Daytime server code

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include "Practical.h"
#include <unistd.h>
#include    <time.h>

static const int MAXPENDING = 5; // Maximum outstanding connection requests

int main(int argc, char *argv[]) {
    time_t  ticks;
    char sendbuffer[BUFSIZE]; // Buffer for sending data to the client

    if (argc != 2) // Test for correct number of arguments
        DieWithUserMessage("Parameter(s)", "<Server Port>");

    in_port_t servPort = atoi(argv[1]); // First arg:  local port

        // Create socket for incoming connections
    int servSock; // Socket descriptor for server
    if ((servSock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0)
        DieWithSystemMessage("socket() failed");

    // Construct local address structure
    struct sockaddr_in servAddr;                  // Local address
    memset(&servAddr, 0, sizeof(servAddr));       // Zero out structure
    servAddr.sin_family = AF_INET;                // IPv4 address family
    servAddr.sin_addr.s_addr = htonl(INADDR_ANY); // Any incoming interface
    servAddr.sin_port = htons(servPort);          // Local port

    // Bind to the local address
    if (bind(servSock, (struct sockaddr*) &servAddr, sizeof(servAddr)) < 0)
        DieWithSystemMessage("bind() failed");

    // Mark the socket so it will listen for incoming connections
    if (listen(servSock, MAXPENDING) < 0)
        DieWithSystemMessage("listen() failed");

    for (;;) { // Run forever

    // Wait for a client to connect
    int clntSock = accept(servSock, (struct sockaddr *) NULL, NULL);
    if (clntSock < 0)
      DieWithSystemMessage("accept() failed");

    // clntSock is connected to a client!
    snprintf(sendbuffer, sizeof(sendbuffer), "%.24s\r\n", ctime(&ticks)); //Create data and time string in outgoing buffer
    ssize_t numBytesSent = send(clntSock, sendbuffer, strlen(sendbuffer), 0); //Send date and time string to the client
    if (numBytesSent < 0)
      DieWithSystemMessage("send() failed");

    close(clntSock); // Close client socket

  }
  // NOT REACHED
}
```

Overview of the ***daytime*** Server

- Refer to the Daytime server code presented in class.
- The operation of the Daytime application is as follows:
  - Client calls CONNECT to connect to the server,
  - Server calls ACCEPT to accept the connection request,
  - Server returns a formatted string using the SEND primitive
  - Server application calls CLOSE to close the connection,
  - Client calls RECV to retrieve the data from the connection,
  - Client closes the connection using the CLOSE primitive,
  - The Client application terminates using *exit(0);*

Overview of the ***daytime*** Server

- Key points to note in the Daytime server code:
  – Server address structure,
  – Calls to bind(), listen() and accept().

The *echo* client-server

- Having explored the **Daytime** application, the next application to examine is the **Echo** *Client* and *Server*.
- The essence of this application is as follows:
  – The Client application sends (**send()**) a string (from the command-line) to the server across an open connection,
  – The Server application reads (**recv()**) the string and returns it to the Client application (**send()**)exactly as it came in,
  – Both applications call for the connection to be closed (**close()**).

The *recv()* primitive

- To complete this task it is necessary to understand the operation of the ***recv()*** primitive.

```
while ((numBytes = recv(sock, recvbuffer, BUFSIZE - 1, 0)) >0)
        {
        recvbuffer[numBytes] = '\0';
        fputs(recvbuffer, stdout);
        }
```

- The following slide outlines some key points about this primitive.

The *recv()* primitive

- The while loop is necessary as the data may not arrive across the connection in a single transfer:
  - Recall that data arriving from the remote socket is stored in the RECV-Q buffer within the local TCP entity,
  - Repeated calls to **recv()** are needed to transfer this data from TCP (the Transport layer) into the application (the Application layer).

- **numBytes** is the return value from **recv():**
  - It represents the number of bytes read from the socket,
  - It returns one of three values:

    <1 represents an error condition,

    0 represents a closed connection i.e. remote application has called **close()**,

    >1 represents an open connection with potentially more data to be received.

Breaking from ***recv()***

- If the recv()n primitive is used as above in the **Echo** *Client* and *Server* applications it will cause problems:
  – Either or both applications will remain inside the loop,
  – Refer to the solution discussed in class.

Addressing

- A key aspect of *Networked Applications,* such as *Daytime and Echo,* is **Addressing**.
- In order for the Client and Server applications to communicate with each other, some form of explicit addressing <u>is</u> required.
- The *Destination* Server application requires an <u>**unambiguous**</u> (<u>**unique**</u>) address in order for the *Source* Client application to initiate a connection request:
  - Uniqueness can only be achieved using <u>both</u> IP <u>and</u> TCP addressing.

Addressing

- Recall that the IP layer is responsible for delivering datagrams/packets to remote hosts across an internetwork:
  - It uses IP addressing to achieve this *host-to-host* delivery.
- However, the data encapsulated inside these datagrams/packets is typically destined for an **application** in the Application Layer.
- With potentially many applications residing in the Application layer, how does the data get to the correct application?
  - ***Transport Layer*** addressing plays a vital role in this task.

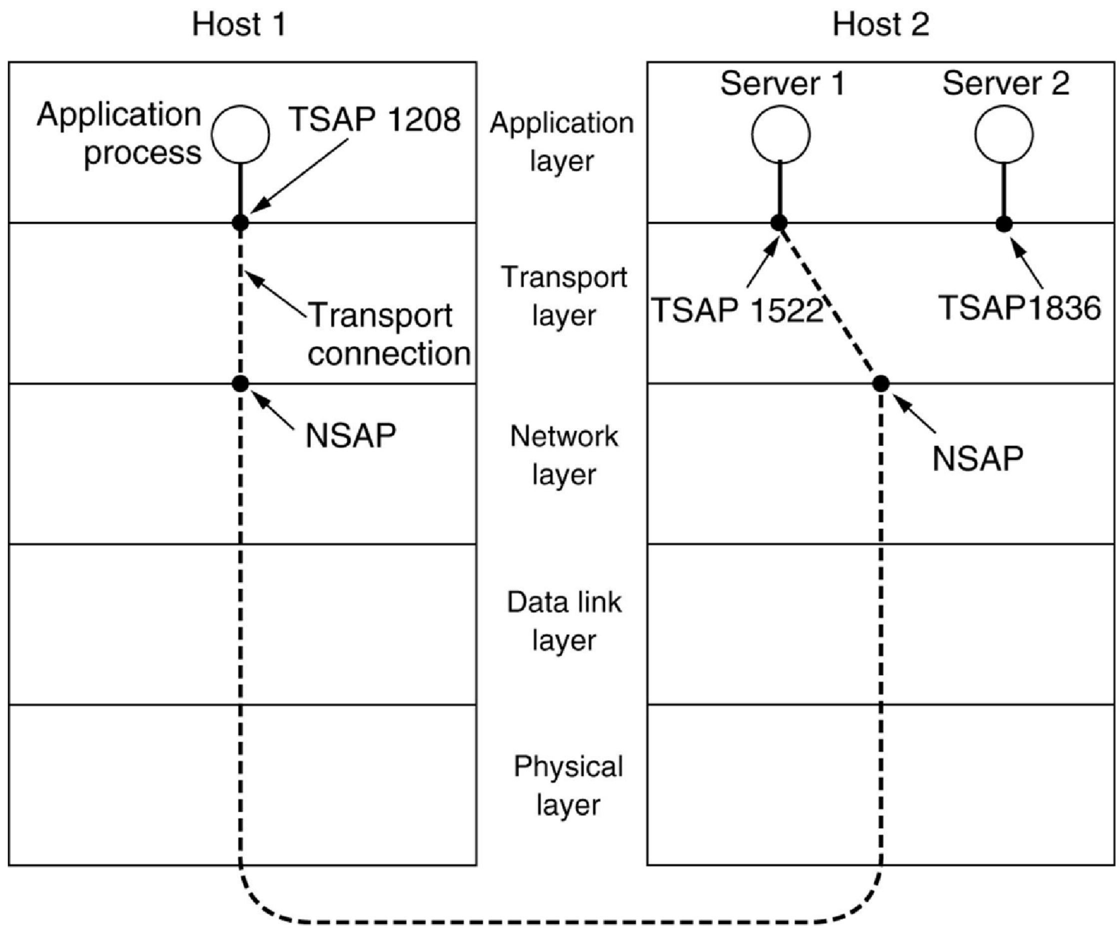
Transport Addressing

- Recall that transport protocols such as TCP provide services to the application layer.
- To ensure unambiguous (unique) addressing of individual applications, the Transport layer provides its own addressing schema separate to the IP layer:
  - These are generally known as Transport Service Access Points (TSAPs),
  - These TSAPs uniquely identify entities in the Transport layer known as *end points,*
  - In TCP parlance these end points are known as ***port numbers*** or more simply ***ports****.*

Transport Addressing

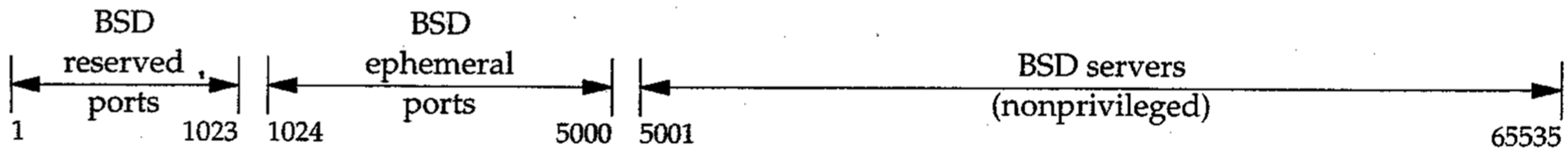
- ***Port*** numbers are used by:
  - Server applications to advertise their services and, to listen for Connection Requests,
  - Client applications to uniquely identify a Server application when making a Connection Request.
- The *network layer* also defines *end points.* These are known as Network Service Access Points (NSAPs):
  - IP addresses are examples of NSAPs.
- The following slide illustrates the relationship between the NSAPs, TSAPs and transport connections.

TSAPs, NSAPs and transport connections



The TCP *Port Number* Range

- TCP Port numbers are sixteen bits long.
- This creates a port number address space comprising approx. 65K addresses (16 bits implies 2^16 addresses) as follows:



The *Port Number* Range

- *Reserved* addresses are for well known applications such as HTTP (port 80), FTP (ports 20 and 21), Telnet (port 23) etc.
  – These can only be allocated by users with SU privileges.
- *Ephemeral* addresses are allocated by TCP to **Client** applications:
  – It is not immediately obvious that Client applications require a port number,
  – However, there needs to be a return address for data from the Server application.

The *Port Number* Range

- *Non-privileged* addresses are for any other applications:
  - This is the range that will be used in the lab exercises.
- It is important to note that the Ephemeral and Non-privileged ranges differ on different OS's:
  - Your home host may use different ranges depending on the OS used.
  - On the Virtual Machines used in the labs, us the following command to view the range:

    sudo sysctl net.ipv4.ip_local_port_range

Examining Addressing Details

- Active connections can be viewed using the *netstat* utility:
  - From the command-line prompt type: ***netstat –ntap***
- This command reveals details of connections that exist within the host OS.
- An explanation of the flags:
  - ‘n’ reveals IP addresses in dotted-decimal notation,
  - ‘t’ filters on TCP addresses only,
  - ‘a’ shows all connections,
  - ‘p’ reveals the process id details for each connection.
- The following slide shows a sample output when the **netstat** command is used.

Examining Addressing Details

From the server end:

Proto	Recv-Q	Send-Q	Local Address	Foreign Address	State	PID/Program name
TCP	0	0	0.0.0.0:1022	0.0.0.0:*	LISTEN	12937/webserver
TCP	0	0	147.252.30.9:1022	147.252.234.34:4136	ESTABLISHED	13268/webserver

From the client end:

Proto	Recv-Q	Send-Q	Local Address	Foreign Address	State	PID/Program name
TCP	0	0	147.252.234.34:4136	147.252.30.9:1022	ESTABLISHED	13267/httpclient

- This shows:
  - A server with *listening* and *connected* sockets on port 1022,
  - A client on an *ephemeral* port 4136.

Socket Identifiers

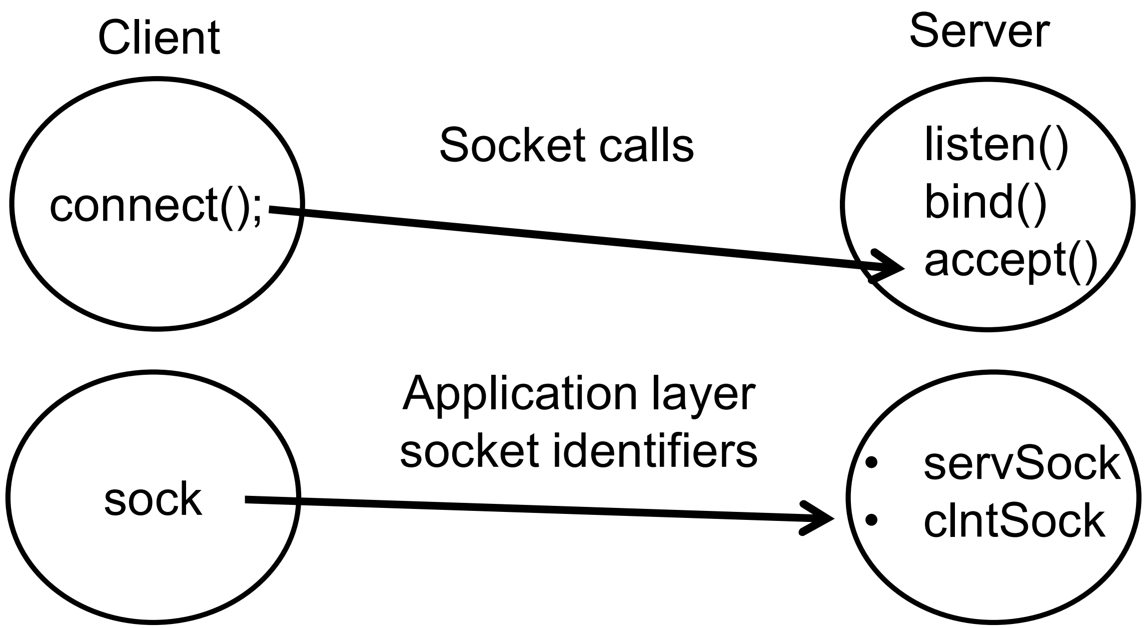
- Note the columns *Local Address* and *Remote Address*:
  - These contain details of the ***socket*** at <u>each</u> end of a connection.
- From a TCP perspective, a ***socket*** is identified by a combination of an NSAP and TSAP separated by a colon (**:**):
  - This is different to how an Application perceives a socket.
  - Applications use an ***integer*** descriptor to identify a socket. Recall the variable names used in the applications developed in lab: sock, servSock, clntSock etc.

Socket Pairs

- Each row in the output from ***netstat*** relates to a connection.
- A TCP connection can be considered as a connection between two socket identifiers; a **local** socket and a **remote** socket.
  - The combination of *Local Address* and *Remote Address* is the ***identifier*** for a connection. It is known as a ***Socket Pair***.
  - An example socket pair: {147.252.30.9:1022, 147.252.234.34:4136}
- ***Socket Pairs*** are how TCP views connections:
  - For each connection, the detail contained in each row is reversed depending on which end of the connection the command is run.
  - Notice how connection identifiers/socket pairs are guaranteed to be unique.

Socket Pairs

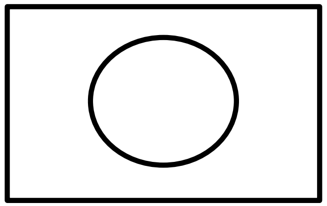
- The following slide reveals how connections are established from a Socket Pair perspective:
  - It assumes the following primitive calls have been made within each of the client an server applications:



Socket pairs before Connection Establishment

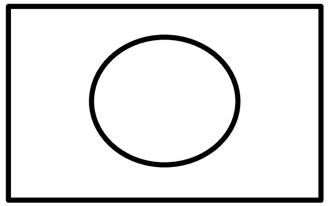
<u>Before</u> Connection Establishment

198.69.10.2



Client-side Socket: {198.69.10.2 : 1500}

206.62.226.35



Server-side Listening Socket:
{0.0.0.0:21, 0.0.0.0:*}
or sometimes written as, {*:21, *:*}

- A Client Connection Request is now sent to : 206.62.226.35, Port 21
- The following slide shows the socket pairs relating to the connection **<u>after</u>** Connection Establishment.

Sockets pairs after Connection Establishment

<u>After</u> Connection Establishment

Host: 198.69.10.2

Host: 206.62.226.35

Server-side **Listening** Socket Pair:
{0.0.0.0:21, 0.0.0.0:*} or, {*:21, *:*}

Client-side **Connected** Socket Pair:

{198.69.10.2:1500, 206.62.226.35:21}

Server-side **Connected** Socket Pair:

{206.62.226.35:21, 198.69.10.2:1500}

Note the reversed sockets