ADVANCED C PROGRAMMING

MODULE 4 ASSESSMENT

1. Explain the connection procedure followed in client server communication.

In client-server communication, the connection procedure typically follows a structured series of steps that enable a client (a device or software that requests data or services) to communicate with a server (a device or software that provides the requested data or services). The specifics can vary depending on the protocols and technologies used, but a general overview of the connection procedure can be described as follows:

1. Address Resolution

- Client Side: The client needs the server's address to communicate. This usually involves resolving a human-readable domain name (like www.example.com) to an IP address, which is done through the Domain Name System (DNS).

- DNS Query: The client sends a query to a DNS server to get the IP address associated with the domain name.

2. Connection Initiation (TCP/IP)

Assuming a TCP/IP connection (common for many applications, including web browsing):

- SYN Packet: The client initiates the connection by sending a SYN (synchronize) packet to the server, specifying the client's initial sequence number for the data it sends.

- SYN-ACK Packet: The server responds with a SYN-ACK (synchronize-acknowledge) packet, acknowledging the client's sequence number and providing its own initial sequence number.

- ACK Packet: The client sends an ACK (acknowledge) packet back to the server, acknowledging the server's sequence number.

This three-way handshake process establishes a reliable connection between the client and the server.

3. Secure Layer (Optional)

For secure communications (e.g., HTTPS), an additional layer of handshake occurs:

- TLS/SSL Handshake: After the TCP connection is established, if encryption is required, a TLS (Transport Layer Security) or SSL (Secure Sockets Layer) handshake is performed. This involves the exchange of certificates, negotiation of encryption algorithms, and the generation and exchange of keys.

4. Data Transfer

- Request: Once the connection is established (and secure if required), the client sends a request to the server. This request could be for a webpage, data submission, API call, etc.

- Response: The server processes the request and sends back a response. This could include the requested data, status messages, errors, or further instructions.

1. What is the use of bind() function in socket programming ?

The `bind()` function in socket programming is used to associate a socket with a specific local IP address and port number. This step is crucial for servers because it defines the address at which the server will listen for incoming connections from clients. Here's a breakdown of its significance and use:

Purpose of `bind()`:

- Identifies a Socket: It tells the operating system that a given socket should be associated with a specific local IP address and port number. This is particularly important for server applications that need to listen for incoming connections on a known IP address and port.

- Enables Port Reuse: Through the `bind()` function, servers can specify a particular port to be reused. This is useful for services that restart frequently and must bind to the same port every time.

- Supports Specific Networking Interfaces: In systems with multiple networking interfaces (e.g., Ethernet, Wi-Fi, virtual networks), `bind()` allows an application to select which interface it wants to listen on. Without binding, the socket might accept connections on all interfaces.

1. What is Datagram Socket?

A datagram socket is a type of socket used in communication between networks where there is no need to establish a connection (i.e) connectionless networks. The main difference is stream sockets can provide reliable, orderly and bidirectional byte stream transfer whereas datagram sockets are unordered, unreliable and are used for transmitting discrete messages. Datagram sockets do not establish a connection between the two nodes or sockets, instead every time they send a message it specifies the destination port or address. Datagram sockets are used with UDP (User Datagram Protocol) which is a connectionless protocol that provide best effort service but doesn’t guarantee whether datagrams will arrive at the destination or not. Datagram sockets also doesn’t have any retransmission mechanisms for lost datagrams. Also, these sockets do not ensure whether the datagrams will arrive in the same order they were sent. Datagram sockets send individual messages unlike stream sockets which send a continuous stream of data. Here, each message has a maximum size and each message is sent along with the specified destination port. Datagram sockets are mainly used for real-time applications which require low latency, examples include Broadcasting, Streaming media, VOIP (Voice-Over IP).

1. Write a server/client model socket program to exchange hello message between them.

**TCP server**

import socket

def start\_server():

# Create a TCP/IP socket

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

# Bind the socket to the address and port

server\_address = ('localhost', 10000)

print('Starting server on {} port {}'.format(\*server\_address))

server\_socket.bind(server\_address)

# Listen for incoming connections

server\_socket.listen(1)

while True:

# Wait for a connection

print('Waiting for a connection...')

connection, client\_address = server\_socket.accept()

try:

print('Connection from', client\_address)

# Receive the data in small chunks and retransmit it

while True:

data = connection.recv(16)

print('Received {!r}'.format(data))

if data:

print('Sending data back to the client...')

connection.sendall(data)

else:

print('No data from', client\_address)

break

finally:

# Clean up the connection

connection.close()

if \_\_name\_\_ == '\_\_main\_\_':

start\_server()

**TCP client**

import socket

def start\_client():

# 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

server\_address = ('localhost', 10000)

print('Connecting to {} port {}'.format(\*server\_address))

client\_socket.connect(server\_address)

try:

# Send data

message = 'Hello, Server!'

print('Sending {!r}'.format(message))

client\_socket.sendall(message.encode())

# Look for the response

amount\_received = 0

amount\_expected = len(message)

while amount\_received < amount\_expected:

data = client\_socket.recv(16)

amount\_received += len(data)

print('Received {!r}'.format(data.decode()))

finally:

# Clean up the connection

client\_socket.close()

if \_\_name\_\_ == '\_\_main\_\_':

start\_client()

1. Write a TCP server-client program to check if a given string is Palindrome

Input: level

Output: Palindrome

Input: Assessment

Output: Not a Palindrome

TCP server

import socket

def is\_palindrome(s):

return s == s[::-1]

def start\_server():

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

server\_address = ('localhost', 10000)

print('Server starting on {} port {}'.format(\*server\_address))

server\_socket.bind(server\_address)

server\_socket.listen(1)

while True:

print('Waiting for a connection...')

connection, client\_address = server\_socket.accept()

try:

print('Connection from', client\_address)

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

print('Received {!r}'.format(data))

if data:

result = "Palindrome" if is\_palindrome(data) else "Not a Palindrome"

print('Sending: ' + result)

connection.sendall(result.encode())

else:

print('No data from', client\_address)

break

finally:

connection.close()

if \_\_name\_\_ == '\_\_main\_\_':

start\_server()

TCP client

import socket

def start\_client():

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

server\_address = ('localhost', 10000)

client\_socket.connect(server\_address)

try:

message = input("Enter a string to check if it's a palindrome: ")

print('Sending {!r}'.format(message))

client\_socket.sendall(message.encode())

data = client\_socket.recv(1024)

print('Received {!r}'.format(data.decode()))

finally:

print('Closing socket')

client\_socket.close()

if \_\_name\_\_ == '\_\_main\_\_':

start\_client()

1. Write an example to demonstrate UDP server-client program

TCP server

import socket

def start\_server():

# Create a UDP socket

server\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM)

# Bind the socket to the server address and port

server\_address = ('localhost', 10000)

print('Starting up on {} port {}'.format(\*server\_address))

server\_socket.bind(server\_address)

while True:

print('\nWaiting to receive message...')

data, address = server\_socket.recvfrom(4096)

print('Received {} bytes from {}'.format(len(data), address))

print(data.decode())

if data:

sent = server\_socket.sendto(data, address)

print('Sent {} bytes back to {}'.format(sent, address))

if \_\_name\_\_ == '\_\_main\_\_':

start\_server()

TCP client

import socket

def start\_client():

# Create a UDP socket

client\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM)

server\_address = ('localhost', 10000)

try:

# Send data

message = 'This is the message. It will be repeated.'

print('Sending {!r}'.format(message))

sent = client\_socket.sendto(message.encode(), server\_address)

# Receive response

print('Waiting to receive...')

data, server = client\_socket.recvfrom(4096)

print('Received {!r}'.format(data.decode()))

finally:

print('Closing socket')

client\_socket.close()

if \_\_name\_\_ == '\_\_main\_\_':

start\_client()