

Sri Lanka Institute of Information Technology

Consensus Based Prime Number Deciding Distributed System

SE5090

Distributed Computing

Assignment

Submitted By:

MS23017856

WICKRAMAARACHCHI W.A.P.C.K

2023 May

# The way of selection of the use case from index number

Index Number : MS23017856

Summation of the index = 2 + 3 + 0 + 1 + 7 + 8 + 5 + 6

= 32

= 3 + 2

= 5

Modulus value = 5%2

= 1

**Use case 1 selected**

# Outline of Approach

* Familiarized with the concepts of distributed systems, consensus algorithms, and prime numbers.
* Chose the Python programming language which comfortable with, and started by implementing a single-node version of the prime number checking algorithm as a Flask application.
* Designed and implemented the distributed version of the algorithm. Considered which consensus algorithm to use, and how to handle node registration and leader election.
* Created a file containing random 5-digit numbers, both prime and non-prime, for testing purposes.
* Implemented the service registry using Consul, and ensured that all nodes could register themselves and discover each other.
* Implemented the Bully algorithm for leader election, and ensured that nodes could elect a leader and begin the prime number checking algorithm.
* Implemented the master node's role, which involved determining the active nodes, creating a schedule for the ranges to be solved by each node, and assigning roles to proposers, acceptors, and learners.
* Implemented the proposer node's role, which involved waiting for schedules from the master, solving their assigned ranges, and sending messages to acceptors.
* Implemented the acceptor node's role, which involved receiving messages from proposers, verifying the validity of the message, and sending the result to the learner node.
* Implemented the learner node's role, which involved counting the number of messages sent by acceptors and deciding whether the number was prime or not.
* Tested the system thoroughly, using the file of random numbers which created earlier, and made sure that all nodes were working correctly and communicating properly.
* Finally, added logging to the system using a side-car proxy and RabbitMQ server and, used API as the communication protocol between nodes.

# Solution Design

The overall application is a distributed consensus algorithm using the Paxos protocol. It has been built as a Flask Application, which exposes APIs that can be used by different nodes in the distributed system to communicate with each other. The Flask Application runs on a specified port number, which is obtained from the command-line arguments. The node name is also obtained from the command-line arguments, which is used to generate a unique node ID.

The application uses RabbitMQServer as a sidecar to handle logs. RabbitMQ is a message broker that enables communication between distributed applications. In this application, RabbitMQServer is used to send and receive messages between different nodes in the distributed system.

Consul has been used as the service register to register each node in the distributed system. Service registration is an essential part of the distributed system, which enables each node to discover and communicate with other nodes in the system. Consul provides a simple and flexible way to register and discover services in a distributed system.

# 3.1 Initializing Node

The application begins by accepting user inputs for the port number and node name and assigning them to respective variables. A unique random node ID is then generated using the UUID1 algorithm, which generates a 64-bit integer. An object is created from the Node class using this ID and the given node name and port number.

The node is then registered in the service registry using Consul. If a master node is already assigned, the node proceeds to run its specific tasks. However, if the master node is not registered, the node announces the start of an election process.

# 3.2 Master Election

In case a master node is not yet selected, an election is announced for all nodes in the system. As nodes send their election messages to higher nodes, all requests are directed to the proxy. Since the initiate\_node method only needs to execute once, the proxy will forward exactly one request to the response-API. Using the bully algorithm, the node with the highest ID is elected as the master. First, all nodes' details are obtained, and any ID higher than the current node's ID is appended to a list. Then, it is checked whether the list is empty for all nodes. If any node's list is empty, that node is the node with the highest ID and becomes the master node. Finally, the master node is announced to all other nodes, and the election process ends.

# 3.3 Role Distribution

Once the master node is selected, it first checks whether the minimum number of active nodes, excluding the master node, is 4. If the minimum number is not met, the master node waits for other nodes to start. Once the minimum number of nodes is met, the master node assigns roles to each node. There are always 2 acceptors and 1 learner, and the rest can be proposers that solve the ranges.

# 3.3 Task Sheduling

After assigning roles, the master node will read a file and obtain a number from it. It will then divide the number by the number of assigned proposers to determine the approximate range length, and generate equal ranges with this length. The number and ranges will then be sent as a JSON-encoded message to the corresponding proposer node, using the format "your range [{"number":"54322","range":"40740:54320"}]". This message will trigger the proposer to begin its process. The Leader node will wait until the learner sends back its decision before selecting the next number in the file, which is done randomly.

# 3.4 Proposer

After receiving the number and range from the master, the proposer will verify whether the number falls into the prime category by dividing it with numbers within the given range. If it is not a prime number, the proposer will send a message containing "acceptor\_values [{"number":"54322","prime\_status":false,"divisor":346}]"

along with the corresponding divisor. If it is a prime number, the proposer will send a message containing "acceptor\_values-[{"number":"54322","prime\_status":true,"divisor":””}]" to a randomly selected acceptor node that is found by checking the node's metadata in the service registry. This message will be written to the node's output file, after which the proposer will wait for the next number and range from the master node.

# 3.4 Acceptor

Upon receiving the aforementioned message, the Accepter will initiate its procedure. If the message indicates that the prime status is false, the Accepter will cross-check it with the divisor to ensure that the number is indeed not a prime number. After the confirmation, it will send the prime status along with the number to the learner as "learner\_values- [{"number":"54322","prime\_status":true}]" and wait for the next number from the master. The message sent will also be recorded in the output file of the node.

# 3.4 Learner

The Learner receives a message from the Accepter, which is then stored. If the number of messages received is equal to the number of proposers, the Learner will analyze the stored messages for prime status. If the analysis results in "true", the Learner will then check if all the proposers have sent their messages. If they have, the Learner will determine that the number is a prime number. If the proposers have not all sent their messages, the number is not considered to be a prime number. The Learner will then inform the Leader node by sending either "leader\_values-{{number}} is a prime number" or "leader\_values-{{number}} is not a prime number". Additionally, the Learner will write the final result to the output file, which will display either "final results number: -{{number}} is not a prime number" or "final results number: -{{number}} is a prime number". For example, if the number is 25919, the output file would display "final results number: 25919 is a prime number".

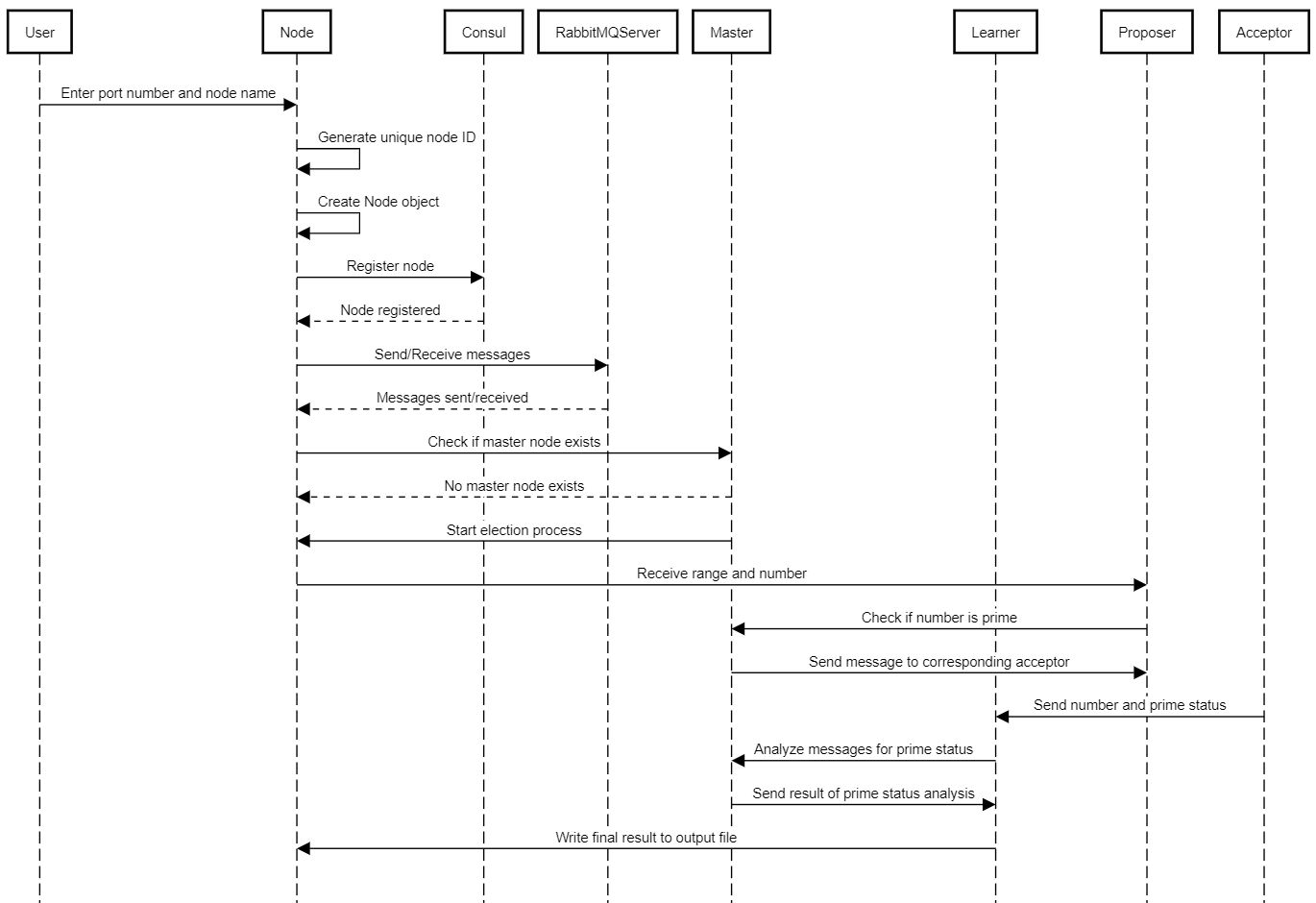
# Sidecar implementation

A sidecar is a helper process that runs alongside a main application and provides additional functionality, such as logging, monitoring, and security. In our distributed consensus algorithm application, we have used a sidecar for logging.

For this purpose, we have used RabbitMQ as the sidecar to handle the logging of the application. RabbitMQ is an open-source message broker that implements the Advanced Message Queuing Protocol (AMQP). It allows for decoupling the sender and receiver of messages, making it ideal for logging in a distributed system.

The RabbitMQ server runs as a sidecar alongside the main application. The application sends log messages to the RabbitMQ server, which in turn stores them in a message queue. This ensures that logging is not a bottleneck in the main application and also provides a centralized log repository that can be easily queried and analyzed.

# Sequence diagram



# Failure scenarios and fault tolerance.

Network failures: Since the application relies on network communication between different nodes, network failures can cause communication errors, message loss, or delays in message delivery.

Node failures: Nodes can fail due to various reasons like hardware failure, software bugs, or network disruptions. Node failure can cause issues like message loss, leader unavailability, and data inconsistency.

Service failures: The service registry, RabbitMQ, or Flask application can fail due to various reasons like hardware failure, software bugs, or network disruptions. Service failure can cause issues like node unavailability, message loss, and data inconsistency.