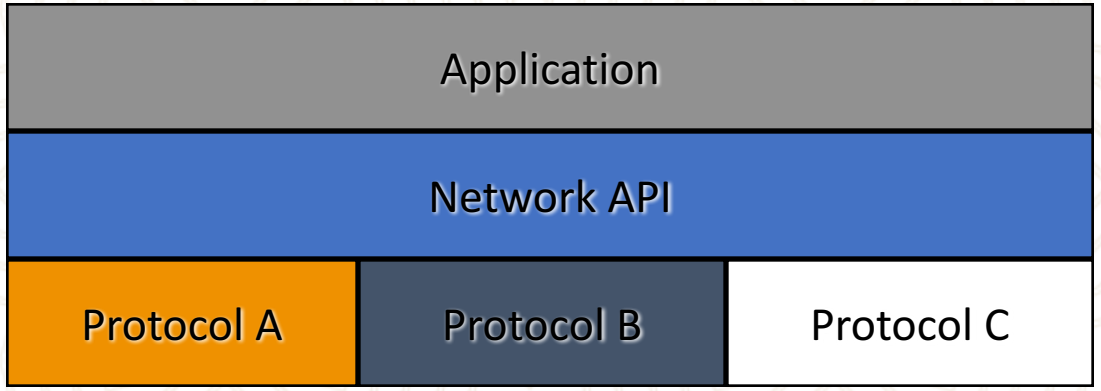
Socket Programming

Z. Cihan TAYŞİ

Network Application Programming Interface (API)

- The services provided (often by the operating system) that provide the interface between application and protocol software.



Network API

- Operating system provides Application Programming Interface (API) for network application
- API is defined by a set of function types, data structures, and constants
- Desirable characteristics of the network interface
  - Simple to use
  - Flexible
    - independent from any application
    - allows program to use all functionality of the network
  - Standardized
    - allows programmer to learn once, write anywhere
- Application Programming Interface for networks is called socket

Sockets

- Sockets provide mechanisms to communicate between computers across a network
  - A socket is an abstract representation of a communication endpoint.

- There are different kind of sockets
  - DARPA Internet addresses (Internet Sockets)
  - Unix interprocess communication (Unix Sockets)
  - CCITT X.25 addresses
  - and many others

- Berkeley sockets is the most popular Internet Socket
  - runs on Linux, FreeBSD, OS X, Windows
  - fed by the popularity of TCP/IP

Types of Internet Sockets

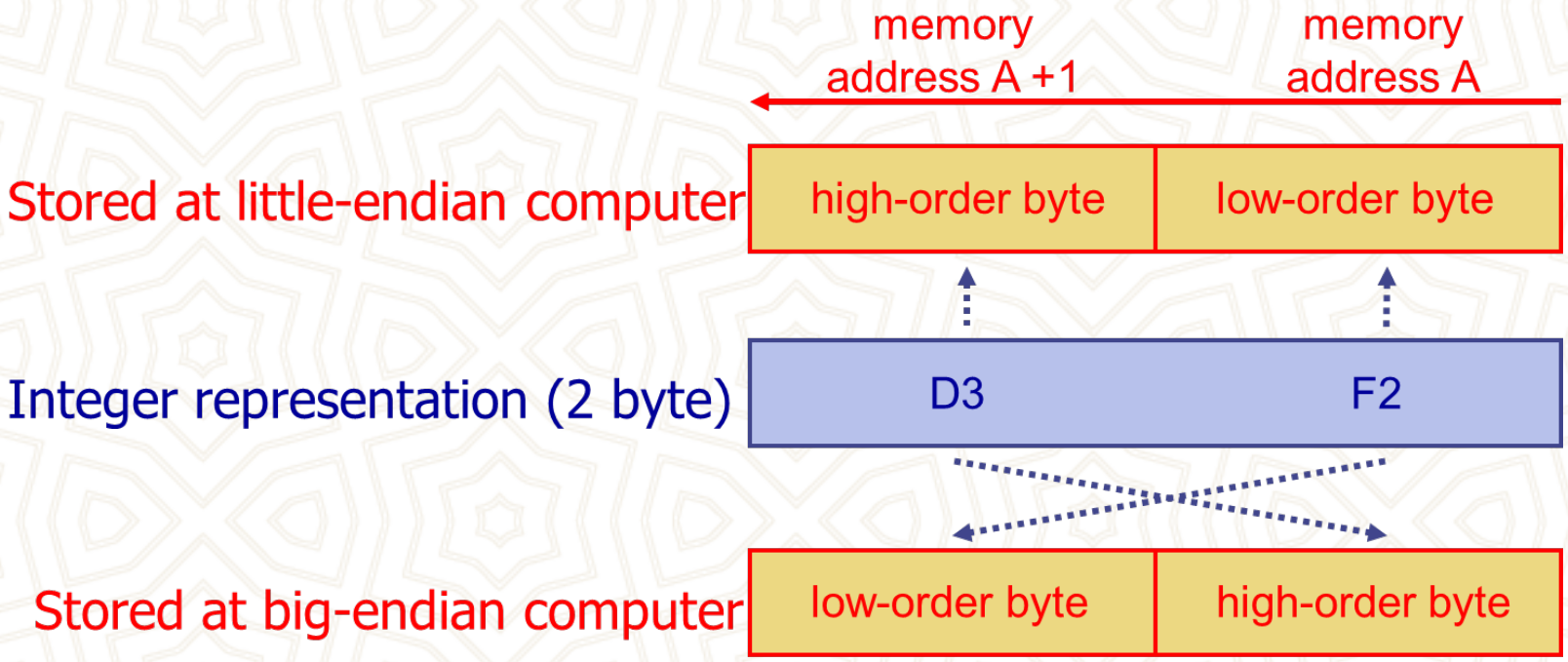
- Different types of sockets implement different communication types (stream vs. datagram)
- Type of socket: stream socket
  - connection-oriented
  - two way communication
  - reliable (error free), in order delivery
  - can use the Transmission Control Protocol (TCP)
  - e.g. telnet, ssh, http
- Type of socket: datagram socket
  - connectionless, does not maintain an open connection, each packet is independent
  - can use the User Datagram Protocol (UDP)
  - e.g. IP telephony
- Other types exist: similar to the one above

Network Programming Tips

- Byte Ordering
- Naming
- Addressing

Byte Ordering of Integers

- Different CPU architectures have different byte ordering



Byte Ordering Problem

- Question: What would happen if two computers with different integer byte ordering communicate?
- Answer:
  - Nothing if they do not exchange integers!
  - But: If they exchange integers, they would get the wrong order of bytes, therefore, the wrong value!

Byte Ordering Solution

- There are two solutions if computers with different byte ordering system want to communicate
  - They must know the kind of architecture of the sending computer (bad solution, it has not been implemented)
  - Introduction of a network byte order. The functions are:

```
uint16_t htons(uint16_t host16bitvalue)
uint32_t htonl(uint32_t host32bitvalue)
uint16_t ntohs(uint16_t net16bitvalue)
uint32_t ntohl(uint32_t net32bitvalue)
```

- Note: use for all integers (short and long), which are sent across the network
  - Including port numbers and IP addresses

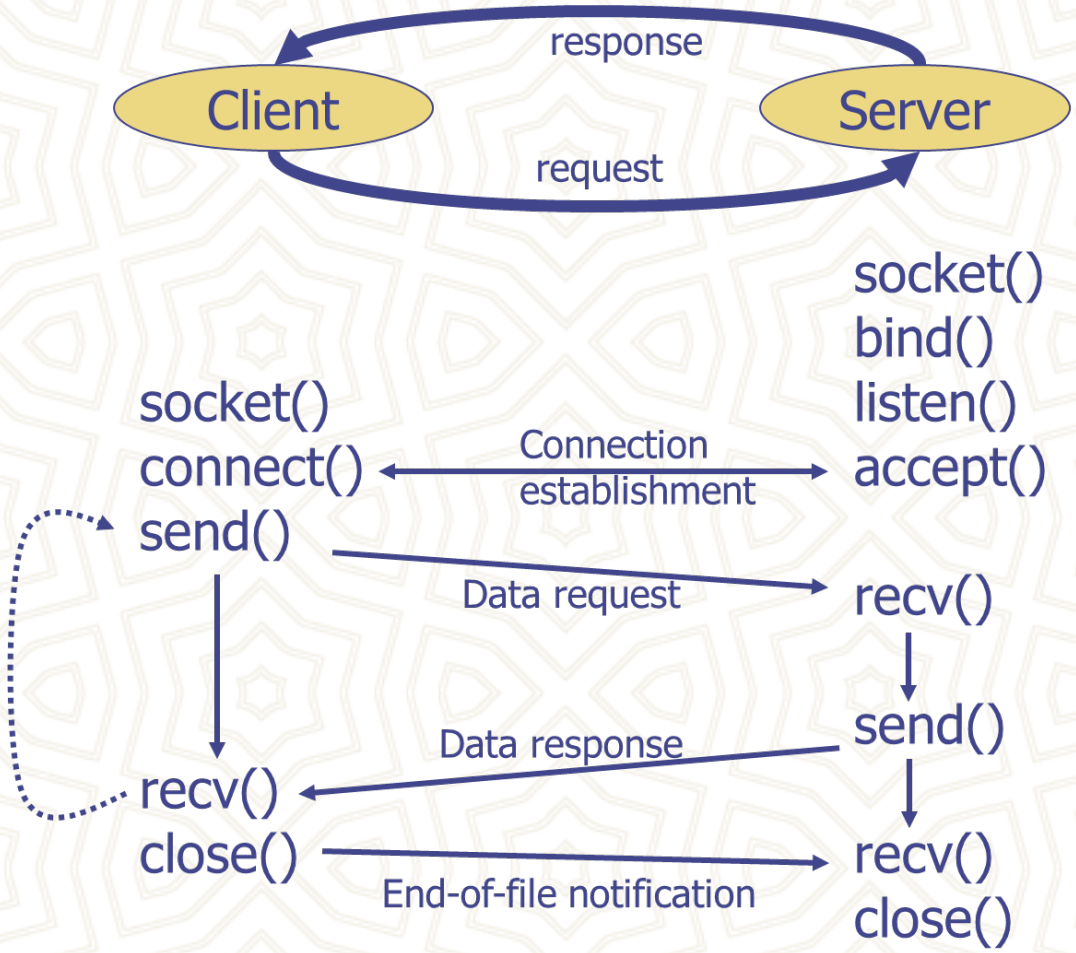
Name and Addressing

- Host name
  - identifies a single host (see Domain Name System slides)
  - variable length string (e.g. www.berkeley.edu)
  - is mapped to one or more IP addresses
- IP Address
  - written as dotted octets (e.g. 10.0.0.1)
  - 32 bits. Not a number! But often needs to be converted to a 32-bit to use.
- Port number
  - identifies a process on a host
  - 16 bit number

Client-Server Architecture

- Client requests service from server
- Server responds with sending service or error message to client

Simple Client-Server Example



UDP Client – Server

CLIENT

- Create stream socket
  - *socket()*
- While
  - *sendto()*
  - *recvfrom()*
- Close the Socket
  - close()

SERVER

- Create stream socket
  - *socket()*
- Bind port to socket
  - *bind()*
- While
  - *recvfrom()*
  - *sendto()*
- Close the Socket
  - close()

TCP Client – Server

CLIENT

- Create stream socket
  - *socket()*
- Connect to server
  - *connect()*
- While still connected:
  - *send()*
  - *recv()*
- Close TCP connection and Socket
  - close()

SERVER

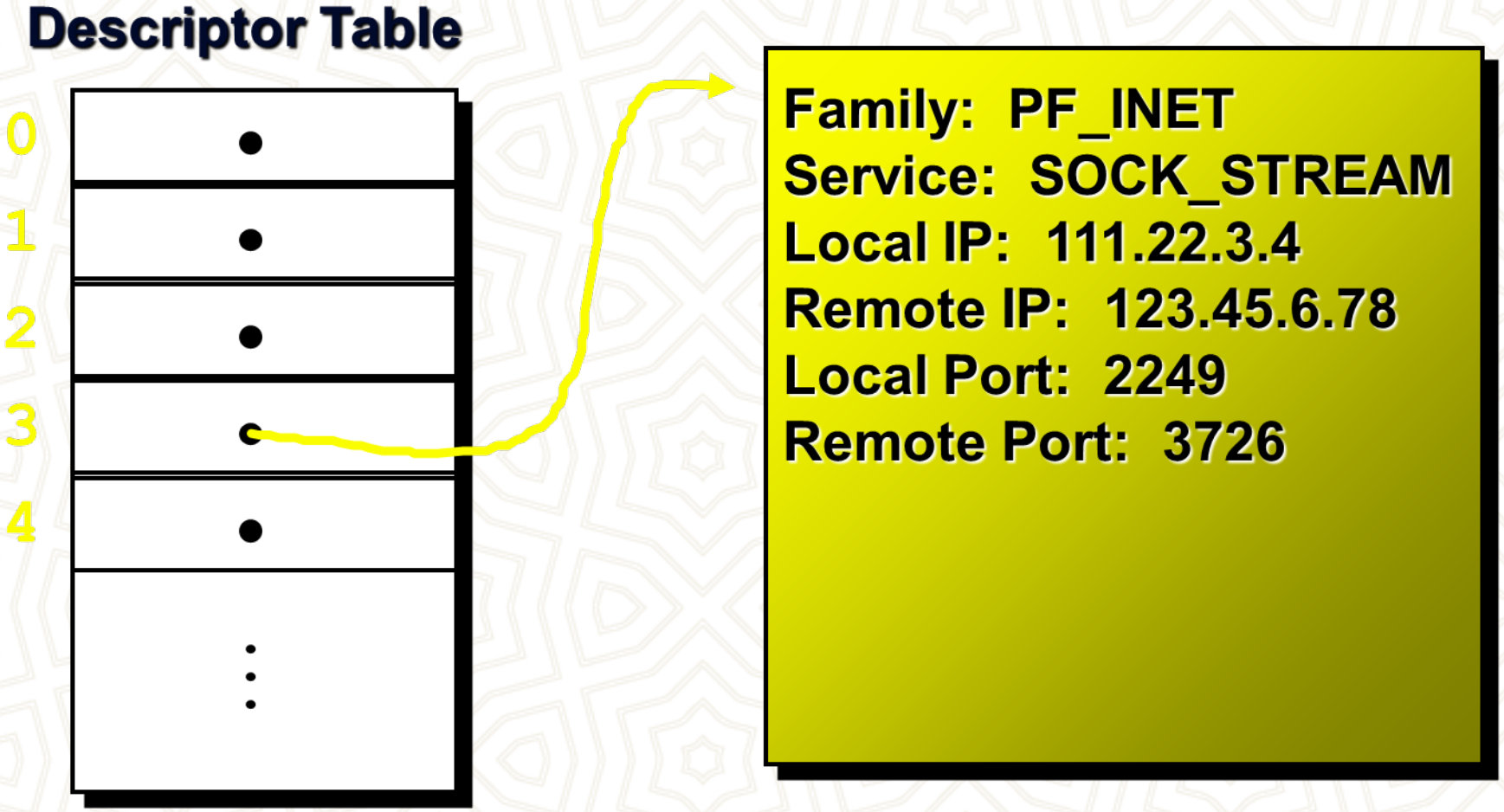
- Create stream socket
  - *socket()*
- Bind port to socket
  - *bind()*
- Listen for new client
  - *listen()*
- While
  - *accept()*
  - *recv()*
  - *send()*)
- Close TCP connection and Socket
  - close()

Creating a Socket

int socket(int family,int type,int proto);

- family specifies the protocol family (**PF_INET** for TCP/IP).
- type specifies the type of service (**SOCK_STREAM**, **SOCK_DGRAM**).
- protocol specifies the specific protocol
  - **AF_INET, AF_IPX, AF_PACKET**

Socket Descriptor Data Structure



Assigning an address to a socket

- The **`bind()`** system call is used to assign an address to an existing socket.

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

const

- **`bind`** returns 0 if successful or -1 on error.

Generic socket addresses

```
struct sockaddr {
      sa_family_t     sa_family;
      char            sa_data[14];
 };
```

- **sa_family** specifies the address type.
- **sa_data** specifies the address value.

struct sockaddr_in (IPv4)

```
struct sockaddr_in {
    short int          sin_family;    // Address family
    unsigned short int sin_port;      // Port number
    struct in_addr     sin_addr;      // Internet address
    unsigned char      sin_zero[8];
    };

    struct in_addr {
        unsigned long s_addr;         // 4 bytes
    };
```

- **Padding of** sin_zeros**:** struct sockaddr_in **has same size as** struct sockaddr

Bind Example

```
int mysock,err;
struct sockaddr_in myaddr;

  mysock = socket(PF_INET,SOCK_STREAM,0);
  myaddr.sin_family = AF_INET;
  myaddr.sin_port = htons( portnum );
  myaddr.sin_addr = htonl( ipaddress);

  err=bind(mysock, (sockaddr *) &myaddr,
      sizeof(myaddr));
```

Sending UDP Datagrams

```
ssize_t sendto( int sockfd,
                void *buff,
                size_t nbytes,
                int flags,
                const struct sockaddr* to,
                socklen_t addrlen);
```

sockfd is a UDP socket

buff is the address of the data (**nbytes** long)

to is the address of a sockaddr containing the destination address.

Return value is the number of bytes sent, or -1 on error.

More sendto()

- The return value of `sendto()` indicates how much data was accepted by the O.S. for sending as a datagram - not how much data made it to the destination.
- There is no error condition that indicates that the destination did not get the data!!!

Receiving UDP Datagrams

```
ssize_t recvfrom( int sockfd,
                  void *buff,
                  size_t nbytes,
                  int flags,
                  struct sockaddr* from,
                  socklen_t *fromaddrlen);
```

`sockfd` is a UDP socket

`buff` is the address of a buffer (**`nbytes`** long)

`from` is the address of a sockaddr.

Return value is the number of bytes received and put into buff, or -1 on error.

More on recvfrom()

- If buff is not large enough, any extra data is lost forever...
- You can receive 0 bytes of data!
- The **`sockaddr`** at **`from`** is filled in with the address of the sender.
- You should set **`fromaddrlen`** before calling.
- If **`from`** and **`fromaddrlen`** are NULL we don't find out who sent the data.

UDP Server

```c
#define BUFLEN 512
#define NPACK 10
#define PORT 9930

void diep(char *s)  {

        perror(s);
        exit(1);
}

int main(void)  {

        struct sockaddr_in si_me, si_other;
        int s, i, slen=sizeof(si_other);
        char buf[BUFLEN];

        if ((s=socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP))==-1)
          diep("socket");

        memset((char *) &si_me, 0, sizeof(si_me));
        si_me.sin_family = AF_INET;
        si_me.sin_port = htons(PORT);
        si_me.sin_addr.s_addr = htonl(INADDR_ANY);
        if (bind(s, &si_me, sizeof(si_me))==-1)
            diep("bind");

        for (i=0; i<NPACK; i++) {
          if (recvfrom(s, buf, BUFLEN, 0, (const struct sockaddr *) &si_other, &slen)==-1)
            diep("recvfrom()");
          printf("Received packet from %s:%d\nData: %s\n\n",
                 inet_ntoa(si_other.sin_addr), ntohs(si_other.sin_port), buf);
        }

        close(s);
        return 0;
}
```

UDP Client

```c
#define BUFLEN 512
#define NPACK 10
#define PORT 9930
#define SRV_IP "127.0.0.1"

void diep(char *s)  {

        perror(s);
        exit(1);
}

int main(void)  {

        struct sockaddr_in si_other;
        int s, i, slen=sizeof(si_other);
        char buf[BUFLEN];

        if ((s=socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP))==-1)
          diep("socket");

        memset((char *) &si_other, 0, sizeof(si_other));
        si_other.sin_family = AF_INET;^M
        si_other.sin_port = htons(PORT);^M
        if (inet_aton(SRV_IP, &si_other.sin_addr)==0) {
          fprintf(stderr, "inet_aton() failed\n");
          exit(1);
        }

        for (i=0; i<NPACK; i++) {
          printf("Sending packet %d\n", i);
          sprintf(buf, "This is packet %d\n", i);
          if (sendto(s, buf, BUFLEN, 0, (const struct sockaddr *) &si_other, slen)==-1)
            diep("sendto()");
        }

        close(s);
        return 0;
}
```

Wait for Connections listen()

int listen(int sockfd, int backlog);

- Puts socket in a listening state, willing to handle incoming TCP connection request.
- Backlog: number of TCP connections that can be queued at the socket.

Accept Connections accept()

int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);

- The **accept**() system call is used with connection-based socket types (**SOCK_STREAM**, **SOCK_SEQPACKET**).
- It extracts the first connection request on the queue of pending connections for the listening socket, *sockfd*, creates a new connected socket, and returns a new file descriptor referring to that socket.
- The newly created socket is not in the listening state. The original socket *sockfd* is unaffected by this call.

Sending Packets - send()

```
int send_packets(char *buffer, int buffer_len) {
        sent_bytes = send(chat_sock, buffer, buffer_len, 0);
if (send_bytes < 0) {
                perror (“send");
}
return 0;
}
```

- Needs socket descriptor,
- Buffer containing the message, and
- Length of the message
- Can also use write()

Receiving packets recv()

ssize_t recv(int *sockfd*, **void ***buf*, **size_t** *len*, **int** *flags*);**

Wait for a maximum of length octets of data on the SOCK_STREAM socket sockfd write data to buffer

A Simple TCP Client – Server

- A Hello World example
  - Server listens at a predefined port
  - Accepts incoming connections
  - Sends Hello World string to clients

- Extra features
  - Address resolution

TCP Client

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

#define PORT 3490               // the port client will be connecting to
#define MAXDATASIZE 100         // max number of bytes we can get
                                // at once
int main(int argc, char *argv[]) {
        int     sockfd, numbytes;
        char    buf[MAXDATASIZE];
        struct  hostent *he;
        struct  sockaddr_in their_addr; // server's address information

        if (argc != 2) {
                fprintf(stderr,"usage: client hostname\n");
                exit(1);
        }

        if ((he=gethostbyname(argv[1])) == NULL) {  // get the host info
                perror("gethostbyname");
                exit(1);
        }

        if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
                perror("socket");
                exit(1);
        }
```

TCP Client

```
        their_addr.sin_family = AF_INET;          // host byte order
        their_addr.sin_port = htons(PORT);       // short, network byte order
        their_addr.sin_addr = *((struct in_addr *)he->h_addr); // already network byte order
        memset(&(their_addr.sin_zero), '\0', 8); // zero the rest of the struct

        if (connect(sockfd, (struct sockaddr *)&their_addr, sizeof(struct sockaddr)) == -1){
                perror("connect");
                exit(1);
        }

        if ((numbytes=recv(sockfd, buf, MAXDATASIZE-1, 0)) == -1) {
                perror("recv");
                exit(1);
        }

        buf[numbytes] = '\0';
        printf("Received: %s",buf);
        close(sockfd);
        return 0;
}
```

TCP Server

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

#define MYPORT 3490    // the port users will be connecting to
#define BACKLOG 10     // how many pending connections queue will hold

int main(void) {

        int sockfd, new_fd;     // listen on sockfd, new connection on new_fd
        struct sockaddr_in my_addr;     // my address information
        struct sockaddr_in their_addr;   // connector's address information
        socklen_t sin_size;

        if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {         perror("socket");
                exit(1);
        }

        my_addr.sin_family = AF_INET;                 // host byte order
        my_addr.sin_port = htons(MYPORT);             // short, network byte order
        my_addr.sin_addr.s_addr = INADDR_ANY; // auto. filled with local IP
        memset(&(my_addr.sin_zero), '\0', 8);          // zero the rest of the struct
```

TCP Server

```
        if (bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr)) == -1) {
                perror("bind");
                exit(1);
        }

        if (listen(sockfd, BACKLOG) == -1) {
                perror("listen");
                exit(1);
        }

        while(1) {      // main accept() loop
                sin_size = sizeof(struct sockaddr_in);
                if ((new_fd = accept(sockfd, (struct sockaddr *)&their_addr, &sin_size)) == -1) {
                        perror("accept");
                        continue;
                }

                printf("server: got connection from %s\n", inet_ntoa(their_addr.sin_addr));

                if (send(new_fd, "Hello, world!\n", 14, 0) == -1)
                        perror("send");

                close(new_fd);
        }
        return 0;
}
```

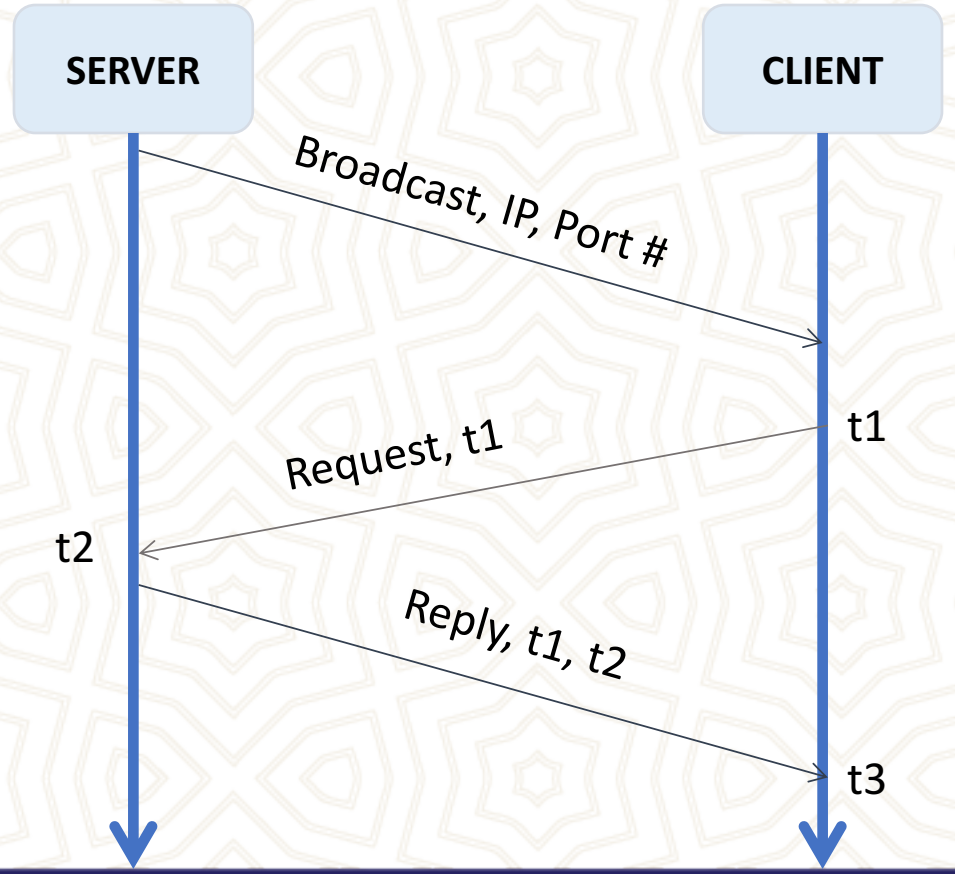
More things to know

- Howto send broadcast packages ?
  - UDP
  - TCP
- Howto implement non-blocking socket ?
  - UDP
  - TCP
- Howto implement **concurrent** servers

Assignment II

- Connection oriented Client-Server application
  - Server broadcasts its information
    - IP Address and port number
    - When receives a packet from the client;
      - it takes system time
      - append this information to the packet
      - sends it back.
  - Clients listens
    - When the server information is available, it connects to the server
    - When connected, it sends a packet with time information
    - When receives the packet back
      - gets time information from the packet
      - Calculate delay for both packets
      - Calculate the turn around time

Assignment II



References

- Man pages
- Jörn Altmann‘s Slides
- *UNIX Network Programming, Volume 1, Second Edition: Networking APIs: Sockets and XTI*