INDIAN INSTITUTE OF INFORMATION TECHNOLOGY KALYANI
Autonomous institution under MHRD, Govt. Of India
&
Department of Information Technology & Electronics, Govt. of West Bengal
WEBEL IT Park Campus (Near Buddha Park), Kalyani -741235, West Bengal
Tel : 033 2582 2240, website : www.iiitkalyani.ac.in

Lab Assignment #07

Submit on or before 14.03.18

Weekly contact : **0 – 0 – 3 (L – T – P)**
Course No. : **CS 612**
Course Title : **Computer Networks**
Instructor-In-Charge : **Dr. SK Hafizul Islam (hafi786@gmail.com)**

Aim

- To know about how client is communicating with a server.
- SOCKET Programming that enables a node (client) to communicate with another node (server).

Client-Server Model

In Client-Server model, an interprocess communication (IPC), two processes will be communicating with each other. One of the two processes, the client, connects to the other process, the server, typically to make a request for information. In this paradigm, the 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. Once a connection is established, both sides can send and receive information.

Server operation depends on two factors: the transport layer protocol (UDP or TCP) and the service method (Iterative or Concurrent). We can have four types of servers:

- **connectionless iterative (UDP)**,
- **connection-oriented concurrent (TCP),**
- connectionless concurrent (UDP),
- connection-oriented iterative (TCP)

Connectionless Iterative Server

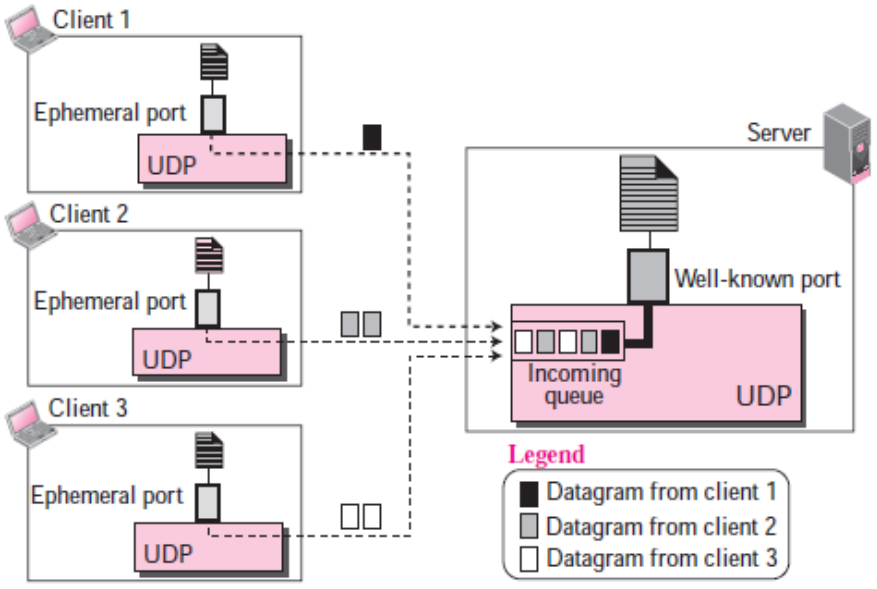


Figure: Connectionless iterative server

The servers that use UDP are normally iterative. The server uses one single port for this purpose, the well-known port. All the datagrams arriving at this port wait in line to be served.

Connection-Oriented Concurrent Server

The servers that use TCP are normally concurrent. This means that the server can serve many clients at the same time. Communication is connection-oriented, which means that a request is a stream of bytes that can arrive in several segments and the response can occupy several segments. A connection is established between the server and each client, and the connection remains open until the entire stream is processed and the connection is terminated.

This type of server cannot use only one port because each connection needs a port and many connections may be open at the same time. Many ports are needed, but a server can use only one well-known port. The solution is to have one well-known port and many ephemeral ports. The server accepts connection requests at the well-known port. **A client can make its initial approach to this port to make the connection. After the connection is made, the server assigns a temporary port to this connection to free the well-known port. Data transfer can now take place between these two temporary ports, one at the client site and the other at the server site.** The well-known port is now free for another client to make the connection. To serve several clients at the same time, a server creates child processes, which are copies of the original process (parent process).

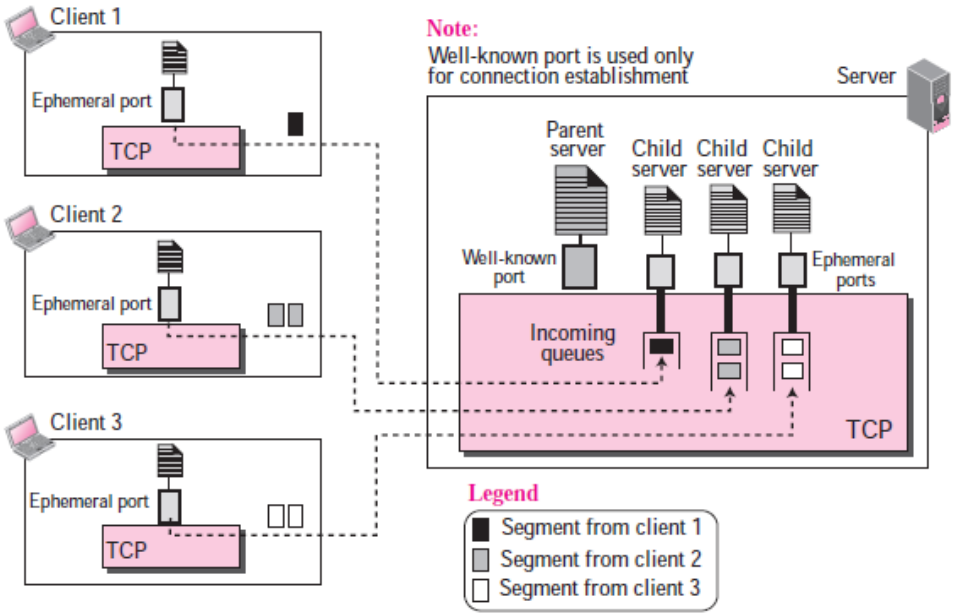
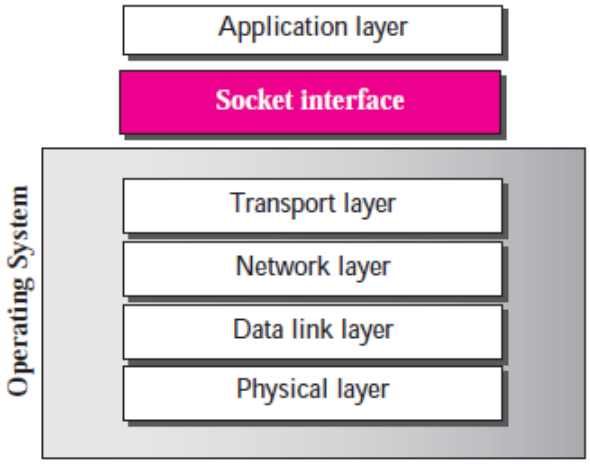


Figure: Connection-oriented concurrent server

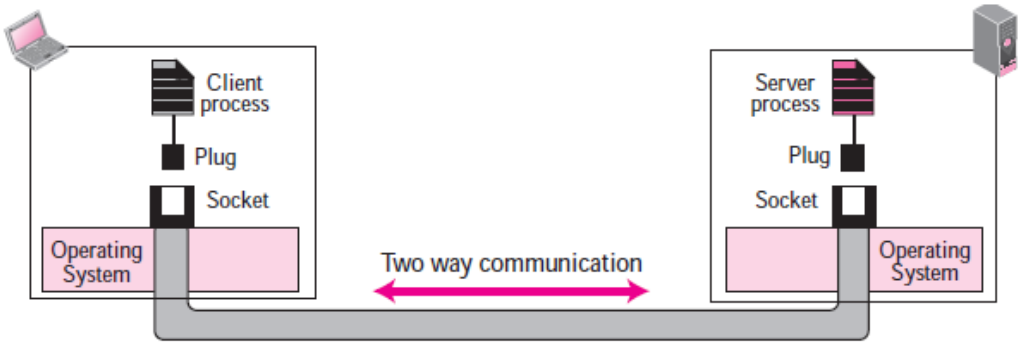
Socket Interfaces

If we need a program to be able to communicate with another program running on another machine, we need a new set of instructions (other than mathematical operations, string manipulation, and input/output access) to tell the transport layer to open the connection, send data to and receive data from the other end, and close the connection. A set of instructions of this kind is normally referred to as an interface. Several interfaces have been designed for communication. Three among them are common: socket interface, transport layer interface (TLI), and STREAM. The socket interface, as a set of instructions, is located between the OS and the application programs. To access the services provided by the TCP/IP protocol suite, an application needs to use the instructions defined in the socket interface.



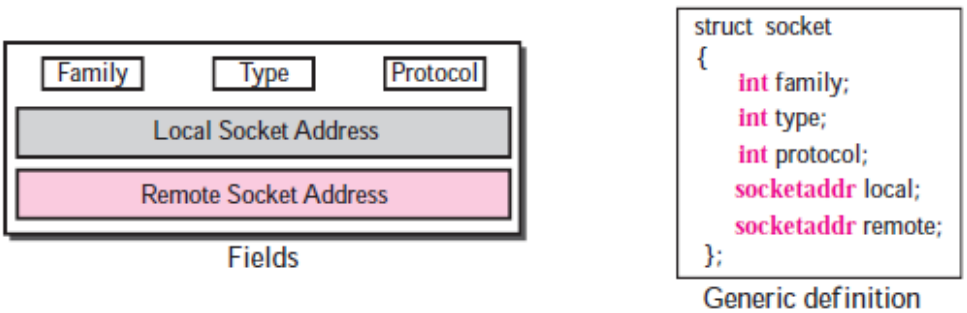
Socket

To use the communication channel, an application program (client/server) needs to request the OS to create a socket. The application program then can plug into the socket to send and receive data. For data communication to occur, a pair of sockets, each at one end of communication, is needed.



Data Structure

The format of data structure to define a socket depends on the underlying language used by the processes. In C language, a socket is defined as a five-field structure.

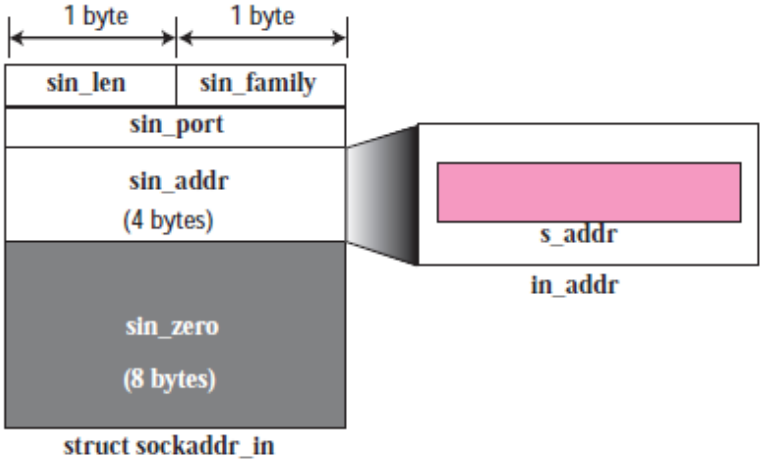


Socket data structure

- **Family:** It defines the protocol group (IPv4, IPv6). The family type we use in TCP/IP is defined by the constant **IF_INET** for IPv4 protocol and IF_INET6 for IPv6 protocol.
- **Type:** It defines four types of sockets: SOCK_STREAM (for TCP), SOCK_DGRAM (for UDP)
- **Protocol:** It defines the protocol that uses the interface. It is set to 0 for TCP/IP protocol suite.
- **Local socket address:** It defines the local socket address (IP address+port number).
- **Remote socket address:** It defines the remote socket address.

Structure of a Socket Address

Several types of socket addresses have been defined in network programming; we define only the one used for IPv4.



IPv4 socket address

The **struct sockaddr_in** has five fields, but the **sin_addr** field is itself a **struct** of type *in_addr* with a one single field *s_addr*.

```
struct   in_addr
{
    in_addr_t          s_addr;          // A 32-bit IPv4 address
}
```

```
struct   sockaddr_in
{
    uint8_t            sin_len;         // length of structure (16 bytes)
    sa_family_t        sin_family;      // set to AF_INET
    in_port_t          sin_port;        // A 16-bit port number
    struct in_addr     sin_addr;        // A 32-bit IPv4 address
    char               sin_zero[8];     // unused
}
```

Functions

The interaction between a process and the operating system is done through a list of predefined functions

- **socket():** The process needs to use the socket function call to create a socket.
  int socket(int family, int type, int protocol)---> A call to this function creates a socket, but only three fields in the socket structure (family, type, and protocol) are filled. If the call is successful, the function returns a unique socket descriptor **sockfd** (a non-negative integer) that can be used to refer to the socket in other calls; if the call is not successful, the operating system returns −1.

- **bind():**To bind the socket to the local computer and local port, the bind() function needs to be called.

int bind(int sockfd, const struct sockaddress* localAddress, socklen_t addrLen)--->This function fills the value for the local socket address (local IP address and local port number). It returns −1 if the binding fails. **sockfd** is the value of the socket descriptor returned from the socket() function call, localAddress is a pointer to a socket address that needs to have been defined (by the system or the programmer), and the addrLen is the length of the socket address.

- **connect():** The connect function is used to add the remote socket address to the socket structure. It returns −1 if the connection fails.
  int connect (int sockfd, const struct sockaddress* remoteAddress, socklen_t addrLen)

- **listen():** The listen function is called only by the TCP server. After TCP has created and bound a socket, it must inform the OS that a socket is ready for receiving client requests. This is done by calling the listen() function.
  int listen(int sockfd, int backlog)---->The backlog is the maximum number of connection requests. The function returns −1 if it fails.

- **accept():** It is used by a server to inform TCP that it is ready to receive connections from clients. This function returns −1 if it fails. Its prototype is shown
  int accept(int sockfd, const struct sockaddress* clientAddr, socklen_t* addrLen)-->The accept() function is a blocking function that, when called, blocks itself until a connection is made by a client. The accept() function then gets the client socket address and the address length and passes it to the server process to be used to access the client.
  - ✓ The call to accept function makes the process check if there is any client connection request in the waiting buffer. If not, accept() makes the process to sleep. The process wakes up when the queue has at least one request.
  - ✓ After a successful call to the accept(), a new socket is created and the communication is established between the client socket and the new socket of the server.
  - ✓ The address received from the accept function fills the remote socket address in the new socket.
  - ✓ The address of the client is returned via a pointer.
  - ✓ The length of address to be returned is passed to the function and also returned via a pointer.
- **send() and recv():** The send function is used by a process to send data to another process running on a remote machine. The recv()is used by a process to receive data from another process running on a remote machine. These functions assume that there is already an open connection between two machines; therefore, it can only be used by TCP. These functions return the number of bytes sends or receive.
  int send (int sockfd, const void* sendbuf, int nbytes, int flags);
  int recv (int sockfd, void* recvbuf, int nbytes, int flags);
  Here sockfd is the socket descriptor; sendbuf is a pointer to the buffer where data to be sent have been stored; recvbuf is a pointer to the buffer where the data received is to be stored. nbytes is the size of data to be sent or received. This function returns the number of actual bytes sent or received if successful and –1 if there is an error.
- **sendto() and recvfrom():** The sendto function is used by a process to send data to a remote process using the services of UDP. The recvfrom function is used by a process to receive data from a remote process using the services of UDP. Since UDP is a connectionless protocol, one of the arguments defines the remote socket address (destination or source).
  int sendto(int sockfd, const void* buffer, int nbytes, int flags, struct sockaddr* destinationAddress, socklen_t addrLen)
  int recvfrom(int sockfd, void* buffer, int nbytes, int flags, struct sockaddr* sourceAddress, socklen_t* addrLen);
- Here sockfd is the socket descriptor, buffer is a pointer to the buffer where data to be sent or to be received is stored, and buflen is the length of the buffer. These functions return the number of bytes sent or received if successful and –1 if there is an error.

- **close():** It is used by a process to close a socket.
  int close (int sockfd)-->The sockfd is not valid after calling this function. The socket returns an integer, 0 for success and –1 for error.

Information in a computer is stored in host byte order, which can be little-endian, in which the little-end byte (least significant byte) is stored in the starting address, or big-endian, in which the big-end byte (most significant byte) is stored in the starting address. However, network programming needs data and other pieces of information to be in big-endian format. Since when we write programs, we are not sure, how the information such as IP addresses and port number are stored in the computer, we need to change them to big-endian format. Two functions are designed for this purpose:

- htons (host to network short)→ It changes a short (16-bit) value to a network byte order (big-endian)
- htonl (host to network long)→ It changes a long (32-bit) value to a network byte order (big-endian)

There are also two functions that do exactly the opposite: ntohs and ntohl.

- uint16_t htons (uint16_t shortValue);
- uint32_t htonl (uint32_t longValue);
- uint16_t ntohs (uint16_t shortValue);
- uint32_t ntohl (uint32_t longValue);

Memory Management Functions

We need some functions to manage values stored in the memory.

- **void *memset (void* destination, int chr, size_t len);**
  memset (memory set) is used to set (store) a specified number of bytes (value of len) in the memory defined by the destination pointer (starting address).
- **void *memcpy (void* destination, const void* source, size_t nbytes);**
  memcpy (memory copy) is used to copy a specified number of bytes (value of nbytes) from part of a memory (source) to another part of memory (destination).
- **int memcmp (const void* ptr1, const void* ptr2, size_t nbytes);**
  memcmp (memory compare), is used to compare two sets of bytes (nbytes) starting from ptr1 and ptr2. The result is 0 if two sets are equal, less than zero if the first set is smaller than the second and greater than zero if the first set is larger than the second.

Address Conversion Functions

We normally like to work with 32-bit IP address in dotted decimal notation. When we want to store the address in a socket, however, we need to change it to a number. Two functions are used to convert an address from a presentation to a number and vice versa.

int inet_pton (int family, const char* stringAddr, void* numericAddr); [presentation to number]
char* inet_ntop (int family, const void* numericAddr, char* stringAddr, int len); [number to presentation]

Required Header Files

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <errno.h>
#include <signal.h>
#include <unistd.h>
#include <string.h>
#include <arpa/innet.h>
#include <sys/wait.h>
```

Assignment

Write two Socket-based C/C++ programs one for server and another for client based on the connectionless iterative communication using UDP. In this programming environment, the client sends a line of text to the server and the server sends the same line back to the client.

Activities of the Server:

1) Create a socket with the socket()
2) Bind the socket to an address using the bind().
3) Send and receive data, use the recvfrom() and sendto() system calls.

Activities of the Client:

1) Create a socket with the socket()
2) Send and receive data, use the recvfrom() and sendto() system calls.