456: Network-Centric Programming

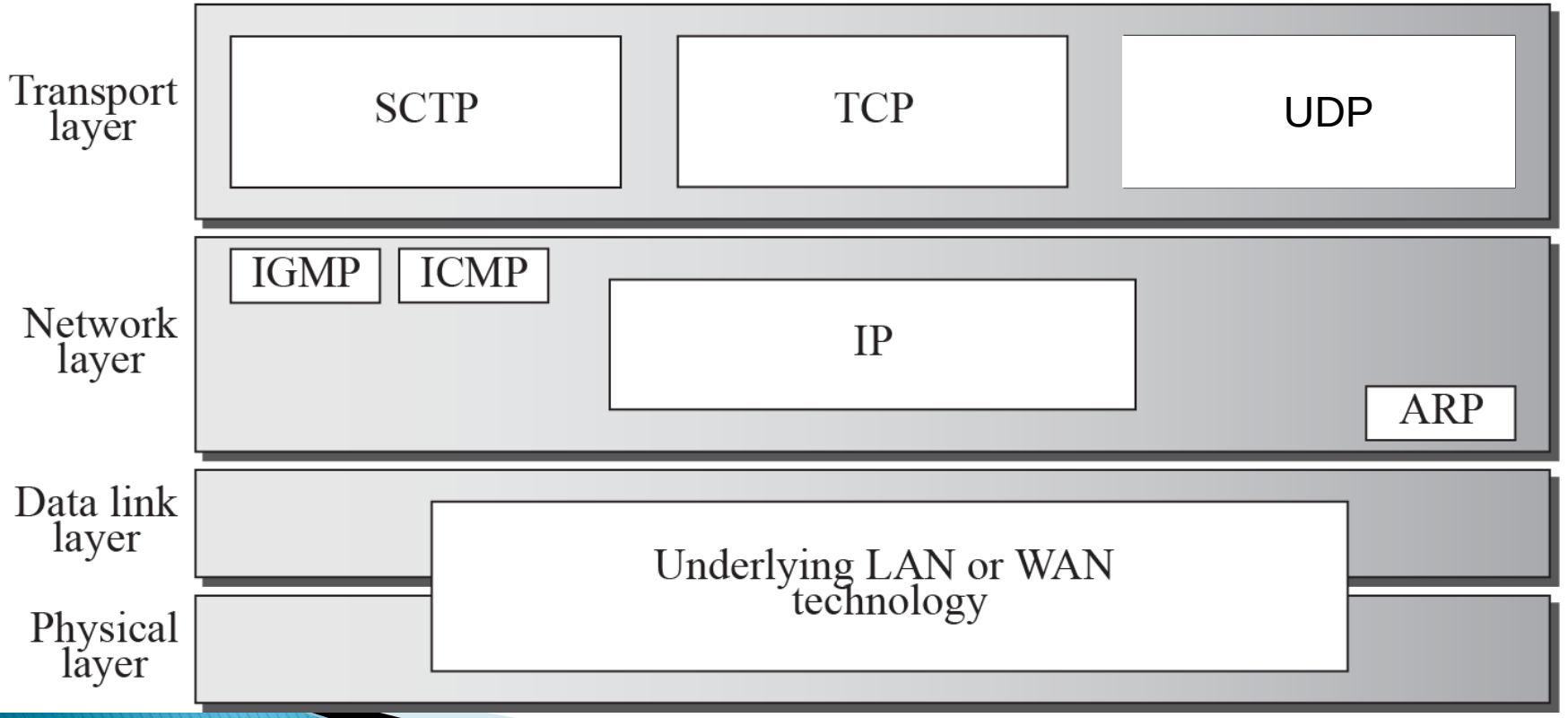
Yingying (Jennifer) Chen

Web: https://www.winlab.rutgers.edu/~yychen/
Email: yingche@scarletmail.rutgers.edu

Department of Electrical and Computer Engineering
Rutgers University

TCP UDP SCTP

TCP: Transmission Control Protocol
UDP: User Datagram Protocol
SCTP: Stream Control Transmission Protocol



TCP (Transmission Control Protocol)

- TCP is a reliable, ordered, and error-checked protocol.
- TCP provides a stream of bytes between applications running on hosts communicating via an Internet Protocol (IP) network.
- TCP is a connection-oriented protocol, which means a connection is established and maintained until the application programs at each end have finished exchanging messages.

UDP (User Datagram Protocol)

- UDP is used primarily for establishing low-latency and loss-tolerating connections between applications on the internet.
- Send short messages called datagrams
- It is an unreliable, connectionless protocol.

SCTP (Stream Control Transmission Protocol)

- SCTP is a protocol for transmitting multiple streams of data at the same time between two end points that have established a connection in a network. Sometimes referred to as "next generation TCP".
- SCTP's multi-streaming allows data to be delivered in multiple, independent streams, so that if there is data loss in one stream, delivery will not be affected for the other streams.

Sockets API

- General interface for network programming and inter-process communication
- A socket is a communication endpoint, a tap into the network
- Network protocol independent but usually used with Internet protocols
  - Allows variety of addressing formats
- Datagram
  - Unordered message-oriented communication
  - Application multiplexing
  - Usually mapped to UDP
- Stream
  - Application multiplexing
  - Reliable, flow controlled data stream
  - Usually mapped to TCP

Creating a socket

- int socket(int domain, int type, int protocol)
  - Returns socket file descriptor

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

int socket(int domain, int type, int protocol);
```

Returns: nonnegative descriptor if OK, −1 on error

- domain: specifies the protocol family
- type: socket type
- protocol: set to a specific protocol type or 0 (select default AF_UNIX)

Creating a socket

- int socket(int domain, int type, int protocol)
- domain: selects the protocol family which will be used for communication.

Domain	Description
AF_INET	IPv4 protocols
AF_INET6	IPv6 protocols
AF_LOCAL	Unix domain protocols
AF_ROUTE	Routing sockets
AF_KEY	Key socket

Creating a socket

- int socket(int domain, int type, int protocol)
- type: socket type, it specifies the communication semantics.

type	Description
SOCK_STREAM	stream socket
SOCK_DGRAM	datagram socket
SOCK_SEQPACKET	sequenced packet socket
SOCK_RAW	raw socket

SOCK_STREAM Provides sequenced, reliable, two-way, connection-based byte streams.
SOCK_DGRAM Supports datagrams (connectionless, unreliable messages of a fixed maximum length).
SOCK_RAW Provides raw network protocol access (you can use your own structures for UDP and TCP packet headers)

SOCK_SEQPACKET gives you the guarantees of SOCK_STREAM (i.e., preservation of ordering, guaranteed delivery, no duplication), but with delineated packet boundaries just like SOCK_DGRAM. So, basically it's a mix of the two protocol types.

In the TCP/IP-family, SCTP implements both SOCK_STREAM (TCP-like) and SOCK_SEQPACKET.

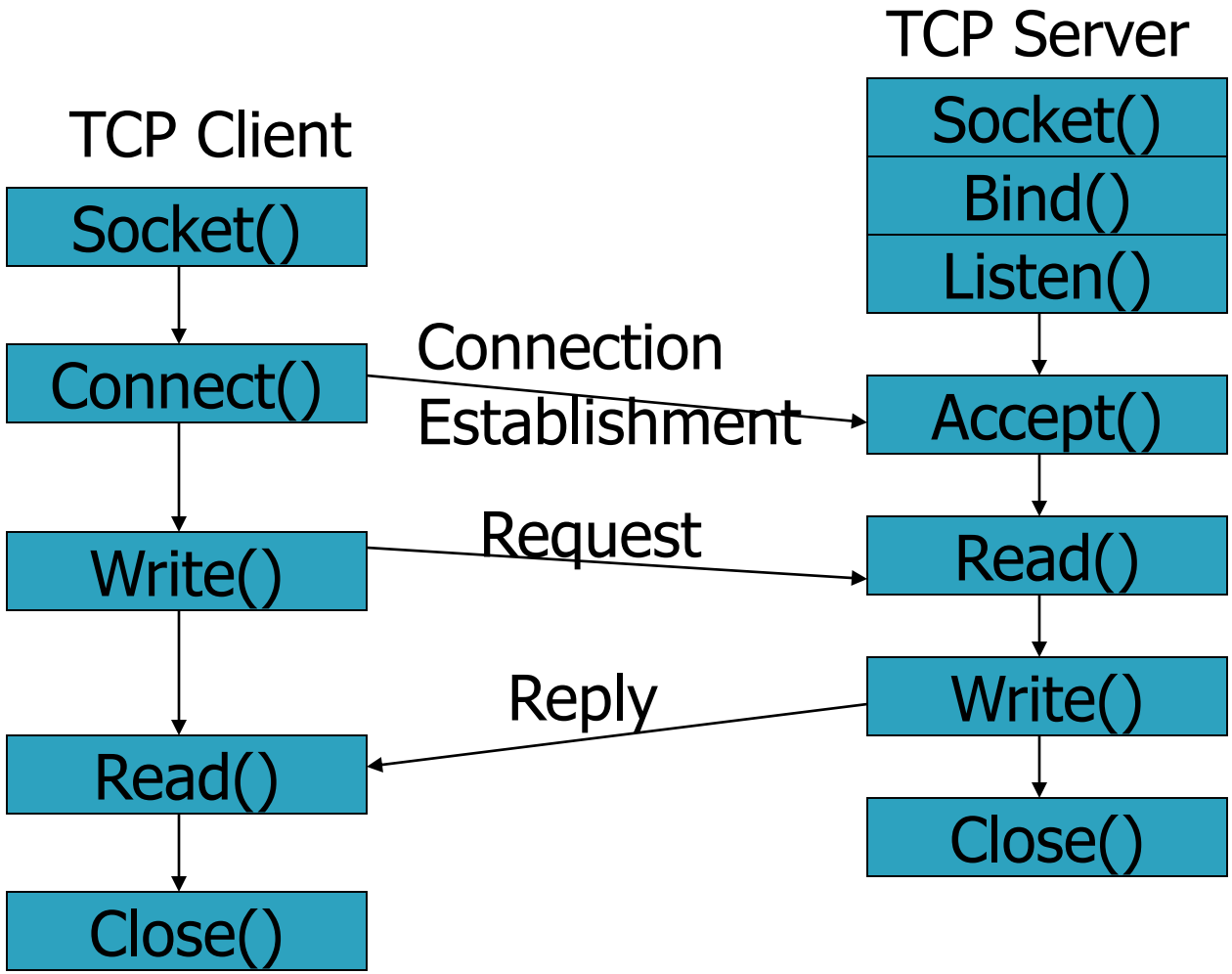
Creating a socket

- int socket(int domain, int type, int protocol)
- protocol：Setting protocol to 0 to select system's default based on the given domain and type.

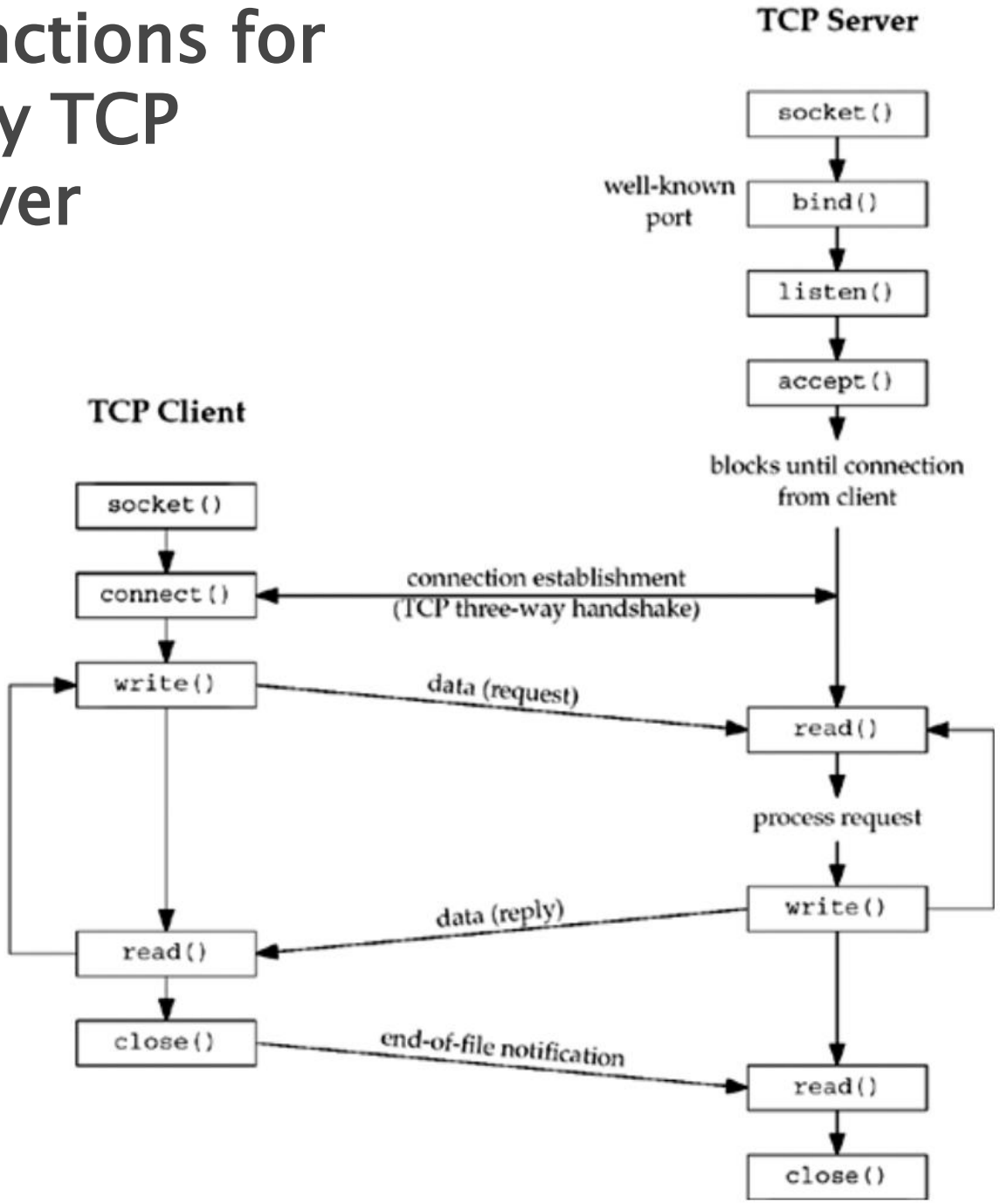
Protocol	Description
IPPROTO_TCP	TCP transport protocol
IPPROTO_UDP	UDP transport protocol
IPPROTO_SCTP	SCTP transport protocol

TCP: Transmission Control Protocol
UDP: User Datagram Protocol
SCTP: Stream Control Transmission Protocol

Typical Socket Implementation of Client-Server Scenario



Socket functions for elementary TCP client/server



Network prefix and Host number

- IP address (v4) is 4 bytes (32bits) long.
- The network prefix identifies a network and the host number identifies a specific host (actually, interface on the network).

network prefix	host number

- How do we know how long the network prefix is?
  - The network prefix used to be implicitly defined (**class-based addressing, A,B,C,D…**)

Class	Left-most Bit	Starting IP Address	Last IP Address
A	0xxx	0.0.0.0	127.255.255.255
B	10xx	128.0.0.0	191.255.255.255
C	110x	192.0.0.0	223.255.255.255
D	1110	224.0.0.0	239.255.255.255
E	1111	240.0.0.0	255.255.255.255

  - The network prefix now is flexible and is indicated by a **prefix**/**netmask (classless).**

Example

Example: www.example.com

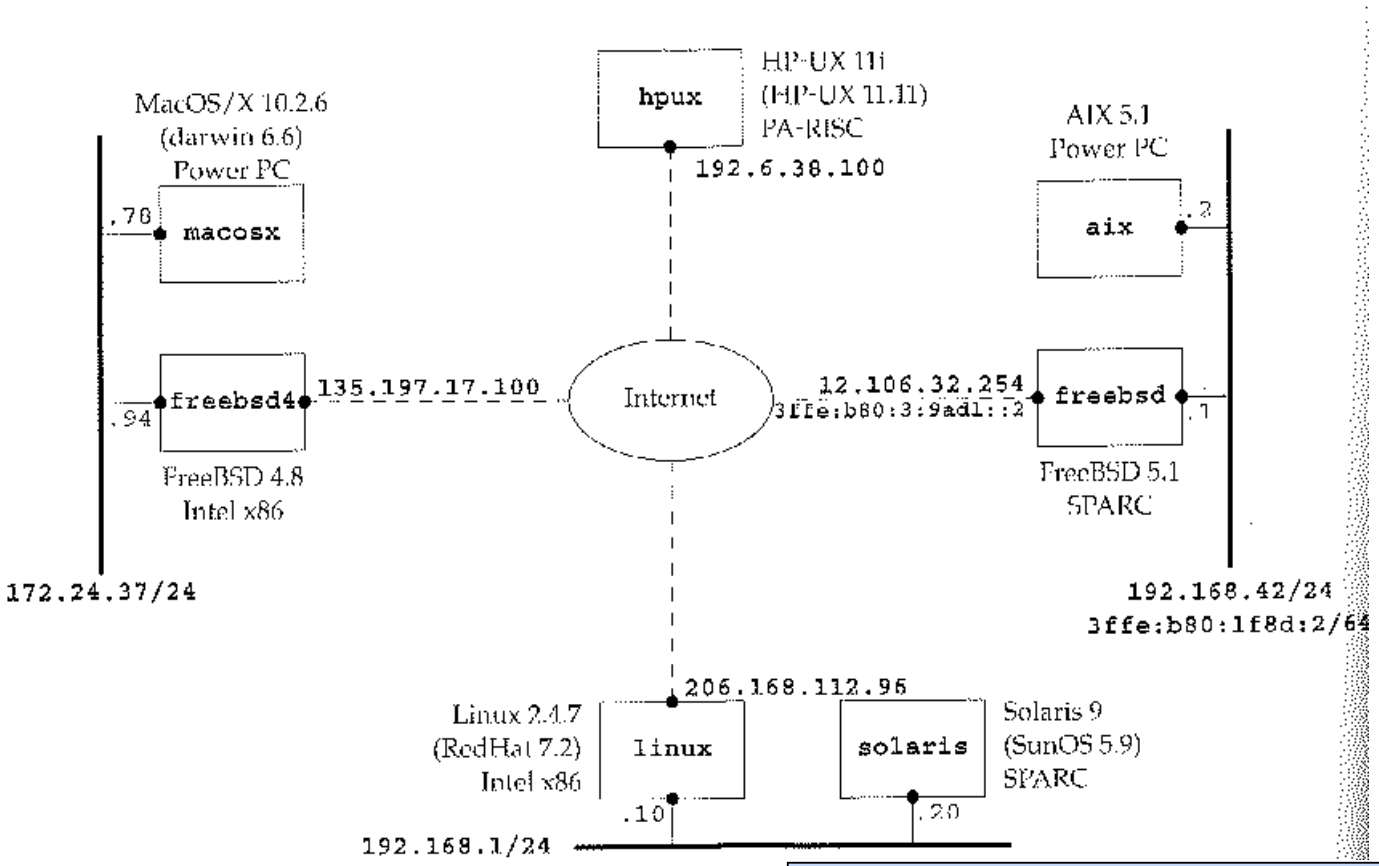
- IP address is 128.143.137.144

- Using Prefix notation IP address is: **128.143.137.144/16**
  - Network prefix is 16 bits long

- Network mask is: **255.255.0.0**
  -----> Network id (IP address **AND** Netmask) is: **128.143.0.0**
  -----> Host number (IP address **AND** inverse of Netmask **0.0.255.255**) is: **137.144**

128.143	137.144

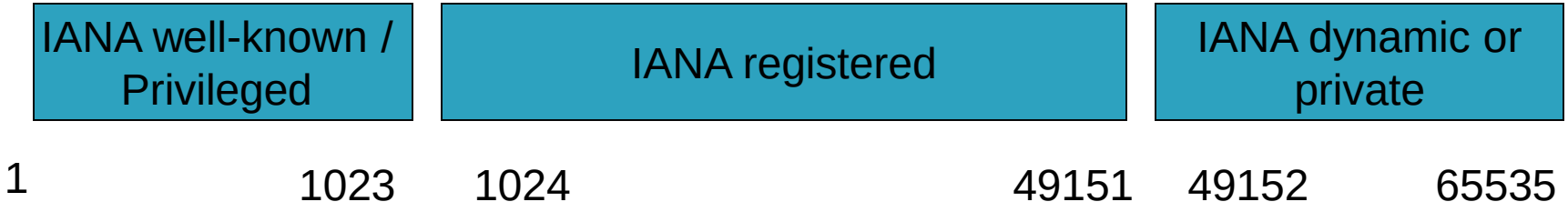
Which IP address?



Internet Assigned Numbers Authority (IANA) reserves these for private IP addresses

Private IP address space	
From	**To**
10.0.0.0	10.255.255.255
172.16.0.0	172.31.255.255
192.168.0.0	192.168.255.255

Internet Assigned Numbers Authority (IANA) Maintains Port Numbers (16bits)



- Common applications
- E.g.
  - Daytime 13
  - http 80
- Root access only

- Registered, less common applications
- E.g.
  - X Window server

- Used for private applications
- Used as ephemeral ports

TCP, UDP, … have separate sets of port numbers

Useful commands

- Network statistics
  - netstat –i

Displays network interfaces and their statistics

```
chen@chen-VirtualBox:~$ netstat -i
Kernel Interface table
Iface   MTU Met   RX-OK RX-ERR RX-DRP RX-OVR    TX-OK TX-ERR TX-DRP TX-OVR Flg
enp0s3     1500 0      44849      0      0 0         32044      0      0      0 B
MRU
lo        65536 0      10873      0      0 0         10873      0      0      0 L
RU
```

- Iface (Name of the interface)
  - Primary network card “eth0”
  - The loopback interface “lo” (a virtual network card that allows the computer to make networking connections to itself)
- MTU (Maximum Transmission Unit this interface can send at a time (in byte)
- Flg
  - B: broadcast capability; M: Multicast capability; L loopback interface

enp0s3 "**ethernet network peripheral # *serial* #”
Ubuntu virtual machine

Useful commands

```
chen@chen-VirtualBox:~$ netstat -i
Kernel Interface table
Iface   MTU Met   RX-OK RX-ERR RX-DRP RX-OVR    TX-OK TX-ERR TX-DRP TX-OVR Flg
enp0s3     1500 0     44849      0      0 0        32044      0      0      0 B
MRU
lo        65536 0     10873      0      0 0        10873      0      0      0 L
RU
```

- RX-OK/ERR/DRP/OVR
  - Statistics about the packets that have been received by the interface
  - OK: correctly received
  - ERR: received but with incorrect checksum
  - DRP: dropped because the receiver buffer is full
  - OVR: dropped because the kernel couldn't get to it in time

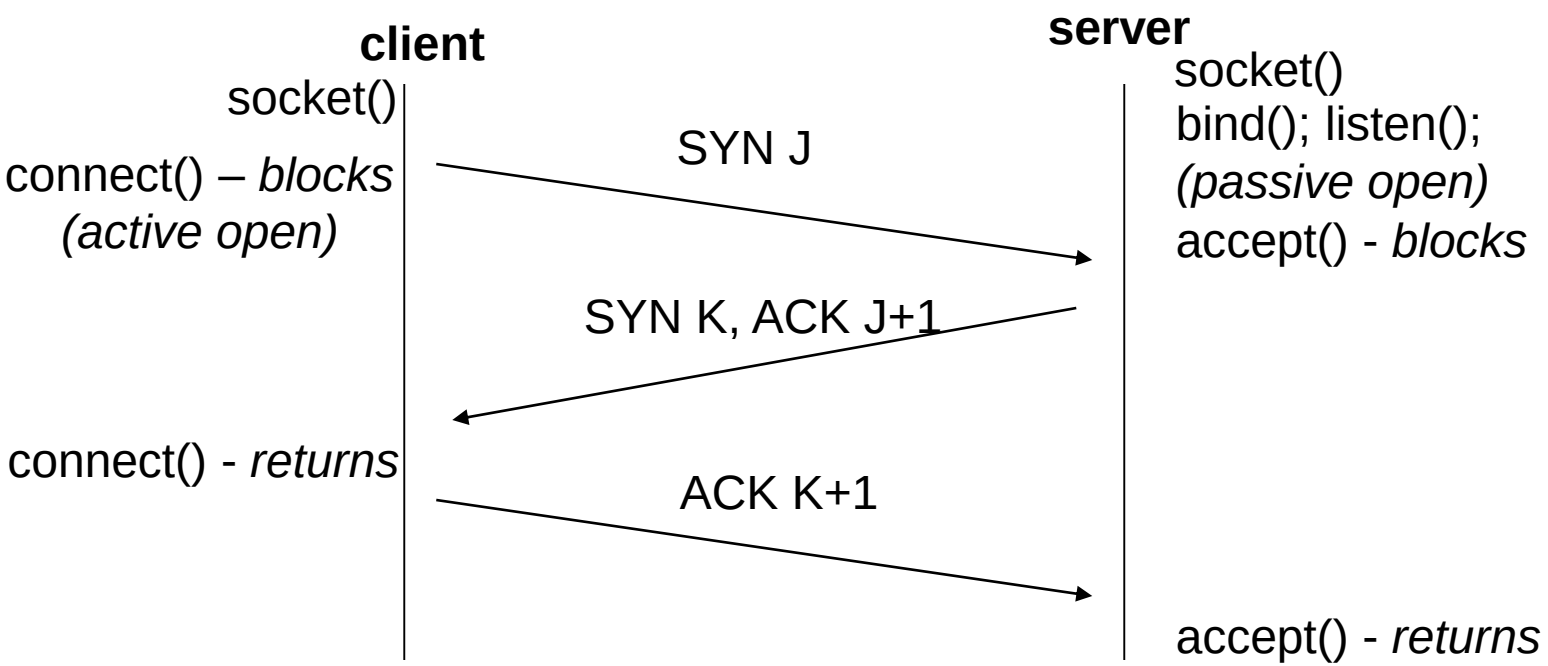
TX-OK/ERR/DRP/OVR show similar statistics at the transmitter

Useful commands

- ifconfig <interface>
  - Show IP address and configuration of interface
- Note the *lo* loopback interface -> good choice for your assignments
  - The loopback device is a special, virtual network interface that your computer uses to communicate with itself.
  - 127.0.0.1; localhost

```
[redacted]-VirtualBox:~$ ifconfig lo
lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:70 errors:0 dropped:0 overruns:0 frame:0
          TX packets:70 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:6402 (6.4 KB)  TX bytes:6402 (6.4 KB)
```

TCP Three-Way Handshake (Connection Establishment)



TCP Three-Way Handshake (Connection Establishment)

- The client issues an active open by calling *connect()*. This causes the client TCP to send a **"synchronize" (SYN)** segment, which tells the server the client's initial sequence number for the data that the client will send on the connection.
  - Normally, there is no with the SYN; it just contains an IP header, a TCP header, and possible TCP options.

- The server must **acknowledge (ACK)** the client's SYN and the server must also send its own SYN containing the initial sequence number for the data that the server will send on the connection.
  - The server sends its SYN and the ACK of the client's SYN in a single segment.

How do client and server know which port to choose?

bind()

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

int bind(int sockfd, struct sockaddr *my_addr, int addrlen);
```

Returns: 0 if OK, −1 on error

- Called by a TCP server
- The bind() function tells the kernel to associate the server's socket address in *my_addr* with the socket descriptor *sockfd*. The *addrlen* argument is sizeof(sockaddr_in)
- Assigns a local address to a socket (e.g. IP address and/or port number)

listen()

- Called by a TCP server
- After socket() and bind() but must be before calling accept()

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

int listen(int sockfd, int backlog);
```

Returns: 0 if OK, −1 on error

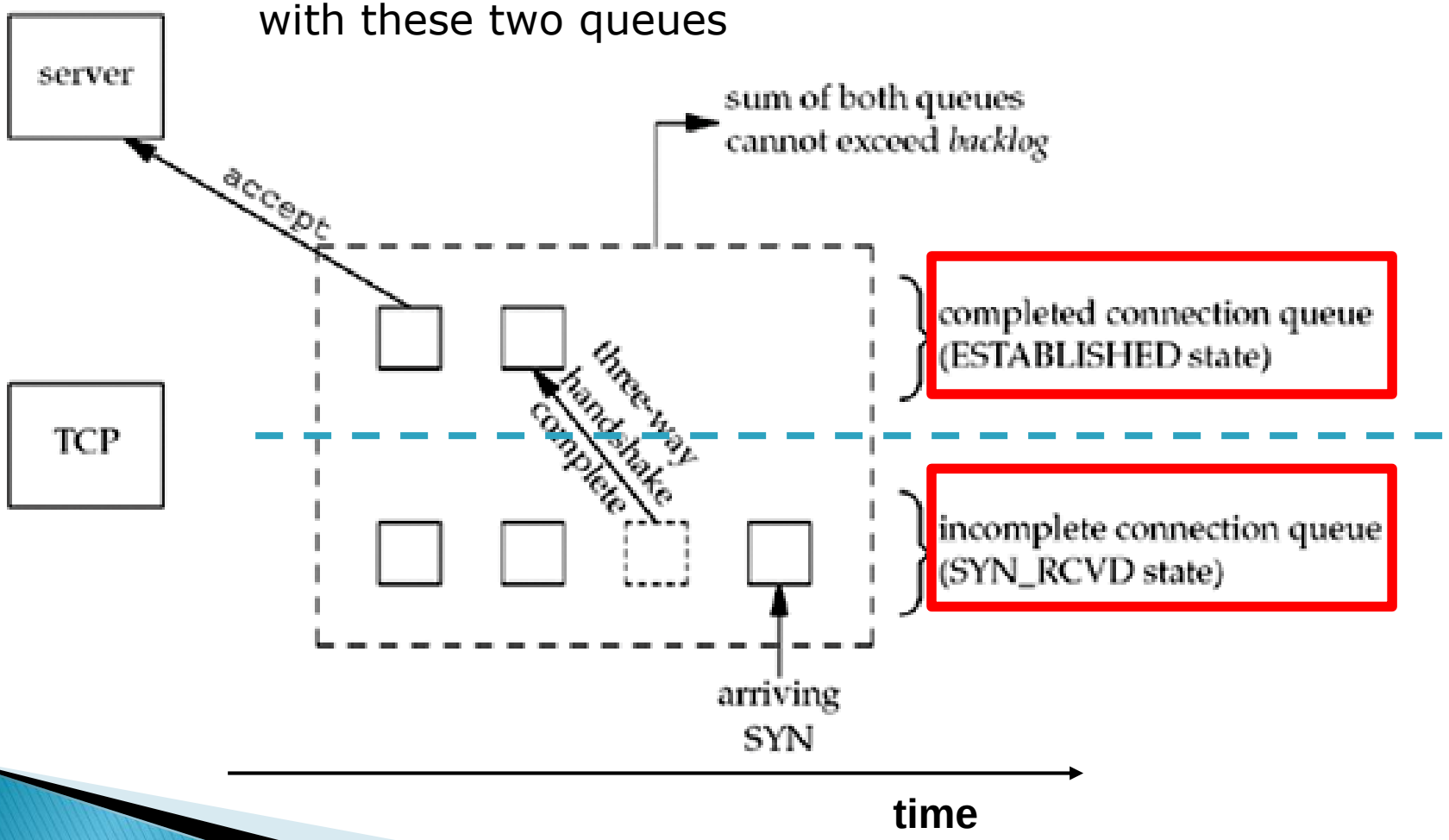
- Backlog specifies the buffer size (queue length) for incoming connections
  - We will typically set it to a large value, such as 1024.
- Configures a socket to start accepting incoming connections
- The kernel maintains two queues:
  - Incomplete connection queue
  - completed connection queue

The two queues maintained by TCP for a listening socket

- An incomplete connection queue
  - Contains an entry for each SYN packet that has arrived from a client for which the server is awaiting completion of the TCP three-way handshake.
  - The socket is in the SYN_RCVD state
- A completed connection queue
  - Contains an entry for each client with whom the TPC three-way handshake has completed
  - The socket is in the ESTABLISHED state

The two queues maintained by TCP for a listening socket

Packets exchange during the connection establishment with these two queues



Connecting to a server

- connect(): a TPC client to establish a connection with a TCP server

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

int connect(int sockfd, struct sockaddr *serv_addr, int addrlen);
```

Returns: 0 if OK, −1 on error

- Requires open socket and destination address
- Several possible errors:
  - ETIMEDOUT – no response received after connection attempt
  - ECONNREFUSED – the server has refused the connection attempt (often wrong IP or port number)
  - EHOSTUNREACH – a router has notified the client that the destination could not be found

accept()

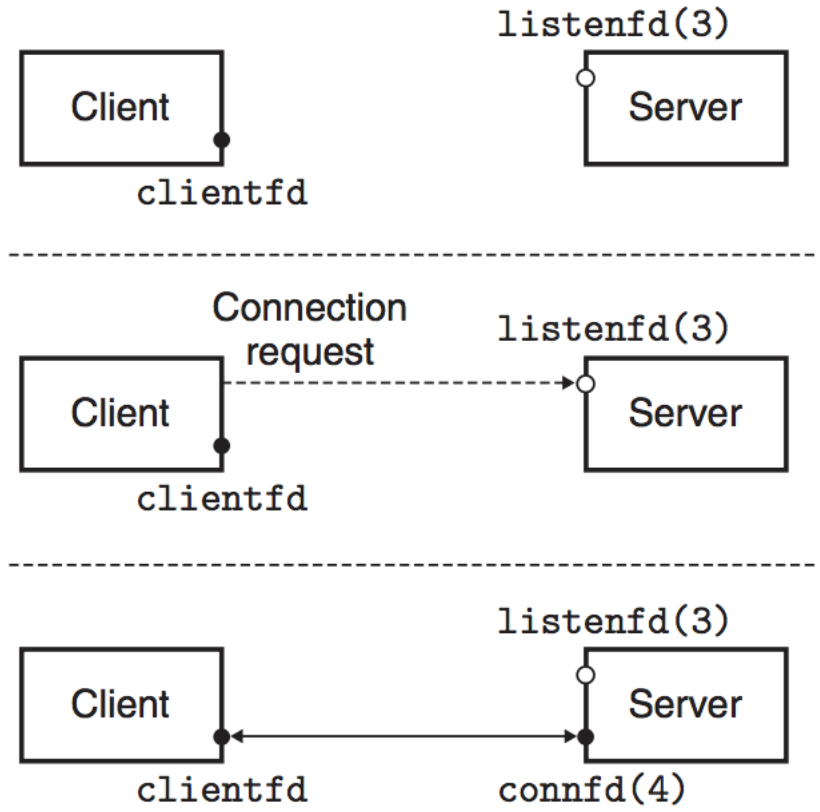
- Called by a TCP server to wait for the request from a client

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

int accept(int listenfd, struct sockaddr *addr, int *addrlen);
                                    Returns: nonnegative connected descriptor if OK, −1 on error
```

- The *listenfd* is the descriptor obtained from *socket()* Returns the next completed connection from the established queue
  - Return value is a different value for the connected socket: *connfd*
- Blocking
- *addr* returns the address of the connected client
- *addrlen* is a value-result argument (takes length of input structure, returns number of bytes stored in this structure)

The roles of the listening and connected descriptors

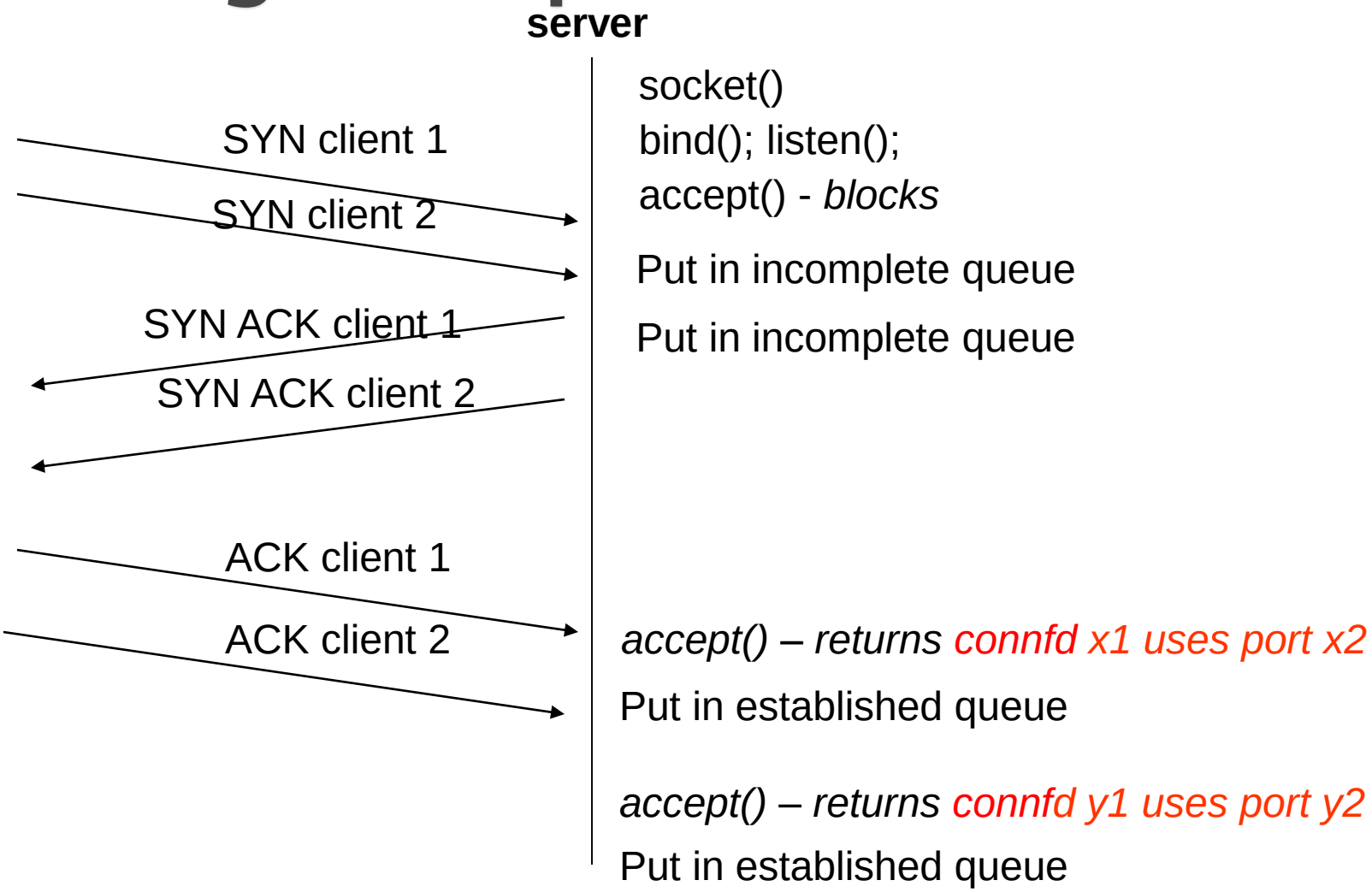


1. Server blocks in `accept`*, waiting for connection request on listening descriptor* `listenfd`*.*

2. Client makes connection request by calling and blocking in `connect`*.*

3. Server returns `connfd` *from* `accept`*. Client returns from* `connect`*. Connection is now established between* `clientfd` *and* `connfd`*.*

Handling multiple clients



Socket Address Structures

sockaddr: socketbits.h (included by socket.h), sockaddr_in: netinet/in.h

```
/* Generic socket address structure (for connect, bind, and accept) */
struct sockaddr {
    unsigned short  sa_family;   /* Protocol family */
    char            sa_data[14]; /* Address data.  */
};

/* Internet-style socket address structure */
struct sockaddr_in  {
    unsigned short  sin_family;  /* Address family (always AF_INET) */
    unsigned short  sin_port;    /* Port number in network byte order */
    struct in_addr  sin_addr;    /* IP address in network byte order */
    unsigned char   sin_zero[8]; /* Pad to sizeof(struct sockaddr) */
};
```

sockaddr: socketbits.h (included by socket.h), sockaddr_in: netinet/in.h

Socket Address Structures

- The *sockaddr* is a ***generic*** socket address.
  - To accept any kind of socket address structure, we define sockets functions to expect a pointer to a generic sockaddr structure, and then require applications to cast pointers to protocol-specific structures to this generic structure.

```
struct  sockaddr_in  servaddr;
....
bind(listenfd, (sockaddr *) &servaddr, sizeof(servaddr));
```

- Internet socket addresses are stored in **16-byte** structures of the type *sockaddr_in*. For Internet applications, the *sin_family* member is *AF_INET*, the *sin_port* member is a 16-bit port number, and the *sin_addr* member is a 32-bit IP address.

Iterative Server (daytime server): lecture_5_code\daytime\daytimetcpsrv.c

```
#include <sys/socket.h>
#include <sys/types.h>
#include <time.h>
#include <strings.h>
#include <netinet/in.h>
#include <stdio.h>
#include <unistd.h>

#define   MAXLINE              4096       /* max text line length */#define
LISTENQ              1024       /* 2nd argument to listen() */

typedef struct sockaddr SA;

int main(int argc, char **argv){

    int                  listenfd, connfd;
    struct     sockaddr_in  servaddr;
    char                 buff[MAXLINE];
    time_t               ticks;
```

Iterative Server (daytime server): lecture_5_code\daytime\daytimetcpsrv.c

```
listenfd = socket(AF_INET, SOCK_STREAM, 0);

bzero(&servaddr, sizeof(servaddr));                    //zero a byte string
servaddr.sin_family      = AF_INET;
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);

/* allows the server to accept a client connection on any interface */

servaddr.sin_port        = htons(13);    /* daytime server, can only be run as root */

bind(listenfd, (SA *) &servaddr, sizeof(servaddr));

listen(listenfd, LISTENQ);

for ( ; ; ) {
        len = sizeof(cliaddr);
        connfd = accept(listenfd, (SA *) &cliaddr, &len);

        ticks = time(NULL);
        snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));    //Write formatted output to sized buffer
        write(connfd, buff, strlen(buff));

        close(connfd);
    }
}
```

Terminal 1

$ sudo ./server

Terminal 2

$ ./client 127.0.0.1

The htonl function converts a 32-bit integer from host byte order to network byte order.
The htons function performs corresponding conversions for 16-bit integers.

Address Conversion

- Convert string representation (“192.168.0.2”) into binary representation
- “Presentation” to/from “Numeric”
  - int inet_pton(family, strptr, addrptr)
  - char* inet_ntop(family, addrptr, strptr, len)

```
#include <arpa/inet.h>
int inet_pton(int family, const char *strptr, void *addrptr);
                                  Returns: 1 if OK, 0 if input not a valid presentation format, -1 on error
const char *inet_ntop(int family, const void *addrptr, char *strptr, size_t len);
                                  Returns: pointer to result if OK, NULL on error
```

The family could `AF_INET`.

Iterative Server (daytime client): lecture_5_code\daytime\daytimetcpcli.c

```
#include <sys/socket.h>
#include <sys/types.h>
#include <time.h>
#include <strings.h>
#include <netinet/in.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <dirent.h>
#include <stdlib.h>
#include <stdarg.h>
#include <arpa/inet.h>

#define   MAXLINE              4096       /* max text line length */#define

typedef struct sockaddr SA;
```

Iterative Server (daytime client): lecture_5_code\daytime\daytimetcpcli.c

```
int main(int argc, char **argv) {
    int    sockfd, n;
    char recvline[MAXLINE + 1];
    struct sockaddr_in     servaddr;

    if (argc != 2)
        err_quit("usage: a.out <IPaddress>");

    if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
        err_sys("socket error");

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_port   = htons(13);        /* daytime server */
    if (inet_pton(AF_INET, argv[1], &servaddr.sin_addr) <= 0)
        err_quit("inet_pton error for %s", argv[1]);
    if (connect(sockfd, (SA *) &servaddr, sizeof(servaddr)) < 0)
        err_sys("connect error")
```

Iterative Server (daytime client): lecture_5_code\daytime\daytimetcpcli.c

```
    while ( (n = read(sockfd, recvline, MAXLINE)) > 0) {
            recvline[n] = 0;  /* null terminate */
            if (fputs(recvline, stdout) == EOF)
                    err_sys("fputs error");
    }
    if (n < 0)
            err_sys("read error");
}
```

Server (hellomsg server): lecture_5_code\hellomsg\server.c

```
#include <stdio.h>
#include <sys/socket.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <string.h>
#include <unistd.h>
#define PORT 8080
{
    int server_fd, new_socket, valread;
    struct sockaddr_in address;
    int opt = 1;
    int addrlen = sizeof(address);
    char buffer[1024] = {0};
    char *hello = "Hello from server";
 //Creating socket file descriptor
      if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0)
      {
          perror("socket failed");
          exit(EXIT_FAILURE);
      }
```

In this code, the client sends a hello message to the server and the server returns a hello message.

Terminal 1

$ ./server

Terminal 2

$ ./client

Server (hellomsg server): lecture_5_code\hellomsg\server.c

```
 address.sin_family = AF_INET;
 address.sin_addr.s_addr = INADDR_ANY;
 address.sin_port = htons( PORT );

            // bind socket to the port 8080
if (bind(server_fd, (struct sockaddr *)&address, sizeof(address))<0)
 {
    perror("bind failed");
    exit(EXIT_FAILURE);
 }
 if (listen(server_fd, 1024) < 0)
 {
    perror("listen");
    exit(EXIT_FAILURE);
 }
 if ((new_socket = accept(server_fd, (struct sockaddr *) &address, (socklen_t*)&addrlen))<0)
 {
    perror("accept");
    exit(EXIT_FAILURE);
 }
```

Server (hellomsg server): lecture_5_code\hellomsg\server.c

```
valread = read( new_socket , buffer, 1024);
    printf("%s\n",buffer );
    write(new_socket , hello , strlen(hello));
    printf("Hello message sent\n");
    return 0;
}
```

Client (hellomsg client): lecture_5_code\hellomsg\client.c

```
// Client side
#include <stdio.h>
#include <sys/socket.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <string.h>
#include <unistd.h>
 #include <arpa/inet.h>
#define PORT 8080

int main(int argc, char const *argv[])
{
    struct sockaddr_in address;
    int sock = 0, valread;
    struct sockaddr_in serv_addr;
    char *hello = "Hello from client";
    char buffer[1024] = {0};
```

Client (hellomsg client): lecture_5_code\hellomsg\client.c

```
if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0)
{
    printf("\n Socket creation error \n");
    return -1;}

 memset(&serv_addr, '0', sizeof(serv_addr));
 serv_addr.sin_family = AF_INET;
 serv_addr.sin_port = htons(PORT);

 // Convert IP addresses from text to binary form
 if(inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr)<=0)
 {
     printf("\nInvalid address/ Address not supported \n");
     return -1;
 }
```

Client (hellomsg client): lecture_5_code\hellomsg\client.c

```
if (connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0)
{
    printf("\nConnection Failed \n");
    return -1;
}
write(sock , hello , strlen(hello));
printf("Hello message sent\n");
valread = read( sock , buffer, 1024);
printf("%s\n",buffer );
return 0;
}
```

Homework Readings

- Readings:
  - Unix Network Programming: Section 3,4
  - Computer Systems – A Programmer‘s Perspective: Section 10.4 + 11.4