Sockets

OPERATING SYSTEMS COURSE

THE HEBREW UNIVERSITY

SPRING 2019

Overview

- **Networking Protocols - Motivation**
- TCP and UDP Transport Protocols
- Sockets
- Technical Material:
  - Socket’s Address
  - DNS and Sockets programing

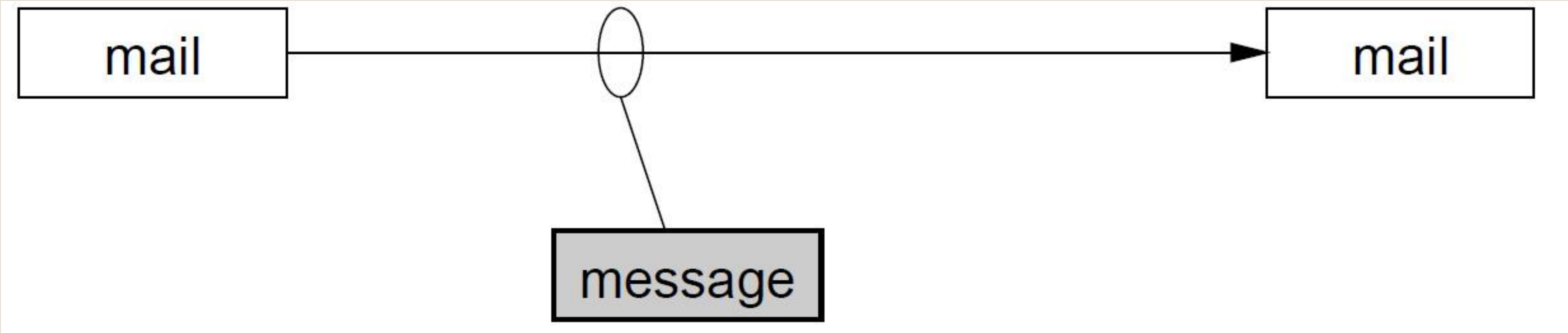
Communication Protocols

- A communications protocol is a system of digital rules for data exchange within or between computers
- Communicating systems use well-defined formats for exchanging messages.
- A protocol must define the syntax, semantics, and synchronization of communication

- Computer networking protocols' implementation is called **protocol stack** or **network stack**

Sending mails

- Email contains
  - Address
  - Data



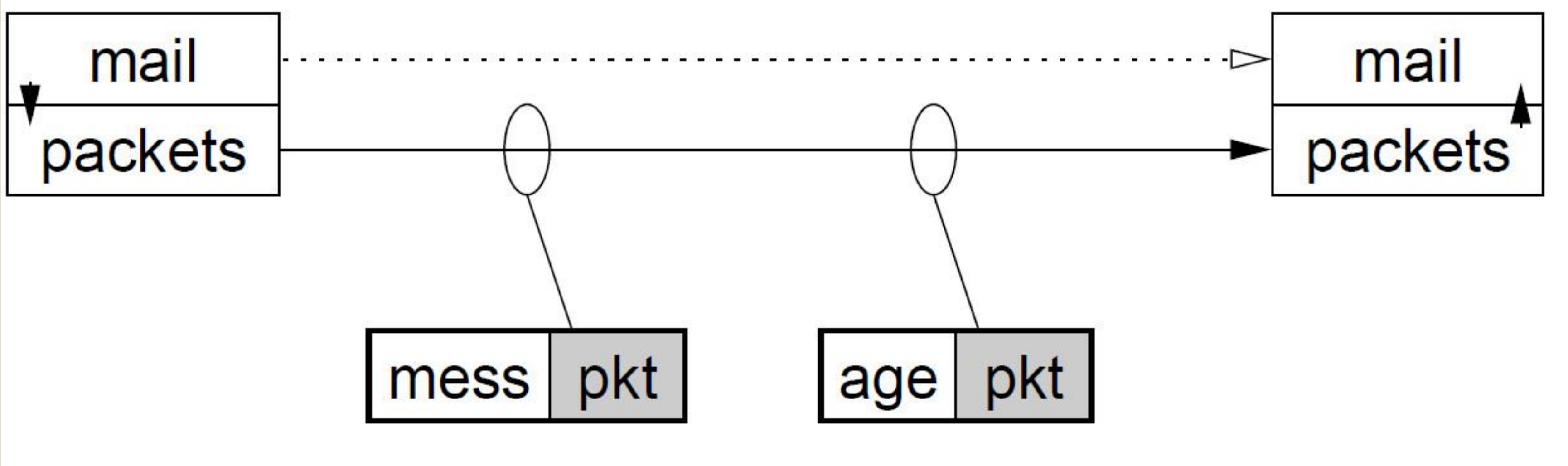
The problem of long messages

- Sending long messages is problematic
  - HW errors have more impact
  - One "wrong bit" and all the message is thrown.

- Simple Solution - users are allowed to send bounded size of messages (e.g. 1K)
  - Not practical.

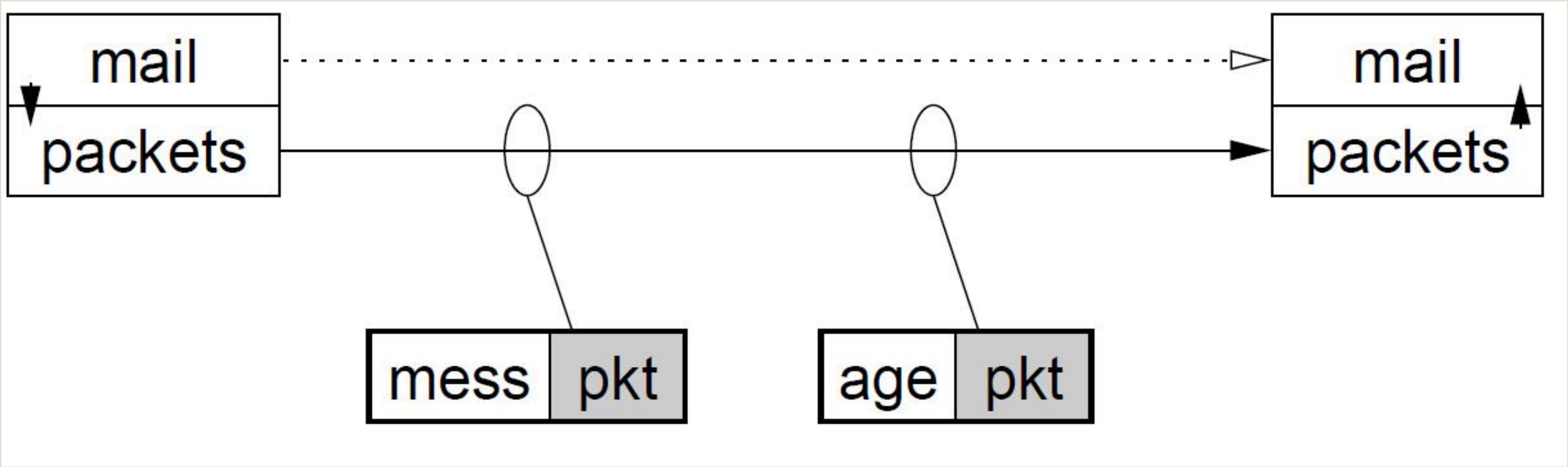
Packetization of long messages

- Adding a “program”, in both edges, that is responsible to break the long messages into shorter ones.



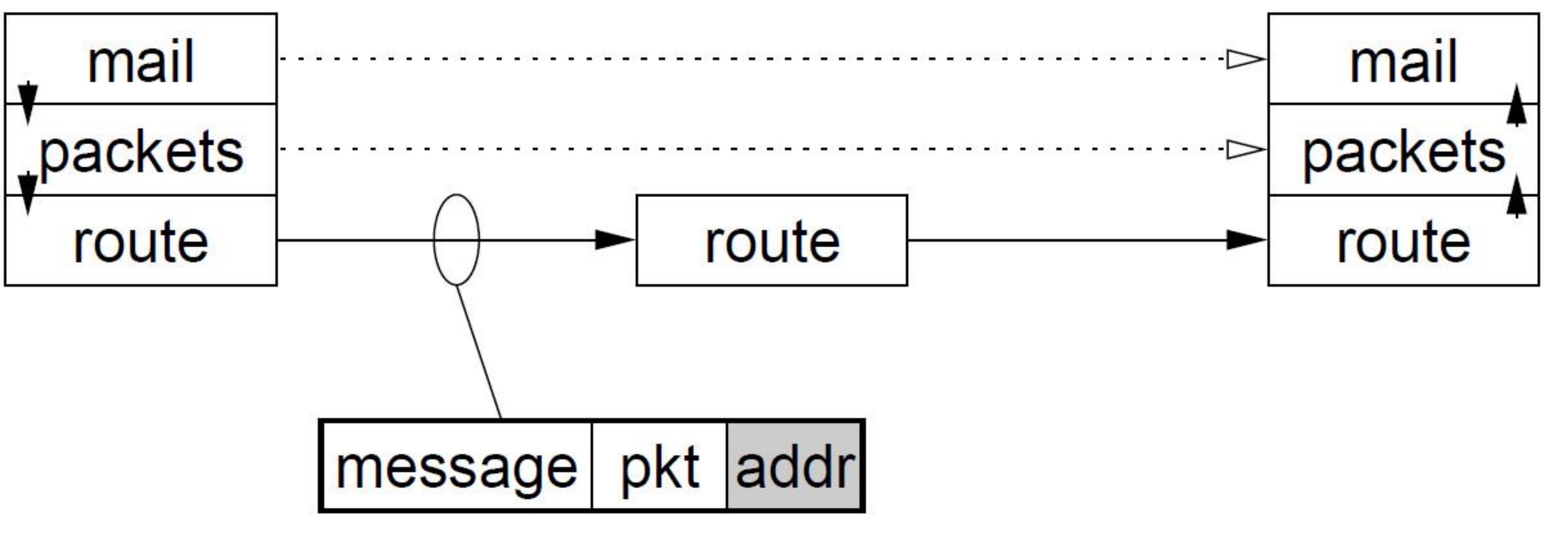
End-to-end control

- After splitting to packets, two problems may occur
  - Lost packet
  - A packet overtakes a previous packet
- End to end control handles these problems



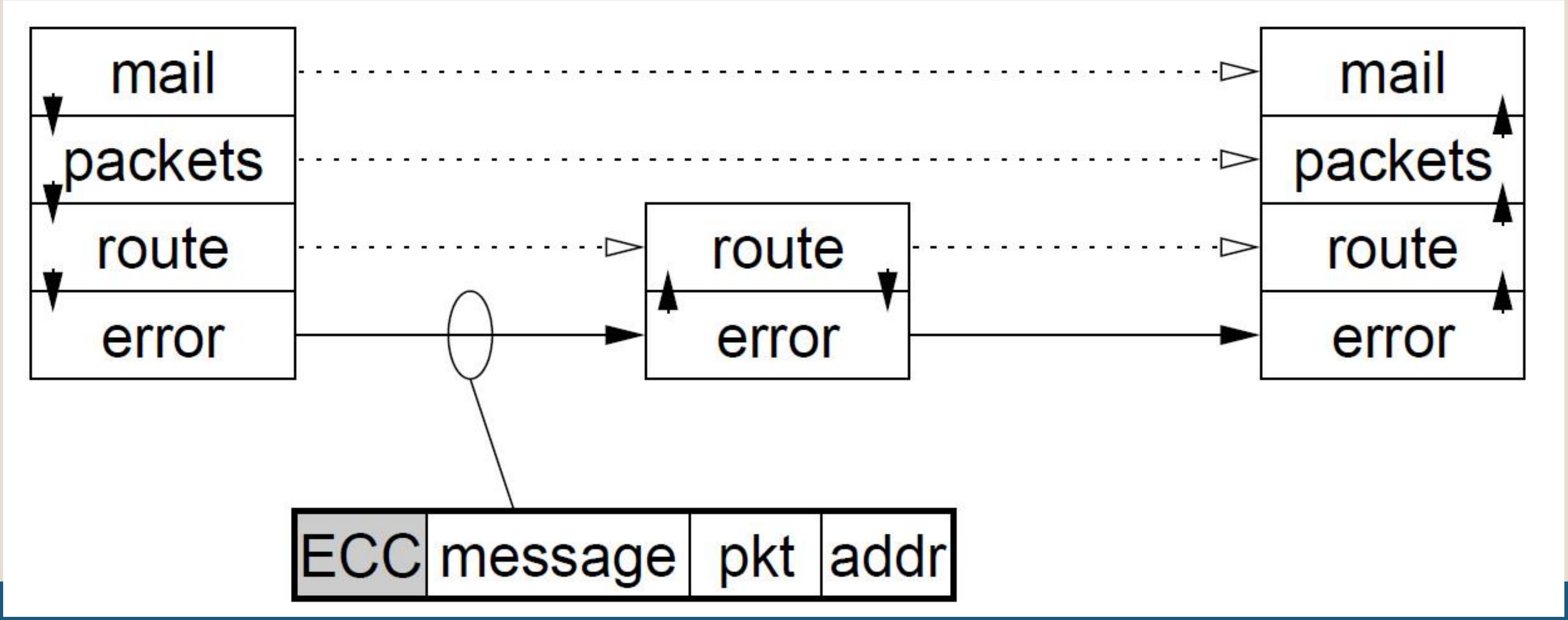
Routing

- If there is no direct link between the computers, a routing is needed
- Routing's information is added to each packet



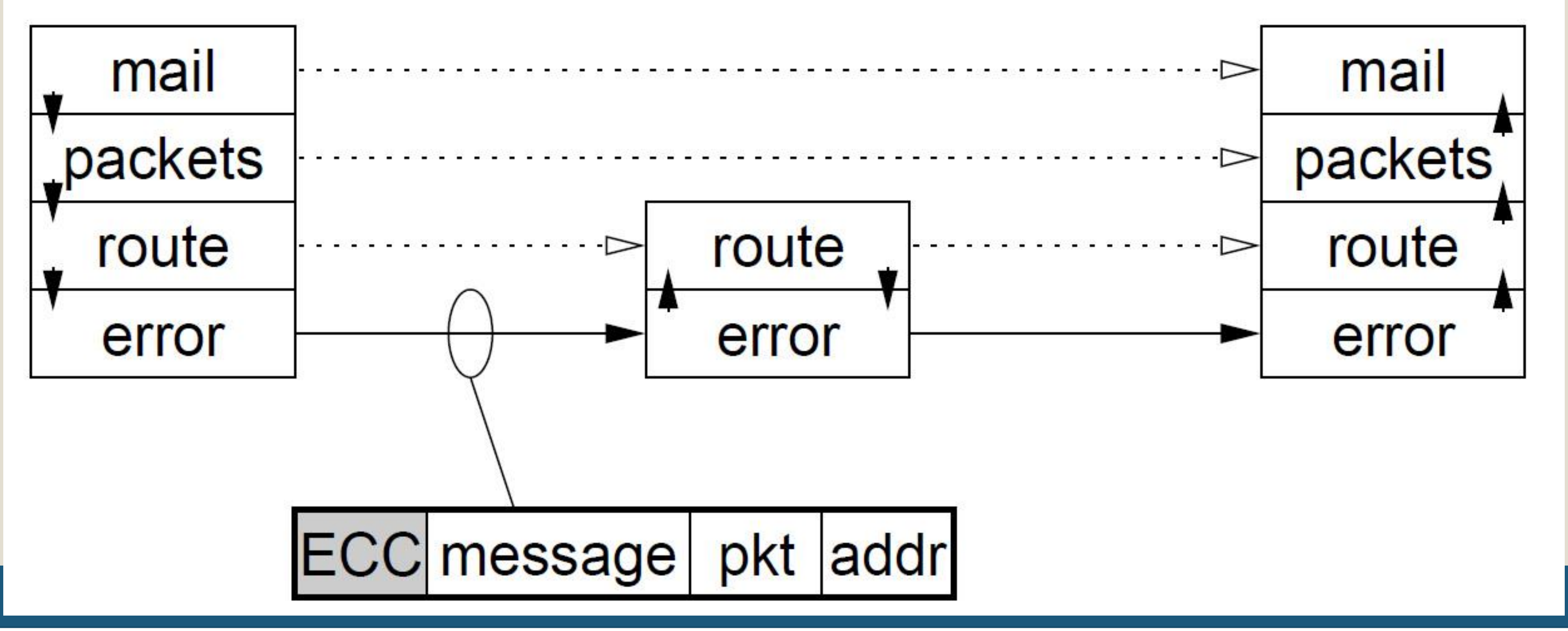
Errors Correction

- Due to noise, not all the packets are sent successfully.
- Additional error-correction code (EEC) is added to the header for handling that.



Protocol Stack

- The sending side - adds the headers.
- The receiving side - use the headers in each layer and then, remove them.
- That way, each layer can "talk directly" with its counter part on the other machine



Internet protocol suite (TCP/IP) – The protocol stack that is used by the Internet

Layer name	Description (Layer's goal)	Protocols
Application	process-to-process communications	HTTP/S, SSH, FTP, DNS
Transport	End-to-end communication services for applications	TCP, UDP
Network / Internet	Transport datagrams (packets) from the originating host across network boundaries, if necessary, to the destination host specified by a network address	IP
Link / Physical	Communications protocols that only operate on the link that a host is physically connected to.	802.11 WiFi, Ethernet

Overview

- Networking Protocols - Motivation
- **TCP and UDP Transport Protocols**
- Sockets
- Technical Material:
  - Socket’s Address
  - DNS and Sockets programing

Transport Layer

- Network Layer (IP) is:
  - Responsible for **end to end transmission**
  - **Unreliable -** Packets might be lost, duplicated, corrupted, delivered out of order.
- Supplies End-to-end communication services for applications.
- Main protocols
  - TCP
  - UDP

Layer
Application
Transport
Network / Internet
Link / Physical

User Datagram Protocol (UDP)

- Thin layer on top of IP
- Also **source and destination *ports***
  - Ports associate a packet with a specific application at each end.
- Adds packet **length + checksum**
  - Guard against corrupted packets
- Still **unreliable**:
  - Duplication, loss, out-of-order is possible.
- **Connectionless**
- **UDP header:**

0 16 31

Source Port	Destination Port
Length	Checksum
Application data	

Transmission Control Protocol (TCP)

- Reliable stream transport
- Connection oriented
- Two ends communicate to agree on details
- Buffering
- Flow control and Congestion Control
- Takes care of lost packets, out of order, duplicates, long delays

Transport Layer Summary

Property	UDP	TCP
Reliable	no	yes
Connection type	Connectionless	Connection oriented
Flow control	No	Yes
Latency	Low	High
Applications	VOIP, Most games	HTTP, HTTPs, FTP, SMTP, Telnet, SSH

Overview

- Networking Protocols - Motivation
- TCP and UDP Transport Protocols
- **Sockets**
- Technical Material:
  - Socket's Address
  - DNS and Sockets programing

Socket programming

Goal: learn how to build client/server application that communicate using sockets

Socket API

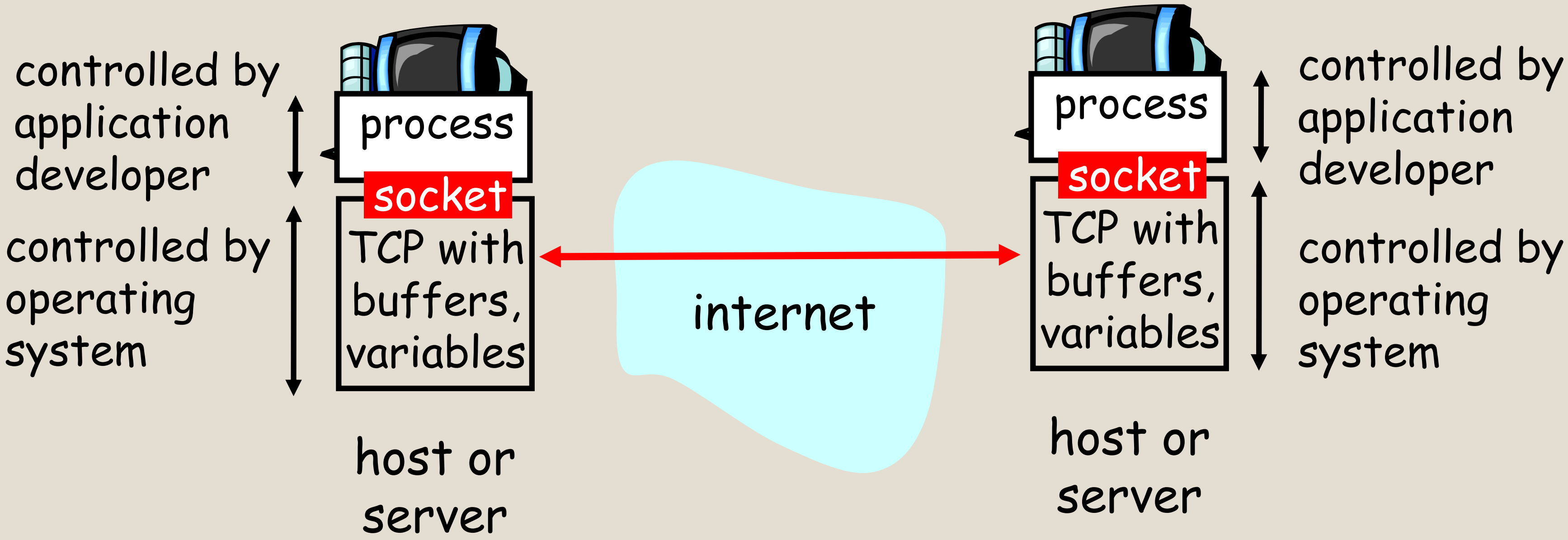
- Explicitly created, used, released by applications
- Client/server paradigm
- Two types of transport service via socket API:
  - unreliable datagram
  - reliable, byte stream-oriented

socket

a *host-local*, *application-created*, *OS-controlled* interface (a "door") into which application process can both send and receive messages to/from another application process

Socket-Programming using TCP

TCP service: reliable transfer of **bytes** from one process to another



Socket programming with TCP

Client must contact server

- server process must first be running
- server must have created socket (door) that welcomes client's contact

Client contacts server by:

- creating client-local TCP socket
- specifying IP address and port number of server process
- When client creates socket: client TCP establishes connection to server TCP

- When contacted by client, server TCP creates **new** socket to communicate with client
  - allows server to talk with multiple clients
  - source port numbers used to distinguish clients

application viewpoint

TCP provides reliable, in-order transfer of bytes between client and server

Streams

- A **stream** is a sequence of characters that flow into or out of a process.
- An **input stream** is attached to some input source for the process, e.g. keyboard or socket.
- An **output stream** is attached to an output source, e.g. monitor or socket.

Socket programming with TCP

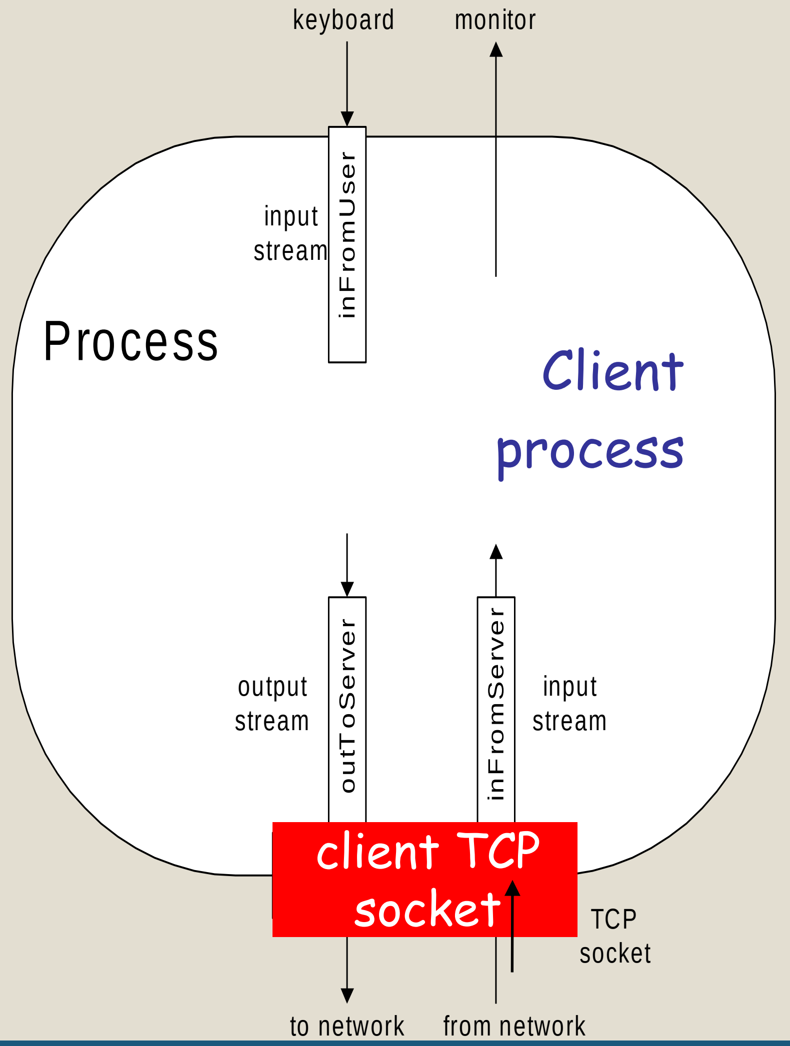
Example client-server app:

1) client reads line from standard input (**`inFromUser`** stream), sends to server via socket (**`outToServer`** stream)

2) server reads line from socket

3) server converts line to uppercase, sends back to client

4) client reads, prints modified line from socket (**`inFromServer`** stream)



Client/server socket interaction: TCP

Server (running on **hostip**)

create socket, port=**x**,
for incoming request:

welcomeSocket =
ServerSocket()

wait for incoming
connection request
connectionSocket =
welcomeSocket.accept()

read request from
connectionSocket

write reply to connectionSocket

Close connectionSocket

TCP
connection setup
"handshake"

Client

create socket,
connect to **hostip**, port=**x**

clientSocket = Socket()

send request using
clientSocket

read reply from
clientSocket

Close clientSocket

Socket programming with UDP

UDP: no "connection state" between client and server

- No handshaking
- Sender explicitly attaches IP address and port of destination to each packet
- Server must extract IP address, port of the sender

UDP: transmitted data may be received out of order, or lost

application viewpoint

UDP provides unreliable transfer of groups of bytes ("datagrams") between client and server

Client/server socket interaction: UDP

Server (running on `hostip`)

create socket, port=**x**,
for incoming request:
serverSocket =
DatagramSocket()

read request from serverSocket

write reply to erverSocket
specifying client host
address, port number

Client

create socket,
clientSocket = DatagramSocket()

Create, address
(**hostip,port=x**)
send datagram request using
clientSocket

read reply from clientSocket

Close clientSocket

Unlike TCP, the same serverSocket stays the same, even for several clients

Overview

- Motivation to protocol stack
- TCP and UDP Transport Protocols
- Sockets – concept

- Technical Material:
  - **Socket's Address**
  - DNS (mapping names to IP)
  - Sockets programing

struct sockaddr


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

- Address family in this presentation: **`AF_INET`**
- Contains a destination address and port number for the socket.

Socket's Address Format

- The addressing format of a socket:
  - **AF_UNIX** addressing uses UNIX pathnames to identify sockets - these sockets are very useful for inter-process communication (**IPC**) on the same machine.
  - **AF_INET** addressing uses Internet protocol addresses (**IP**s) which are four-byte numbers usually written as four decimal numbers separated by periods (such as 192.9.200.10). In addition to the machine address, there is also a port number which allows more than one AF_INET socket on each machine.
- We focus on **AF_INET** addresses, as they are the most useful and widely used.

struct sockaddr_in


```
struct sockaddr_in {
  short                sin_family;
  unsigned short       int sin_port;
  struct in_addr       sin_addr;
  unsigned char        sin_zero[8];
};


struct in_addr {
  uint32_t    s_addr;
};
```

This structure makes it easy to reference elements of the socket address.

struct sockaddr_in

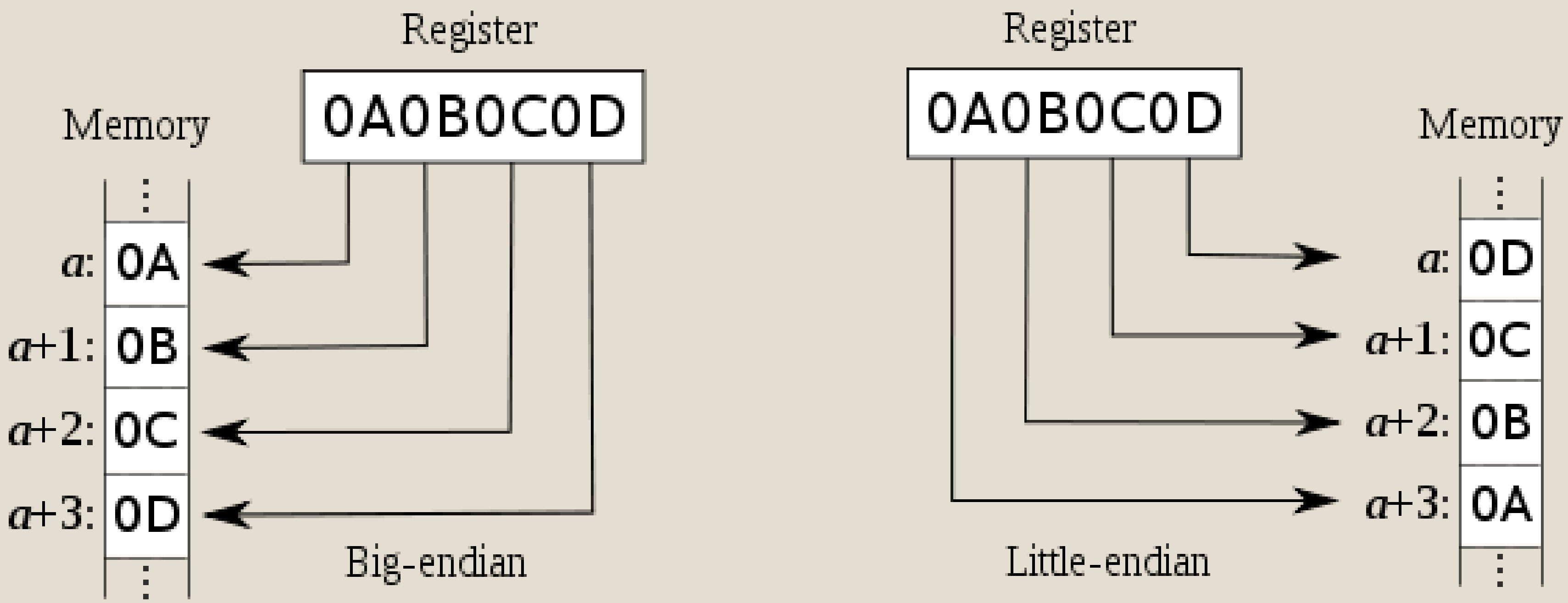
- A pointer to a **struct sockaddr_in** can be cast to a pointer to a **struct sockaddr** and vice-versa.
- Note that **sin_zero** should be set to all zeros with the function **memset().**
- **sin_family** corresponds to **sa_family** in a **sockaddr** and should be set to **"AF_INET".**
- **sin_port** and **sin_addr** must be in *Network Byte Order*!

Structs and Data Handling

- There are two byte orderings:
  - Most significant byte first.
  - Least significant byte first.

- In order to convert "Host Byte Order“ to Network Byte Order, you have to call a function.

Big\Little Endian



Big-endian (“Network bytes order”) is the most common convention in data networking

Little-endian is popular (though not universal) among microprocessors

Conversion Functions

- There are two types that you can convert: short and long. These functions work for the unsigned variations as well:
  - **`htons()`** - "Host to Network Short“
  - **`htonl()`** - "Host to Network Long“
  - **`ntohs()`** - "Network to Host Short“
  - **`ntohl()`** - "Network to Host Long“
- Be portable! Remember: put your bytes in “Network Byte Order” before you put them on the network.

IP Addresses


```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
struct sockaddr_in my_addr;
my_addr.sin_family = AF_INET;
my_addr.sin_port = htons(3490);
inet_aton("10.12.110.57",&(my_addr.sin_addr));
memset(&(my_addr.sin_zero), '\0', 8);
```

inet_aton():

- Convert address from the Ip V4 numbers-and-dots notation into binary form (in network byte order).
- Unlike practically every other socket related function, returns non-zero on success, and zero on failure.

getpeername

int getpeername(int sockfd,struct sockaddr *addr,int *addrlen);

- **Description**: Get the address of the other end of a connected stream socket.
- **Return value**: 0 on success, -1 in case of an error.
- **Arguments**:
  - ***sockfd*** - the FD of the connected stream socket.
  - **addr** is a pointer to a **struct sockaddr** that will hold the information about the other side of the connection.
  - **addrlen** indicates on the addr’s length. Should be initialized to **sizeof(struct sockaddr).** If the value is not big enough, getpeername increases this value

Overview

- Motivation to protocol stack
- TCP and UDP Transport Protocols
- Sockets – concept

- Technical Material:
  - Socket's Address
  - **DNS (mapping names to IP)**
  - Sockets programing

Domain Name Service

- DNS is a service which maps human-readable address (a.k.a. host names) to IP addresses.
- The function **`gethostname()`** returns the name of the computer that your program is running on.
- The name can then be used to determine the IP address of your local machine:

```
#include <netdb.h>
struct hostent*
gethostbyname(const char *name);
```

- returns a pointer to the filled ***`struct hostent`***, or **`NULL`** on error.

struct hostent

```
struct hostent {
   //Official name of the host
   char *h_name;
   //Alternate names
   char **h_aliases;
   //usually AF_INET
   int h_addrtype;
   //length of each address
   int h_length;
   //network addresses for the host in
  N.B.O
   char **h_addr_list;
};

#define h_addr h_addr_list[0]
```

Example – getip program

Demonstration of how to use

`gethostbyname`

`struct hostent`

```
int main(int argc, char *argv[]) {
  struct hostent *h;
  if (argc != 2) {
     fprintf(stderr, "usage: getip 
  address\n");
      exit(1);
  }
  if ((h=gethostbyname(argv[1])) == 
  NULL) {
     fprintf(stderr, "gethostbyname ");
     exit(1);
  }
  printf("Host name : %s\n", h-
  >h_name);
  printf("IP Address : %s\n",
      inet_ntoa(*((struct in_addr *)h-
  >h_addr)));

  return 0;
}
```

Overview

- Motivation to protocol stack
- TCP and UDP Transport Protocols
- Sockets – concept

- Technical Material:
  - Socket's Address
  - DNS (mapping names to IP)
  - **Sockets programing**

First Step - Creating a Socket

- A socket is used to allow one process to speak to another, very much like the telephone is used to allow one person to speak to another.
- First, you must create a socket to listen for connections.
- This is done by using socket() function.
- *int socket(int domain, int type, int protocol);*

Socket's Type Parameter

- The type of the data in the socket:
  - **SOCK_STREAM** indicates that data will come across the socket as a stream of characters.
  - **SOCK_DGRAM** indicates that data will come in bunches (called *datagrams*).
  - **SOCK_RAW** allows bypassing the layers and writing/reading all bytes in the packet.

Socket's Protocol Parameter

- We can use a "0" value to choose the default protocol.
- Usually there is only one supported protocol.

Second Step – Binding an Address

- **Second Step:** give the socket an address to listen on.

- This is just as you get a telephone number so that you can receive calls using the **bind()** function.

- *int bind(int sockfd, const struct sockaddr *addr,*
  socklen_t addrlen);

Third Step – Listening

- Sockets have the **ability to queue incoming connection requests**, which is a lot like having "call waiting" for your telephone.
- If you are busy handling a connection, the connection request will wait until you can deal with it.
- The **listen()** function is used to recommend the maximum number of requests that will be queued before requests start being denied.
- *int listen(int sockfd, int backlog);*

Connection establishment (1)

This example demonstrates how we practically use the first three steps.

```
int establish(unsigned short portnum) {
    char myname[MAXHOSTNAME+1];
    int s;
    struct sockaddr_in sa;
    struct hostent *hp;
    //hostnet initialization
    gethostname(myname, MAXHOSTNAME);
    hp = gethostbyname(myname);
    if (hp == NULL)
        return(-1);
    //sockaddrr_in initlization
    memset(&sa, 0, sizeof(struct sockaddr_in));
    sa.sin_family = hp->h_addrtype;
    /* this is our host address */
    memcpy(&sa.sin_addr, hp->h_addr, hp->h_length);
    /* this is our port number */
    sa.sin_port= htons(portnum);
```

Taken from: http://www.mit.edu/afs.new/sipb/user/web/src/sockd.c

Connection establishment (2)

This example demonstrates how we practically use the first three steps.

```
        …

        /* create socket */
        if ((s= socket(AF_INET, SOCK_STREAM, 0)) < 0)
            return(-1);

        if (bind(s , (struct sockaddr *)&sa , sizeof(struct sockaddr_in)) < 0) {
            close(s);
            return(-1);
        }

        listen(s, 3); /* max # of queued connects */
        return(s);
}
```

Taken from: http://www.mit.edu/afs.new/sipb/user/web/src/sockd.c

Fourth Step – Waiting for Calls

- After creating a socket to get calls, you must wait for calls to that socket using the **accept()** function.
- Calling accept() is analogous to **picking up the telephone** if it's ringing.
- Accept() **returns a new socket** which is connected to the caller.
- *int accept(int sockfd, struct sockaddr *cli_addr, socklen_t *cli_addrlen)*

Accept connections

wait for a connection to occur on a socket created with establish()

```
int get_connection(int s) {
	int t; /* socket of connection */

	if ((t = accept(s,NULL,NULL)) < 0)
		return -1;
	return t;
}
```

The Client

- You now know how to create a socket that will accept incoming calls. How do you call it?
- As with the telephone, you must first have the phone before using it to call. You use the socket() function to do this.
- After getting a socket and giving it an address, you use the connect() function to try to connect to a listening socket.
- *int connect(int sockfd, const struct sockaddr *serv_addr,*
  socklen_t addrlen);

Connect to a socket (1)

First part, init the address

```
int call_socket(char *hostname, unsigned short
                    portnum) {

    struct sockaddr_in sa;
    struct hostent *hp;
    int s;

    if ((hp= gethostbyname (hostname)) == NULL)
    {
           return(-1);
    }

    memset(&sa,0,sizeof(sa));
    memcpy((char *)&sa.sin_addr , hp->h_addr ,
              hp->h_length);
    sa.sin_family = hp->h_addrtype;
    sa.sin_port = htons((u_short)portnum);

    ....
```

Taken from: http://www.mit.edu/afs.new/sipb/user/web/src/sockd.c

Connect to a socket (2)

Second part, connect to the server

```
    ....

    if ((s = socket(hp->h_addrtype,
    SOCK_STREAM,0)) < 0) {
            return(-1);
    }

    if (connect(s, (struct sockaddr *)&sa , sizeof(sa))
    < 0) {
            close(s);
            return(-1);
    }

    return(s);
}
```

Sending and Reading Data

- Now that you have a connection between sockets you want to send data between them.

- The read() and write() functions are used to do this, just as they are for normal files.

- You **don't** get back the same number of characters that you asked for, so you must loop until you have read the number of characters that you want.

Read Data code

```
int read_data(int s, char *buf, int n) {
    int bcount;      /* counts bytes read */
    int br;          /* bytes read this pass */
    bcount= 0; br= 0;

    while (bcount < n) { /* loop until full buffer */
        br = read(s, buf, n-bcount))
        if ((br > 0)  {
            bcount += br;
            buf += br;
        }
        if (br < 1) {
            return(-1);
        }
    }
    return(bcount);
}
```

Server has multiple FD

- The server may have multiple FD (sockets)
  - One (or more) that listens to new connections.
  - FDs that created by **accept**() and are getting their service currently.

- To handle several FD, the server may
  1. Use multiple threads
     1. A thread per socket.
     2. Threads pool
  2. Use Select().

select

int select (int nfds, fd_set *read-fds, fd_set *write-fds, fd_set *except-fds, struct timeval *timeout)

- **Description**:
  - Blocks the calling process until there is activity on any of the specified sets of file descriptors, or until the timeout period has expired.
  - The file descriptors specified by the read-fds, write-fds and except-fds are checked to see if they are ready for reading, writing and checked for exceptional conditions.
  - A null pointer passed to ignore checking in this type.
  - A file descriptor is considered ready for reading if a read call will not block

select

int select (int nfds, fd_set *read-fds, fd_set *write-fds, fd_set *except-fds, struct timeval *timeout)

- **Return value**:
  if select succeeds, it returns the number of ready socket descriptions. select returns 0 if the time limit expires before any socket is selected. If there is an error, select returns -1.

select

int select (int nfds, fd_set *read-fds, fd_set *write-fds, fd_set *except-fds, struct timeval *timeout)

- **Arguments** :
  - ***nfds*** – specifies the maximal number of sockets to check.
  - **readfds** – specifies the file descriptors to be checked for being ready to read.
  - **writefds** – specifies the file descriptors to be checked for being ready to write.
  - **exceptfds** – specifies the file descriptors to be checked for error conditions pending.
  - **timeout** – controls how long the select() function shall take before timing out.

fd_set Manipulations

- fd_set - Represent a set of file descriptors.
- FD_ZERO(fd_set **fdset*);
  - Initializes the file descriptor set *fdset* to have zero bits for all file descriptors
- FD_CLR(int *fd*, fd_set **fdset*);
  - Clears the bit for the file descriptor *fd* in the file descriptor set *fdset*.
- FD_SET(int *fd*, fd_set **fdset*);
  - Sets the bit for the file descriptor *fd* in the file descriptor set *fdset*.
- FD_ISSET(int *fd*, fd_set **fdset*);
  - Returns a non-zero value if the bit for the file descriptor *fd* is set in the file descriptor set pointed to by *fdset*, and 0 otherwise.

Select flow

Example

```
MAX_CLIENTS = 30;
fd_set clientsfds;
fd_set readfds;

FD_ZERO(&clientsfds);
FD_SET(serverSockfd, &clientsfds);
FD_SET(STDIN_FILENO, &clientsfds);

While (stillRunning) {
    readfds = clientsfds;

    if (select(MAX_CLIENTS+1, &readfds, NULL,
                NULL, NULL) < 0) {
          terminateServer();
          return -1;
    }

    ....
```

Select flow

Example

```
....

if (FD_ISSET(serverSockfd, &readfds)) {
        //will also add the client to the clientsfds
        connectNewClient();
}

if (FD_ISSET(STDIN_FILENO, &readfds)) {
        serverStdInput();
}

else {
        //will check each client  if it's in readfds
        //and then receive a message from him
        handleClientRequest();
}
}
```

Summary Stream Socket

Server Side	Client Side
1. `socket();`	1. `socket();`
2. `bind();`	2. `connect();`
3. `listen();`	3. `read()/write();`
4. `accept();`	
5. `read()/write();`	

Socket programming with UDP

- There are many references on the web.

- A simple and a good one:
  www.abc.se/~m6695/udp.html