Introduction to Socket programming with C

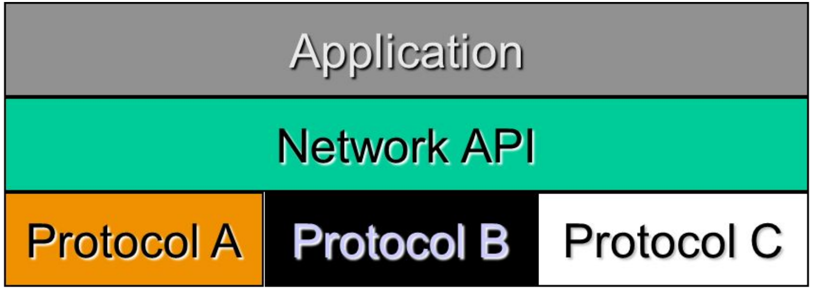
Why do we need sockets?

- ❑ A socket is a communications connection endpoint that you can name and address in a network
- ❑ Provides an abstraction for interprocess communication services (often provided by the operating system) that provide the interface between application and protocol software
- ❑ Socket programming shows how to use socket APIs to establish communication links between remote and local processes

How sockets work

Sockets are commonly used for client and server interaction. Typical system configuration places the server on one machine, with the clients on other machines. The clients connect to the server, exchange information, and then disconnect.

Goal: learn how to build client/server applications that communicate using sockets

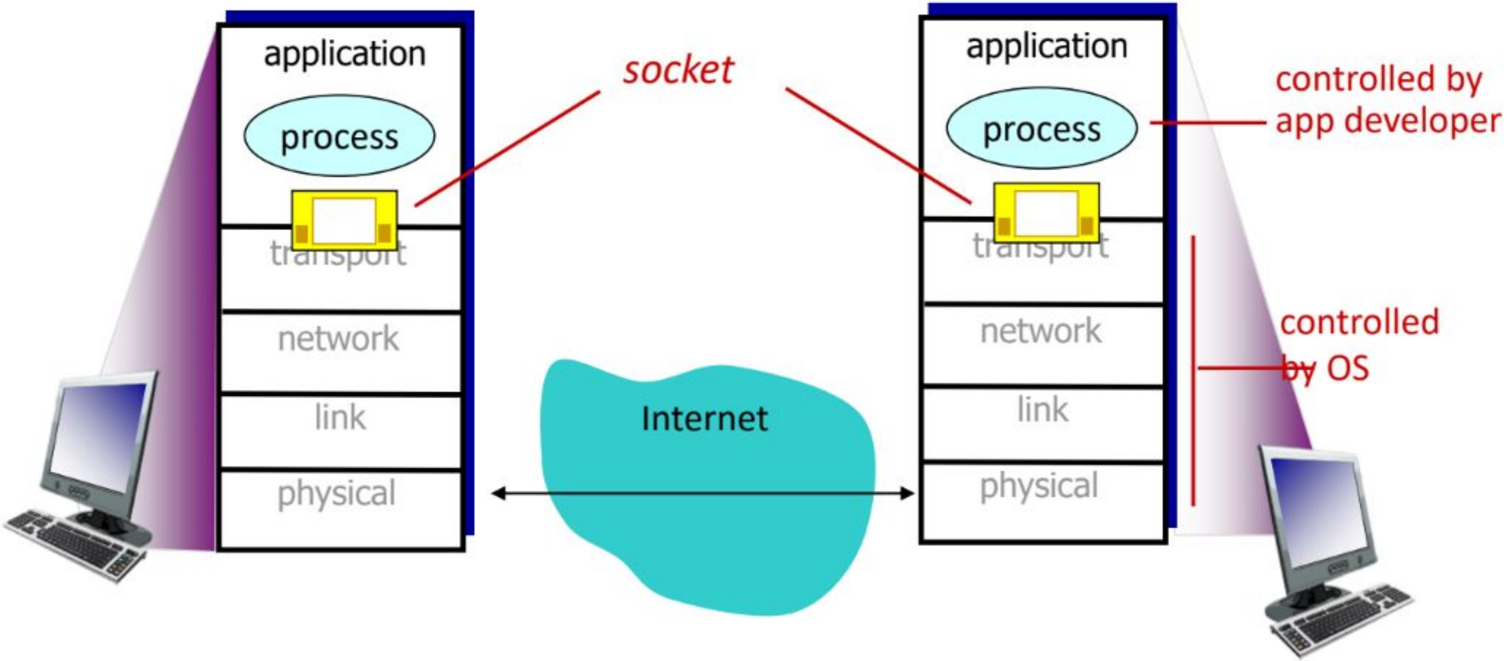


Socket Programming: Basics

1. The server application must be running before the client can send anything.
2. The server must have a socket through which it sends and receives messages. The client also need a socket.
3. Locally, a socket is identified by a port number.
4. In order to send messages to the server, the client needs to know the IP address and the port number of the server.

Port number is analogous to an apartment number. All doors (sockets) lead into the building, but the client only has access to one of them, located at the provided number.

Contd..


application
process
transport
network
link
physical
socket
Internet
application
process
transport
network
link
physical
controlled by
app developer
controlled
by OS

Functions

1. Define an “endpoint” for communication
2. Initiate and accept a connection
3. Send and receive data
4. Terminate a connection gracefully

Examples

File transfer apps (FTP), Web browsers (HTTP), Email (SMTP/ POP3), etc…

Types of Sockets

Socket programming Two socket types for two transport services:

UDP – *unreliable datagram*

TCP – *reliable, byte stream-oriented*

Application Example:

1. **client reads a line of data from its keyboard and sends data to server**
2. **server receives the data and converts characters to uppercase**
3. **server sends modified data to client**
4. **client receives modified data and displays line on its screen**

Contd..

- Two different types of sockets :
  1. Stream
  2. datagram

stream socket: (a. k. a. connection-oriented socket)

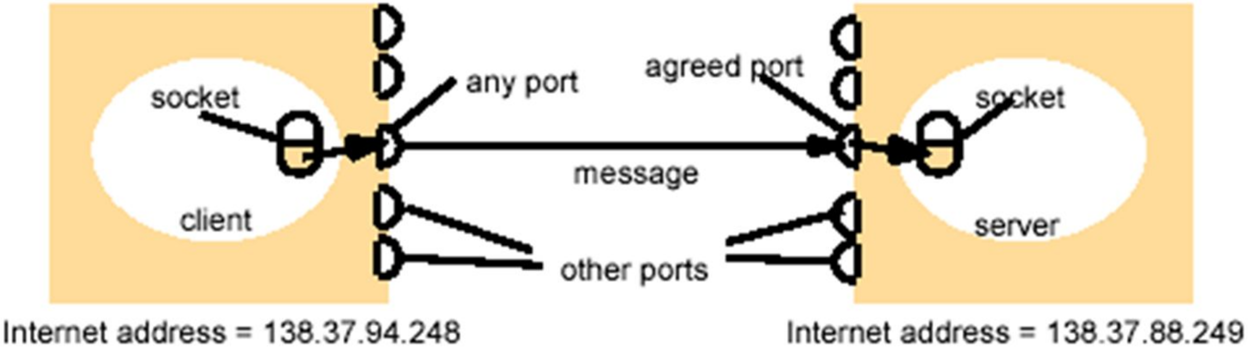
- It provides reliable, connected networking service Error free;
- no out-of-order packets (uses TCP)
- applications: telnet, ssh, http etc.

datagram socket: (a. k. a. connectionless socket)

- It provides unreliable, best- effort networking service
- Packets may be lost; may arrive out of order (uses UDP)
- applications: streaming audio/video (realplayer)

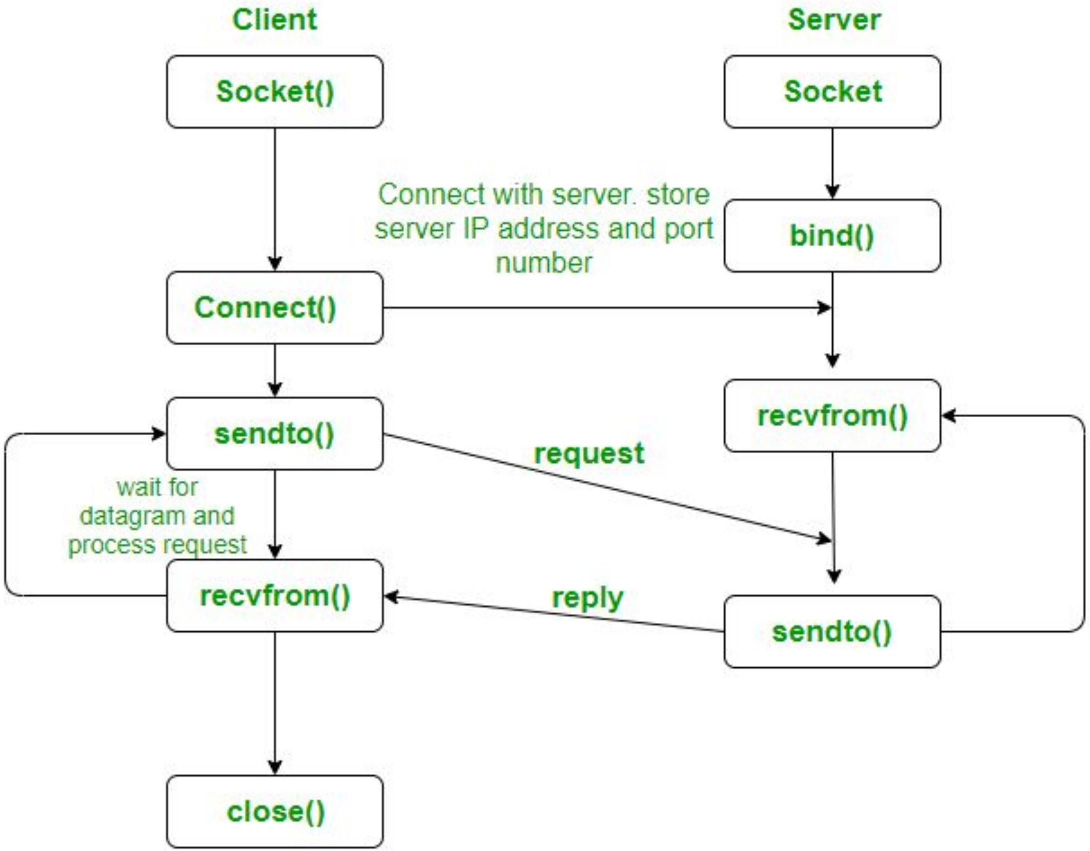
Addresses, Ports and Sockets

- Like apartments and mailboxes
  - You are the application
  - Your apartment building address is the address
  - Your mailbox is the port
  - The post-office is the network
  - The socket is the key that gives you access to the right mailbox



socket
client
any port
agreed port
socket
server
message
other ports
Internet address = 138.37.94.248
Internet address = 138.37.88.249

State diagram of *UDP* Socket


Client
Server
Socket()
Socket
Connect with server. store server IP address and port number
bind()
Connect()
recvfrom()
sendto()
request
wait for datagram and process request
recvfrom()
reply
sendto()
close()

Server side code of *UDP* Socket

```
int main()
{
    char buffer[100];
    char *message = "Hello Client";
    int listenfd, len;
    struct sockaddr_in servaddr, cliaddr;
    bzero(&servaddr, sizeof(servaddr));

  // Create a UDP Socket
    listenfd = socket(AF_INET, SOCK_DGRAM, 0);
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(PORT);
    servaddr.sin_family = AF_INET;
```

family	AF_INET AF_INET6 AF_UNIX
type	SOCK_STREAM SOCK_DGRAM SOCK_RAW
protocol	0

s = socket(family, type, protocol);

htonl : host byte order to network byte order conversion

Contd..

```
// bind server address to socket descriptor
   bind(listenfd, (struct sockaddr*)&servaddr, sizeof(servaddr));
```

```
bind (socket, name, namelen);
```

```
//receive the datagram
   len = sizeof(cliaddr);
   int n = recvfrom(listenfd, buffer, sizeof(buffer), 0, (struct sockaddr*)&cliaddr,&len);
```

```
int recvfrom (int socket, char *buffer,
              int length, int flags,
              struct sockaddr *address,
              int *address_length);
```

```
//receive message from server
   buffer[n] = '\0';
   puts(buffer);
// send the response
   sendto(listenfd, message, MAXLINE, 0,
         (struct sockaddr*)&cliaddr, sizeof(cliaddr));
}
```

```
sendto( int socket,
        char *buffer,
        int length, int flags,
        struct sockaddr *address,
        int address_len);
```

Client side code of *UDP* Socket

```
int main()
{
    char buffer[100];
    char *message = "Hello Server";
    int sockfd, n;
    struct sockaddr_in servaddr;

    // clear servaddr
    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_addr.s_addr = inet_addr("127.0.0.1");
    servaddr.sin_port = htons(PORT);
    servaddr.sin_family = AF_INET;

    // create datagram socket
    sockfd = socket(AF_INET, SOCK_DGRAM, 0);
```

Contd..

```
// connect to server
  if(connect(sockfd, (struct sockaddr *)&servaddr, sizeof(servaddr)) < 0)
  {   printf("\n Error : Connect Failed \n");
      exit(0); }
```

```
connect( int socket,
struct sockaddr *address,
            int address_len)
```

```
// request to send datagram
// no need to specify server address in sendto
// connect stores the peers IP and port
  sendto(sockfd, message, MAXLINE, 0, (struct sockaddr*)NULL, sizeof(servaddr));
```

```
sendto( int socket, char *buffer,
int length, int flags, struct sockaddr
*address, int address_len);
```

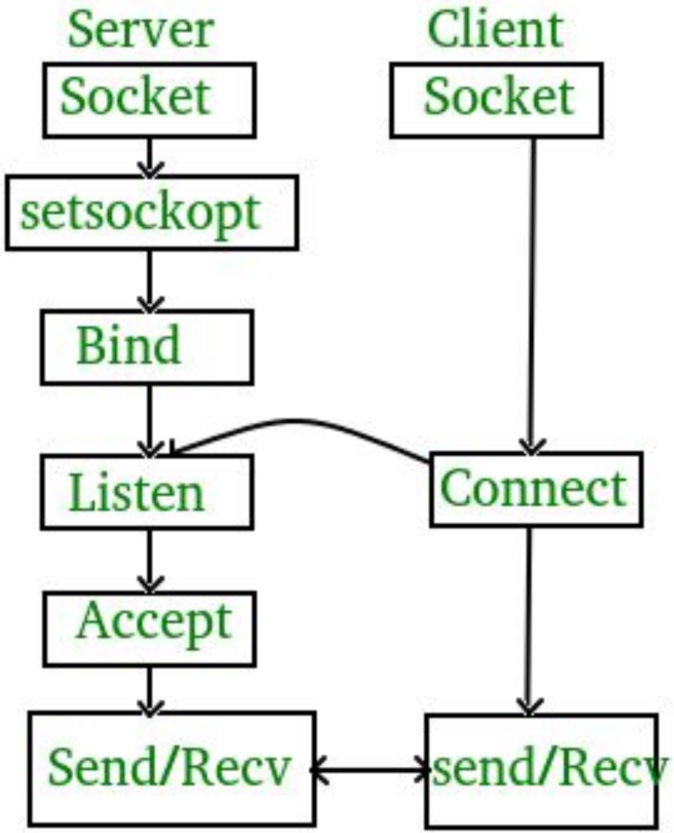
```
// waiting for response
  recvfrom(sockfd, buffer, sizeof(buffer), 0, (struct sockaddr*)NULL, NULL);
  puts(buffer);

// close the descriptor
  close(sockfd);
}
```

```
recvfrom( int socket, char *buffer,
          int length, int flags,
          struct sockaddr *address,
          int *address_length);
```

Source code link

shorturl.at/fqzB8

State diagram of *TCP* Socket



Server side steps of *TCP* Socket

TCP Server –

1. **using create(), Create TCP socket.**
2. **using bind(), Bind the socket to server address.**
3. **using listen(), put the server socket in a passive mode, where it waits for the client to approach the server to make a connection**
4. **using accept(), At this point, connection is established between client and server, and they are ready to transfer data.**
5. **Go back to Step 3.**

Client side steps of *TCP* Socket

TCP Client –

1. **Create TCP socket.**

2. **connect newly created client socket to server.**

Source code link

shorturl.at/fqzB8

server_TCP.c

client_TCP.c

Thank you