CO323 - Lab 05
Introduction to Socket Programming - I

1) Sockets

A socket is a conceptual endpoint for a logical network between two processes. i.e., a socket is a 'point' where data can be sent to or received from another process. A socket is bound to a port number so that the TCP layer can identify the application that data is destined to be sent to.

Recap: TCP and UDP protocols

There are two major protocols used for network traffic in the transport layer on the Internet. They are namely, Transmission Control Protocol (TCP) and User Datagram Protocol (UDP). Both of them reside in the Transport Layer of the OSI model. The transport layer is the layer that focuses on process-to-process communication. i.e., breaking the message into manageable pieces (packets) and multiplexing data between processes.

The difference between the two in the view of the programmer is the reliability of the two. TCP aims to provide an error-free and reliable data transfer while the UDP expects the upper layers to treat the data transfer as inherently unreliable and handle the errors and ordering of packets. To provide reliability, TCP creates a 'circuit', essentially a path for the packets to travel to the destination. This preserves the ordering of the packets and prevents duplication. Therefore, these are guaranteed when using TCP.

UDP on the other hand does not create such a path. The packets are on their own to figure out a path from the source to the destination. This might result in packet drops, duplication of packets, and out-of-order sequencing of packets. Therefore UDP does not provide any guarantee for any of these. Both of these protocols are implemented on top of Internet Protocol (IP). A basic protocol in the Internet Layer of the OSI model. This is the layer that focuses on host-to-host communication. Internet Protocol essentially treats the underlying network as unreliable and does not provide any error checking or sequencing services.

---

2) **Creating a UDP server**

A server is the one who waits for a client to connect to. Therefore the client should know the IP address and the port of the server. But the server does not have to know the clients' address. The following steps should be taken to write a UDP server.

Algorithm

1. create a socket with appropriate options for UDP.
2. initialize a *sockaddr_in* struct with appropriate options for our server address.
3. bind the socket with the created *sockaddr_in* struct.
4. send and receive messages from clients that *sockaddr_in* struct is configured to use.

```c
/* Sample UDP server */
#include <sys/socket.h>
#include <netinet/in.h>
#include <strings.h>
#include <stdio.h>

int main(int argc, char ** argv) {
  int sockfd, n;
  struct sockaddr_in servaddr, cliaddr;
  socklen_t len;
  char mesg[1000];
  char * banner = "Hello UDP client! This is UDP server";

  // Create a socket
  sockfd = socket(AF_INET, SOCK_DGRAM, 0);

  // Initialize server address
  servaddr.sin_family = AF_INET;
  servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
  servaddr.sin_port = htons(32000);

  // Bind the socket
  bind(sockfd, (struct sockaddr * ) & servaddr, sizeof(servaddr));

  // Send and Receive
  len = sizeof(cliaddr);
  n = recvfrom(sockfd, mesg, 1000, 0, (struct sockaddr * ) & cliaddr, & len);
  sendto(sockfd, banner, n, 0, (struct sockaddr * ) & cliaddr, sizeof(cliaddr));
  mesg[n] = 0;
  printf("Received: %s\n", mesg);

  return 0;
}
```

The socket() call

The *socket()* function call creates a socket. The return value of this call is an integer called a socket descriptor. In UNIX-like systems, this is a file descriptor, and common operations on file descriptors such as *read()* and *write()* still work with it. Additionally, specialized functions such as *send, sendto(), sendmsg(), recv(), recvfrom()* and *recvmsg()* can also be used with socket descriptors to send and receive data.

To create a socket three arguments are needed. First, a domain. Different domains exist. For example, the *AF_UNIX* option is used to communicate in the local domain (i.e., inside the localhost itself) and the *AF_INET6* option is used to communicate using IPv6. The option *AF_INET* specifies that we're using IPv4 to communicate.

The next argument specifies the type of socket. *SOCK_DGRAM* specifies that the socket is of type UDP.

The last argument specifies a protocol. This is useful if more than one protocol is supported by the specified type of socket. In this case, there is only one Therefore this number is kept as zero.

Recap: Input-output of UNIX systems

Most of the I/O devices are treated as files in UNIX systems. A common interface *read()* and *write()* is provided to the application programmer by the kernel to deal with these.

The sockaddr_in struct

This struct stores IP socket address. An IP socket address contains an IP address of an interface as well as a port number. It contains the following fields:

1. *sin_family*: specifies an address family. Must be *AF_INET*

2. *sin_addr*: stores a struct of *in_port_t* which contains only one field: *s_addr* which stores the IP address in the network byte order

   The network and host byte orders and *htonl()* and *htons()* functions
   There are two ways the memory of a computer can be interpreted. They are:
   - the value in the lowest address is considered as the most significant byte (named Big Endian)
   - the value in the lowest address is considered as the least significant byte (named Little Endian).

   This is called the 'byte order' of the architecture. The x86 and x86_64 architectures are little-endian while the network communication is big-endian. Whichever the byte order of the two is, *htonl()* and *htons()* functions convert the

byte order of the host machine to the byte order of the network. In the struct, it is the convention to store the port number in the network byte order.

3. *sin_port*: Stores the port to be used in the network byte order.

Bind() call

This gives the socket a 'name' with the given IP address. This registers that the socket is using that IP address. Since the server is the initial receiver of the data, this is an essential step. The *bind()* function expects the first argument to be in *struct sockaddr** type. This is essentially the same as struct sockaddr_in in most applications and the casting is done just to get rid of a possible warning.

In the program, you might notice that *recv_from* has an uninitialized argument *cliaddr* of type *struct sockaddr_in** as the second argument. The use of *recvfrom()* over *read()* or *recv()* is that, once the data is received, it fills the fields of *cliaddr* with the information of the source the data is received from. This way, the server could figure out the IP address information of the sender and reply to it. As with *read(), recv()* is always blocking

3) **Creating a UDP client**

Unlike the server, the client does not have to bind the socket since the server does not need to know the clients' IP address initially before the client initiates the communication with the server.

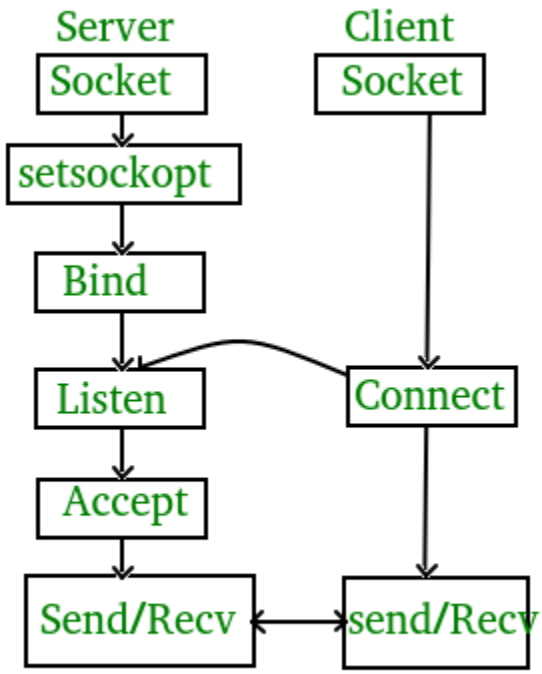
Algorithm

1. Create a socket
2. Initialize a sockaddr_in struct with the information of the server intended to connect to
3. send and receive messages

```
/* Sample UDP client */
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char ** argv) {
  int sockfd, n;
  struct sockaddr_in servaddr;
  char sendline[] = "Hello UDP server! This is UDP client";
  char recvline[1000];

  if (argc != 2) {
    printf("usage:%s <IP address>\n", argv[0]);
    return -1;
  }

  // Create a socket
  sockfd = socket(AF_INET, SOCK_DGRAM, 0);

  // Initialize server address
  bzero( & servaddr, sizeof(servaddr));
  servaddr.sin_family = AF_INET;
  servaddr.sin_addr.s_addr = inet_addr(argv[1]);
  servaddr.sin_port = htons(32000);

  // Send and Receive
  sendto(sockfd, sendline, strlen(sendline), 0, (struct sockaddr * ) & servaddr,
sizeof(servaddr));
  n = recvfrom(sockfd, recvline, 10000, 0, NULL, NULL);
  recvline[n] = 0;
  printf("Received: %s\n", recvline);
  return 0;
}
```

4) **Creating a TCP server and Client**

Creating a TCP server and client is similar to the creation of a UDP server and a client. However, there are several differences. Since a TCP connection has to be established before sending data unlike UDP, TCP servers listen to incoming connection requests via *listen()*. When a TCP client sends a *connect()* request, the TCP server accepts the connection by function *accept()*.



Source: https://www.geeksforgeeks.org/tcp-server-client-implementation-in-c/

Exercises:

1. A server does not reply only to one client and exits. It waits indefinitely, replying for all clients until it is forced to terminate by the host computer. Modify the server to do this
2. An echo UDP server is a simple server that receives a message from a client and sends the same message back to the client. Write a simple echo server that achieves this task.
3. Write a time server (UDP) that, when connected, sits in a loop and sends the current time each second to the receiver.
4. Create a TCP server that implements the hypothetical Capitalizer Service Protocol (CSP) to provide a capitalizer service.
    - CSP works as follows:
        - First, the client initializes the connection by sending an integer that represents the number of sentences it is going to send. (Let's say the number is n)
        - Then, the CSP server sends the characters 'ack' back to the client, acknowledging the message.
        - CSP server then sits in a loop that iterates 'n' times. At each iteration the server does the following:
            - Receive a sentence from the client
            - Capitalize it
            - Send it back to the client

Submit a zip file **E17XXX_Lab05.zip** (XXX is your E Number) which contains the following.

1. **ex1_client_UDP.c** and **ex1_server_UDP.c**
2. **ex2_client_UDP.c** and **ex2_server_UDP.c**
3. **ex3_client_UDP.c** and **ex3_server_UDP.c**
4. **ex4_client_TCP.c** and **ex4_server_TCP.c**

References

1) https://www.tutorialspoint.com/difference-between-tcp-and-udp
2) https://www.geeksforgeeks.org/udp-server-client-implementation-c/
3) https://www.educative.io/edpresso/how-to-implement-udp-sockets-in-c
4) https://www.geeksforgeeks.org/tcp-server-client-implementation-in-c/
5) https://www.educative.io/edpresso/how-to-implement-tcp-sockets-in-c