INTER-PROCESS COMMUNICATION – BULLY ALGORITHM

Using Python

Ahmed Sheta.

Table of Contents

Introduction

To create an application that fulfills the given requirements, I had to learn about IPC and distributed systems as it was outside of my domain. The first challenge is to go back to my university lectures as well as sweeping through YouTube to familiarize myself with the fundamental knowledge to get started on the task, even if its just an introduction.

Task Synopsis

To create an application that fulfills the given tasks, we will need to:

1. Learn about IPC and choose an appropriate mechanism for our program.
2. Implement a program that can communicate with other instances of itself via IPC.
3. Implement the Bully Election Algorithm to elect a coordinator from among the instances.
4. Each instance of the program should run as a separate console application and be able to communicate with each other.
5. Each instance should have a unique identifier.
6. Instances should print to console a time-stamped message for any kind of communication.
7. New instances should wait to detect the current existing coordinator at a pre-configured interval and declare themselves as a coordinator if one is not detected.
8. The coordinator instance can be closed at any time, which should trigger a new election between all other running instances.
9. New instances can be run at any time, even during an election.

IPC Mechanism Used

To implement this application, we will need to use Python's IPC mechanisms. There are many IPC mechanisms available in Python, including pipes, sockets, and shared memory. For this project, we will use message passing as the IPC mechanism.

I have used message passing to facilitate communication between processes and coordinate their actions. Specifically, the processes communicate with each other by sending and receiving messages, allowing them to determine which process should become the coordinator and which process has the minimum value in its assigned chunk of the array.

Without message passing, the processes would not be able to share information with each other and coordinate their actions. Each process would only be able to act on its own local data and could not determine the overall status of the system. By using message passing, the processes can work together to achieve a common goal, which in this case is finding the minimum value in the array.

Algorithm and Design

The Bully election algorithm is a distributed algorithm for leader election that enables a group of processes to elect a leader or coordinator. This algorithm involves assigning a unique ID to each process, and the process with the highest ID is designated as the coordinator. If a process detects that the current coordinator is not responding, it initiates an election message to all the processes with higher IDs, and the process with the highest ID becomes the new coordinator.

Our implementation of the Bully election algorithm leverages IPC (Inter-Process Communication) through message passing. We have utilized Python's multiprocessing module to create multiple processes and Lock to prevent race conditions during the election of the coordinator. Each process is equipped with a unique ID and a list of all **NodeWrapper** instances in the system.

The **NodeWrapper** class represents a process and contains methods for message sending and receiving, initiating, and waiting for an election, and dividing the array into chunks to find the minimum value. The ***send_message***() method sends a message to a specified process using the receiver's process ID, while the ***receive_message***() method receives messages from other processes and processes them according to their content.

At the process's startup, a **NodeWrapper** instance is initialized with its process ID, list of all processes, and the array to be divided. It also sets a message handler for incoming messages, which handles messages related to the election, such as starting an election, receiving an election message, and declaring a coordinator. Additionally, it divides the array into chunks, finds the minimum value in the assigned chunk, and sends it to the coordinator.

In the event that a process detects that the coordinator is not responding, it initiates an election by calling the ***start_election***() method. This method sends an election message to all processes with higher IDs and waits for a coordinator to be elected. If the process receives a coordinator message, it sets the coordinator to be the sender and calls the ***divide_array***() method.

This implementation of the Bully election algorithm with IPC through message passing allows multiple processes to elect a leader and distribute the workload among them. It also ensures that the elected coordinator has the highest ID and that the election process is free from race conditions.

Bonus Task

we used the NodeWrapper class to represent each process and handle the distribution of tasks among them. After the coordinator is elected, it creates a large array with random numbers and divides the array into chunks, one chunk for each instance, using the ***divide_array***() method.

Each instance then receives a chunk of the array and finds the minimum value in the assigned chunk using the ***find_minimum***() method. The process sends the minimum value back to the coordinator using the ***send_message***() method.

The coordinator receives the minimum value from each process and stores it in a list. Once all the processes have responded, the coordinator finds the minimum value in all the responses using the ***min***() function and displays it to the user.

Overall, the approach was to divide the workload among the processes using message passing and ensure that each process returns its result to the coordinator. The coordinator then combines the results to find the minimum value in the entire array. The implementation also ensures that the election process is safe from race conditions using locks and that each process handles incoming messages appropriately.

Testing

The three test classes are related to the Python IPC app which implements the Bully Algorithm.

The first class, **TestBully**, contains tests for the **Bully** class, which is responsible for running the Bully Algorithm. It has three tests:

test_main_valid_input(): tests the main function of the **Bully** class with valid input.

test_main_invalid_input(): tests the main function of the **Bully** class with invalid input.

test_start(): tests the start function of the **Bully** class.

test_collect_results(): tests the collect_results function of the **Bully** class.

The second class, also named **TestBully**, contains tests for the **Bully** class but with a different focus:

test___init__(): tests the initialization of the **NodeWrapper** object.

The third class, **TestNodeWrapper**, contains tests for the **NodeWrapper** class, which represents a node in the distributed system. It has four tests:

test_start_election(): tests the ***start_election()*** function of the **NodeWrapper** class.

test_divide_array(): tests the ***divide_array()*** function of the **NodeWrapper** class.

test_find_minimum(): tests the ***find_minimum()*** function of the **NodeWrapper** class.

test_send_message(): tests the ***send_message()*** function of the **NodeWrapper** class.

Source Code Documentation

The code includes three classes: main.py, BullyAlgorithm.py, and NodeWrapper.py.

Class: *main*.py

Method: main()

The *main()* method initializes the program by taking an input from the user for the number of processes to be created. It validates the user's input and creates a '**Bully'** object with the given number of processes. The Bully Algorithm is started, and the task is distributed among the processes. The minimum value from each process is collected and printed out.

Class: *BullyAlgorithm*.py

Class: Bully

The **Bully** class has four methods: ***__init__, start, distribute_task, and collect_results.***

Method: ***init***(self, num_processes)

This method initializes the **Bully** object. It creates a list of **NodeWrapper** objects and an array of 1000 random integers. It initializes each **NodeWrapper** object with its process ID, the list of processes, and the array to be processed. The first **NodeWrapper** object is assigned to be the initial coordinator. The result

attribute is a dictionary that stores the results of the computation, and the coordinator attribute is assigned to the first **NodeWrapper** object.

Method: ***start***(self)

This method runs the Bully Algorithm until all results are collected. It waits for 3 seconds before starting the election process. For each **NodeWrapper** object that has no coordinator, it starts the election process. The current coordinator of each **NodeWrapper** object is logged. If all results have been collected, it finds the minimum value and prints it out. The **results** dictionary is reset to be empty. A message is sent to the coordinator to let it know that the results have been collected. It then waits for 3 seconds before distributing the task again.

Method: ***distribute_task***(self)

This method divides a chunk of an array into several bits. It divides the array into chunks of equal size and assigns a chunk to each **NodeWrapper** object. The result is recorded and logged.

Method: ***collect_results***(self)

This method returns the minimum value from each **NodeWrapper**. The minimum value is calculated and returned.

Class: *NodeWrapper*.py

Class: NodeWrapper

The **NodeWrapper** class is a wrapper around a node in a distributed system that allows for communication between nodes using message passing. It has several methods that handle the election process and chunk division of an array among the nodes in the system.

It has four methods: ***__init__, set_message_handler, send_message, start_election, wait_for_coordinator, declare_victory, receive_message, divide_array, find_minimum***

Attribute: **process_id** (int): Current Process ID

Attribute: **coordinator** (int): Stores the ID of the process elected as the coordinator.

Attribute: **processes** (list): Stores a list of all **NodeWrapper** instances in the system.

Attribute: **start_time** (float): Stores the start time of the program.

Attribute: **election_lock** (**multiprocessing.Lock**): A lock to protect the **start_election()** method from race conditions.

Attribute: **array** (list): Stores the array to be divided among the processes.

Attribute: **chunk** (list): Stores the chunk of the array assigned to this process.

Attribute: ***message_handler*** (function): registers the message handler callback function to handle incoming messages.

Method: ***__init__(self, process_id, processes, array):*** Initializes a **NodeWrapper** instance with the process ID, list of all processes, and array to be divided.

Method: ***set_message_handler(self, handler):*** Allows an external object to register a message handler callback function to handle incoming messages.

Method: ***send_message(self, message, receiver_id):*** Sends a message to a specified process.

Method: ***start_election(self):*** Initiates the election process.

Method: ***wait_for_coordinator(self):*** Waits for a coordinator to be elected as the winner.

Method: ***declare_victory(self):*** Declares the current process the coordinator.

Method: ***receive_message(self, message, sender_id):*** Used to pass messages between processes.

Method: ***divide_array(self):*** Divides an array into chunks and assigns a smaller chunk to each process.

Method: ***find_minimum(self):*** Finds the minimum value in the assigned chunk and sends it to the coordinator.