Distributed Systems and Cloud Computing B3 Project: Election Distributed Algorithms

Mattia Di Battista (0304938)
Università degli studi di Roma "Tor vergata"
Rome, Italy
mattia.dibattista@alumni.uniroma2.eu

***Abstract*—Election distributed algorithms are a specific implementation of consensus distributed algorithms in which the aim is to find a leader. In this work, Chang and Roberts, and Bully algorithms are implemented with several services. Furthermore, we deploy the whole decentralized network on Docker containers and execute it on an AWS EC2 instance.**

***Index Terms*—Ring-based algorithm, Bully algorithm, TCP, Docker, AWS EC2, Ansible**

I. Introduction

This work aims to implement two election distributed algorithms, make a decentralized network of nodes using **Docker** containers and execute the whole application on an **AWS EC2** instance. In the following, we describe the services used by the algorithms, the implementation of the algorithms (i.e., **Chang and Roberts** and **Bully** algorithms), the whole architecture where to perform them, and three tests used to demonstrate the operation.

II. Services

A. Register Service

The **register service** provides knowledge of the entire network to all nodes belonging to it. Moreover, it generates a unique random ID for each member of the topology (see VII).

The register node listens by default on the **TCP** port *1234* (it can be changed from *SDCC/sdcc/config.json*). The listening window is kept open for a *SOCKET_TIMEOUT* period (defined in *SDCC/sdcc/register/src/constants.py*) after which a node receives the member's network list.

Initially, every node has an ID equal to *DEFAULT_ID* (by default set to *-1*). Instead, the register is identified by value *0*. After the register phase, a unique ID to each node is associated.

A node generates multiple sockets for whole its activity. During the register phase, two sockets are created: the first used to communicate with the register node and the second one used later to listen to eventual packets from other nodes. IP address, port, and ID of this latest must be sent before its use to make known to other nodes that information.

B. Heartbeat

The **heartbeat service** allows detaching crashes by the coordinator nodes.

The service is kept active using a thread that sends heartbeat messages to the leader through a dedicated **TCP** socket (different from the two defined in II-A). After the heartbeat message the thread waits for a while (i.e., *TOTAL_DELAY* period defined in *SDCC/sdcc/node/src/constants.py*). If the timeout occurs a crash is found out and consequently a new election is started. Only the ACK message indicates a running coordinator.

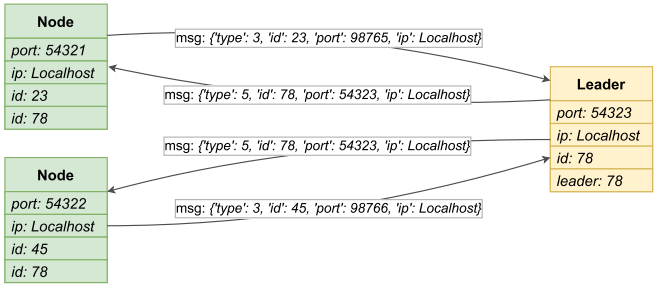

Fig. 1. Heartbeat service invoked by two nodes.

Heartbeat messages are sent periodically based on the *HEARTBEAT_TIME* constant. If the received message has a different type from HEARTBEAT or the sender is not the current coordinator the packet is ignored and the thread keeps listening for the remaining time.

C. Verbose

The **verbose flag** shows all messages exchanged (i.e., received, sent) throughout the node lifetime[1] (see VII). To activate the verbose flag is necessary to pass the *-v* parameter from the command line (see VI).

The main info shown are:

- Timestamp
- Node info (i.e., IP address, port number, ID)
- Receiver/Sender info (i.e., IP address, port number, ID)
- Message content

The **Logging** library is used to define the message syntax.

```
def set_logging() -> logging:
    logging.basicConfig(
        level=logging.DEBUG,
        format="[%(levelname)s]
            %(asctime)s\n%(message)s",
        datefmt='%b-%d-%y %I:%M:%S'
    )
    return logging
```

[1]Also register node implements this service.

Method to instantiate a logging object.

D. Delay

To stress more the system, the **delay method** is introduced in *SDCC/sdcc/node/src/helper.py*. It generates a period waited by the sender to forward the current packet. This may cause the receiver timeout to expire.

```
def delay(flag: bool, ub: int):
    if flag:
        delay = randint(0, floor(ub*1.5))
        time.sleep(delay)
```

It is activated by default when tests are performed (see. IV), but is also configured by the command line through the *-d* flag.

III. Algorithm Implementation

What follows does not describe how the algorithms work, but only how particular aspects are implemented (reference at [1]).

The implementation consists of the abstract class [2] and election distributed algorithms classes that extend the first one.

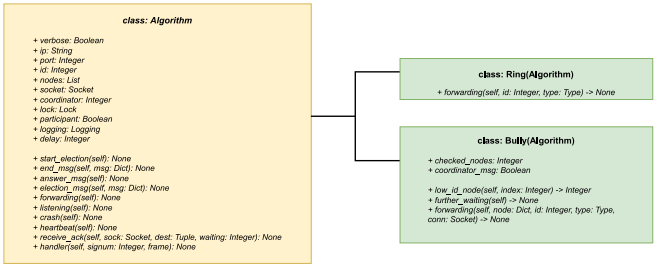

Fig. 2. Logic implementation of the classes.

Six types of messages can be exchanged between nodes:

```
class Type(Enum):
    ELECTION = 0
    END = 1
    ANSWER = 2
    HEARTBEAT = 3
    REGISTER = 4
    ACK = 5
```

ANSWER type is used only by the **Bully algorithm**, whereas REGISTER is sent during register phase.

Both algorithms begin with run the listening thread after which they start an election. Only after completing these two phases, the heartbeat can start.

```
def __init__(self, ...):
        ...
        self.lock = Lock()

        thread = Thread(target=self.listening)
        thread.daemon = True
        thread.start()
        self.start_election()
        Algorithm.heartbeat(self)
```

Many data are accessed simultaneously from multiple threads, thus a **lock** is defined to manage shared resources.

```
self.lock.acquire()
if self.participant or (self.coordinator in
    [self.id, DEFAULT_ID]):
    self.lock.release()
    continue
```

Example of **lock** management in heartbeat method.

A. Chang and Roberts Algorithm

The algorithm is suitable for a collection of processes arranged in a logical ring. Each process has a communication channel to the next one in the ring. All messages are sent clockwise around the ring. The ID's node is used to define the **ring network**: the next node is the one with the greatest ID then current and the lowest among others.

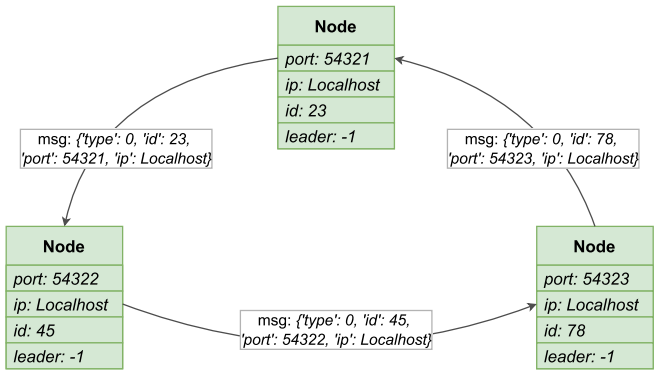

Fig. 3. Election started by node with *id=23* in Ring topology.

When a leader crash occurs or the node's timer associated with a heartbeat message elapses, the leader is removed from the nodes list, therefore remaining nodes can not interact with it even if it is still active.

B. Bully Algorithm

With different respect to the **Ring-based algorithm**, the **Bully algorithm** assumes that each process knows which processes have higher identifiers and that it can communicate with all such processes. That information is sent by the register node (as described in II-A)

1) Scenarios: Unlike in the III-A, for the **Bully algorithm**, the current leader is not removed from the nodes list during leader crash or timeout scenarios.

The considered cases when the leader is executing are:

1) If the leader delays sending the ACK packet, other nodes start a new election that will produce the next coordinator also if the previous one is running[3].

[2]Defined in *SDCC/sdcc/node/src/Algorithm.py*

[3]In the next election it will be elected again.

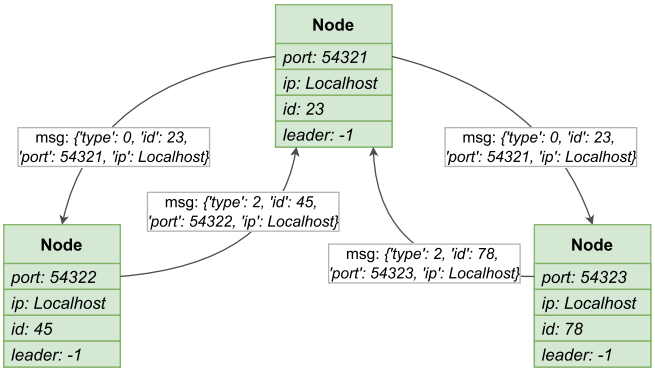

Fig. 4. Election started by node with *id=23* using **Bully algorithm**.

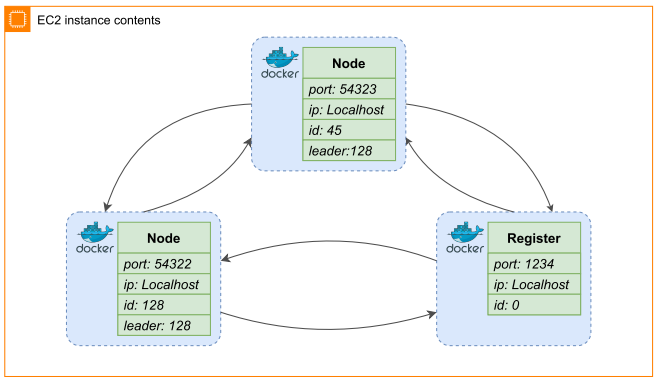

Fig. 5. Deployment using **AWS EC2** instance and **Docker** containers.

2) If the leader is stopped[4] a new election will start. In the meanwhile all messages sent to the sleeping node are queued, so when it wakes up[5] will receive those messages and begin the leader again.

IV. Tests

The following tests are performed:

1) *Test A*: a generic node fails
2) *Test B*: the coordinator fails
3) *Test C*: both generic and coordinator nodes fail

To interrupt a specific node **psutil** library is used. It kills a process that is listening on a certain **TCP** port[6]. That mechanism exploits the sorted list of nodes send by the register. E.g., the fact that the coordinator node occupies the last position in the list is used in *Test B*.

```
def kill_node(self, port: int):
      for proc in process_iter():
         for conns in
             proc.connections(kind='inet'):
            if conns.laddr.port == port:
               proc.send_signal(signal.SIGINT)
```

The network is keep running for a while before the test and after the interruption to show its activity.

Test execution is interactive (see VII) means that the user chooses which test executes sees logging information on the terminal and sets which algorithm has to be performed.

As described in II-D to stress more the application, the delay option is active.

V. Deployment

The network is deployed on an **AWS EC2** instance where every node runs on a **Docker** container. **Docker Compose** is used to automate the container's creation[7].

By default, **Docker Compose** creates a network where containers can communicate with each other, so that is used.

To automate the deployment procedure **Ansible** is used to install **Docker** and to forward the code application on an **EC2** instance. See VII.

VI. How To Use

The application can be run in two different ways:

1) Local execution without **Docker** containers
2) Remote execution on **AWS EC2** instance using **Docker** containers [8](as shown in V)

The complete list of commands is available here.

VII. Running Examples

Fig. 6. Register phase from three generic nodes and register node.

References

[1] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems: Concepts and Design (International Computer Science)*. 4th rev. ed. Addison-Wesley Longman, Amsterdam, 2005. ISBN: 0321263545.

[4]Using $Ctrl + z$ combination.
[5]Using *fg* command.
[6]It requires root privileges (see VI).
[7]See *SDCC/sdcc/docker-compose.yml*, *SDCC/sdcc/node/Dockerfile* and *SDCC/sdcc/register/Dockerfile*.
[8]That execution requires an **AWS** account.

Fig. 7. Example of the message showed by register node.

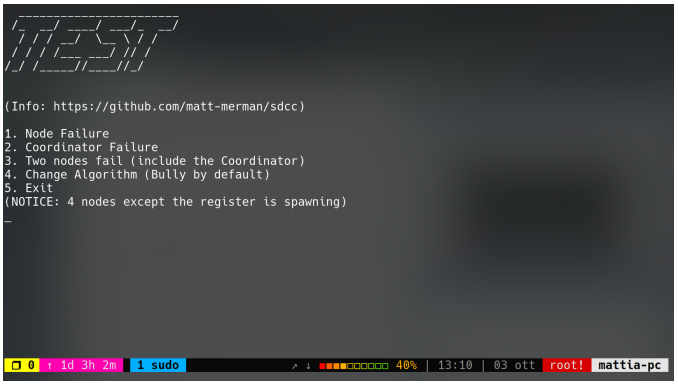

Fig. 8. User interface to tests execution

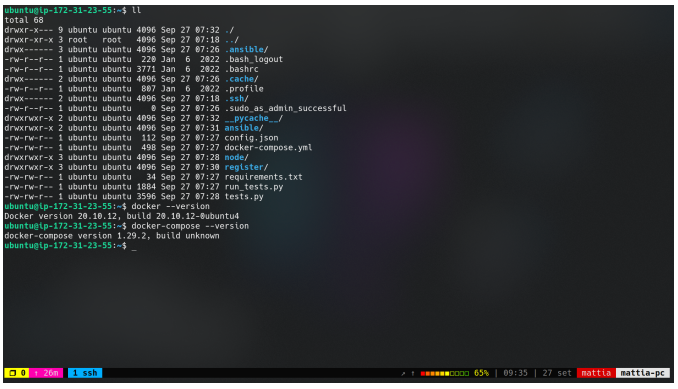

Fig. 9. **Docker**, **Docker Compose** and application code info on an **AWS EC2** instance.