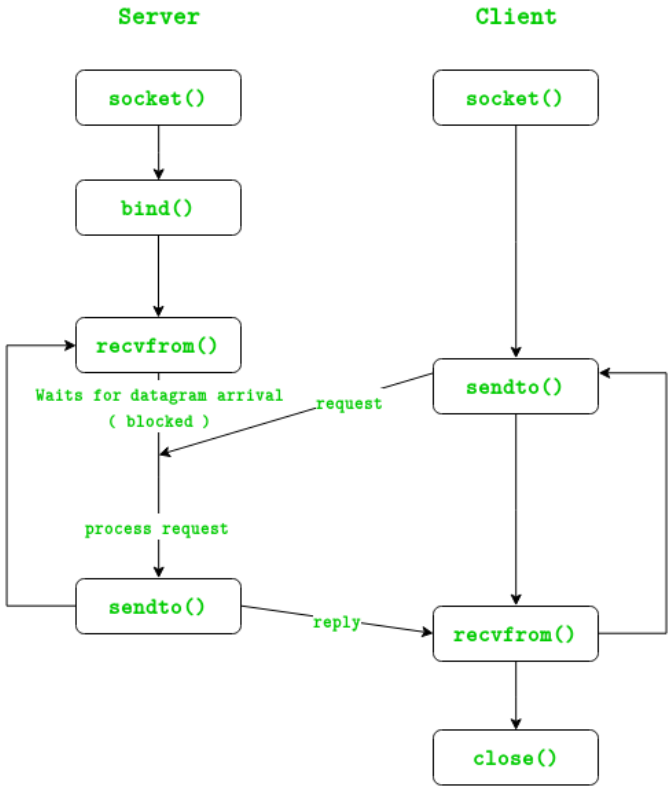
Program – 1

AIM: Implement concurrent day-time client-server application.

Introduction and Theory

There are two major transport layer protocols to communicate between hosts : TCP and UDP. In UDP, the client does not form a connection with the server like in TCP and instead just sends a datagram. Similarly, the server need not accept a connection and just waits for datagrams to arrive. Datagrams upon arrival contain the address of sender which the server uses to send data to the correct client.



Socket

A socket is a combination of IP address and port on one system. On each system a socket exists for a process interacting with the socket on other system over the network. A combination of local socket and the socket at the remote system is also known a 'Four tuple' or '4-tuple'. Each connection between two processes running at different systems can be uniquely identified through their 4-tuple.

Function Descriptions

`socket()`

Creates an UN-named socket inside the kernel and returns an integer known as socket descriptor. This function takes domain/family as its first argument. For Internet family of ipv4 addresses we use AF_INET. The second argument 'SOCK_STREAM' specifies that the transport layer protocol that we want should be reliable i.e. It should have acknowledgement techniques. The third

argument is generally left zero to let the kernel decide the default protocol to use for this connection. For connection oriented reliable connections, the default protocol used is TCP.

`bind()`

Assigns the details specified in the structure 'serv_addr' to the socket created in the step above. The details include, the family/domain, the interface to listen on(in case the system has multiple interfaces to network) and the port on which the server will wait for the client requests to come.

`listen()`

With second argument as '10' specifies maximum number of client connections that server will queue for this listening socket. After the call to listen(), this socket becomes a fully functional listening socket.

`accept()`

The server is put to sleep and when for an incoming client request, the three-way TCP handshake is complete, the function **`accept ()`** wakes up and returns the socket descriptor representing the client socket. **`Accept()`** is run in an infinite loop so that the server is always running and the delay or sleep of 1 sec ensures that this server does not eat up all your CPU processing. As soon as server gets a request from client, it prepares the date and time and writes on the client socket through the descriptor returned by **`accept().`**

Algorithm / Processes

```
server()
  create UDP socket
  Bind socket to address
  wait for datagram from client
  process and reply to client request
  repeat while server is active
```

```
client()
  create UDP socket
  send request to server
  wait for datagram from server
  process and reply from server
  close socket and exit
```

Program – 1

Code

Server

```c
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <time.h>
int main()
{
        struct sockaddr_in sa; // Socket address data structure
        int sockfd, coontfd; // Source and destination addresses
        char str[1025]; // Buffer to hold the out-going stream
        time_t tick; // System time data structure

        sockfd = socket(AF_INET, SOCK_STREAM, 0); // New socket
created

    // Checking for valid socket
        if (sockfd < 0)
        {
                printf("Error in creating socket\n");
                exit(0);
        }
        else
        {
                printf("Socket Created\n");
        }

    // Clearing and assigning type and address to the socket
        printf("Socket created\n");
        bzero(&sa, sizeof(sa));
    memset(str, '0', sizeof(str)); // clearing the buffer
        sa.sin_family = AF_INET;
        sa.sin_port = htons(5600);
        sa.sin_addr.s_addr = htonl(INADDR_ANY);

    // binding and verifying the socket to address
        if (bind(sockfd, (struct sockaddr*)&sa, sizeof(sa))<0)
        {
                printf("Bind Error\n");
        }
        else
                printf("Binded\n");

    // starts the server with a max client queue size set as 10
        listen(sockfd, 10);

    // server run
        while(1)
        {
```

Program – 1

```
                coontfd = accept(sockfd, (struct sockaddr*)NULL
,NULL); // Accept a request from client
                printf("Accepted\n");
        tick = time(NULL);
        snprintf(str, sizeof(str), "%.24s\r\n", ctime(&tick)); //
read sys time and write to buffer
        printf("sent\n");
        printf("%s\n", str);
                write(coontfd, str, strlen(str)); // send buffer to
client
        }
        close(sockfd); // close the socket
        return 0;
}
```

Client

```
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <arpa/inet.h>
int main()
{
        struct sockaddr_in sa;  // Socket address data structure

        int n, sockfd; // read and source
        char buff[1025]; // buffer to store the read stream
        sockfd = socket(PF_INET, SOCK_STREAM, 0); // New socket
created

    // Checking for valid socket
        if (sockfd < 0)
        {
                printf("Error in creation\n");
                exit(0);
        }
        else
                printf("Socket created\n");

    // Clearing and assigning type and address to the socket
        bzero(&sa, sizeof(sa));
        sa.sin_family = AF_INET;
        sa.sin_port = htons(5600);

    // establishing and verifying the connection
        if (connect(sockfd, (struct sockaddr_in*)&sa, sizeof(sa)) <
0)
        {
                printf("Connection failed\n");
                exit(0);
```

Program – 1

```
40          }
41          else
42                  printf("Connection made\n");
43
44      // Reading and priting data from the server after verification
45          if ( n = read(sockfd, buff, sizeof(buff)) < 0)
46          {
47                  printf("Read Error\n");
48                  exit(0);
49          }
50          else
51          {
52          printf("Read message: %s\n", buff);
53                  printf("%s\n", buff);
54          printf("Done with connection, exiting\n");
55          }
56          close(sockfd); // Closing the socket
57          return 0;
58  }
```

Results and Outputs:

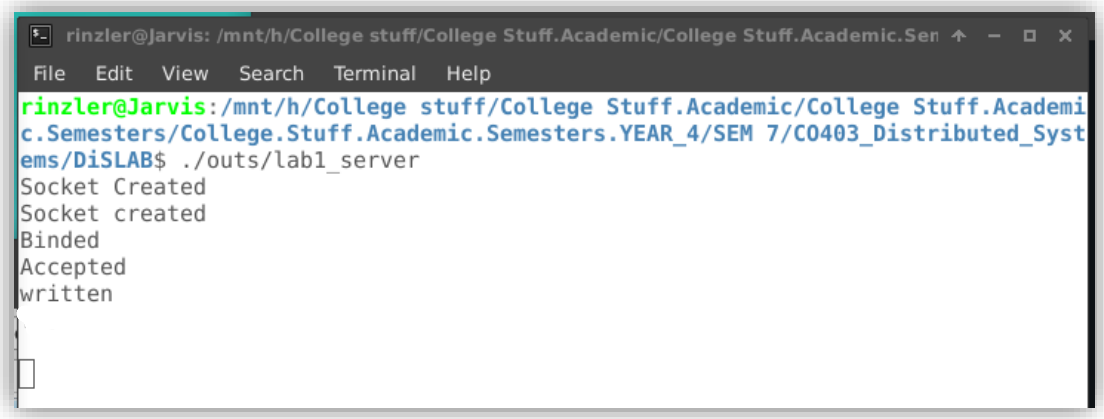


Figure 1 Server

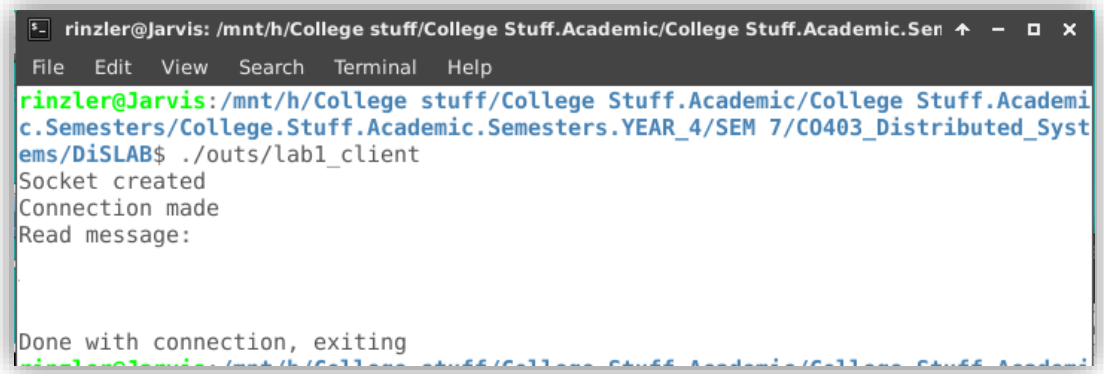


Figure 2 Client

Program – 1

Findings and Learnings:

1. We successfully implemented a date-time client-server.
2. UDP is a connectionless protocol where the server waits for a request from a client to become active. Each connection is treated as a new one\
3. On a local system i.e. within the same computer, the loop back address should be used as the argument to the client.
4. The connect procedure follows the Three way handshake process to establish the connection.