University of Central Punjab

(Faculty of Information Technology)

Course: CSNC 2411

Computer Communications and Networks (Lab)

Lab 2

Socket Programming:
Introduction to Socket API

(UDP server)

Lab Manual 02

Objectives

- What is a Socket?
- Socket Types (UDP/TCP)
- Socket Descriptors
- IP address, Port Number & Socket Address
- Byte Ordering
- Socket Address Structures
- The client-server model
- UDP Socket API
- UDP client and server communication

Reference Material

What is a Socket?

From a logical perspective, a Socket is a communication end point identified by a socket descriptor. It is not a physical entity, such as a connection on your network card. A socket address is specified by IP address and port number.

Socket Types

There are mainly two types of sockets:

Datagram Sockets : use UDP to provide best-effort datagram transpsort service
Stream Sockets : use TCP to provide reliable byte-stream service

Note: In this lab we will discuss only UDP/Datagram sockets.

Socket/File Descriptors

File descriptors are normally small non-negative integers that the kernel uses to identify the files being accessed by a particular process. Whenever it opens an existing file or creates a new file, the kernel returns a file descriptor that is used to read or write the file. As we will see in this course, sockets are based on a very similar mechanism (socket descriptors).

IP Address

IP4 addresses are 32 bits long. They are expressed commonly in what is known as dotted decimal notation. Each of the four bytes which makes up the 32 address are expressed as an integer value (0 – 255) and separated by a dot. For example, 138.23.44.2 is an example of an IP4 address in dotted decimal notation.

The importance of IP addresses follows from the fact that each host on the Internet has a unique IP address. Thus, although the Internet is made up of many networks of networks with many

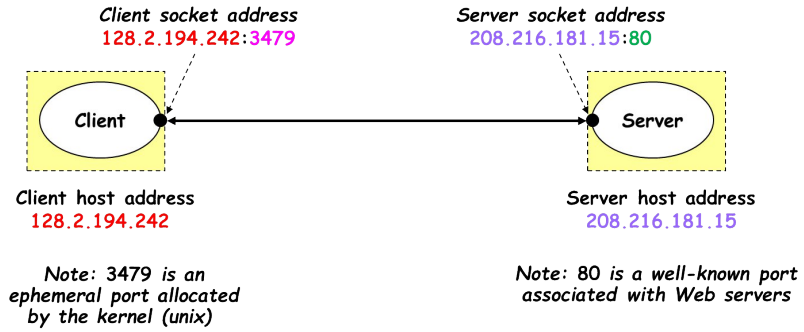
different types of architectures and transport mediums, it is the IP address which allows any two hosts on the Internet to communicate with each other.

Port Numbers & Types

TCP and UDP port numbers ranges: values $0 – 2^{16}$ (65,536 ports)
Well Known Ports (0 - 1023) used by system processes for well-known services e.g, Telnet: 23, E-mail: 25, Http: 80, etc
Registered Ports (1024 - 49151) e.g, Kaaza: 1214, IPSec: 1293, Web Proxy: 8080
Dynamic or Ephemeral Ports (49152 - 65535) used by clients automatically allocated by kernel on temporary basis.

Socket Address

A socket address is the combination of IP address + Port Number.



Byte Ordering

Port numbers and IP Addresses are represented by multi-byte data types which are placed in packet headers for the purpose of routing and multiplexing. Port numbers are two bytes (16 bits) and IPv4 addresses are 4 bytes (32 bits).
A problem arises when transferring multi-byte data types between different architectures. Say Host A uses a "big-endian" architecture and sends a packet across the network to Host B which uses a "little-endian" architecture. If Host B looks at the IP address to see if the packet is for him, it will interpret the bytes in the opposite order and will wrongly conclude that it is not his packet. The Internet uses big-endian and we call it the network-byte-order.
We have the following functions to convert host-byte-ordered values into network-byte-ordered values and vice versa:

To convert IP4 Addresses (32 bits):

Host -> Network
unit32_t htonl(uint32_t hostIPaddr)
Network -> Host
Unit32_t ntohl(uint32_t hostIPaddr)

To convert port numbers (16 bits):

Host -> Network
unit16_t htons(uint16_t hostPortNumber)

Network -> Host
unit16_t ntohs(uint16_t netPortNumber)

Socket Address Structures

Socket API requires the use of specific address structures to hold values such as, IP address, port number, and protocol type, etc. IPv4 socket address structure is named sockaddr_in and is defined by including the <netinet/in.h> header.

```
struct sockaddr_in {
  uint8_t  sin_len;          /* length of structure (16)*/
  sa_family_t  sin_family;   /* AF_INET*/
  in_port_t  sin_port;       /* 16 bit TCP or UDP port number */
  struct  in_addr  sin_addr; /* 32 bit IPv4 address*/
  char  sin_zero[8];         /* not used but always set to zero */
};

struct  in_addr {
in_addr_t  s_addr;           /*32 bit IPv4 network byte ordered address*/
};
```

Generic Socket Address Structure

A socket address structure is always passed by reference as an argument to any socket functions. But any socket function that takes one of these pointers as an argument must deal with socket address structures from any of the supported protocol families.

A problem arises in declaring the type of pointer that is passed. With ANSI C, the solution is to use void * (the generic pointer type). But the socket functions predate the definition of ANSI C and the solution chosen was to define a generic socket address as follows:

```
struct  sockaddr {
  uint8_t  sa_len;
  sa_family_t  sa_family;    /* address family: AD_xxx value */
  char  sa_data [14];
};
```

Process

An executing instance of a program is called a process. Sometimes, task is used instead of process with the same meaning. UNIX guarantees that every process has a unique identifier called the process ID. The process ID is always a non-negative integer.

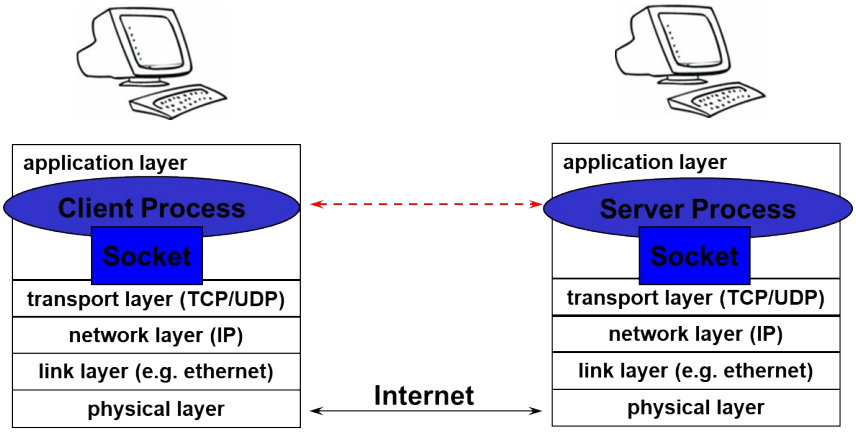
The Client-Server model

The client-server model is one of the most used communication paradigms in networked systems. Clients normally communicate with one server at a time. From a server's perspective, at any point in time, it is not unusual for a server to be communicating with multiple clients. **In this Lab the server will communicate with one client only.** Client needs to know of the existence of and the address of the server, but the server does not need to know the address of (or even the existence of) the client prior to the connection being established

Client/Server: processes running on same or different hosts

Client Requests: sends a message to server to perform a task

Server Responds: performs task & sends back reply



Server

- long-running application processes (daemons) e.g, web server, or mail server
- created typically at boot-time by OS
- run continuously in background

Client

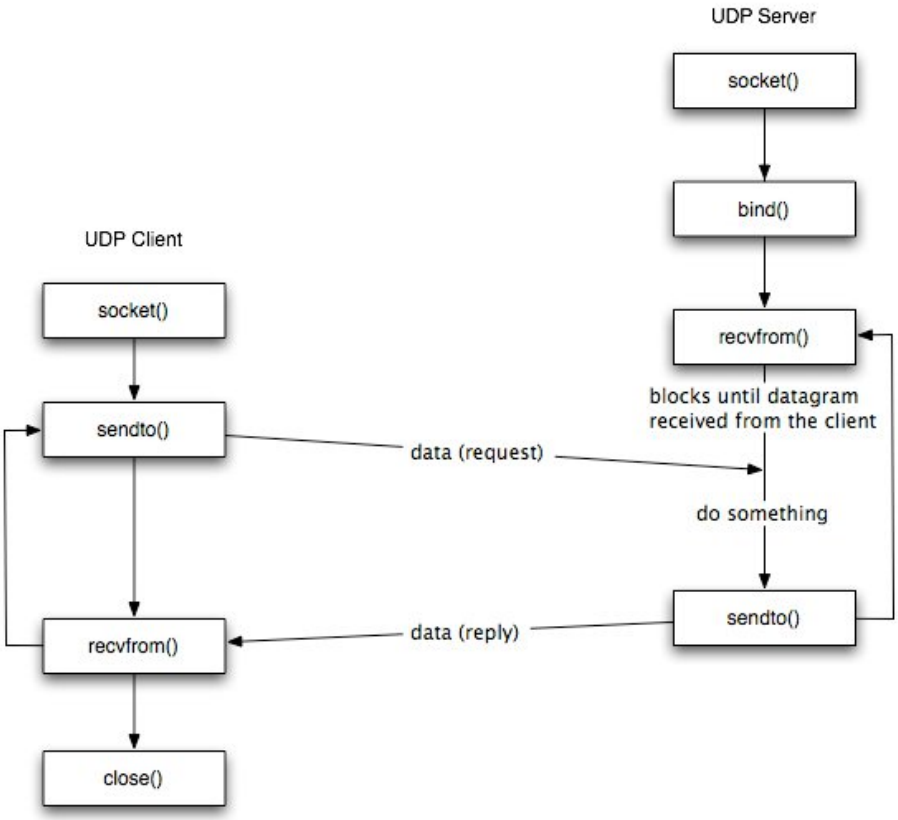
- application that accesses a remote service on another computer e.g, web browser
- Client does not need a well-known port
- usually assigned an ephemeral port dynamically by kernel
- port can also be selected by application

UDP (User Datagram Protocol)

- UDP is a simple transport-layer protocol. The application writes a message to a UDP socket, which is then encapsulated in a UDP datagram, which is further encapsulated in an IP datagram, which is sent to the destination.
- There is no guarantee that a UDP will reach the destination that the order of the datagrams will be preserved across the network or that datagrams arrive only once.

- The problem of UDP is its lack of reliability: if a datagram reaches its final destination but the checksum detects an error, or if the datagram is dropped in the network, it is not automatically retransmitted.
- Each UDP datagram is characterized by a length. The length of a datagram is passed to the receiving application along with the data.
- No connection is established between the client and the server and, for this reason, we say that UDP provides a connection-less service.

UDP Client/Server Flow Diagram



Create Socket

int ***socket*** (int ***family***, int ***type***, int ***protocol***);

create a socket

returns socket descriptor; -1 on error and sets errno

family : address family / protocol family

- AF_INET for IPv4, AF_INET6 for IPv6

type : type of communication

- SOCK_STREAM for TCP
- SOCK_DGRAM for UDP
- SOCK_RAW for Raw socket

protocol: protocol within family

- typically 0 (except for raw socket)

Bind Socket

int ***bind*** (int ***sockfd***, struct sockaddr* ***serverAddr***, int ***addrlen***);

bind a socket to a local IP address & port number
returns 0 on success; -1 on failure and sets errno

sockfd : socket descriptor (returned from socket)
serverAddr : includes IP address and port number

- IP address set by kernel if value passed is INADDR_ANY, else set by caller
- port number set by kernel if value passed is 0, else set by caller

addrlen : length of address structure

- sizeof (struct sockaddr_in)

Send Data

int ***sendto*** (int ***sockfd***, char* ***buf***, size_t ***nbytes***, int ***flags***, struct sockaddr* ***to***, int ***addrlen***);

Write data to a datagram socket (UDP)
Returns number of bytes written or -1; also sets errno

sockfd : socket descriptor
buf : data buffer to send from
nbytes : number of bytes to try to write
flags : set to 0 *(for a simple UDP client/server)*
to : socket addr structure (IP & Port) where to send data
addrlen : length of addr structure

Receive Data

int ***recvfrom*** (int ***sockfd***, char* ***buf***, size_t ***nbytes***, int ***flags***, struct sockaddr* ***from***, int* ***addrlen***);

Read data from a datagram socket (UDP)
Returns number of bytes read or -1; also sets errno on

sockfd : socket descriptor
buf : buffer to recv data
nbytes : number of bytes to read
flags : set to 0 *(for a simple UDP client/server)*
from : socket addr structure filled with IP & Port of sender

Close Socket

int ***close*** (int ***sockfd***);

Socket marked as closed, no more read/write calls

Returns 0 on success; -1 on failure & sets errno

sockfd : socket descriptor

Lab Tasks

Task 1. **You are given two c files named client and server. Your task is to run both files and understand the scenario.** **[15 marks]**

Task 2. **Write client server program in which client sends his roll number and server responds with his name.** **[15 marks]**

Task 3. **In Task 1, client and server communicate for short period and then server closes its socket. Modify the above behavior so that the server remains active until forced to terminate.** **[20 marks]**

Note: Client will communicate for a short period and then close its socket as in Task 1.