Chandy-Lamport algorithm for global states

Networks and Distributed Systems

Pr´actica 1

Author

Marius Sorin Crisan - 721609

SCHOOL OF ENGINEERING AND ARCHITECTURE 2020-2021

´Indice

1. Introducci´on

In this work, the design, implementation and validation of a library for
obtain global states of a distributed system. The solution provided is based on the algorithm
by Chandy-Lamport [1]. To validate the library, ShiHiz [2] will be used, a
distributed debugging, based on vector clocks[3]. The definitive test will be carried out in distributed
on three machines from the university's 1.02 laboratory.

2. Development

In this section we will describe the architecture of the system once deployed in distributed
as well as the types of messages that are sent between the different components and nodes of the system.
The behavior of the system when sending a message to another node and obtaining the message is also described.
global state.

2.1. Architecture

All nodes in the system are connected to each other through a point-to-point TCP connection.
Through this connection, both application messages and related information are sent.
to the library developed in this practice. The system configuration/structure is specified in a
json file. This file contains the name, IP, port, etc. of each of the nodes that form the
distributed system.

In the design phase, the decision was made to develop a library that was as decoupled as possible.
possible from the main application. As a consequence, the software of each node is composed of
three well-differentiated components: application, node or messaging service and the SnapNode service
for obtaining global states (Fig. 1).

The application is the simplest of the three components since it is an RPC server that
exposes two methods: one to send a group or individual message (SendGroup) and the other to
start the process of obtaining the global state (MakeSnapshot). At the end of the latter, the
global status in the corresponding log file. Additionally, this component has a goroutine
that receives messages sent by other nodes to the node in question and records them in the file
of log mentioned above so that the user has proof of it.

The Node component works as a broker or messaging service.
On the one hand, it listens on a user-specified TCP port to receive messages from others.
nodes. The information received may be a flag, an application message, or the local state of a
node. In all three cases, it sends this information to the SnapNode component (channels chRecvMsg Mark
and chRecvAllState) since it will be used by the algorithm for obtaining global states [4]. Besides,
If it is an application message, it is sent to the App component since it will be used by

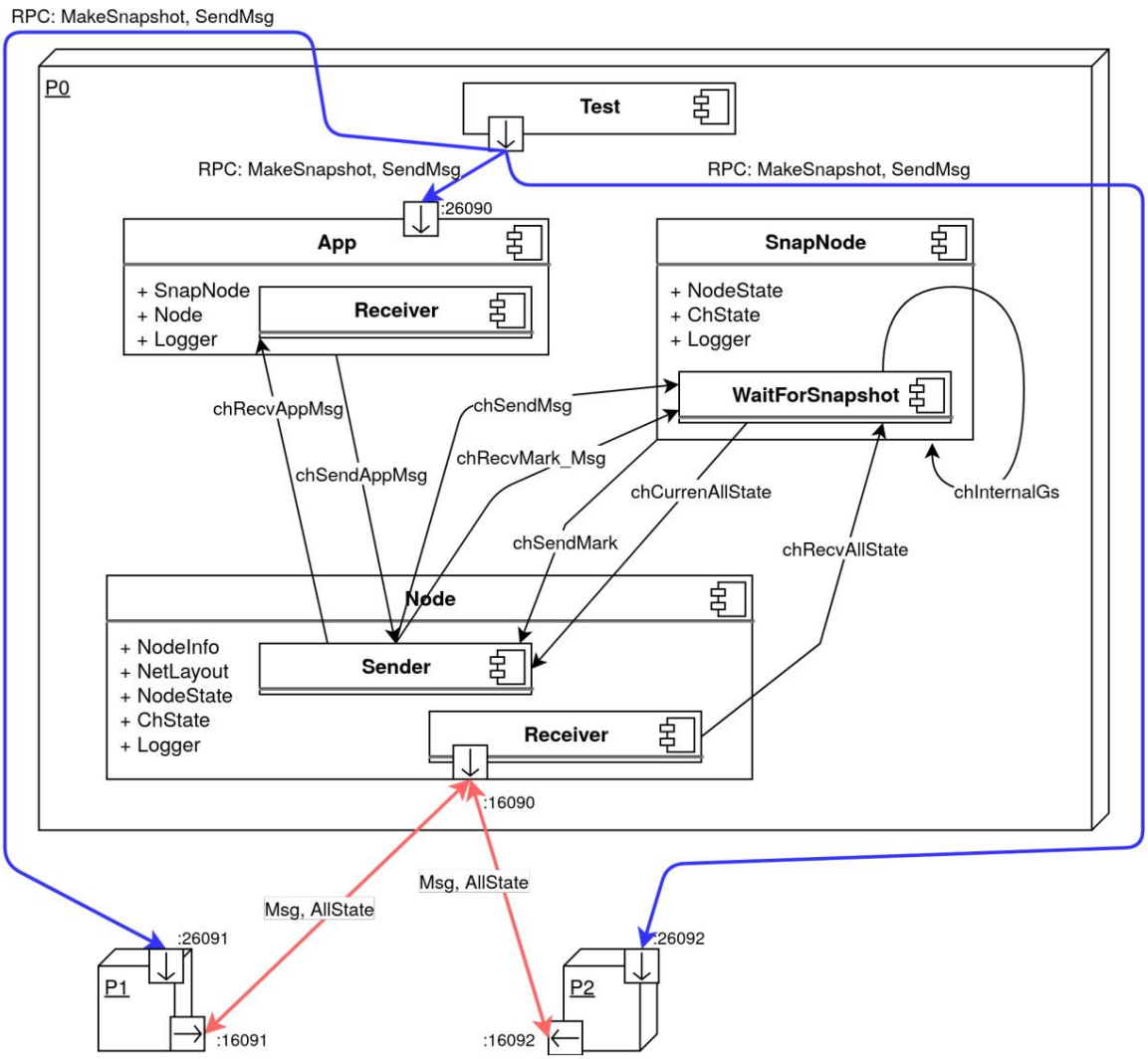

Figure 1: System deployment diagram in three nodes.

this (e.g. to show it to the user).

On the other hand, the sender service is another goroutine that serves requests from the application and the component SnapNode. The only request that can be made from the application is to send a message to another node (chSendAppMsg). Upon receiving this request, the message is sent to the corresponding node using the tcp connection and also to the SnapNode. The possible requests generated by SnapNode are to send to rest of nodes a mark (chSendMark) or the local state (chCurrentState).

Finally, the SnapNode component is responsible for carrying out the process of obtaining the global state. Receives sent/received messages, flags and local statuses from other processes by means of the chSendMsg, chRecvMark Msg and chRecvAllState channels, respectively. To notify sending local state or a mark uses the chSendMark and chCurrentAllState channels.

Regarding system debugging, a logging system has been configured for all the components of the same node. In this way, information about the execution of the project can be recorded. system with four different logging levels (trace, info, warning and error). When the battery runs out

designed test procedure (See sec. 3), a bash script combines the files generated on each node into a single file and orders the events based on the local time of the node in which said event has been registered.

Regarding the type of data sent by the network, it can be a message, a mark or the status location of a process (Fig. 2). All this information is sent over the same TCP channel used by the application. The GoVector library transforms this information when sending it to add the vector clock used for debugging.

The message is composed of the name of the sender node and the body of the message. The only one The difference between an application message and a brand is its body, since the body of a mark contains only the string ”MARK”.

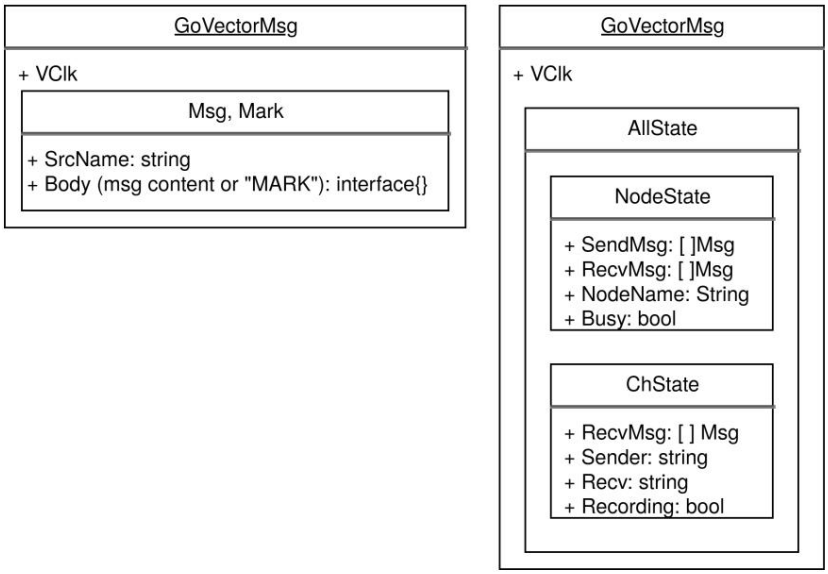

Figure 2: Types of messages sent between nodes.

Regarding the local state, it is composed of the state of the node and the incoming channels. Each of the incoming channels store the messages received since the obtaining process began of the global state (postrecording). On the other hand, sent and received messages are stored in the node from the last global state and until the beginning of the process (prerecording).
All recorded messages are stored in such a way that the sender can be obtained and recipient of any of them.

2.2. Behavior

Figure 3 shows the state diagram of each of the components in each node. Regarding the application, the receiver (bottom) performs an action repeatedly: it receives messages and saves them in the log file. The main thread of this component receives requests RPC. These can be sending a message or getting global status. Upon receiving the latter, one invokes to the MakeSnapshot method of the SnapNode component to start the algorithm.

In the SnapNode, upon receiving the MakeSnapshot request, the current state of the node is saved, that is, the messages sent and received since the last snapshot and the messages received begin to be recorded for the rest of the channels. It then notifies the Node component that a flag must be sent (the first) and remains blocked until it receives, through the chInternalGs channel, the global state resulting.

The WaitForSnapshot goroutine waits for messages from the rest of the nodes. If it is the first brand you receive saves the state of the local process. Once the marks of all the nodes in the system have been received, it sends its local state to the rest through Node and waits until it receives the local state of everyone the processes. Finally, if it is the process that started the algorithm, send the global state to the thread principal.

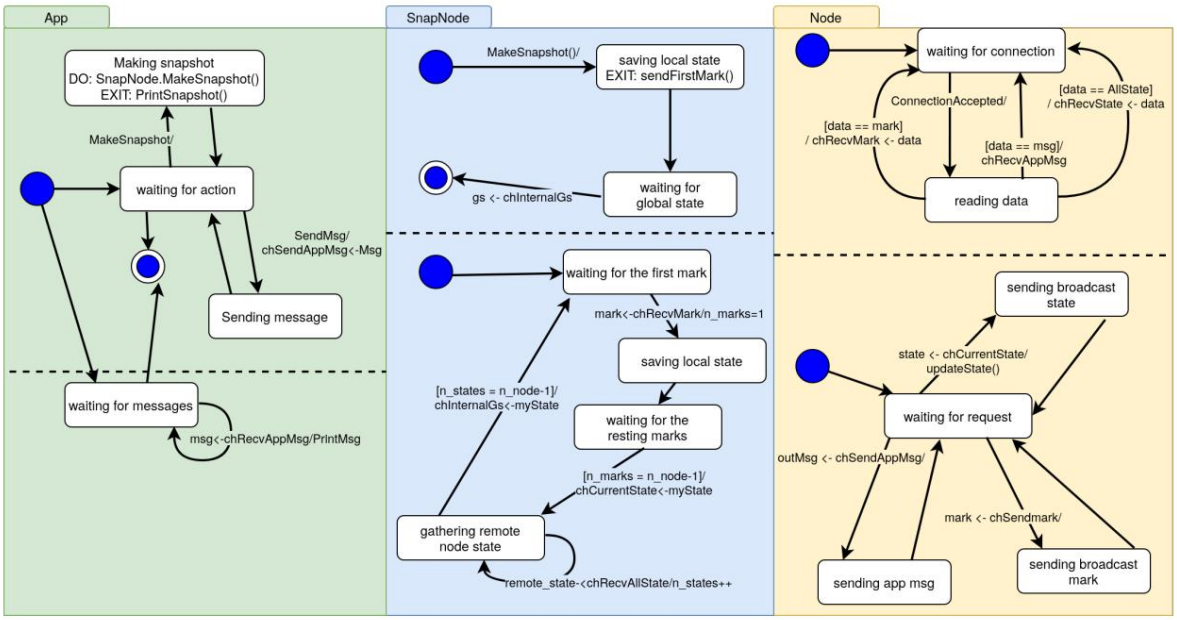

Figure 3: State diagram of a system node. On the left is the application component, in the center the Snapnode and on the right the Node.

As for the Node component, the Receiver goroutine receives messages from other nodes and processes them. All received messages (flags, application messages and local states) are sent to the SnapNode. Additionally, if it is an application message, it is sent to the application as well.

The Sender goroutine receives requests from both the application (send messages) and the SnapNode (brands and local state). Messages are sent only to the nodes specified in it, while that flags and states must be sent to all nodes in the system.

3. Tests

To run the tests from a single machine and in a simple way, use the library Go RPC. To do this, a Go test binary is run on a single machine. This program reads information related to the structure of the distributed system from a json file. Next, establishes an ssh connection with each of the nodes and runs the application (App component).

After an initialization time, the application will register an RPC server. Afterwards, the program test opens an RPC connection with each of the nodes. These connections are used to send messages between the nodes or start the process of obtaining global states of the system (Fig. 1). In all the tests carried out, three nodes are used and their operation has been verified both locally and on 3 different machines in laboratory 1.02.

In the first test the global state is obtained without any application message sent. In This case you get a global state with no message stored.

In the second test, node P0 sends a message to P2 and then obtains the global state. To the To complete this test, the status of P0 should contain the message as sent and the status of P2 as received. This test verifies that the state of the nodes is stored correctly.

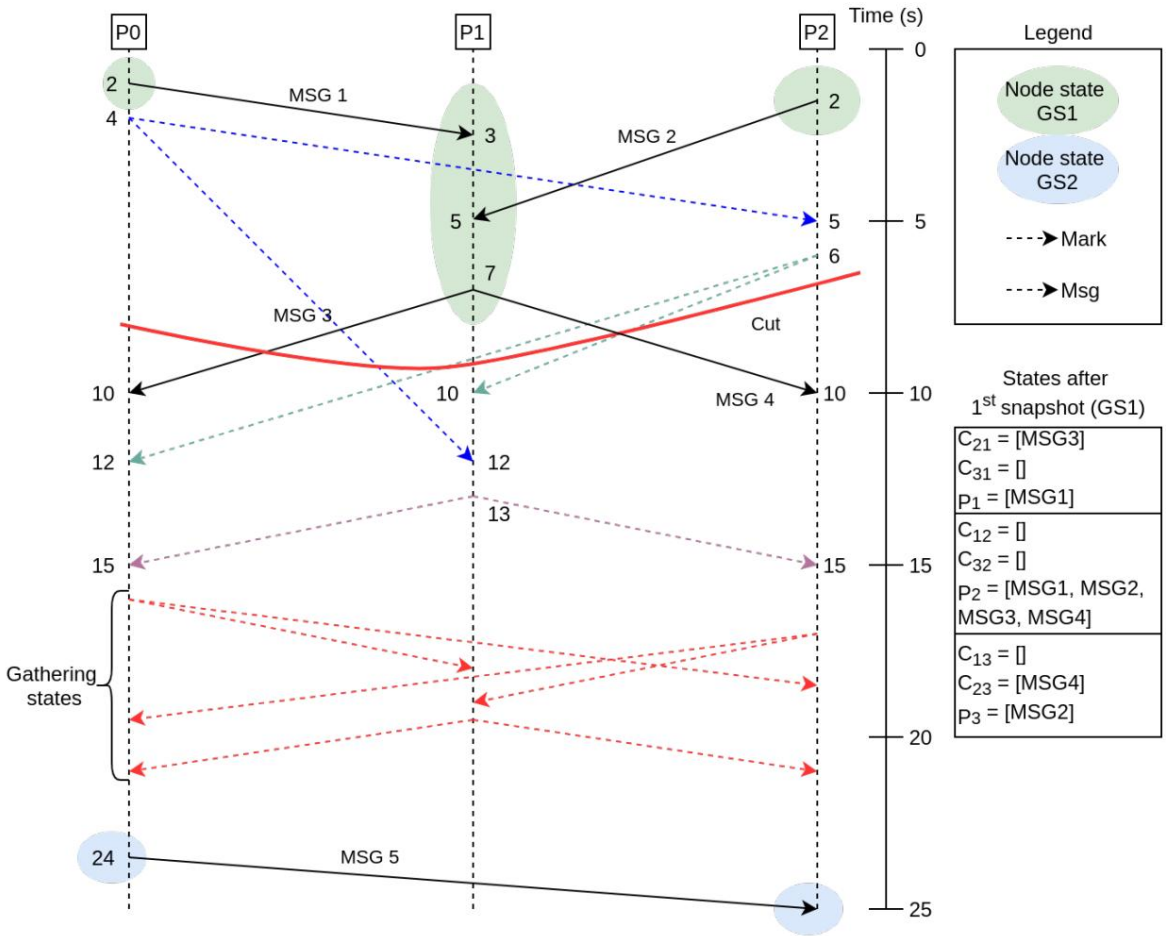

Figure 4: Test case with messages in transit during obtaining the global state.

The last test aims to verify that the messages in transit during the execution of the algorithm Chandy-Lamport are stored correctly. To do this, it is proposed to obtain the execution trace of figure 4. The numbers that appear next to the messages correspond to the time elapsed

from the beginning of the test. These temporary stamps mark the approximate times at which must receive or send a network message. To achieve this scenario, delays are introduced in the messages. Therefore, it must be taken into account that the execution trace of this test may vary depending on network saturation.

Through “MSG 1” messages [2] ”MSG 2” is verified again that the status of the node. With the messages ”MSG 3 [2] "MSG 4" checks that the states are correctly recorded of the channels when there are messages in transit. The first global state is displayed in the box on the right part of figure 4. When sending the message ”MSG 5” the global state has already been obtained. Finally, the global state is obtained again (it does not appear in the graph) to verify that only They store in the local state of the node the messages that have occurred after the last global state.

Looking at Figure 5, it can be seen that the execution trace of test 3 is very similar. to the intended (Fig. 4).

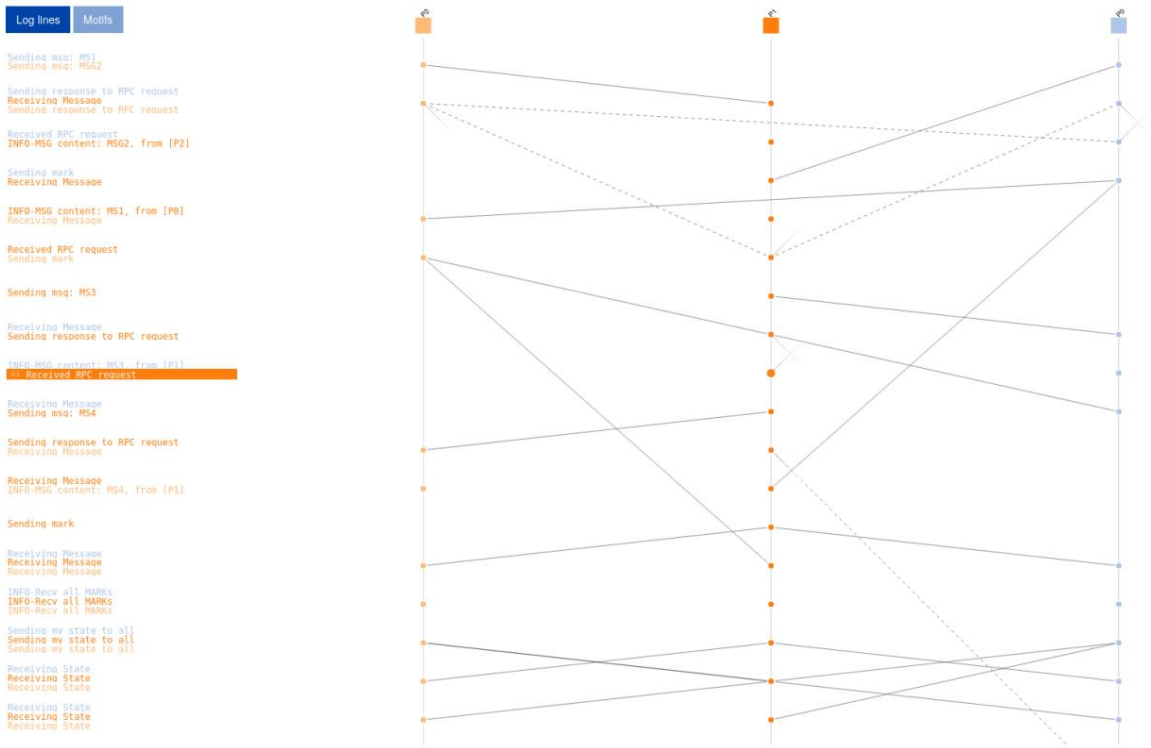

Figure 5: ShiViz trace from test run 3.

4. Difficulties encountered

The main difficulty encountered was when compiling the program since there was a problem of incompatibility between the version of GoVector and the msgpack library used by it. To solve The version of the msgpack library had to be changed to one from December 2018 and modified in the source code two functions.

Regarding the development of the solution, the greatest difficulty has been the synchronization of the messages sent between nodes for the third test carried out.

5. Conclusions

In this practice it has been possible to develop a library to obtain the global state of a distributed system. This has helped to consolidate the knowledge taught in the subject about vector clocks, global states and distributed debugging.
During its development, the usefulness of a graphic tool has been verified.
like ShiViz that allows you to observe the message transit between nodes since it is much easier detect an error, for example, by looking at the log files of each node.

6. References

[1] K Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. ACM Transactions on Computer Systems (TOCS), 3(1):63–75, 1985.

[2] Ivan Beschastnikh, Patty Wang, Yuriy Brun, and Michael D. Ernst. Debugging distributed systems: Challenges and options for validation and debugging. Communications of the ACM, 59(8):32–37, August 2016.

[3] Ivan Beschastnikh. Github.com/govector, October 2018.

[4] Lindsey Kuper. An example run of the chandy-lamport snapshot algorithm, April 2019.