Index

Experiment No: 01

Experiment Name : Implementation and Establishment of Socket Client-Server Communication in Python

Objective

To implement and understand basic client-server communication using Python socket programming, demonstrating how data is exchanged between two connected systems over a network.

Theory

A socket is an endpoint for sending or receiving data across a computer network. Python's socket module provides a low-level interface to network communications using the TCP/IP protocol.

Types of Sockets:

1. TCP (Stream Sockets): Reliable, connection-oriented communication.

2. UDP (Datagram Sockets): Unreliable, connectionless communication.

In this experiment, we use TCP sockets for establishing a reliable connection between a client and a server.

Tools and Environment

Programming Language: Python 3.x
IDE/Editor: VS Code / PyCharm / IDLE
OS: Windows / Linux
Modules Used: socket

Implementation(Code)

a) Server Program (server.py):

```
import socket

server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

host = socket.gethostname()

port = 12345

server_socket.bind((host, port))

server_socket.listen(1)

print("Server is listening...")

conn, addr = server_socket.accept()

print("Connection from:", addr)
```

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

print("Client says:", data)

conn.send("Hello Client, this is Server!".encode())

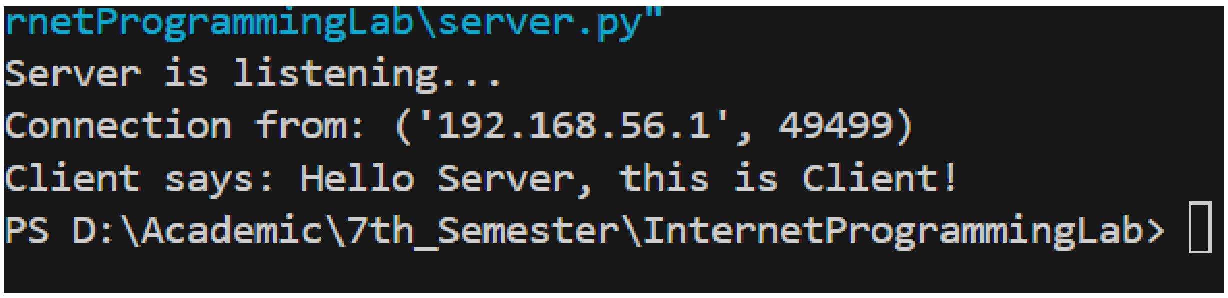
conn.close()

server_socket.close()

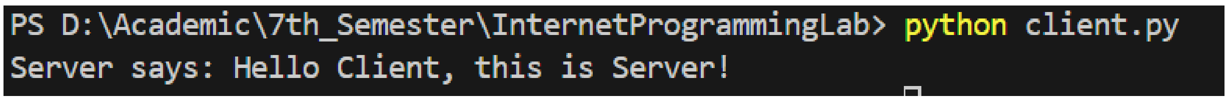
b) Client Program (client.py):

```
import socket

client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

host = socket.gethostname()

port = 12345

client_socket.connect((host, port))

client_socket.send("Hello Server, this is Client!".encode())

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

print("Server says:", response)

client_socket.close()
```

Output:



```
rnetProgrammingLab\server.py"
Server is listening...
Connection from: ('192.168.56.1', 49499)
Client says: Hello Server, this is Client!
PS D:\Academic\7th_Semester\InternetProgrammingLab>
```



```
PS D:\Academic\7th_Semester\InternetProgrammingLab> python client.py
Server says: Hello Client, this is Server!
```

Result : Successfully established a two-way communication between a client and a server using Python's socket module.

Learning Outcomes :

Understood the basics of socket programming.
Learned how to use bind(), listen(), accept(), connect(), send(), and recv() functions.
Established and tested TCP-based communication between two Python scripts.

Conclusion : This lab demonstrated how socket programming enables reliable communication between two devices in a network. By implementing client and server codes, the concept of connection establishment and message exchange was successfully illustrated.

Experiment No: 02

Experiment Name : Implementation and Establishment of Echo Socket Client-Server Communication in Python .

Objective

To design and implement an Echo Client-Server model in Python where the server echoes back the same message it receives from the client, demonstrating two-way TCP communication.

Theory

An Echo Server is a network application that receives data from a client and sends the exact same data back to the client. It helps understand how data transmission and reception work over a TCP socket

Types of Sockets:

1. TCP (Stream Sockets): Reliable, connection-oriented communication.

2. UDP (Datagram Sockets): Unreliable, connectionless communication.

In this experiment, we use TCP sockets for establishing a reliable connection between a client and a server.

Tools and Environment

Programming Language: Python 3.x
IDE/Editor: VS Code / PyCharm / IDLE
OS: Windows / Linux
Modules Used: socket

Implementation(Code)

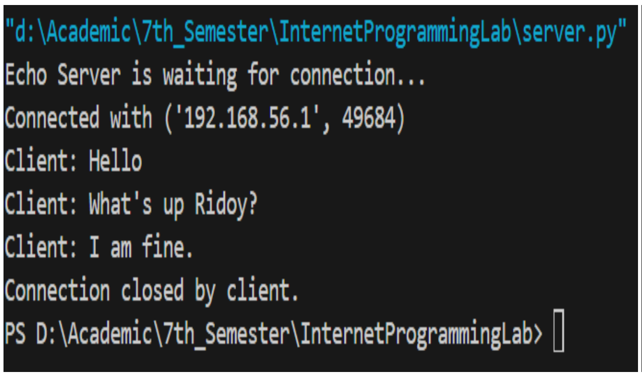
a) Server Program (server.py):

```
import socket

server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

host = socket.gethostname()

port = 12346

server_socket.bind((host, port))

server_socket.listen(1)

print("Echo Server is waiting for connection...")

client_socket, addr = server_socket.accept()

print(f"Connected with {addr}")

while True:

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

    if not data or data.lower() == 'exit':

        print("Connection closed by client.")
        break
    print("Client:", data)
    client_socket.send(data.encode())
client_socket.close()
server_socket.close()
```

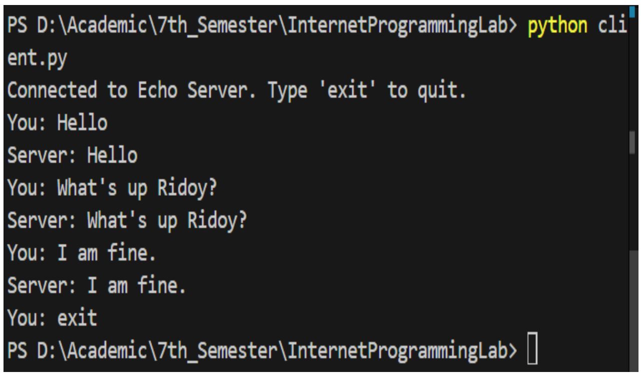
b) Client Program (client.py):

```
import socket
client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
host = socket.gethostname()
port = 12346
client_socket.connect((host, port))
print("Connected to Echo Server. Type 'exit' to quit.")
while True:
    message = input("You: ")
    client_socket.send(message.encode())
    if message.lower() == 'exit':
        break
    response = client_socket.recv(1024).decode()
    print("Server:", response)
client_socket.close()
```

Output:



```
"d:\Academic\7th_Semester\InternetProgrammingLab\server.py"
Echo Server is waiting for connection...
Connected with ('192.168.56.1', 49684)
Client: Hello
Client: What's up Ridoy?
Client: I am fine.
Connection closed by client.
PS D:\Academic\7th_Semester\InternetProgrammingLab>
```



```
PS D:\Academic\7th_Semester\InternetProgrammingLab> python cli
ent.py
Connected to Echo Server. Type 'exit' to quit.
You: Hello
Server: Hello
You: What's up Ridoy?
Server: What's up Ridoy?
You: I am fine.
Server: I am fine.
You: exit
PS D:\Academic\7th_Semester\InternetProgrammingLab>
```

Result : Successfully implemented an Echo Socket Client-Server model in Python. The server correctly echoed back all messages sent from the client until the client terminated the connection.

Learning Outcomes : Understood how to maintain persistent TCP connections. Learned how to send and receive data using Python sockets. Implemented bidirectional communication between client and server.

Conclusion : This lab demonstrated how socket programming can be extended to implement a real-time echo system. The client sends messages to the server, and the server responds immediately with the same message — confirming correct data exchange over a network.

Experiment No: 03

Experiment Name: Implementation and Establishment of Socket Server and Multi-Threaded Client Communication in Python

Objective

To implement and understand socket-based client-server communication using Python where a server handles multiple clients simultaneously through multi-threading, demonstrating concurrent communication over a network.

Theory

In basic socket programming, a server can handle only one client at a time. However, real-world network applications require the server to communicate with multiple clients concurrently. This is achieved using **multi-threading**, where each client connection is handled in a separate thread.

Python supports multi-threading using the `threading` module. By combining sockets with threads, a server can efficiently manage multiple client connections at the same time.

Key Concepts:

- **Multi-threading:** Running multiple threads concurrently within a process.
- **Thread per Client Model:** Each client connection is handled by a separate thread.
- **TCP Sockets:** Reliable, connection-oriented communication.
- **Concurrency:** Ability to process multiple client requests simultaneously.

Tools and Environment

- **Programming Language:** Python 3.x
- **IDE/Editor:** VS Code / PyCharm / IDLE
- **Operating System:** Windows / Linux
- **Modules Used:** `socket`, `threading`

Implementation (Code)

a)Server(lab3_server.py):

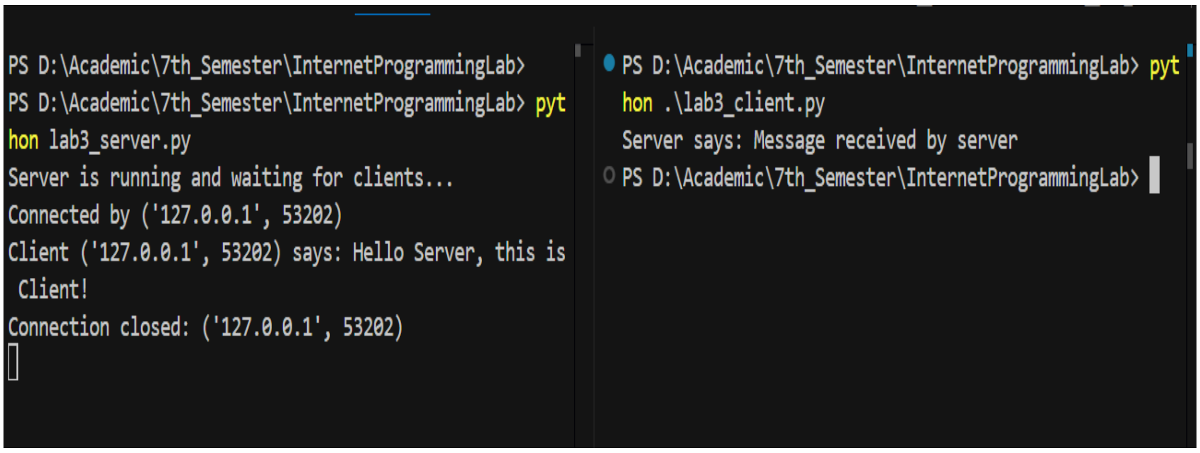
```
import socket
import threading
```

```
def client_handler(connection, address):
    print("Client connected from:", address)

    message = connection.recv(1024).decode()
    print("Message from client:", message)

    reply = "Message received by server"
    connection.send(reply.encode())

    connection.close()
    print("Client disconnected:", address)

def start_server():
    host = "127.0.0.1"
    port = 1019

    server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server.bind((host, port))
    server.listen(5)

    print("Server is running and waiting for clients...")

    while True:
        conn, addr = server.accept()
        thread = threading.Thread(
            target=client_handler,
            args=(conn, addr)
        )
        thread.start()


if __name__ == "__main__":
    start_server()
```

b)Client(lab3_client.py)

```
import socket

def start_client():
    host = "127.0.0.1"
    port = 1019

    client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    client.connect((host, port))

    message = "Hello Server, this is Client!"
    client.send(message.encode())

    response = client.recv(1024).decode()
    print("Server says:", response)

    client.close()

if __name__ == "__main__":
    start_client()
```

Output:



```
PS D:\Academic\7th_Semester\InternetProgrammingLab>
PS D:\Academic\7th_Semester\InternetProgrammingLab> python lab3_server.py
Server is running and waiting for clients...
Connected by ('127.0.0.1', 53202)
Client ('127.0.0.1', 53202) says: Hello Server, this is Client!
Connection closed: ('127.0.0.1', 53202)
```

```
PS D:\Academic\7th_Semester\InternetProgrammingLab> python .\lab3_client.py
Server says: Message received by server
PS D:\Academic\7th_Semester\InternetProgrammingLab>
```

Result:

Successfully established socket communication between a multi-threaded server and multiple clients using Python, enabling concurrent message exchange.

Learning Outcomes:

- Understood the concept of multi-threaded server design.
- Learned how to use Python's `threading` module.
- Implemented concurrent client-server communication.
- Gained experience in handling multiple network connections.

Conclusion :

This experiment demonstrated the use of multi-threading in socket programming to handle multiple client connections concurrently. By assigning a separate thread to each client, the server efficiently managed simultaneous communications, making the system scalable and suitable for real-world network applications.

Experiment No: 04

Experiment Name: Implementation and Establishment of an HTTP Server Responding with a Webpage in Python

Objective

To implement and understand a basic HTTP server using Python that responds with a simple webpage, demonstrating how web servers handle HTTP requests and send HTTP responses to clients (web browsers).

Theory

An HTTP server is a software application that handles requests from clients (such as web browsers) using the HyperText Transfer Protocol (HTTP). When a client sends an HTTP request, the server processes it and returns an HTTP response containing headers and content (usually HTML).

Python provides built-in support for creating simple HTTP servers using the http.server module. This module allows developers to quickly implement a basic web server capable of serving static HTML content over a network.

Key Concepts:

- HTTP Request: Sent by a client to request a resource.
- HTTP Response: Sent by the server containing status codes and webpage data.
- Port: A communication endpoint (commonly port 8000 for testing).
- Webpage (HTML): Content displayed in the client's browser.

Tools and Environment

- Programming Language: Python 3.x
- IDE/Editor: VS Code / PyCharm / IDLE
- Operating System: Windows / Linux
- Modules Used: http.server, socketserver

Implementation (Code)

a) HTTP Server Program (lab4_server.py):

```
from http.server import SimpleHTTPRequestHandler
```

```
import socketserver
PORT = 1019

Handler = SimpleHTTPRequestHandler

with socketserver.TCPServer(("", PORT), Handler) as httpd:
    print("HTTP Server is running at port", PORT)
    httpd.serve_forever()
```

b) HTML Webpage (lab4_index.html):

```
<!DOCTYPE html>
<html>
<head>
    <title>Python HTTP Server</title>
</head>
<body>
    <h1>Welcome to Python HTTP Server</h1>
    <p>Hello Mr Toufike-Ur-Rahman Ridoy</p>
    <h3>Whats up?</h3>
</body>
</html>
```

Execution Steps:

1. Run the server program using:

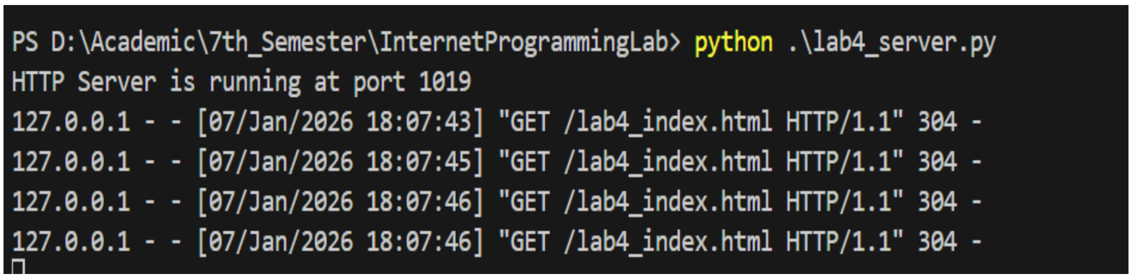
   ```
   python lab4_server.py
   ```

2. Open a web browser.

   Enter the URL:

   ```
   http://localhost:1019/lab4_index.html
   ```

Output:



```
PS D:\Academic\7th_Semester\InternetProgrammingLab> python .\lab4_server.py
HTTP Server is running at port 1019
127.0.0.1 - - [07/Jan/2026 18:07:43] "GET /lab4_index.html HTTP/1.1" 304 -
127.0.0.1 - - [07/Jan/2026 18:07:45] "GET /lab4_index.html HTTP/1.1" 304 -
127.0.0.1 - - [07/Jan/2026 18:07:46] "GET /lab4_index.html HTTP/1.1" 304 -
127.0.0.1 - - [07/Jan/2026 18:07:46] "GET /lab4_index.html HTTP/1.1" 304 -
```

Result:

Successfully implemented an HTTP server in Python that responds to client requests by serving a webpage using the built-in http.server module.

Learning Outcomes:

- Understood the fundamentals of HTTP protocol.
- Learned how a web server handles client requests.
- Gained experience using Python’s http.server module.
- Successfully served an HTML webpage using Python.

Conclusion :

This experiment demonstrated how Python can be used to implement a basic HTTP server capable of serving web content. By running the server and accessing it through a web browser, the concept of HTTP request-response communication was clearly illustrated. This experiment forms a foundation for understanding web server development and network-based applications.

Experiment No: 07

Experiment Name: Test html code and a web page with previous Lab-5 and Lab-6 HTTP server .

Objective

The objective of this lab is to test HTML code and render a web page using the HTTP servers created in **Lab 05** and **Lab 06**, demonstrating the process of serving HTML content over a network.

Theory

In previous labs, a basic HTTP server was created using Python's http.server module. In this experiment, instead of embedding HTML inside the Python script, an external HTML file is served.

This demonstrates:

- Separation of backend and frontend
- Serving static HTML files
- File handling in Python
- Real-world web server behavior

When the browser sends a GET request:

- The server reads the HTML file
- Sends HTTP response headers
- Sends HTML file content as response body

Materials Required

- Computer with Python 3 installed
- Text editor (e.g., VS Code, Sublime, or Notepad++)
- Web browser (Chrome, Firefox, Edge, etc.)
- HTML files to test

Application

Serving HTML files is the foundation of:

- Static website hosting
- Web development
- Backend frameworks
- Production web servers

Implementation Procedure

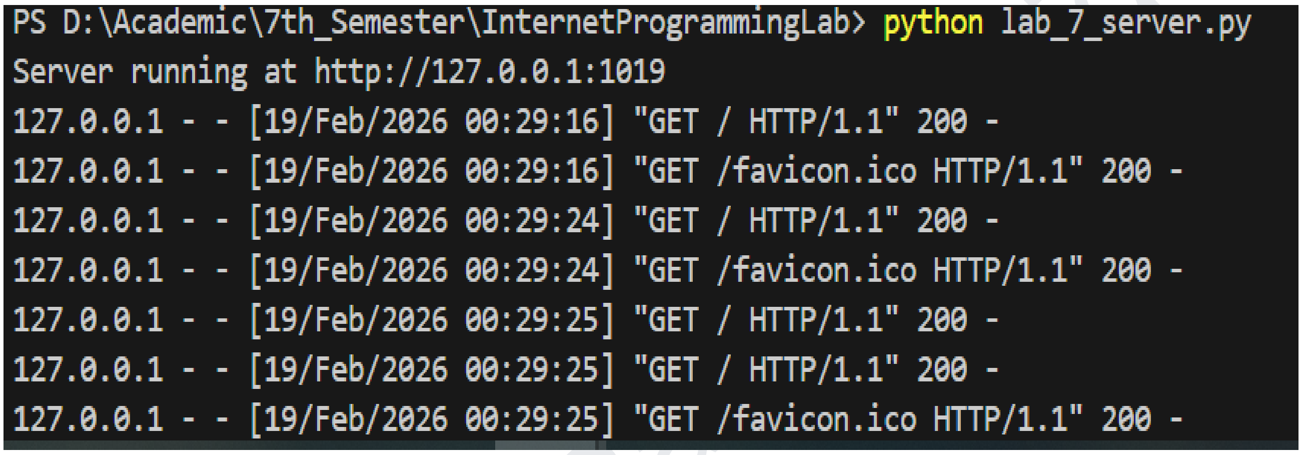
index.html

```
<!DOCTYPE html>
<html>
<head>
    <title>TUR Test Page</title>
</head>
<body>
    <h1>Welcome to My Test Page</h1>
    <p>Hello Mr.Toufike-Ur-Rahman Ridoy.</p>
</body>
</html>
```

Lab_7_server.py

```
from http.server import HTTPServer, BaseHTTPRequestHandler
HOST = "127.0.0.1"
PORT = 1019
class FileHandler(BaseHTTPRequestHandler):
    def do_GET(self):
        # Send response status code
        self.send_response(200)
        # Send headers
        self.send_header("Content-type", "text/html")
        self.end_headers()
        # Open and read the HTML file
        with open("index.html", "r") as file:
            content = file.read()

        # Write content to the client

        self.wfile.write(content.encode("utf-8"))

if __name__ == "__main__":

    print(f"Server running at http://{HOST}:{PORT}")

    server = HTTPServer((HOST, PORT), FileHandler)

    server.serve_forever()
```

Output:



```
PS D:\Academic\7th_Semester\InternetProgrammingLab> python lab_7_server.py
Server running at http://127.0.0.1:1019
127.0.0.1 - - [19/Feb/2026 00:29:16] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [19/Feb/2026 00:29:16] "GET /favicon.ico HTTP/1.1" 200 -
127.0.0.1 - - [19/Feb/2026 00:29:24] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [19/Feb/2026 00:29:24] "GET /favicon.ico HTTP/1.1" 200 -
127.0.0.1 - - [19/Feb/2026 00:29:25] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [19/Feb/2026 00:29:25] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [19/Feb/2026 00:29:25] "GET /favicon.ico HTTP/1.1" 200 -
```

Execution Steps

1. Create index.html in the same folder.

2. Save Python file.

3. Run: python server.py

4. Open browser and visit: http://127.0.0.1:1019

Results

The HTTP server successfully reads the external HTML file and displays it in the browser, confirming correct integration between HTML frontend and Python backend.

Conclusion

The HTML page was successfully tested and served using the previously implemented HTTP servers. This experiment strengthens understanding of web hosting fundamentals and file based content delivery.

Experiment No: 09

Experiment Name: Configuration of Mobile with New DNS Address.

Objective

The objective of this lab is to learn how to configure a mobile device with a new DNS (Domain Name System) address to improve internet connectivity, speed, or access region-specific services.

Theory

DNS (Domain Name System) is responsible for translating domain names (like www.google.com) into IP addresses that computers and mobile devices can understand. By changing the DNS server on a mobile device, users can:

- Improve browsing speed.
- Access geo-restricted content.
- Enhance privacy and security.
- Avoid ISP-level filtering or censorship.

Common public DNS addresses include:

- **Google DNS:** 8.8.8.8 and 8.8.4.4
- **Cloudflare DNS:** 1.1.1.1 and 1.0.0.1
- **OpenDNS:** 208.67.222.222 and 208.67.220.220

Materials Required

- Mobile device (Android or iOS)
- Active Wi-Fi or mobile internet connection
- Knowledge of network settings

Application

Custom DNS configuration is widely used for:

• Network troubleshooting

• Enhanced privacy

• Parental control filtering

• Faster browsing performance

Implementation Procedure

Method 1: Configure DNS in WiFi Settings

1. Open Mobile Settings

2. Go to WiFi settings

3. Select connected network

4. Choose Modify Network

5. Change IP Settings to Static

6. Enter DNS 1: 8.8.8.8

7. Enter DNS 2: 8.8.4.4

8. Save settings

Method 2: Private DNS (Android)

1. Go to Settings

2. Select Network & Internet

3. Choose Private DNS

4. Select Private DNS provider hostname

5. Enter: dns.google

6. Save configuration

Testing and Verification

After configuration:

• Open browser and visit a website
• Use DNS checker website
• Verify internet connectivity
• Confirm DNS server change

Results

- DNS configuration successfully changed on the mobile device.
- Internet speed and browsing reliability improved.
- New DNS addresses are now active for the Wi-Fi network.

Conclusion

In this lab, we successfully configured a mobile device with a new DNS address. This process helps in faster web browsing, increased security, and access to region-specific content. Understanding DNS and how to modify it is crucial for network optimization and troubleshooting.

Experiment No: 10

Experiment Name: Access the Site Using Local DNS Server (`csebatcheight1019.com`)

Objective

The objective of this lab is to simulate the behavior of a DNS server locally and access a website using a custom domain (`csebatcheight1019.com`). This demonstrates how DNS resolves domain names to IP addresses and how HTTP servers serve web pages in a local environment.

Theory

DNS (Domain Name System): Translates human-readable domain names (e.g., `csebatcheight1019.com`) into IP addresses (e.g., `127.0.0.2`) so computers can locate resources on a network.

Local DNS Server: Can be simulated using Python's `socket` module to map custom domains to IP addresses for testing.

HTTP Server: Serves web pages over HTTP, allowing the client browser to display HTML content.

Lab Concept:

1. DNS server resolves a domain name to a local IP.
2. DNS client queries the server for the IP.
3. HTTP server hosts the website accessible via the resolved IP or domain.

Materials Required

- Computer with Python 3 installed
- Administrator access to edit hosts file (optional for browser testing)
- Web browser (Chrome, Firefox, Edge, etc.)
- Python IDE or text editor (VS Code, Sublime, etc.)

Files Used

1. **`lab10_dns_server.py`** – Python UDP server mapping `csebatcheight1019.com` → `127.0.0.2`.
2. **`lab10_dns_client.py`** – Queries the DNS server for the IP of a domain.
3. **`lab10_http_server.py`** – Python HTTP server serving `index.html`.
4. **`lab10_index.html`** – HTML page displaying "Welcome to csebatcheight1019.com"

Application

Serving HTML files is the foundation of:

- Static website hosting
- Web development
- Backend frameworks
- Production web servers

Implementation Procedure

Lab10_index.py

```
<!DOCTYPE html>
<html>
<head>
<title>Local DNS Website</title>
</head>
<body>
<h1>Welcome to csebatcheight1019.com</h1>
<p>This site is accessed using a local DNS server.</p>
</body>
</html>
```

Lab10_dns_server.py

```
import socket
HOST = "127.0.0.2"
PORT = 8080 # safe lab port (reusable)
dns_records = {
"csebatcheight1019.com": "127.0.0.2"
}
server = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
#Make port reusable
server.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
server.bind((HOST, PORT))
print("Local DNS Server running...")
print(f"Listening on {HOST}:{PORT}")
while True:
data, addr = server.recvfrom(1024)
domain = data.decode().strip().lower()
print("DNS Query:", domain)
ip = dns_records.get(domain, "Domain not found")
server.sendto(ip.encode(), addr)
```

Lab10_dns_client.py

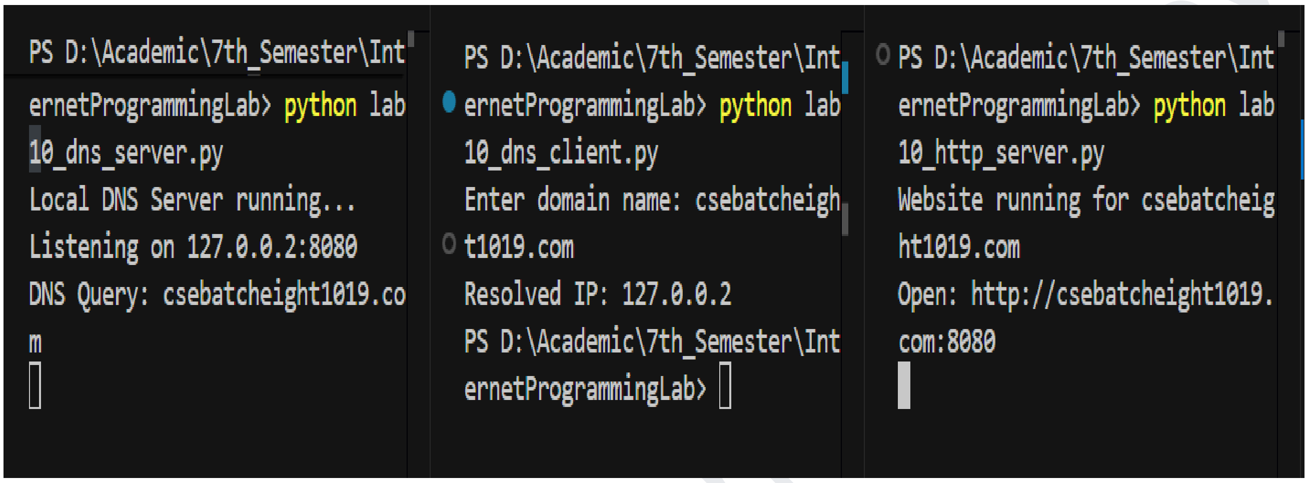
```
import socket
HOST = "127.0.0.2"
PORT = 8080
client = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
domain = input("Enter domain name: ")
client.sendto(domain.encode(), (HOST, PORT))
ip, _ = client.recvfrom(1024)
print("Resolved IP:", ip.decode())
client.close()
```

Lab10_http_server.py

```
from http.server import HTTPServer, SimpleHTTPRequestHandler
class ReusableHTTPServer(HTTPServer):
allow_reuse_address = True
```

```
HOST = "127.0.0.2"

PORT = 8080

server = ReusableHTTPServer((HOST, PORT), SimpleHTTPRequestHandler)

print("Website running for csebatcheight1019.com")

print(f"Open: http://csebatcheight1019.com:{PORT}")

server.serve_forever()
```

Output:



```
PS D:\Academic\7th_Semester\Int
ernetProgrammingLab> python lab
10_dns_server.py
Local DNS Server running...
Listening on 127.0.0.2:8080
DNS Query: csebatcheight1019.co
m
```

```
PS D:\Academic\7th_Semester\Int
ernetProgrammingLab> python lab
10_dns_client.py
Enter domain name: csebatcheigh
t1019.com
Resolved IP: 127.0.0.2
PS D:\Academic\7th_Semester\Int
ernetProgrammingLab>
```

```
PS D:\Academic\7th_Semester\Int
ernetProgrammingLab> python lab
10_http_server.py
Website running for csebatcheig
ht1019.com
Open: http://csebatcheight1019.
com:8080
```

Results

Local DNS server successfully resolved `csebatcheight1019.com` to `127.0.0.2`.

Local HTTP server hosted the HTML page, which was accessible via the resolved IP.

Demonstrated full workflow: **Domain → DNS → IP → HTTP Server → Browser**.

Conclusion

In Lab 10, we successfully simulated a local DNS server and accessed a web page using a custom domain name. This lab illustrates the core concept of DNS resolution and the interaction between DNS servers, clients, and HTTP servers in a networked environment.