**LAB 9**

Aim: To implement Stateful and Stateless File Server.

**Lab Outcome:**

Describe the concepts of distributed File Systems with some case studies.

**Theory:**

Stateful and Stateless are two types of servers used in computer networks.

A stateful server is one that maintains the state of the client/server relationship across

multiple requests. It keeps track of information about the client's previous interactions

with the server, such as the client's session data, authentication credentials, and other

contextspecific information.

**For example :**

a stateful server is a web server that maintains user session information across multiple

requests. When a user logs in to a web application, the server stores the user's session

data (such as their login credentials) and uses that information to authenticate the user

on subsequent requests.

A stateless server does not keep track of any information about the client's previous

interactions. Each request is treated as an independent event, and the server does not

maintain any session data or other context-specific information.

**For example :**

a stateless server is a Domain Name System (DNS) server that provides IP addresses for

domain names. Each request to the DNS server is independent of any previous requests,

and the server does not maintain any session data or other context-specific information.

Overall, the choice between stateful and stateless servers depends on the specific

requirements of the application and the trade-offs between scalability, reliability, and

ease of implementation.

Here's a diagram to illustrate the difference between the two:

STATEFUL SERVER STATELESS SERVER

+-------------------+ +-------------------+

| Application | | Application |

+-------------------+ +-------------------+

| Session Data | | N/A |

+-------------------+ +-------------------+

| Authentication| | Authentication|

+-------------------+ +-------------------+

| Context Info | | N/A |

+-------------------+ +-------------------+

**Stateful System**

**Server.py**

import socket

import threading

import os

class FileSystem:

def \_\_init\_\_(self, root\_path):

self.root\_path = root\_path

def create\_file(self, path):

full\_path = os.path.join(self.root\_path, path)

with open(full\_path, 'w') as f:

f.write('')

return "File created successfully"

def read\_file(self, path):

full\_path = os.path.join(self.root\_path, path)

with open(full\_path, 'r') as f:

content = f.read()

return content

def write\_file(self, path, content):

full\_path = os.path.join(self.root\_path, path)

with open(full\_path, 'w') as f:

f.write(content)

return "File written successfully"

def delete\_file(self, path):

full\_path = os.path.join(self.root\_path, path)

os.remove(full\_path)

return "File deleted successfully"

class StatefulFileServer:

def \_\_init\_\_(self, host, port, file\_system):

self.host = host

self.port = port

self.file\_system = file\_system

self.sessions = {}

def run(self):

with socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) as sock:

sock.bind((self.host, self.port))

sock.listen()

while True:

conn, addr = sock.accept()

client\_thread = threading.Thread(target=self.handle\_client, args=(conn, addr))

client\_thread.start()

def handle\_client(self, conn, addr):

addr\_1 = input("Enter portNumber: ")

client\_id = addr[0] + ":" + addr\_1

print("Client: %s" % client\_id)

session\_data = self.sessions.get(client\_id, {})

print(f"Stored data for client {client\_id}: {session\_data}")

# Parse client request and perform file system operation

request = conn.recv(1024).decode()

response = self.handle\_request(request, session\_data)

# Send response to client

conn.sendall(response.encode())

# Update session data

self.sessions[client\_id] = session\_data

conn.close()

def handle\_request(self, request, session\_data):

tokens = request.split()

command = tokens[0]

session\_data[command] = tokens[1]

if command == "CREATE":

path = tokens[1]

return self.file\_system.create\_file(path)

elif command == "READ":

path = tokens[1]

return self.file\_system.read\_file(path)

elif command == "WRITE":

path = tokens[1]

content = ' '.join(tokens[2:])

return self.file\_system.write\_file(path, content)

elif command == "DELETE":

path = tokens[1]

return self.file\_system.delete\_file(path)

else:

return "Unknown command"

if \_\_name\_\_ == "\_\_main\_\_":

file\_system = FileSystem('./filesystem')

server = StatefulFileServer('localhost', 12345, file\_system)

server.run()

**Client.py**

import socket

class StatefulFileClient:

def \_\_init\_\_(self, host, port):

self.host = host

self.port = port

def run(self):

with socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) as sock:

sock.connect((self.host, self.port))

while True:

# Get user input

user\_input = input("> ")

# Send user input to server

sock.sendall(user\_input.encode())

# Receive and print response from server

response = sock.recv(1024).decode()

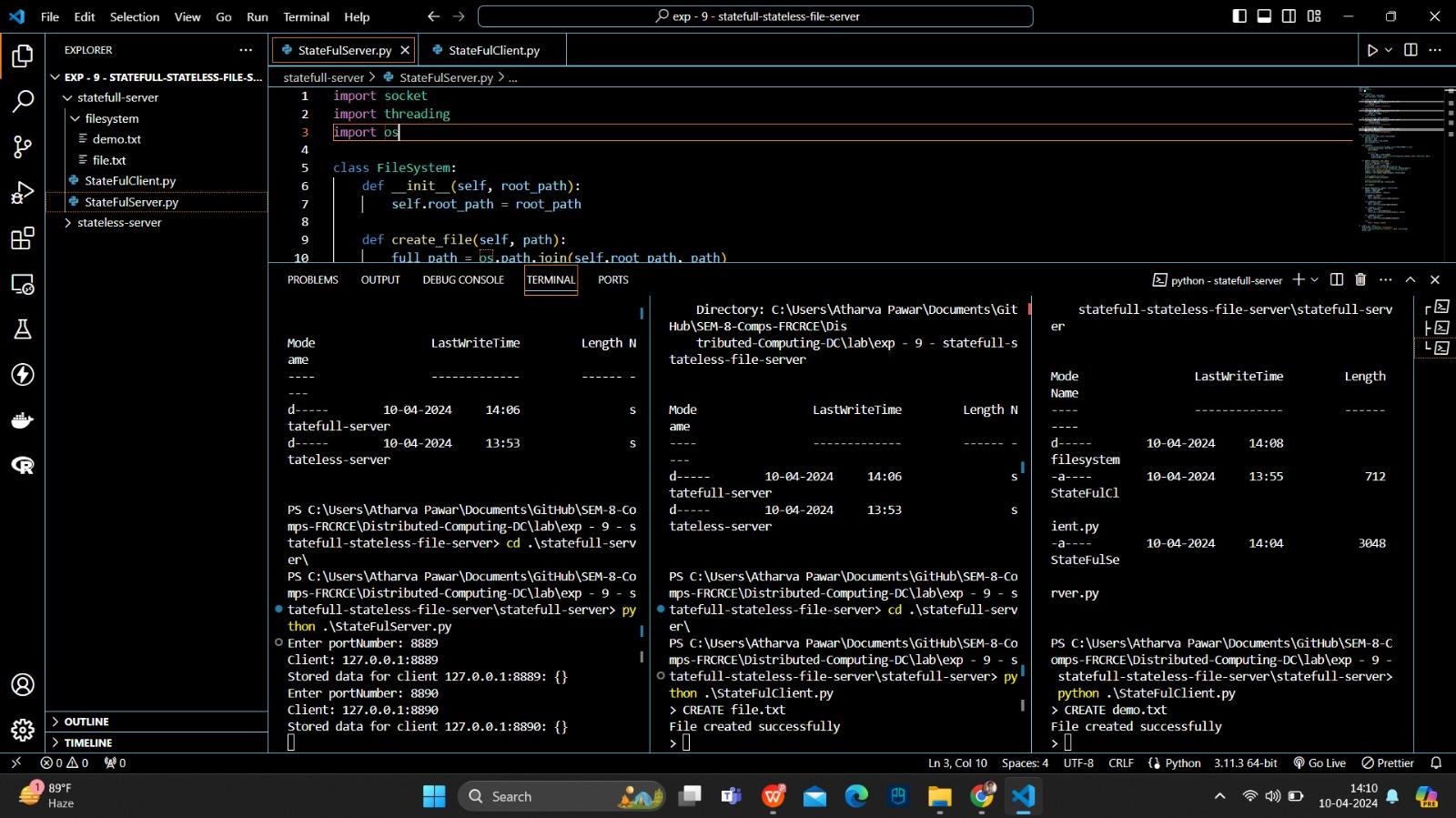
print(response)

if \_\_name\_\_ == "\_\_main\_\_":

client = StatefulFileClient('localhost', 12345)

client.run()

**Output :**



**Stateless System**

**Server.py**

import socket

import threading

import os

class FileSystem:

def \_\_init\_\_(self, root\_path):

self.root\_path = root\_path

def create\_file(self, path):

full\_path = os.path.join(self.root\_path, path)

with open(full\_path, 'w') as f:

f.write('')

return "File created successfully"

def read\_file(self, path):

full\_path = os.path.join(self.root\_path, path)

with open(full\_path, 'r') as f:

content = f.read()

return content

def write\_file(self, path, content):

full\_path = os.path.join(self.root\_path, path)

with open(full\_path, 'w') as f:

f.write(content)

return "File written successfully"

def delete\_file(self, path):

full\_path = os.path.join(self.root\_path, path)

os.remove(full\_path)

return "File deleted successfully"

class StatefulFileServer:

def \_\_init\_\_(self, host, port, file\_system):

self.host = host

self.port = port

self.file\_system = file\_system

self.sessions = {}

def run(self):

with socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) as sock:

sock.bind((self.host, self.port))

sock.listen()

while True:

conn, addr = sock.accept()

client\_thread = threading.Thread(target=self.handle\_client, args=(conn, addr))

client\_thread.start()

def handle\_client(self, conn, addr):

# addr\_1 = input("Enter portNumber: ")

client\_id = addr[0] + ":" + str(addr[1])

print("Client: %s" % client\_id)

# Parse client request and perform file system operation

request = conn.recv(1024).decode()

response = self.handle\_request(request)

# Send response to client

conn.sendall(response.encode())

# Update session data

conn.close()

def handle\_request(self, request):

tokens = request.split()

command = tokens[0]

# session\_data[command] = tokens[1]

if command == "CREATE":

path = tokens[1]

return self.file\_system.create\_file(path)

elif command == "READ":

path = tokens[1]

return self.file\_system.read\_file(path)

elif command == "WRITE":

path = tokens[1]

content = ' '.join(tokens[2:])

return self.file\_system.write\_file(path, content)

elif command == "DELETE":

path = tokens[1]

return self.file\_system.delete\_file(path)

else:

return "Unknown command"

if \_\_name\_\_ == "\_\_main\_\_":

file\_system = FileSystem('./filesystem')

server = StatefulFileServer('localhost', 12345, file\_system)

server.run()

**Client.py**

import socket

class StatefulFileClient:

def \_\_init\_\_(self, host, port):

self.host = host

self.port = port

def run(self):

with socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) as sock:

sock.connect((self.host, self.port))

while True:

# Get user input

user\_input = input("> ")

# Send user input to server

sock.sendall(user\_input.encode())

# Receive and print response from server

response = sock.recv(1024).decode()

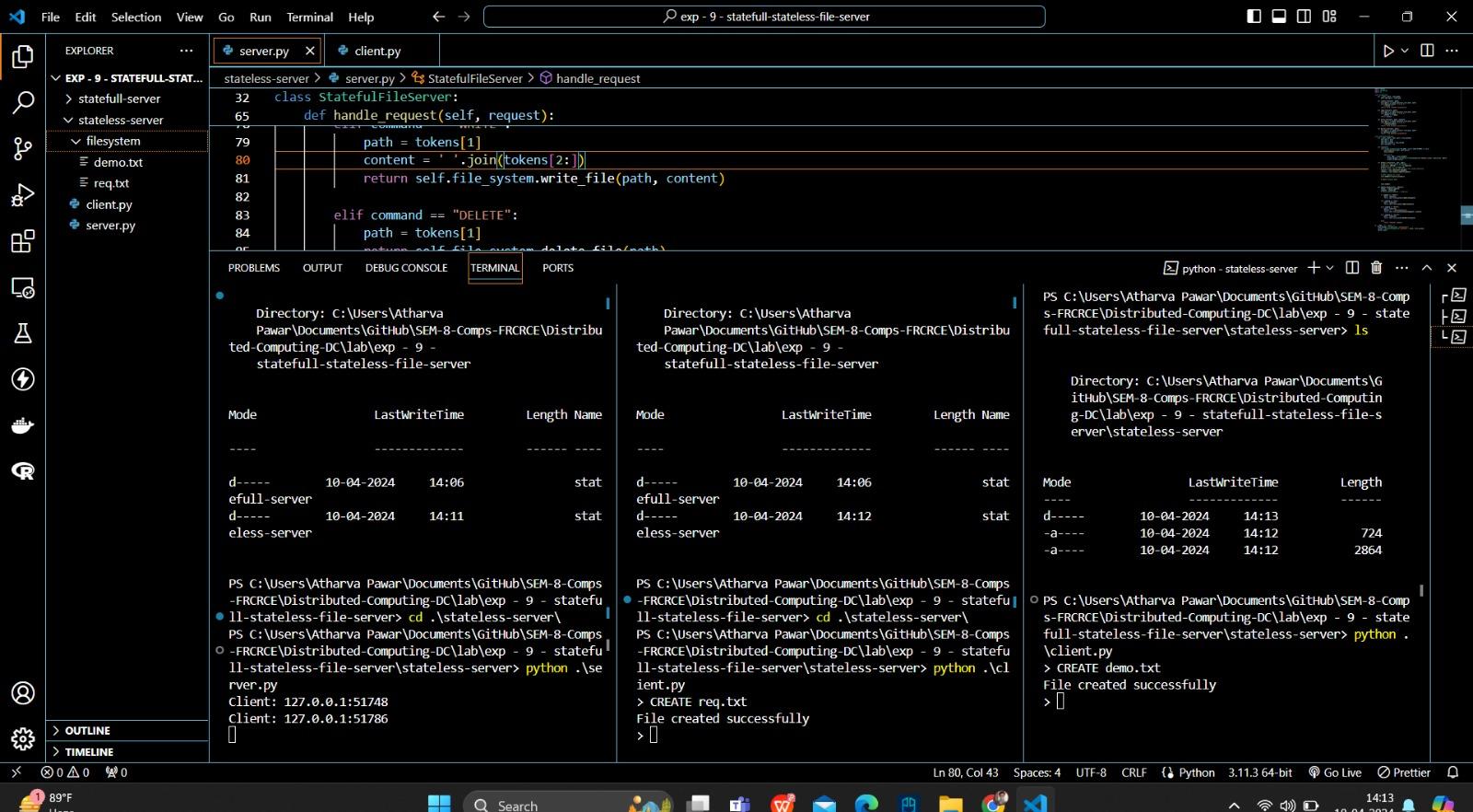
print(response)

if \_\_name\_\_ == "\_\_main\_\_":

client = StatefulFileClient('localhost', 12345)

client.run()

**Output :**



Conclusions :

1. Implemented Stateful and Stateless Server

2. Compared the performance of stateful and stateless servers in handling client

requests.

3. The response times of the stateless server were consistently faster and more

stable compared to the stateful server.

4. Since the stateful server needs to keep track of client sessions, it requires more

resources and processing power, which can result in slower response times and

higher variability.

5. It is important to consider the requirements of the application and choose the

appropriate server architecture based on those requirements.

Postlab Questions:

1. Compare Stateful and stateless servers

1. Stateful Servers:

- Maintain Session State: Stateful servers maintain session state information for each client connection. This means that the server remembers the context or the state of the communication with each client throughout the duration of the session.

- Resource Intensive: Since stateful servers need to keep track of client states, they tend to be more resource-intensive compared to stateless servers. This is especially true for applications with a large number of concurrent connections.

- Scalability Challenges: Scaling stateful servers can be challenging because as the number of clients increases, the server needs to manage and keep track of more states, which can lead to bottlenecks and performance issues.

- Examples: Database servers, session-based web applications where user sessions are maintained on the server side, multiplayer online games where the server tracks the game state for each player.

2. Stateless Servers:

- No Session State: Stateless servers do not maintain any session state information. Each client request is processed independently, without any knowledge of previous interactions. This makes stateless servers simpler and more scalable.

- Scalability: Stateless servers are inherently more scalable because they do not need to manage session state. They can easily distribute requests across multiple servers without worrying about maintaining session affinity or synchronization between servers.

- Horizontal Scalability: Stateless servers can achieve horizontal scalability by adding more servers to handle increasing loads. Since each request is independent, they can be load-balanced across multiple servers.

- Examples: RESTful APIs, serverless architectures, content delivery networks (CDNs), where each request contains all the necessary information for processing and does not rely on server-side state.

2. Explain: ‘Exactly Once’ call semantics

"Exactly Once" call semantics refers to a guarantee provided by a system or protocol that ensures that an operation or function is executed exactly once, and only once, even in the presence of failures, retries, or other anomalies. This guarantee is crucial in distributed systems where operations may be retried due to network errors, crashes, or other failures, which could potentially lead to duplicate executions and incorrect results if not handled properly.

In the context of distributed systems and messaging protocols, achieving exactly once semantics can be challenging due to the following factors:

1. Network Failures: Network failures can cause messages to be lost, delayed, or duplicated. This can lead to situations where a message is processed more than once if not handled correctly.

2. Message Duplication: Even without network failures, messages can be duplicated due to retries, timeouts, or other issues in the communication layer.

3. Node Failures: Nodes in a distributed system may crash or become temporarily unavailable. If a node fails after processing a message but before acknowledging its completion, it may result in message reprocessing and potential duplicates when the node recovers.

To achieve exactly once semantics, distributed systems typically employ various techniques such as:

1. Idempotent Operations: Designing operations to be idempotent ensures that performing the operation multiple times has the same effect as performing it once. This allows systems to safely retry operations without causing unintended side effects.

2. Message Deduplication: Systems can use message deduplication techniques to identify and eliminate duplicate messages before processing them. This often involves assigning unique identifiers to messages and maintaining a record of processed messages to detect and filter out duplicates.

3. Transaction Management: Using transactional mechanisms to ensure that operations are atomic, consistent, isolated, and durable (ACID) can help achieve exactly once semantics. Transactions provide a way to bundle multiple operations into a single unit of work, ensuring that either all operations are completed successfully or none are.

4. Acknowledgement and Commitment: Employing acknowledgement mechanisms where the receiver acknowledges the successful processing of a message can help ensure that messages are processed exactly once. This requires careful coordination between the sender and receiver to handle failures and retries correctly.