++:LAB exercises for socket programming

Lab 1: Introduction to Socket Programming

- **Objective:** Understand basic concepts of socket programming and create a simple client-server application.
- **Tasks:**
  - Create a TCP client and server.
  - Exchange a simple message between the client and the server.

Lab 2: UDP Communication

- **Objective:** Learn about UDP sockets and their differences from TCP sockets.
- **Tasks:**
  - Create a UDP client and server.
  - Send and receive messages using UDP sockets.

Lab 3: File Transfer via Sockets

- **Objective:** Implement a basic file transfer system using sockets.
- **Tasks:**
  - Create a client-server application that transfers a file from the client to the server.
  - Ensure the file is correctly received and saved on the server.

Lab 4: Multi-client Handling

- **Objective:** Learn to handle multiple clients using multi-threading or multi-processing.
- **Tasks:**
  - Modify the TCP server to handle multiple clients concurrently.
  - Test the server with multiple clients connecting and communicating simultaneously.

Lab 5: Chat Application

- **Objective:** Develop a simple chat application that allows multiple clients to communicate with each other.
- **Tasks:**
  - Implement a TCP-based chat server.
  - Allow multiple clients to join and send messages to each other through the server.

Lab 6: Broadcast and Multicast

- **Objective:** Explore broadcast and multicast communication.

- **Tasks:**
    - Create a UDP-based application that sends broadcast messages.
    - Implement a multicast application where messages are sent to multiple clients in a multicast group.

Lab 7: Secure Socket Programming with SSL/TLS

- **Objective:** Learn to secure socket communications using SSL/TLS.
- **Tasks:**
    - Set up a basic SSL/TLS client and server using libraries like OpenSSL or Python's `ssl` module.
    - Exchange encrypted messages between the client and server.

Lab 8: Non-blocking and Asynchronous Sockets

- **Objective:** Understand non-blocking and asynchronous socket programming.
- **Tasks:**
    - Implement a non-blocking TCP server using the `select` module or similar mechanisms.
    - Create an asynchronous client-server application using libraries like `asyncio`.

Lab 9: Socket Programming with HTTP Protocol

- **Objective:** Implement a simple HTTP server using sockets.
- **Tasks:**
    - Create a basic HTTP server that can serve static HTML files.
    - Handle basic HTTP requests like GET and POST.

Lab 10: Advanced File Transfer with Error Handling and Compression

- **Objective:** Develop a more robust file transfer application with additional features.
- **Tasks:**
    - Implement error handling to manage network issues and interruptions.
    - Add file compression before transfer to reduce bandwidth usage.
    - Ensure file integrity after transfer using checksums or hashes.

These lab exercises will progressively build students' understanding and skills in socket programming, preparing them for real-world network programming challenges.

Lab 1: Introduction to Socket Programming

Objective:

Understand basic concepts of socket programming and create a simple client-server application.

Tasks:

- Create a TCP client and server.
- Exchange a simple message between the client and the server.

Lab Sheet

Introduction

In this lab, you will learn the basic concepts of socket programming. Sockets are endpoints for sending and receiving data across a network. You will create a simple TCP client-server application where the client sends a message to the server, and the server responds back to the client.

Prerequisites

- Basic knowledge of programming (preferably in Python)
- Understanding of networking concepts (IP addresses, ports, etc.)

Tools

- Python (you can download it from python.org)
- Any text editor or Integrated Development Environment (IDE)

Lab Setup

Ensure that Python is installed on your machine. You can check this by running the following command in your terminal or command prompt:

```
bash

python --version
```

Instructions

Part 1: Creating the TCP Server

1. Open your text editor or IDE and create a new file named `tcp_server.py`.
2. Write the following code in `tcp_server.py`:

```
python

import socket

def start_server():
    # Define server address and port
    server_address = '127.0.0.1'
    server_port = 65432

    # Create a TCP/IP socket
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    # Bind the socket to the address and port
    server_socket.bind((server_address, server_port))

    # Listen for incoming connections
    server_socket.listen(1)
    print(f"Server is listening on {server_address}:{server_port}")

    while True:
        # Wait for a connection
        connection, client_address = server_socket.accept()
        try:
            print(f"Connection from {client_address}")

            # Receive the data in small chunks and print it
            while True:
                data = connection.recv(1024)
                if data:
                    print(f"Received: {data.decode('utf-8')}")
                    # Send a response back to the client
                    connection.sendall(b"Message received")
                else:
                    break
        finally:
            # Clean up the connection
            connection.close()

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

3. Save the file.

Part 2: Creating the TCP Client

1. Create a new file named `tcp_client.py`.
2. Write the following code in `tcp_client.py`:

```
python

import socket

def start_client():
    # Define server address and port
    server_address = '127.0.0.1'
    server_port = 65432

    # Create a TCP/IP socket
    client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    # Connect the socket to the server's address and port
    client_socket.connect((server_address, server_port))

    try:
        # Send a message to the server
        message = "Hello, Server!"
        client_socket.sendall(message.encode('utf-8'))

        # Receive the response from the server
        response = client_socket.recv(1024)
        print(f"Received: {response.decode('utf-8')}")
    finally:
        # Close the connection
        client_socket.close()

if __name__ == "__main__":
    start_clien     t()
```

3. Save the file.

Running the Lab

1. Open two terminal windows or command prompts.
2. In the first terminal, navigate to the directory where `tcp_server.py` is saved and run the server:

   ```
   bash

   python tcp_server.py
   ```

3. In the second terminal, navigate to the directory where `tcp_client.py` is saved and run the client:

   ```
   bash

   python tcp_client.py
   ```

Expected Output

- The server terminal should display:

  ```
  vbnet

  Server is listening on 127.0.0.1:65432
  Connection from ('127.0.0.1', <random_port>)
  Received: Hello, Server!
  ```

- The client terminal should display:

  ```
  makefile

  Received: Message received
  ```

Conclusion

In this lab, you have successfully created a simple TCP client-server application. You have learned how to create sockets, bind them to addresses, listen for connections, and send/receive data. This forms the foundation for more advanced socket programming tasks.

Questions

1. What is the difference between TCP and UDP?
2. Why do we need to bind the server socket to an address and port?
3. What happens if you try to connect the client to a different port than the one the server is listening on?

By completing this lab, you have taken the first step in understanding socket programming. Future labs will build on these concepts, introducing more complexity and functionality.

Lab 2: UDP Communication

Objective:

Learn about UDP sockets and their differences from TCP sockets.

Tasks:

- Create a UDP client and server.
- Send and receive messages using UDP sockets.

Lab Sheet

Introduction

In this lab, you will learn about UDP (User Datagram Protocol) sockets. Unlike TCP, UDP is a connectionless protocol that does not guarantee message delivery, order, or duplicate protection. This makes UDP suitable for applications where speed is critical and occasional packet loss is acceptable.

Prerequisites

- Basic knowledge of programming (preferably in Python)
- Understanding of networking concepts (IP addresses, ports, etc.)

Tools

- Python (you can download it from python.org)
- Any text editor or Integrated Development Environment (IDE)

Lab Setup

Ensure that Python is installed on your machine. You can check this by running the following command in your terminal or command prompt:

```
bash

python --version
```

Instructions

Part 1: Creating the UDP Server

1. Open your text editor or IDE and create a new file named `udp_server.py`.
2. Write the following code in `udp_server.py`:

```
python

import socket

def start_server():
    # Define server address and port
    server_address = '127.0.0.1'
    server_port = 65432

    # Create a UDP socket
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

    # Bind the socket to the address and port
    server_socket.bind((server_address, server_port))

    print(f"Server is listening on {server_address}:{server_port}")

    while True:
        # Receive message from client
        data, client_address = server_socket.recvfrom(1024)
        print(f"Received: {data.decode('utf-8')} from {client_address}")

        # Send a response back to the client
        response = b"Message received"
        server_socket.sendto(response, client_address)

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

3. Save the file.

Part 2: Creating the UDP Client

1. Create a new file named `udp_client.py`.
2. Write the following code in `udp_client.py`:

```
python

import socket

def start_client():
    # Define server address and port
    server_address = '127.0.0.1'
    server_port = 65432

    # Create a UDP socket
    client_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

    # Send a message to the server
    message = "Hello, Server!"
    client_socket.sendto(message.encode('utf-8'), (server_address,
server_port))

    # Receive the response from the server
    response, server = client_socket.recvfrom(1024)
    print(f"Received: {response.decode('utf-8')}")

    # Close the socket
    client_socket.close()

if __name__ == "__main__":
```

```
start_client()
```

3. Save the file.

Running the Lab

1. Open two terminal windows or command prompts.
2. In the first terminal, navigate to the directory where `udp_server.py` is saved and run the server:

   ```
   bash

   python udp_server.py
   ```

3. In the second terminal, navigate to the directory where `udp_client.py` is saved and run the client:

   ```
   bash

   python udp_client.py
   ```

Expected Output

- The server terminal should display:

  ```
  vbnet

  Server is listening on 127.0.0.1:65432
  Received: Hello, Server! from ('127.0.0.1', <random_port>)
  ```

- The client terminal should display:

  ```
  makefile

  Received: Message received
  ```

Conclusion

In this lab, you have successfully created a simple UDP client-server application. You have learned how to create UDP sockets, bind them to addresses, and send/receive data without establishing a connection. This highlights the key differences between UDP and TCP, including the lack of connection establishment and reliability in UDP.

Questions

1. What are the main differences between TCP and UDP?

2. Why might you choose to use UDP over TCP in certain applications?
3. What potential issues might arise from using UDP in a network application?

By completing this lab, you have gained an understanding of UDP sockets and their use cases. Future labs will explore more advanced network programming concepts and applications.

Lab 3: File Transfer via Sockets

Objective:

Implement a basic file transfer system using sockets.

Tasks:

- Create a client-server application that transfers a file from the client to the server.
- Ensure the file is correctly received and saved on the server.

Lab Sheet

Introduction

In this lab, you will implement a basic file transfer system using TCP sockets. The client will send a file to the server, and the server will save the received file to disk. This exercise will help you understand how to handle binary data and ensure data integrity during transfer.

Prerequisites

- Basic knowledge of programming (preferably in Python)
- Understanding of networking concepts (IP addresses, ports, etc.)
- Basic understanding of file handling in Python

Tools

- Python (you can download it from python.org)
- Any text editor or Integrated Development Environment (IDE)

Lab Setup

Ensure that Python is installed on your machine. You can check this by running the following command in your terminal or command prompt:

```
bash

python --version
```

Instructions

Part 1: Creating the TCP Server

1. Open your text editor or IDE and create a new file named `file_server.py`.
2. Write the following code in `file_server.py`:

```
python

import socket

def start_server():
    # Define server address and port
    server_address = '127.0.0.1' #if another server use ‘0.0.0.0’
    server_port = 65432

    # Create a TCP/IP socket
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    # Bind the socket to the address and port
    server_socket.bind((server_address, server_port))

    # Listen for incoming connections
    server_socket.listen(1)
    print(f"Server is listening on {server_address}:{server_port}")

    while True:
        # Wait for a connection
        connection, client_address = server_socket.accept()
        try:
            print(f"Connection from {client_address}")

            # Open a file to write the incoming data
            with open('received_file.txt', 'wb') as file:
                while True:
                    data = connection.recv(1024)
                    if data:
                        file.write(data)
                    else:
                        break
            print("File received and saved as 'received_file.txt'")
        finally:
            # Clean up the connection
            connection.close()

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

3. Save the file.

Part 2: Creating the TCP Client

1. Create a new file named `file_client.py`.
2. Write the following code in `file_client.py`:

```
python

import socket

def start_client():
    # Define server address and port
    server_address = '127.0.0.1'
    server_port = 65432

    # Create a TCP/IP socket
    client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    # Connect the socket to the server's address and port
    client_socket.connect((server_address, server_port))

    # Open the file to be sent
    file_path = 'file_to_send.txt'
    with open(file_path, 'rb') as file:
        # Send the file data to the server
        while True:
            data = file.read(1024)
            if not data:
                break
            client_socket.sendall(data)

    # Close the connection
    client_socket.close()
    print(f"File '{file_path}' sent to the server.")

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

3. Save the file.

Part 3: Preparing the File to Send

1. Create a text file named `file_to_send.txt` in the same directory as your Python scripts.
2. Add some sample text or data to this file that will be sent to the server.

Running the Lab

1. Open two terminal windows or command prompts.
2. In the first terminal, navigate to the directory where `file_server.py` is saved and run the server:

   ```
   bash

   python file_server.py
   ```

3. In the second terminal, navigate to the directory where `file_client.py` is saved and run the client:

   ```
   bash

   python file_client.py
   ```

Expected Output

- The server terminal should display:

  ```
  vbnet

  Server is listening on 127.0.0.1:65432
  ```

```
Connection from ('127.0.0.1', <random_port>)
File received and saved as 'received_file.txt'
```

- The client terminal should display:

  ```
  arduino
  
  File 'file_to_send.txt' sent to the server.
  ```

- Verify that the `received_file.txt` on the server contains the same content as `file_to_send.txt` on the client.

Conclusion

In this lab, you have successfully implemented a basic file transfer system using TCP sockets. You have learned how to handle binary data and ensure that files are correctly transferred and saved. This is a fundamental skill in network programming, useful for various applications like file sharing, remote backups, and more.

Questions

1. What are the key differences between transferring text data and binary data over sockets?
2. How can you ensure data integrity during file transfer?
3. What modifications would you need to make to transfer files of different types (e.g., images, PDFs)?

By completing this lab, you have gained practical experience in file transfer using sockets, a crucial aspect of network programming. Future labs will continue to build on these concepts, introducing more complexity and advanced features.

Lab 4: Multi-client Handling

Objective:

Learn to handle multiple clients using multi-threading or multi-processing.

Tasks:

- Modify the TCP server to handle multiple clients concurrently.
- Test the server with multiple clients connecting and communicating simultaneously.

Lab Sheet

Introduction

In this lab, you will learn how to handle multiple clients concurrently using multi-threading or multi-processing in a TCP server. This is essential for creating scalable network applications that can serve multiple clients at the same time.

Prerequisites

- Basic knowledge of programming (preferably in Python)
- Understanding of networking concepts (IP addresses, ports, etc.)
- Basic understanding of multi-threading or multi-processing

Tools

- Python (you can download it from python.org)
- Any text editor or Integrated Development Environment (IDE)

Lab Setup

Ensure that Python is installed on your machine. You can check this by running the following command in your terminal or command prompt:

```
bash

python --version
```

Instructions

Part 1: Creating the Multi-client TCP Server

1. Open your text editor or IDE and create a new file named `multi_client_server.py`.
2. Write the following code in `multi_client_server.py`:

```
python

import socket
import threading
```

```
def handle_client(connection, client_address):
    print(f"Connection from {client_address}")
    try:
        while True:
            data = connection.recv(1024)
            if data:
                print(f"Received from {client_address}:
{data.decode('utf-8')}")
                connection.sendall(b"Message received")
            else:
                break
    finally:
        connection.close()

def start_server():
    # Define server address and port
    server_address = '127.0.0.1'
    server_port = 65432

    # Create a TCP/IP socket
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    # Bind the socket to the address and port
    server_socket.bind((server_address, server_port))

    # Listen for incoming connections
    server_socket.listen(5)
    print(f"Server is listening on {server_address}:{server_port}")

    while True:
        # Wait for a connection
        connection, client_address = server_socket.accept()
        # Handle the new connection in a separate thread
        client_thread = threading.Thread(target=handle_client,
args=(connection, client_address))
        client_thread.start()

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

3. Save the file.

Part 2: Creating the TCP Client

1. Create a new file named `multi_client.py`.
2. Write the following code in `multi_client.py`:

```
python

import socket

def start_client(message):
    # Define server address and port
    server_address = '127.0.0.1'
    server_port = 65432

    # Create a TCP/IP socket
    client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    # Connect the socket to the server's address and port
    client_socket.connect((server_address, server_port))

    try:
        # Send a message to the server
        client_socket.sendall(message.encode('utf-8'))

        # Receive the response from the server
        response = client_socket.recv(1024)
        print(f"Received: {response.decode('utf-8')}")
    finally:
        # Close the connection
        client_socket.close()

if __name__ == "__main__":
    message = "Hello from client!"
    start_client(message)
```

3. Save the file.

Running the Lab

1. Open a terminal window or command prompt.
2. Navigate to the directory where `multi_client_server.py` is saved and run the server:

   ```
   bash

   python multi_client_server.py
   ```

3. Open multiple terminal windows or command prompts.
4. In each terminal, navigate to the directory where `multi_client.py` is saved and run the client with different messages:

   ```
   bash

   python multi_client.py
   ```

Expected Output

- The server terminal should display messages indicating connections from multiple clients and the messages received from each client.
- Each client terminal should display:

  ```
  makefile
  ```

```
Received: Message received
```

Conclusion

In this lab, you have successfully modified a TCP server to handle multiple clients concurrently using multi-threading. You have learned how to create a scalable server that can serve multiple clients simultaneously. This is a crucial skill for developing robust network applications.

Questions

1. What are the benefits of using multi-threading in a TCP server?
2. How can you ensure thread safety when handling multiple clients?
3. What are the potential drawbacks of using multi-threading for handling multiple clients?

By completing this lab, you have gained practical experience in handling multiple clients concurrently in a TCP server using multi-threading. Future labs will explore more advanced concepts and applications in network programming.

Lab 5: Chat Application

Objective:

Develop a simple chat application that allows multiple clients to communicate with each other.

Tasks:

- Implement a TCP-based chat server.
- Allow multiple clients to join and send messages to each other through the server.

Lab Sheet

Introduction

In this lab, you will develop a simple chat application using TCP sockets. The application will consist of a server that can handle multiple clients simultaneously, allowing them to send and receive messages to and from each other.

Prerequisites

- Basic knowledge of programming (preferably in Python)
- Understanding of networking concepts (IP addresses, ports, etc.)
- Basic understanding of multi-threading or multi-processing

Tools

- Python (you can download it from python.org)
- Any text editor or Integrated Development Environment (IDE)

Lab Setup

Ensure that Python is installed on your machine. You can check this by running the following command in your terminal or command prompt:

```
bash

python --version
```

Instructions

Part 1: Creating the Chat Server

1. Open your text editor or IDE and create a new file named `chat_server.py`.
2. Write the following code in `chat_server.py`:

```
python

import socket
import threading

clients = []
```

```
def broadcast(message, current_client):
    for client in clients:
        if client != current_client:
            try:
                client.send(message)
            except:
                client.close()
                clients.remove(client)

def handle_client(client_socket):
    while True:
        try:
            message = client_socket.recv(1024)
            if message:
                broadcast(message, client_socket)
            else:
                break
        except:
            clients.remove(client_socket)
            client_socket.close()
            break

def start_server():
    # Define server address and port
    server_address = '127.0.0.1'
    server_port = 65432

    # Create a TCP/IP socket
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    # Bind the socket to the address and port
    server_socket.bind((server_address, server_port))

    # Listen for incoming connections
    server_socket.listen(5)
    print(f"Server is listening on {server_address}:{server_port}")

    while True:
        # Accept a new connection
        client_socket, client_address = server_socket.accept()
        print(f"Connection from {client_address}")

        # Add the new client to the list of clients
        clients.append(client_socket)

        # Handle the new connection in a separate thread
        client_thread = threading.Thread(target=handle_client,
args=(client_socket,))
        client_thread.start()

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

3. Save the file.

Part 2: Creating the Chat Client

1. Create a new file named `chat_client.py`.
2. Write the following code in `chat_client.py`:

```
python

import socket
import threading

def receive_messages(client_socket):
    while True:
        try:
            message = client_socket.recv(1024).decode('utf-8')
            if message:
                print(message)
        except:
            print("An error occurred!")
            client_socket.close()
            break

def start_client():
    # Define server address and port
    server_address = '127.0.0.1'
    server_port = 65432

    # Create a TCP/IP socket
    client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    # Connect the socket to the server's address and port
    client_socket.connect((server_address, server_port))

    # Start a thread to receive messages from the server
    receive_thread = threading.Thread(target=receive_messages,
args=(client_socket,))
    receive_thread.start()

    # Send messages to the server
    while True:
        message = input()
        client_socket.send(message.encode('utf-8'))

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

3. Save the file.

Running the Lab

1. Open a terminal window or command prompt.
2. Navigate to the directory where `chat_server.py` is saved and run the server:

   ```
   bash
   ```

```
python chat_server.py
```

3. Open multiple terminal windows or command prompts for each client.
4. In each terminal, navigate to the directory where `chat_client.py` is saved and run the client:

   ```bash
   python chat_client.py
   ```

5. In each client terminal, type messages and observe that they are received by other connected clients.

Expected Output

- The server terminal should display messages indicating connections from multiple clients.
- Each client terminal should display messages sent by other clients.

Conclusion

In this lab, you have successfully implemented a TCP-based chat server and client that allow multiple clients to communicate with each other. You have learned how to handle multiple clients concurrently and broadcast messages to all connected clients.

Questions

1. How does the server differentiate between messages from different clients?
2. What are some potential issues that could arise in a real-world chat application?
3. How could you extend this chat application to support private messaging between clients?

By completing this lab, you have gained practical experience in developing a multi-client chat application using TCP sockets. Future labs will continue to build on these concepts, introducing more advanced features and optimizations.

Lab 6: Broadcast and Multicast

Objective:

Explore broadcast and multicast communication.

Tasks:

- Create a UDP-based application that sends broadcast messages.
- Implement a multicast application where messages are sent to multiple clients in a multicast group.

Lab Sheet

Introduction

In this lab, you will explore broadcast and multicast communication using UDP sockets. Broadcast allows a message to be sent to all devices in a local network, while multicast allows a message to be sent to a specific group of devices.

Prerequisites

- Basic knowledge of programming (preferably in Python)
- Understanding of networking concepts (IP addresses, ports, etc.)
- Basic understanding of UDP sockets

Tools

- Python (you can download it from python.org)
- Any text editor or Integrated Development Environment (IDE)

Lab Setup

Ensure that Python is installed on your machine. You can check this by running the following command in your terminal or command prompt:

```
bash

python --version
```

Instructions

Part 1: Creating a UDP Broadcast Application

UDP Broadcast Sender

1. Open your text editor or IDE and create a new file named `broadcast_sender.py`.
2. Write the following code in `broadcast_sender.py`:

```
python

import socket
import time
```

```
def broadcast_message():
    # Define the broadcast address and port
    broadcast_address = '<broadcast>'
    broadcast_port = 65432

    # Create a UDP socket
    udp_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

    # Set the socket option to enable broadcast
    udp_socket.setsockopt(socket.SOL_SOCKET, socket.SO_BROADCAST, 1)

    while True:
        message = "Hello, Broadcast!"
        udp_socket.sendto(message.encode('utf-8'), (broadcast_address,
broadcast_port))
        print(f"Broadcasted: {message}")
        time.sleep(2)  # Wait for 2 seconds before sending the next message

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

3. Save the file.

UDP Broadcast Receiver

1. Create a new file named `broadcast_receiver.py`.
2. Write the following code in `broadcast_receiver.py`:

```
python

import socket

def receive_broadcast():
    # Define the listening address and port
    listen_address = '0.0.0.0'
    listen_port = 65432

    # Create a UDP socket
    udp_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

    # Bind the socket to the listening address and port
    udp_socket.bind((listen_address, listen_port))

    print(f"Listening for broadcasts on {listen_address}:{listen_port}")

    while True:
        message, address = udp_socket.recvfrom(1024)
        print(f"Received from {address}: {message.decode('utf-8')}")

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

3. Save the file.

Part 2: Creating a UDP Multicast Application

UDP Multicast Sender

1. Create a new file named `multicast_sender.py`.
2. Write the following code in `multicast_sender.py`:

```
python

import socket
import struct
import time

def multicast_message():
    # Define the multicast group and port
    multicast_group = '224.1.1.1'
    multicast_port = 65432

    # Create a UDP socket
    udp_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

    # Set the socket option to enable multicast
    ttl = struct.pack('b', 1)
    udp_socket.setsockopt(socket.IPPROTO_IP, socket.IP_MULTICAST_TTL, ttl)

    while True:
        message = "Hello, Multicast!"
        udp_socket.sendto(message.encode('utf-8'), (multicast_group,
multicast_port))
        print(f"Multicasted: {message}")
        time.sleep(2)  # Wait for 2 seconds before sending the next message

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

3. Save the file.

UDP Multicast Receiver

1. Create a new file named `multicast_receiver.py`.
2. Write the following code in `multicast_receiver.py`:

```
python

import socket
import struct

def receive_multicast():
    # Define the multicast group and port
    multicast_group = '224.1.1.1'
    multicast_port = 65432

    # Create a UDP socket
    udp_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

    # Bind the socket to the multicast port
    udp_socket.bind(('', multicast_port))

    # Set the socket option to join the multicast group
    group = socket.inet_aton(multicast_group)
    mreq = struct.pack('4sL', group, socket.INADDR_ANY)
    udp_socket.setsockopt(socket.IPPROTO_IP, socket.IP_ADD_MEMBERSHIP, mreq)

    print(f"Listening for multicast messages on {multicast_group}:
{multicast_port}")

    while True:
        message, address = udp_socket.recvfrom(1024)
        print(f"Received from {address}: {message.decode('utf-8')}")

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

3. Save the file.

Running the Lab

Part 1: Running the Broadcast Application

1. Open a terminal window or command prompt.
2. Navigate to the directory where `broadcast_sender.py` is saved and run the broadcast sender:

   ```
   bash

   python broadcast_sender.py
   ```

3. Open another terminal window or command prompt.
4. Navigate to the directory where `broadcast_receiver.py` is saved and run the broadcast receiver:

   ```
   bash

   python broadcast_receiver.py
   ```

Part 2: Running the Multicast Application

1. Open a terminal window or command prompt.
2. Navigate to the directory where `multicast_sender.py` is saved and run the multicast sender:

   ```
   bash

   python multicast_sender.py
   ```

3. Open another terminal window or command prompt.
4. Navigate to the directory where `multicast_receiver.py` is saved and run the multicast receiver:

   ```
   bash

   python multicast_receiver.py
   ```

Expected Output

- The broadcast receiver should display the messages sent by the broadcast sender.
- The multicast receiver should display the messages sent by the multicast sender.

Conclusion

In this lab, you have successfully created UDP-based applications for both broadcast and multicast communication. You have learned how to send and receive broadcast messages in a local network and how to send and receive multicast messages in a specific group.

Questions

1. What are the key differences between broadcast and multicast communication?
2. In what scenarios would you prefer using multicast over broadcast?
3. How can you handle message filtering in a multicast application to ensure only relevant messages are processed?

By completing this lab, you have gained practical experience in broadcast and multicast communication using UDP sockets. Future labs will explore more advanced networking concepts and applications.

Lab 7: Secure Socket Programming with SSL/TLS

Objective:

Learn to secure socket communications using SSL/TLS.

Tasks:

- Set up a basic SSL/TLS client and server using libraries like OpenSSL or Python's ssl module.
- Exchange encrypted messages between the client and server.

Lab Sheet

Introduction

In this lab, you will learn how to secure socket communications using SSL/TLS. SSL (Secure Sockets Layer) and its successor TLS (Transport Layer Security) provide encrypted communication over a computer network. You will set up a basic SSL/TLS client and server using Python's `ssl` module and exchange encrypted messages.

Prerequisites

- Basic knowledge of programming (preferably in Python)
- Understanding of networking concepts (IP addresses, ports, etc.)
- Basic understanding of socket programming

Tools

- Python (you can download it from python.org)
- OpenSSL (for generating certificates, can be installed from openssl.org)
- Any text editor or Integrated Development Environment (IDE)

Lab Setup

Ensure that Python and OpenSSL are installed on your machine. You can check this by running the following commands in your terminal or command prompt:

```
bash

python --version
openssl version
```

Instructions

Part 1: Generating SSL/TLS Certificates

1. Open a terminal window or command prompt.
2. Navigate to the directory where you want to store your certificates.
3. Run the following commands to generate a self-signed certificate and private key:

```
bash

openssl genpkey -algorithm RSA -out private_key.pem
openssl req -new -key private_key.pem -out cert_request.csr
openssl x509 -req -days 365 -in cert_request.csr -signkey
private_key.pem -out certificate.pem
```

Part 2: Creating the SSL/TLS Server

1. Open your text editor or IDE and create a new file named `ssl_server.py`.
2. Write the following code in `ssl_server.py`:

```
python

import socket
import ssl

def start_server():
    # Define server address and port
    server_address = '127.0.0.1'
    server_port = 65432

    # Create a TCP/IP socket
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    # Bind the socket to the address and port
    server_socket.bind((server_address, server_port))

    # Listen for incoming connections
    server_socket.listen(5)
    print(f"Server is listening on {server_address}:{server_port}")

    # Wrap the socket with SSL
    context = ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER)
    context.load_cert_chain(certfile='certificate.pem',
keyfile='private_key.pem')

    while True:
        # Accept a new connection
        client_socket, client_address = server_socket.accept()
        print(f"Connection from {client_address}")

        # Wrap the client socket with SSL
        ssl_client_socket = context.wrap_socket(client_socket,
server_side=True)

        try:
            message = ssl_client_socket.recv(1024).decode('utf-8')
            print(f"Received from {client_address}: {message}")
            ssl_client_socket.send("Message received
securely".encode('utf-8'))
        finally:
            ssl_client_socket.close()

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

3. Save the file.

Part 3: Creating the SSL/TLS Client

1. Create a new file named `ssl_client.py`.
2. Write the following code in `ssl_client.py`:

```
python

import socket
import ssl

def start_client():
    # Define server address and port
    server_address = '127.0.0.1'
    server_port = 65432

    # Create a TCP/IP socket
    client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    # Wrap the socket with SSL
    context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
    context.load_verify_locations('certificate.pem')

    ssl_client_socket = context.wrap_socket(client_socket,
server_hostname=server_address)

    # Connect the socket to the server's address and port
    ssl_client_socket.connect((server_address, server_port))

    try:
        message = "Hello, Secure Server!"
        ssl_client_socket.send(message.encode('utf-8'))
        response = ssl_client_socket.recv(1024).decode('utf-8')
        print(f"Received: {response}")
    finally:
        ssl_client_socket.close()

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

3. Save the file.

Running the Lab

1. Open a terminal window or command prompt.
2. Navigate to the directory where `ssl_server.py` is saved and run the server:

   ```
   bash

   python ssl_server.py
   ```

3. Open another terminal window or command prompt.
4. Navigate to the directory where `ssl_client.py` is saved and run the client:

   ```
   bash

   python ssl_client.py
   ```

Expected Output

- The server terminal should display messages indicating connections from the client and the messages received securely.
- The client terminal should display:

  ```
  makefile

  Received: Message received securely
  ```

Conclusion

In this lab, you have successfully set up a basic SSL/TLS client and server using Python's `ssl` module. You have learned how to secure socket communications and exchange encrypted messages between the client and server.

Questions

1. What are the benefits of using SSL/TLS for socket communication?
2. How does SSL/TLS ensure the security and integrity of the messages?
3. What are the potential challenges or limitations of implementing SSL/TLS in real-world applications?

By completing this lab, you have gained practical experience in securing socket communications using SSL/TLS. Future labs will explore more advanced security concepts and applications.

Lab 8: Non-blocking and Asynchronous Sockets

Objective:

Understand non-blocking and asynchronous socket programming.

Tasks:

- Implement a non-blocking TCP server using the select module or similar mechanisms.
- Create an asynchronous client-server application using libraries like asyncio.

Lab Sheet

Introduction

In this lab, you will learn about non-blocking and asynchronous socket programming. Non-blocking sockets allow you to perform other tasks while waiting for network operations to complete, and asynchronous programming enables you to handle multiple tasks concurrently using libraries like `asyncio`.

Prerequisites

- Basic knowledge of programming (preferably in Python)
- Understanding of networking concepts (IP addresses, ports, etc.)
- Basic understanding of socket programming

Tools

- Python (you can download it from python.org)
- Any text editor or Integrated Development Environment (IDE)

Lab Setup

Ensure that Python is installed on your machine. You can check this by running the following command in your terminal or command prompt:

```
bash

python --version
```

Instructions

Part 1: Implementing a Non-blocking TCP Server

1. Open your text editor or IDE and create a new file named `non_blocking_server.py`.
2. Write the following code in `non_blocking_server.py`:

```
python

import socket
import select

def start_non_blocking_server():
    # Define server address and port
    server_address = '127.0.0.1'
    server_port = 65432

    # Create a TCP/IP socket
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_socket.setblocking(False)

    # Bind the socket to the address and port
    server_socket.bind((server_address, server_port))

    # Listen for incoming connections
    server_socket.listen(5)
    print(f"Server is listening on {server_address}:{server_port}")

    # List of sockets to monitor for incoming connections
    sockets_list = [server_socket]
    clients = {}

    def receive_message(client_socket):
        try:
            message = client_socket.recv(1024)
            if not message:
                return False
            return message.decode('utf-8')
        except:
            return False

    while True:
        read_sockets, _, exception_sockets = select.select(sockets_list, [],
sockets_list)

        for notified_socket in read_sockets:
            if notified_socket == server_socket:
                client_socket, client_address = server_socket.accept()
                client_socket.setblocking(False)
                sockets_list.append(client_socket)
                clients[client_socket] = client_address
                print(f"Accepted new connection from {client_address}")
            else:
                message = receive_message(notified_socket)
                if message is False:
                    print(f"Closed connection from
{clients[notified_socket]}")
                    sockets_list.remove(notified_socket)
                    del clients[notified_socket]
                    continue

                print(f"Received message from {clients[notified_socket]}:
{message}")
                notified_socket.send("Message received".encode('utf-8'))

        for notified_socket in exception_sockets:
            sockets_list.remove(notified_socket)
            del clients[notified_socket]

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

3. Save the file.

Part 2: Creating an Asynchronous Client-Server Application using `asyncio`

Asynchronous Server

1. Create a new file named `async_server.py`.
2. Write the following code in `async_server.py`:

```
python

import asyncio

async def handle_client(reader, writer):
    address = writer.get_extra_info('peername')
    print(f"Accepted connection from {address}")

    while True:
        data = await reader.read(100)
        if not data:
            print(f"Closed connection from {address}")
            writer.close()
            await writer.wait_closed()
            break

        message = data.decode()
        print(f"Received message from {address}: {message}")

        response = "Message received"
        writer.write(response.encode())
        await writer.drain()

async def start_async_server():
    server = await asyncio.start_server(handle_client, '127.0.0.1', 65432)
    address = server.sockets[0].getsockname()
    print(f"Server is listening on {address}")

    async with server:
        await server.serve_forever()

if __name__ == "__main__":
    asyncio.run(start_async_server())
```

3. Save the file.

Asynchronous Client

1. Create a new file named `async_client.py`.

2. Write the following code in `async_client.py`:

```
python

import asyncio

async def tcp_client(message):
    reader, writer = await asyncio.open_connection('127.0.0.1', 65432)

    print(f'Sending: {message}')
    writer.write(message.encode())
    await writer.drain()

    data = await reader.read(100)
    print(f'Received: {data.decode()}')

    print('Closing the connection')
    writer.close()
    await writer.wait_closed()

if __name__ == "__main__":
    message = "Hello, Async Server!"
    asyncio.run(tcp_client(message))
```

3. Save the file.

Running the Lab

Part 1: Running the Non-blocking TCP Server

1. Open a terminal window or command prompt.
2. Navigate to the directory where `non_blocking_server.py` is saved and run the server:

   ```
   bash

   python non_blocking_server.py
   ```

3. Open another terminal window or command prompt.
4. Use `telnet` or create a simple client script to connect to the server and send messages.

Part 2: Running the Asynchronous Client-Server Application

1. Open a terminal window or command prompt.
2. Navigate to the directory where `async_server.py` is saved and run the server:

   ```
   bash

   python async_server.py
   ```

3. Open another terminal window or command prompt.
4. Navigate to the directory where `async_client.py` is saved and run the client:

   ```
   bash

   python async_client.py
   ```

Expected Output

- For the non-blocking TCP server, the terminal should display messages indicating connections from clients and the messages received.
- For the asynchronous client-server application, the server terminal should display messages indicating connections from clients and the messages received. The client terminal should display:

  ```
  vbnet

  Sending: Hello, Async Server!
  Received: Message received
  Closing the connection
  ```

Conclusion

In this lab, you have successfully implemented non-blocking and asynchronous socket programming using the `select` module and `asyncio` library. You have learned how to handle multiple connections concurrently without blocking the main thread.

Questions

1. What are the advantages of non-blocking and asynchronous socket programming over traditional blocking socket programming?
2. How does the `select` module help in handling multiple connections in a non-blocking manner?
3. What are the key features of the `asyncio` library that make it suitable for asynchronous programming?

By completing this lab, you have gained practical experience in non-blocking and asynchronous socket programming. Future labs will explore more advanced networking concepts and applications.

Lab 9: Socket Programming with HTTP Protocol

Objective:

Implement a simple HTTP server using sockets.

Tasks:

- Create a basic HTTP server that can serve static HTML files.
- Handle basic HTTP requests like GET and POST.

Lab Sheet

Introduction

In this lab, you will learn how to implement a simple HTTP server using sockets. You will create a basic HTTP server that can serve static HTML files and handle basic HTTP requests such as GET and POST.

Prerequisites

- Basic knowledge of programming (preferably in Python)
- Understanding of networking concepts (IP addresses, ports, etc.)
- Basic understanding of socket programming and HTTP protocol

Tools

- Python (you can download it from python.org)
- Any text editor or Integrated Development Environment (IDE)

Lab Setup

Ensure that Python is installed on your machine. You can check this by running the following command in your terminal or command prompt:

```
bash

python --version
```

Instructions

Part 1: Creating a Basic HTTP Server

1. Open your text editor or IDE and create a new file named `http_server.py`.
2. Write the following code in `http_server.py`:

```
python

import socket

def handle_request(request):
    # Parse HTTP request
    headers = request.split('\r\n')
    first_line = headers[0].split(' ')
    method = first_line[0]
    path = first_line[1]

    # Generate response based on request method
    if method == 'GET':
        return handle_get(path)
    elif method == 'POST':
        return handle_post(headers, request)
    else:
        return 'HTTP/1.1 405 Method Not Allowed\r\n\r\n'

def handle_get(path):
    if path == '/':
        path = '/index.html'

    try:
        with open(f'.{path}', 'r') as file:
            content = file.read()
        response = 'HTTP/1.1 200 OK\r\nContent-Type: text/html\r\n\r\n' +
content
    except FileNotFoundError:
        response = 'HTTP/1.1 404 Not Found\r\n\r\n'

    return response

def handle_post(headers, request):
    # For simplicity, we won't actually handle POST data in this example
    response = 'HTTP/1.1 200 OK\r\nContent-Type: text/html\r\n\r\nPOST
request received'
    return response

def start_server():
    server_address = '127.0.0.1'
    server_port = 8080

    # Create a TCP/IP socket
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    # Bind the socket to the address and port
    server_socket.bind((server_address, server_port))

    # Listen for incoming connections
    server_socket.listen(5)
    print(f"Server is listening on {server_address}:{server_port}")

    while True:
        # Accept a new connection
        client_socket, client_address = server_socket.accept()
        print(f"Connection from {client_address}")

        # Receive the request
        request = client_socket.recv(1024).decode('utf-8')
        print(f"Request: {request}")

        # Handle the request and generate a response
        response = handle_request(request)
        # Send the response
        client_socket.sendall(response.encode('utf-8'))

        # Close the connection
        client_socket.close()

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

3. Save the file.

Part 2: Creating Static HTML Files

1. Create a directory named `static` in the same directory as your `http_server.py` file.
2. Inside the `static` directory, create a file named `index.html` and add the following content:

```
html

<!DOCTYPE html>
<html>
<head>
    <title>My Simple HTTP Server</title>
</head>
<body>
    <h1>Welcome to My Simple HTTP Server</h1>
    <p>This is a basic HTTP server implemented using Python sockets.</p>
</body>
</html>
```

3. Save the file.

Running the Lab

1. Open a terminal window or command prompt.
2. Navigate to the directory where `http_server.py` is saved and run the server:

   ```
   bash

   python http_server.py
   ```

3. Open a web browser and navigate to `http://127.0.0.1:8080`. You should see the content of the `index.html` file displayed in the browser.
4. To test POST requests, you can use tools like `curl` or Postman. For example, using `curl`:

   ```
   bash
   ```

```
curl -X POST http://127.0.0.1:8080
```

Expected Output

- When navigating to `http://127.0.0.1:8080` in a web browser, the content of the `index.html` file should be displayed.
- When making a POST request using `curl` or Postman, the response should be `POST request received`.

Conclusion

In this lab, you have successfully implemented a simple HTTP server using sockets. You have learned how to serve static HTML files and handle basic HTTP requests like GET and POST.

Questions

1. What are the key components of an HTTP request and response?
2. How can you extend the HTTP server to handle other types of requests or serve different types of files?
3. What are some potential security considerations when implementing an HTTP server?

By completing this lab, you have gained practical experience in implementing an HTTP server using sockets. Future labs will explore more advanced web server concepts and applications.

Lab 10: Advanced File Transfer with Error Handling and Compression

Objective:

Develop a more robust file transfer application with additional features.

Tasks:

- Implement error handling to manage network issues and interruptions.
- Add file compression before transfer to reduce bandwidth usage.
- Ensure file integrity after transfer using checksums or hashes.

Lab Sheet

Introduction

In this lab, you will enhance a basic file transfer application to include error handling, file compression, and integrity checks. These features will make the file transfer process more robust, efficient, and reliable.

Prerequisites

- Basic knowledge of programming (preferably in Python)
- Understanding of networking concepts (IP addresses, ports, etc.)
- Basic understanding of socket programming

Tools

- Python (you can download it from python.org)
- Any text editor or Integrated Development Environment (IDE)

Lab Setup

Ensure that Python is installed on your machine. You can check this by running the following command in your terminal or command prompt:

```
bash
Copy code
python --version
```

Instructions

Part 1: Setting Up the Server

1. Open your text editor or IDE and create a new file named `advanced_file_transfer_server.py`.
2. Write the following code in `advanced_file_transfer_server.py`:

```
python
Copy code
import socket
import zlib
import hashlib

def handle_client(client_socket):
    try:
        # Receive the file size
        file_size = int(client_socket.recv(1024).decode('utf-8'))
        client_socket.send("ACK".encode('utf-8'))

        # Receive the file data
        received_data = b""
        while len(received_data) < file_size:
            packet = client_socket.recv(4096)
            if not packet:
                break
            received_data += packet

        # Decompress the received data
        decompressed_data = zlib.decompress(received_data)

        # Calculate the received file hash
        received_file_hash = client_socket.recv(64).decode('utf-8')

        # Calculate the hash of the received file data
        calculated_hash = hashlib.sha256(decompressed_data).hexdigest()

        # Verify the file integrity
        if received_file_hash == calculated_hash:
            with open("received_file", "wb") as file:
                file.write(decompressed_data)
            client_socket.send("File received successfully and
verified".encode('utf-8'))
        else:
            client_socket.send("File integrity check failed".encode('utf-8'))

    except Exception as e:
        print(f"Error: {e}")
        client_socket.send("Error during file transfer".encode('utf-8'))

    finally:
        client_socket.close()

def start_server():
    server_address = '127.0.0.1'
    server_port = 65432

    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_socket.bind((server_address, server_port))
    server_socket.listen(5)
    print(f"Server is listening on {server_address}:{server_port}")

    while True:
        client_socket, client_address = server_socket.accept()
        print(f"Connection from {client_address}")
        handle_client(client_socket)

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

3. Save the file.

Part 2: Setting Up the Client

1. Create a new file named `advanced_file_transfer_client.py`.
2. Write the following code in `advanced_file_transfer_client.py`:

```
python
Copy code
import socket
import zlib
import hashlib

def send_file(file_path, server_address, server_port):
    try:
        with open(file_path, "rb") as file:
            file_data = file.read()

        # Compress the file data
        compressed_data = zlib.compress(file_data)

        # Calculate the file hash
        file_hash = hashlib.sha256(file_data).hexdigest()

        # Create a TCP/IP socket
        client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

        # Connect to the server
        client_socket.connect((server_address, server_port))

        # Send the file size
        client_socket.send(str(len(compressed_data)).encode('utf-8'))
        client_socket.recv(3)  # Wait for ACK

        # Send the compressed file data
        client_socket.sendall(compressed_data)

        # Send the file hash
        client_socket.send(file_hash.encode('utf-8'))

        # Receive the server response
        response = client_socket.recv(1024).decode('utf-8')
        print(f"Server response: {response}")

    except Exception as e:
        print(f"Error: {e}")

    finally:
        client_socket.close()

if __name__ == "__main__":
    file_path = "file_to_send"
    server_address = '127.0.0.1'
    server_port = 65432
    send_file(file_path, server_address, server_port)
```

3. Save the file.

Running the Lab

1. Prepare a file to be transferred. Create a file named `file_to_send` in the same directory as your client script.
2. Open a terminal window or command prompt.
3. Navigate to the directory where `advanced_file_transfer_server.py` is saved and run the server:

   ```
   bash
   Copy code
   python advanced_file_transfer_server.py
   ```

4. Open another terminal window or command prompt.
5. Navigate to the directory where `advanced_file_transfer_client.py` is saved and run the client:

   ```
   bash
   Copy code
   python advanced_file_transfer_client.py
   ```

Expected Output

- The server terminal should display messages indicating connections from the client and the status of the file transfer.
- The client terminal should display the server response indicating the success or failure of the file transfer.

Conclusion

In this lab, you have successfully implemented an advanced file transfer application with error handling, file compression, and integrity checks. You have learned how to manage network issues, reduce bandwidth usage, and ensure file integrity during transfer.

Questions

1. What are the advantages of compressing files before transferring them over the network?
2. How does error handling improve the robustness of a file transfer application?
3. What are some common methods to ensure file integrity after transfer, and how do they work?

By completing this lab, you have gained practical experience in developing a robust file transfer application with additional features. Future labs will explore more advanced networking and application development concepts.