**WEEK-9**

**Networking in Python**

**Description:**

This unit explores the concepts of **network programming using Python**, focusing on how to build client-server applications using the built-in socket module. Students will gain hands-on experience in creating both TCP and UDP-based communication systems.

Key topics include:

* Basics of sockets and IP communication
* Creating TCP and UDP clients and servers
* Sending and receiving messages over the network
* Handling multiple connections using threading
* Implementing real-time chat or file transfer functionality

This unit is essential for understanding how distributed systems communicate. It enables students to build real-world networked applications and lays the foundation for advanced concepts such as REST APIs and web socket communication.

**Write a Python program to demonstrate the use of threading and synchronization with a shared resource.**

**Description:**

In this experiment:

* Two threads (Thread-1 and Thread-2) try to increment a shared global counter.
* A Lock is used to synchronize access to the shared resource so that only one thread can update the counter at a time.
* The lock.acquire() and lock.release() (or with lock:) prevent race conditions, ensuring thread-safe execution.

This demonstrates how synchronization is critical when multiple threads access or modify shared data simultaneously.

**Program :**

import threading

import time

# Shared resource

counter = 0

# Lock object for synchronization

lock = threading.Lock()

def increment\_counter(thread\_name):

 global counter

 for \_ in range(5):

 time.sleep(1)

 lock.acquire()

 try:

 current = counter

 print(f"{thread\_name} read counter: {current}")

 counter = current + 1

 print(f"{thread\_name} incremented counter to: {counter}")

 finally:

 lock.release()

# Create threads

thread1 = threading.Thread(target=increment\_counter, args=("Thread-1",))

thread2 = threading.Thread(target=increment\_counter, args=("Thread-2",))

# Start threads

thread1.start()

thread2.start()

# Wait for threads to complete

thread1.join()

thread2.join()

print("Final Counter Value:", counter)

**Output:**

****

**Write a Python program to create a simple multi-client chat server using sockets and threading.**

 **Description:**

This experiment introduces how to build a multi-client chat server in Python using socket programming and the threading module. Unlike basic single-client programs, this server can handle multiple clients at once — a critical feature in modern networking applications like chat rooms, collaborative tools, or multiplayer games.

The server listens for client connections and spawns a new thread for each client to handle communication simultaneously. When one client sends a message, the server relays it to all connected clients, creating a simple group chat effect.

This helps students learn:

* Concurrent handling of clients using threading
* Broadcasting messages to multiple sockets
* Managing client connections in real time

**Program: Server.py**

import socket

import threading

def handle\_client(conn, addr):

 print(f"New connection from {addr}")

 conn.send("Welcome to the chat! Type 'bye' to exit.".encode())

 while True:

 try:

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

 if not data:

 break

 print(f"{addr} says: {data}")

 if data.lower() in ['bye', 'exit']:

 conn.send("Goodbye! Connection closed.".encode())

 break

 else:

 conn.send(f"Server received: {data}".encode())

 except:

 break

 print(f"Connection closed with {addr}")

 conn.close()

server = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

server.setsockopt(socket.SOL\_SOCKET, socket.SO\_REUSEADDR, 1)

server.bind(('localhost', 12347))

server.listen()

print("Multithreaded Server is running on port 12347...")

while True:

 conn, addr = server.accept()

 thread = threading.Thread(target=handle\_client, args=(conn, addr))

 thread.start()

**Output:**

****

**Program: client.py**

import socket

client = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

client.connect(('localhost', 12345))

welcome = client.recv(1024).decode()

print("Server:", welcome)

while True:

 msg = input("You: ")

 client.send(msg.encode())

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

 print("Server:", response)

 if msg.lower() in ['bye', 'exit']:

 break

client.close()

**Output:**

****

**Viva Questions – Unit 4: Networking in Python**

1. **What is a thread in Python and how is it different from a process?**A thread is a lightweight unit of execution within a process. Threads share the same memory space, while processes run in separate memory spaces. Threads are useful for I/O-bound tasks; processes are better for CPU-bound tasks.
2. **What is the purpose of the Lock object in the threading module?**A Lock is used to synchronize access to shared resources in multithreaded programs. It prevents race conditions by ensuring only one thread can access a critical section at a time.
3. **What is the use of the join() method in threading?**The join() method makes the main program wait until the thread has finished its execution. It ensures that threads complete before the program exits.
4. **Can you explain what a race condition is in multithreading?**A race condition occurs when multiple threads access and modify shared data simultaneously, leading to unpredictable results. It happens when threads interfere with each other due to lack of synchronization.
5. **How do you create a thread using the Thread class in Python?**You can create a thread using:

import threading

def task():

 print("Running...")

t = threading.Thread(target=task)

t.start()