CMR
INSTITUTE OF
TECHNOLOGY

USN | | | | | | | | | | |

Internal Assesment Test - I

Sub:	Client Server Programming							Code:	14SCN41
Date:	27/03/2017	Duration:	90 mins	Max Marks:	50	Sem:	IV	Branch:	Mtech(CNE)

Answer Any FIVE FULL Questions

		Marks	OBE	
			CO	RBT
1	Explain any five terminology and concepts of client server model.	[10]	CO1	L1
2.	Discuss the concurrency in Network and Server with the help of neat diagram	[10]	CO1	L1
3.a	Explain the concept of time slicing with suitable C program.	[06]	CO1	L1
3.b	What are two basic approaches for Network communication?	[04]	CO2	L2
4	Explain the major system calls for socket.	[10]	CO2	L2
5.a	How domain name is mapped into IP address? Explain briefly	[05]	CO2	L2
5.b	How to identify port number by service name? Explain Briefly	[05]	CO2	L2
6.a	What is the difference between connected and unconnected UDP socket? .	[05]	CO2	L2
6.b	How to close a UDP connection? Explain briefly	[05]	CO2	L2

Course Outcomes		PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
CO1:	Describe the Client Server Model, Concurrency in Client Server Software, Protocol Interface and basic system calls in UNIX	0	0	0	0	0	0	0	0	0	0	0	0
CO2:	Explain the Berkeley Socket interface, System calls for designing the client Software	0	0	1	0	0	1	0	0	0	0	0	0
CO3:	Programming the client software for Daytime, Time and Echo service	2	0	2	2	0	0	0	0	0	0	0	0
CO4:	Differentiate between different types of connection and servers.	0	0	0	0	0	0	0	0	0	0	0	0
CO5:	Programming the server software for Daytime, Time and Echo service	2	0	2	2	0	0	0	0	0	0	0	0

Cognitive level	KEYWORDS
L1	List, define, tell, describe, identify, show, label, collect, examine, tabulate, quote, name, who, when, where, etc.
L2	summarize, describe, interpret, contrast, predict, associate, distinguish, estimate, differentiate, discuss, extend
L3	Apply, demonstrate, calculate, complete, illustrate, show, solve, examine, modify, relate, change, classify, experiment, discover.
L4	Analyze, separate, order, explain, connect, classify, arrange, divide, compare, select, explain, infer.
L5	Assess, decide, rank, grade, test, measure, recommend, convince, select, judge, explain, discriminate, support, conclude, compare, summarize.

PO1 - *Engineering knowledge*; PO2 - *Problem analysis*; PO3 - *Design/development of solutions*; PO4 - *Conduct investigations of complex problems*; PO5 - *Modern tool usage*; PO6 - *The Engineer and society*; PO7- *Environment and sustainability*; PO8 – *Ethics*; PO9 - *Individual and team work*; PO10 - *Communication*; PO11 - *Project management and finance*; PO12 - *Life-long learning*

#132, AECS Layout, IT Park Road, Kundalahalli, Bangalore – 560 037
T:+9180 28524466 / 77

CMR INSTITUTE
OF TECHNOLOGY

Scheme and Solution – (IAT 1)

Department of Computer Science and Engineering

Client Server Programming (14SCN41)

Scheme

Q.1 Explanation on five terminologies of Client Server Model each carrying 2 marks (2*5=10 marks).

Q.2 Explanation of concurrency in network(5 marks)
Explanation of concurrency in server (5 marks)

Q.3 A. Time Slicing Concept explanation(2 marks)
Program (4 marks)
B. Explanation of two basic approaches for network communication each carrying 2 marks (2*2= 4 marks)

Q.4 Explanation of all major 8 socket calls(10 marks)

Q.5 A. Explanation of How domain name is mapped into IP address (2 marks)
Pseudo code (3 marks)
B. Explanation of How to identify port number by service name (2 marks)
Pseudo code (3 marks)

Q.6 A. Difference between connected and unconnected socket(5 marks)
Explanation on shutdown and close call (5 marks)

Solution

Q.1 Explain any five terminology and concepts of client server model.

Clients and Servers

In general, an application that initiates peer-to-peer communication is called a *client.* End users usually invoke client software when they use a network service. Most client software consists of conventional application programs. Each time a client application executes, it contacts a server, sends a request, and awaits a response. When the response arrives, the client continues processing. Clients are often easier to build than servers, and usually require no special system privileges to operate. By comparison, a *server is* any program1 that waits for incoming communication requests from a client. The server receives a client's request, performs the necessary computation, and returns the result to the client.

Privilege and Complexity

Because servers often need to access data, computations, or protocol ports that the operating system protects, server software usually requires special system privileges. Because a server executes with special system privilege, care must be taken to ensure that it does not inadvertently pass privileges on to the clients that use it.

Servers must contain code that handles the issues of:

Authentication - verifying the identity of the client

Authorization - determining whether a given client is permitted to access the service the server supplies

Data security - guaranteeing that data is not unintentionally revealed or compromised

Privacy - keeping information about an individual from unauthorized access

Protection - guaranteeing that network applications cannot abuse system resources.

Parameterization of Clients

Some client software provides more generality than others. In particular, some client software allows the user to specify both the remote machine on which a server operates and the protocol port number at which the server is listening.
Conceptually, software that allows a user to specify a protocol port number has more input parameters than other software, so we use the term *fully parameterized client* to describe it. Many TELNET client implementations interpret an optional second argument as a port number. To specify only a remote machine, the user supplies the name of the remote machine:

telnet *machine-name*

Given only a machine name, the *telnet* program uses the well-known port for the TELNET service. To specify both a remote machine and a port on that machine, the user specifies both the machine name and the port number:

telnet *machine-name port*

Full parameterization is especially useful when testing a new client or server because it allows testing to precede independent of the existing software already in use. For example, a programmer can build a TELNET client and server pair, invoke them using nonstandard protocol ports, and proceed to test the software without disturbing standard services. Other users can continue to access the old TELNET service without interference during the testing.

Standard Vs. Nonstandard Client Software

Standard application services consist of those services defined by TCP/IP and assigned well-known, universally recognized protocol port identifiers; we consider all others to be *locally-defined application services* or *nonstandard application services.*
The distinction between standard services and others is only important when communicating outside the local environment.

Although TCP/IP defines many standard application protocols, most commercial computer vendors supply only a handful of standard application client programs with their TCP/IP software. For example, TCP/IP software usually includes a *remote terminal client* that uses the standard TELNET protocol for remote login, an *electronic mail client* that uses the standard SMTP protocol to transfer electronic mail to a remote system, a file *transfer client that* uses the standard FTP protocol to transfer files between two machines, and a *Web browser* that uses the standard HTTP protocol to access Web documents.
Of course, many organizations build customized applications that use TCP/IP to communicate. Customized, nonstandard applications range from simple to complex, and include such diverse services as image transmission and video teleconferencing,

Stateless Vs. Stateful Servers

Information that a server maintains about the status of ongoing interactions with clients is called *state information.* Servers that do not keep any state information are called *stateless servers;* others are called *stateful servers*.
Te desire for efficiency motivates designers to keep state information in servers. Keeping a small amount of information in a server can reduce the size of messages that the client and server exchange, and can allow the server to respond to requests quickly. Essentially, state information allows a server to remember what the client requested previously and to compute an incremental response as each new request arrives. By contrast, the motivation for statelessness lies in protocol reliability: state information in a server can become incorrect if messages are lost, duplicated, or delivered out of order, or if the client computer crashes and reboots. If the server uses incorrect state information when computing a response, it may respond incorrectly.

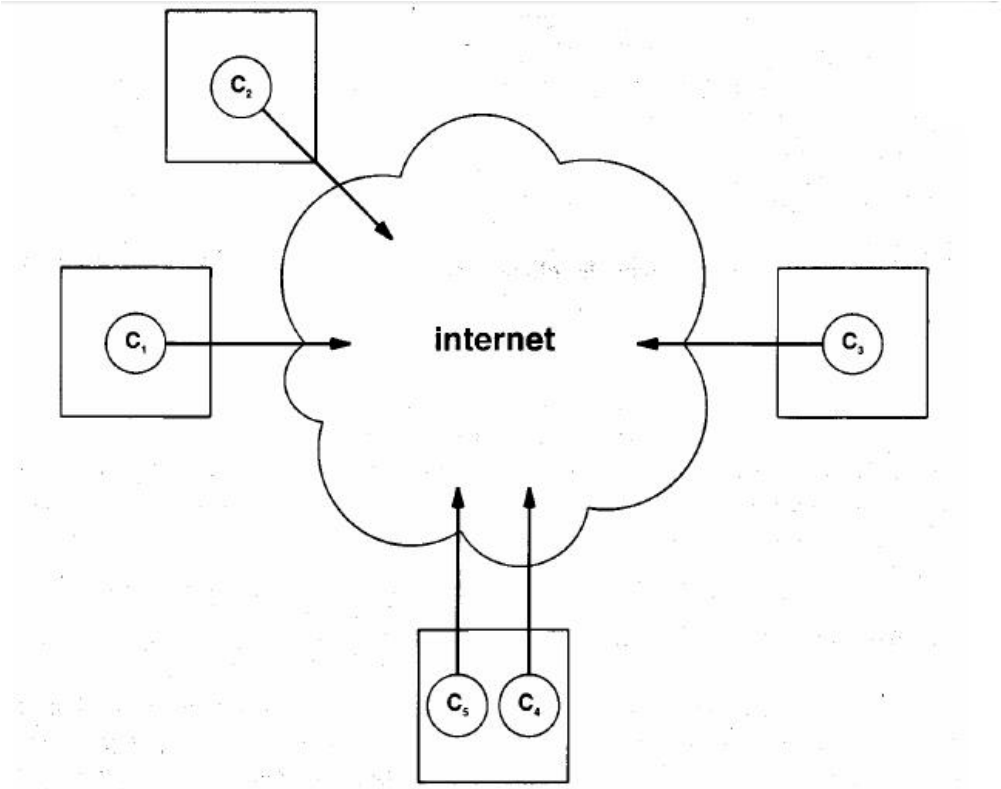
Q.2 Discuss the concurrency in Network and Server with the help of neat diagram

Concurrency in Network

- The term *concurrency* refers to real or apparent simultaneous computing.
- Concurrent processing is fundamental to distributed computing and occurs in many forms.
- Among machines on a single network, many pairs of application programs can communicate concurrently, sharing the network that interconnects them. For example,

application A on one machine may communicate with application B on another machine, while application C on a third machine communicates with application D on a fourth. Although they all share a single network, the applications appear to proceed as if they operate independently. The network hardware enforces access rules that allow each pair of communicating machines to exchange messages. The access rules prevent a given pair of applications from excluding others by consuming all the network bandwidth

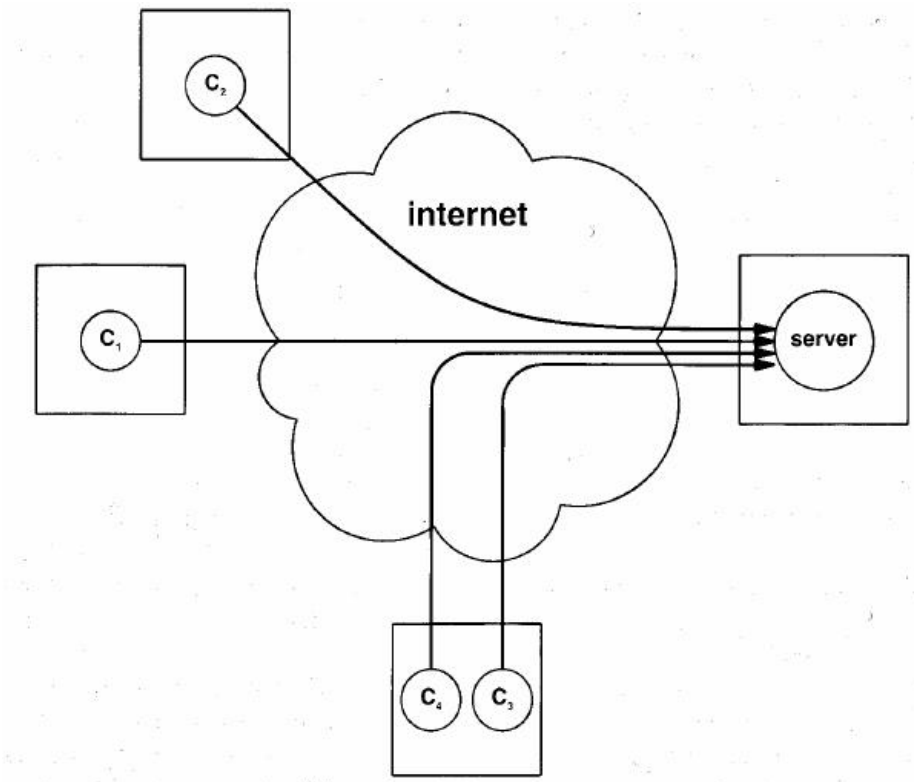
- In addition to concurrency among clients on a single machine, the set of all clients on a set of machines can execute concurrently. Figure illustrates concurrency among client programs running on several machines.



Concurrency among client programs occurs when users execute them on multiple machines simultaneously or when a multitasking operating system allows multiple copies to execute concurrently on a single computer.

Concurrency in Server

- In contrast to concurrent client software, concurrency within a server requires considerable effort. As figure shows, a single server program must handle incoming requests concurrently.



Server software must be explicitly programmed to handle concurrent requests because multiple clients contact a server using its single, well-known protocol port.

To understand why concurrency is important, consider server operations that require substantial computation or communication. For example, think of a remote login server. It operates with no concurrency; it can handle only one remote login at a time. Once a client contacts the server, the server must ignore or refuse subsequent requests until the first user finishes. Clearly, such a design limits the utility of the server, and prevents multiple remote users from accessing a given machine at the same time.

Q.3 A. Explain the concept of time slicing with C program

- The term *time slicing* describes systems that share the available CPU among several processes concurrently. For example, if a time slicing system has only one CPU to allocate and a program divides into two processes, one of the processes will execute for a while, then the second will execute for a while, then the first will execute again, and so on. If the time slicing system has many processes, it runs each for a short time before it runs the first one again.
- A time slicing mechanism attempts to allocate the available processing equally among all available processes. If only two processes are eligible to execute and the computer has a single processor, each receives approximately 50% of the CPU. If N processes are

eligible on a computer with a single processor, each receives approximately 1/N of the CPU. Thus, all processes appear to proceed at an equal rate, no matter how many processes execute. With many processes executing, the rate is low; with few, the rate is high.

Program

```
#include <stdlib.h>

#include <stdio.h>

int sum;

main() {

int i;

sum = 0;

fork();

for (i=1 ; i <=10000 ; i++) {

printf("The value of i is %d\n", i);

fflush(stdout);

sum += i;

}

printf ("The total is %d\n", sum);

exit (0)

}
```

- When the resulting concurrent program is executed on the same system as before, it emits 20,002 lines of output. However, instead of all output from the first process followed by all output from the second process, output from both processes is mixed together. In one run, the first process iterated 74 times before the second process executed at all. Then the second process iterated 63 times before the system switched back to the first process.

Q.3 B. What are the two basic approaches for network communication?

The operating system designer follows two approaches to define the system calls when the TCP/IP protocol software is installed.

- The designer invents entirely new system calls that applications use to access TCP/IP. The designer itself defines the names and parameters for each system call, and this approach is seldom used.
- The designer attempts to use basic conventional I/O system calls to access TCP/IP. The designer overloads system calls so that they work with network protocols as well as conventional I/O devices.

Q.4 Explain the major system calls for socket.

Major System Calls Used With Sockets

The socket provides two types of system calls. Primary calls that provide access to the underlying functionality and utility routines that help the programmer. A socket can be used by a client or by a server, for stream transfer (TCP) or datagram (UDP) communication, with a specific remote endpoint address (usually needed by a client) or with an unspecified remote endpoint address (usually needed by a server).

1. The Socket Call

It creates a new socket for network communication and returns a socket descriptor. This call also returns -1 in case of error. The arguments to this call are

- Protocol family (Ex. *PF_INET* for TCP/IP).
- Type of service it needs (i.e. SOCK_STREAM for TCP or SOCK_DGRAM for UDP).
- Protocol number to use. It uses 0 to specify the default protocol for given family and type.

Syntax: retcode = socket (family, type, protocol);

2. Connect Call

A client calls *connect* to establish an active connection to a remote server. If the socket uses TCP, connect uses the 3-way handshake to establish connection. The arguments to this call are

- Socket descriptor.
- remote machine's IP address
- protocol port number

- Length of address.

Once a connection has been made, a client can transfer data across it.

Syntax: retcode = connect (socket descriptor, addr, length);

3. Write Call

Both client and server use write to send the data. Client will send the request and server will send the response. This call also returns -1 in case of error .The arguments to this call are

- Socket descriptor
- The address of buffer containing data
- Number of bytes in buffer

Usually, *write* copies outgoing data into buffers in the operating system kernel, and allows the application to continue execution while it transmits the data across the network. If the system buffers become full, the call to *write* may block temporarily until TCP can send data across the network and make space in the buffer for new data.

Syntax: retcode = write (socket descriptor, buffer, length);

4. Read Call

Both clients and servers use *read* to receive data from a TCP connection. Usually, after a connection has been established, the server uses *read* to receive a request that the client sends by calling *write.* After sending its request, the client uses *read* to receive a reply. The arguments to this call are

- Socket descriptor
- The address of buffer which will receive data
- Number of bytes in buffer
- Flags: control bits that specify whether to receive out of band data and whether to look ahead for messages.

Read extracts data bytes that have arrived at that socket, and copies them to the user's buffer area. If no data has arrived, the call to *read* blocks until it does. If more data has arrived than fits into the buffer, *read* only extracts enough to fill the buffer. If less data has arrived than fits into the buffer, *read* extracts all the data and returns the number of bytes it found. But in case of UDP connection if the buffer cannot hold the entire message, it fills the buffer with respect to its capacity and discards the remainder.

Syntax: retcode= read (socket descriptor, buffer, length, flags)

5. Close Call

Once a client or server finishes using a socket, it calls *close* to deallocate it. If only one process is using the socket, *close* immediately terminates the connection and deallocates the socket. If n processes share a socket, the reference count will be n. Each time any process calls the close it decrements the reference count. Once the reference count is zero the socket will be deallocated.
Syntax: retcode= close (socket descriptor);

6. The Bind Call

When a socket is created, it does not have any notion of endpoint addresses (neither the local nor remote addresses are assigned). An application calls *bind* to specify the local endpoint address for a socket. The call takes arguments that specify a socket descriptor and an endpoint address. For TCP/IP protocols, the endpoint address uses the *sockaddr_in* structure, which includes both an IP address and a protocol port number. Primarily, servers use *bind* to specify the well-known port at which they will await connections.

7. The Listen Call

When a socket is created, the socket is neither *active (i.e.,* ready for use by a client) nor *passive (i.e.,* ready for use by a server) until the application takes further action. Connection-oriented servers call *listen* to place a socket in *passive mode* and make it ready to accept incoming connections. Most servers consist of an infinite loop that accepts the next incoming connection, handles it, and then returns to accept the next connection. Even if handling a given connection takes only a few milliseconds, it may happen that a new connection request arrives during the time the server is busy handling an existing request. To ensure that no connection request is lost, a server must pass *listen* an argument that tells the operating system to enqueue connection requests for a socket. Thus, one argument to the *listen* call specifies a socket to be placed in passive mode, while the other specifies the size of the queue to be used for that socket.

8. The Accept Call

For TCP sockets, after a server calls *socket* to create a socket, *bind* to specify a local endpoint address, and *listen* to place it in passive mode, the server calls *accept* to extract the next incoming connection request. An argument to *accept* specifies the socket from which a

connection should be accepted. *Accept* creates a new socket for each new connection request, and returns the descriptor of the new socket to its caller. The server uses the new socket only for the new connection; it uses the original socket to accept additional connection requests

Q.5 A. How domain name is mapped into IP address? Explain briefly

- A client must specify the address of a server using structure *sockaddr_in.* Doing so means converting an address in dotted decimal notation (or a domain name in text form) into a 32-bit IP address represented in binary. Converting from dotted decimal notation to binary is trivial. Converting from a domain name, however, requires considerably more effort.
- *Gethostbyname* takes an ASCII string that contains the domain name for a machine. It returns the address of a *hostent* structure that contains, among other things, the host's IP address in binary. The *hostent* structure is declared in include file *netdb.h*

```
struct hostent {
char      *h_name;              /* official host name */
char     **h_aliases;           /* other aliases      */
int        h_addrtype;          /* address type       */
int        h_length;            /* address length     */
char     **h_addr_list;         /* list of addresses  */
};
#define    h_addr h_addr_list[0]
```

- Consider a simple example of name conversion. Suppose a client has been passed the domain name *merlin.cs.purdue.edu* in string form and needs to obtain the IP address. The client can call *gethostbyname* as in

```
struct hostent *hptr;
char *examplenam = "merlin.cs.purdue.edu";

if ( hptr = gethostbyname( examplenam ) ) {
    /* IP address is now in hptr->h_addr */
} else {
    /* error in name - handle it */
}
```

Q.5 B. How to identify port number by service name? Explain Briefly

- Most client programs must look up the protocol port for the specific service they wish to invoke. For example, a client of an SMTP mail server needs to look up the well-known port assigned to SMTP. To do so, the client invokes library function *getservbyname,* which takes two arguments: a string that specifies the desired service and a string that specifies the protocol being used. It returns a pointer to a structure of type *servent,* also defined in include file *netdb.h:*

```
struct servent {
char *s_name;          /* official service name */
char **s_aliases;      /* other aliases         */
int sort;              /* port for this service */
char *s_proto;         /* protocol to use       */
```

- If a TCP client needs to look up the official protocol port number for SMTP, it calls *getservbyname,* as in the following example:

```
struct servent *sptr;
if (sptr = getservbyname( "smtp", "tcp" )) {
      /* port number is now in sptr->s_port */
} else {
      /* error occurred - handle it */
}
```

Q.6 A. What is difference between connected and unconnected UDP socket

- Client applications can use UDP in one of two basic modes: *connected* and *unconnected.*
- In connected mode, the client uses the *connect* call to specify a remote endpoint address (i.e., the server's IP address and protocol port number). Once it has specified the remote endpoint, the client can send and receive messages much like a TCP client does.
- In unconnected mode, the client does not connect the socket to a specific remote endpoint. Instead, it specifies the remote destination each time it sends a message.
- The chief advantage of connected UDP sockets lies in their convenience for conventional client software that interacts with only one server at a time: the application only needs to specify the server once no matter how many datagrams it sends.

- The chief advantage of unconnected sockets lies in their flexibility; the client can wait to decide which server to contact until it has a request to send. Furthermore, the client can easily send each request to a different server.

Q.6 B. How to close a UDP connection? Explain Briefly

Closing A Socket That Uses UDP

- A UDP client calls *close* to close a socket and release the resources associated with it. Once a socket has been closed, the UDP software will reject further messages that arrive addressed to the protocol port that the socket had allocated.
- However, the machine on which the *close* occurs does not inform the remote endpoint that the socket is closed.
- Therefore, an application that uses connectionless transport must be designed so the remote side knows how long to retain a socket before closing it.

Partial Close for UDP

- *Shutdown* can be used with a connected UDP socket to stop further transmission in a given direction.
- Unfortunately, unlike the partial close on a TCP connection, when applied to a UDP socket, *shutdown* does not send any messages to the other side. Instead, it merely marks the local socket as unwilling to transfer data in the direction specified.
- Thus, if a client shuts down further output on its socket, the server will not receive any indication that the communication has ceased.