CS-356 Computer Networks Lab
Lab Session-4

Socket Programming in C++

What is a Socket?

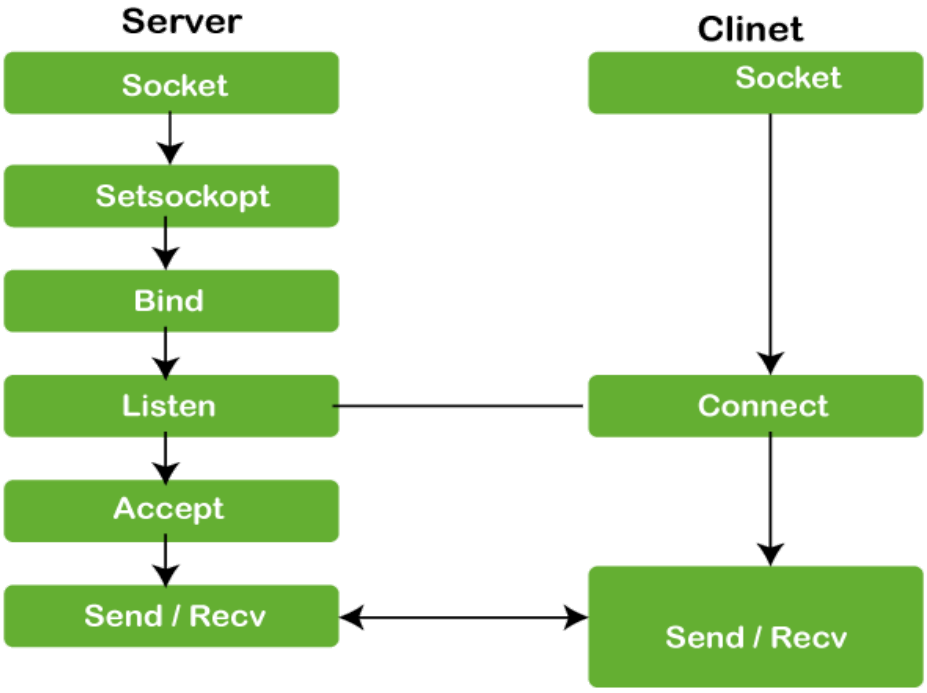
A socket is a software endpoint that facilitates communication between two programs over a network. It provides a standard mechanism for processes on different devices to communicate with each other. In networking terms, a socket consists of an IP address and a port number, and it enables data exchange between a client and a server.

There are two types of sockets: Stream Sockets (TCP) and Datagram Sockets (UDP).

Stream Sockets (TCP): These provide a reliable, connection-oriented communication. Data is sent as a stream of bytes, and it ensures that the data is delivered in the correct order without loss.

Datagram Sockets (UDP): These provide a connectionless, unreliable communication. Data is sent as individual packets (datagrams), and there is no guarantee of delivery or order.

Let's look at a simple program in C++ to understand the workings of Sockets.



Procedure of Client-Server Communication

There are some procedures that we have to follow to establish client-server communication. These are as follows.

1. **Socket:** With the help of a socket, we can create a new communication.
2. **Bind:** With the help of this we can, we can attach the local address with the socket.
3. **Listen:** With this help; we can accept the connection.
4. **Accept:** With this help; we can block the incoming connection until the request arrives.
5. **Connect:** With this help; we can attempt to establish the connection.
6. **Send:** With the help of this; we can send the data over the network.
7. **Receive:** With this help; we can receive the data over the network.
8. **Close:** With the help of this, we can release the connection from the network.

At Server:

```
#include <iostream>
#include <cstring>
#include <arpa/inet.h>
#include <unistd.h>

int main() {
    // Create a socket
    int serverSocket = socket(AF_INET, SOCK_STREAM, 0);
    if (serverSocket == -1) {
        std::cerr << "Error creating socket\n";
        return -1;
    }

    // Bind the socket to an IP address and port
    sockaddr_in serverAddress;
    serverAddress.sin_family = AF_INET;
    serverAddress.sin_addr.s_addr = INADDR_ANY; // Listen on any available
interface
    serverAddress.sin_port = htons(8080); // Use port 8080

    if (bind(serverSocket, (struct sockaddr*)&serverAddress,
sizeof(serverAddress)) == -1) {
        std::cerr << "Error binding socket\n";
        close(serverSocket);
        return -1;
    }

    // Listen for incoming connections
    if (listen(serverSocket, 5) == -1) {
        std::cerr << "Error listening for connections\n";
```

```
        close(serverSocket);
        return -1;
    }

    // Accept a connection
    int clientSocket = accept(serverSocket, nullptr, nullptr);
    if (clientSocket == -1) {
        std::cerr << "Error accepting connection\n";
        close(serverSocket);
        return -1;
    }

    // Receive data from the client
    char buffer[1024];
    ssize_t bytesRead = recv(clientSocket, buffer, sizeof(buffer), 0);
    if (bytesRead == -1) {
        std::cerr << "Error receiving data\n";
        close(serverSocket);
        close(clientSocket);
        return -1;
    }

    // Display the received message
    std::cout << "Received message from client: " << buffer << "\n";

    // Close sockets
    close(serverSocket);
    close(clientSocket);

    return 0;
}
```

At Client:

```
#include <iostream>
#include <cstring>
#include <arpa/inet.h>
#include <unistd.h>

int main() {
    // Create a socket
    int clientSocket = socket(AF_INET, SOCK_STREAM, 0);
    if (clientSocket == -1) {
        std::cerr << "Error creating socket\n";
        return -1;
    }
```

```
    // Connect to the server
    sockaddr_in serverAddress;
    serverAddress.sin_family = AF_INET;
    serverAddress.sin_addr.s_addr = inet_addr("127.0.0.1"); // Server's IP
address
    serverAddress.sin_port = htons(8080); // Server's port

    if (connect(clientSocket, (struct sockaddr*)&serverAddress,
sizeof(serverAddress)) == -1) {
        std::cerr << "Error connecting to server\n";
        close(clientSocket);
        return -1;
    }

    // Send data to the server
    const char* message = "Hello Message";
    if (send(clientSocket, message, strlen(message), 0) == -1) {
        std::cerr << "Error sending data\n";
        close(clientSocket);
        return -1;
    }

    // Close socket
    close(clientSocket);

    return 0;
}
```

Client.cpp

```cpp
#include <iostream>
#include <cstring>
#include <arpa/inet.h>
#include <unistd.h>

int main() {
    // Create a socket
    int clientSocket = socket(AF_INET, SOCK_STREAM, 0);
    if (clientSocket == -1) {
        std::cerr << "Error creating socket\n";
        return -1;
    }

    // Connect to the server
    sockaddr_in serverAddress;
    serverAddress.sin_family = AF_INET;
    serverAddress.sin_addr.s_addr = inet_addr("127.0.0.1"); // Server's IP address
    serverAddress.sin_port = htons(8080); // Server's port

    if (connect(clientSocket, (struct sockaddr*)&serverAddress, sizeof(serverAddress)) == -1) {
        std::cerr << "Error connecting to server\n";
        close(clientSocket);
        return -1;
    }

    // Send data to the server
    const char* message = "Hello Message";
    if (send(clientSocket, message, strlen(message), 0) == -1) {
        std::cerr << "Error sending data\n";
        close(clientSocket);
        return -1;
    }

    // Close socket
    close(clientSocket);

    return 0;
}
```

Server.cpp

```cpp
#include <cstring>
#include <arpa/inet.h>
#include <unistd.h>

int main() {
    // Create a socket
    int serverSocket = socket(AF_INET, SOCK_STREAM, 0);
    if (serverSocket == -1) {
        std::cerr << "Error creating socket\n";
        return -1;
    }

    // Bind the socket to an IP address and port
    sockaddr_in serverAddress;
    serverAddress.sin_family = AF_INET;
    serverAddress.sin_addr.s_addr = INADDR_ANY; // Listen on any available interface
    serverAddress.sin_port = htons(8080); // Use port 8080

    if (bind(serverSocket, (struct sockaddr*)&serverAddress, sizeof(serverAddress)) == -1) {
        std::cerr << "Error binding socket\n";
        close(serverSocket);
        return -1;
    }

    // Listen for incoming connections
    if (listen(serverSocket, 5) == -1) {
        std::cerr << "Error listening for connections\n";
        close(serverSocket);
        return -1;
    }

    // Accept a connection
    int clientSocket = accept(serverSocket, nullptr, nullptr);
    if (clientSocket == -1) {
        std::cerr << "Error accepting connection\n";
        close(serverSocket);
        return -1;
    }

    // Receive data from the client
    char buffer[1024];
    ssize_t bytesRead = recv(clientSocket, buffer, sizeof(buffer), 0);
    if (bytesRead == -1) {
        std::cerr << "Error receiving data\n";
        close(serverSocket);
        close(clientSocket);
        return -1;
    }

    // Display the received message
    std::cout << "Received message from client: " << buffer << "\n";

```

```
cse@cse-OptiPlex-7000:~$ cd Documents/
cse@cse-OptiPlex-7000:~/Documents$ g++ Server.cpp
cse@cse-OptiPlex-7000:~/Documents$ ./a.out
Received message from client: Hello Message
cse@cse-OptiPlex-7000:~/Documents$
```

```
cse@cse-OptiPlex-7000:~$ cd Documents/
cse@cse-OptiPlex-7000:~/Documents$ g++ Client.cpp
cse@cse-OptiPlex-7000:~/Documents$ ./a.out
cse@cse-OptiPlex-7000:~/Documents$
```

Explanation of key functions and arguments:

1. **socket():** Creates a new socket with a specified domain (AF_INET for IPv4), type (SOCK_STREAM for TCP), and protocol (0 for default).

2. **bind():** Associates a socket with a specific IP address and port.

3. **listen():** Puts the server socket in a state where it's listening for incoming connections.

4. **accept()**: Accepts an incoming connection, creating a new socket for communication with the client.

5. **connect()**: Establishes a connection to a server.

6. **send():** Sends data over the established connection.

7. **recv()**: Receives data from the connected socket.

8. **close()**: Closes a socket.

In this example, the server listens on port 8080, and the client connects to the server's IP address on that port. The client sends the "Hello Message," and the server receives and prints it.

Let's go through the purpose of the specific header files and the functions used in the C++ codes for both the server and the client.

Common Header Files:

unistd (#include <unistd.h>):

Purpose: POSIX operating system API.
Usage: Used for system calls such as `close()` to close sockets.

arpa/inet (#include <arpa/inet.h>):

Purpose: Definitions for internet operations.
Usage: Used for data types like sockaddr_in and functions related to IP address manipulation, such as `inet_addr()`.

Server Specific Header Files:

sys/socket (#include <sys/socket.h>):

Purpose: Socket-related system calls.
Usage: Used for creating sockets (`socket()`), binding sockets to addresses (`bind()`), listening for incoming connections (`listen()`), and accepting connections (`accept()`).

Client Specific Header Files:

netinet/in (#include <netinet/in.h>):

Purpose: Internet address family and structure definitions.
Usage: Used for defining the sockaddr_in structure, which is used for specifying IP addresses and ports.

Important Functions:

1. socket() Function:

- Purpose: Creates a new socket.
- Arguments:
  - `int domain`: Specifies the communication domain (e.g., `AF_INET` for IPv4).
  - `int type`: Specifies the socket type (e.g., `SOCK_STREAM` for TCP).
  - `int protocol`: Specifies the protocol (usually set to 0 for default).
- Return Value:
  - On success, returns a file descriptor for the new socket.
  - On failure, returns -1.
- Reasons for Choices in Code:
  - In both server and client codes: `socket(AF_INET, SOCK_STREAM, 0)`.
  - This creates a new IPv4 socket using TCP. It's a common choice for simple client-server communication.

2. bind() Function:

- Purpose: Associates a socket with a specific IP address and port.
- Arguments:
  - `int sockfd`: The socket file descriptor.
  - `const struct sockaddr *addr`: A pointer to a structure containing the address information.
  - `socklen_t addrlen`: The size of the address structure.
- Return Value:
  - On success, returns 0.
  - On failure, returns -1.
- Reasons for Choices in Code:
  - In the server code: `bind(serverSocket, (struct sockaddr*)&serverAddress, sizeof(serverAddress))`.

- Associates the server socket with the specified IP address (`INADDR_ANY` for any available interface) and port (8080).

3. listen() Function:

- Purpose: Puts the server socket in a state where it's listening for incoming connections.
- Arguments:
  - `int sockfd`: The socket file descriptor.
  - `int backlog`: The maximum number of pending connections that can be queued up.
- Return Value:
  - On success, returns 0.
  - On failure, returns -1.
- Reasons for Choices in Code:
  - In the server code: `listen(serverSocket, 5).`
  - Sets the server socket to listen for incoming connections with a maximum backlog of 5 pending connections.

4. accept() Function:

- Purpose: Accepts an incoming connection, creating a new socket for communication with the client.
- Arguments:
  - `int sockfd`: The socket file descriptor.
  - `struct sockaddr *addr`: A pointer to a structure that will store the address of the connecting client.
  - `socklen_t *addrlen`: A pointer to the size of the address structure.
- Return Value:
  - On success, returns a new file descriptor for the accepted socket.
  - On failure, returns -1.
- Reasons for Choices in Code:
  - In the server code: `accept(serverSocket, nullptr, nullptr).`
  - Accepts an incoming connection, and the new socket is used for communication with the connected client.

5. connect() Function:

- Purpose: Establishes a connection to a server.
- Arguments:
  - `int sockfd`: The socket file descriptor.
  - `const struct sockaddr *addr`: A pointer to the server's address structure.
  - `socklen_t addrlen`: The size of the server's address structure.
- Return Value:

        - On success, returns 0.
        - On failure, returns -1.
    - Reasons for Choices in Code:
        - In the client code: `connect(clientSocket, (struct sockaddr*)&serverAddress, sizeof(serverAddress))`.
        - Connects the client to the specified server's IP address (`127.0.0.1`) and port (`8080`).

6. send() Function:

- Purpose: Sends data over a socket.
- Arguments:
    - `int sockfd`: The socket file descriptor.
    - `const void *buf`: A pointer to the data to be sent.
    - `size_t len`: The size of the data.
    - `int flags`: Optional flags.
- Return Value:
    - On success, returns the number of bytes sent.
    - On failure, returns -1.
- Reasons for Choices in Code:
    - In the client code: `send(clientSocket, message, strlen(message), 0)`.
    - Sends the "Hello Message" to the server.

7. recv() Function:

- Purpose: Receives data from a socket.
- Arguments:
    - `int sockfd`: The socket file descriptor.
    - `void *buf`: A pointer to the buffer where the received data will be stored.
    - `size_t len`: The size of the buffer.
    - `int flags`: Optional flags.
- Return Value:
    - On success, returns the number of bytes received.
    - On failure, returns -1.
- Reasons for Choices in Code:
    - In the server code: `recv(clientSocket, buffer, sizeof(buffer), 0)`.
    - Receives data from the connected client and stores it in the `buffer`.

8. close() Function:

- Purpose: Closes a socket.
- Arguments:
    - `int sockfd`: The socket file descriptor.

- Return Value:
  - On success, returns 0.
  - On failure, returns -1.
- Reasons for Choices in Code:
  - In both server and client codes: `close(serverSocket)` and `close(clientSocket)`.
  - Closes the server and client sockets when they are no longer needed.

Understand the below code for the UDP socket:

Server:

```
#include <iostream>
#include <arpa/inet.h>
#include <unistd.h>

int main() {
    // Create a UDP socket
    int serverSocket = socket(AF_INET, SOCK_DGRAM, 0);
    if (serverSocket == -1) {
        std::cerr << "Error creating socket\n";
        return -1;
    }

    // Bind the socket to an address and port
    sockaddr_in serverAddress;
    serverAddress.sin_family = AF_INET;
    serverAddress.sin_addr.s_addr = INADDR_ANY; // Listen on any available
interface
    serverAddress.sin_port = htons(8080); // Use port 8080

    if (bind(serverSocket, (struct sockaddr*)&serverAddress,
sizeof(serverAddress)) == -1) {
        std::cerr << "Error binding socket\n";
        close(serverSocket);
        return -1;
    }
```

```
    // Receive data from the client
    char buffer[1024];
    sockaddr_in clientAddress;
    socklen_t clientAddrLen = sizeof(clientAddress);

    ssize_t bytesRead = recvfrom(serverSocket, buffer, sizeof(buffer), 0,
                                 (struct sockaddr*)&clientAddress,
&clientAddrLen);

    if (bytesRead == -1) {
        std::cerr << "Error receiving data\n";
        close(serverSocket);
        return -1;
    }

    // Display the received message
    std::cout << "Received message from client: " << buffer << "\n";

    // Close socket
    close(serverSocket);

    return 0;
}
```

Client:

```
#include <iostream>
#include <arpa/inet.h>
#include <unistd.h>

int main() {
    // Create a UDP socket
    int clientSocket = socket(AF_INET, SOCK_DGRAM, 0);
    if (clientSocket == -1) {
        std::cerr << "Error creating socket\n";
```

```
        return -1;
    }

    // Set up the server address
    sockaddr_in serverAddress;
    serverAddress.sin_family = AF_INET;
    serverAddress.sin_addr.s_addr = inet_addr("127.0.0.1"); // Server's IP
address
    serverAddress.sin_port = htons(8080); // Server's port

    // Send data to the server
    const char* message = "Hello UDP Server";
    ssize_t bytesSent = sendto(clientSocket, message, strlen(message), 0,
                               (struct sockaddr*)&serverAddress,
sizeof(serverAddress));

    if (bytesSent == -1) {
        std::cerr << "Error sending data\n";
        close(clientSocket);
        return -1;
    }

    // Close socket
    close(clientSocket);

    return 0;
}
```

Explanation:

1. The server uses `SOCK_DGRAM` when creating the socket to indicate that it is a UDP socket.
2. Both server and client use `sendto` and `recvfrom` functions for sending and receiving data in a connectionless manner. They include the destination/originating address information in each packet.

3. The server binds to a specific port using `bind` just like in the TCP example.
4. The client sends a message to the server using `sendto`.
5. The server receives the message along with the client's address using `recvfrom`.
6. Error handling in this example is minimal for clarity, and in a production environment, you should handle errors more gracefully and implement proper resource management.

Assignment:

In this programming assignment, you will write a client ping program. Your client will send a simple ping message to a server, receive a corresponding pong message back from the server, and determine the delay between when the client sent the ping message and received the pong message. This delay is called the Round Trip Time (RTT). The functionality provided by the client and server is similar to that provided by the standard ping program available in modern operating systems. However, standard ping programs use the Internet Control Message Protocol (ICMP). But we won't use ICMP; instead, we will create a nonstandard (but simple!) UDP-based ping program.
Your ping program is to send 10 ping messages to the target server over UDP. For each message, your client is to determine and print the RTT when the corresponding pong message is returned. Because UDP is an unreliable protocol, a packet sent by the client or server may be lost. For this reason, the client cannot wait indefinitely for a reply to a ping message. You should have the client wait up to one second for a reply from the server; if no reply is received, the client should assume that the packet was lost and print a message accordingly.