COE 768
Lab 2: Servers running on TCP

An Internet server at application layer uses either TCP or UDP as a transport protocol. In this lab, we will study servers operating on TCP.

I. TCP Connection Mechanism

In this section, we will use the Echo service to study the TCP connection establishment and termination mechanism.

1. Enable Wireshark in VM1 and start the capture process.

2. Start an "Echo" server in VM1:

   $./echo_server [port_number]

3. Start an "Echo" client on VM2:

   $./echo_client [server_IP_address] [port_number]

   From the client, send one or two messages.

4. TCP is a connection-oriented protocol - a logical connection must be established between client and server before data transfer (Figure 1). With this understanding, find out from the captured data in Wireshake the number of TCP packets exchanged to setup a TCP connection.

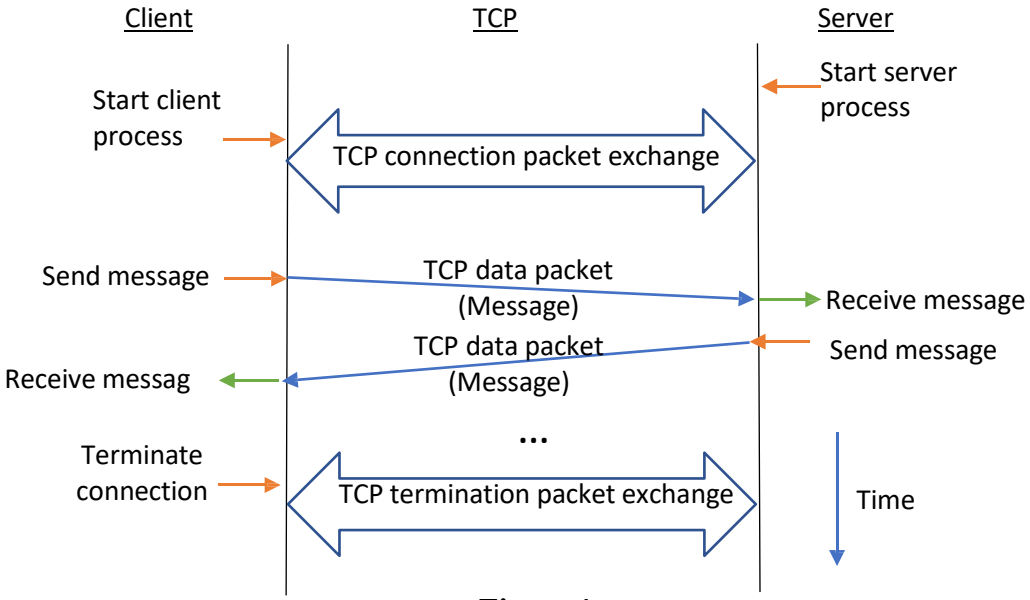

Figure 1

5. Now terminate the client process (but keep the server process running). The termination of the client also causes the termination of the TCP connection (Figure 1). Find out from the captured data in Wireshark the number of packets exchanged for the termination process.

II. Concurrent Server

A concurrent server is a server that can provide service to multiple clients concurrently. In this section, we will study how the concurrent feature of the TCP concurrent server is implemented.

6. Open another terminal on VM1, type

   $./ps -a

   The output should show that one echo_server process is running on the VM.

7. Start an "Echo" client on VM2:

   $./echo_client [server_IP_address] [port_number]

8. Repeat step 6 again. You will find out there are two echo server processes running. The second server process is a child process created by the original concurrent server. The child process is the one that has a TCP connection with and provides service to the client.

9. Open another terminal in VM2 and start a second client. You will now find that there are 3 server processes running in VM1, one is the original server while the other two are child processes. Each child process provides service to one client. Figure 2 below illustrates the concept.

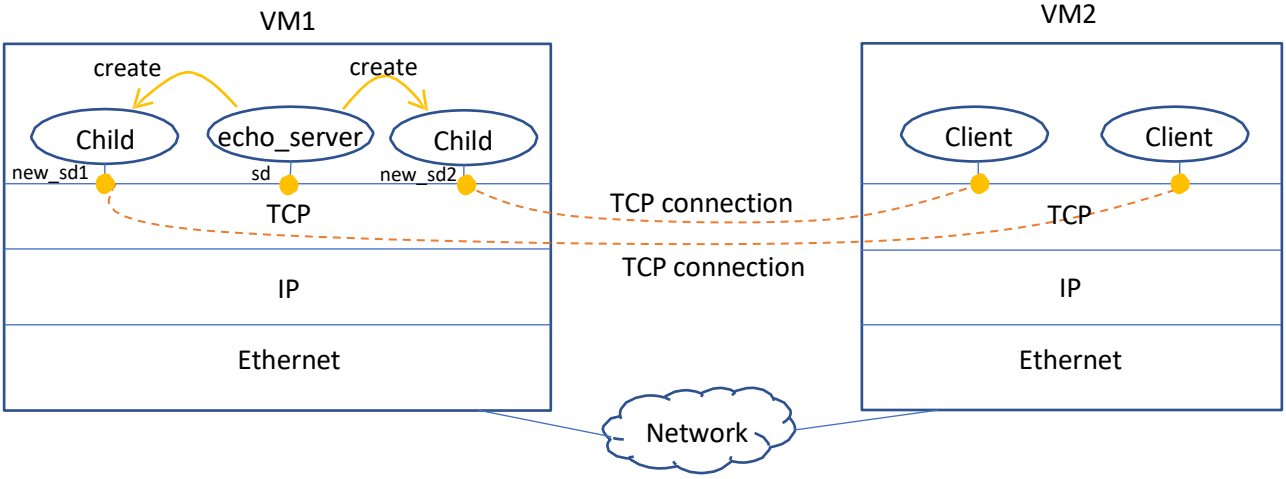

Figure 2

10. You can also verify that there are two TCP connections by using netstat utility on either VM1 or VM2:

    $./netstat -t

III. Socket Programming

In this part of the lab, we will study the source files of echo_server.c and echo_client.c to gain some knowledge on how to write network applications based on TCP socket programming. By the end of this part of the lab, you should be able to write a simple network application.

Server Program

The server makes the following sequence of system calls to make itself available for TCP connection:

```
sd = socket(AF_INET, SOCK_STREAM, 0);
bind(sd, (struct sockaddr *)&server, sizeof(server));
listen(sd, 5);
new_sd = accept(sd, (struct sockaddr *)&client, &client_len);
```

The server makes a *socket* system call to create a socket through which it can access the TCP/IP service. The call returns a descriptor (sd) whose function is similar to a file descriptor. The subsequent socket system calls through that socket will refer to this descriptor. The argument "SOCK_STREAM" specifies that the server will use TCP as the transport-layer service. The server then calls *bind* to assign IP address and port number to the just opened socket. Next, the server calls *listen* to place a socket in passive (server) mode and tells the TCP module to enqueue TCP connection requests with maximum queue size of 5. Finally, the server calls *accept* to accept an incoming connection request. The system call *accept* blocks the return until a TCP connection request, which is issued by the client, arrives. When a TCP connection request arrives, the call will return with a new socket descriptor, which can be used by the child process to communicate with the client (see Figure 2).

The return of the *accept* system call indicates that a TCP connection has just been established. Subsequently, the server executes the following code:

```
switch (fork()){
        case 0:         /* child */
                (void) close(sd);
                exit(echod(new_sd));
        default:                /* parent */
                (void) close(new_sd);
                break;
        case -1:
                fprintf(stderr, "fork: error\n");
}
```

The *fork* system call creates a child process which has the same code as the server process. Both processes will continue execute at the point where the *fork* returns. The *fork* system call returns a positive integer (the original server process ID) to the original process and 0 to the child process. The original process closes the new socket and goes back to listen to the original socket for new client connection request. The child process closes the original socket and uses the new socket to communicate with the client by executing the function *echod*.

```
int echod(int sd)
{
        char    *bp, buf[BUFLEN];
        int     n, bytes_to_read;

        while(n = read(sd, buf, BUFLEN))
                write(sd, buf, n);
        close(sd);

        return(0);
}
```

Inside *echod*, the server calls *read* to wait for the message from the client. The *read* system call blocks the return until the server receives a message from the client. When *read* returns, the client message will be in the character buffer (buf) and the return value is the length of the message in bytes. The server just resends the message back to the client using the *write* system call. Note that the application does not need to care how the message is sent. The protocols at the lower layers, such as TCP, IP and Ethernet, will take care of it. When the client terminates the TCP connection, *read* returns 0, which breaks the while loop, close the socket and exits.

Client Program

The client makes two system calls to connect to the server.

```
sd = socket(AF_INET, SOCK_STREAM, 0);
connect(sd, (struct sockaddr *)&server, sizeof(server)
```

After the socket system call, the client pushes the server address information (server IP address and port number) into a structure variable called “server”. It then calls *connect* to make a TCP connection with the server. If *connect* returns with a non-negative number, a TCP connection has been established with the server.

The rest of the client code is shown below.

```
printf("Transmit: \n");
while(n=read(0, sbuf, BUFLEN)){                 /* get user message */
        write(sd, sbuf, n);                     /* send it out */
```

```
                printf("Receive: \n");
                bp = rbuf;
                bytes_to_read = n;
                while ((i = read(sd, bp, bytes_to_read)) >
                        0){  bp += i;
                        bytes_to_read -=i;
                  }
                  write(1, rbuf, n);
                  printf("Transmit: \n");
}

close(sd);
```

Once the TCP connection has been established, the client calls *read* to read a message from the terminal (descriptor 0 refers to standard input, i.e. the terminal). The system call *read* only returns after the user (that is you) typed a message and hit <CR>. After reading the message, the client calls *write* to write a message to the TCP socket. The effect of this is that the message is sent to the server across the network. The client then calls *read* to wait for the message echoed back. Since TCP may not send back the message in one packet, the client keeps reading until the number of bytes received is equal to the number of bytes sent. The last step of the echo process is that the client writes the message back to the terminal (descriptor 1 refers to standard output, which is again the terminal). Finally, if the user hits <CTRL-D>, the client will break out from the loop and close the TCP socket, which in turn triggers the termination of TCP connection. (Note: The termination is only triggered when the application that closes the socket is the last application attached to the socket.) Figure 3 (below) illustrates the sequences of system calls made in the server and client. It also shows TCP packet exchanges associated with those system calls.

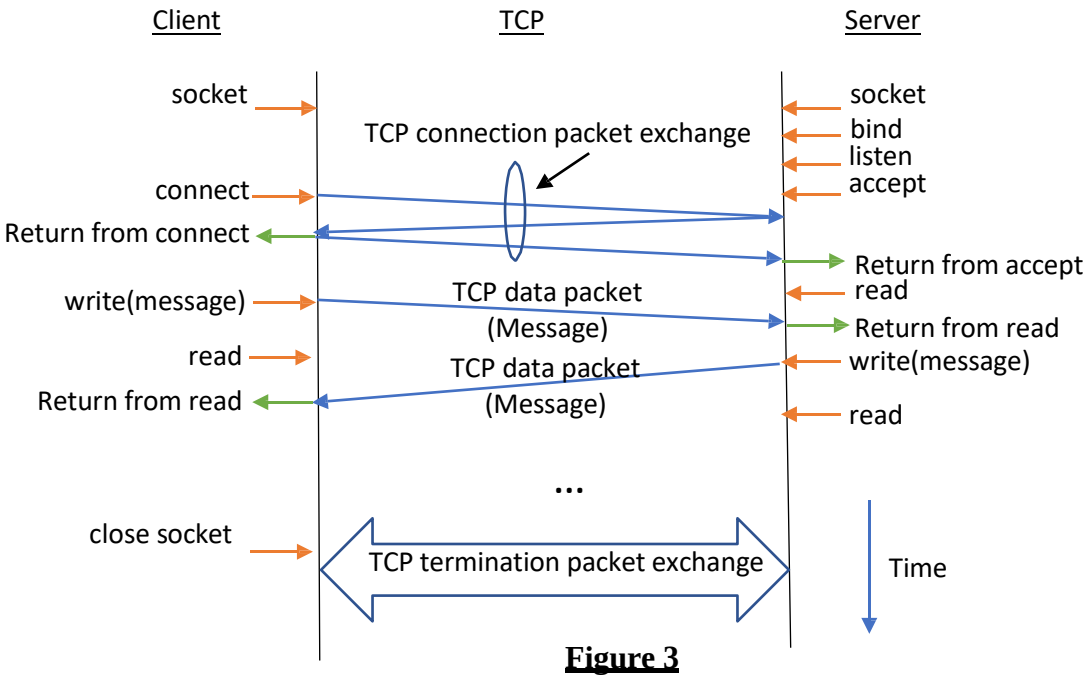

Figure 3

Your assignment in part III

You are required to write a simple "Hello" application. In this application, after the establishment of a TCP connection, the server sends a "Hello" message to the client. After sending the message, the server closes the TCP connection and exits. In the client side, the client will wait for the "Hello" message. Once it received a message, it displays the message at the terminal. The client also exits when it finds that the TCP connection is terminated (Note: the termination of a TCP connection will force the *read* system call to return with value 0.). Figure 4 illustrates the sequence of system calls and the associated packet exchanges. Implement the application by modifying the echo client and server programs.

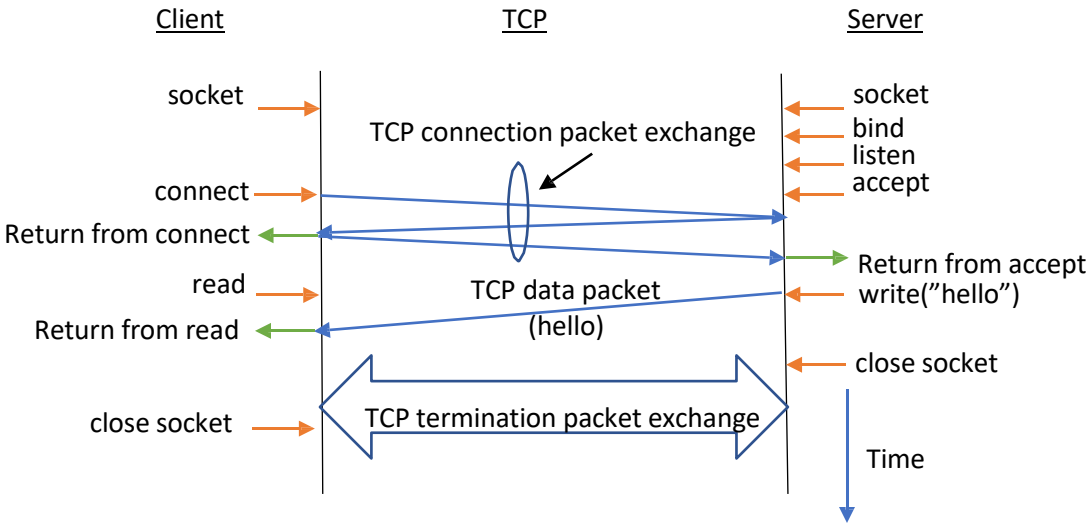

Figure 4

What you needed to demonstrate to your TA

1. Show your Wireshark captured data in part I. Identify the TCP connection packets and termination packets.

2. In part II, start 2 clients and show that the concurrent server created two child processes to service the clients.

3. Demonstrate your programs in part III.