Nodes, a Python Framework for Distributed Algorithms research

Matteo Lugli, mat. 196569
283122@studenti.unimore.it

Abstract: This work introduces Nodes, a Python framework designed to streamline the implementation of distributed algorithms for research, testing, and educational purposes. The framework provides an intuitive interface that allows researchers and students to write distributed algorithms in a syntax closely resembling pseudo-code, while abstracting away the complexities of network programming. Although primarily intended for academic and experimental use rather than production environments, Nodes can be extended to practical applications with minimal effort. The framework leverages Python's standard socket library to create a transparent abstraction layer over low-level communication mechanisms, enabling users to focus on algorithmic design rather than networking details. The released package includes implementations of several state-of-the-art distributed algorithms as examples, and features built-in visualization capabilities using Matplotlib for monitoring message exchanges between nodes.

1 Introduction

Distributed algorithm research focuses on designing and analyzing algorithms that operate across multiple interconnected nodes, where each component has limited knowledge of the global state. These algorithms are fundamental for enabling coordination, communication, and decision-making in decentralized environments. In today's computing landscape, the shift towards autonomous agents and complex distributed systems such as multi-agent systems, IoT networks and edge computing has made distributed algorithms more critical than ever. Autonomous agents, often integrated with AI, require robust distributed algorithms to collaborate effectively, share information, and make decisions in dynamic and uncertain environments. For example, in swarm robotics, distributed algorithms enable agents to self-organize and achieve collective goals without centralized control. The rise of distributed computing paradigms also demands advancements in consensus protocols, resource allocation, and load balancing to ensure efficiency and resilience. Before deploying a distributed application, essential steps include prototyping, algorithm testing, and complexity analysis. In this context, having an intuitive framework with a simple interface is highly beneficial, allowing developers to carry out these steps with minimal effort. Python is an excellent language for this purpose, as it is easy to use and provides a vast ecosystem of well-supported libraries, making it ideal for rapid prototyping and experimentation. In this paper, I introduce Nodes, a Python framework designed to facilitate the implementation, testing, and analysis of distributed algorithms. Nodes aims to provide an intuitive and lightweight environment for researchers and students by abstracting the complexities of low-level network programming. It allows users to define node behavior in a manner similar to pseudo-code while maintaining realistic distributed execution semantics. The framework leverages Python's standard socket library to provide a simple yet powerful abstraction over message-passing mechanisms. The remainder of this article is organized as follows. Section 2 discusses related work and compares Nodes with existing frameworks. Section 3 presents the architecture and core components of Nodes, including the communication manager, initializer, node structure, protocol design, and visualizer. Section 4 provides case studies, demonstrating the effectiveness of Nodes in implementing distributed algorithms such as Lamport's mutual exclusion protocol. Finally, Section 5 concludes the paper and outlines potential future improvements to the framework.

2 Related work

The field of multi-agent system simulators and distributed algorithm simulation software is diverse, offering various tools tailored to different research needs. Among them, NetLogo [1] stands out as one of the most widely used simulators in scientific research, particularly for modeling autonomous agents and swarm-based systems. Its intuitive interface, coupled with a dedicated programming language, makes it well-suited for these types of simulations, enabling efficient development and analysis of complex agent interactions. NetLogo is built around an agent-based modeling paradigm where "turtles" move on a grid of "patches", primarily targeting social and natural science simulations. Nodes instead is specifically designed for distributed algorithms research, with explicit support for message passing, network protocols, and distributed computing primitives. NetLogo

updates agents one at a time in a predetermined order, making it fundamentally sequential, while Nodes runs nodes as separate processes that operate truly in parallel, accurately reflecting real distributed system behavior. Finally, NetLogo uses discrete time steps where all agents update in sequence within each tick, while Nodes handles real asynchronous message passing and timing issues, which is essential for realistic distributed algorithm testing. Another distributed algorithm simulator is Sinalgo [2], which focuses on verifying the correctness of algorithms. It is written in Java, whereas Nodes is developed in Python. To the best of my knowledge, the work that most closely aligns with the goals and implementation of Nodes is the distributed framework [3] used in the Distributed Systems course at Jagiellonian University, implemented in Go.

3 Methods

In this section I will present the core implementation of Nodes, focusing on its architecture, key components, and the design choices. If you are interested in the codebase or if want to become a contributor, please refer to the official repository [4].

3.1 Comunication Manager

ComunicationManager is the base class that handles the exchange of messages between entities. It handles 2 objects: a message listener and a message queue. The first is a object that inherits from the `threading.Thread` Python class. It's execution starts as soon as the main entity is created, and it constantly listens for incoming messages and pushes them into the queue. This class exposes some methods such as `receive_message()`, used to pop the first message from the queue, or `insert_message()` to force the insertion of a message in the queue. Both the server (`Initializer`) and the client (`Node`) inherit from this class. Thanks to this implementation, message reception is a non-blocking operation. The message listener runs on a separate thread, allowing nodes to continue performing other operations while simultaneously listening for incoming messages. This approach ensures that nodes can execute their algorithms efficiently without being blocked by message reception, closely resembling the behavior of real distributed systems where computation and communication occur concurrently.

3.2 Initializer

This class mimics the system administrator who sets up the network and connects nodes (3.3) according to the provided network model. The user can import the initializer (also referred to as "server") in its *server.py* file, which will be the entry point of the program (the actual script that needs to be executed). The initializer object will take care of the setup by creating all of the nodes as separate processes and it will start a simple communication protocol which ensures that all of the nodes have the necessary information to start the actual protocol. In particular, it sends to each node a dictionary containing the association between the ID of the connected nodes and the IP and port to send messages via UDP. This way, each node will have a small "DNS" server. If the node receives all of the information correctly, it sends an ACK message back to the server. When the server receives ACK from every node, it can send the wakeup message to trigger the start of the actual protocol. Once the protocol ends or any node runs into errors, the server will broadcast the error message received by the faulty node to all of the other nodes to inform them. A simple diagram of a ring network, illustrating the connections between nodes and the server, is presented in Figure 1. The connections between the nodes and the initializer are shown as dashed lines, as they are only used for the initial setup phase and the protocol termination procedure.

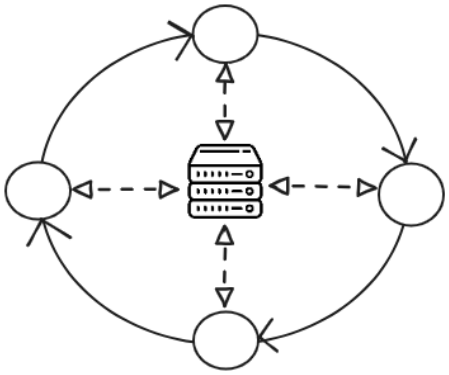

Figure 1: *A simple diagram of a ring network illustrating the connections between nodes and the server. Curved lines are unidirectional connections between nodes (circles), while dashed lines are bidirectional connections between nodes and the initializer.*

3.3 Node

The `Node` class is the core of the framework. It offers several primitives to exchange messages with other nodes or with the server. Some of these are `send_to_all()`, `send_to_all_except()` and `send_back()`, which are particularly useful when writing protocols that closely resemble pseudocode. Specifically, the first sends a message to all neighboring nodes, the second excludes a specified node, and the latter sends a message back to the server. The mentioned methods are wrappers around the `_send()` primitive,

which handles the socket at the lowest level. I chosed to use UDP instead of TCP primarily to minimize overhead. Maintaining a persistent connection with all neighboring nodes for every node would introduce unnecessary complexity and resource consumption. Additionally, UDP allows for a simpler and more flexible implementation, making the code easier to modify and maintain. Since the framework is primarily executed in a local environment, the reliability guarantees provided by TCP were not a critical requirement, further justifying the decision to use UDP datagrams. The main issue with using UDP is the non-guarantee of FIFO (First In First Out) reception on communication channels. This problem will be further discussed in section 4.
The server creates nodes as separate processes, so that they have no shared memory and they are forced to use socket communication to exchange information with each other. Another possible design choice was to utilize shared memory. However, to better align with the distributed computing paradigm, I decided to keep the nodes more isolated, even at the cost of relying on sockets for communication. The user defines the behavior of the nodes using the *client.py* file, where the only required steps are to create the node object using command-line arguments (which are passed by the initializer), import or define the desired protocol (3.4), instantiate the corresponding protocol object (which takes the node object as input), and run the protocol using the `run()` primitive. The user can define its own custom node by writing a class that inherits from `Node`. For example, a user might need methods to simplify the implementation of algorithms that operate on ring structures. In this case, the user could write a custom class that inherits from `Node` and implements the necessary primitives (such as `send_other_way()`, which forwards the message in the direction opposite to that of reception).

3.4 Protocol

To implement a new distributed algorithm, users simply extend the `Protocol` class and implement two key methods: `setup()` for initialization, `handle_message()` for message processing logic. Additionaly, users can extend the `cleanup()` to modify the termination procedures. For instance, nodes can override this method to send messages containing performance metrics such as total message count, computation time, or final node states back to the initializer for analysis. This design pattern ensures consistency across implementations while maintaining flexibility. For example, the already implemented protocols inside the `Protocol` subpackage demonstrate how a complex leader election algorithms can be implemented by primarily focusing on state transitions and message handling logic, without dealing with low-level networking details. The `Protocol` class also provides built-in support for performance analysis through automatic message counting and logging capabilities, enabling researchers to easily analyze algorithm complexity and behavior. Figure 3 shows the UML diagram section related to protocols.

3.5 Visualizer

To help developers understand and debug protocol execution, the framework includes a basic visualization component. The `Visualizer` class provides a simple real-time view of message exchanges between nodes using NetworkX [5] and Matplotlib [6]. It displays the network topology as a graph and shows message flow using color-coded arrows, labeled with the message type and content. This visual feedback can be helpful during protocol development and testing phases to track message propagation and verify basic algorithm behavior. The visualization can be enabled through a flag in the initializer when needed during development, and disabled when running performance tests. The visualization system operates through a message replication mechanism: when nodes send messages to each other, they simultaneously replicate these messages to the visualizer. This centralized approach gives the visualizer global knowledge of all network communications, allowing it to construct a complete view of message exchanges. However, since the framework is primarily designed for asynchronous algorithm development, correctly visualizing these message exchanges is particularly challenging. To address this issue, I implemented workarounds such as introducing temporal delays on nodes to make message flow visualization more comprehensible. This way the visualization thread is able to display the whole queue of messages at once and wait for more messages to arrive in the next ”round”. Figure 2 is a screenshot of a running visualization of a leader election algorithm on a ring network.
Figure 4 displays the UML diagram section related to the communication between the main components of the framework, including the visualizer.

3.6 Message

The `Message` class provides the fundamental infrastructure for message serialization, de-serialization, and type registration. Each message used for communication must inherit from this base class, ensuring consistency. The class maintains a registry of message types through the `_message_types` class variable.

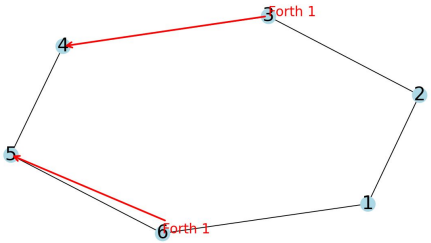

Figure 2: *Screenshot of a running visualization of the leader election "Controlled Distance" algorithm.*

The base class provides several essential methods for message handling. Messages are serialized to UTF-8 encoded JSON strings before network transmission through the `serialize()` method. Conversely, the `deserialize()` static method reconstructs message objects from received bytes by decoding the JSON data and using the registered message type to instantiate the appropriate message class. To support this serialization process, messages can convert themselves to dictionaries through the `to_dict()` method, which captures the message class name, command, and sender information. Developers can create new message types by defining new classes that inherit from the base `Message` class. These classes must implement the required serialization methods and be registered as new message types. The registration decorator (`@Message.register`) handles the integration of new message types automatically. Figure 5 is the UML diagram section related to the message hierarchy.

4 Case Studies

Users can find several examples of distributed algorithms implemented using Nodes in the *Tests* folder in the Github repository. These examples serve as both demonstrations of the framework's capabilities and as a reference to implement new algorithms. In this section, the focus is on the Lamport mutual exclusion algorithm, which provides a particularly interesting case study due to its unique characteristics and requirements. Lamport's algorithm allows processes in a distributed system to access a shared resource (critical section) without conflicts by establishing a total ordering of events. What makes this algorithm particularly notable is its reliance on:

1. Logical timestamps: each node maintains a counter that increases with local events and message receipts.
2. Message queuing: nodes must maintain and sort request queues.
3. FIFO communication channels: the algorithm assumes that messages are delivered in the order they are sent.

The `Node` class provides a FIFO mode through its constructor parameter `(fifo=True)`, which automatically handles sequence numbers for messages between nodes. When enabled, the framework tracks sequence numbers for sent messages and ensures they are processed in order, addressing one of Lamport's core assumptions. In particular, when the flag is set, the framework adds to every node-to-node message an integer starting from zero representing the sequence number. When a node receives a message, if it does not have the expected sequence number (previously received sequence number from that node plus one), the node forces the insertion of the message back in the queue. This way, message chains will be proccessed in the correct order, emulating the TCP message ordering guarantee. Additionaly, to solve the message queuing, I decided to use a priority queue (thanks to the `heapq` [7] python package), which is helpful in keeping track of the message with the lowest timestamp. Thanks to this implementation, the lookup in the queue to check if the highest priority request is associated with a node's ID can be done in constant time by the node itself. Finally, Nodes' messaging capability is not confined to the `handle_message()` function but extends throughout the codebase. For example, in the implementation described above, 'Release Critical Resource' messages are dispatched at the conclusion of parallel threads simulating access to a shared critical resource.

5 Conclusions

In this work, I presented Nodes, a Python framework designed to facilitate the implementation, testing, and analysis of distributed algorithms. The framework provides an intuitive interface that allows users to define node behavior in a manner similar to pseudocode, reducing the complexity of developing distributed systems. By leveraging Python's standard socket library, Nodes abstracts away the socket communication layer, enabling researchers and educators to focus on algorithmic design rather than low-level networking details. Nodes distinguishes itself from other existing simulation tools by offering true parallel execution of nodes as separate processes, ensuring a realistic emulation of distributed system behavior. Unlike traditional agent-based modeling environments such as NetLogo, Nodes operates on an asynchronous message-passing paradigm, better reflecting real-world distributed computing scenarios. Furthermore, the framework includes a set of built-in, state-of-the-art distributed algorithms as examples, making it a valuable resource for both educational

and research purposes. A key feature of Nodes is its built-in visualization component, which enables real-time monitoring of message exchanges using Matplotlib and NetworkX. Although the visualization system effectively provides insights into protocol execution, it currently relies on workarounds to handle asynchronous message flows, which presents an opportunity for future improvements.

Concerning future work, the framework can be improved to further extend its capabilities. The visualizer can be refined to improve real-time representations of asynchronous message exchanges and support offline visualization. Additionally, it can be reasonable to implement a back-end that does not rely on socket communication to reduce the computational overhead, allowing users to enable a more efficient local simulation mode. Additionally, more state-of-the-art distributed algorithms could be implemented. Lastly, integrating support for machine learning libraries such as PyTorch and Scikit-learn will further expand Nodes' potential for research in AI-driven distributed systems.

References

[1] Uri Wilensky. *NetLogo*. http://ccl.northwestern.edu/netlogo/. Northwestern University, Evanston, IL: Center for Connected Learning and Computer-Based Modeling, 1999. URL: http://ccl.northwestern.edu/netlogo/.

[2] SinALgo. *SinALgo*. https://sinalgo.github.io/. Accessed: 2025-02-22.

[3] Krzysztof Turowski. *Distributed Algorithm Go Framework*. https://github.com/krzysztof-turowski/distributed-framework. Accessed: 2025-02-22.

[4] Matteo Lugli. *Distributed Python Framework*. https://github.com/theElandor/Nodes/. Accessed: 2025-02-22.

[5] Networkx. *NetworkX*. https://networkx.org/. Accessed: 2025-02-22.

[6] Matplotlib. *Matplotlib*. https://matplotlib.org/. Accessed: 2025-02-22.

[7] heapq. *heapq*. https://docs.python.org/3/library/heapq.html. Accessed: 2025-02-22.

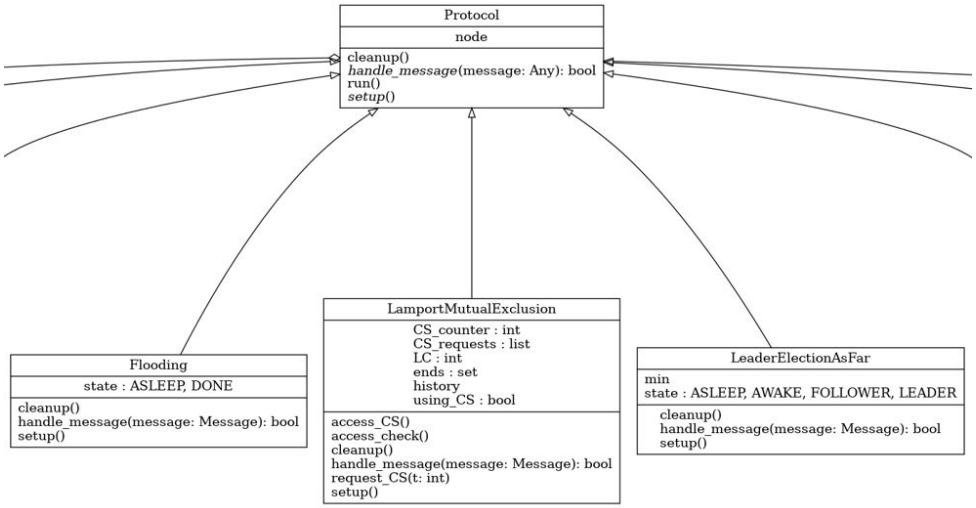

Figure 3: *UML diagram displaying* Protocol *class and other derived protocols.*

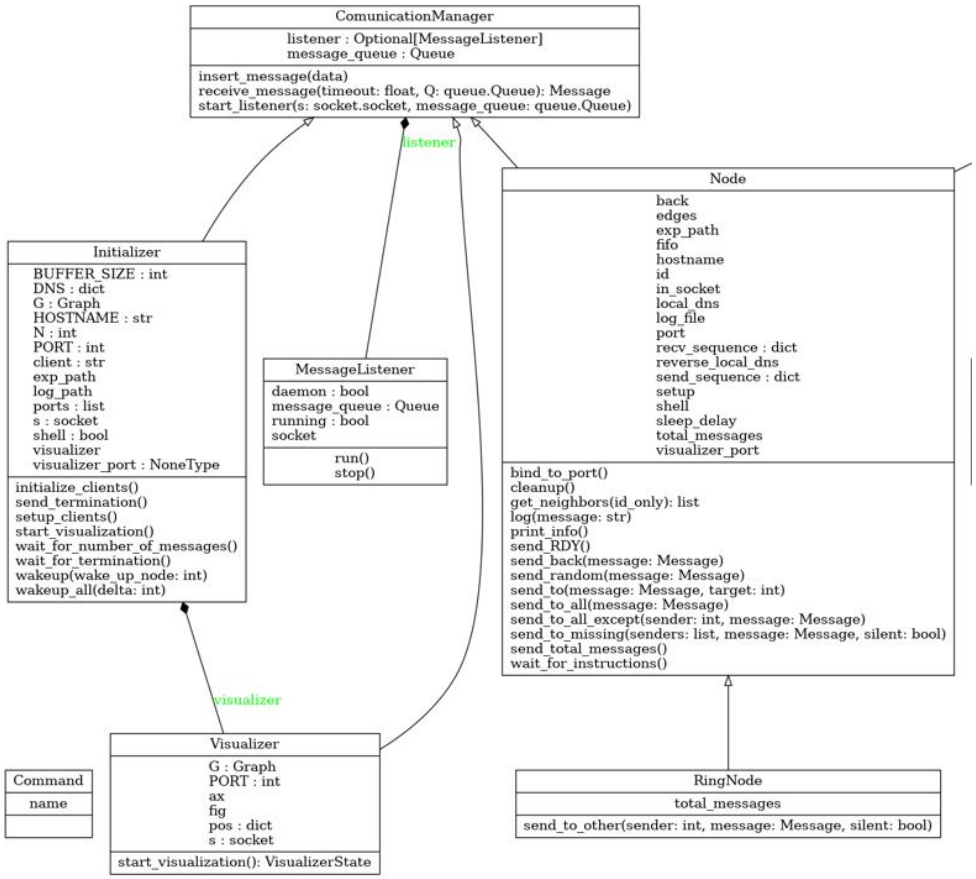

Figure 4: *UML diagram displaying* ComunicationManager, Initializer, Node, MessageListener, Visualizer *classes. These are the most important classes of the framework.*

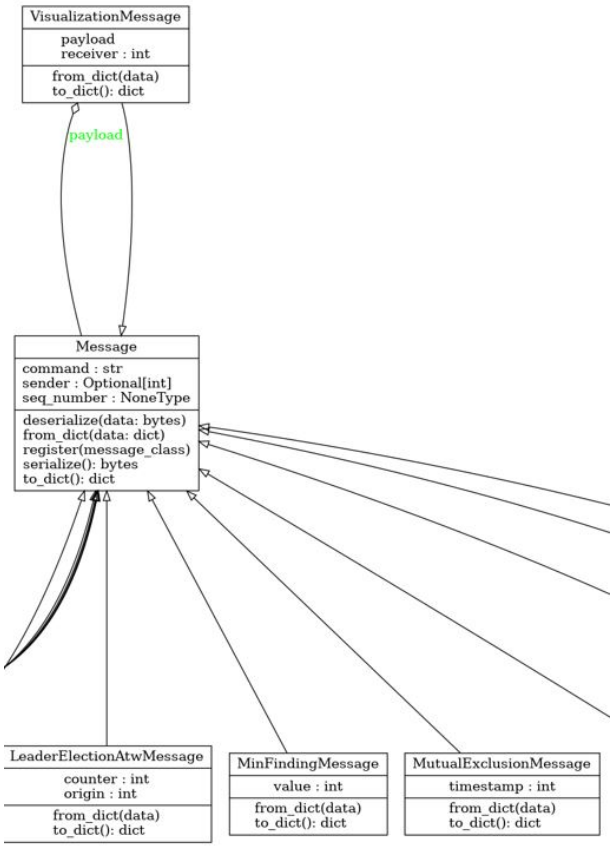

Figure 5: *UML diagram displaying* VisualizationMessage*,* Message *and other message classes.*