SOCKET PROGRAMMING

PRILIMINERIES

- The client server communication the actual communication is happening between two endpoints.

- The endpoints are nothing but two processes. Let us call the process at the client side as the client process and at the server side as the server process hereafter.

- In that way a client server communication is nothing but inter process communication, where the processes are usually running in two different machines connected to some network

How to Distinguish an Endpoint?

- Definitely in someway, the client need to distinguish the server process (from thousands of m/c s in the network, each runs possibly a number of processes) , to get it request serviced.

- For this the server process must run at some well known address, which can be used to uniquely identify the process from many processes that are currently running in the machines. This addresses in client server communication is known as the ports. (Remember the popular services always runs at well-known ports, ftp (TCP port 21), http (TCP port 80), which enables the clients to always connected to a server for a popular service.

How to distinguish the machine?

- The port can be used to identify the process. The How can the client identify the server machine from thousands of machines in the network.

- This can be accomplished by the machine address. And in the Internet domain this is nothing but the IP Address. As you know it is a 32 bit integer (IPV4), which contains the network id and the machine id within the network.

WHAT ABOUT THE PROTOCOL?

- Remember a TCP port, say 3000 is different from a UDP port 3000.
- So to identify a service, the client must know on which protocol, the service is running.
- Remember in Internet Domain, the connection oriented services runs using protocol TCP and the Datagram (Message oriented / Connection less) Services runs using the protocol UDP.

SOCKET

- So the protocol, the machine address and the port determines the endpoint of communication. It is nothing but a socket.

- So a server socket has three fields. Protocol, Server IP address and the port number.

- For example, if you want to write a server which is providing some connection-less service, following constitute a socket.

  (udp, 192.168.0.2 , 2000)

How can a server respond?

- Definitely the client can send some messages to the server, as it knows the server end point and since it is well-known.
- In most cases the client is expecting some response from the server.
- So How can the server respond, without having the end point at the client side.

EPHEMERAL PORT FOR THE CLIENT

- So when a client wants to connect to the server running on some well known address using well known protocol, the protocol module at the client side assigns an arbitrary, but currently un used port greater than 1024 for the client process.

- The client is not at all interested on this port, but actually the other end point i.e., the server is concerned about this as it needs to respond to the client request. Such a port is usually termed as an ephemeral port.

ASSOCIATION?

- For an effective communication there should be two end points. i.e., the server and the client.
- The server and the client sockets together determines the association for communication.
- (protocol, Server IP, Server Port, Client IP, Client Port) determines an association. Remember for communication to happen this association must be unique.

SERVER FILLS UP THE ASSOCIATION

- When the client connects to the server, the server fills up the association , i.e., fills up the client machine address and the client ephemeral port, so that the server can respond back.

A Simple Client and Server

- Let's write a simple server servicing on UDP port, say 5000.
- When the client connects to the server, The server responds with the message "Hello" for a Client message "Hi".

THE MODEL

server

1. Create the End point.
2. Receive the Message “Hi” and print it.
3. Fill up the Association.
4. Send the Message “Hello”, to the client.

client

1. Create the End point.
2. Send the Message “Hi” to the well known server Address.
3. Receive the Message “Hello” and print it.

Remember

A client always assumes there is a server, to get it’s request serviced. i.e., the server process must be started first.

Creating the Endpoint at Server

- The following two system calls (sys/socket.h) create the end point of communication.

- socket()
- **bind()**

Creates the Endpoint

COMPLETES THE ASSOCIATION

recvfrom() is a blocking call, which will waits un till some client connects and send a message (“Hi”) to the server. Once the message has arrived revcvfrom() receives the message and it also fills up the client details to complete the association. This association can be later used to send some response (“Hello”) to the client using the sendto() call.

LET US SEE THE CODE

- Before going to analyse the code, let's see some structures, where you can fill the machine address and the port information.

STRUCT SOCKADDR

```
struct sockaddr
   {
        unsigned short sa_family;
        char sa_data[14];
   };
```

Members

- *sa_family* : To specify the Socket address family.
- *sa_data* : Maximum size of all the different socket address structures.

SOCKADDR MEMBERS

sa_family: There are different protocol families. For the internet family, the macro AF_INET is used to specify it. There are other families like AF_UNIX, AF_XNS, ….

sa_data : is capable of holding the address and port information of all types of socket families. Remember different family has different types of addresses and hence the length of the valid filled in data area may be different from one type to another type. For example the Internet family uses 4 byte IP address with 2 byte port while XNS uses a 10 byte address with 2 byte port. The unfilled area will be filled with 0.

INTERNET DOMAIN SPECIF STRUCTURE

```
struct sockaddr_in
    {
        short sin_family;              //set to AF_INET
        unsigned short sin_port;       //set to the well known port
        struct in_addr sin_addr;       //set to the IP Address
        char sin_zero[8];              //kept as 0
    };
```

Parameters

- *sin_family* Address family (must be **AF_INET**).
- *sin_port* IP port.
- *sin_addr* IP address.
- *sin_zero* Padding to make structure the same size as **SOCKADDR**.

(Here 8 bytes needed to be padded, in some other domain with different addressing scheme, the number of bytes padded may be different)

STRUCT IN_ADDR

- The third member of struct sockaddr_in is another structure variable sin_addr of type struct in_addr. This is capable of holding the 32 bit ip address.

```
struct in_addr
{
unsigned long s_addr; // load with inet_aton()
};
```

SERVER

```
struct sockaddr_in server;

bzero ((char*) &server, sizeof (server));
server.sin_family = AF_INET;
inet_aton ("127.0.0.1", &server.sin_addr);
server.sin_port = htons (3000);
```

The above code fragments fills in the internet specific address. Remember we used the string handling function bzero (string.h) to initialise the structure with 0 instead of explicitly setting 8 byte long, sin_zero to zero. 127.0.0.1 is the localhost or the local loop address where we want to create the server process.

htons (arpa/inet.h) – host to network short, function convert the 3000 to the network byte order short value. Network byte order is the default standard to be followed.

CREATE THE ENDPOINT OF COMMUNICATION

unsigned int sockfd; **<sys/socket.h>**

sockfd = socket (AF_INET, SOCK_DGRAM, IPPROTO_UDP);

bind (sockfd, (struct sockaddr*)&server, sizeof (server));

socket() call takes the protocol family (in our case it is Internet family), type (message/datagram oriented connection less service) and the protocol to support it. socket calls returns a non zero integer as the valid socket descriptor.

bind() binds the socket descriptor received from the socket call with the address, which is already filled in struct sockaddr_in, and hence complete the creation of the endpoint at the server side.

RECEIVE MESSAGE FROM THE CLIENT AND COMPLETE THE ASSOCIATION

struct sockaddr_in client;
unsigned int clientlen;
char rmsg[100],
recvfrom (sockfd, rmsg, 3, 0, (struct sockaddr*) &client, &clientlen);

recvfrom is a blocking system call which waits un till the clients has placed it's message. Once the message has placed it completes the association by filling the details of the client (from the udp header (ephemeral port) and from the IP header (client's IP Address) of the incoming message) to the address specified by the 5th and 6th arguments. It also reads n bytes (here max. 10) and place it in the buffer (rmsg). Also notice that the address of the internet specific client address structure is type casted to the generic socket structure, so that the same function can be utilised for the different domains. See the 4th argument is flag and is given as 0. 0 is specified for normal message. If it is some urgent data (buffering is avoided in such case) you can give it as MSG_OOB (out of band message) or MSG_PEEK, etc.

SENDING A MESSAGE TO THE CLIENT

```
char rmsg[100], smsg[] = “Hello";
sendto (sockfd, smsg, 6, 0, (struct sockaddr*)&client,
    clientlen);
```

Note that the last argument is not a pointer, which was a pointer in the recvfrom call. It is because the length has to be filled in by the recvfrom function call.

SERVER.C

```
#include <string.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/types.h>
#include <sys/socket.h>

#include <stdio.h>
int main()
{
        struct sockaddr_in server;
        struct sockaddr_in client;
        unsigned int sockfd, clientlen;
        char rmsg[100], smsg[] = "Hello";

        sockfd = socket (AF_INET, SOCK_DGRAM, IPPROTO_UDP);
        bzero ((char*) &server, sizeof (server));
        server.sin_family = AF_INET;
        inet_aton ("127.0.0.1", &server.sin_addr);
        server.sin_port = htons (3000);

        bind (sockfd, (struct sockaddr*)&server, sizeof (server));

        recvfrom (sockfd, rmsg, 3, 0, (struct sockaddr*)&client, &clientlen);
        printf ("%s", rmsg);
        sendto (sockfd, smsg, 6, 0, (struct sockaddr*)&client, clientlen);
        return 1;
}
```

CLIENT.C

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <string.h>

int main()
{
        struct sockaddr_in serv_addr, client_addr;
        unsigned int sockfd, client_len;

        sockfd = socket (AF_INET, SOCK_DGRAM, 0);
        bzero ((char*)&serv_addr, sizeof (serv_addr));
        serv_addr.sin_family = AF_INET;
        inet_aton ("127.0.0.1", &serv_addr.sin_addr);
        serv_addr.sin_port = htons (3000);

        bzero ((char*) &client_addr, sizeof (client_addr));
        client_addr.sin_family = AF_INET;
        inet_aton ("127.0.0.1", &client_addr.sin_addr);
        client_addr.sin_port = htons (0);

        bind (sockfd, (struct sockaddr*)&client_addr, sizeof (client_addr));

        char msg[100] = "Hi";
        sendto (sockfd, msg, 3, 0, (struct sockaddr*)&serv_addr, sizeof (serv_addr));
        recvfrom (sockfd, msg, 6, 0, (struct sockaddr*)&client_addr, &client_len);
        printf ("%s", msg);
        return 0;
}
```

CLIENT

You can see in the client program, the details of both the client and the server are filled in the respective structures and you are binding the client address with the socket descriptor. Notice that the port given for the client is 0. This will cause the UDP module assigns an unused ephemeral port for the client process.

How to run the program

- save both the client and server as client.c and server.c
- open two terminals
  - type gcc server.c –o server in the first terminal followed by ./server . You can see the server waiting for some message .
  - type gcc client.c –o client in the second terminal follwed by ./client.

AUM NAMASHIVAYA