInfo` checks if we have collected all expected group-info messages. If yes (and we are in `GsCollecting`), we reset the state to `Idle` and clear `localInformId` and `numInMemory`. When receiving an `Informack`, if its ID matches our `localInformId`, we add the sender to `ack`; otherwise, we requeue it. Then `recvAllInformAck` is called.

The `recvAllInformAck` checks if the `ack` set covers the current active group. If yes, it cancels the timer, transitions to `Introducer`, and calls `updateOp` to apply the group update. When receiving a `Finish` or `Joinfail`, the node knows that joining has failed.

Other helper functions like `idleDetectFail`, `introDetectFail`, `idleLeave`, `tryDeleteInform`, `selfUpdate`, `informTimeout` detect failures or leave events and send appropriate group updates. For example, `idleDetectFail` marks lost nodes for removal, `tryDeleteInform` sends `Inform` messages trying to inform other group members about the changes, and `selfUpdate` handles the case when the group list is empty after deleting the failed or left nodes.

A fixed set of message types and handlers covers all group membership events that can occur.

4. Code execution and runtime behavior

In this section, we illustrate the actual runtime behavior of our protocol. We explain how each node starts up, joins the group, and processes messages. The printed logs show state changes and membership updates in a clear, step-by-step manner. Sample outputs from three nodes highlight the join sequence and confirm the protocol's correctness.

As illustrated in Listing 7, the `Main.hs` module handles command-line arguments, node creation, and group creation or joining. Depending on the argument count, a node either becomes the first member (creating a new group) or attempts to join an existing group via an introducer. After the node joins an existing group, it broadcasts a message in the group.

We use `getArgs` to read command-line parameters. If three arguments are provided (`hostStr`, `portStr`, `intervalStr`), this node becomes the first member. It calls `createNode`, then `createGroup`, and prints a startup message. Finally, it loops forever to keep the process alive. If five arguments are provided (including `introHost` and `introPortStr`), this node joins an existing group. After `createNode`, it prints a start message, calls `join`, waits eight seconds to let the joining be completed, broadcasts "I joined" to all current members, and then loops forever. Any other argument pattern prints "Wrong args."

The runtime interactions of three nodes are shown in the following outputs. Node 1 starts the group, and Nodes 2 and 3 join sequentially through the speci-

fied introducers. Each node prints a series of internal operations that reflect the progress of the protocol. Messages such as `Joinreq`, `Inform`, and `Operation` show the steps of membership negotiation and state propagation. Besides the messages of the group membership protocol, the client group broadcast message "I joined" is also printed out in the output.

As demonstrated in Listing 8, the first node (Node 1) starts by printing its own identifier and the message-fetch interval. This log confirms that the UDP socket and secure channel are ready. Shortly after, Node 1 receives a `Joinreq` message from Node 2. Upon handling this request, it invokes `recvJoinRequest`, which moves Node 1 into the `WaitingInformAck` state and broadcasts an `Inform` message to its (currently empty) group. Since there are no other members yet, `recvAllInformAck` immediately finds that all acknowledgments have "arrived," causing Node 1 to transition to the `Introducer` role. In its introducer role, Node 1 executes `updateOp`, calculates the new group membership (which now just includes Node 2), and prints the updated `groupList`. Later, when Node 1's `handleReceive` thread decodes an `Inform` from Node 2 (indicating that Node 2 is notifying the joining of Node 3), Node 1's `recvInform` call acknowledges it, moves to `Informed`, and starts an informed-state timer. Finally, Node 1 receives an `Operation` message from Node 2. This `recvOp` call merges Node 3 into Node 1's `groupList`, so that it now shows both Node 2 and Node 3, and then returns Node 1 to `Idle`. The last line, "I joined," is a client broadcast message that simply confirms Node 3 is connected via Node 1.

As presented in Listing 9, node 2 begins by printing its own identifier and interval. It immediately sends a `Joinreq` to Node 1. Soon after, Node 2's `handleReceive` thread picks up a `Groupstruct` from Node 1. The `recvGroupStruct` call records Node 1 in its `groupList`, and moves Node 2 into `GsCollecting`. Since there is only one member to collect, `recvAllGsInfo` quickly finds that the size of `groupList` matches `numInMemory`, causing Node 2 to become `Idle`, which means Node 2 joined the group successfully. The printed "groupBroadcast True" confirms that the broadcast sent by the client has succeeded.

Next, a `Joinreq` from Node 3 arrives. Node 2's `recvJoinRequest` logs "[recvJoinRequest] executing" and broadcasts an `Inform` to its current group (which contains only Node 1). Then, when Node 2 sees an `Informack` from Node 1, `recvInformAck` adds Node 1 to its `ack` set. Because Node 2 was already in `WaitingInformAck`, `recvAllInformAck` now moves it to `Introducer` and schedules `updateOp`. When `updateOp` runs, Node 2 computes its new group (Nodes 1 and 3), sends an `Operation` to Node 1, sends an `Groupstruct` to Node 3, and prints its updated `groupList`. Finally, the printed-out string "I joined from 127.0.0.1:10002" shows that Node 2 has received the client broadcast message from Node 3.

As shown in Listing 10, node 3 logs its own startup similarly. It sends a `Joinreq` to Node 2, as indicated by "[sendJoinRequest] executing." Soon after, Node 3 decodes a `Groupstruct` from Node 2. Its `recvGroupStruct`

adds Node 2 to `groupList` and moves into `GsCollecting` (it also knows the size of the current group). Next, Node 3 sees a `Gsinfo` from Node 1. Since Node 1's `informId` matches Node 3's `localInformId`, `recvGsInfo` adds Node 1 to `groupList` and logs "[recvGsInfo] executing." Finally, `recvAllGsInfo` recognizes that Node 3 has now gathered information from both Node 1 and Node 2, so it changed Node 3 to `Idle`, which means Node 3 has joined the group. The final "groupBroadcast True" shows that Node 3 has sent a group broadcast to the entire group to announce its arrival.

Figure 1 illustrates the two-round join process. In round 1, Node 2 sends a `Joinreq` to Node 1, receives a `Groupstruct` reply from Node 1, and then sends `ClientMsg1` to Node 1. In round 2, Node 3 issues a `Joinreq` to Node 2. Node 2 forwards a `Inform` to Node 1. Node 1 returns a `Informack` to Node 2. Node 2 completes the update by sending `Operation` to Node 1 and sending `Groupstruct` to Node 3. Then, Node 1 delivers `Gsinfo` to Node 3. Finally, Node 3 sends `ClientMsg2` to both Node 2 and Node 1.

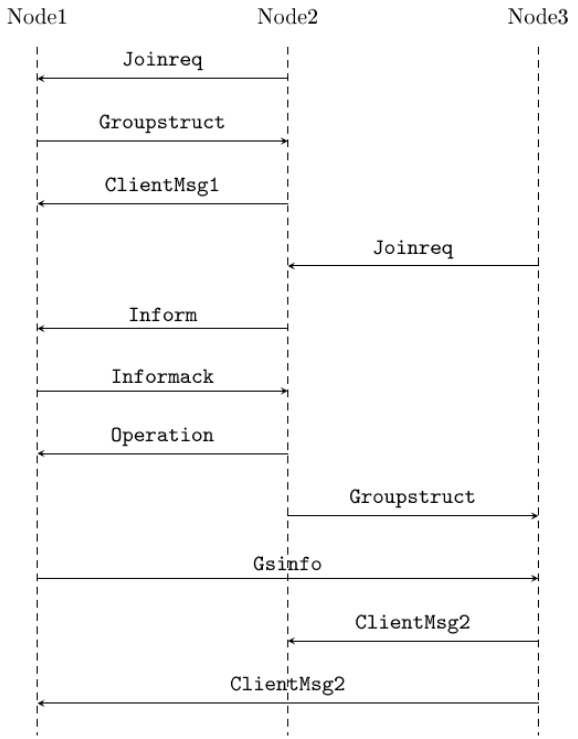

Figure 1. Message flow across three nodes

The client messages are not required for updating group membership. We use it to verify that the group broadcast works correctly after the group membership has been updated. This sequence ensures that each join is coordinated through an inform/ack phase and that group state remains consistent.

The runtime logs display the execution of the protocol, showing clear state transitions for the joining, inform/ack, and update phases. Each node's output confirms that membership messages propagate in the correct order and demonstrates that the implementation faithfully realizes the intended membership protocol under real execution.

5. Benchmark description

In this section, we describe the design of our energy benchmark. We outline the roles of the Coordinator and Sensor Nodes, the parameters they exchange, and the method used to measure energy usage. We also detail the baseline tests that isolate pure communication costs and introduce the normalized energy ratio used to compare protocols.

We developed a benchmark to measure the total energy consumption of the protocol in an IoT-like scenario. The system consists of one *Coordinator* and multiple *Sensor Nodes*. After all nodes have started and connected, the Coordinator sends each Sensor Node three parameters: the signal frequency f, the duration t, and the Coordinator's address.

The Coordinator then begins estimating its own energy consumption over a period of $t+5$ seconds. Upon receiving f and t, each Sensor Node estimates its energy usage over $t+5$ seconds and transmits a sensor message to the Coordinator at an interval of f for exactly t seconds. The Coordinator logs or stores all received messages. Finally, we sum the energy consumption estimation results (produced by the energy profiling tool) from the Coordinator and all Sensor Nodes to obtain the total energy consumption of the distributed system.

In our tests, the Coordinator and Sensor Nodes run on separate, homogeneous machines. The duration t is fixed at 60 seconds. We conduct experiments by varying the signal frequency f (100 milliseconds, 500 milliseconds, 1000 milliseconds, 2000 milliseconds), the message size L (100 bytes, 200 bytes, 400 bytes) and the number of Sensor Nodes N (5, 10, 20, 30, 40, 50, as permitted by lab conditions).

This benchmark reflects realistic IoT use cases. In industrial monitoring, message rates are high (f = 100–500 milliseconds) with small payloads (L = = 100–200 bytes). In environmental sensing, rates are lower (f = 1000–2000 milliseconds) with larger payloads (L = 200–400 bytes).

To measure pure communication costs, we perform a baseline test between two machines, A and B. Machine A sends messages of size L at a frequency of f for a duration of t, while B only listens and discards packets. Both machines remain lightly loaded, and unrelated services are disabled.

The energy is measured via CPU RAPL counters or external energy consumption measurement tools at each end. When comparing to multi-node protocol tests, we scale the two-node baseline energy $E_{\text{base}}(f, L, t)$ linearly to

N nodes (each sending at f). We then define the normalized energy ratio.

$$I(f, L, t; N) = \frac{E^{(N)}_{\text{proto}}(f, L, t)}{N E_{\text{base}}(f, L, t)},$$

where $E^{(N)}_{\text{proto}}$ is the total energy with protocol logic, and E_{base} is the two-node pure-communication energy.

6. Evaluation implementation

In this section, we provide a detailed description of the four implementations used in our evaluation. The lazy baseline test implementation relies on raw sockets and the `Secure-UDP` library. It supports two roles, `receiver` and `sender`. Each role requires both an IP address and a port to be specified on the command line.

In `receiver` mode, the code parses the local IP and port, creates a datagram socket, configures and starts a SecureUDP channel, and then runs energy measurement. It listens for 60 seconds, repeatedly calling `Sec.getReceived` to retrieve decrypted messages and print them.

In `sender` mode, the program binds its socket, configures a SecureUDP channel, and begins energy profiling; it then builds a fixed-size payload, computes the send count from the given interval and duration, and in a tight loop invokes `Sec.sendMessages` at each interval before cleanly closing the socket. The energy profiler is launched via an external AMD tool on a background thread.

The SWIM baseline test uses `net.Listen-Packet` and `net.DialUDP` for communication, and an `exec.Command` to start the AMD energy tracer.

In the lazy test, a coordinator waits for sensors to join, then broadcasts a `START` message with frequency, duration, and size. Each sensor parses the `START` and sends its own stream of messages back to the coordinator. Energy tracing begins just before data exchange.

The SWIM test uses the `memberlist` library for membership. After the group forms, the coordinator sends a `START` packet via UDP to each member's data port. Sensors receive this single packet, then push data back at the given interval. Energy recording is triggered at the same point.

7. Energy measurement results

In this section, we report the energy measurements gathered during our experiments. All experiments were conducted on two laptops running Windows 11: one equipped with an AMD Ryzen 7 7840HS CPU and 32 GB of RAM and the other with an AMD Ryzen 7 8745HS CPU and 16 GB of RAM. Energy

measurements were collected via AMD's AMDuProfCLI tool with a sampling interval of 200 milliseconds over 65 seconds, while each test ran for 60 seconds of active message exchange. We vary the send interval $i \in \{200, 500, 1000\}$ milliseconds and message size $m \in \{100, 200, 400\}$ bytes in all tests.

As shown in Figure 2, in the baseline tests, the Haskell implementation with SecureUDP consistently draws around 21 Watts, roughly three times the 6 to 7 Watts observed for the Go with UDP test. This difference likely stems from the overhead of SecureUDP, as well as heavier runtime and garbage collection activity in Haskell. Neither the send interval nor the message size strongly affects the average power.

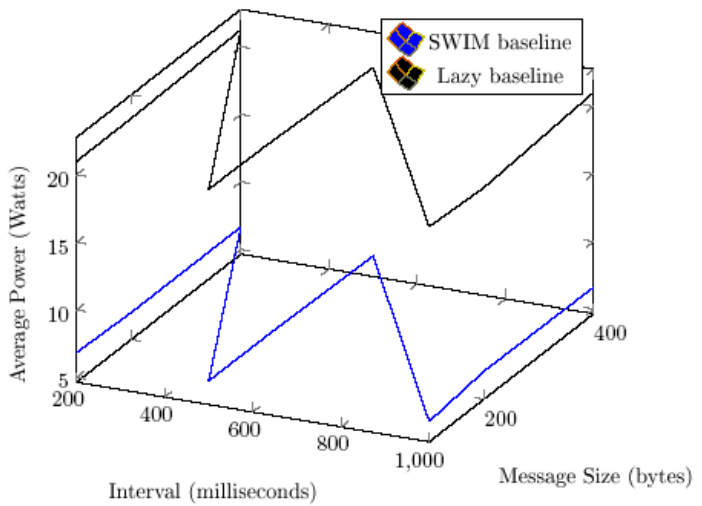

Figure 2. Baseline energy consumption for pure message exchange on two nodes.

As shown in Figure 3, when running the complete protocol logic, the Lazy protocol implementation consumes around 19–20 Watts on average, compared to 6–7 Watts for the SWIM protocol implementation. The SecureUDP channel plus group management logic and Haskell runtime overhead may account for most of the extra draw. We observe a slight downward trend at longer intervals, as fewer messages per second result in reduced transmission activity.

As shown in Figure 4, the normalized energy index falls below 1 for the lazy protocol in all configurations, indicating that the custom Haskell protocol, despite its higher absolute draw, is more sustainable relative to its baseline than SWIM is. Both indices vary only slightly with interval and message size since baseline and protocol costs change in a similar proportion.

The figure shows that both protocols produce the same surface shape under identical test settings, with the vertical axis indicating the normalized energy index. Overall, under this benchmark test the energy index of the Lazy protocol is lower than that of the SWIM protocol, indicating greater sustainability.

We assume the Lazy protocol benefits from the removal of the periodic heartbeat messages. By eliminating them, the Lazy protocol sends fewer control messages when there is no change in group membership. However, its absolute power draw is higher, primarily due to the costs of SecureUDP and the Haskell runtime. Interval and payload size have only a limited impact on these tests.

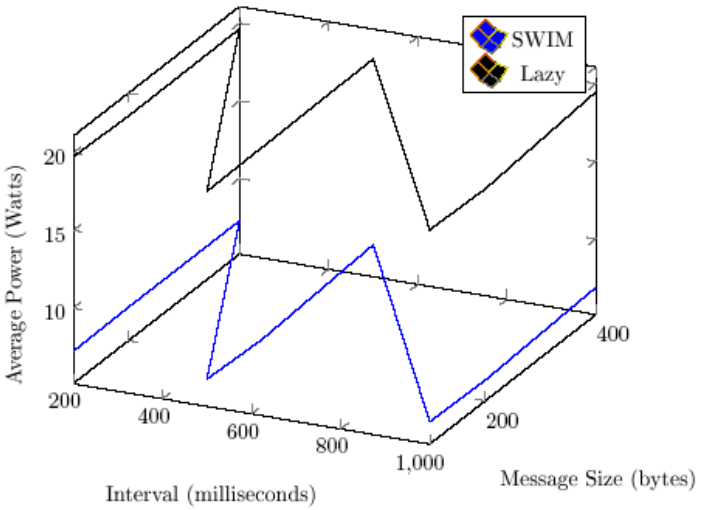

Figure 3. Energy consumption of the protocol implementation

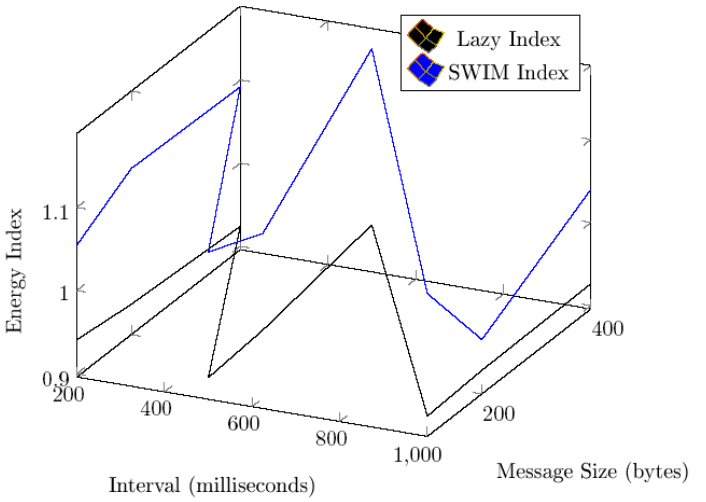

Figure 4. Normalized energy index comparing protocol to baseline.

8. Conclusion

We have presented a lazy group membership protocol implemented in Haskell, designed to reduce periodic messages. The energy-focused benchmark shows that, while the absolute power draw is higher due to SecureUDP and the Haskell runtime, the lazy protocol's normalized energy index is low, indicating better sustainability relative to pure communication costs. The primary benefit of the lazy protocol is energy sustainability, as it eliminates the need for periodic messages, resulting in lower energy consumption.

Overall tests demonstrate consistent feature behavior across intervals and message sizes, confirming that the protocol runs efficiently under varying loads and validating the accuracy of our energy measurements through repeated trials.

The future work includes optimizing the implementation by improving the underlying reliable message sending library. We also plan to explore adaptive timeout schemes to reduce latency, integrate with real IoT deployments for field validation, and include new formal verification based on the implementation to strengthen correctness guarantees.

A. Code execution

```
{−# LANGUAGE OverloadedStrings #−}
module Main (module Main) where

import Protocol
import System.Environment (getArgs)
import Control.Monad (forever)
import Control.Concurrent (threadDelay)

main :: IO ()
main = do
  args <− getArgs
  case args of
    [hostStr, portStr, intervalStr] −> do
      let portR = read portStr :: Int
          interval = read intervalStr :: Int
      node@(nodeC,_,_,_) <− createNode hostStr portR interval
      createGroup node
      putStrLn $ "First node started " ++ show nodeC
      forever $ threadDelay 10000000
    [hostStr, portStr, intervalStr, introHost, introPortStr] −> do
      let portR = read portStr :: Int
          interval = read intervalStr :: Int
          introPort = read introPortStr :: Int
          introId = NodeId { host = ipToTuple introHost, port = introPort }
      node@(nodeC,_,_,_) <− createNode hostStr portR interval
```

```
    putStrLn $ "Node started" ++ show nodeC
    join node introId
    threadDelay 8000000
    s <- groupBroadcast node "I joined"
    case s of
      True -> putStrLn "groupBroadcast True"
      False -> putStrLn "groupBroadcast False"
    forever $ threadDelay 10000000
  _ -> do
    putStrLn "Wrong args"
```

Listing 7. Main.hs implementation for lazy protocol

```
stack exec lazy-exe 127.0.0.1 10000 500000
Node created with NodeId: NodeId {host = (127,0,0,1), port = 10000}
First node started Node Id NodeId {host = (127,0,0,1), port = 10000}
msgFetchInterval 500000
[handleReceive] Decoded group control message:
Joinreq {sender = NodeId {host = (127,0,0,1), port = 10001}}
[recvJoinRequest] executing
[recvAllInformAck] executing
[updateOp] executing, groupListfromList [NodeId {host = (127,0,0,1),
port = 10001}]
[handleReceive] Decoded group control message:
Inform {informId = (NodeId {host = (127,0,0,1), port = 10001},1)}
[handleReceive] Decoded client message: I joined from 127.0.0.1:10001
[recvInform] executing
[handleReceive] Decoded group control message:
Operation {informId = (NodeId {host = (127,0,0,1), port = 10001},1),
newMembers = fromList [NodeId {host = (127,0,0,1), port = 10002}],
removeMembers = fromList []}
[recvOp] executing, groupListfromList [NodeId{host = (127,0,0,1), port = 10001},
NodeId {host = (127,0,0,1), port = 10002}]
[handleReceive] Decoded client message: I joined from 127.0.0.1:10002
```

Listing 8. Node 1 Execution Output

```
stack exec lazy-exe 127.0.0.1 10001 500000 127.0.0.1 10000
Node created with NodeId: NodeId {host = (127,0,0,1), port = 10001}
Node startedNode Id NodeId {host = (127,0,0,1), port = 10001}
msgFetchInterval 500000
[sendJoinRequest] executing
[handleReceive] Decoded group control message: Groupstruct {memberNum = 1,
informId = (NodeId {host = (127,0,0,1), port = 10000},1), introLeave = False}
[recvGroupStruct] executing, groupListfromList [NodeId {host = (127,0,0,1),
port = 10000}]
[recvAllGsInfo] executing, groupListfromList [NodeId {host = (127,0,0,1),
port = 10000}]
[handleReceive] Decoded group control message:
Joinreq {sender = NodeId {host = (127,0,0,1), port = 10002}}
```

```
[recvJoinRequest] executing
groupBroadcast True
[handleReceive] Decoded group control message:
Informack {informId = (NodeId {host = (127,0,0,1), port = 10001},1),
sender = NodeId {host = (127,0,0,1), port = 10000}}
[recvInformAck] executing
[recvAllInformAck] executing
[updateOp] executing, groupListfromList [NodeId {host = (127,0,0,1),
port = 10000},NodeId {host = (127,0,0,1), port = 10002}]
[handleReceive] Decoded client message: I joined from 127.0.0.1:10002
```

Listing 9. Node 2 Execution Output

```
stack exec lazy−exe 127.0.0.1 10002 500000 127.0.0.1 10001
Node created with NodeId: NodeId {host = (127,0,0,1), port = 10002}
Node startedNode Id NodeId {host = (127,0,0,1), port = 10002}
msgFetchInterval 500000
[sendJoinRequest] executing
[handleReceive] Decoded group control message: Groupstruct {memberNum = 2,
informId = (NodeId {host = (127,0,0,1), port = 10001},1), introLeave = False}
[recvGroupStruct] executing, groupListfromList [NodeId {host = (127,0,0,1),
port = 10001}]
[handleReceive] Decoded group control message:
Gsinfo {informId = (NodeId {host = (127,0,0,1),
port = 10001},1), sender = NodeId {host = (127,0,0,1), port = 10000}}
[recvGsInfo] executing, groupListfromList [NodeId {host = (127,0,0,1),
port = 10000}, NodeId {host = (127,0,0,1), port = 10001}]
[recvAllGsInfo] executing, groupListfromList [NodeId {host = (127,0,0,1),
port = 10000},NodeId {host = (127,0,0,1), port = 10001}]
groupBroadcast True
```

Listing 10. Node 3 Execution Output

References

[1] **Barrientos, F.J.A.C.,** secureUDP: Setups secure (unsorted) UDP packet transfer, *Hackage* https://hackage.haskell.org/package/secureUDP

[2] **Cheng, Q., C. Hsu and L. Harn,** Lightweight noninteractive membership authentication and group key establishment for WSNs, *Mathematical Problems in Engineering*, vol. 2020, 2020, Article ID 1452546. https://doi.org/10.1155/2020/1452546

[3] **Das, A., I. Gupta and A. Motivala,** SWIM: Scalable weakly-consistent infection-style process group membership protocol, in *Proc. Int. Conf. Dependable Systems and Networks* (*DSN*), IEEE, 2002, pp. 303–312. https://doi.org/10.1109/DSN.2002.1028914

[4] **HashiCorp,** Gossip Protocol, *GitHub*
https://github.com/hashicorp/serf/blob/master/docs/internals/gossip.html.markdown

[5] **HashiCorp,** Introduction to Serf, *GitHub*
https://github.com/hashicorp/serf/blob/master/docs/intro/index.html.markdown

[6] **Kim, C. and J. Ahn,** Causal order protocol based on virtual synchronous group membership in wireless sensor networks, *Int. J. Control and Automation*, **8(2)** 2015, 9–20.
https://doi.org/10.14257/ijca.2015.8.2.02

[7] **Shi, H., M. Fan, Y. Zhang, M. Chen, X. Liao and W. Hu,** An effective dynamic membership authentication and key management scheme in wireless sensor networks, in *2021 IEEE Wireless Communications and Networking Conference* (*WCNC*), IEEE, Nanjing, China, 2021, pp. 1–6.
https://doi.org/10.1109/WCNC49053.2021.9417320

Jianhao Li
https://orcid.org/0009-0000-0556-6423
ELTE Eötvös Loránd University
Faculty of Informatics
Department of Programming Languages and Compilers
H-1117 Budapest, Pázmány Péter sétány 1/C.
Hungary
lijianhao@inf.elte.hu

Viktória Zsók
https://orcid.org/0000-0003-4414-6813
ELTE Eötvös Loránd University
Faculty of Informatics
Department of Programming Languages and Compilers
H-1117 Budapest, Pázmány Péter sétány 1/C.
Hungary
zsv@inf.elte.hu