Chandy-Lamport Algorithm in Action: Consistent Global Snapshot for Distributed Applications Using Go and Docker

Matteo Basili
Department of Civil Engineering and Computer Engineering
University of Rome Tor Vergata
Rome, Italy
matteo.basili@students.uniroma2.eu

***Abstract*—The Chandy-Lamport algorithm is a well-known method for recording a consistent global state in distributed systems. This project explores the practical implementation of the Chandy-Lamport algorithm using the Go programming language. The implementation is tested on a pipeline distributed application deployed with Docker Compose on an EC2 instance. The primary objective is to capture consistent snapshots of the system's state, including the local state of each process and the state of inter-process communication channels. Our approach demonstrates the feasibility and efficiency of applying the Chandy-Lamport algorithm in modern distributed environments. The results confirm the algorithm's effectiveness in ensuring consistency across distributed systems. This work provides valuable insights for developers and researchers aiming to implement robust global state recording mechanisms in distributed applications.**

Keywords—Chandy-Lamport Algorithm, Distributed Systems, Global Snapshot, Consistency, Go Programming Language, Docker Compose, EC2 Deployment, Pipeline Application, State Recording, Inter-process Communication

I. Introduction

In modern computer systems, distributed computing plays a crucial role in enabling scalability, fault tolerance, and efficient resource utilization. Distributed systems are characterized by their ability to coordinate multiple autonomous components to achieve a common goal. One fundamental challenge in distributed systems is the recording of a **consistent** and coherent global state, which is essential for debugging, monitoring, and maintaining system integrity. The Chandy-Lamport algorithm, proposed by Leslie Lamport and K. Mani Chandy, addresses this challenge by providing a method for recording a consistent state of a distributed system.

In this project, we aim to implement the Chandy-Lamport algorithm using the **Go** programming language and deploy it in a distributed application scenario. The primary objective is to demonstrate the feasibility and effectiveness of the algorithm in real-world distributed environments. Additionally, we seek to explore the practical challenges and considerations involved in implementing and deploying distributed algorithms.

This paper is structured as follows: Section II provides an overview of the Chandy-Lamport algorithm and its theoretical foundations. Section III details the solution design, outlining the architecture and components of our system. Section IV delves into the solution details, describing the specific implementation approach and technologies employed. Section V explains the deployment process, including how **Docker Compose** and **EC2** instances were used to distribute and test the system. Section VI presents the experimental setup and results, highlighting the performance and behavior of our implementation under various conditions. Finally, Section VII discusses the implications of our findings, and the challenges encountered.

II. Background and Theory

A. Introduction to the Chandy-Lamport Algorithm

In distributed systems, a snapshot is a representation of the global state of the system captured at a specific point in time. It includes the local states of each process and the states of communication channels between processes.

The Chandy-Lamport algorithm is a distributed algorithm used to record a consistent global snapshot of a distributed system, despite the asynchronous and concurrent nature of its components.

B. Consistent Snapshots in Distributed Systems

Formally, a consistent snapshot in a distributed system is one for which, given any two events A and B in the system, where $A \rightarrow B$ (i.e. *A causally precedes B* [1]), if event B is included in the snapshot, then it must be event A also included.

This principle is crucial for maintaining the logical coherence of the snapshot. Consider two processes, P1 and P2, where P1 sends a message to P2 in event *A*, and P2 receives it in event *B*. If a snapshot captures the state where P2 has received the message (event *B*), consistency requires that the snapshot also include the event where P1 sent the message (event *A*).

C. Assumptions of the Snapshot Algorithm

To achieve the consistency, the Chandy-Lamport algorithm relies on several key assumptions about the system and its communication infrastructure [2]:

- There are no failures in message delivery. All messages sent between processes arrive intact and exactly once, without loss or duplication.

- No process crushes during execution.
- There exists a pair of channels for each pair of processes P and Q in the system (one from P to Q and one from Q to P). This ensures that every process can communicate directly with any other process.
- The communication channels between processes are unidirectional and preserve the order of message delivery (FIFO - First In, First Out). This ensures that messages are received in the same order they were sent.
- Any process in the system can initiate the snapshot algorithm. This flexibility allows for decentralized initiation without needing a central coordinator.

D. Description of the Algorithm

The algorithm captures consistent snapshots using special control messages known as marker messages. The operation of the algorithm can be summarized as follows [3], [4].

1) The observer process (the process taking a snapshot):

a) Saves its own local state.

b) Sends a marker message on all its output channels.

c) Starts recording the messages it receives on all its input channels.

2) A process Pi that has received a message marker for the first time from a process Pj on the channel Cji:

a) Saves its own local state.

b) Marks the channel Cji as empty.

c) Sends a marker message on all its output channels.

d) Starts recording the messages it receives on all its input channels except Cji.

3) A process Pi that has received a message marker from a process Pj on the channel Cji and that has already see a marker: Stops recording on the channel Cji.

4) A process that has received a marker message from all other processes: Sends its state to all the others.

Combining these, the observer constructs a comprehensive global snapshot, capturing the state of the entire system and all inter-process communications at the snapshot moment.

E. Properties and Guarantees

- Snapshots are consistent and coherent.
- The algorithm operates without interfering with the normal activities of the processes, allowing the system to continue functioning as usual during the snapshot procedure.
- The algorithm ensures termination, meaning all processes complete their part of the snapshot procedure after a finite number of steps.
- The algorithm supports multiple initiators, allowing multiple processes to initiate snapshots simultaneously without conflict.
- The algorithm is scalable and adaptable to various network topologies, performing efficiently even in large and complex distributed systems.

F. Complexity and Limitations

We consider our system as a directed graph D = (N, A) in which:

- The set N of nodes represents the set of processes.
- The set A of arcs represents the set of communication channels between the various processes.

Then, the complexity of the algorithm, in terms of messages exchanged, is O(|A|), where |A| is the number of arches in the graph and equal to |N| * (|N| - 1), provided that for each node there is a channel towards all the other nodes.

Other important considerations must be made regarding the potential limitations that the Chandy-Lamport algorithm places before us, in addition to the limits imposed by the assumptions made previously.

- High Communication Overhead: significant overhead in large systems with many processes and dense interconnections.
- State Recording Overhead: computational and memory overhead can be prohibitive.
- Latency Introduced by Snapshot Operations: can degrade performance in real-time and high-performance applications.

G. Utility of Snapshots

In distributed systems, you may need to record the global state of the system for several reasons [5]:

- To create checkpoints. If for some reason the application fails, this checkpoint can be reused.
- Garbage collection. A snapshot can be used to remove objects that have no references.
- To detect deadlocks.
- For debugging.

III. Solution Design

In this section, we describe the design and architecture of our distributed system that implements the Chandy-Lamport algorithm and its test in our project. The goal is to have a library that is as decoupled as possible from the main application.

A. System Architecture

Our system is composed of two main entities: the node (or process) and the distributed pipeline application for the test (**App**). Each node is divided into three components: **NodeApp**, **Process** and **SnapNode**. The interactions between these components enable the execution of the application, message passing, and snapshot capturing functionalities.

- NodeApp: This is the application layer of the node, which acts as an RPC server for the App. It provides two main methods: *SendAppMsg* and *MakeSnapshot*.

- Process: This is the core messaging component responsible for handling the transmission of all messages between nodes: it works like a message broker.
- SnapNode: This component manages the global snapshot process. It implements the logic of the Chandy-Lamport algorithm.

So, a possible view of our system is shown in Fig. 1.

B. Communication Protocol

The communication between the App and the NodeApp component is implemented using RPC over TCP. This setup allows the App to invoke methods on the NodeApp to either send application messages or initiate a snapshot. When the App wants to send a message within the network, it calls the SendAppMsg method on the target NodeApp. For initiating a snapshot, the MakeSnapshot method is called.

Processes communicate directly with each other over TCP connections. Each Process listens on a designated port for incoming messages and forwards outgoing messages to the target Process(es). Messages between Processes are handled in a FIFO manner, ensuring that the order of messages is preserved.

C. Snapshot Process

The snapshot process is initiated by calling the MakeSnapshot method on a NodeApp. This method triggers the SnapNode component to start recording the local state and send marker messages to all its connected neighbors. The SnapNode records the state of the local Process and captures any in-transit messages as part of the global snapshot. It is also responsible to collect all the states of the system processes used to build the global snapshot that it sends to the NodeApp. The latter will take care of returning the global state to the App.

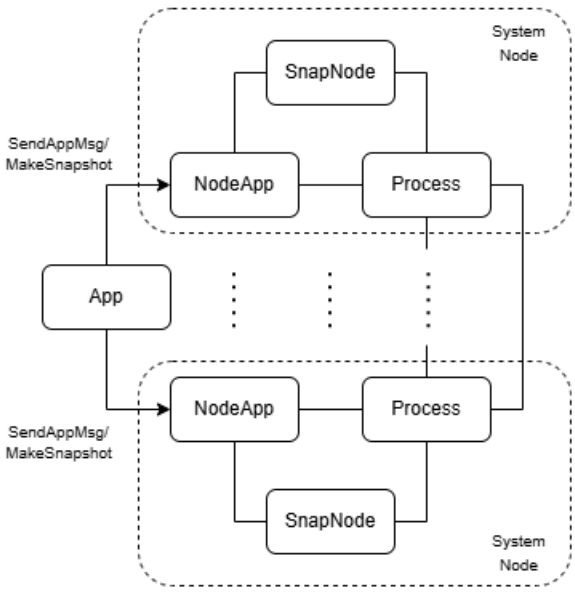

Fig. 1. Architecture of the system. *Directed arcs indicate RPC calls, while undirected arcs indicate simple interactions.*

IV. Solution Details

In this section, we provide a detailed description of the implementation specifics. This includes the technical aspects of our solution, such as the programming constructs, libraries, and tools used, and how we addressed key implementation challenges.

All the project's code is located in a Github repository [7]. All the folders, files and code calls mentioned in this section (and also in the next ones) are located within this repository.

A. Programming Language

We chose the Go programming language for its concurrency support and simplicity in building distributed systems [6]. Go's built-in features, such as goroutines and channels, made it an ideal choice for implementing the message-passing mechanisms required by the Chandy-Lamport algorithm.

B. Data Structures

Some of the fundamental structures used to implement the application are defined in the “src/utils” folder, such as message types, process status and communication channels between system components. In particular, refer to the files "message.go", "channel.go" and "state.go".

- A marker message is differentiated from an application message by the *IsMarker* flag.
- There are channels that carry data relating to marker messages (*MarkChannels*), those relating to application messages (*AppMsgChannels*) and those relating to local states (*StatesChannels*).
- The complete state (*FullState*) of a process is composed of the local state (*NodeState*) plus the states of all its incoming channels (*ChState*).

C. Event Management

Regarding system debugging, a logging system has been configured for all components of the same node. In this way we can record information about the execution of the system with four different logging levels (*trace*, *info*, *warning* and *error*). At the end of the execution of the designed testing application (see paragraph G), a bash script (“merge-output.sh”) combines the log files generated on each node into a single file and sorts the events based on the local time of the node in which the events have been registered (see paragraph H).

With this log system, we can keep track of all messages sent within the network, so we can verify the consistency of the global snapshots created.

When a message is sent within the network, whether it is an application message, a marker or a state, we use the *GoVector* library [8]. At the time of sending, it transforms this information by adding the vector clock to the message used for debugging [9].

D. NodeApp Implementation

NodeApp is the simplest of the three components, since it is an RPC server that exposes two methods (see “node_app.go” file in the “src/main” folder): one to send an application message (SendAppMsg()) and the other to start the process to achieve a

global state from the SnapNode (MakeSnapshot()). Once the latter is complete, the global status is printed in the corresponding log file. Furthermore, this component has a goroutine (*recvAppMsg()*) that receives messages sent by the other nodes and records them in the log file mentioned above, so that the user has proof of it.

Fig. 2 shows the state diagram of NodeApp [10].

E. Process Implementation

The Process component works like a messaging service (see "process.go" file in the "src/process" folder). On the one hand, it listens on a user-specified TCP port to receive messages from other nodes via a goroutine (*receiver()*). The information received can be a marker, an application message or the local status of a node. In all these cases, it sends the information to the SnapNode component (through the *MarkCh.RecvCh* and *StatesCh.RecvCh* channels) in order to be used by the algorithm to obtain global states. Furthermore, if it is an application message, this information is sent to the NodeApp component that will use it (for example to show it to the user).

On the other hand, there is the *sender()* service, which is another goroutine that serves the requests of the NodeApp and the SnapNode components. The only request that can be made from the application is to send a message to another node (through the *AppMsgCh.SendToProcCh* channel). After receiving this request, the message is sent to the corresponding node using the TCP connection and to the SnapNode. The potential requests generated by SnapNode are two: the markers sending (through the *MarkCh.SendCh* channel) and the node state sending (through the *StatesCh.CurrCh* channel) to the other nodes.

To better understand the situation, Fig. 3 shows the state diagram of Process.

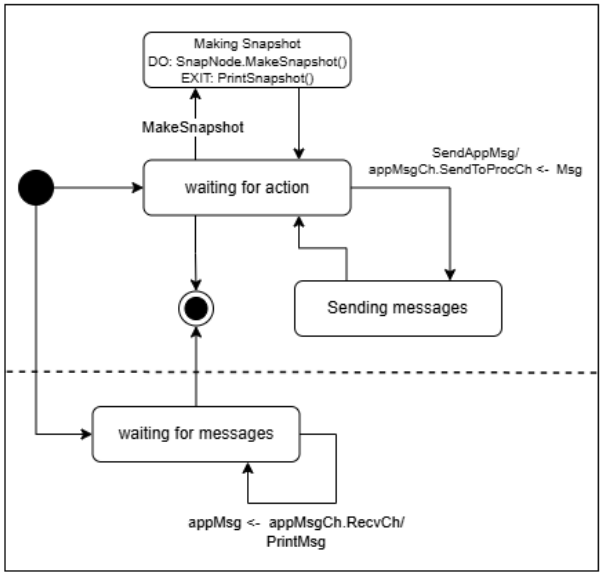

Fig. 2. NodeApp state diagram. *The part below the dotted line indicates the recvAppMsg() goroutine.*

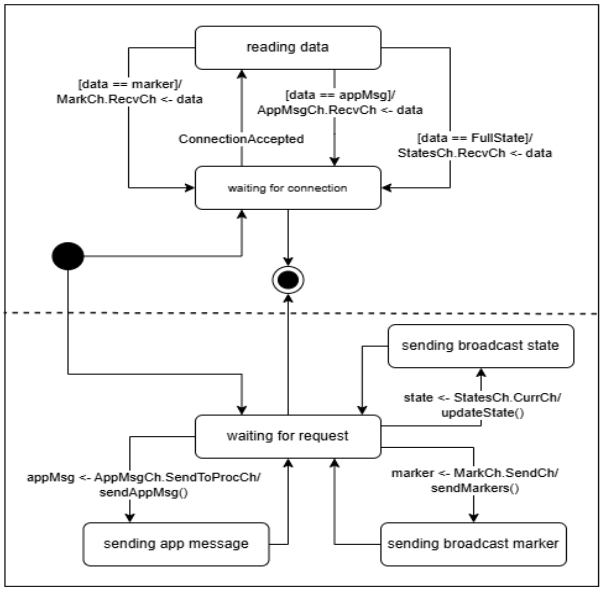

Fig. 3. Process state diagram. *The part above the dotted line represents the behavior of the receiver() goroutine while the part below shows the behavior of the sender() goroutine.*

F. SnapNode Implementation

The SnapNode component is responsible for carrying out the entire global state process (see "snapshot.go" file in the "src/snapshot" folder). It provides the *MakeSnapshot()* method and launches a goroutine (*wait()*) that manages the messages received from the other processes through the related channels: application messages are received via *SendMsgCh*, markers via *MarkCh.RecvCh* and states via *StatesCh.RecvCh*. To notify a local state or a marker sending to the Process component, are used the *StatesCh.CurrCh* and *MarkCh.SendCh* channels.

When the MakeSnapshot() method is invoked (from a goroutine), the node saves its local state (*saveProcState()*), sends marker messages (*sendBroadMark()*), and begins recording on all incoming channels (*startRecChs()*). Then, it waits until the global state is created.

When a marker is received, the node follows exactly the steps of the Chandy-Lamport algorithm: it checks if the marker received is the first one that the node receives; if so, the node saves its local state, marks the channel from which the marker has been received as empty, sends marker messages and starts recording on all incoming channels. If not, the node stops recording on the channel from which the marker has been received (*stopRecCh()*).

When all the markers have been received, the node waits to receive all the states from the other nodes and finally constructs the global state of the system from these states. After that, it sends the created snapshot to the goroutine that started the Snapshot process through the *InternalGsCh* channel, so that the goroutine can return it to the application.

Fig. 4 shows this entire procedure described through the state diagram.

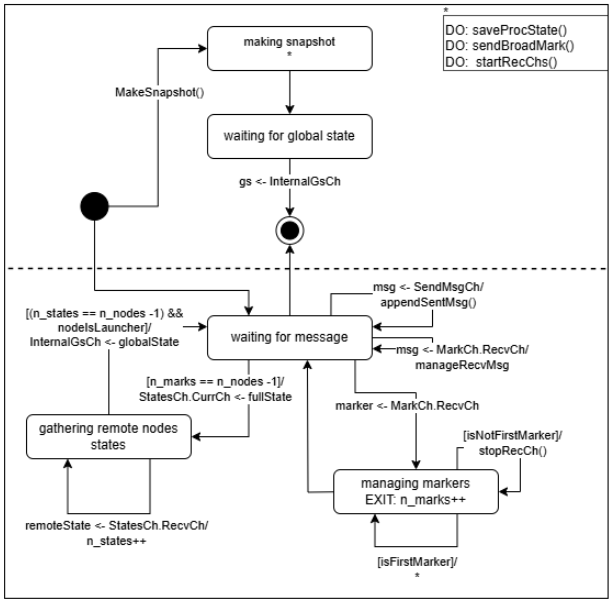

Fig. 4. SnapNode state diagram. *The part above the dotted line represents the behavior of the goroutine that has called the MakeSnapshot() method while the part below shows the behavior of the wait() goroutine.*

G. Pipeline Distributed Application

The solution is tested on a pipeline distributed application that works in this way (see "app.go" file): all processes start with the same (integer) balance (in dollars); every second, each process transfers funds (a random integer between 1 and 100 dollars) to another random process; a process, chosen also randomly, takes a snapshot of the system every two seconds. This whole process lasts 10 seconds. The transferred funds and the collected snapshots are displayed to the user.

An output directory will be created with network process and GoVector logs, which can be useful for monitoring, verifying, diagnosing, and replaying distributed system states.

In this situation, the local state of a process is represented by the balance and the messages sent and received.

H. Setup and Code Run

First of all, we need to configure our network: to do this, we must modify the "net_config.json" file. For each network node it is necessary to insert:

- The index within the network (starting from 0).
- The name ("P" + node index).
- The IP address.
- The RPC port.
- The port of the application that manages the node.

Then, it is necessary to insert the initial balance of the nodes and the maximum number of connection attempts ("sendAttempts") between the system components, beyond which the system get crushed.

Once our network is configured, we must launch the various processes: this can be done by running the "run-processes.sh" bash script. Once our processes are running, we can run our testing application ("app.go").

At the end of the execution, in the "output" folder we will be able to take a look at the generated log files. In particular, launching the "merge-output.sh" script will combine the individual log files and create the file "output/GoVector/completeGoVectorLog.log", which can be used to print the application-generated trace of the Chandy-Lamport algorithm to Shiviz [11].

For more details, see the "README.md" file in the repository.

I. Testing

Some unit tests have been written to verify the correct functioning of some system features (see "src/utils" folder), such as the creation of communication channels, the creation of messages, reading the configuration file, and the creation of states.

V. Deployment

The solution was deployed using Docker Compose and an Amazon Web Services (AWS) EC2 instance [12], [13]. This approach enabled the creation of a flexible and scalable distributed environment, facilitating the development of the pipeline distribute application.

A. Docker Compose

For each node/process of the system and for the testing application, a dedicated *Dockerfile* was generated [14]. The contents of the Dockerfile include:

- Base image: Each container uses an official Go image as its base (golang:1.22), ensuring that all nodes have the same development environment.
- Setting the working directory: The */app* directory is chosen as the working directory for each container.
- Installing dependencies: The "go.mod" and "go.sum" files are copied, and the Go dependencies are downloaded using *go mod download*.
- Source code compilation: The application source code is copied to the working directory, and is compiled into an executable binary.
- Port exposure: Each container exposes the ports necessary for internal system communication.
- Startup command: The container runs the compiled binary, specifying the arguments (in case of process Dockerfile) necessary for the node to run (such as the node ID, application port, and network configuration file).

Once all the Dockerfiles have been created, the "docker-compose.yml" file will be created, too. This file includes the following configurations:

- Testing Application Service: A separate container is dedicated to the testing application, defined with the

name “App Service”. This container is built from the app Dockerfile and depends on node containers (*depends_on*), ensuring that nodes are started before the testing application.

- Node Services: Each node is defined as a separate service (“p0”, “p1”, etc.), each with its own Docker container.
- Communication network: All services are connected via a Docker bridge network defined in the file (*chandy-lamport-net*), which facilitates communication between containers without the need of complex external network configurations.

If there are many nodes in the system, to make the creation of all these Docker files easier and more dynamic, a bash script will be created (“generate-docker-files.sh”). This reads the “net_config.json” configuration file and creates a Dockerfile for each node ("Dockerfile.P0", "Dockerfile.P1", ...), the Dockerfile for the application (“Dockerfile.app”) and finally the docker-compose.yml file.

B. EC2 Instance

An EC2 instance was chosen with adequate specifications to support the intended workload, including sufficient CPU, memory, and network bandwidth (specifically, a t2.micro instance was used). The instance was configured with a Docker-compatible operating system (Amazon Linux 2023 AMI).

Once the EC2 instance was configured and launched, the docker-compose.yml file (along with all project code) was transferred and launched on the instance using standard Docker Compose commands. This made it possible to start and manage all containerized services with a single command, maintaining consistency with the local development environment.

C. Benefits

Using Docker Compose and AWS EC2 together offers multiple benefits [12], [15]:

1) Ease of scalability: The ability to scale Docker services and the elasticity of EC2 instances allow to easily adapt the system to the needs of the workload.

2) Simplified control: Docker Compose allows to define and manage multi-container applications in a single YAML file. This simplifies the complex task of orchestrating and coordinating various services, making easier to manage and replicate the application environment.

3) Rapid application development: Compose caches the configuration used to create a container. When a user restarts a service that has not changed, Compose re-uses the existing containers. Re-using containers means that users can make changes to their environment very quickly.

4) Portabiliy across environments: Compose supports variables in the Compose file. Users can use these variables to customize their composition for different environments.

5) Secure: Amazon EC2 works to provide security and robust networking functionality for compute resources.

6) Reliable: Amazon EC2 offers a highly reliable environment where replacement instances can be rapidly and predictably commissioned. The service runs within Amazon’s proven network infrastructure and data centers. The Amazon EC2 Service Level Agreement commitment is 99.99% availability for each Amazon EC2 Region.

7) Completely controlled: Users have complete control of their instances including root access and the ability to interact with them. Users can stop any instance while retaining the data on the boot partition, and then subsequently restart the same instance using web service APIs. Instances can be rebooted remotely using web service APIs, and users also have access to their console output.

VI. Results

In this section we present the results obtained from the implementation of our distributed system that uses the Chandy-Lamport algorithm to capture consistent snapshots.

Our main goal was to test if the Chandy-Lamport algorithm could capture consistent snapshots in a distributed system of nodes. To validate this, we ran our application multiple times, with different numbers of processes. At each execution, with the help of the log files, we examined the captured states of the nodes and the messages in transit during the execution of the snapshots.

We also tested the system's ability to manage multiple snapshots initiated simultaneously from different nodes (see “globalSnapshot_test.go” file). It has been proven that the algorithm works well in scenarios with multiple initiators, being able to capture consistent snapshots in each case without interference between parallel executions. This confirms the robustness of the Chandy-Lamport algorithm in handling complex situations in distributed systems.

VII. Discussion

In this paper, we have presented an implementation of the Chandy-Lamport algorithm in the Go programming language for capturing consistent global snapshots in distributed systems. Through our experimental evaluation, we have demonstrated the effectiveness of the algorithm in maintaining snapshot consistency across different workload conditions.

Our results indicate that the Chandy-Lamport algorithm performs well in capturing global snapshots in distributed systems with scalable performance and reliability. The implementation successfully achieves the objectives set at the beginning of the project, providing a robust solution for recording consistent global states in distributed applications.

The main challenge was faced during the execution of the program. Indeed, it was not easy to connect the testing application with the multiple application components of the nodes, because of the use of the "RPCDial" function provided by the "govec" package of the GoVector library. Essentially, this feature did not allow messages sent over the network to be correctly encoded. To solve this problem, instead of this function, we used the "Dial" from the "net/rpc/jsonrpc" Go library.

Regarding the development of the solution, the biggest difficulty was in correctly using the Go channels to manage the snapshot process.

CONCLUSIONS

In this project it was possible to develop a library to obtain the global state of a distributed system. This helped to progress the subject knowledge of vector clocks, global states, and distributed debugging.

During the development, we have verified the usefulness of a graphical tool such as ShiViz because it allows to observe the transit of messages between nodes. Moreover, it ensures to detect an error in a much simpler way than, for example, examining the files log of each node.

To further improve the quality and reliability of the system, we should conduct additional tests, such as integration, robustness and stress tests.

The implications of our work extend beyond the specific implementation of the Chandy-Lamport algorithm. Our findings contribute to the broader understanding of distributed algorithms and their practical applications. Our work provides valuable insights and lays the groundwork for future research advancements in this area.

REFERENCES

[1] Leslie Lamport, "Time, clocks, and the ordering of events in a distributed system." Communications of the ACM 21.7 (1978): 558-565.

[2] Wikipedia, "Chandy-Lamport Algorithm." Available online: https://en.wikipedia.org/wiki/Chandy-Lamport_algorithm

[3] Chandy, K. Mani, and Leslie Lamport, "Distributed snapshots: determining global states of a distributed system." ACM Transactions on Computer Systems (TOCS) 3.1 (1985): 63-75.

[4] Lindsey Kuper, “CSE138 (Distributed Systems) L6: Chandy-Lamport snapshot algorithm,” [Video]. Youtube, Apr. 17, 2021. Available online: https://www.youtube.com/watch?v=WK3FuD7f9g8

[5] GeeksforGeeks, “Chandy-Lamport’s Global State Recording Algorithm.” Available online: https://www.geeksforgeeks.org/chandy-lamports-global-state-recording-algorithm/

[6] Go Programming Language, "Official Documentation." Available online: https://golang.org/doc/

[7] MatteoBasili, "chandy-lamport-project," [GitHub Repository], 2024. Available online: https://github.com/MatteoBasili/chandy-lamport-project

[8] DistributedClocks, “GoVector,” [Github Repository], 2014. Available online: https://github.com/DistributedClocks/GoVector

[9] V. Cardellini, "Sincronizzazione nei Sistemi Distribuiti," Corso di Sistemi Distribuiti e Cloud Computing, Università degli Studi di Roma Tor Vergata, Macroarea di Ingegneria, Dipartimento di Ingegneria Civile e Ingegneria Informatica, A.A. 2023/2024. [Slide Presentation].

[10] D. Falessi, "Introduction to State Diagrams," Corso di Ingegneria del Software e Progettazione Web, Università degli Studi di Roma Tor Vergata, Macroarea di Ingegneria, Dipartimento di Ingegneria Civile e Ingegneria Informatica, A.A. 2021/2022. [Slide Presentation].

[11] Bestchai, “Shiviz,” bestchai.bitbucket.io. Available online: https://bestchai.bitbucket.io/shiviz/

[12] Docker Compose, "Documentation." Available online: https://docs.docker.com/compose/

[13] Amazon Web Services, "EC2 Documentation." Available online: https://aws.amazon.com/documentation/ec2/

[14] Docker, “Dockerfile reference,” Docker Documentation. Available online: https://docs.docker.com/reference/dockerfile/

[15] Amazon Web Services, “Getting Started with Amazon EC2.”Available online: https://aws.amazon.com/it/ec2/ec2-get-started/