Networks (Tutorial). Week 4

Shinnazar Seytnazarov, PhD

Innopolis University

s.seytnazarov@innopolis.ru

January 28, 2022

Topic of the lecture

- Electronic Mail
- SMTP
- POP3
- IMAP4
- DNS
- P2P Applications
- Socket Programming with UDP and TCP

Topic of the tutorial

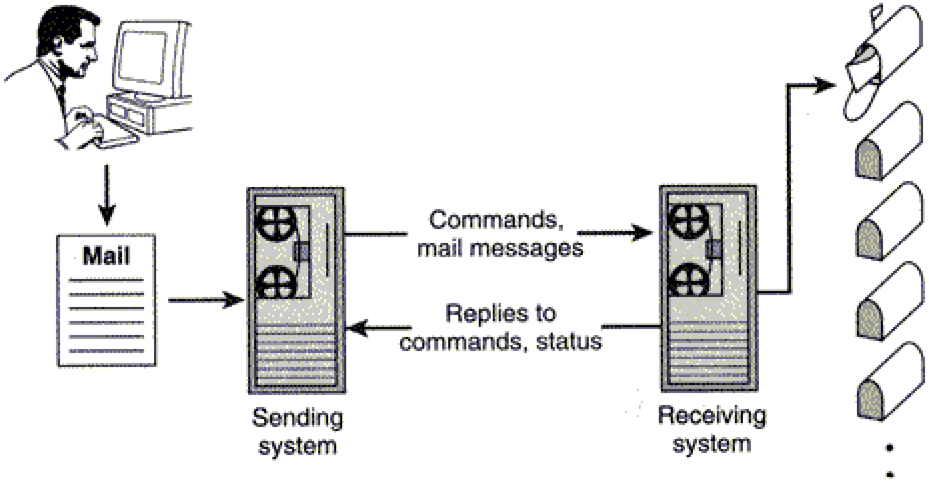
- Email Protocol Suit
- TCP Socket Programming

Topic of the lab

- TCP Socket Programming

Email

- E-mail has become an essential means to communicate.



The Mail Protocol Suite

- The most common mail protocols include:
  - SMTP (Simple Mail Transfer Protocol)
  - POP3 (Post Office Protocol)
  - IMAP4 (Internet Message Access Protocol)
  - MIME (Multipurpose Internet Mail Extensions)

Standards to Exchange Information

- ASCII stands for the "American Standard Code for Information Interchange".
- It was designed in the early 60's, as a standard character set for computers and electronic devices.
- ASCII is a 7-bit character set containing 128 characters.
- It contains the numbers from 0-9, the upper and lower case English letters from A to Z, and some special characters.
- The character sets used in modern computers and over the Internet.

ASCII Character Set

- For Example
  - ASCII complaint computer use:

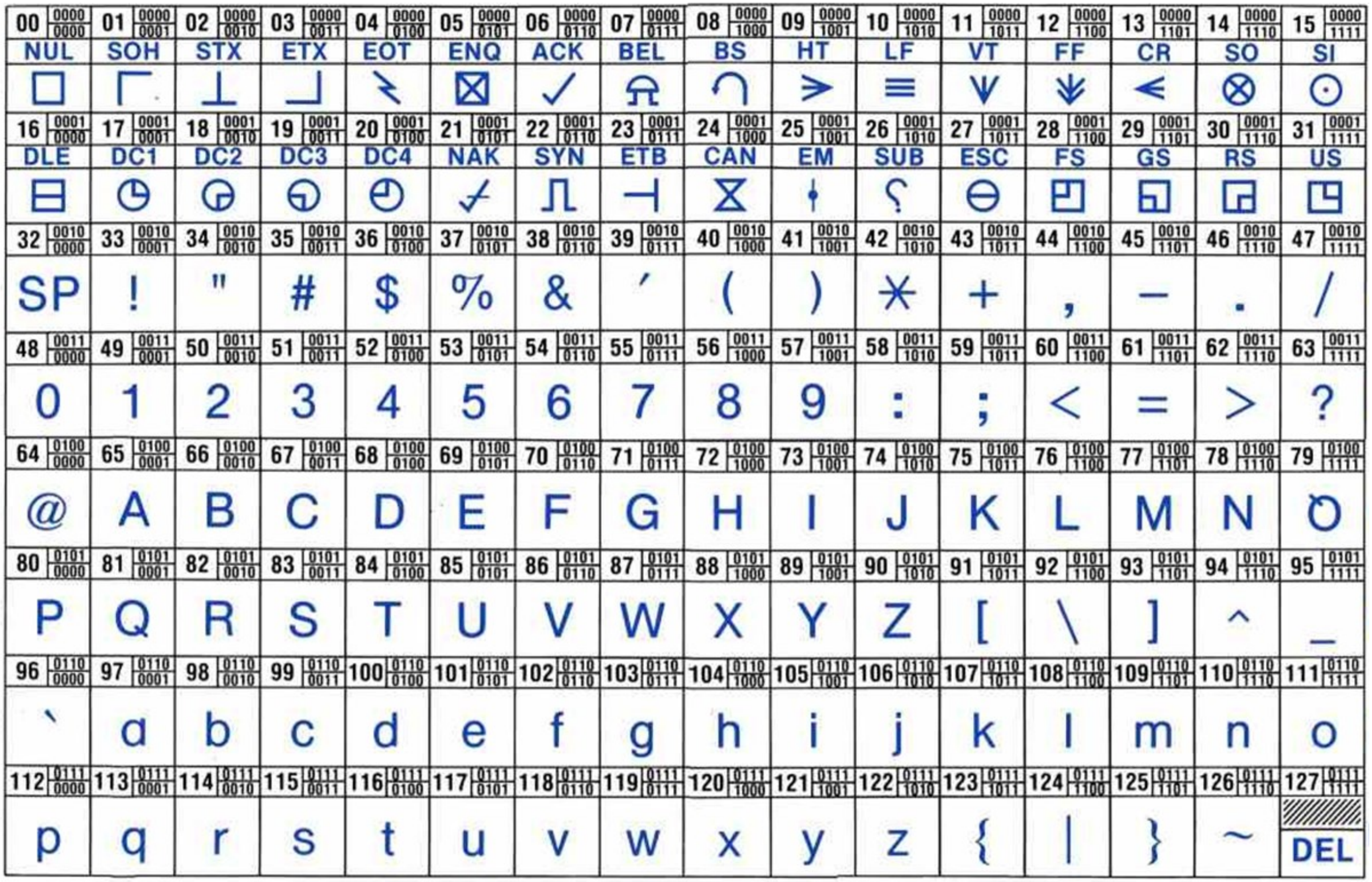
65 to represent A

Binary

0100 0001

This standard is unable to support many languages characters. For instance: Russian

ASCII Character Set



00 0000 0000 NUL	01 0000 0001 SOH	02 0000 0010 STX	03 0000 0011 ETX	04 0000 0100 EOT	05 0000 0101 ENQ	06 0000 0110 ACK	07 0000 0111 BEL	08 0000 1000 BS	09 0000 1001 HT	10 0000 1010 LF	11 0000 1011 VT	12 0000 1100 FF	13 0000 1101 CR	14 0000 1110 SO	15 0000 1111 SI
16 0001 0000 DLE	17 0001 0001 DC1	18 0001 0010 DC2	19 0001 0011 DC3	20 0001 0100 DC4	21 0001 0101 NAK	22 0001 0110 SYN	23 0001 0111 ETB	24 0001 1000 CAN	25 0001 1001 EM	26 0001 1010 SUB	27 0001 1011 ESC	28 0001 1100 FS	29 0001 1101 GS	30 0001 1110 RS	31 0001 1111 US
32 0010 0000 SP	33 0010 0001 !	34 0010 0010 "	35 0010 0011 #	36 0010 0100 $	37 0010 0101 %	38 0010 0110 &	39 0010 0111 '	40 0010 1000 (	41 0010 1001)	42 0010 1010 *	43 0010 1011 +	44 0010 1100 ,	45 0010 1101 -	46 0010 1110 .	47 0010 1111 /
48 0011 0000 0	49 0011 0001 1	50 0011 0010 2	51 0011 0011 3	52 0011 0100 4	53 0011 0101 5	54 0011 0110 6	55 0011 0111 7	56 0011 1000 8	57 0011 1001 9	58 0011 1010 :	59 0011 1011 ;	60 0011 1100 <	61 0011 1101 =	62 0011 1110 >	63 0011 1111 ?
64 0100 0000 @	65 0100 0001 A	66 0100 0010 B	67 0100 0011 C	68 0100 0100 D	69 0100 0101 E	70 0100 0110 F	71 0100 0111 G	72 0100 1000 H	73 0100 1001 I	74 0100 1010 J	75 0100 1011 K	76 0100 1100 L	77 0100 1101 M	78 0100 1110 N	79 0100 1111 O
80 0101 0000 P	81 0101 0001 Q	82 0101 0010 R	83 0101 0011 S	84 0101 0100 T	85 0101 0101 U	86 0101 0110 V	87 0101 0111 W	88 0101 1000 X	89 0101 1001 Y	90 0101 1010 Z	91 0101 1011 [	92 0101 1100 \	93 0101 1101]	94 0101 1110 ^	95 0101 1111 _
96 0110 0000 `	97 0110 0001 a	98 0110 0010 b	99 0110 0011 c	100 0110 0100 d	101 0110 0101 e	102 0110 0110 f	103 0110 0111 g	104 0110 1000 h	105 0110 1001 i	106 0110 1010 j	107 0110 1011 k	108 0110 1100 l	109 0110 1101 m	110 0110 1110 n	111 0110 1111 o
112 0111 0000 p	113 0111 0001 q	114 0111 0010 r	115 0111 0011 s	116 0111 0100 t	117 0111 0101 u	118 0111 0110 v	119 0111 0111 w	120 0111 1000 x	121 0111 1001 y	122 0111 1010 z	123 0111 1011 {	124 0111 1100 |	125 0111 1101 }	126 0111 1110 ~	127 0111 1111 ▨ DEL

Source: http://www.mathcs.emory.edu/~cheung/Courses/170/Syllabus/05/char1.html

Unicode

- Unicode is a generalization of ASCII using 16-bits.
- It allows up to 65536 distinct members.
- Unicode includes ISO 8859-1 as the first 256 members and then
  - Cyrillic characters
  - Both traditional and simplified Chinese,
  - Japanese and Thair characters,
  - Korean and many more languages
- Few email tools support Unicode today.

Rich Text Format (RTF)

- RTF is ASCII text with some additions to allow specification of formatting information:
  - Type face, font size and color.

- Sending email in RTF have a more professional appearance

- Two competing rich text standards
  - Microsoft's RTF (used in MS Word and Exchange)
  - MIME (Multipurpose Internet Mail Extension)

- NOTE: Good systems can keep track which system supports rich text format and automatically convert into simple ASCII for anyone else.

Multipurpose Internet Mail Extension (MIME)

- MIME is an Internet standard that extends the format of email to support:
  - **ASCII** characters
  - **Binary:** Non-text – audio, video, images, application programs etc.

- MIME was designed mainly for SMTP

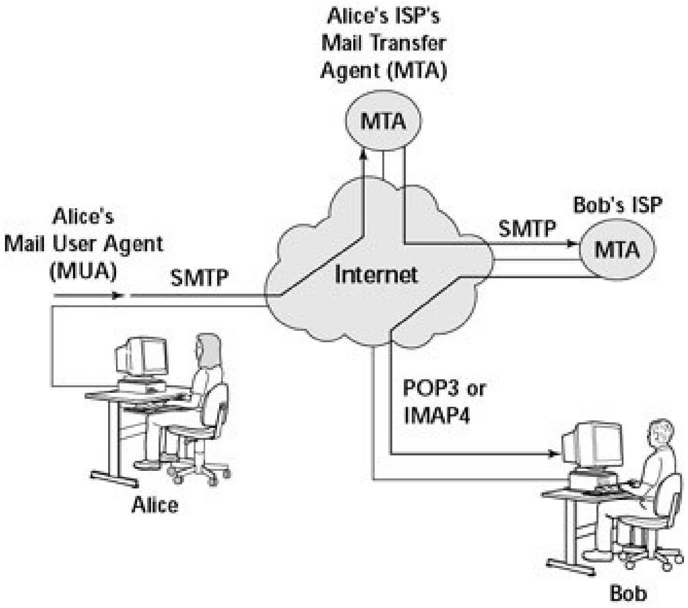
- The content types defined by MIME standards are also of importance in communication protocols outside of email, such as
  - HTTP for the World Wide Web.

Email Attachment

- An email attachment is a binary file
  - **Special Case:** It could be an ASCII
- It is not in the main body part of the message
- An attachment tags along with the main body part goes thorough the mail distribution system
- On the recipient's end, there is an indication that attachments have been received.

Message Transfer Agent (MTA)

- MTA is also know as an email server
- It is responsible for collecting mail from and distributing mail to user agents



- The exact method used for mail transport depends on the underlying network.
- In TCP/IP networks: the mail-transport agent delivers mail using SMTP.

Socket

- We can communicate with application which is located in different systems through communication method named Socket.

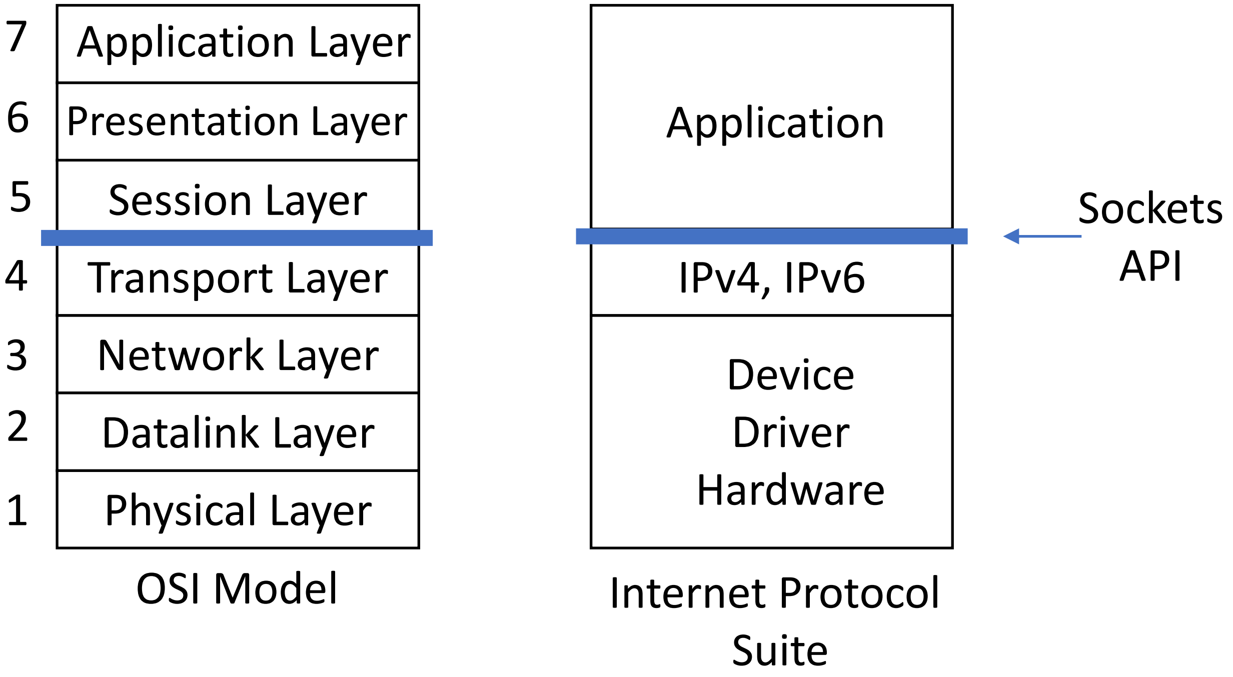
- Socket is the end point of two way communication link between two programs running on network.

- You can also communicate between two processes through socket on same machine but mostly it is used for communicating over network.

- Socket is supported by different operating systems like window, MAC, Linux, Unix etc.

Socket

- Socket programming interface in relation to the protocol stack



Types of Socket

- Three types of socket
  - Stream socket
  - Datagram socket
  - Raw socket

Types of Socket

- **Stream socket**
  - Reliable
  - Connection-oriented
  - TCP

- **Datagram socket**
  - Unreliable
  - Connectionless
  - UDP

- **RAW socket**
  - It provides RAW data transfer directly over IP Protocol (no transport layer)

Stream Socket

- When a computer is connected to the internet, it gets an IP address.
- This address uniquely identifies a computer on the Internet.
- In order to connect one to another, it must know the IP address of the host machine.
- In addition to the IP address, the process must know what “port” the other process is listening on.

Stream Socket

- **TCP Connection:** direct virtual pipe between client's socket and server's connection socket.

- A server is the process that is constantly running, listening for socket connections.

- The client process, attempts to make contact with the server, when needed.

TCP client/server Model Description

1. Client initiates contact with server.

2. Server must be ready
    - Not dormant (running).
    - Have a socket that welcomes initial contact from client process
        - Welcoming socket

3. Client initiates TCP connection to the server
    - Creates client socket (address of server process, port number).

TCP client/server Model Description

4. Three-way handshake
   - Client knocks server welcoming socket.
   - Server hears the knock and creates new socket to particular client (connection socket).
   - TCP connection exist between client's socket and server's new socket.

5. Client sends data via its client socket to server via connection socket and TCP guarantees the delivery of data
   - It means that TCP provides reliable byte-stream service between client and service processes.

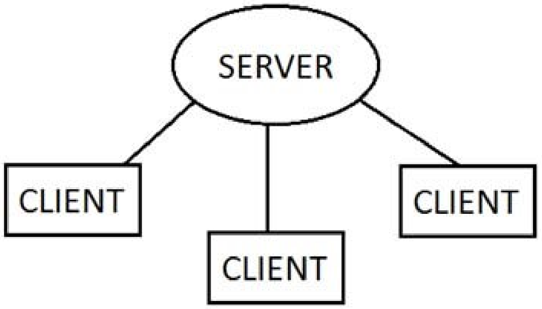
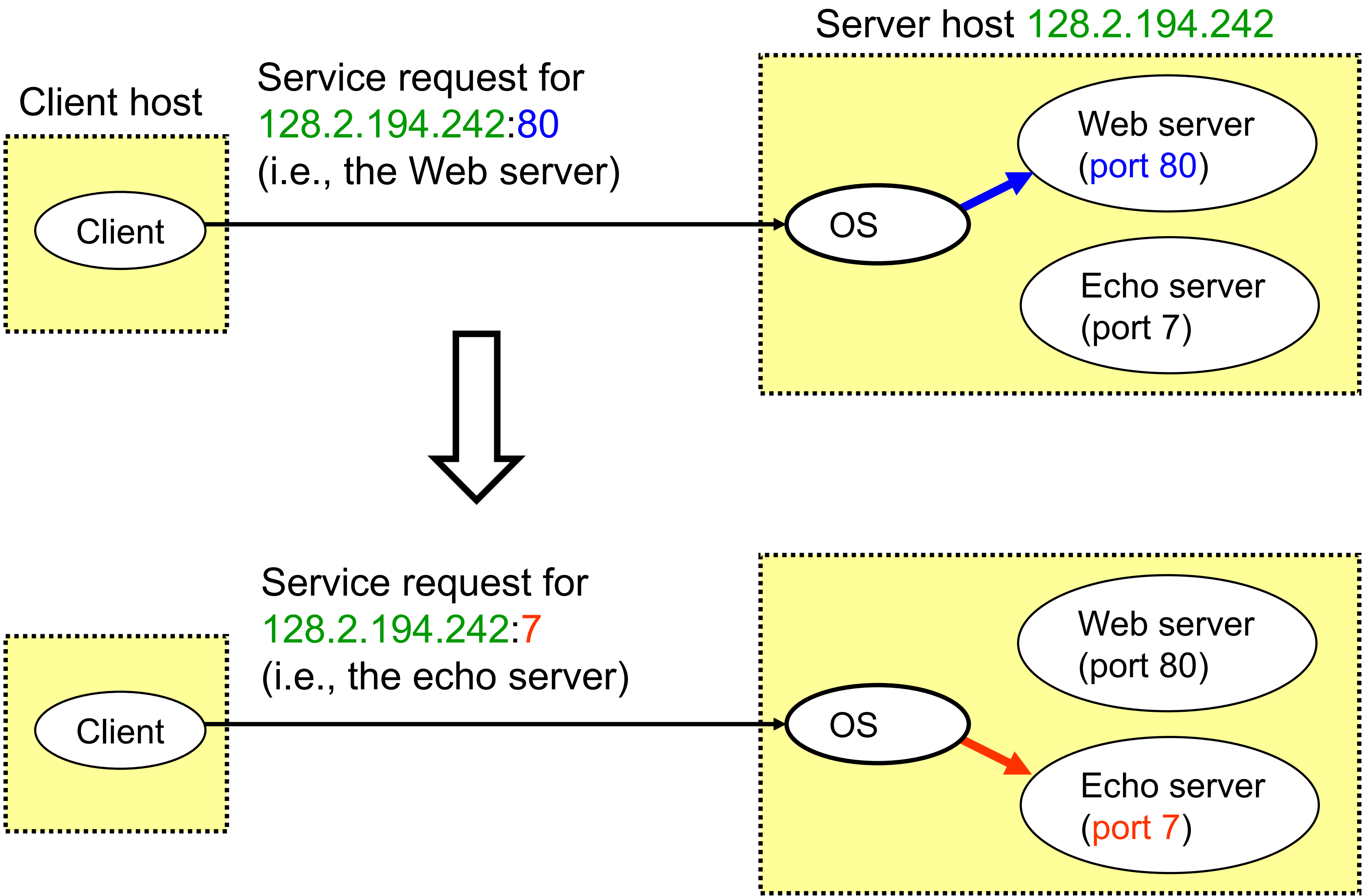


Image Source: https://www.engineersgarage.com/tutorials/socket-linux

An Analogy!!

- **Socket:** telephone
- **Bind:** assign telephone number to a telephone
- **Listen:** turn on the ringer so that you can hear the phone call
- **Connect:** dial a phone number
- **Accept:** answer the phone
- **Read/write:** talking
- **Close:** hang up

Using Ports to Identify Services



UNIX Socket API

- In UNIX, everything is like a file
  - All input is like reading a file
  - All output is like writing a file

- API implemented as system calls
  - For Example: connect, send, recv, close, …

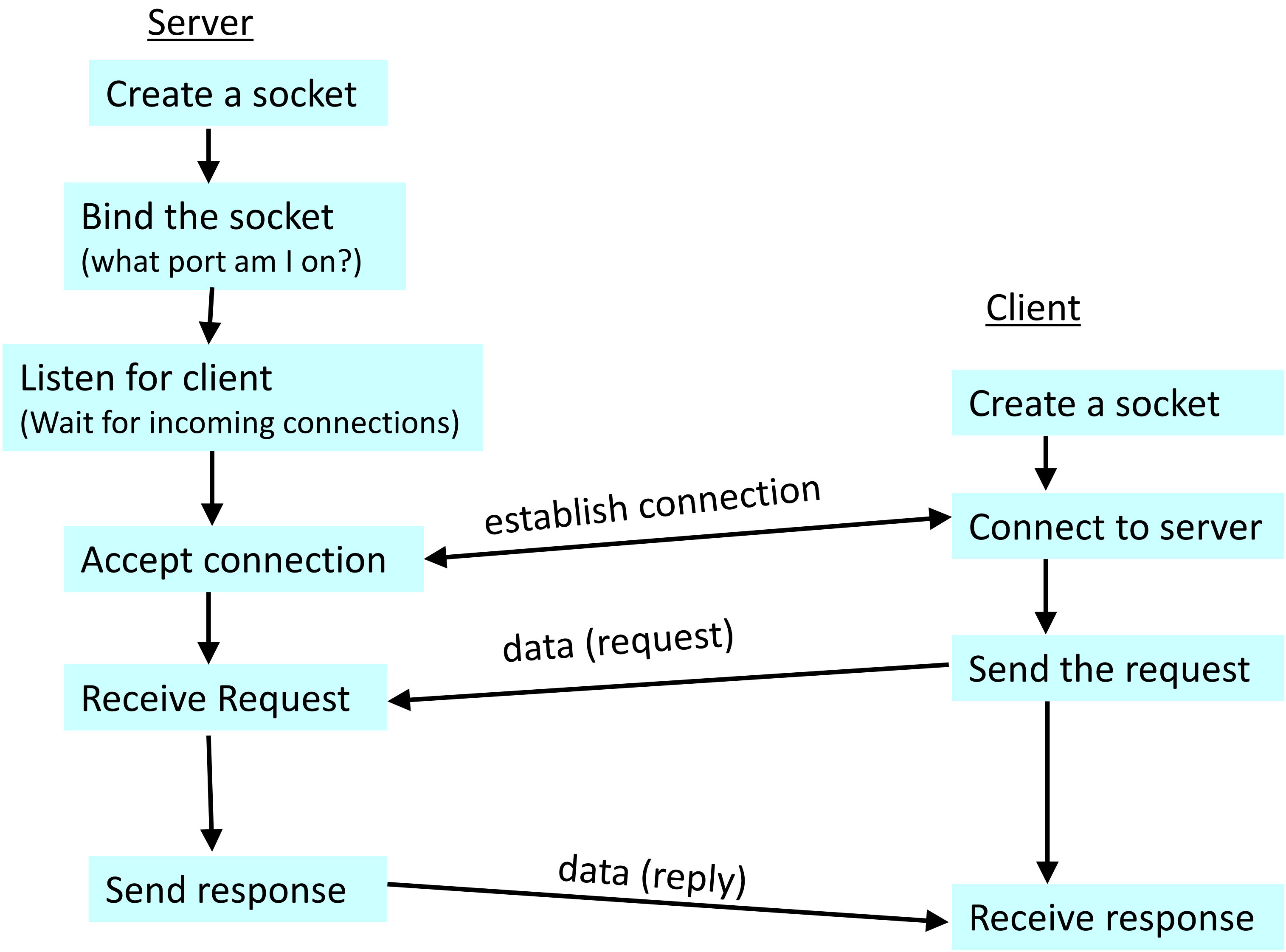
Steps involved in establishing a socket on the client

- Client side steps to create socket
  - Create a socket with the socket() system call
  - Connect the socket to the address of the server using the connect() system call
  - Send and receive data using send() and recv() system calls.

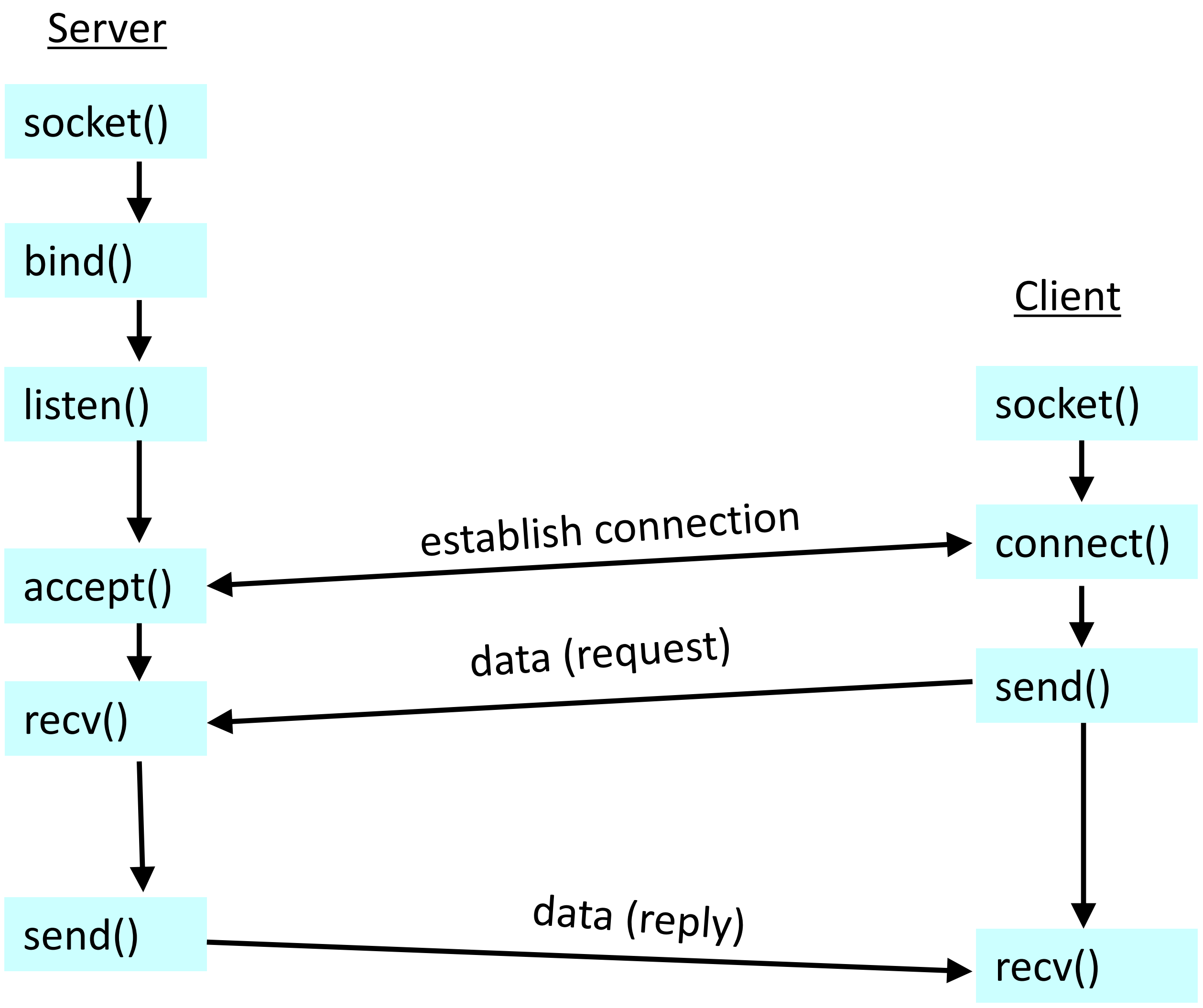
Steps involved in establishing a socket on the server

- Server side steps to create socket
  - Create a socket with the socket() system call
  - Bind the socket to an address using the bind() system call. For a server socket on the Internet, an address consists of a port number on the host machine.
  - Listen for connections with the listen() system call
  - Accept a connection with the accept() system call.
  - Send and receive data

Client-Server Communication
Stream Sockets (TCP): Connection-oriented



Connection-oriented Example (Stream Sockets -TCP)



Client: Learning Server Address/Port

- Server typically known by name and service
  - E.g., "www.cnn.com" and "http"

- Need to translate into IP address and port #
  - E.g., "64.236.16.20" and "80"

- Get address info with given host name and service

  -
    ```
    int getaddrinfo(    char *node,
                        char *service
                        struct addrinfo *hints,
                        struct addrinfo **result)
    ```

  - *node: host name (e.g., "www.cnn.com") or IP address
  - *service: port number or service listed in */etc/services* (e.g. ftp)
  - hints: points to a *struct addrinfo* with known information

Client: Learning Server Address/Port (cont.)

- Data structure to host address information

```
struct addrinfo {
    int     ai_flags;
    int     ai_family;     //e.g. AF_INET for IPv4
    int     ai_socketype;  //e.g. SOCK_STREAM for TCP
    int     ai_protocol;   //e.g. IPPROTO_TCP
    size_t  ai_addrlen;
    char    *ai_canonname;
    struct  sockaddr       *ai_addr; // point to sockaddr struct
    struct  addrinfo       *ai_next;
}
```

- Example

```
hints.ai_family = AF_UNSPEC;     // don't care IPv4 or IPv6
hints.ai_socktype = SOCK_STREAM; // TCP stream sockets
int status = getaddrinfo("www.cnn.com", ”80", &hints, &result);
// result now points to a linked list of 1 or more addrinfos
// etc.
```

Client: Creating a Socket

- Creating a socket
  - `int socket(int domain, int type, int protocol)`
  - Returns a file descriptor (or handle) for the socket

- Domain: Protocol family
  - PF_INET for IPv4
  - PF_INET6 for IPv6

- Type: Semantics of the communication
  - SOCK_STREAM: reliable byte stream (TCP)
  - SOCK_DGRAM: message-oriented service (UDP)

- Protocol: Specific protocol
  - UNSPEC: unspecified
  - (PF_INET and SOCK_STREAM already implies TCP)

- Example

```
sockfd = socket(result->ai_family, result->ai_socktype, result->ai_protocol);
```

Client: Connecting Socket to the Server

- Client contacts the server to establish connection
  - Associate the socket with the server address/port
  - Acquire a local port number (assigned by the OS)
  - Request connection to server, who hopefully accepts
  - connect is **blocking**

- Establishing the connection

```
– int connect(int sockfd,
                    struct sockaddr *server_address,
                    socketlen_t addrlen )
```

  - **Args:** socket descriptor, server address, and address size
  - Returns 0 on success, and -1 if an error occurs

- For Example:

```
connect(sockfd, result->ai_addr, result->ai_addrlen);
```

Client: Sending Data

- **Sending data**
  - `int send(int sockfd, void *msg, size_t len, int flags)`

- **Arguments**
  - Socket descriptor, pointer to buffer of data to send, and length of the buffer

- **Returns**
  - The number of bytes written, and -1 on error

- **Note**
  - Send is **blocking**: return only after data is sent
  - Write short messages into a buffer and send once

Client: Receiving Data

- **Receiving data**

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

- **Arguments**
  - Socket descriptor, pointer to buffer to place the data, size of the buffer, and flags

- **Returns**
  - The number of characters read (where 0 implies "end of file"), and -1 on error

Server: Server Preparing its Socket

- Server creates a socket and binds address/port
  - Server creates a socket, just like the client does
  - Server associates the socket with the port number

- Create a socket
  - `int socket(int domain, int type, int protocol)`

- Bind socket to the local address and port number

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

Server: Allowing Clients to Wait

- Many client requests may arrive
  - Server cannot handle them all at the same time
  - Server could reject the requests, or let them wait

- Define how many connections can be pending
  - `int listen(int sockfd, int backlog)`
  - Arguments: socket descriptor and acceptable backlog
  - Returns a 0 on success, and -1 on error
  - Listen is **non-blocking**: returns immediately

- What if too many clients arrive?
  - Some requests don't get through
  - The Internet makes no promises…
  - And the client can always try again

Server: Accepting Client Connection

- Now all the server can do is wait…
  - Waits for connection request to arrive
  - **Blocking** until the request arrives
  - And then accepting the new request

- Accept a new connection from a client
  - `int accept(int sockfd, struct sockaddr *addr, socketlen_t *addrlen)`
  - **Arguments:**
    - sockfd, structure that will provide client address and port, and length of the structure
  - **Returns**
    - Descriptor of socket for this new connection

Client and Server: Cleaning House

- Once the connection is open
  - Both sides and read and write
  - Two unidirectional streams of data
  - In practice, client writes first, and server reads
  - … then server writes, and client reads, and so on

- Closing down the connection
  - Either side can close the connection
  - … using the `int close(int sockfd)`

- What about the data still “in flight”
  - Data in flight still reaches the other end
  - So, server can `close()` before client finishes reading

Server: One Request at a Time?

- Serializing requests is inefficient
  - Server can process just one request at a time
  - All other clients must wait until previous one is done

- May need to time share the server machine
  - Alternate between servicing different requests
    - Do a little work on one request, then switch when you are waiting for some other resource
    - “Nonblocking I/O”
  - Or, use a different process/thread for each request
    - Allow OS to share the CPU(s) across processes
  - Or, some hybrid of these two approaches

Handle Multiple Clients using fork()

- Steps to handle multiple clients
  - Go to a loop and accept connections using accept()
  - After a connection is established, call fork()
    - To create a new child process to handle it
  - Go back to listen for another socket in the parent process
  - close() when you are done.

Real Clients and Servers?

- Apache Web server
  - Open source server first released in 1995
  - Software available online at http://www.apache.org
- Mozilla Web browser
  - http://www.mozilla.org/developer/
- Sendmail
  - http://www.sendmail.org/
- BIND Domain Name System
  - Client resolver and DNS server
  - http://www.isc.org/index.pl?/sw/bind/
- …

Acknowledgements

- This tutorial was prepared by M.Fahim, G.Succi, and A.Tormasov

Reference

- This tutorial is based on the following documents as well as lecture materials over the internet.
  - http://users.cs.cf.ac.uk/Dave.Marshall/Internet/node86.html
  - http://alumni.cs.ucr.edu/~ecegelal/TAw/socketTCP.pdf