Formal Specification and Functional Programming Implementation of Distributed Lazy Group Membership Protocol

Jianhao Li and Viktória Zsók[1]

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, zsv@inf.elte.hu

Abstract. In distributed systems, nodes typically collaborate to accomplish tasks. Many distributed systems adopt groups or clusters as architectural units to enhance the management and coordination of nodes. The group membership protocol serves a crucial role in ensuring a consistent understanding of member status across all nodes in the group. Group membership protocols are essential in various real-world distributed systems, such as IoT sensor networks, smart agriculture, environmental monitoring, satellite communications, remote infrastructure monitoring, smart buildings, industrial automation, intelligent transportation systems, and medical monitoring systems. Group membership protocols are traditionally categorized as heartbeat-based or gossip-based. Both protocol categories require frequent heartbeat or ping messages, facing challenges in energy efficiency, especially in systems with low failure rates and high energy consumption requirements. This paper introduces a novel group membership protocol with two main algorithms that eliminate the need for periodic ping or heartbeat messages. This protocol is designed for Symmetric P2P Systems using the actor model and ensures strong consistency. We provide formal specifications and verifications using TLA+ to ensure the correctness of our algorithms. The safety and liveness properties of the specifications are verified with the TLC model checker. Additionally, the protocol implementation in functional programming language is provided and evaluated.

Keywords: Distributed systems · Group membership protocol · Formal specification · Actor model · Concurrent functional programming.

1 Introduction

Group membership protocols [19] play a crucial role in many real-world distributed systems. In IoT sensor networks, for example, group membership protocols are essential. In smart agriculture, sensors need to be dynamically added or removed to meet changing agricultural demands, which indicates the need for a protocol to manage these nodes. Similarly, environmental monitoring systems must dynamically adjust sensor deployments as monitoring regions and parameters change.

Satellite communication systems require group membership protocols to maintain coordination, especially when new satellites are launched or old ones are decommissioned. Remote infrastructure monitoring systems, such as those for power grids or oil pipelines, rely on group membership protocols to manage the addition of new monitoring nodes and the replacement of failed nodes. Smart building systems, like Building Management Systems (BMS) [23], need to manage the dynamic addition and maintenance of devices and sensors. Industrial automation systems, including factory automation and monitoring systems, frequently change and upgrade production equipment and monitoring nodes, requiring a protocol to ensure system coordination. Intelligent transportation systems (ITS) [4] need to dynamically adjust and expand traffic monitoring devices to accommodate changing traffic patterns. In medical monitoring systems, such as remote patient monitoring, devices are dynamically added, removed, and replaced, necessitating a reliable group membership protocol to maintain system coordination.

However, the nodes in these systems typically experience very low failure rates but have high energy consumption requirements. For instance, sensors in smart agriculture are usually battery-powered and seldom fail due to rigorous environmental testing. Environmental monitoring system [20] sensors are often installed in fixed positions, requiring high energy efficiency and experiencing rare failures due to stable environments. Satellites in satellite communication systems need highly energy-efficient communication and operation systems, but node (satellite) failures are rare due to high design and manufacturing standards. Sensor networks monitoring power lines and substations require long-term stable operation with high energy demands but low equipment failure rates. Sensors deployed along oil pipelines for leak and pressure detection are designed with strict energy requirements and low failure rates due to the difficulty of maintenance. Sensors and control nodes in smart buildings, such as HVAC (heating, ventilation, and air conditioning) systems [25], lighting, and security, are designed for energy efficiency and have low failure rates due to stable environments. Sensors and actuators in industrial automation systems, used for monitoring and controlling factory production lines, face low failure rates due to stable environments and robust design but require strict energy efficiency to reduce operational costs. Traffic monitoring sensors for intelligent transportation systems, deployed in fixed positions like roads and bridges, require low energy consumption to extend device lifespan and experience low failure rates due to stable environments. Wearable devices and sensors in medical monitoring systems need long-term low-power operation with very low failure rates due to rigorous testing and certification.

The group membership protocol in distributed systems involves ensuring that all members in the group maintain an up-to-date and consistent group member list of active members [6].

The current group membership protocols can be categorized into two types: the heartbeat-based group membership protocol and the gossip-based group membership protocol.

The heart-beating group membership protocol is the traditional type. However, implementations of the heart beating suffer from scalability limitations, which im-

pose network loads that grow quadratically with group size, compromise response times, or false positive frequency [7].

Gossip-based group membership protocol is created to provide greater scalability than the heartbeat-based group membership protocol. Each node forwards messages to a small set of "gossip partners" chosen randomly from the entire group members in a gossip-based group membership protocol [2].

However, taking the well-known SWIM algorithm [7] as an example, the gossip-based protocol still needs each node to periodically send ping messages to a random number of other nodes in the local table and to initiate a failure confirmation based on the acknowledgment timeout, requesting assistance from several other nodes to verify the failure. SWIM uses an infection-style dissemination mechanism because hardware and IP multicasts are infrequently enabled on most networks and operating systems due to administrative reasons. These are also the reasons why the basic SWIM protocol can only utilize a costly broadcast or an inefficient point-to-point messaging scheme [7].

In systems where node failures are rare and energy efficiency is critical, traditional group membership protocols with periodic messages are unsustainable. Even reducing the frequency of heartbeat messages still results in unnecessary periodic communications, consuming significant network bandwidth. Moreover, many sensor networks use a duty cycling strategy to save energy by periodically switching sensors between active and sleep states. Protocols like SWIM, gossip-based protocols, and Chord [21] rely on periodic messages, which can conflict with duty cycling strategies. Periodic messages require nodes to wake up regularly for communication, increasing energy consumption and negating the benefits of duty cycling. Alternatively, prioritizing duty cycling can lead to message transmission delays and communication failures, reducing the effectiveness and consistency of the group membership protocol.

To address the above issues, we have designed a novel group membership protocol with two key components: a group list consistency algorithm and a lazy failure detection algorithm.

Our lazy failure detection algorithm significantly reduces communication overhead by avoiding periodic ping or heartbeat messages. When no tasks are present, communication overhead is nearly nonexistent. In the lazy failure detection algorithm, the ping messages are not sent based on time intervals. Only when a task-related message transmission fails does the sending node become suspicious of the receiving node's potential failure. Ping messages are only used when a node suspects another node's failure and seeks other nodes' assistance for confirmation during the failure confirmation phase.

Weak or eventual consistency algorithms might not provide sufficient guarantees for group list consistency in the absence of periodic messages since those algorithms allow some nodes to be temporarily unaware of the existence of new nodes. Therefore, a strong consistency algorithm design is necessary to ensure that all group members are aware of group list updates after a complete system operation.

In Symmetric P2P Systems [12], all nodes have peer-to-peer status, no central control point or master-slave relationship, and all nodes have the same roles and

rights. Symmetric P2P systems are more suitable for the above sensor systems because the decentralized architecture can avoid single points of failure and improve the scalability and fault tolerance of the system. The P2P system can easily add or remove nodes to adapt to dynamic changes in the system. So, our algorithm design is for Symmetric P2P Systems.

Our protocol leverages the actor model [1], where each actor is an independent computational entity with its state and behavior. Actors communicate asynchronously through message passing instead of directly accessing shared resources, which is well-suited for handling concurrent communication tasks efficiently.

This design adheres to the non-Byzantine model [14], where nodes fail by stopping operations without sending false or malicious messages.

The challenge in designing this group list consistency algorithm is that we need to consider the possibility of node failure and messages being received unordered. In addition, under the architecture requirements of symmetry P2P and the actor model, as few messages as possible are used to achieve strong consistency in the group list.

Existing distributed algorithms do not fully align with the assumptions and requirements of our target scenario. Distributed mutual-exclusion algorithm [13] does not account for node failures and assumes ordered message reception. At the same time, the commit protocols (like two-phase commit protocol [5], three-phase commit protocol [5] and Paxos commit algorithm [8]) and the consensus algorithms (like Paxos [14] and Raft [17]) do not meet the assumptions of symmetric P2P systems. Therefore, we have developed a new group list consistency algorithm tailored to these needs. Detailed related work can be found in section 7.

We define the specification of the two algorithms of the group membership protocol using TLA+ [15,24], which enables us to express the algorithm designs precisely. Moreover, formal verification with TLA+ provides us with more information about the correctness of the algorithms since TLA+ is a formal specification language, functioning as a high-level language for modeling distributed programs and systems. The specification contains specific safety and liveness properties that ensure the group list consistency (described in subsection 2.1), and the fail is eventually confirmed (described in subsection 3.1).

Although some research uses TLA+ for formal verification of distributed systems, TLA+ specifications specifically targeting algorithms of group membership protocol are relatively less. Existing TLA+ applications are more concentrated in distributed databases, consensus algorithms (such as Paxos and Raft), and other fields. We hope our work can be seen as one more step towards formalizing and proofing distributed algorithms and providing other software architects with practical experience in formal specification.

In summary, this paper introduces a novel group membership protocol containing two algorithms. It also provides formal specifications, verifications and implementations for these two algorithms.

In the following sections, first, the design of two algorithms is introduced with the problem, methodology, algorithm description, formal specification, and message complexity in section 2 and in section 3, respectively; next, the specification

verification is presented in section 4; afterward, the functional programming implementation is described and evaluated in section 5; next, the algorithms' features, limitations and future work are discussed in section 6; next, related work is analyzed in section 7; finally, the conclusion is summarized in section 8.

2 Group list consistency algorithm

In this section, the group list consistency algorithm is introduced. The algorithm description in natural language is easier to understand than in formal specifications. However, it is easier to produce ambiguity. Therefore, it is suggested that the implementation be constructed according to the specifications.

2.1 Problem

The problem is how to achieve the group list consistency following the actor model and the symmetric P2P architecture. Sustainability and scalability should also be considered during the design phase.

The group list consistency algorithm specification assumes that:

(A1) *The receiver will eventually receive the message as long as it is active.* This assumption is fulfilled by the underlying communication method.

(A2) *Nodes may experience non-byzantine failures [3].* The nodes that have not failed will send responses in a certain amount of time for the message they have to respond to. The failed node loses all the information. (The nodes do not have stable storage that can survive failure and restart).

Considering the existence of the environment without stable storage and the high cost of stable storage, we do not assume all the nodes in the system has stable storage that can survive failure and restart. Therefore, the restarted node needs to rejoin the group.

The requirements of the algorithm specification:

(R1) *Single introducer constraint*: Only at most one node in the group is in the *introducer* state, which can send the group structure to the new group member and inform the existing group members about the new member.

This requirement prevents the group list inconsistency caused by synchronous group joining. Suppose two new group members join the group simultaneously; if two nodes are processing their joining requests separately and simultaneously, the group structure sent back to the new group members may not contain another new group member.

(R2) *Idle group consistency*: If all the group members are idle (which means no changing is happening; there can be changes before or after), then their group list is consistent (for every member u, the group list of u equals the set of all members excluding u).

(R3) *Eventually group idle*: Eventually, for all n in the set of members, the node state of n will become idle. The R2 and R3 imply the eventual consistency of the group lists among the group members.

(R4) *Reliability*: the design considers the non-byzantine failure of a node and message delays. The specification contains the action that lets the node fail. The nodes' failure does not block the system before they recover.

(R5) *Decentralization, symmetry, independence of nodes, and adherence to the actor model.*

Symmetry is a property of P2P systems that declares that all nodes do not have unique capabilities that distinguish them from the rest of the nodes [12]. There is no static centralized manager role of the algorithm. The new member can send the join request to any existing group member. Each existing member in the group has the same role and can temporarily become the *introducer* after receiving the join request. There is no need for a particular node to be more powerful. Additionally, we do not use shared resources (shared variable, shared database) and do not use one node or several nodes to maintain the group structure specifically. All the group nodes keep their own local group structure list. It is also for the scalability consideration (no central bottleneck) and for reliability consideration (inherently free of a single point of failure).

This requirement also demands specifying the system following the actor model. Firstly, each node performs computations and sends messages based solely on its local states without using shared variables. Secondly, since the actor model assumes distributed systems are composed of distributed components that can interact with each other asynchronously [1], it is necessary to specify the mailbox of nodes and the asynchronous communication among the nodes.

(R6) *Sustainability*: the system decreases the number of messages and the hops of each message. The amount of messages should be as low as possible when the new members join the group.

2.2 Methodology

The algorithm is designed to coordinate the nodes through message passing, like most of the existing distributed algorithms. Under the requirements of the actor model and symmetric P2P architecture (nodes have equal roles, no shared resources, and unordered messages), the algorithm achieves group list consistency considering possible node failures. Unnecessary features, such as aborts and event ordering, are excluded. We minimize the number of messages and communication rounds in different design versions to enhance energy efficiency. Additionally, scalability is addressed by implementing the forwarding and batch processing mechanism of join requests.

2.3 Algorithm description

Initially, a node outside the group begins as a *joiner* and sends a joining request to any existing group member B to start the joining process with a joining timeout. If the *joiner* receives a group structure message from any existing group member, it joins the group successfully, updates its local group list according to the information it receives, and becomes an idle group member. If the *joiner* receives a join fail

message, it knows it failed to join the group. Suppose the group structure message does not arrive within the joining timeout. In that case, the *joiner* will check if B has failed (information provided by the lazy failure detection algorithm). If B fails, the *joiner* knows it failed to join the group; if B does not fail, reset the joining timeout. Once the join request has been forwarded to other group members, the *joiner* will receive a handler update message and update the related local information.

When a failure is detected, the idle members update their deletion informing list. Nodes intending to leave voluntarily add themselves to the deletion informing list.

If an idle node finds out its group list except the deletion informing list is an empty set (which means it is the only member in the group), it updates its group list without sending any message.

After receiving a joining request or its delete inform list is not empty, the group node in *idle* state becomes *waitingInformAck* and sends informing messages to all the group members except the nodes in its deletion informing list. Then, it waits for the inform acknowledgments with an inform timeout. If It receives all the inform acknowledgments from all the not-failed nodes of the group list except the deletion informing list, it enters the *introducer* state.

For scalability, the node in *introducer* state can still receive joining requests from other new members or existing group members and detect failed nodes. The node in *introducer* state can also carry out the update operation by sending operation messages (ask the group members to deal with the group list update operation) to all the group members except the delete inform list. It also sends a group structure message to the *joiner* and updates its own local group list.

After the inform timeout of the node B in *waitingInformAck* state is exceeded, if B has not yet received lock acknowledgments from more than half of the not-failed nodes of its group list except the deletion informing list and it has received a lock request message from another node in the state *waitingInformAck* C with larger node ID, then will give up becoming *introducer* by sending:

- finish messages to all the nodes of its group list,
- fail joining message to the related *joiner*,
- inform acknowledgment message to the node C.

The node ID can be changed to other comparable unique values of nodes.

After receiving an inform message, the group members in state *idle* change to state *informed* with an informed timeout and send back the inform acknowledgment message.

For every group member D in the state *informed*: it changes to state *idle* after receiving a finish message; it forwards the join request to the node informed D after receiving a join request from a new group member; it includes the new group members to the local group list, sends back the operation acknowledgment message and change to state *idle* after receiving the operation message; if the node informed the node D failed and the informed timeout is elapsed, it changes to state *idle*.

The details of local information, such as the *informId*, are eliminated from this description.

2.4 Formal specification

As shown in Figure 1, *SingleIntroducerConstraint* formally specifies the safety property **R1** in section 2.

$$SingleIntroducerConstraint \triangleq Cardinality(\{m \in members : localInfos[m][\text{“nodeState”}] = \text{“introducer”}\}) \leq 1$$

Fig. 1. The *SingleIntroducerConstraint* invariant of the group list consistency algorithm

As shown in Figure 2, *IdleGroupConsistency* formally specifies the safety property **R2** in section 2.

$$SingleIntroducerConstraint \triangleq Cardinality(\{m \in members : localInfos[m][\text{“nodeState”}] = \text{“introducer”}\}) \leq 1$$

Fig. 2. The *IdleGroupConsistency* invariant of the group list consistency algorithm

As shown in Figure 3, *EventuallyGroupIdle* specifies the liveness property **R3** in section 2.

$$EventuallyGroupIdle \triangleq \Diamond(\forall\, n \in members :\ localInfos[n][\text{“nodeState”}] = \text{“idle”})$$

Fig. 3. The *EventuallyGroupIdle* property of the group list consistency algorithm

As shown in Figure 4, after receiving the inform message, the group member nodes in the idle state are informed by the inform message sender, and they send a inform acknowledgment back.

2.5 Message complexity

The most significant energy consumption arises from CPU operations and network communication in many low-power devices [16]. In the group list consistency algorithm, we assume that the transmission distance of all communications is the same and only consider the impact of the number of messages on sustainability. In requirement (**R5**), we mentioned that this algorithm should have a low number of messages under the ideal situation (new members join the group in order with some intervals).

```
RecvInform(n) ≜
  ∧ localInfos[n]["nodeState"] = "idle"
  ∧ ∃ i ∈ 1 .. Len(mailboxes[n]) :
    LET
      Msg ≜ mailboxes[n][i]
      InID ≜ Msg["informId"]
      IAck ≜ [type ↦ "informack",
        informId ↦ InID, sender ↦ n]
    IN
      ∧ Msg["type"] = "inform"
      ∧ localInfos' = [localInfos EXCEPT
        ![n]["nodeState"] = "informed", ![n]["informId"] = InID]
      ∧ mailboxes' = [mailboxes EXCEPT
        ![InID["intro"]] = Append(@, IAck),
        ![n] = RemoveByIndex(@, i)]
      ∧ UNCHANGED ⟨members⟩
```

Fig. 4. The *RecvInform* action of the group list consistency algorithm

Within the context of the distributed transaction commit problem, transaction managers [8] are responsible for orchestrating the various steps involved in committing a transaction, managing resource locks, and handling any potential conflicts or failures that may arise during the process. Suppose N is the number of transaction managers (for the distributed transaction commit problem) and the number of existing group members (for the group joining problem). The number of messages in the ideal situation can be calculated: there is one joining request, $N-1$ of inform messages, $N-1$ of inform acknowledgments, N of operation and group structure messages, and N of operation and group structure acknowledgments. There are $4*N-1$ messages in total. Accordingly, there are five message delays [8].

For comparison, we use two distributed asynchronous algorithms dealing with the distributed consistency problem: the two-phase commit protocol [5] and the Paxos commit algorithm [8]. The differences are discussed in detail in section 7. The two-phase commit protocol requires four message delays and uses about $3*N$ messages for larger values of N [8]. The Paxos commit algorithm requires five message delays and uses about $4*N$ messages for larger values of N [8].

	Two-Phase Commit	Paxos Commit	Actor group joining
Message delays	4	5	5
Messages	3*N - 1	4*N	4*N-1

Table 1. Message complexity comparison

3 Lazy failure detection algorithm

This algorithm can provide the fail information needed in the group list consistency algorithm.

Under sustainable consideration, in an actor model system, an actor does not need to be concerned by another node's failure status if there are no message interactions. Therefore, this algorithm achieves the *lazy* characteristic by not using a heartbeat and gossip mechanisms. The failure detection process is initiated only after the acknowledgment timeout of task-related messages rather than being performed periodically.

To provide the node failure-related information of the lazy failure detection algorithm to the group list consistency algorithm, we can set some messages in the group list consistency algorithm as task-related messages or send additional task-related messages when failure information is required.

3.1 Problem

The problem with designing the lazy failure detection algorithm is how to ensure specific safety and liveness properties without periodic messages.

The lazy failure detection algorithm shares the same assumptions as the group list consistency algorithm.

3.2 Methodology

This algorithm employs a timeout mechanism but replaces the conventional heartbeat mechanism with task-related message timeouts for failure detection. A consensus mechanism reduces false positives caused by transient network issues. The lazy failure detection algorithm is specialized. It utilizes a timeout-based failure detector within an asynchronous model, where no upper bounds on message delivery delays and processing times are assumed. Consequently, the timeout-based failure detector may incorrectly suspect failures due to message delays and processing times, thus not meeting the accuracy properties of conventional failure detectors.

Furthermore, eliminating periodic messages means it cannot satisfy the completeness properties typical of standard failure detection algorithms. Specifically, when there are no task-related messages, the algorithm cannot ensure that every failed node will eventually be permanently suspected by all (or some) running nodes. Therefore, unique properties need to be designed for this algorithm.

3.3 Algorithm description

When a node fails to receive the acknowledgment of a task-related message, it starts the failure detection process by sending assist requests to a fixed number of nodes that are randomly chosen from the group nodes (except itself, the task-related message receiver, and the confirmed failed nodes). After receiving the assist

requests, the group member nodes send detection messages to the task-related message receiver and send the detection results to the task-related message sender. The task-related message receiver sends detection responses once it receives detection messages.

3.4 Formal specification

As shown in Figure 5, *NoFalseFailureDetection* specifies a liveness property: for every group node, if aways no messages are sent to the node, it will not be confirmed to failed.

$$
\begin{array}{l}
NoFalseFailureDetection \triangleq \\
\quad \forall\, n \in GroupNodes : \\
\qquad (\Box(\neg\exists\, i \in 1\,..\,Len(mailboxes[n]) : \\
\qquad\quad mailboxes[n][i][\text{“type”}] = \text{“CMsg”}) \\
\qquad \Rightarrow n \notin confirmedFailedNodes)
\end{array}
$$

Fig. 5. The *NoFalseFailureDetection* property of the *lazy* failure detection algorithm

As shown in Figure 6, this algorithm has one more liveness property: for all the nodes, if a node A failed, and there is another node B that tried to send a message M to this failed node A and the detect process of this message M is completed, then eventually the node A will be confirmed failed.

$$
\begin{array}{l}
EventualFailureConfirmation \triangleq \\
\quad \forall\, n \in GroupNodes : (((\exists\, i \ \in 1\,..\,Len(mailboxes[n])) : \\
\qquad \wedge\, mailboxes[n][i][\text{“type”}] = \text{“CMsg”} \\
\qquad \wedge\, mailboxes[n][i][\text{“cMsgId”}] \in detectRecords) \\
\qquad \wedge\, localInfos[n][\text{“nodeState”}] = \text{“Fail”}) \\
\quad \rightsquigarrow (n \in confirmedFailedNodes))
\end{array}
$$

Fig. 6. The *EventualFailureConfirmation* property of the *lazy* failure detection algorithm

3.5 Message complexity

Assuming one actor (or node) will query other A nodes once it fails to receive the acknowledgment, the lazy failure detection algorithm needs $4 * A$ messages for the failure detection.

Assuming t is the time and N is a function of node number, $N(t)$ is the number of nodes present in the system at time t. Similarly, $M_j(t)$ is the number of the task-related message at time t, and $P_f(t)$ is the the probability that failed to receive acknowledgment in time for a task-related message at time t. Then, the M_l equals the number of messages of the lazy failure detection algorithm in the time slot $[t_1, t_2]$.

$$M_l = (4A) \cdot \int_{t_1}^{t_2} N(t) \cdot M_j(t) \cdot P_f(t)\, dt$$

In comparison, assume the algorithms (or protocols) with heartbeat or ping messages have constants for the frequency of the heartbeat or ping message M_p and the number of nodes to which a node should send periodic messages, B. The constants M_p and B usually need to be configured according to the system, and the coverage rate and the number of messages need to be balanced. Then, the M_f stands for the number of messages of the algorithms (or protocols) with periodic messages in the time slot $[t_1, t_2]$.

$$M_f = M_p \cdot B \cdot \int_{t_1}^{t_2} N(t)\, dt$$

As we can see from the formulae, the lazy failure detection has a sustainability advantage when the $M_j(t) \cdot P_f(t)$ is smaller. In other words, this algorithm enhances system sustainability when the task-related messages are fewer or the probability that failed to receive an acknowledgment for a task-related message within the timeout is small. However, the task-related messages among all the group members with reasonable frequency can reduce the cases where failed nodes are not confirmed for a long time. Therefore, this algorithm suits systems with rare failures and reasonably configured timeouts.

4 Formal verification

This section presents the formal verification of the specifications of the algorithms. The TLC model checker is a type of formal verifier that can analyze the possible behaviors of the algorithms. TLC model checker enumerates all the possible behaviors of the systems or algorithm. It contains the combination within which there are many actions between the message-sending action and the message-receiving action, representing the message delay scenario. In that scenario, even if a message arrives early, it may be received (or processed) much later when multiple sub-actions are enabled simultaneously.

All the instances of model checking in this section passed the TLC model checking process, which indicates the specified safety properties are true in every reachable state and the liveness properties hold for every possible behavior of the algorithms.

The machine used for model checking has the following specifications: AMD Ryzen 7 7840HS processor with Radeon 780M Graphics 3.80 GHz, 32 GB of RAM, and Windows 11 as operating system.

4.1 Group list consistency algorithm

Before the model checking, we include the formulae *TypeOK*, *SingleIntroducerConstraint* and *IdleGroupConsistency* as invariants of the model and include the formula *EventuallyGroupIdle* as properties of the model. When we run the test, the TLC will check if those formulas are true in every reachable state (invariant) or for every possible behavior (properties). For the additional TLC options: we set the number of worker threads to 16; set the fraction of physical memory allocated to TCL to 90%; turned off the profiling feature; and checked the option of verifying temporal properties upon termination only.

To run the TLC model checker, we need to provide the value of the constants.

When we provide the *{2,3,4}* and *{2,3}* for the constants *Nodes* and *InitMembers*, the TLC model checker console shows that no error was found when the model checking was completed. 113440090 states were generated, and 71160463 distinct states were found. The fingerprint collision probability is optimistically calculated as 1.6E-4. The depth of the complete state graph search is 58. The model-checking process finished in 332015 milliseconds.

When we provide the *{2,3,4}* and *{2}* for the constants *Nodes* and *InitMembers*, the TLC model checker console shows that no error was found when the model checking was completed. 188905223 states were generated, and 118734590 distinct states were found. The fingerprint collision probability is optimistically calculated as 4.5E-4. The depth of the complete state graph search is 63. The model-checking process finished in 616768 milliseconds.

All the results we show below have been confirmed by the TLC that the model checking was completed, and no error has been found. The states increase as the number of initial members decreases for the same number of nodes. It is reasonable because more nodes and fewer initial members result in more possible behaviors.

4.2 Lazy failure detection algorithm

The algorithm has passed the TLC model checking, including checks for the conventional invariant *TypeOK*, liveness property *NoFalseFailureDetection* and *EventualFailureConfirmation*. Below are the values of the declared constants of the model: *{2,3,4}* (*GroupNodes*), *1* (*AssistNum*), and *1* (*MaxCMsgCount*).

The TLC model checker console shows that no error was found when the model checking was completed. 137905831 states were generated, and 30968036 distinct states were found. The fingerprint collision probability is optimistically calculated as 1.8E-4. The depth of the complete state graph search is 30. The model-checking process finished in 2732687 milliseconds.

5 Implementation

Using Haskell to implement a TLA+ specification offers several advantages. Haskell is a purely functional programming language. Reducing the impure code ensures that the software is solid [18]. Additionally, the functional nature of TLA+ closely

matches functional programming paradigms, making Haskell an ideal choice for translating specifications into code. This matching facilitates a more direct and intuitive mapping from mathematical models to executable code. It helps maintain the rigor and correctness of the original TLA+ definitions in the implementation process.

Haskell has the advantage of providing a large number of reliable and robust packages. For instance, the `Data.Serialize` module from the `Cereal` package [10] enable us to facilitate straightforward encoding and decoding of various messages; the `Data.Set module` from the `containers` package [11] offers an efficient implementation of sets, making the transition from TLA+ specifications to Haskell implementations smoother; and the `SecureUDP` package [9] ensures message delivery through mechanisms like resending and timeouts while preserving the asynchronous, out-of-order nature of communication, which also fulfills the assumption A1 in subsection 2.1.

The Listing 1.1 shows the implementation of the node state and the messages. The default generic implementations of the put and get functions can be included by making the `Message` type deriving `Generic` type class and declaring a `Serialize` instance for Message without any function definition when the *DeriveGeneric* and *DefaultSignatures* language extensions are enabled.

```
data NodeState = Idle | Informed | WaitingInformAck | Introducer | Fail
  | NotInGroup | Joining | FailedJoin | Leave deriving (Show, Eq)

type NodeId = Word16
type InformID = (NodeId, Word32)

data Message
    = Inform {informId :: InformID}
    | Finish {informId :: InformID}
    | Gsack {sender :: NodeId}
    | Joinreq {sender :: NodeId}
    | Joinfail {sender :: NodeId}
       | Handlerupdate {
            new :: NodeId
          , sender :: NodeId}
    | Groupstruct {
            currentMembers :: Set.Set NodeId
       , intro :: NodeId}
    | Informack {
            informId :: InformID
       , sender   :: NodeId}
    | Operation {
            informId :: InformID
       , newMembers    :: Set.Set NodeId
       , removeMembers :: Set.Set NodeId}
    deriving (Eq, Show, Generic)

instance Serial.Serialize Message
```

Listing 1.1. The NodeState and Message

The Listing 1.2 shows the `LocalInfo` definition and the function examples for actions `idleLeave`, `selfUpdate`, and `informedTimeout`. Other functions check the enabling conditions of the above actions.

```
data LocalInfo = LocalInfo
  { nodeState    :: NodeState
```

```
  , localInformId:: InformID
  , introCounter :: Word32
  , inMemory     :: NodeId
  , toAdd        :: Set.Set NodeId
  , toDelete     :: Set.Set NodeId
  , deleteInform :: Set.Set NodeId
  , ack          :: Set.Set NodeId
  , groupList    :: Set.Set NodeId
  } deriving (Show, Eq, Generic)

idleLeave :: NodeId -> LocalInfo -> LocalInfo
idleLeave self localInfo =
  localInfo{deleteInform=Set.insert self (deleteInform localInfo)}

selfUpdate :: LocalInfo -> LocalInfo
selfUpdate localInfo =
  localInfo{groupList=Set.empty, toDelete=Set.empty}

informedTimeout :: LocalInfo -> LocalInfo
informedTimeout localInfo =
  localInfo{nodeState=Idle, localInformId = (0,0)}
...
```

Listing 1.2. The LocalInfo and function examples

The Listing 1.3 shows part of the test of the implementation. The function *idleLeave* and the *Inform* message encoding and decoding are tested.

```
main :: IO ()
main = hspec $ do
  describe "Protocol␣functions" $ do
      it "idleLeave␣should␣add␣self␣to␣deleteInform" $ do
      let initialInfo = LocalInfo
            { nodeState = Idle
            , localInformId = (1, 1)
            , introCounter = 0
            , inMemory = 0
            , toAdd = Set.empty
            , toDelete = Set.empty
            , deleteInform = Set.empty
            , ack = Set.empty
            , groupList = Set.empty
            }
          self = 99
          updatedInfo = idleLeave self initialInfo
      Set.member self (deleteInform updatedInfo) `shouldBe` True
...
  describe "Message␣encoding␣and␣decoding" $ do
    it "Inform␣message␣should␣encode␣and␣decode␣correctly" $ do
      let message = Inform { informId = (1, 100) }
          encoded = Serial.encode message
          decoded = Serial.decode encoded
      decoded `shouldBe` Right message
```

Listing 1.3. The test

The energy consumption estimation of this protocol's Haskell implementation is compared to the gossip-based solution with a different number of nodes and task messages sending frequency in a fixed period.

6 Discussion

Using the protocol presented in this paper, after the network is partitioned, based on the failure detection results, a large group of nodes will gradually be divided into multiple small groups according to the network partition. We are designing protocol supplementary algorithms related to group merge, node self-checking, and security. After the network partition is restored, the small groups can be restored to a large group by merging.

Lazy failure detection can be converted to an algorithm similar to the heartbeat mechanism by disguising heartbeat information as job information and sending it periodically. Compared with heartbeat information, it should be called periodical detection information; however, the principle is similar: sacrificing sustainability in exchange for sensitivity to failure.

When the nodes exchange task-related messages frequently, the speed of detecting failures is not compromised in the lazy failure detection algorithm. However, this approach sacrifices the speed of failure detection when task-related messages between nodes are infrequent. In extreme cases where no nodes in the group are functioning, and there is no communication, failures will not be detected. Such scenarios are acceptable under this protocol. In other words, this algorithm permits the failure of permanently isolated nodes undetected. From the protocol's perspective, the failure status of nodes in an inactive system (assuming the system activity of the actor model is mainly message communication) is non-essential. During idle periods, all machines should conserve energy and rest. When tasks emerge, it will then be determined which nodes have failed. Therefore, this protocol is unsuitable for systems with strict requirements on the detection speed of failed nodes or that rarely communicate between nodes.

Most distributed algorithms rely on a large number of message passing to ensure consistency. As the number of nodes increases, the message overhead will increase linearly or faster, causing network congestion and increased latency, leading to scalability limitations. The group list consistency algorithm includes a join requests forwarding mechanism under scalability consideration to ease the problem. The join requests are processed in a bunch to decrease the round of consistency coordination and the number of messages. However, scalability limitations still exist.

In the group list consistency algorithm, the node in *introducer* state only plays a temporary role as a guide when a new node joins and does not have permanent special privileges or control capabilities. Therefore, the existence of the *introducer* does not violate the principle of the symmetric P2P system.

Since network delay and system load vary, a fixed timeout will cause additional communication overhead in group list consistency algorithm and lead to misjudgment in lazy failure detection algorithm. Therefore, according to the network conditions and system load, a mechanism for dynamically adjusting the timeout can be added to this protocol in the actual implementation.

This protocol is designed for systems with rare failures and communication overhead-reduction requirements. According to the lazy feature, it may also suit data centers with reliable nodes and an upper limit of message delivery delay.

However, evaluations of the implementations in different environments are needed to confirm this statement.

7 Related work

The group list consistency algorithm's primary distinctions from other distributed algorithms or protocols are the requirements and the solved problems.

Lamport's distributed mutual-exclusion algorithm [13] is a prominent solution often considered for addressing mutual exclusion and synchronization in distributed systems. It can be used to create a distributed mutex to lock the distributed group list modification. However, upon reviewing the algorithm, we have discovered that its assumptions and requirements do not align with our specific scenario. Our system cannot fulfill the assumption that the message can be received in order, as we have designed it based on the actor model and connectionless transportation protocol. Moreover, the distributed mutual-exclusion algorithm requires the active participation of all processes, whereas we must account for potential node failures in our system. The primary variation is that our system does not require the total ordering of events. In our scenario, it is possible for the *introducer* who made the earlier lock requests to give up the lock requests. Although we still require mutual exclusion to ensure only one *introducer* enters the critical section, we do not require enforcing a specific order.

Distributed transaction commit algorithms or protocols, like the two-phase commit protocol [5], the three-phase commit protocol [5], and the Paxos commit algorithm [8], can also maintain the group list consistency in a group membership protocol: the group members decide about the group list modifications. However, comparing the group list consistency algorithm with the distributed transaction commit algorithm, the members in our system do not need to abort; our system has a decentralized symmetry architecture requirement. Furthermore, this algorithm solves the blocking problem using timeouts (JoinReqTimeout, InformTimeout, InformedTimeout) and failure detection, which differs from the approach of the three-phase commit protocol. Besides, this algorithm has a forward join requests mechanism to increase the scalability.

In our design, to ensure the independence of nodes, each node has a local group list that contains information about group member nodes' addresses or node IDs, excluding itself. Therefore, we also need to ensure the distributed consistency of the local group list. Since distributed consistency is involved, we also considered two classic distributed consensus algorithms: Paxos [14], and Raft [17]. We can also develop a group membership protocol using distributed consensus algorithms. However, these algorithms do not match our sustainability requirements because of the periodical messages. Additionally, this algorithm is designed specifically for group member updates. Furthermore, the group list consistency algorithm needs to be designed with the actor model for symmetric P2P systems.

8 Conclusion

The paper presented a novel sustainable group membership protocol containing two main algorithms. The design, formal specification, verification, and functional programming implementation of the two main algorithms of the protocol were introduced. Our approach eliminates the need for periodic messages, reducing communication overhead through a lazy failure detection algorithm. The group list consistency algorithm provides a consistency solution for a decentralized distributed system following the actor model and symmetric P2P systems architecture. Scalability and sustainability are both considered during the design phase.

This work represents an attempt to formally specify and verify distributed systems or algorithms under specific conditions. Practical exercises have demonstrated that high-level abstract mathematical languages, such as TLA+, can significantly aid system architects in the design and verification phases. Such methodologies contribute to a heightened confidence in the system's correctness.

In future work, our protocol can be enhanced with more supplementary algorithms for advanced functional features and security considerations. It can be further proved with the help of the symbolic model checker APALACHE, property reduction methodology, and SMT-based interactive theorem prover Ivy [22]. Further optimizing the protocol can be done for different types of distributed systems and exploring additional applications in environments with varying failure rates and communication patterns.

References

1. Agha, G.A., Kim, W.: Actors: A unifying model for parallel and distributed computing. Journal of Systems Architecture **45**(15), 1263–1277 (1999)
2. Allavena, A., Demers, A.J., Hopcroft, J.E.: Correctness of a gossip based membership protocol. In: Aguilera, M.K., Aspnes, J. (eds.) Proceedings of the Twenty-Fourth Annual ACM Symposium on Principles of Distributed Computing, PODC 2005, Las Vegas, NV, USA, July 17-20, 2005. pp. 292–301. ACM (2005)
3. Aspnes, J.: Notes on theory of distributed systems. CoRR **arXiv, abs/2001.04235** (2020)
4. Bazzan, A.L., Klügl, F.: Introduction to intelligent systems in traffic and transportation. Springer Nature (2022)
5. Bernstein, P.A., Hadzilacos, V., Goodman, N., et al.: Concurrency control and recovery in database systems, vol. 370. Addison-wesley Reading (1987)
6. Birman, K.P.: Guide to reliable distributed systems: building high-assurance applications and cloud-hosted services. Springer Science & Business Media (2012)
7. Das, A., Gupta, I., Motivala, A.: Swim: scalable weakly-consistent infection-style process group membership protocol. In: Proceedings International Conference on Dependable Systems and Networks. pp. 303–312 (2002)
8. Gray, J., Lamport, L.: Consensus on transaction commit. ACM Trans. Database Syst. **31**(1), 133–160 (mar 2006)
9. Hackage Contributors: secureudp - haskell package. `https://hackage.haskell.org/package/secureUDP` (2017), accessed: 2024-12-10

10. Hackage Contributors: cereal - haskell package. https://hackage.haskell.org/package/cereal (2022), accessed: 2024-12-10
11. Hackage Contributors: containers - haskell package. https://hackage.haskell.org/package/containers (2023), accessed: 2024-12-10
12. Hasan, R., Anwar, Z., Yurcik, W., Brumbaugh, L., Campbell, R.: A survey of peer-to-peer storage techniques for distributed file systems. In: International Conference on Information Technology: Coding and Computing (ITCC'05)-Volume II. vol. 2, pp. 205–213. IEEE (2005)
13. Lamport, L.: Time, clocks and the ordering of events in a distributed system. Communications of the ACM 21, 7 (July 1978), 558-565. pp. 558–565 (1978)
14. Lamport, L.: Paxos made simple. ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001) pp. 51–58 (December 2001)
15. Lamport, L.: Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley Longman Publishing Co., Inc., USA (2002)
16. Mohapatra, S., Rahimi, M.R., Venkatasubramanian, N.: Power-aware middleware for mobile applications. In: Ahmad, I., Ranka, S. (eds.) Handbook of Energy-Aware and Green Computing - Two Volume Set, pp. 193–224. Chapman and Hall/CRC (2012)
17. Ongaro, D., Ousterhout, J.: In search of an understandable consensus algorithm. In: Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference. p. 305–320. USENIX ATC'14, USENIX Association, USA (2014)
18. O'Sullivan, B., Goerzen, J., Stewart, D.B.: Real world haskell: Code you can believe in. " O'Reilly Media, Inc." (2008)
19. Reiter, M.: A secure group membership protocol. IEEE Transactions on Software Engineering **22**(1), 31–42 (1996). https://doi.org/10.1109/32.481515
20. Saputri, F.R., Dhaneswari, S.F.: Sensor design for building environment monitoring system based on blynk. Ultima Computing: Jurnal Sistem Komputer **14**(1), 36–41 (2022)
21. Stoica, I., Morris, R., Liben-Nowell, D., Karger, D., Kaashoek, M., Dabek, F., Balakrishnan, H.: Chord: a scalable peer-to-peer lookup protocol for internet applications. IEEE/ACM Transactions on Networking **11**(1), 17–32 (2003). https://doi.org/10.1109/TNET.2002.808407
22. Tran, T.H.: Symbolic Verification of TLA+ Specifications with Applications to Distributed Algorithms. Ph.D. thesis, Ph. D. thesis, Technische Universität Wien (2023)
23. Uzair, M., Yacoub Al-Kafrawi, S., Manaf Al-Janadi, K., Abdulrahman Al-Bulushi, I.: A low-cost iot based buildings management system (bms) using arduino mega 2560 and raspberry pi 4 for smart monitoring and automation. International journal of electrical and computer engineering systems **13**(3), 219–236 (2022)
24. Wayne, H.: Practical TLA+: Planning Driven Development. Apress (2018)
25. Yuan, X., Pan, Y., Yang, J., Wang, W., Huang, Z.: Study on the application of reinforcement learning in the operation optimization of hvac system. In: Building Simulation. vol. 14, pp. 75–87. Springer (2021)