CMR
INSTITUTE OF
TECHNOLOGY

USN

Internal Assesment Test - II

Sub:	Client Server Programming							Code:	14SCN41
Date:	12/05/2017	Duration:	90 mins	Max Marks:	50	Sem:	IV	Branch:	Mtech(CNE)

Answer Any FIVE FULL Questions

		Marks	OBE	
			CO	RBT
1A	What is the importance of procedure for Programming? How the procedure library is designed for client programs.	[04]	CO3	L2
B	Explain the implementation details of ConnectTCP and ConnnectUDP.	[06]	CO3	L2
2.	What are Daytime Service and Time Service? Explain the difference between Daytime Service and Time Service.	[10]	CO3	L2
3	What is Echo Service? Describe the problem of server deadlock.	[10]	CO3	L2
4	Write a program for procedure which forms the connection to the server.	[10]	CO3	L3
5	Write a client side program which accesses the Daytime Service.	[10]	CO3	L3
6	Describe algorithm for iterative Connection oriented server algorithm and concurrent connectionless server algorithm. .	[10]	CO4	L1
7	How the performance of stateless server can be optimized? Explain briefly.	[10]	CO4	L1

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*

Scheme and Solution – (IAT II)

Department of Computer Science and Engineering

Client Server Programming (14SCN41)

Q.1A. What is the importance of procedure for Programming? How the procedure library is designed for client programs.

Importance of procedure for programming

- Most programmers understand the advantage of dividing large, complex programs into a set of procedures: a modular program becomes easier to understand, debug, and modify than an equivalent monolithic program.
- If programmers design procedures carefully, they can reuse them in other programs. Finally, choosing procedures carefully can also make a program easier to port to new computer systems.
- Conceptually, procedures raise the level of the language that programmers use by hiding details.
- Using procedures helps avoid repetition by providing higher-level operations. Once a particular algorithm has been encoded in a procedure, the programmer can use it in many programs without having to consider the implementation details again.
- A careful use of procedures is especially important when building client and server programs. First, because network software includes declarations for items like endpoint addresses, building programs that use network services involves a myriad of tedious details not found in conventional programs. Using procedures to hide those details reduces the chance for error. Second, much of the code needed to allocate a socket, bind addresses, and form a network connection is repeated in each client; placing it in procedures allows programmers to reuse the code instead of replicating it. Third, because TCP/IP was designed to interconnect heterogeneous machines, network applications often operate on many different machine architectures. Programmers can use procedures to isolate operating system dependencies, making it easier to port code to a new machine.

An Example Procedure Library for Client Programs

The first step of designing a procedure library is abstraction: a programmer must imagine high-level operations that would make writing programs simpler. For example, an application programmer might imagine two procedures that handle the work of allocating and connecting a socket:

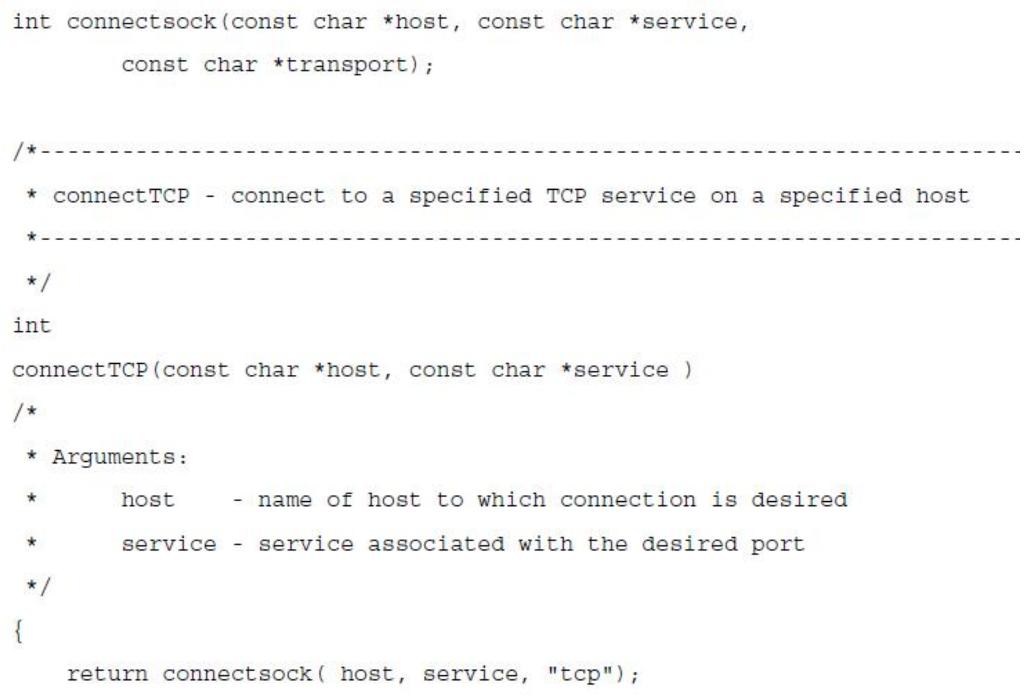
socket = connectTCP(*machine*, *service);*

and

socket = connectUDP(*machine, service);*

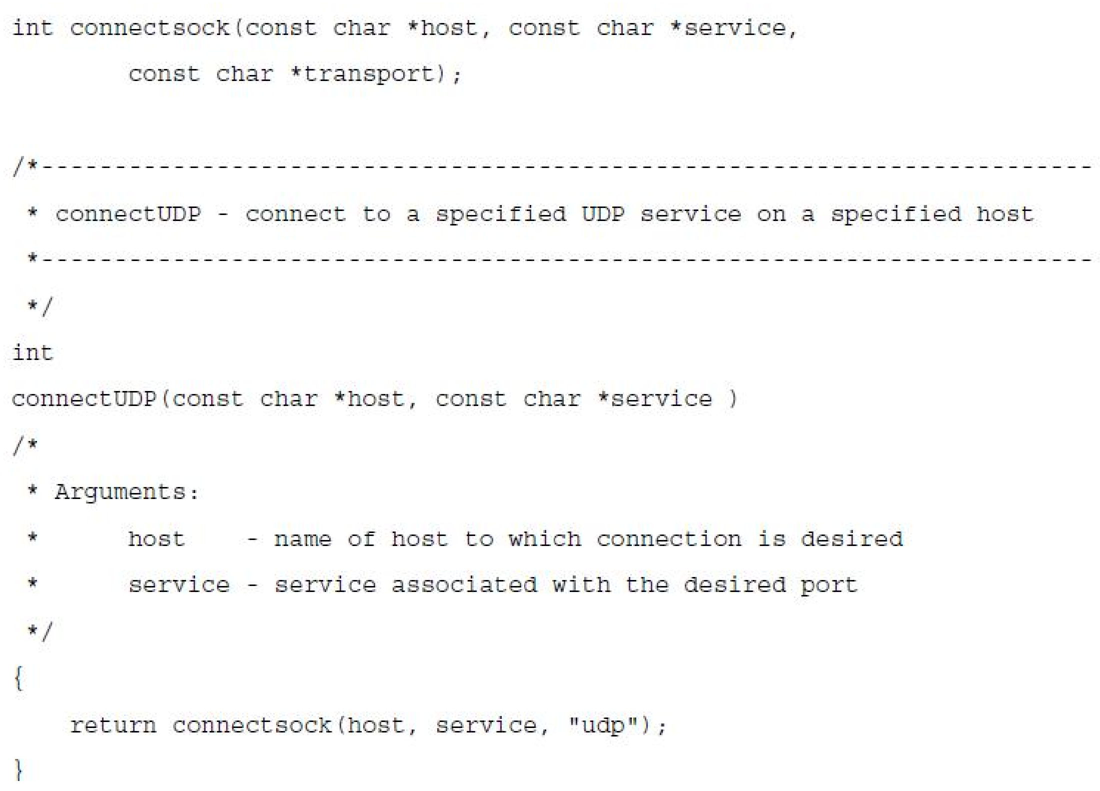
Q.1.B. Explain the implementation details of ConnectTCP and ConnnectUDP.

Implementation of ConnectTCP



```
int connectsock(const char *host, const char *service,
        const char *transport);

/*------------------------------------------------------------------------
 * connectTCP - connect to a specified TCP service on a specified host
 *------------------------------------------------------------------------
 */
int
connectTCP(const char *host, const char *service )
/*
 * Arguments:
 *      host    - name of host to which connection is desired
 *      service - service associated with the desired port
 */
{
    return connectsock( host, service, "tcp");
```

Implementation of ConnectUDP



```
int connectsock(const char *host, const char *service,
        const char *transport);

/*------------------------------------------------------------------------
 * connectUDP - connect to a specified UDP service on a specified host
 *------------------------------------------------------------------------
 */
int
connectUDP(const char *host, const char *service )
/*
 * Arguments:
 *      host    - name of host to which connection is desired
 *      service - service associated with the desired port
 */
{
    return connectsock(host, service, "udp");
}
```

Q.2 What are Daytime Service and Time Service? Explain the difference between Daytime Service and Time Service

Daytime Service

- The TCP/IP standards define an application protocol that allows a user to obtain the date and time of day in a format fit for human consumption. The service is officially named the *DAYTIME service.*
- To access the DAYTIME service, the user invokes a client application. The client contacts a server to obtain the information, and then prints it.

  For example,

  DAYTIME could supply a date in the form:

weekday, month day, year time-timezone

like

Thursday, February 22, 1996 17:37:43-PST

- The standard specifies that DAYTIME is available for both TCP and UDP. In both cases, it operates at protocol port 13.

- The TCP version of DAYTIME uses the presence of a TCP connection to trigger output: as soon as a new connection arrives, the server forms a text string that contains the current date and time, sends the string, and then closes the connection. Thus, the client need not send any request at all. In fact, the standard specifies that the server must discard any data sent by the client.
- The UDP version of DAYTIME requires the client to send a request. A request consists of an arbitrary UDP datagram. Whenever a server receives a datagram, it formats the current date and time, places the resulting string in an outgoing datagram, and sends it back to the client. Once it has sent a reply, the server discards the datagram that triggered the response.

Time Service

- TCP/IP defines a service that allows one machine to obtain the current date and time of day from another. Officially named *TIME,* the service is quite simple: a client program executing on one machine sends a request to a server executing on another.
- Whenever the server receives a request, it obtains the current date and time of day from the local operating system, encodes the information in a standard format, and sends it back to the client in a response.
- To avoid the problems that occur if the client and server reside in different time zones, the TIME protocol specifies that all time and date information must be represented in *Universal Coordinated Time3,* abbreviated UCT or UT.
- Thus, a server converts from its local time to universal time before sending a reply, and a client converts from universal time to its local time when the reply arrives.
- Unlike the DAYTIME service, which is intended for human users, the TIME service is intended for use by programs that store or manipulate times. The TIME protocol always specifies time in a 32-bit integer, representing the number of seconds since an *epoch date.* The TIME protocol uses midnight, January l, 1900, as its epoch.
- Using an integer representation allows computers to transfer time from one machine to another quickly, without waiting to convert it into a text string and back into an integer. Thus, the TIME service makes it possible for one computer to set its timeofday clock from the clock on another system.

Q.3 What is Echo Service? Describe the problem of server deadlock.

Echo Service

- TCP/IP standards specify an *ECHO service* for both UDP and TCP protocols.
- ECHO server merely returns all the data it receives from a client. Despite their simplicity, ECHO services are important tools that network managers use to test reachability, debug protocol software, and identify routing problems.

- The TCP ECHO service specifies that a server must accept incoming connection requests, read data from the connection, and write the data back over the connection until the client terminates the transfer. Meanwhile, the client sends input and then reads it back.

Server Deadlock

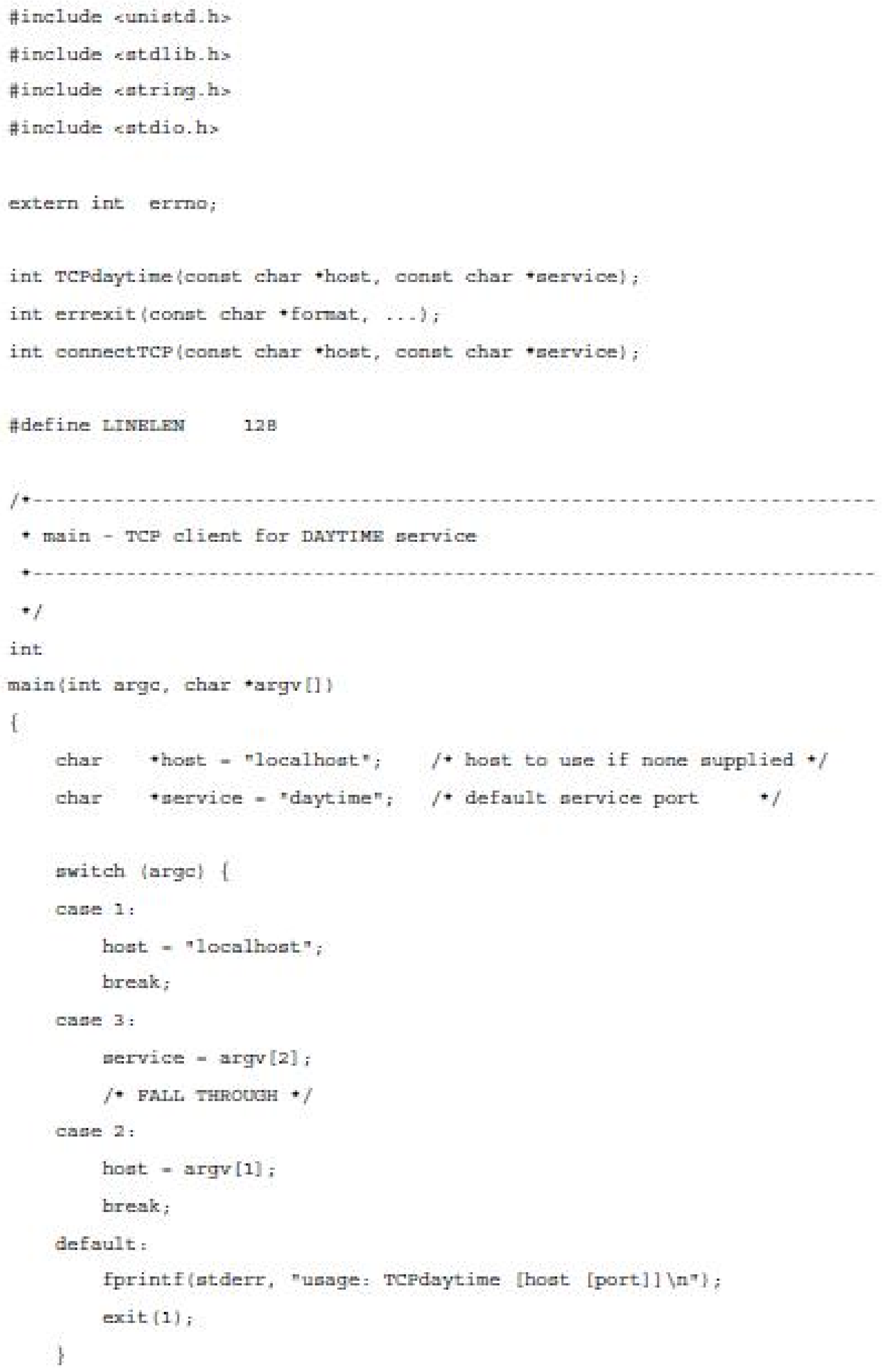
- To understand how deadlock can happen, consider an iterative, connection-oriented server. Suppose some client application, *C*, misbehaves. In the simplest case, assume C makes a connection to a server, but never sends a request. The server will accept the new connection, and call read to extract the next request. The server process blocks in the call to *read* waiting for a request that will never arrive.
- Server deadlock can arise in a much more subtle way if clients misbehave by not consuming responses. For example, assume that a client *C* makes a connection to a server sends it a sequence of requests, but never reads the responses. The server keeps accepting requests, generating responses, and sending them back to the client.
- At the server, TCP protocol software transmits the first few bytes over the connection to the client. Eventually, TCP will fill the client's receive window and will stop transmitting data. If the server application program continues to generate responses, the local buffer TCP uses to store outgoing data for the connection will become full and the server process will block.
- Deadlock arises because processes block when the operating system cannot satisfy a system call. In particular, a call to *write* will block the calling process if TCP has no local buffer space for the data being sent; a call to *read* will block the calling process until TCP receives data.
- For concurrent servers, only the single slave process associated with a particular client blocks if the client fails to send requests or read responses. For a single-process implementation, however, the central server process will block. If the central server process blocks, it cannot handle other connections. The important point is that any server using only one process can be subject to deadlock.

Q.4 Write a program for procedure which forms the connection to the server.

```
 #define __USE_BSD   1

 #include <sys/types.h>
 #include <sys/socket.h>

 #include <netinet/in.h>
 #include <arpa/inet.h>

 #include <netdb.h>
 #include <string.h>
 #include <stdlib.h>

#ifndef INADDR_NONE
#define INADDR_NONE 0xffffffff
#endif  /* INADDR_NONE */

typedef unsigned short u_short;
extern int  errno;

int errexit(const char *format, ...);
```

```
int
connectsock(const char *host, const char *service, const char *transport )
/*
 * Arguments:
 *      host      - name of host to which connection is desired
 *      service   - service associated with the desired port
 *      transport - name of transport protocol to use ("tcp" or "udp")
 */
{
    struct hostent  *phe;   /* pointer to host information entry    */
    struct servent  *pse;   /* pointer to service information entry */
    struct protoent *ppe;   /* pointer to protocol information entry*/
    struct sockaddr_in sin; /* an Internet endpoint address      */
    int s, type;    /* socket descriptor and socket type    */


    memset(&sin, 0, sizeof(sin));
    sin.sin_family = AF_INET;

/* Map service name to port number */
if ( pse = getservbyname(service, transport) )
    sin.sin_port = pse->s_port;
else if ( (sin.sin_port = htons((u_short)atoi(service))) == 0 )
    errexit("can't get \"%s\" service entry\n", service);

/* Map host name to IP address, allowing for dotted decimal */
if ( phe = gethostbyname(host) )
    memcpy(&sin.sin_addr, phe->h_addr, phe->h_length);

 else if ( (sin.sin_addr.s_addr = inet_addr(host)) == INADDR_NONE )
     errexit("can't get \"%s\" host entry\n", host);

 /* Map transport protocol name to protocol number */
 if ( (ppe = getprotobyname(transport)) == 0)
     errexit("can't get \"%s\" protocol entry\n", transport);
    /* Use protocol to choose a socket type */
    if (strcmp(transport, "udp") == 0)
        type = SOCK_DGRAM;
    else
        type = SOCK_STREAM;

    /* Allocate a socket */
    s = socket(PF_INET, type, ppe->p_proto);
    if (s < 0)
        errexit("can't create socket: %s\n", strerror(errno));

    /* Connect the socket */
    if (connect(s, (struct sockaddr *)&sin, sizeof(sin)) < 0)
        errexit("can't connect to %s.%s: %s\n", host, service,
            strerror(errno));
    return s;
}
```

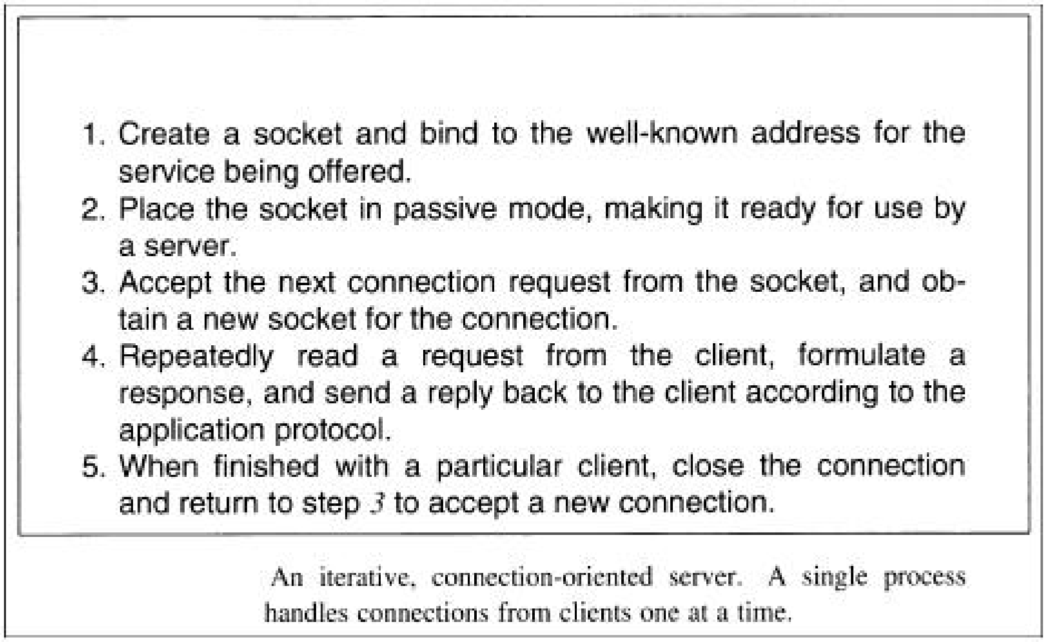
Q.5 Write a client side program which accesses the Daytime Service.



```
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

extern int  errno;

int TCPdaytime(const char *host, const char *service);
int errexit(const char *format, ...);
int connectTCP(const char *host, const char *service);

#define LINELEN     128

/*------------------------------------------------------------------------
 * main - TCP client for DAYTIME service
 *------------------------------------------------------------------------
 */
int
main(int argc, char *argv[])
{
    char    *host = "localhost";    /* host to use if none supplied */
    char    *service = "daytime";   /* default service port     */

    switch (argc) {
    case 1:
        host = "localhost";
        break;
    case 3:
        service = argv[2];
        /* FALL THROUGH */
    case 2:
        host = argv[1];
        break;
    default:
        fprintf(stderr, "usage: TCPdaytime [host [port]]\n");
        exit(1);
    }
    TCPdaytime(host, service);
    exit(0);
}

/*------------------------------------------------------------------------
 * TCPdaytime - invoke Daytime on specified host and print results
 *------------------------------------------------------------------------
 */
TCPdaytime(const char *host, const char *service)
{
    char    buf[LINELEN+1];     /* buffer for one line of text  */
    int s, n;           /* socket, read count       */

    s = connectTCP(host, service);

    while( (n = read(s, buf, LINELEN)) > 0) {
        buf[n] = '\0';      /* ensure null-terminated   */
        (void) fputs( buf, stdout );
    }
}
```

Q.6 Describe algorithm for iterative Connection oriented server algorithm and concurrent connectionless server algorithm.

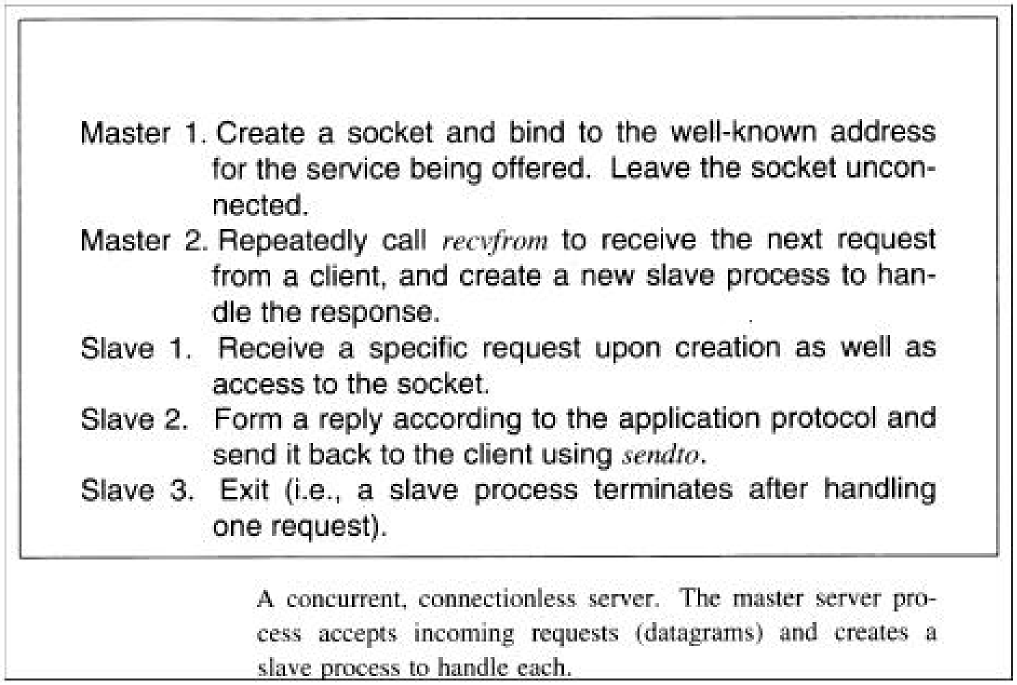
An Iterative, Connection-Oriented Server Algorithm



1. Create a socket and bind to the well-known address for the service being offered.
2. Place the socket in passive mode, making it ready for use by a server.
3. Accept the next connection request from the socket, and obtain a new socket for the connection.
4. Repeatedly read a request from the client, formulate a response, and send a reply back to the client according to the application protocol.
5. When finished with a particular client, close the connection and return to step *3* to accept a new connection.

An iterative, connection-oriented server. A single process handles connections from clients one at a time.

A Concurrent, Connectionless Server Algorithm



Master 1. Create a socket and bind to the well-known address for the service being offered. Leave the socket unconnected.

Master 2. Repeatedly call *recvfrom* to receive the next request from a client, and create a new slave process to handle the response.

Slave 1. Receive a specific request upon creation as well as access to the socket.

Slave 2. Form a reply according to the application protocol and send it back to the client using *sendto*.

Slave 3. Exit (i.e., a slave process terminates after handling one request).

A concurrent, connectionless server. The master server process accepts incoming requests (datagrams) and creates a slave process to handle each.

Q.7 How the performance of stateless server can be optimized? Explain briefly.

- Consider a connectionless server that allows clients to read information from files stored on the server's computer. To keep the protocol stateless, the designer requires each client request to specify a file name, a position in the file, and the number of bytes to read. The most straightforward server implementation handles each request independently: it opens the specified file, seeks to the specified position, reads the specified number of bytes, sends the information back to the client, and then closes the file.
- To optimize server performance, the programmer decides to maintain a small table of file information as Figure shows below
- The programmer uses the client's IP address and protocol port number as an index into the table, and arranges for each table entry to contain a pointer to a large buffer of data from the file being read. When a client issues its first request, the server searches the table and finds that it has no record of the client. It allocates a large buffer to hold data from the file, allocates a new table entry to point to the buffer, opens the specified file, and reads data into the buffer.
- It then copies information out of the buffer when forming a reply. The next time a request arrives from the same client, the server finds the matching entry in the table, follows the pointer to the buffer, and extracts data from it without opening the file.
- Once the client has read the entire file, the server deallocates the buffer and the table entry, making the resources available for use by another client.

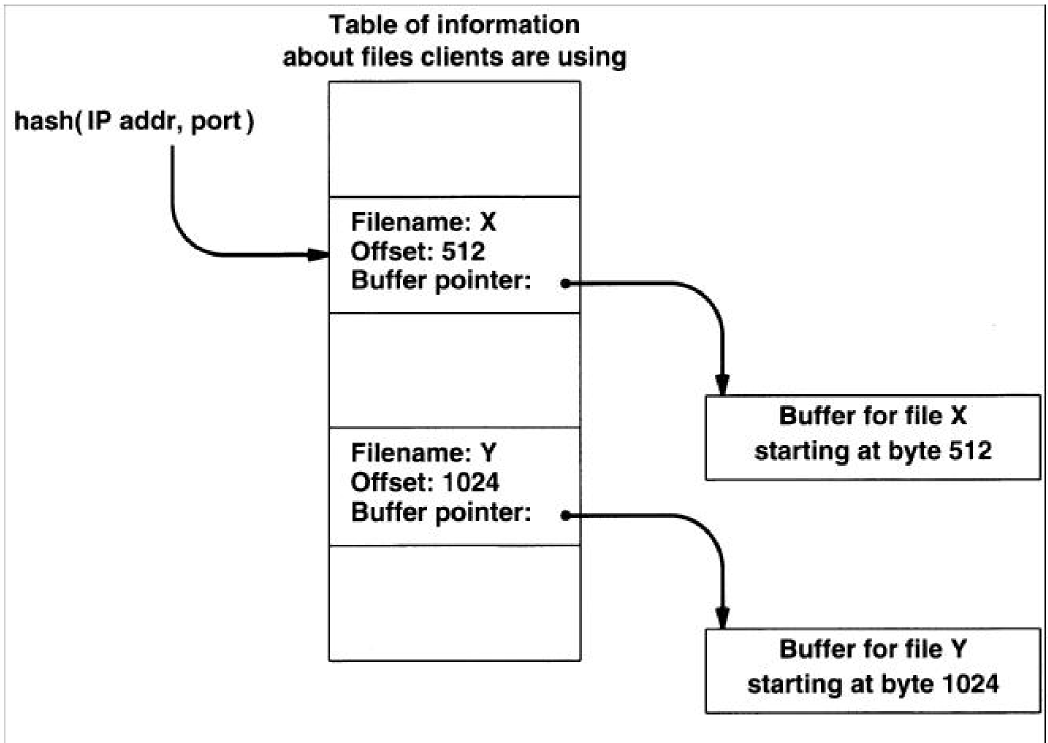


Table of information about files clients are using
hash(IP addr, port)
Filename: X
Offset: 512
Buffer pointer:
Filename: Y
Offset: 1024
Buffer pointer:
Buffer for file X starting at byte 512
Buffer for file Y starting at byte 1024

A table of information kept to improve server performance. The server uses the client's IP address and protocol port number to find an entry. This optimization introduces state information.

- The server also compares the file specified in a request with the file name in the table entry to verify that the client is still using the same file as the previous request.
- Adding the proposed table changes the server in a subtle way, however, because it introduces state information. Of course, state information chosen carelessly could introduce errors in the way the server responds. For example, if the server used the client's IP address and protocol port number to find the buffer without checking the file name or file offset in the request, duplicate or out-of-order requests could cause the server to return incorrect data.
- Unfortunately, even a small amount of state information can cause a server to perform badly when machines, client programs, or networks fail. To understand why, consider what happens if one of the client programs fails (i.e., crashes) and must be restarted.
- Chances are high that the client will ask for an arbitrary protocol port number and UDP will assign a new protocol port number different from the one assigned for earlier requests. When the server receives a request from the client, it cannot know that the client has crashed and restarted, so it allocates a new buffer for the file and a new slot in the table. Consequently, it cannot know that the old table entry the client was using should be removed. If the server does not remove old entries, it will eventually run out of table slots.