**LAB : 4**

1. **Imports and Function Definitions**:

import socket import threading import time

* This part imports necessary modules: **socket** for socket programming, **threading** for handling multiple connections concurrently, and **time** for timestamp handling.

1. **handle\_slave Function**:

def handle\_slave(slave\_socket, slave\_address, timestamps): try: timestamp = float(slave\_socket.recv(1024).decode()) timestamps.append(timestamp) print(f"Received timestamp {timestamp} from {slave\_address}") except Exception as e: print(f"Error handling {slave\_address}: {e}")

* This function is responsible for handling each slave node.
* It receives a timestamp from a slave, converts it to a float, appends it to the **timestamps** list, and prints a message indicating the received timestamp and the slave node's address.
* If an error occurs during the process, it prints an error message.

1. **synchronize\_clocks Function**:

def synchronize\_clocks(slave\_sockets): try: timestamps = [] threads = [] for slave\_socket, slave\_address in slave\_sockets: thread = threading.Thread(target=handle\_slave, args=(slave\_socket, slave\_address, timestamps)) threads.append(thread) thread.start() for thread in threads: thread.join() average\_timestamp = sum(timestamps) / len(timestamps) for slave\_socket, slave\_address in slave\_sockets: slave\_socket.send(str(average\_timestamp).encode()) print(f"Sent average timestamp {average\_timestamp} to {slave\_address}") except Exception as e: print(f"Error during synchronization: {e}")

* This function coordinates the clock synchronization process.
* It creates a list **timestamps** to store timestamps received from slave nodes.
* For each slave node, it starts a thread to handle the reception of timestamps using the **handle\_slave** function.
* After all threads finish execution, it calculates the average timestamp from all received timestamps.
* Then, it sends the average timestamp back to each slave node.
* Any exceptions occurring during synchronization are caught and printed.

1. **start\_master\_server Function**:

def start\_master\_server(port): master\_server = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) master\_server.bind(('127.0.0.1', port)) master\_server.listen(5) slave\_sockets = [] print("Master node is running and waiting for slaves to connect.") while True: slave\_socket, slave\_address = master\_server.accept() slave\_sockets.append((slave\_socket, slave\_address)) print(f"Slave connected: {slave\_address}") synchronize\_clocks(slave\_sockets)

* This function initializes the master server, binds it to a specified port, and starts listening for incoming connections.
* When a slave node connects, it accepts the connection, adds the slave socket and address to the **slave\_sockets** list, and prints a message indicating the connection.
* It then initiates the clock synchronization process by calling the **synchronize\_clocks** function with the list of slave sockets.
* This function runs indefinitely, continuously accepting connections and synchronizing clocks whenever a new slave node connects.

1. **Imports**:

import socket import time

* These imports are for the **socket** module, used for socket programming, and the **time** module, used for time-related operations.

1. **synchronize\_clock Function**:

def synchronize\_clock(master\_address): try: slave\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) slave\_socket.connect(master\_address) while True: local\_timestamp = time.time() slave\_socket.send(str(local\_timestamp).encode()) synchronized\_time = float(slave\_socket.recv(1024).decode()) print(f"Synchronized time received: {synchronized\_time}") time.sleep(5) except Exception as e: print(f"Error during synchronization: {e}")

* This function is responsible for synchronizing the slave node's clock with the master node's clock.
* It connects to the master node using the provided **master\_address**.
* Inside a loop that runs indefinitely:
  + It retrieves the current local timestamp using **time.time()**.
  + Sends the local timestamp to the master node after encoding it as a string.
  + Receives the synchronized time from the master node after decoding it from bytes to a string and converting it to a float.
  + Prints the synchronized time received from the master node.
  + Sleeps for 5 seconds before repeating the synchronization process.
* Any exceptions that occur during the synchronization process are caught, and an error message is printed.

1. **Driver Code**:

if \_\_name\_\_ == '\_\_main\_\_': master\_address = ('127.0.0.1', 8080) synchronize\_clock(master\_address)

* This part ensures that the **synchronize\_clock** function is executed only if the script is run directly (not imported as a module).
* It specifies the address of the master node (**('127.0.0.1', 8080)** in this case) and calls the **synchronize\_clock** function with this address.

The Berkeley Algorithm is a clock synchronization algorithm designed to synchronize the clocks of multiple computers on a network. It was developed to address the issue of time discrepancies between different machines, which can arise due to factors like clock drift, network delays, and variations in system clock speeds.

The algorithm works by having one node, typically referred to as the master node, collect the time from all the other nodes, called slave nodes. The master node calculates the average time from the collected times and then sends this average time back to each slave node, which adjusts its clock accordingly.

def handle\_slave(slave\_socket, slave\_address, timestamps): try: timestamp = float(slave\_socket.recv(1024).decode()) timestamps.append(timestamp) print(f"Received timestamp {timestamp} from {slave\_address}") except Exception as e: print(f"Error handling {slave\_address}: {e}")

This function **handle\_slave** is responsible for receiving a timestamp from a slave node, decoding it, converting it to a float, appending it to a list of timestamps, and printing a message indicating the received timestamp and the address of the slave node.

Here's a detailed explanation with an example:

* **Function Definition**:
  + **def handle\_slave(slave\_socket, slave\_address, timestamps):**: This function takes three parameters:
    - **slave\_socket**: The socket object representing the connection to the slave node.
    - **slave\_address**: The address (e.g., IP address and port) of the slave node.
    - **timestamps**: A list where the received timestamp will be stored.
* **Try-Except Block**:
  + **try:**: The code inside this block tries to execute the following operations.
  + **except Exception as e:**: If any exception occurs during the execution of the try block, it is caught, and the code inside this block is executed. The exception is stored in the variable **e**.
* **Receiving and Decoding Timestamp**:
  + **timestamp = float(slave\_socket.recv(1024).decode())**: This line receives data (presumably a timestamp) from the slave node using the **recv** method of the socket object. The received data is then decoded from bytes to a string using the **decode** method and converted to a float using the **float** function. The converted timestamp is stored in the variable **timestamp**.
* **Appending Timestamp to List**:
  + **timestamps.append(timestamp)**: The timestamp received from the slave node is appended to the **timestamps** list, which holds timestamps received from all slave nodes.
* **Printing Received Timestamp**:
  + **print(f"Received timestamp {timestamp} from {slave\_address}")**: This line prints a message indicating the received timestamp and the address of the slave node from which it was received. It uses f-strings to format the message.
* **Error Handling**:
  + **print(f"Error handling {slave\_address}: {e}")**: If any error occurs during the execution of the try block (e.g., socket communication error), this line prints an error message indicating the address of the slave node and the specific error that occurred.

Example: Suppose the function is called with the following arguments:

handle\_slave(slave\_socket, ('192.168.1.2', 5000), timestamps)

* If the slave node sends the timestamp **1622145800.123456**, the function will receive it, decode it, convert it to a float, append it to the **timestamps** list, and print a message like:

Received timestamp 1622145800.123456 from ('192.168.1.2', 5000)

* If an error occurs during this process (e.g., the connection is closed unexpectedly), the function will catch the exception and print an error message indicating the address of the slave node and the specific error.

def synchronize\_clocks(slave\_sockets): try: timestamps = [] threads = [] for slave\_socket, slave\_address in slave\_sockets: thread = threading.Thread(target=handle\_slave, args=(slave\_socket, slave\_address, timestamps)) threads.append(thread) thread.start() for thread in threads: thread.join() average\_timestamp = sum(timestamps) / len(timestamps) for slave\_socket, slave\_address in slave\_sockets: slave\_socket.send(str(average\_timestamp).encode()) print(f"Sent average timestamp {average\_timestamp} to {slave\_address}") except Exception as e: print(f"Error during synchronization: {e}")

This function **synchronize\_clocks** coordinates the clock synchronization process between the master node and slave nodes. Here's a detailed explanation with an example:

* **Function Definition**:
  + **def synchronize\_clocks(slave\_sockets):**: This function takes one parameter:
    - **slave\_sockets**: A list of tuples, where each tuple contains a socket object representing the connection to a slave node and the address of that slave node.
* **Try-Except Block**:
  + **try:**: The code inside this block tries to execute the following operations.
  + **except Exception as e:**: If any exception occurs during the execution of the try block, it is caught, and the code inside this block is executed. The exception is stored in the variable **e**.
* **Collecting Timestamps from Slave Nodes**:
  + The function initializes an empty list **timestamps** to store timestamps received from slave nodes.
  + It also initializes an empty list **threads** to store thread objects for each slave node.
* **Starting Threads for Each Slave Node**:
  + The function iterates over each **(slave\_socket, slave\_address)** tuple in the **slave\_sockets** list.
  + For each tuple, it creates a new thread using the **threading.Thread** class, with the **handle\_slave** function as the target and passing the **slave\_socket**, **slave\_address**, and **timestamps** as arguments.
  + The thread object is appended to the **threads** list, and then the thread is started using the **start** method.
* **Waiting for Threads to Complete**:
  + After starting all threads, the function iterates over each thread in the **threads** list and calls the **join** method on each thread. This ensures that the function waits for all threads to complete before proceeding.
* **Calculating Average Timestamp**:
  + Once all threads have completed and timestamps have been collected from all slave nodes, the function calculates the average timestamp by summing all timestamps in the **timestamps** list and dividing by the number of timestamps.
* **Broadcasting Average Timestamp to Slave Nodes**:
  + The function then iterates over each **(slave\_socket, slave\_address)** tuple in the **slave\_sockets** list.
  + For each tuple, it sends the calculated average timestamp to the corresponding slave node by encoding it as a string and sending it over the socket connection.
  + It also prints a message indicating the average timestamp sent to each slave node.
* **Error Handling**:
  + If any error occurs during the synchronization process (e.g., socket communication error or calculation error), the exception is caught, and an error message is printed.

Example: Suppose **slave\_sockets** contains two tuples representing connections to two slave nodes:

slave\_sockets = [(socket1, ('192.168.1.2', 5000)), (socket2, ('192.168.1.3', 5000))]

Assume the timestamps received from the two slave nodes are **[1622145800.123456, 1622145800.234567]**. The function will calculate the average timestamp **(1622145800.123456 + 1622145800.234567) / 2 = 1622145800.1780115** and send this average timestamp back to each slave node. Additionally, it will print messages like:

Sent average timestamp 1622145800.1780115 to ('192.168.1.2', 5000) Sent average timestamp 1622145800.1780115 to ('192.168.1.3', 5000)

This demonstrates how the function coordinates the clock synchronization process between the master node and slave nodes.

def start\_master\_server(port): master\_server = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) master\_server.bind(('127.0.0.1', port)) master\_server.listen(5) slave\_sockets = [] print("Master node is running and waiting for slaves to connect.") while True: slave\_socket, slave\_address = master\_server.accept() slave\_sockets.append((slave\_socket, slave\_address)) print(f"Slave connected: {slave\_address}") synchronize\_clocks(slave\_sockets)

This function **start\_master\_server** initializes the master server, binds it to a specified port, listens for incoming connections from slave nodes, and coordinates the clock synchronization process. Here's a detailed explanation with an example:

* **Function Definition**:
  + **def start\_master\_server(port):**: This function takes one parameter:
    - **port**: An integer representing the port number on which the master server will listen for connections.
* **Initializing Master Server**:
  + **master\_server = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)**: This line creates a new socket object using IPv4 and TCP/IP protocol.
  + **master\_server.bind(('127.0.0.1', port))**: This line binds the socket to the localhost (**127.0.0.1**) and the specified port.
  + **master\_server.listen(5)**: This line starts listening for incoming connections, with a maximum backlog of 5 pending connections.
* **Accepting Connections from Slave Nodes**:
  + The function initializes an empty list **slave\_sockets** to store tuples of slave sockets and addresses.
  + **print("Master node is running and waiting for slaves to connect.")**: This line prints a message indicating that the master node is up and running, waiting for slave nodes to connect.
  + **while True:**: This initiates an infinite loop to continuously accept connections from slave nodes.
* **Accepting a Slave Connection**:
  + **slave\_socket, slave\_address = master\_server.accept()**: This line blocks until a connection is received from a slave node. Upon connection, it returns a new socket object representing the connection (**slave\_socket**) and the address of the slave node (**slave\_address**).
  + **slave\_sockets.append((slave\_socket, slave\_address))**: This line appends a tuple containing the slave socket and address to the **slave\_sockets** list.
  + **print(f"Slave connected: {slave\_address}")**: This line prints a message indicating that a slave node has connected, along with its address.
* **Synchronizing Clocks with Connected Slaves**:
  + After a slave node connects, the function calls the **synchronize\_clocks** function with the list of slave sockets (**slave\_sockets**) as an argument. This initiates the clock synchronization process with the newly connected slave node(s).

Example: Suppose the master server is started on port 8080. After running the **start\_master\_server(8080)** function, the server will continuously listen for incoming connections from slave nodes. If a slave node with IP address **'192.168.1.2'** and port **5000** connects to the master server, the server will print a message like:

Master node is running and waiting for slaves to connect. Slave connected: ('192.168.1.2', 5000)

The function will then proceed to synchronize clocks with the newly connected slave node. This process repeats indefinitely, allowing multiple slave nodes to connect and synchronize their clocks with the master node.

def synchronize\_clock(master\_address): try: slave\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) slave\_socket.connect(master\_address) while True: local\_timestamp = time.time() slave\_socket.send(str(local\_timestamp).encode()) synchronized\_time = float(slave\_socket.recv(1024).decode()) print(f"Synchronized time received: {synchronized\_time}") time.sleep(5) except Exception as e: print(f"Error during synchronization: {e}")

This function **synchronize\_clock** represents the behavior of a slave node in the clock synchronization process. Here's a detailed explanation with an example:

* **Function Definition**:
  + **def synchronize\_clock(master\_address):**: This function takes one parameter:
    - **master\_address**: A tuple representing the address (IP address and port) of the master node to which the slave will connect.
* **Try-Except Block**:
  + **try:**: The code inside this block tries to execute the following operations.
  + **except Exception as e:**: If any exception occurs during the execution of the try block, it is caught, and the code inside this block is executed. The exception is stored in the variable **e**.
* **Initializing Socket and Connecting to Master**:
  + **slave\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)**: This line creates a new socket object for the slave node using IPv4 and TCP/IP protocol.
  + **slave\_socket.connect(master\_address)**: This line establishes a connection to the master node using the provided **master\_address**.
* **Sending Local Timestamp to Master**:
  + Inside a loop that runs indefinitely (**while True:**):
    - **local\_timestamp = time.time()**: This line retrieves the current local timestamp of the slave node using the **time.time()** function.
    - **slave\_socket.send(str(local\_timestamp).encode())**: This line encodes the local timestamp as a string, sends it to the master node over the socket connection after encoding it to bytes.
* **Receiving Synchronized Time from Master**:
  + **synchronized\_time = float(slave\_socket.recv(1024).decode())**: This line receives data (presumably the synchronized time) from the master node using the **recv** method of the socket object. The received data is then decoded from bytes to a string using the **decode** method and converted to a float using the **float** function. The synchronized time is stored in the variable **synchronized\_time**.
* **Printing Synchronized Time**:
  + **print(f"Synchronized time received: {synchronized\_time}")**: This line prints a message indicating the synchronized time received from the master node.
* **Sleeping for 5 Seconds**:
  + **time.sleep(5)**: This line pauses the execution of the slave node for 5 seconds before repeating the synchronization process. This delay helps in controlling the rate at which the slave node sends timestamps to the master node.
* **Error Handling**:
  + If any error occurs during the synchronization process (e.g., socket communication error or conversion error), the exception is caught, and an error message is printed.

Example: Suppose the master node is running on the localhost (**127.0.0.1**) on port **8080**. The **synchronize\_clock** function is called with the master address **('127.0.0.1', 8080)**. The slave node continuously sends its local timestamp to the master node, receives the synchronized time from the master, and prints it. This process repeats every 5 seconds until an error occurs or the program is terminated.

Clock synchronization refers to the process of ensuring that the clocks across multiple devices or nodes in a network are aligned or adjusted to maintain a consistent and accurate time reference. In a distributed system where multiple devices communicate and coordinate with each other, having synchronized clocks is crucial for various tasks such as data consistency, coordination of distributed processes, and logging events accurately.

**socket.AF\_INET** and **socket.SOCK\_STREAM** are constants representing parameters for creating a socket object in Python using the **socket** module.

* **socket.AF\_INET**: This constant represents the address family for IPv4. It specifies that the socket will use the IPv4 protocol for communication. In networking, an address family defines the format of addresses that a socket can communicate with. In this case, **AF\_INET** indicates that the socket will use IPv4 addresses.
* **socket.SOCK\_STREAM**: This constant represents the type of socket, specifically a stream socket. It indicates that the socket will provide a reliable, sequenced, and two-way connection-based byte stream. Stream sockets are commonly used for TCP (Transmission Control Protocol) connections, which ensure reliable and ordered delivery of data. With a stream socket, data is transmitted as a continuous stream of bytes, preserving the order of transmission and ensuring that all data is delivered without loss or duplication.