Yusuf Selaboy

VUG/CSC/21/5772

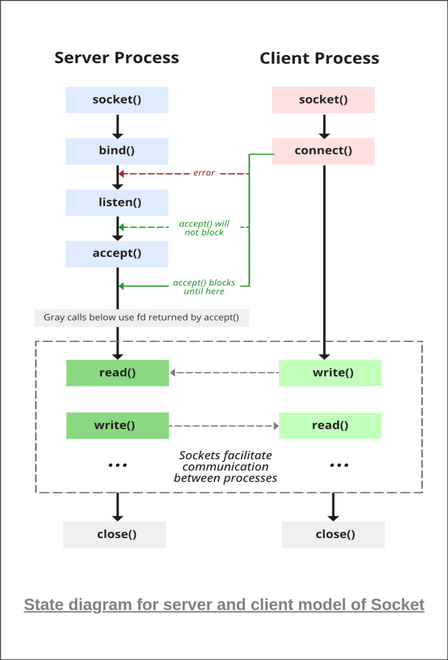
Cyber security

27th April, 2024.

1. **Socket Programming in C**

Socket programming is a way of connecting two nodes on a network to communicate with each other. One socket (node) listens on a particular port at an IP, while the other socket reaches out to the other to form a connection. The server forms the listener socket while the client reaches out to the server.

**State Diagram for Server and Client Model**

****

**Stages for Server**

**1. Socket Creation**

int sockfd = socket(domain, type, protocol)

sockfd: socket descriptor, an integer (like a file handle)

domain: integer, specifies communication domain. We use AF\_ LOCAL as defined in the POSIX standard for communication between processes on the same host. For communicating between processes on different hosts connected by IPV4, we use AF\_INET and AF\_I NET 6 for processes connected by IPV6.

type: communication type

SOCK\_STREAM: TCP(reliable, connection-oriented)

SOCK\_DGRAM: UDP(unreliable, connectionless)

protocol: Protocol value for Internet Protocol(IP), which is 0. This is the same number that appears on the protocol field in the IP header of a packet.(man protocols for more details)

**2. Setsockopt**

This helps in manipulating options for the socket referred by the file descriptor sockfd. This is completely optional, but it helps in reuse of address and port. Prevents error such as: “address already in use”.

int setsockopt(int sockfd, int level, int optname, const void \*optval, socklen\_t optlen);

**3. Bind**

int bind(int sockfd, const struct sockaddr \*addr, socklen\_t addrlen);

After the creation of the socket, the bind function binds the socket to the address and port number specified in addr(custom data structure). In the example code, we bind the server to the localhost, hence we use INADDR\_ANY to specify the IP address.

**4. Listen**

int listen(int sockfd, int backlog);

It puts the server socket in a passive mode, where it waits for the client to approach the server to make a connection. The backlog, defines the maximum length to which the queue of pending connections for sockfd may grow. If a connection request arrives when the queue is full, the client may receive an error with an indication of ECONNREFUSED.

**5. Accept**

int new\_socket= accept(int sockfd, struct sockaddr \*addr, socklen\_t \*addrlen);

It extracts the first connection request on the queue of pending connections for the listening socket, sockfd, creates a new connected socket, and returns a new file descriptor referring to that socket. At this point, the connection is established between client and server, and they are ready to transfer data.

**Stages for Client**

1. Socket connection: Exactly the same as that of server’s socket creation

2. Connect: The connect() system call connects the socket referred to by the file descriptor sockfd to the address specified by addr. Server’s address and port is specified in addr.

int connect(int sockfd, const struct sockaddr \*addr, socklen\_t addrlen);

Implementation

Here we are exchanging one hello message between server and client to demonstrate the client/server model.

**C Program to Create Server**

Server.c

// Server side C program to demonstrate Socket

// programming

#include <netinet/in.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <sys/socket.h>

#include <unistd.h>

#define PORT 8080

int main(int argc, char const\* argv[])

{

int server\_fd, new\_socket;

ssize\_t valread;

struct sockaddr\_in address;

int opt = 1;

socklen\_t addrlen = sizeof(address);

char buffer[1024] = { 0 };

char\* hello = "Hello from server";

// Creating socket file descriptor

if ((server\_fd = socket(AF\_INET, SOCK\_STREAM, 0)) < 0) {

perror("socket failed");

exit(EXIT\_FAILURE);

}

// Forcefully attaching socket to the port 8080

if (setsockopt(server\_fd, SOL\_SOCKET,

SO\_REUSEADDR | SO\_REUSEPORT, &opt,

sizeof(opt))) {

perror("setsockopt");

exit(EXIT\_FAILURE);

}

address.sin\_family = AF\_INET;

address.sin\_addr.s\_addr = INADDR\_ANY;

address.sin\_port = htons(PORT);

// Forcefully attaching socket to the port 8080

if (bind(server\_fd, (struct sockaddr\*)&address,

sizeof(address))

< 0) {

perror("bind failed");

exit(EXIT\_FAILURE);

}

if (listen(server\_fd, 3) < 0) {

perror("listen");

exit(EXIT\_FAILURE);

}

if ((new\_socket

= accept(server\_fd, (struct sockaddr\*)&address,

&addrlen))

< 0) {

perror("accept");

exit(EXIT\_FAILURE);

}

valread = read(new\_socket, buffer,

1024 - 1); // subtract 1 for the null

// terminator at the end

printf("%s\n", buffer);

send(new\_socket, hello, strlen(hello), 0);

printf("Hello message sent\n");

// closing the connected socket

close(new\_socket);

// closing the listening socket

close(server\_fd);

return 0;

}

**C Program to Create Client**

client.c

// Client side C program to demonstrate Socket

// programming

#include <arpa/inet.h>

#include <stdio.h>

#include <string.h>

#include <sys/socket.h>

#include <unistd.h>

#define PORT 8080

int main(int argc, char const\* argv[])

{

int status, valread, client\_fd;

struct sockaddr\_in serv\_addr;

char\* hello = "Hello from client";

char buffer[1024] = { 0 };

if ((client\_fd = socket(AF\_INET, SOCK\_STREAM, 0)) < 0) {

printf("\n Socket creation error \n");

return -1;

}

serv\_addr.sin\_family = AF\_INET;

serv\_addr.sin\_port = htons(PORT);

// Convert IPv4 and IPv6 addresses from text to binary

// form

if (inet\_pton(AF\_INET, "127.0.0.1", &serv\_addr.sin\_addr)

<= 0) {

printf(

"\nInvalid address/ Address not supported \n");

return -1;

}

if ((status

= connect(client\_fd, (struct sockaddr\*)&serv\_addr,

sizeof(serv\_addr)))

< 0) {

printf("\nConnection Failed \n");

return -1;

}

send(client\_fd, hello, strlen(hello), 0);

printf("Hello message sent\n");

valread = read(client\_fd, buffer,

1024 - 1); // subtract 1 for the null

// terminator at the end

printf("%s\n", buffer);

// closing the connected socket

close(client\_fd);

return 0;

}

**Output**

Client:Hello message sent

Hello from server

Server:Hello from client

Hello message sent

**Compiling**

gcc client.c -o client

gcc server.c -o server

1. **15 Common Ports**

Port number is a 16-bit numerical value that ranges from 0 to 65535. Well-known port (0-1023), registered port (1024-49151), and dynamic port is three types of port number space. (49152-65535).

These ports can be opened and used by software applications and operating system services to send and receive data over networks (LAN or WAN) that employ certain protocols (eg TCP, UDP).

For example, we use 80 for HTTP-web-based plain-text surfing and 443 for HTTPS-web-based encrypted websites in our daily work.

**Functions of ports**

When interacting over the Internet, TCP and UDP protocols make connections, recompile data packages after the transfer, and then deliver them to applications on the recipient’s device. For this handover to work, the operating system must install and open the gateway for the transfer. Each door has a unique code number. After transmission, the receiving system uses the port number to determine where the data should be sent. The port numbers of the sender and receiver are always included in the data packet.

**15 Significant Ports**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Serial Number** | **Port Number** | **Service name** | **Transport protocol** | **Description** | |
|  | 7 | Echo | TCP, UDP | Echo service | |
|  | 20 | FTP-data | TCP, SCTP | File Transfer Protocol data transfer | |
|  | 21 | FTP | TCP, UDP, SCTP | File Transfer Protocol (FTP) control connection | |
|  | 22 | SSH-SCP | TCP, UDP, SCTP | Secure Shell, secure logins, file transfers (scp, sftp), and port forwarding | |
|  | 23 | Telnet | TCP | Telnet protocol—unencrypted text communications | |
|  | 25 | SMTP | TCP | Simple Mail Transfer Protocol, used for email routing between mail servers | |
|  | 53 | DNS | TCP, UDP | Domain Name System name resolver | |
|  | 69 | TFTP | UDP | Trivial File Transfer Protocol | |
|  | 80 | HTTP | TCP, UDP, SCTP | Hypertext Transfer Protocol (HTTP) uses TCP in versions 1.x and 2. | HTTP/3 uses QUIC, a transport protocol on top of UDP |
|  | 88 | Kerberos | TCP, UDP | Network authentication system | |
|  | 102 | Iso-tsap | TCP | ISO Transport Service Access Point (TSAP) Class 0 protocol | |
|  | 110 | POP3 | TCP | Post Office Protocol, version 3 (POP3) | |
|  | 135 | Microsoft EPMAP | TCP, UDP | Microsoft EPMAP (End Point Mapper), also known as DCE/RPC Locator service, used to remotely manage services including DHCP server, DNS server, and WINS. Also used by DCOM | |
|  | 137 | NetBIOS-ns | TCP, UDP | NetBIOS Name Service, used for name registration and resolution | |
|  | 139 | NetBIOS-ssn | TCP, UDP | NetBIOS Session Service | |

1. **What is an Open Port?**

In cyber security, the term open port refers to a TCP or UDP port number that is configured to accept packets. In contrast, a port that rejects connections or ignores all packets is a closed port.

Ports are an integral part of the Internet's communication model. All communication over the Internet is exchanged via ports.

**5 Free Open Port Check Tools**

There are free tools available that can help you identify whether your sensitive resources are exposed through open ports.

1**. Nmap**

Nmap (short for Network Mapper) is one of the most popular free open-source port scanning tools available. It offers many different port scanning techniques including TCP half-open scans.

2. **Wireshark**

Wireshark is a free network sniffing tool that's used to detect malicious activity in network traffic. This tool can also be used to detect open ports.

3. **Angry IP Scanner**

Angry IP scanner is a free network scanner offering a suite of network monitoring tools.

4. **NetCat**

NetCat is a free port scanning tool that uses the TCP/IP protocol across different connections.

5. **Advanced IP Scanner**

Advanced IP scanner is a windows solution that can analyze IP addresses and ports.

**How Can I Monitor My Open Ports?**

On a small network with relatively few IP addresses, finding and closing open ports isn't a massive task. However, as you likely know, on larger networks with a content flow of new devices, monitoring and managing open ports can be extremely time-consuming. In addition to the ports themselves, the underlying services using those ports need to be monitored too.

The good news is that these open ports and services are facing the public Internet, so they can be scanned by continuous monitoring technology like UpGuard's attack surface management platform. The UpGuard platform explicitly checks for nearly 200 services running across thousands of ports, and reports on any services we can't identify, as well as any open ports with no services detected.

1. **OSI model**

The Open Systems Interconnection (OSI) model is a reference model from the International Organization for Standardization (ISO) that "provides a common basis for the coordination of standards development for the purpose of systems interconnection." In the OSI reference model, the communications between systems are split into seven different abstraction layers: Physical, Data Link, Network, Transport, Session, Presentation, and Application.

The model partitions the flow of data in a communication system into seven abstraction layers to describe networked communication from the physical implementation of transmitting bits across a communications medium to the highest-level representation of data of a distributed application. Each intermediate layer serves a class of functionality to the layer above it and is served by the layer below it. Classes of functionality are realized in all software development through all standardized communication protocols.

**Layer 1: Physical layer**

The physical layer is responsible for the transmission and reception of unstructured raw data between a device, such as a network interface controller, Ethernet hub, or network switch, and a physical transmission medium. It converts the digital bits into electrical, radio, or optical signals. Layer specifications define characteristics such as voltage levels, the timing of voltage changes, physical data rates, maximum transmission distances, modulation scheme, channel access method and physical connectors. This includes the layout of pins, voltages, line impedance, cable specifications, signal timing and frequency for wireless devices. Bit rate control is done at the physical layer and may define transmission mode as simplex, half duplex, and full duplex. The components of a physical layer can be described in terms of the network topology. Physical layer specifications are included in the specifications for the ubiquitous Bluetooth, Ethernet, and USB standards. An example of a less well-known physical layer specification would be for the CAN standard.

**Layer 2: Data link layer**

The data link layer provides node-to-node data transfer—a link between two directly connected nodes. It detects and possibly corrects errors that may occur in the physical layer. It defines the protocol to establish and terminate a connection between two physically connected devices. It also defines the protocol for flow control between them.

**Layer 3: Network layer**

The network layer provides the functional and procedural means of transferring packets from one node to another connected in "different networks". A network is a medium to which many nodes can be connected, on which every node has an address and which permits nodes connected to it to transfer messages to other nodes connected to it by merely providing the content of a message and the address of the destination node and letting the network find the way to deliver the message to the destination node, possibly routing it through intermediate nodes. If the message is too large to be transmitted from one node to another on the data link layer between those nodes, the network may implement message delivery by splitting the message into several fragments at one node, sending the fragments independently, and reassembling the fragments at another node. It may, but does not need to, report delivery errors.

**Layer 4: Transport layer**

The transport layer provides the functional and procedural means of transferring variable-length data sequences from a source host to a destination host from one application to another across a network, while maintaining the quality-of-service functions. Transport protocols may be connection-oriented or connectionless.

**Layer 5: Session layer**

The session layer creates the setup, controls the connections, and ends the teardown, between two or more computers, which is called a "session". Common functions of the session layer include user logon (establishment) and user logoff (termination) functions. Including this matter, authentication methods are also built into most client software, such as FTP Client and NFS Client for Microsoft Networks. Therefore, the session layer establishes, manages and terminates the connections between the local and remote application. The session layer also provides for full-duplex, half-duplex, or simplex operation[citation needed], and establishes procedures for check pointing, suspending, restarting, and terminating a session between two related streams of data, such as an audio and a video stream in a web-conferencing application. Therefore, the session layer is commonly implemented explicitly in application environments that use remote procedure calls.

**Layer 6: Presentation layer**

The presentation layer establishes data formatting and data translation into a format specified by the application layer during the encapsulation of outgoing messages while being passed down the protocol stack, and possibly reversed during the encapsulation of incoming messages when being passed up the protocol stack. For this very reason, outgoing messages during encapsulation are converted into a format specified by the application layer, while the conversions for incoming messages during encapsulation are reversed.

**Layer 7: Application layer**

The application layer is the layer of the OSI model that is closest to the end user, which means both the OSI application layer and the user interact directly with a software application that implements a component of communication between the client and server, such as File Explorer and Microsoft Word. Such application programs fall outside the scope of the OSI model unless they are directly integrated into the application layer through the functions of communication, as is the case with applications such as web browsers and email programs. Other examples of software are Microsoft Network Software for File and Printer Sharing and Unix/Linux Network File System Client for access to shared file resources.