Design of a Sustainable Membership Protocol

Jianhao Li, Viktória Zsók
lijianhao288@gmail.com,zsv@inf.elte.hu
Eötvös Loránd University, Faculty of Informatics
Department of Programming Languages and Compilers
Budapest, Hungary

ABSTRACT

Membership protocols are pivotal in middleware systems, traditionally categorized as heartbeat-based or gossip-based. While the former faces scalability challenges, the latter, like the SWIM algorithm, requires frequent ping messages, leading to potential energy wastage. Addressing this, we present the "lazy membership protocol", a more sustainable solution that triggers ping messages only upon failure suspicion, and a node only becomes suspicious when it fails to receive a business message. It optimally balances failure detection speed based on message frequency and promotes energy conservation. Tailored for energy-conscious systems, this protocol diverges from conventional methods and forms the basis for a new distributed functional programming language with three key algorithms. Our approach aims to guide developers' sustainable design and provide a theoretical reference for the design of distributed communication for functional programming languages.

KEYWORDS

Distributed system, Membership protocol, Actor model, Algorithm, Specification

ACM Reference Format:
Jianhao Li, Viktória Zsók. 2023. Design of a Sustainable Membership Protocol. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

To simplify distributed application development, distributed communication has become essential in some modern functional programming languages. Taking the well-known functional programming language, Erlang [5], as an example, Distributed Erlang provides developers with a mechanism enabling Erlang processes across multiple nodes to communicate and collaborate transparently. However, the TCP connections and heartbeats in the fully connected Distributed Erlang cause overhead in both the nodes and the network [2], thereby constraining scalability. Consequently, many innovative system designs based on Distributed Erlang have emerged, offering solutions that remarkably enhance scalability. These designs commonly adopt innovative architectural designs to reduce the number of connections, such as Riak Core [11] using fully connected nodes as gateways to connect other functional nodes and leveraging consistent hashing [2], and the SD Erlang [3] that uses gateway nodes belonging to multiple groups to connect various groups.

Distinct from the abovementioned research, our goal is to construct a new actor model communication system based on connectionless characteristics. Besides the standard properties, its design emphasizes sustainability and scalability. It integrates reliable data frame transmission and sustainable membership protocol at the application layer. Not only does it leverage architectural designs to enhance the overall system scalability by including various group connection mechanisms, but it also attempts to increase the size of individual groups. The group membership protocol is the foundation for the typical group communication service. In this paper, our primary goal lies in describing the design of the sustainable membership protocol to provide a theoretical foundation or design reference for the distributed communication service for the next generation of functional programming languages or new distributed communication services in existing functional programming languages.

In this paper, we define the specification of the first algorithm of the membership protocol using TLA+ [7]. TLA+ is a formal specification language, functioning as a high-level language for modeling distributed programs and systems. Though it is not a functional programming language, it shares some philosophical similarities with them: TLA+ specifications are expressed in mathematical terms, just as functional programming is rooted in solid mathematical foundations, especially lambda calculus; Pure functions in functional programming are deterministic, producing invariant outputs for identical inputs while, TLA+ specifications outline the system behavior across all possible states and transitions, akin to describing all potential outputs of pure functions; TLA+ is declarative, focusing on the "what" rather than the "how", and functional programming is similarly declarative, describing the definition rather than the exact steps. Furthermore, in TLA+, some function definitions require understanding recursion, which is particularly intuitive for researchers familiar with functional programming languages. Using functional programming languages as a modeling language for distributed algorithms may also be an interesting subject, though it is beyond the scope of this paper.

Lots of existing middleware systems also utilize the membership protocol. The current membership protocols can be categorized into two types: the heartbeat-based membership protocol and the gossip-based membership protocol.

Conference'17, July 2017, Washington, DC, USA
© 2023 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

The heart-beating protocol is the traditional membership protocol. However, implementations of the heart beating suffer from scalability limitations, which impose network loads that grow quadratically with group size, compromise response times, or false positive frequency [4].

Gossip-based membership protocol is created to provide greater scalability than the heartbeat-based membership protocol. Each node forwards messages to a small set of "gossip partners" chosen randomly from the entire group members in a gossip-based membership protocol [1].

However, taking the well-known SWIM algorithm [4] 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, based on the acknowledgment timeout to initiate a failure confirmation, requesting assistance from several other nodes to verify the failure.

For systems where node failures are infrequent, messages are transmitted frequently, but where there is a high demand for sustainability and sensitivity to energy consumption, utilizing heartbeat-based or gossip-based protocols may lead to energy waste. Consequently, we have designed a novel sustainable membership protocol that avoids the need for periodic ping or heartbeat messages and offers weak consistency, which we call the "lazy membership protocol".

The ping messages are not sent based on time intervals. Only when a business message transmission fails the sending node becomes 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. This approach sacrifices the speed of failure detection when business messages between nodes are infrequent. However, when nodes exchange business messages frequently, the speed of detecting failures is not compromised, and a notable enhancement in system sustainability is achieved.

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. 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 nonessential. During idle periods, all machines should conserve energy and rest. When tasks emerge, it will then be determined which nodes have failed.

The mechanism of sending non-business-related messages to some other nodes periodically, regardless of inter-node business communication, ensured constant speed of failure detection but is not sustainable enough for some systems.

This protocol is highly suitable for systems where failures are rare, or messages are frequently sent, while sustainability is strongly emphasized, especially in energy-sensitive scenarios. It also fits systems that can tolerate sending messages to non-group members, with the nodes themselves responsible for message filtering.

We aim to design membership protocol systems in this category, contrasting traditional classifications and inspiring system architect's sustainability design. Moreover, this protocol can be the group communication construct of a new distributed functional programming language.

This membership protocol includes three main algorithms: the actor group joining algorithm, the lazy failure detection algorithm, and the group connect algorithm.

SWIM uses an infection-style dissemination mechanism because hardware multicast and IP multicast are infrequently enabled on most networks and operating systems due to administrative reasons. That is why the basic SWIM protocol can only utilize a costly broadcast or an inefficient point-to-point messaging scheme [4].

This protocol is designed based on a scalable, reliable, connectionless, and efficient underlying actor model communication construct that supports broadcast and point-to-point messaging, such as the Uactor, a distributed communication system we are currently developing. We have presented and submitted a tutorial about it at the SusTrainable Summer School 2022 [12].

In the following sections, first, we introduce the actor group joining algorithm, including its TLA+ specification [7], model check results, and energy consumption estimation of two design branches; next, the lazy failure detection algorithm is explained; afterward, the group connect algorithm is presented; next, the related works are discussed; finally, the conclusion is provided with the future works.

2 ACTOR GROUP JOINING ALGORITHM

This algorithm specification assumes that:

- (A1) The receiver will eventually receive the message as long as it is active

This algorithm has been designed for the actor model group, considering using a connectionless transport protocol.

The communicator module provides a certain level of reliability in message transportation. Consequently, under the defined specifications, it can be reasonably assumed that the receiver will eventually receive the message as long as it remains active. However, it is essential to note that in practical implementation, certain factors, such as the number of resend attempts, may impose limitations and prevent the fulfillment of the assumption.

Here are the requirements of the algorithm specification:

- (R1) Liveness property: Once a node becomes a group member, it will be included in all the not failed members' local group tables (except itself).
  All the group members are a subset or equal to the local group structure table of all the group members that are not failed. The local group structure tables can have more node addresses (or node IDs). When a group member node broadcasts a message to a group, it is allowed in this system to send messages to nodes that are not in the group. The nodes not in the group are responsible for ignoring the noise. However, when a group member broadcasts a message to the group, it should try to send it to all the existing group members.
  This requirement also includes the new member. So, the group structure sent back from the introducer to the new group member contains all the existing group members. Then, we need the lock to ensure that, at the same time, only one new node is added to the group structure table of all the existing group members. If there is no lock and two new nodes, A and B, are joining the group at the same

time, the group structure replied by their introducer may not contain each other. (A does not know B, B does not know A).

- (R2) Decentralization, independence of nodes, and adherence to the philosophy of the actor model.
  We do not use shared resources (shared variable, shared database) and do not use one node or several nodes to maintain the group structure. All the group nodes keep their own local group structure table. It is also for the scalability consideration (no central bottleneck) and for reliability consideration (inherently free of a single point of failure). This requirement gives rules for the specification. Firstly, each node performs computations and sends messages based solely on its local states without using shared variables. In the specification, a node's actions may assist in modifying imagination variable members. However, the node will not utilize the values of these members to calculate or send messages. This variable serves only to aid in utilizing the TLC to check whether the specification meets certain invariants and liveness properties. Accessing other nodes' states to check if it is failed is a simplified fail detection process and does not violate this rule. Secondly, it is necessary to specify the mailbox of nodes and the asynchronous communication among the nodes.
- (R3) Reliability: the design considers the failure of a node and message delays.
- (R4) Sustainability: the system decreases the number of messages and the hops of each message.

2.1 Specification

The complete specification can be found in Appendix A.

The specification includes three constants: *Nodes*, *InitMembers*, and *MaxClock*. *Nodes* represent all the nodes, including both the already included in the group and the nodes that have not yet joined. *InitMembers* represent the nodes already present in the group, and *MaxClock* represents the maximum value of the clock.

We can understand *Nodes* as the Node IDs for each node. The exclusion of 0 is to reserve a zero value, and the exclusion of 1 prevents TLC from automatically converting the function set into a sequence.

When running TLC to check the TLA+ specification, we must provide constant values to TLC. For example, *Nodes* would have a value of 2, 3, 4, 5, *InitMembers* would have a value of 2, 4, and *MaxClock* would have a value of 5. In the assumption section, we impose certain conditions on the constants. TLC will report an error if the provided constant values do not satisfy these conditions.

Members have six states: *unlock*, *lock*, *introducer*, *crit1*, *crit2*, and fail. In addition to the member states, Node states include three additional states: *notInGroup*, *joining*, and *failedJoin*.

The specification consists of three variables: *members*, *localInfos*, and *mailboxes*. *TypeOK* is the first invariant in the specification. By checking this invariant, we can ensure that the three variables do not have unexpected values throughout the model-checking process.

The *members* is an imagination variable. It represents a set of nodes that includes all the nodes that have joined the group. As indicated in the first line of the *TypeOK* definition, *members* are a subset of natural numbers (a natural number represents each node). There is no need to directly implement this variable in the actual implementation of software that conforms to this specification. It serves solely to assist in verifying certain variables and properties. Each action does not perform calculations based on the value of this variable, but it assists in updating this value. Only during the initialization process, for convenience, we use the values of *members* to set the initial local group table of members. The *localInfos* function maps each node to its local information. *[Nodes -> LocalInfo]* represents the set of all functions with *Nodes* as the domain and *LocalInfo* as the range. *Nodes* is the set that needs to be provided when running TLC. *LocalInfo* is a record set where the text before the colon represents the field name, and the text after the colon represents the range of that field. A record is a special type of function with a domain of strings.

LocalInfo contains seven fields, including *nodeState*, which records the state of the node, *lockId*, which stores the corresponding *lockId* (e.g., the *introducer* will store its *lockId*, and other members that are locked by a node will also store the corresponding *lockId*). They will only execute certain actions (e.g., unlock message and operation message) after confirming the *lockId*. The clock records the number of times the local node attempts to lock other nodes. The *inMemory* is used to remember a node; for example, the *introducer* remembers which node's join request message it is currently processing, or a new member sending a join request message remembers which node it sent the message to (who its *introducer* is).

The clock is the local logical clock, but other nodes will not update their clocks upon receiving it. It is also not used as part of the distributed version. This is because this system does not require a total ordering of all system events. The clock differentiates between different lock requests from the same *introducer*.

When accessing *localInfos*, the value of *localInfos[m]* represents the local information of node m. Moreover, the *localInfos[m]["nodeState"]* value represents the state of node *m*. *Ack* records which members have already sent lock acknowledgment messages to the local node. *Opack* records which members have sent operation acknowledgment messages or group structure acknowledgment messages to the local node. *groupTable* is the local communication table of the node. Ideally, when no new members are joining, a member's group table contains all group members except itself.

Mailboxes represent the mailboxes of all nodes. It maps each node to its corresponding mailbox. We use a sequence of messages to simulate the mailbox. Similar to a record, a sequence is also a special type of function with a domain from 1 to n. The specification for mailboxes addresses our consideration for requirement R2.

There are a total of 9 types of messages. Some message types contain a node, some contain a set of nodes, and some contain a *lockId*, where a *lockId* is an element of a set *LockID*. *LockID* is a tuple set that includes the fields *intro* and *clock*. *intro* field records who owns this lock, and the *clock* field records the number of times this *introducer* has attempted to lock.

Init is the first action executed during TLC runtime, assigning initial values to the three variables. If a node is included in *InitMembers*, its state is *unlock*, and its group table consists of all nodes in *InitMembers* except itself. If a node is not included in *InitMembers*, its state is *notInGroup*, and its group table is empty. The initial value of the Clock is 1. The values of *InMemory* and *lockId* are their corresponding zero values. The *ack* and *opack* are both empty sets.

The function *RemoveByIndex* allows us to remove an element from a sequence based on its index, and the result is the updated sequence after the removal. We will use this function to remove messages that have been processed.

The sub-action *SendJoinRequest* involves two nodes, represented by the parameters *n* and *i*. Here, *n* refers to the new node, and *i* refers to the *introducer*. There are two enable conditions for this action. The first condition is that the state of n is *notInGroup*, and the second is that i is *unlock*. During implementation, the new member (n) cannot determine the state of the *introducer* (i). One possible implementation approach is to reject or ignore the join request immediately if the *introducer*'s state is not *unlocked*. However, to simplify the specification, we have not included this detail and instead rely on the enable conditions to determine when the action can occur. In this action, we modify the values of the variables *localInfos* and *mailboxes* in the next state. The character ' indicates that it is the next state. We change the state of *n* to *joining* and store the *introducer*'s node ID in the *inMemory* field. Then, we send a join request message to the *introducer*'s mailbox. The *sender* field in the message contains the node ID of n. Due to the A1 assumption, we do not simulate the network but instead directly add the message to the recipient's mailbox. The variable members remain unchanged.

The sub-action *RecvGroupStruct* has three enable conditions:

- The local node n is in the *joining* state.
- There is a *groupstruct* message in the node's mailbox.
- The *intro* field of the *groupstruct* message matches the introducer node ID stored in the node's memory.

Once the enable conditions are satisfied, this sub-action will modify the *localinfo* variable in the next state. It changes the state of the node itself to *unlock* and updates its local group table set based on the *currentMembers* field in the *groupstruct* message. Then, it resets the *inMemory* field to its default zero value. After this sub-action, the node considers itself to have joined the group. In addition, this sub-action also modifies the *mailboxes* to simulate message exchange. It sends a group structure acknowledgment message to the *introducer*, informing them it has updated its state. It also removes the processed *groupstruct* message from the mailbox. Considering R2, in this specification, when receiving a message, it uses the condition "exists a message in the mailbox" instead of retrieving the mailbox's header to create an environment where message order is intentionally made unpredictable. However, in implementation, messages are typically processed one by one or using pattern matching. The unordered transportation of messages may result in messages sent earlier arriving later in the mailbox. Implementing the enable condition for messages can involve checking each message and determining if it satisfies the enabling conditions for a specific sub-action. The message is kept in the mailbox if the conditions are not met.

Similar to the *RecvGroupStruct* sub-action described above, the *RecvJoinFail* sub-action states that if a node in the *joining* state receives a *joinfail* message from the corresponding *introducer*, it will modify its state to *failedJoin*, set the *inMemory* value to zero, and remove the processed message from the mailbox. We assign a separate state for the node that failed to join to reduce the number of states generated during TLC checking. If the failed node were to return to the *notInGroup*state, it could continue sending join requests to other nodes. While this is closer to real-world scenarios, a node that fails to join will likely attempt to rejoin or try joining with a different *introducer*. However, allowing such retries would significantly increase the computational complexity of the TLC model checking. Therefore, to reduce the complexity of the model calculation, we eliminate the opportunity for the node that failed to join to attempt joining again.

The *JoinReqTimeout* sub-action describes a scenario where a node in the *joining* state, and its *inMemory* variable is a non-zero value (indicating the existence of an *introducer*), detects that the corresponding *introducer* has failed. In such a case, the node will modify its state in the next state to *failedJoin*. The check for *introducer* failure in this context simplifies the failure detection process. When a node's join request times out, the failure detection process can be initiated in an actual implementation. This process typically involves message exchange and the judgment of suspicious nodes by multiple nodes, including group members. Based on the voting results, the failed status of the suspicious node is determined. Upon detecting the failure of the *introducer*, the node will abandon this action. If group members participate in the failure detection process, they will notify other group members to remove the failed node from their local group table. The design consideration for handling failures reflects our consideration of requirement R3. The TLC model checking process also considers message delays by specifying sub-actions for receiving different messages. TLC exhaustively explores all possible executions, so even if a message arrives early, it may be received (or handled) much later when multiple sub-actions are enabled simultaneously.

The *RecvJoinRequest* sub-action specifies the regular action of the *introducer* handling a join request. The action runs under the following conditions: the node *m*'s node state is *unlock*; there is a join request message in the mailbox of node *m*. After running this sub-action, in the next state, the node state of 'm' will become *introducer*. The *lockId* will contain the node ID of the current node and the current clock value, where the clock value is incremented by 1. The value of *inMemory* will be set to the value of the *sender* field in the join request message (the introducer *m* will remember which node is requesting to join the group). Finally, we will modify the value of the mailbox in the next state, *m* will send a lock request message to all nodes in its local group table, requesting to lock them. Additionally, the processed join request message will be deleted from the mailbox of *m*.

The *RecvLockAck* sub-action is about a node handling a lock acknowledge message. The enable condition does not specify that only the *introducer* can satisfy this enable condition. This is because, to minimize messages remaining in mailboxes, even an *introducer* that has given up locking other members may receive delayed lock acknowledgments (with a different *lockID* than its current *lockID*). In such a case, the node 'm' will delete this message. If the lock acknowledgment message's *lockId* matches the current *lockID* of the node, the node will add the sender to its *ack* set, which indicates that the node is now aware that the sender has acknowledged it.

In the *RecvAllLockAck* sub-action, if the node *m* is in the state of an *introducer* and the ack set of node *m* contains all the non-failed nodes in its local group table, it means that *m* as an *introducer* has received lock acknowledge messages from all the other group members who have not failed. In the next state, node *m* will transition to the *crit1* state, indicating that it has entered the critical section. The assessment of failures here simplifies the process of fail detection (through message exchanges) into checking directly whether the other nodes have failed.

In this algorithm, multiple nodes can become *introducer*s. However, according to requirement R1, we design it so that only one *introducer* can enter the critical section, which requires some *introducer*s to relinquish the lock on other nodes. During implementation, the developer can consider adopting the approach of random timeouts, similar to the RAFT algorithm [10]. Alternatively, the developer can follow the approach described in this specification, where the node ID is the criterion for deciding whether to give up locking other group members.

In the enable condition of this sub-action, the first condition is that node *m* is in the state of an introducer. The second condition is that node *m* has received less than half of the lock acknowledgments (without considering the failure cases). We do not want *introducers* who have already received many lock acknowledgments to give up the lock. The third condition is that the *introducer*'s mailbox already contains a lock request message from another node (indicating that another node wants to lock node *m*). The final condition is that the lock request message received contains a lock ID with a sender's node ID greater than the node *m*'s node ID, indicating that we prefer introducers with smaller node IDs to relinquish the lock. This final condition can be replaced with another comparable value based on the actual scenario and performance testing as long as it is unlikely to be the same among different nodes. For example, we have compared the clock values to determine whether an introducer should give up. However, it resulted in excessive states during TLC model checking. Therefore, we chose to use node IDs instead.

If all the conditions are satisfied, the sender of the lock request will lock node *m* in the next state. The node state of Node *m* will become *lock*, the lock ID will be updated according to the lock request, and *inMemory* will be set to zero. The introducer no longer needs to remember which node's join request it was processing before relinquishing the lock.

Node *m* will also make several changes to the next state of its mailbox:

- It will inform all existing members that it has relinquished the attempt to lock other nodes by sending them an unlock message (including the corresponding lock ID). This way, if the sender later locks node *m*, it can process the unlock message to unlock itself.
- Node *m* will notify the sender of the lock request that it has been locked by sending a lock acknowledgment message to the sender.
- Node *m* will delete the lock request message from its mailbox once it has been processed (this step is repeated in many subsequent sub-actions and will not be introduced repeatedly in the following explanations).

CritOp specifies that if a node's node state is *crit1*, it will send an Operation message to all nodes in its local group table and a group structure message (containing all the current nodes in the group) to the newly joined node. Afterward, this node will modify its node state to *crit2*. (rather than strictly following the mathematical semantics of TLA+, I have explained it as a program to make it easier to understand. Most English explanations of specifications are written for readers who are not familiar with TLA+. People familiar with TLA+ usually find it clearer and more accurate to read the specification rather than the English explanation directly.)

As defined in *CritRecvOpAck*, when a node's node state becomes *crit2*, if there are operation messages or group structure acknowledge messages in its mailbox, and these messages are relevant to the current state of node m, the sender's node ID associated with the acknowledgment (*ack*) will be stored in node m's *opack* set.

As defined in *CritExit*, suppose node m's node state is *crit2* and it has received operation acknowledgments from all other group members as well as group structure acknowledgments from the new member (m's *opack* set includes the node IDs of all non-failed nodes in m's local group table and the new node). In that case, node m will exit the critical section in the next state. The new member will be added to *m*'s group table, and all other local information will be restored to its initial values. The operation of filtering the group table for nodes whose state is not *fail* in this action is also a simplification of the failure detection process. The new member is also added to the imagination variable *members* in this action.

The sub-action *RecvLockReq* specifies that if a node m is in the *unlock* state and receives a lock request message, this node will be locked by the sender of the lock request message. The node's node state will be modified to *lock*, and its local lock ID will be updated according to the *lockId* of the message. Additionally, the node will send a lock acknowledgment message to the sender of the lock request message.

If a node in the *lock* state receives an unlock message and the Lock ID in the unlock message is relevant to the current lock ID of node m, then node m will unlock (change its state to *unlock*) and restore the lock ID to a zero value.

A node in the *lock* state has a timeout mechanism that checks whether the corresponding introducer has failed. If the introducer is detected to be failed after the timeout, the node will automatically unlock.

Upon receiving an operation message, a node in the *lock* state will also check the lock ID in the message. If the lock ID matches its local lock ID, the node will add the new node to its local group table. It will then restore its state to the initial *unlock* state and send an operation ack message to the sender of the operation message. In order to satisfy the requirement R4 for sustainability and minimize message overhead, we allow nodes to unlock directly after processing an operation message rather than having the introducer notify everyone to unlock after receiving all operation acks. Since a node that has finished processing the operation message is in the *unlock* state, it may become an introducer before another introducer exits the critical section. However, this is not a problem as we ensure that only one introducer enters the critical section (reaches the *crit1* state).

The action *NodeFail* convert a node into the *fail* state. In this specification, no action can restore a failed node.

The *Next* action contains all the previously introduced sub-actions. Most actions are quantified based on the existence of a member. Actions that may occur for the nodes that are not in the *members* set include RecvGroupStruct, RecvJoinFail, and JoinReqTimeout. SendJoinRequest involves two quantifiers: one for the member (introducer) and another for non-member nodes (new nodes). Any member or node joining may fail (we are not concerned with the failure of other nodes).

The temporal operator WF specifies the action within parentheses as weak fairness. The fairness specified in the specification guides TLC in checking the specification (for example, if WF is specified, TLC will not deliberately avoid executing the action if it remains continuously enabled).

The specification Spec defines the initial state, next actions, and fairness of the state machine.

Inv1_Mutex specifies the mutex-related invariant, which is also a security property. It states that the number of nodes in the *crit1* node state in the (group) members is less than or equal to 1. *Prop1_LocalTable* specifies the liveness property concerning local tables of group members. If node n becomes a member for all nodes, it should appear in the local group table of all non-failed group members. *Inv1_Mutex* and *Prop1_LocalTable* are specifications for requirement R1. TLC can check if the specification satisfies these two properties.

2.2 Removing nodes

Removing nodes (whether leaving the group or being detected as failed) is more straightforward than joining. All that is required is to send update messages to the list of nodes. We do not need to consider the updates of the local group table of the node being removed. Unlike the join process, there's no need to consider the distributed mutual exclusion problem. For instance, if a group nearly simultaneously removes nodes 'a' and 'b', other working nodes will receive messages about removing both 'a' and 'b' (as stated in A1). Even if 'a' does not receive the message about 'b' being removed, it doesn't matter. After all, 'a's entire local group table should be deleted or considered invalid. This approach further enhances the speed of membership update dissemination.

2.3 Model check results

This section presents the TLC model check results of the specification.

The machine we used in this test has the following equipment specifications: processor, AMD Ryzen 7 5800H with Radeon Graphics 3.20 GHz; RAM, 16.0 GB; Os, Windows 10.

To run the TLC model checker, we need to provide the value of the constants. For the constant *MaxClock* we have given a value of 5 throughout the model check process. For the constants *Nodes* and *InitMembers*, we provide the following combinations for the test: (1) 3n2i; (2) 3n1i; (3) 4n3i; (4) 4n2i; (5) 4n1i; (6) 5n4i; (7) 5n3i; (8) 5n2i. ("(2) 3n1i" means the configuration ID is 2. There are three elements in the *Nodes* set and one element in the *InitMembers* set.)

For the initial nodes, we have tied different node ID combinations. For example, for two initial nodes, we tried nodes 2 and 3, and nodes 2 and 3. We found they have the same number of states. Therefore we do not draw distinct points in the results picture for those combinations.

The test is meaningless if the number of initial members is zero (no one is initially in the group) or equal to the nodes set (All the

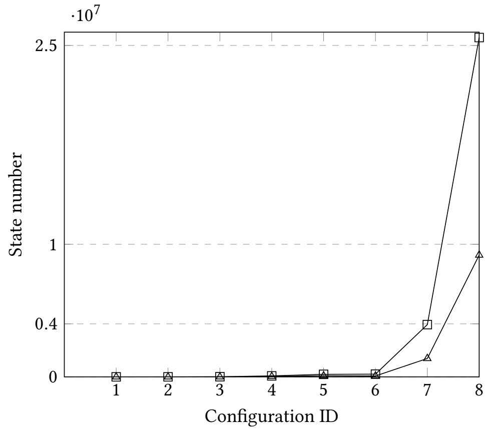

Figure 1: TLC model check results with fail action. The line with square marks is the number of states generated. The line with triangle marks is the number of distinct states found

nodes are already in the group). Therefore, we do not test those combinations.

The *5n1i* combination is also not included since the TLC model check of this combination with the *NodeFail* sub-action is beyond the memory limit of the machine for this test.

Before each test, we include the formulas *TypeOK* and *Inv1_Mutex* as the invariant of the model and include the formula *Prop1_LocalTable* as the 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). We will uncheck the deadlock test because TLC treats the stopping as deadlock, and our specification can stop (all nodes joined or failed).

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.

All the results we show below have confirmed by the TLC that the model checking was completed, and no error has been found.

As shown in Figure 1, the number of states is more likely to increase as the nodes increase. 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 bring more possible situations.

As shown in Figure 2, If we comment out the *NodeFail* sub-action (do not consider the fail situation), the number of the state decreases compared to the results in Figure 1. Since the *fail* sub-action will bring more possibilities to the model-checking process.

2.4 Energy consumption estimation

To enhance the sustainability of the protocol, we divided the Actor group joining algorithm into two distinct design branches, each represented by TLA+ specifications that only consider ideal operational scenarios (*upload and cite the link later*). We then implemented simplified versions of these two branches using the Go programming language (*upload and cite the link later*). Under controlled variables and on a single machine (the same machine that ran the

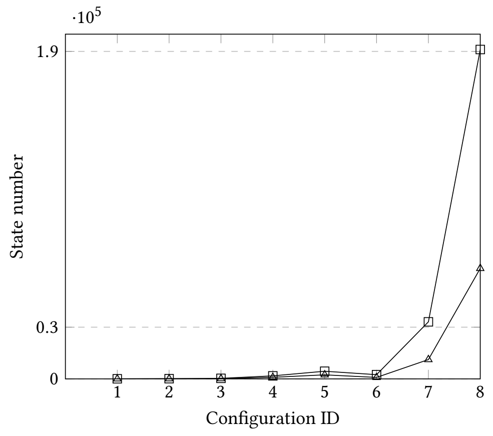

Figure 2: `TLC model check results without fail action.`

model checking test), we utilized AMD uProf to obtain energy consumption estimation results for the implementations of both design branches.

We got the estimation values of the implementations by running the following command after running the programs and summing up the average power of all the cores and sockets:

```
AMDuProfCLI.exe timechart --eventpower
-o C:\Temp\Poweroutput --interval 500
--duration 50
```

The results for Design Branch A, which typically has a larger message size but fewer messages, were: 1673.14 Watts, 1652.66 Watts, and 1671.36 Watts. Design Branch B, generally characterized by a smaller message size and a larger number of messages, got the results: 1713.29 Watts, 1717.74 Watts, and 1718 Watts.

Subsequently, the actor group joining algorithm adopted Design Branch A.

3 LAZY FAILURE DETECTION ALGORITHM

The liveness property of the lazy failure detection algorithm: As long as a node communicates with other working nodes (where these working nodes send messages to it), if this node fails, its failure status will eventually be known by all other functioning nodes.

Under the sustainable consideration, in an actor model system, an actor does not need to be concerned with another node's failure status with no message interactions. In other words, this algorithm permits the failure of permanently isolated nodes to go undetected.

This algorithm does not use a heartbeat and gossip mechanisms to achieve the "lazy" characteristic. After node 'a' sends a message to another node 'b', if an acknowledgment from 'b' is not received within a specific duration 'n', 'a' waits an additional time period 'm':

- If, before the end of 'm', 'a' does not send messages to any other node besides 'b', it initiates the failure confirmation process for 'b'. During this process, messages are sent to other nodes. If none of them responds, 'a' should consider the possibility that it may have been excluded from the group. (The failure confirmation process involves attempting to contact more nodes in the group table. Based on the received acks, 'a' can determine whether it has been excluded from the group.)
- If, before the end of 'm', 'a' sends messages to nodes other than 'b', and more than x% of them do not send back the acknowledgment, then 'a' should begin to evaluate whether it has been excluded from the group.
- If, before the end of 'm', 'a' sends messages to nodes other than 'b', and less than x% of them do not send back the acknowledgment, then 'a' initiates the failure confirmation process for 'b'.

When a node receives a message from the group, it checks whether the sender is a group member. If the sender is not a recognized member, the node, to further reduce message traffic and enhance sustainability, will ignore the message rather than send a refusal response.

Consequently, Nodes confirmed as failed, even if wrongly identified or if they recover after a failure, must rejoin the group. This is due to their removal from the local group table by other group members.

4 GROUP CONNECT ALGORITHM

The group connect algorithm is designed to improve scalability. Within each group, certain nodes are dynamically chosen to act as "diplomatic nodes" (or "diplomatic actors"). The responsibilities of these diplomatic nodes include:

- Receiving group update information from the diplomatic nodes of other groups and maintaining a diplomatic communication table containing information on the other groups' diplomatic nodes.
- Forwarding the received group information to members of their own group.
- Notifying the diplomatic nodes of other groups about changes within their own group.

5 RELATED WORKS

The Actor group joining algorithm's primary distinction from other solutions addressing distributed mutual exclusion lies in its requirements:

Lamport's distributed mutual-exclusion algorithm [9] is a prominent solution often considered for addressing the mutual exclusion and synchronization in distributed systems. However, upon reviewing the algorithm, we have discovered that its assumptions and requirements do not align with our specific scenario. One critical assumption of the algorithm states, "For any two processes Pi and Pj, the messages sent from Pi to Pj are received in the same order as they are sent." Unfortunately, our system cannot fulfill this assumption, as we have designed it based on the actor model and connectionless transportation protocol to improve scalability. Moreover, the original 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 other words, we do not need to adhere to the condition that "different requests for the resource must be granted in the order in which they are made." In our scenario, it is possible for the introducer who made the earlier request to give it up. Although we still require mutual exclusion to ensure only one introducer enters the critical section, we do not require enforcing a specific order.

In our design, to ensure the independence of nodes, each node has a local group table that contains information about group member nodes' addresses or node IDs, excluding itself. Therefore, as stated by R1, we also need to ensure the distributed consistency of local group tables, although the consistency requirement is low (allowing for redundant nodes). Because distributed consistency is involved, we considered three classic algorithms: 2PC [6], Paxos [8], and Raft [10]. However, we found that these algorithms do not match our system requirements. Compared to the 2PC algorithm: Members in this system do not need to abort. In this algorithm, to reduce message overhead, the introducer (or coordinator) does not send additional messages to all members after receiving confirmation messages from other members (or acceptors). Additionally, unlike 2PC, this algorithm does not send the operation content to members during the lock phase (or Prepare Phase in 2PC) to prevent members from mistakenly executing operations in advance and to reduce message size. Furthermore, this algorithm solves the blocking problem using timeouts (lock request timeout, lock timeout) and failure detection (when joining timeout, dealing with the acknowledgments), which differs from the approach of introducing a prepare phase in 3PC. Compared to the Paxos and Raft algorithms: This algorithm eliminated the heartbeat mechanism and leader election mechanism to increase system sustainability at the expense of reducing the level of consistency. Additionally, this algorithm is designed specifically for group member joining. By manually joining nodes one by one and waiting for success, we can significantly reduce the number of times this algorithm has multiple introducers and performs calculations on whether to abandon, significantly reducing message overhead and consequently reducing energy consumption. However, if multiple nodes join simultaneously, this algorithm can also handle it and ensure system safety and liveness properties. Raft and Paxos are more suitable for scenarios with many dense requests, such as distributed caching applications. Furthermore, our underlying communicator, designed with a connectionless transportation protocol and actor model, may address the scalability issue of Raft algorithm clusters (this requires further implementation and testing of the algorithm). Furthermore, this algorithm needs to be designed for the actor model. The actor model philosophy needs to be included in the specification to provide more confidence for the system design.

The lazy failure detection algorithm's most significant differentiation from other failure detection algorithms is its "lazy" behavior: not periodically sending ping or heartbeat messages.

The group connect algorithm, similar to sd_erlang [3], leverages group connections to enhance scalability. However, the primary difference is that the sd_erlang uses gateway nodes (intersection nodes) as connection points between groups; the diplomatic nodes in our algorithm are members of a single group and do not belong to two groups simultaneously. Moreover, the S_group internally operates as a fully connected network; our group design is based on a connectionless transportation layer, allowing a larger maximum group size. Additionally, our designed overlying communicator doesn't need virtual machines and global data, plans to offer a variety of communication mechanisms within the group during implementation, and the selection of diplomatic nodes is also dynamic.

The primary distinction between the group connect algorithm, and the hierarchical membership protocol [13] is that our algorithm's groups do not exist in a hierarchy and are not follow a tree structure. Failure detection is also independent. Furthermore, in our model, the diplomatic nodes belong to just one group, and their designation is dynamic.

6 CONCLUSION

In conclusion, middleware systems nowadays primarily use membership protocols like heartbeat-based and gossip-based. Each has its own strengths but also faces challenges in scalability and energy efficiency. The "lazy membership protocol" provides a fresh approach to this challenge. Unlike conventional methods, it avoids regular pings or heartbeats. This focus ensures better energy use, making it ideal for systems where node failures are rare but communication is frequent. We have divided the design of the sustainable protocol into three algorithms: actor group joining algorithm, lazy failure detection algorithm, and group connect algorithm.

We hope our work can inspire system architects to give more consideration to sustainability and provide a theoretical reference for the distributed communication service for the next generation of functional programming languages. Our next steps involve further refining each algorithm's specifications and enhancing the algorithms' correctness, efficiency, and sustainability. Utilizing TLA+ to describe systems can illuminate design flaws. Therefore, the ultimate design adheres to the algorithm specifications. We also intend to verify the protocol's design with additional experimental results of the implementation.

REFERENCES

[1] André Allavena, Alan J. Demers, and John E. Hopcroft. 2005. Correctness of a gossip based membership protocol. In *Proceedings of the Twenty-Fourth Annual ACM Symposium on Principles of Distributed Computing, PODC 2005, Las Vegas, NV, USA, July 17-20, 2005.* Marcos Kawazoe Aguilera and James Aspnes, (Eds.) ACM, 292–301. DOI: 10.1145/1073814.1073871.

[2] Francesco Cesarini and Steve Vinoski. 2016. *Designing for scalability with Erlang/OTP: implement robust, fault-tolerant systems.* " O'Reilly Media, Inc.".

[3] Natalia Chechina, Huiqing Li, Amir Ghaffari, Simon Thompson, and Phil Trinder. 2016. Improving the network scalability of erlang. *Journal of Parallel and Distributed Computing*, 90-91, 22–34. DOI: https://doi.org/10.1016/j.jpdc.2016.01.002.

[4] A. Das, I. Gupta, and A. Motivala. 2002. Swim: scalable weakly-consistent infection-style process group membership protocol. In *Proceedings International Conference on Dependable Systems and Networks*, 303–312. DOI: 10.1109/DSN.2002.1028914.

[5] [n. d.] Index - erlang/otp. https://www.erlang.org/. Accessed: 2023-08-20. ().

[6] Jim Gray and Leslie Lamport. 2004. Consensus on Transaction Commit. Tech. rep. MSR-TR-2003-96. ACM Transactions on Database Systems 31, 1 (2006), 133-160. Microsoft, (Jan. 2004). https://www.microsoft.com/en-us/research/publication/consensus-on-transaction-commit/.

[7] Leslie Lamport. 2002. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley Longman Publishing Co., Inc., USA. ISBN: 032114306X.

[8] Leslie Lamport. 1998. The part-time parliament. *ACM Transactions on Computer Systems 16, 2 (May 1998), 133-169. Also appeared as SRC Research Report 49. This paper was first submitted in 1990, setting a personal record for publication delay that has since been broken by [60].*, (May 1998). ACM SIGOPS Hall of Fame Award in 2012. https://www.microsoft.com/en-us/research/publication/part-time-parliament/.

[9] Leslie Lamport. 1978. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM 21, 7 (July 1978), 558-565. Reprinted in*

several collections, including Distributed Computing: Concepts and Implementations, McEntire et al., ed. IEEE Press, 1984., (July 1978), 558–565. 2000 PODC Influential Paper Award (later renamed the Edsger W. Dijkstra Prize in Distributed Computing). Also awarded an ACM SIGOPS Hall of Fame Award in 2007. https://www.microsoft.com/en-us/research/publication/time-clocks-ordering-events-distributed-system/.

[10] Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference* (USENIX ATC'14). USENIX Association, Philadelphia, PA, 305–320. ISBN: 9781931971102.

[11] [n. d.] Introducing riak core. https://riak.com/posts/business/introducing-riak-core/index.html?p=6033.html. Accessed: 2023-08-20. ().

[12] [n. d.] Sustrainable summer school 2022. https://sustrainable.uniri.hr/. Accessed: 2023-08-20. ().

[13] P. D. V. van der Stok, M. M. M. P. J. Claessen, and D. Alstein. 1994. A hierarchical membership protocol for synchronous distributed systems. In *Dependable Computing — EDCC-1*. Klaus Echtle, Dieter Hammer, and David Powell, (Eds.) Springer Berlin Heidelberg, Berlin, Heidelberg, 599–616. ISBN: 978-3-540-48785-2.

A THE SPECIFICATION OF ACTOR GROUP JOINING ALGORITHM

```
──────────────────────── MODULE GroupSpec31 ────────────────────────
EXTENDS Integers, Sequences, FiniteSets

CONSTANT Nodes, InitMembers, MaxClock

ASSUME
   ∧ Nodes ⊆ (Nat \ {0, 1})
   ∧ InitMembers ⊆ Nodes
   ∧ MaxClock ∈ Nat

MemberStates ≜ {"unlock", "lock", "introducer", "crit1", "crit2", "fail"}

NodesStates ≜ MemberStates ∪ {"notInGroup", "joining", "failedJoin"}

VARIABLES members, localInfos, mailboxes

vars ≜ ⟨members, localInfos, mailboxes⟩

Clock ≜ 0 . . MaxClock

NodesWithZero ≜ Nodes ∪ {0}

LockID ≜ [intro : NodesWithZero, clock : Clock]

Message ≜
[type : {"lockreq", "unlock"}, lockId : LockID]
∪
[type : {"gsack", "joinreq", "joinfail"}, sender : Nodes]
∪
[type : {"groupstruct"}, currentMembers : SUBSET Nodes, intro : Nodes]
∪
[type : {"lockack", "opack"}, lockId : LockID, sender : Nodes]
∪
[type : {"operation"}, lockId : LockID, newMember : Nodes]

LocalInfo ≜ [nodeState : NodesStates, lockId : LockID, clock : Clock,
inMemory : NodesWithZero, ack : SUBSET Nodes, opack : SUBSET Nodes,
groupTable : SUBSET Nodes]

ZeroLockID ≜ [intro ↦ 0, clock ↦ 0]

TypeOK ≜
   ∧ members ⊆ Nat
   ∧ localInfos ∈ [Nodes  → LocalInfo]
   ∧ mailboxes ∈  [Nodes → Seq(Message)]

Init ≜
   ∧ members = InitMembers
   ∧ localInfos = [p ∈ Nodes ↦ [nodeState ↦
     IF p ∈ InitMembers THEN "unlock" ELSE  "notInGroup",
     lockId ↦ ZeroLockID, clock ↦ 1, inMemory ↦ 0,  ack ↦ {},
     opack ↦ {},
     groupTable ↦
        IF p ∈ InitMembers THEN InitMembers \ {p} ELSE  {}]]
   ∧ mailboxes = [q ∈ Nodes ↦ ⟨⟩]

RemoveByIndex(s, i) ≜
  SubSeq(s, 1, i − 1) ∘ SubSeq(s, i + 1, Len(s))

 New member
```

```
SendJoinRequest(n, i) ≜
  ∧ localInfos[n][“nodeState”] = “notInGroup”
  ∧ localInfos[i][“nodeState”] = “unlock”
  ∧ localInfos′ = [localInfos EXCEPT ![n][“nodeState”] = “joining”,
     ![n][“inMemory”] = i]
  ∧ mailboxes′ = [mailboxes EXCEPT ![i] =
     Append(@, [type ↦ “joinreq”, sender ↦ n])]
  ∧ UNCHANGED ⟨members⟩

RecvGroupStruct(n) ≜
  ∧ localInfos[n][“nodeState”] = “joining”
  ∧ ∃ i ∈ 1 .. Len(mailboxes[n]) :
     LET
         Msg ≜ mailboxes[n][i]
         Gsack ≜ [type ↦ “gsack”, sender ↦ n]
     IN
         ∧ Msg[“type”] = “groupstruct”
         ∧ Msg[“intro”] = localInfos[n][“inMemory”]
         ∧ localInfos′ = [localInfos EXCEPT
            ![n][“nodeState”] = “unlock”,
            ![n][“groupTable”] = Msg[“currentMembers”],
            ![n][“inMemory”] = 0]
         ∧ mailboxes′ = [mailboxes EXCEPT ![Msg[“intro”]] =
            Append(@, Gsack), ![n] = RemoveByIndex(@, i)]
         ∧ UNCHANGED ⟨members⟩

RecvJoinFail(n) ≜
  ∧ localInfos[n][“nodeState”] = “joining”
  ∧ ∃ i ∈ 1 .. Len(mailboxes[n]) :
     LET
         Msg ≜ mailboxes[n][i]
     IN
         ∧ Msg[“type”] = “joinfail”
         ∧ Msg[“sender”] = localInfos[n][“inMemory”]
         ∧ localInfos′ = [localInfos EXCEPT
            ![n][“nodeState”] = “failedJoin”,
            ![n][“inMemory”] = 0]
         ∧ mailboxes′ = [mailboxes EXCEPT ![n] = RemoveByIndex(@, i)]
         ∧ UNCHANGED ⟨members⟩

JoinReqTimeout(n) ≜
  ∧ localInfos[n][“nodeState”] = “joining”
  ∧ localInfos[n][“inMemory”] ≠ 0
  ∧ localInfos[localInfos[n][“inMemory”]][“nodeState”] = “fail”
  ∧ localInfos′ = [localInfos EXCEPT ![n][“nodeState”] = “failedJoin”]
  ∧ UNCHANGED ⟨members, mailboxes⟩

 Introducer
RecvJoinRequest(m) ≜
  ∧ localInfos[m][“nodeState”] = “unlock”
  ∧ ∃ i ∈ 1 .. Len(mailboxes[m]) :
     LET
         Msg ≜ mailboxes[m][i]
         C ≜ localInfos[m][“clock”]
         LoID ≜ [intro ↦ m, clock ↦ C]
         LockReqM ≜ [type ↦ “lockreq”, lockId ↦ LoID]
     IN
         ∧ Msg[“type”] = “joinreq”
```

```
        $\land\ localInfos' = [localInfos$ EXCEPT
          $![m][$“nodeState”$] =$ “introducer”,
          $![m][$“lockId”$] = LoID,\ \ ![m][$“clock”$] = @ + 1,$
          $![m][$“inMemory”$] = Msg[$“sender”$]]$
        $\land\ mailboxes' = [[r \in Nodes \mapsto$
          IF $r \in localInfos[m][$“groupTable”$]$
           THEN $Append(mailboxes[r], LockReqM)$
           ELSE $mailboxes[r]]$ EXCEPT $![m] = RemoveByIndex(@, i)]$
        $\land$ UNCHANGED $\langle members \rangle$

$RecvLockAck(m) \triangleq$
  $\land\ localInfos[m][$“nodeState”$] \in (MemberStates \setminus \{$“fail”$\})$
  $\land\ \exists\, i \in 1 \,..\, Len(mailboxes[m]):$
    LET
        $Msg \triangleq mailboxes[m][i]$
        $LoID \triangleq Msg[$“lockId”$]$
        $UnlockM \triangleq [type \mapsto$ “unlock”, $lockId \mapsto LoID]$
    IN
        $\land\ Msg[$“type”$] =$ “lockack”
        $\land\ LoID[$“intro”$] = m$
        $\land$ IF $LoID = localInfos[m][$“lockId”$]$
            THEN $localInfos' = [localInfos$ EXCEPT
              $![m][$“ack”$] = @ \cup \{Msg[$“sender”$]\}]$
            ELSE UNCHANGED $\langle localInfos \rangle$
        $\land\ mailboxes' = [mailboxes$ EXCEPT
          $![m] = RemoveByIndex(@, i)]$
        $\land$ UNCHANGED $\langle members \rangle$

$RecvAllLockAck(m) \triangleq$
  $\land\ localInfos[m][$“nodeState”$] =$ “introducer”
  $\land\ localInfos[m][$“ack”$] =$
    $\{x \in localInfos[m][$“groupTable”$]:$
    $localInfos[x][$“nodeState”$] \neq$ “fail”$\}$
  $\land\ localInfos' = [localInfos$ EXCEPT $![m][$“nodeState”$] =$ “crit1”$]$
  $\land$ UNCHANGED $\langle members, mailboxes \rangle$

$LockReqTimeout(m) \triangleq$
   $\land\ localInfos[m][$“nodeState”$] =$ “introducer”
   $\land\ Cardinality(localInfos[m][$“ack”$]) \leq$
     $Cardinality(localInfos[m][$“groupTable”$]) \div 2$
   $\land\ \exists\, i \in 1 \,..\, Len(mailboxes[m]):$
     LET
         $UnlockM \triangleq [type \mapsto$ “unlock”,
           $lockId \mapsto localInfos[m][$“lockId”$]]$
         $JoinF \triangleq [type \mapsto$ “joinfail”, $sender \mapsto m]$
         $NewM \triangleq localInfos[m][$“inMemory”$]$
         $Msg \triangleq mailboxes[m][i]$
         $NewLoID \triangleq Msg[$“lockId”$]$
         $LAck \triangleq [type \mapsto$ “lockack”,
           $lockId \mapsto NewLoID, sender \mapsto m]$
     IN
         $\land\ Msg[$“type”$] =$ “lockreq”
         $\land\ NewLoID[$“intro”$] > m$
         $\land\ localInfos' = [localInfos$ EXCEPT
           $![m][$“nodeState”$] =$ “lock”,
           $![m][$“lockId”$] = NewLoID,$
           $![m][$“inMemory”$] = 0]$
        $\land$ $mailboxes' = [[r \in Nodes \mapsto$
          IF $r \in localInfos[m]$[“groupTable”]
          THEN $Append(mailboxes[r],\ UnlockM)$
          ELSE $mailboxes[r]]$ EXCEPT
          $![NewM] = Append(@,\ JoinF)$,
          $![NewLoID$[“intro”]$] = Append(@,\ LAck)$,
          $![m] = RemoveByIndex(@,\ i)]$
        $\land$ UNCHANGED $\langle members \rangle$

$CritOp(m) \triangleq$
  $\land$ $localInfos[m]$[“nodeState”] = “crit1”
  $\land$ LET
        $Gs \triangleq [type \mapsto$ “groupstruct”,
          $currentMembers \mapsto localInfos[m]$[“groupTable”] $\cup \{m\}$,
          $intro \mapsto m]$
        $NewM \triangleq localInfos[m]$[“inMemory”]
        $Op \triangleq [type \mapsto$ “operation”,
          $lockId \mapsto localInfos[m]$[“lockId”], $newMember \mapsto NewM]$
      IN
        $\land$ $mailboxes' = [[r \in Nodes \mapsto$
          IF $r \in localInfos[m]$[“groupTable”]
          THEN $Append(mailboxes[r],\ Op)$
          ELSE $mailboxes[r]]$ EXCEPT
          $![NewM] = Append(@,\ Gs)]$
        $\land$ $localInfos' = [localInfos$ EXCEPT
          $![m]$[“nodeState”] = “crit2”]
  $\land$ UNCHANGED $\langle members \rangle$

$CritRecvOpAck(m) \triangleq$
  $\land$ $localInfos[m]$[“nodeState”] = “crit2”
  $\land$ $\exists i \in 1\,..\,Len(mailboxes[m])$ :
    LET
      $Msg \triangleq mailboxes[m][i]$
      $LoID \triangleq Msg$[“lockId”]
    IN
      $\land$ $Msg$[“type”] = “opack”
      $\land$ $LoID = localInfos[m]$[“lockId”]
      $\land$ $localInfos' = [localInfos$ EXCEPT
        $![m]$[“opack”] $= @ \cup \{Msg$[“sender”]$\}]$
      $\land$ $mailboxes' = [mailboxes$ EXCEPT
        $![m] = RemoveByIndex(@,\ i)]$
      $\land$ UNCHANGED $\langle members \rangle$

$CritRecvGsAck(m) \triangleq$
  $\land$ $localInfos[m]$[“nodeState”] = “crit2”
  $\land$ $\exists i \in 1\,..\,Len(mailboxes[m])$ :
    LET
      $Msg \triangleq mailboxes[m][i]$
      $Sender \triangleq Msg$[“sender”]
    IN
      $\land$ $Msg$[“type”] = “gsack”
      $\land$ $Sender = localInfos[m]$[“inMemory”]
      $\land$ $localInfos' = [localInfos$ EXCEPT
        $![m]$[“opack”] $= @ \cup \{Sender\}]$
      $\land$ $mailboxes' = [mailboxes$ EXCEPT
        $![m] = RemoveByIndex(@,\ i)]$
      $\land$ UNCHANGED $\langle members \rangle$
```

```
$CritExit(m) \triangleq$
    $\land$ $localInfos[m][\text{"nodeState"}] = \text{"crit2"}$
    $\land$ LET
          $NewM \triangleq localInfos[m][\text{"inMemory"}]$
      IN
          $\land$ $localInfos[m][\text{"opack"}] =$
            $\{x \in localInfos[m][\text{"groupTable"}]$
             $: localInfos[x][\text{"nodeState"}] \neq \text{"fail"}\} \cup \{NewM\}$
          $\land$ $localInfos' = [localInfos$ EXCEPT
            $![m][\text{"nodeState"}] = \text{"unlock"},$
            $![m][\text{"lockId"}] = ZeroLockID,$
            $![m][\text{"inMemory"}] = 0,\ ![m][\text{"ack"}] = \{\},$
            $![m][\text{"opack"}] = \{\},$
            $![m][\text{"groupTable"}] = @ \cup \{NewM\}]$
          $\land$ $members' = members \cup \{NewM\}$
          $\land$ UNCHANGED $\langle mailboxes \rangle$
```

Members

```
$RecvLockReq(m) \triangleq$
    $\land$ $localInfos[m][\text{"nodeState"}] = \text{"unlock"}$
    $\land$ $\exists\, i \in 1 \,..\, Len(mailboxes[m]) :$
      LET
          $Msg \triangleq mailboxes[m][i]$
          $LoID \triangleq Msg[\text{"lockId"}]$
          $LAck \triangleq [type \mapsto \text{"lockack"},$
             $lockId \mapsto LoID,\ sender \mapsto m]$
      IN
         $\land$ $Msg[\text{"type"}] = \text{"lockreq"}$
         $\land$ $localInfos' = [localInfos$ EXCEPT
           $![m][\text{"nodeState"}] = \text{"lock"},\ ![m][\text{"lockId"}] = LoID]$
         $\land$ $mailboxes' = [mailboxes$ EXCEPT
           $![LoID[\text{"intro"}]] = Append(@, LAck),$
           $![m] = RemoveByIndex(@, i)]$
         $\land$ UNCHANGED $\langle members \rangle$

$RecvUnlock(m) \triangleq$
    $\land$ $localInfos[m][\text{"nodeState"}] = \text{"lock"}$
    $\land$ $\exists\, i \in 1 \,..\, Len(mailboxes[m]) :$
      LET
          $Msg \triangleq mailboxes[m][i]$
          $LoID \triangleq Msg[\text{"lockId"}]$
      IN
         $\land$ $Msg[\text{"type"}] = \text{"unlock"}$
         $\land$ $LoID = localInfos[m][\text{"lockId"}]$
         $\land$ $localInfos' = [localInfos$ EXCEPT
           $![m][\text{"nodeState"}] = \text{"unlock"},$
           $![m][\text{"lockId"}] = ZeroLockID]$
         $\land$ $mailboxes' = [mailboxes$ EXCEPT
           $![m] = RemoveByIndex(@, i)]$
         $\land$ UNCHANGED $\langle members \rangle$

$LockTimeout(m) \triangleq$
    $\land$ $localInfos[m][\text{"nodeState"}] = \text{"lock"}$
    $\land$ LET
         $LockBy \triangleq localInfos[m][\text{"lockId"}][\text{"intro"}]$
      IN
         $localInfos[LockBy][\text{"nodeState"}] = \text{"fail"}$
```

```
  $\land\ localInfos' = [localInfos\ \text{EXCEPT}$
    $![m][\text{"nodeState"}] = \text{"unlock"},$
    $![m][\text{"lockId"}] = ZeroLockID]$
  $\land\ \text{UNCHANGED}\ \langle members,\ mailboxes\rangle$

$RecvOp(m)\ \triangleq$
  $\land\ localInfos[m][\text{"nodeState"}] = \text{"lock"}$
  $\land\ \exists\, i \in 1\,..\,Len(mailboxes[m]) :$
    LET
        $Msg\ \triangleq\ mailboxes[m][i]$
        $LoID\ \triangleq\ Msg[\text{"lockId"}]$
        $OpAck\ \triangleq\ [type \mapsto \text{"opack"},$
          $lockId \mapsto LoID\ ,\ sender \mapsto m]$
    IN
        $\land\ Msg[\text{"type"}] = \text{"operation"}$
        $\land\ LoID = localInfos[m][\text{"lockId"}]$
        $\land\ localInfos' = [localInfos\ \text{EXCEPT}$
          $![m][\text{"nodeState"}] = \text{"unlock"},$
          $![m][\text{"lockId"}] = ZeroLockID,$
          $![m][\text{"groupTable"}] = @ \cup \{Msg[\text{"newMember"}]\}]$
        $\land\ mailboxes' = [mailboxes\ \text{EXCEPT}$
          $![LoID[\text{"intro"}]] = Append(@,\ OpAck),$
          $![m] = RemoveByIndex(@,\ i)]$
        $\land\ \text{UNCHANGED}\ \langle members\rangle$
```

Nodes

```
$NodeFail(n)\ \triangleq$
  $\land\ localInfos[n][\text{"nodeState"}] \neq \text{"fail"}$
  $\land\ localInfos' = [localInfos\ \text{EXCEPT}$
    $![n][\text{"nodeState"}] = \text{"fail"}]$
  $\land\ \text{UNCHANGED}\ \langle members,\ mailboxes\rangle$

$Next\ \triangleq$
  $\lor\ \exists\, i \in members : \exists\, n \in (Nodes \setminus members) :$
    $SendJoinRequest(n,\ i)$
  $\lor\ \exists\, n\ \in (Nodes \setminus members) : RecvGroupStruct(n)$
  $\lor\ \exists\, n\ \in (Nodes \setminus members) : RecvJoinFail(n)$
  $\lor\ \exists\, n\ \in (Nodes \setminus members) : JoinReqTimeout(n)$
  $\lor\ \exists\, m \in members : RecvJoinRequest(m)$
  $\lor\ \exists\, m \in members : RecvLockAck(m)$
  $\lor\ \exists\, m \in members : RecvAllLockAck(m)$
  $\lor\ \exists\, m \in members : LockReqTimeout(m)$
  $\lor\ \exists\, m \in members : CritOp(m)$
  $\lor\ \exists\, m \in members : CritRecvOpAck(m)$
  $\lor\ \exists\, m \in members : CritRecvGsAck(m)$
  $\lor\ \exists\, m \in members : CritExit(m)$
  $\lor\ \exists\, m \in members : RecvLockReq(m)$
  $\lor\ \exists\, m \in members : RecvUnlock(m)$
  $\lor\ \exists\, m \in members : LockTimeout(m)$
  $\lor\ \exists\, m \in members : RecvOp(m)$
  $\lor\ \exists\, n\ \in (members \cup \{x \in (Nodes \setminus members) :$
    $localInfos[x][\text{"nodeState"}] = \text{"joining"}\}) : NodeFail(n)$

$Fairness\ \triangleq$
  $\land\ \text{WF}_{vars}(\exists\, n\ \in (Nodes \setminus members) : RecvJoinFail(n))$
  $\land\ \text{WF}_{vars}(\exists\, m \in members : RecvJoinRequest(m))$
  $\land\ \text{WF}_{vars}(\exists\, m \in members : RecvUnlock(m))$
  $\land\ \text{WF}_{vars}(\exists\, m \in members : RecvLockReq(m))$
```

$\land\ \mathrm{WF}_{vars}(\exists\, m \in members : RecvLockAck(m))$
$\land\ \mathrm{WF}_{vars}(\exists\, m \in members : RecvAllLockAck(m))$
$\land\ \mathrm{WF}_{vars}(\exists\, m \in members : CritOp(m))$
$\land\ \mathrm{WF}_{vars}(\exists\, m \in members : RecvOp(m))$
$\land\ \mathrm{WF}_{vars}(\exists\, m \in members : CritRecvOpAck(m))$
$\land\ \mathrm{WF}_{vars}(\exists\, m \in members : CritRecvGsAck(m))$
$\land\ \mathrm{WF}_{vars}(\exists\, m \in members : CritExit(m))$
$\land\ \mathrm{WF}_{vars}(\exists\, n \in (Nodes \setminus members) : RecvGroupStruct(n))$

$Spec \triangleq Init \land \Box[Next]_{vars} \land Fairness$

$Inv1\_Mutex \triangleq Cardinality(\{m \in members :$
$localInfos[m][\text{“nodeState”}] = \text{“crit1”}\}) \leq 1$

$Prop1\_LocalTable \triangleq \forall\, n \in Nodes : (n \in members)$
$\leadsto (\forall\, m \in \{x \in members \setminus \{n\} :$
$localInfos[x][\text{“nodeState”}] \neq \text{“fail”}\} :$
$n \in localInfos[m][\text{“groupTable”}])$

Received 7 August 2023