Operating Systems

Lecture 4.4

Socket Programming

Instructor: Muhammad Arif Butt, PhD

Lecture Agenda

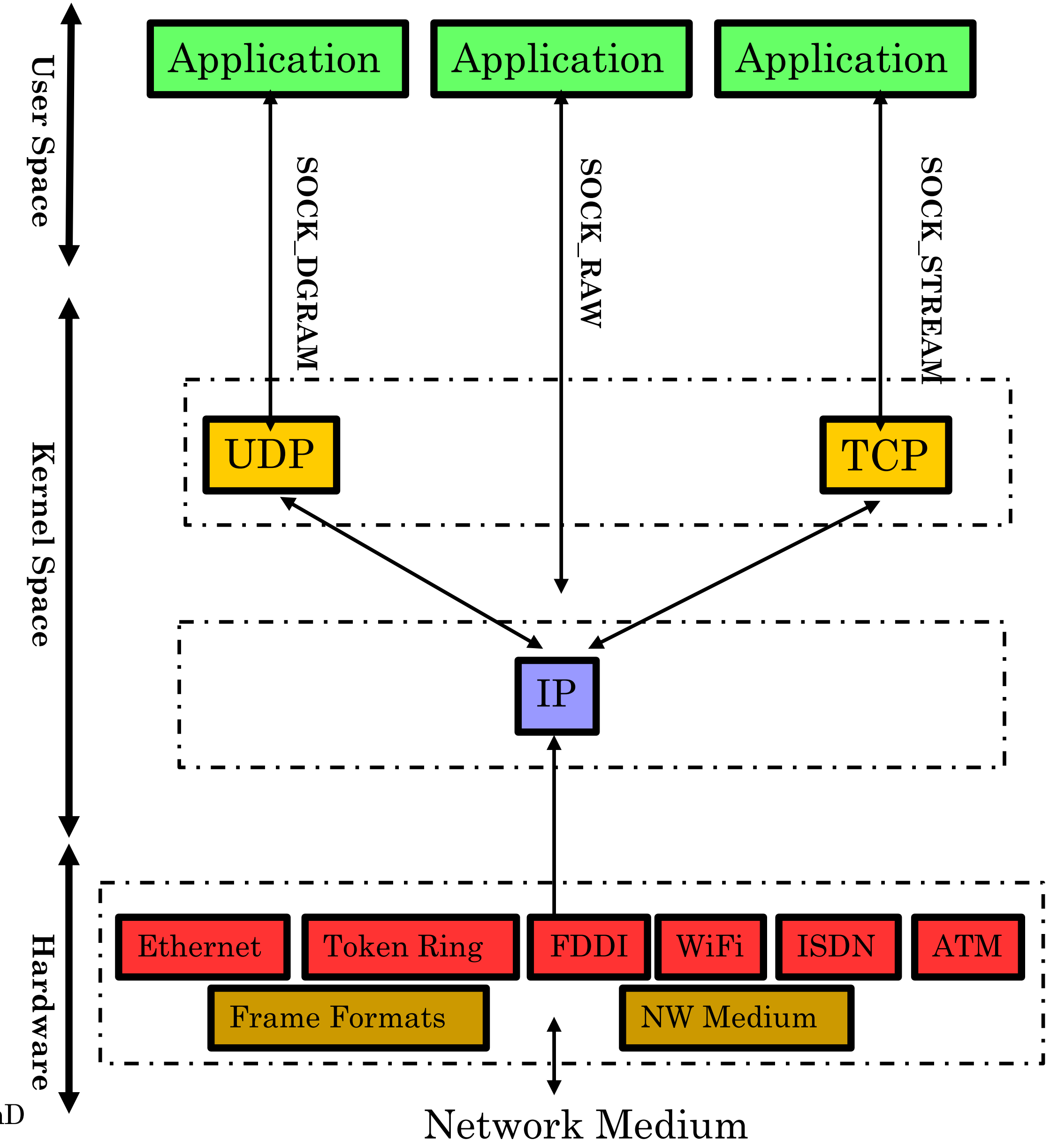
- TCP/IP Stack
- Addressing Schemes Used On TCP/IP Layers
- Client Server Paradigm
- How Stream Sockets Work?
- POSIX Socket API For TCP Client
- POSIX Socket API For TCP Server

TCP/IP Stack

TCP/IP Stack

- *Internet* or *Internetwork* is a network of networks that connects different computer networks, allowing hosts on all the networks to communicate with one another.
- Although various internetworking protocols have been devised, but TCP/IP has become the dominant protocol suite developed by DARPA (Defense Advanced Research Projects Agency).
- TCP/IP is a layered architecture with different protocols working on different layers.
- A networking protocol is a set of rules defining how information is to be transmitted across a network, specifying:
- How the data to be exchanged is encoded?
- How the sending and receiving events are coordinated among the participants?

Protocols in TCP/IP Suite



Protocols in TCP/IP Suite

Application Layer

- Consist of processes that uses the NW
- Provides programming interface used for building a program
- Protocols used are http, telnet, ftp, smtp, ssh
- Addresses are string based URIs (URL, URN)

Transport Layer

- Provides host-to host communication
- Protocols used are TCP, UDP, RAW
- 16 bits Port numbers are used for addressing

Internet Layer

- Break data into fragments small enough for transmission via link layer
- Routing data across internet
- Protocols used are IP, ARP, ICMP, IGMP
- IPv4 and IPv6 are used for addressing

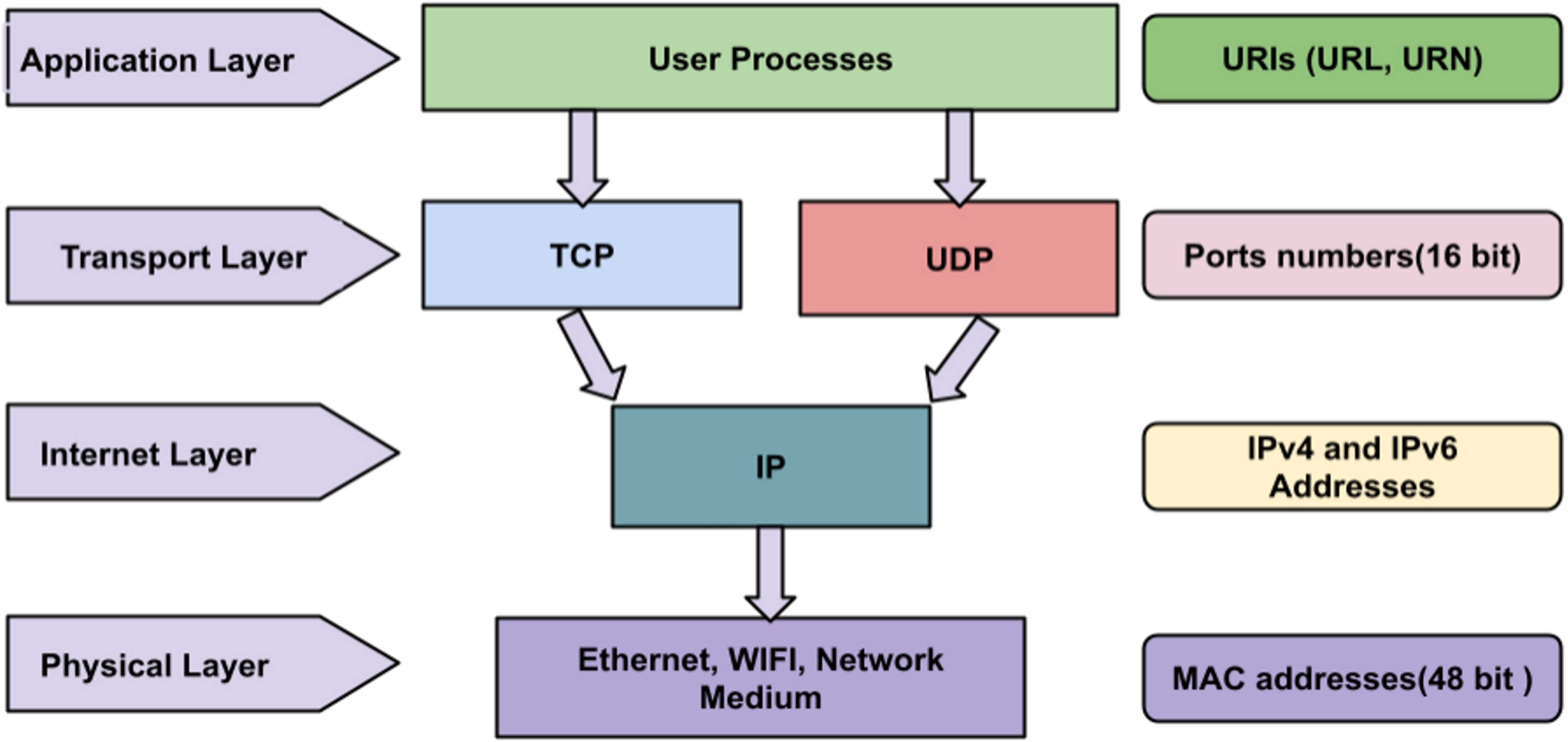
Link Layer/ Physical

- Place packets on the NW medium and receiving packets off the NW medium
- NW access methods used are Ethernet, Token ring, FDDI, ISDN, SONET, ATM
- 48 bit Mac address are used for addressing

Addressing Schemes Used On TCP/IP Layers

Instructor: Muhammad Arif Butt, PhD

Addressing Schemes Used On TCP/IP Layers



Addressing on the Application Layer

- The Internet Assigned Numbers Authority (IANA) oversees the assignment of domain names to organizations. These domain names can have multiple strings separated by periods. Each host on the Internet is uniquely identified by a Fully Qualified Domain Name (FQDN), which consists of two parts:

  `hostname.domain-name`

- These FQDNs are stored in a hierarchical and decentralized database that maps hostnames to their corresponding IP addresses. The service that performs the lookup is called Domain Name System (DNS) or Berkley Internet Name Domain (BIND) specified in RFC 882 and RFC 883.
- **URL (Uniform Resource Locator):** A URL identifies a resource located on a specific host within a domain. Its format is:

  `protocol://hostname.domain-name[: port]/path-to-resource`

  For Example: http://pucit.pu.edu.pk:80/academics/timetable-pucit.html

Organizations can add prefixes to their domain names to define hosts. For example, in the above example pu.edu.pk is the domain-name, while pucit is the suffix to define its subdomain.

Addressing on the Transport Layer (…)

The transport layer addresses are called Port Numbers. A 16 bit integer used to identify a specific process to which a NW message is to be forwarded when it arrives at a host. There may be a machine which is running both the `http` and `ssh` service. The `http` process will be listening on port 80, while `ssh` process will be listening on port 22

- **Well Known / Reserved Ports (0 to 1023):** These are permanently assigned to specific applications (also known as services). For example, ssh daemon uses port 22. Well known ports are assigned numbers by a central authority the Internet Assigned Number Authority (http://www.iana.org)
- **Registered Ports (1024 to 49151):** IANA also records registered ports, which are allocated to application developers on a less stringent basis
- **Dynamic/Private/Ephemeral Ports (49152 to 65535):** IANA specifies the ports in the range 49152 to 65535 as dynamic or private, with the intention that these ports can be used by local applications. If an application doesn't select a particular port (i.e., it doesn't bind() its socket to a particular port), then TCP and UDP assign a unique ephemeral port (i.e., short-lived) number to the socket
- **View `/etc/services` file on your UNIX machine for details**

Addressing on Transport Layer

Protocol	Port	Service
echo	7	IPC testing
daytime	13	Provides current date and time
ftp-data, ftp	20, 21	File Transfer Control (TCP)
ssh	22	Secure Shell for secure Remote Login facility (TCP)
telnet	23	Remote login facility (TCP)
smtp	25	Simple Mail Transfer Protocol (TCP)
time	37	Provides standard time
bootps, bootpc	67, 68	Bootp server and client (UDP)
tftp	69	Trivial File Transfer Protocol (UDP)
finger	79	Provides information about a user
http	80, 8080	Web Server (TCP)
sunrpc	111	Sun Remote Procedure Call
NTP	123	Network Time Protocol (UDP)
https	443	Secure Web Server (TCP)
RMI Registry	1099	Registry for Remote Method Invocation
NFS	2049	Network File Server (UDP)

Classful Addressing on the Internet Layer (IPv4)

Class A IP Addresses

- **Total Addresses:** *$2^7-2 = 126$ networks*
- **Range:** 1.0.0.0 to 126.0.0.0 — **0.x.x.x** and **127.x.x.x** are reserved, and is the reason for subtracting 2 from 2^7
- **Hosts per Network:** *$2^{24}-2 = 16777214$ hosts*
- **Subnet Mask:** 255.0.0.0 or /8

0	Net ID (7)	Host ID (24)

Class C IP Addresses

- **Total Addresses:** *$2^{21} = 2097152$ networks*
- **Range:** 192.0.0.0 to 223.255.255.0
- **Hosts per Network:** *$2^8-2 = 254$ hosts*
- **Subnet Mask:** 255.255.255.0 or /24

110	Net ID (21)	Host ID (8)

Class B IP Addresses

- **Total Addresses:** *$2^{14} = 16384$ networks*
- **Range:** 128.0.0.0 to 191.255.0.0
- **Hosts per Network:** *$2^{16}-2 = 65534$ hosts*
- **Subnet Mask:** 255.255.0.0 or /16

10	Net ID (14)	Host ID (16)

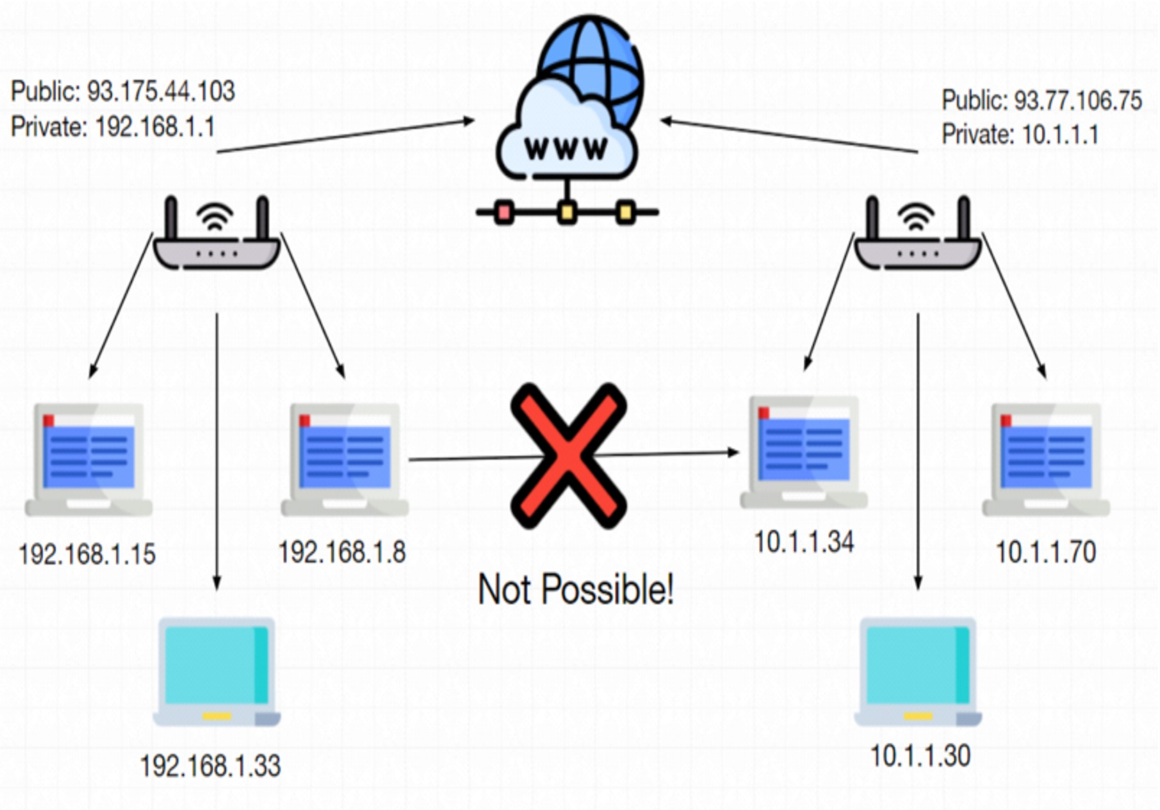
Every valid IP Address of a class lie between the Network Address and the Broadcast Address of that class.

Classful Addressing on the Internet Layer (IPv4)

- There exist ***class D*** and ***Class E*** addresses as well. Class D addresses (224.0.0.0 to 239.255.255.255) are used for multi-cast communication, while Class E addresses (225.0.0.0 to 255.255.255.255) are not assigned for public use rather reserved by the IETF for future use.
- The ***network address*** for a specific class is represented with all bits as ZERO in the host portion of the address.
- The ***broadcast address*** for a specific class is represented with all bits as ONES in the host portion of the address.
- The ***subnet mask address*** for a specific class is represented with all bits as ONES in the network portion and with all bits as ZERO in the host portion. To get the network address you just bit-wise AND the IP address with the subnet mask. All routing is performed based on the NW address.
- **Classless Internet Domain Routing (CIDR):** In 1993, CIDR was introduced that revolutionized IP address allocation and routing by eliminating the rigid boundaries of classful addressing. It offers the advantages of efficient allocation of IP addresses and flexible sub-netting. This helped to meet the growing demand of Internet and the limited address space of IPv4 (4 billion). In CIDR the address 192.168.10.0/25 means the first 25 bits of the IP address are used for the NW portion.

Public IP Addresses (IPv4)

- **Public IP Addresses** as mentioned on the previous page are unique across the entire Internet and are used for communication over the Internet, making them accessible from any device globally.
- Public IP addresses are routable on the internet and are assigned to devices that need to be reachable from outside the local network, such as web servers, email servers, and network gateways.
- Devices having public IP addresses are exposed to potential security risks as they are accessible from the Internet.



Private IP Addresses (IPv4)

Private IP Addresses: IETF has designed three address ranges (one for each class) as private, which are commonly used for devices within a local area network, such as computers, laptops, printers and smartphones:

- 10.0.0.0 to 10.255.255.255
- 172.16.0.0 to 172.31.255.255
- 192.168.0.0 to 192.168.255.255

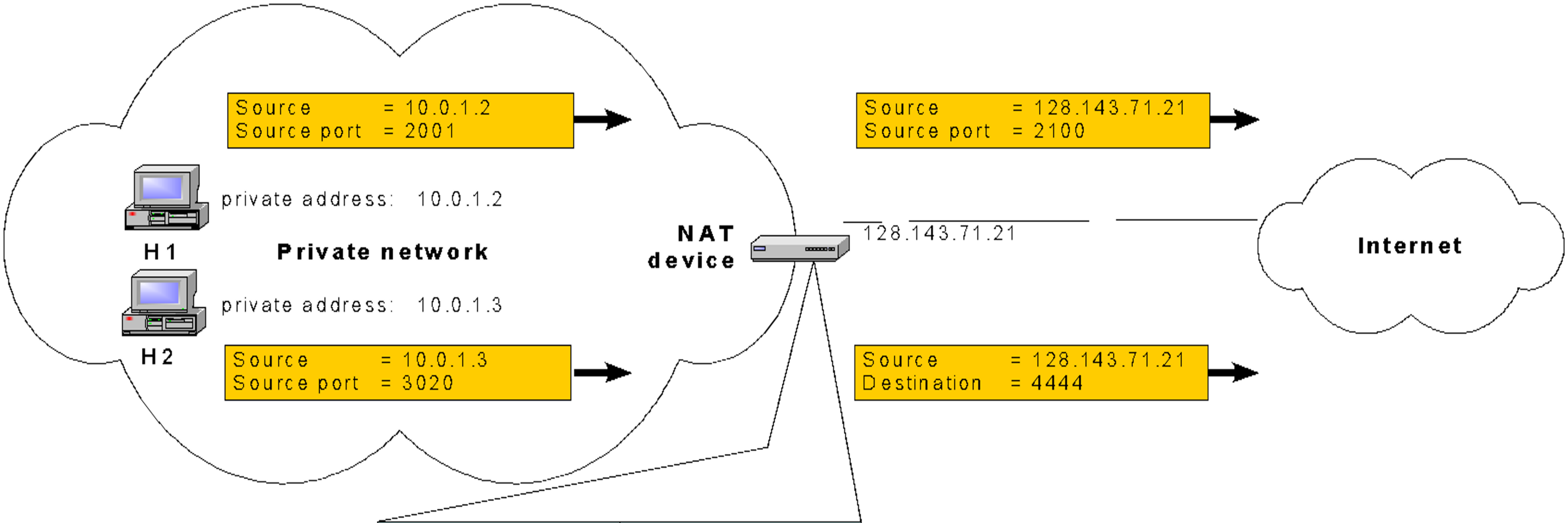
The devices having private IP addresses are non-routable, i.e., not directly exposed to the public Internet, providing a layer of security by keeping internal devices hidden from external threats. They can only be used either on a fully disconnected NW or on a NW behind firewall.

100$ Question:

How can a device having a private IP address accesses the resources on the Internet having public IP addresses?

NetWork Address Translation (NATing), that allows a single device called gateway computer (router) having a public IP address to act as an agent between the Internet and the private NW. A gateway computer is an entry/exit point in a LAN, that receives incoming requests from devices having private IP addresses and send it to the Internet with its own public IP address. So, this means that a single public IP address can represent an entire group of computers on the Internet.

Private IP Addresses (IPv4) (Cont.)



Private Address	Public Address
10.0.1.2/2001	128.143.71.21/2100
10.0.1.3/3020	128.143.71.21/4444

“CIDR and NATing has significantly extended the useful life of IPv4”

Addressing on the Physical Layer

- **MAC Address Format:**
  - A 48-bit address is used on the physical layer.
  - Divided into two parts:
    - **Organizationally Unique Identifier (OUI):** The most significant 3 bytes (e.g., **00-50-56**).
    - **Network Interface Specific Identifier:** The least significant 3 bytes (e.g., **C0-00-01**).

- **MAC Address Assignment:**
  - Manufacturers request an OUI from the IEEE to ensure a unique prefix for their devices.
  - The manufacturer then assigns a unique identifier to the remaining 3 bytes for each device.
  - This ensures a globally unique MAC address for every device.

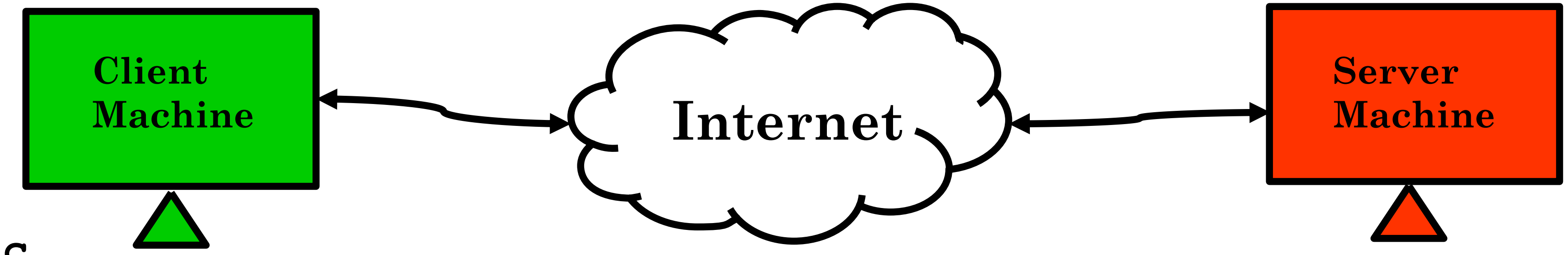
- **Routing and Address Resolution:**
  - If the destination IP address is outside the local network, the packet is sent to a configured gateway for routing.
  - If the destination IP address is within the same local network, the Address Resolution Protocol (ARP) is used to find the corresponding MAC address from the IP address.

Organizationally Unique Identifier 00-50-56	Network Interface Specific Identifier C0-00-01

Client Server Paradigm

Instructor: Muhammad Arif Butt, PhD

Client Server Paradigm



nc
telnet
ssh
Web Browsers

telnet	(23)
FTP	(20/21)
SSH	(22)
DHCP	(546/547)
NFS	(2049)

Stateful
Long Connections

echo	(7)
discard	(9)
daytime	(13)
chargen	(19)
time	(37)
DNS	(53)
HTTP	(80/8080)
HTTPS	(443)
NTP	(123)

Stateless
Short Connections

Server Design Based on Connection Mode & Execution Strategy:

- Iterative connectionless
- **Iterative connection-oriented**
- Concurrent connectionless
- Concurrent connection oriented

What is a Socket?

A socket is a communication end point to which an application can write data (to be sent to the underlying network) and from which an application can read data. The process/application can be related or unrelated and may be executing on the same or different machines

- From IPC point of view, a socket is a full-duplex IPC channel that may be used for communication between related or unrelated processes executing on the same machine or across networked systems using TCP/IP or other protocols.

- Available APIs for socket communication are:
  - Berkley/POSIX sockets (Linux, UNIX, macOS)
  - Winsock for MS Windows

Types of Sockets

Internet Sockets:

- **Stream Sockets (SOCK_STREAM):** Provide reliable, connection-oriented communication over TCP. Ideal for applications that require guaranteed data delivery, such as web servers or SSH.
- **Datagram Sockets (SOCK_DGRAM):** Offer connectionless, unreliable communication using UDP. Suitable for fast, low-overhead communication like DNS, VoIP, or real-time video streaming.

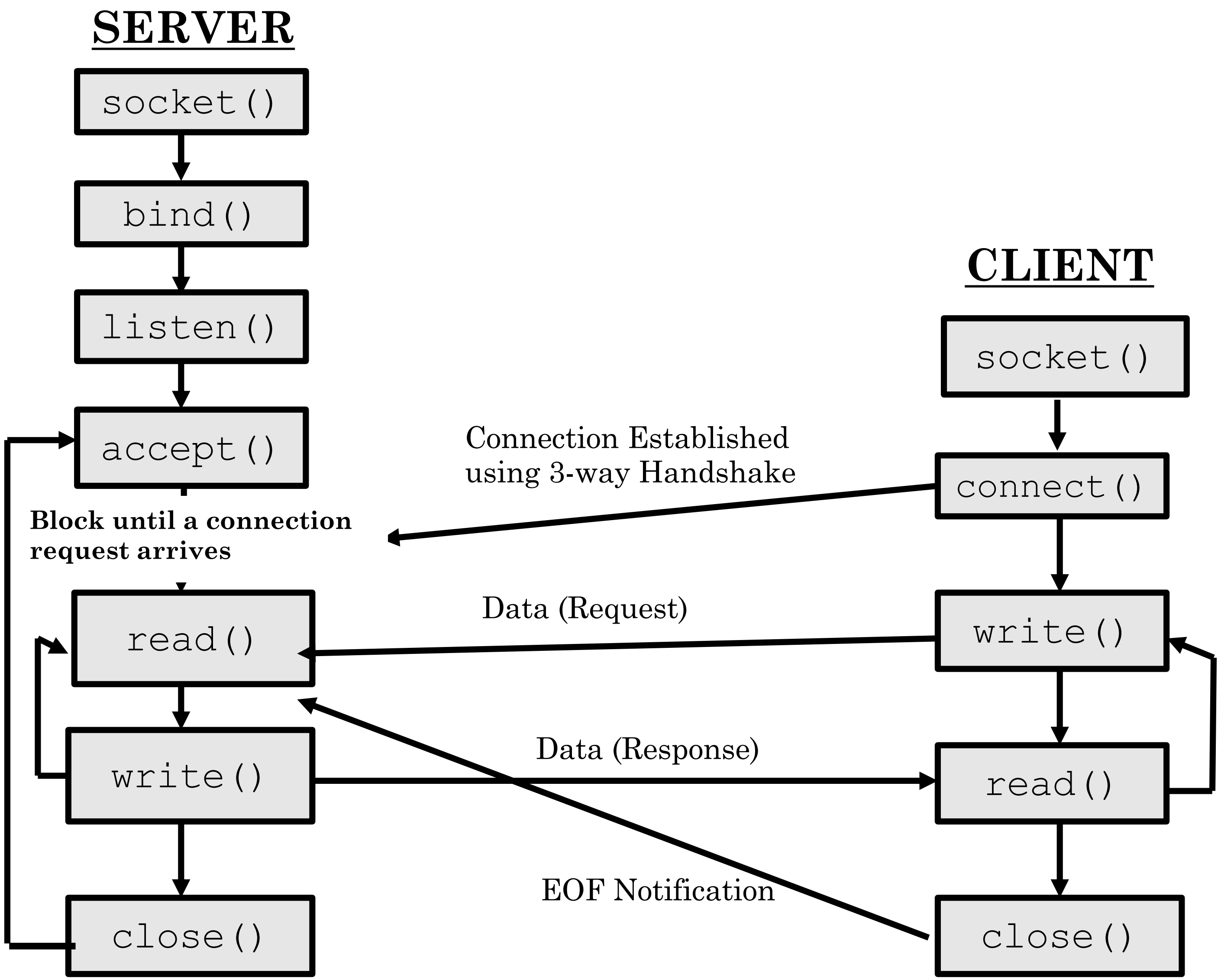
UNIX Domain Sockets:

- Provide inter-process communication (IPC) on the same host using the file system as an address namespace. Faster and more secure than internet sockets for local communication between processes
  - Many system services (systemd) use UDS to interact with client applications.
  - Nginx/Apache often use UDS to communicate with backend servers like PHP-FPM, or Node.js.
  - PostgreSQL, MySQL, Redis, and MongoDB support client connections via UDS.
  - Apps running in sandboxes (like Chromium) may use UDS to comm with trusted host services.

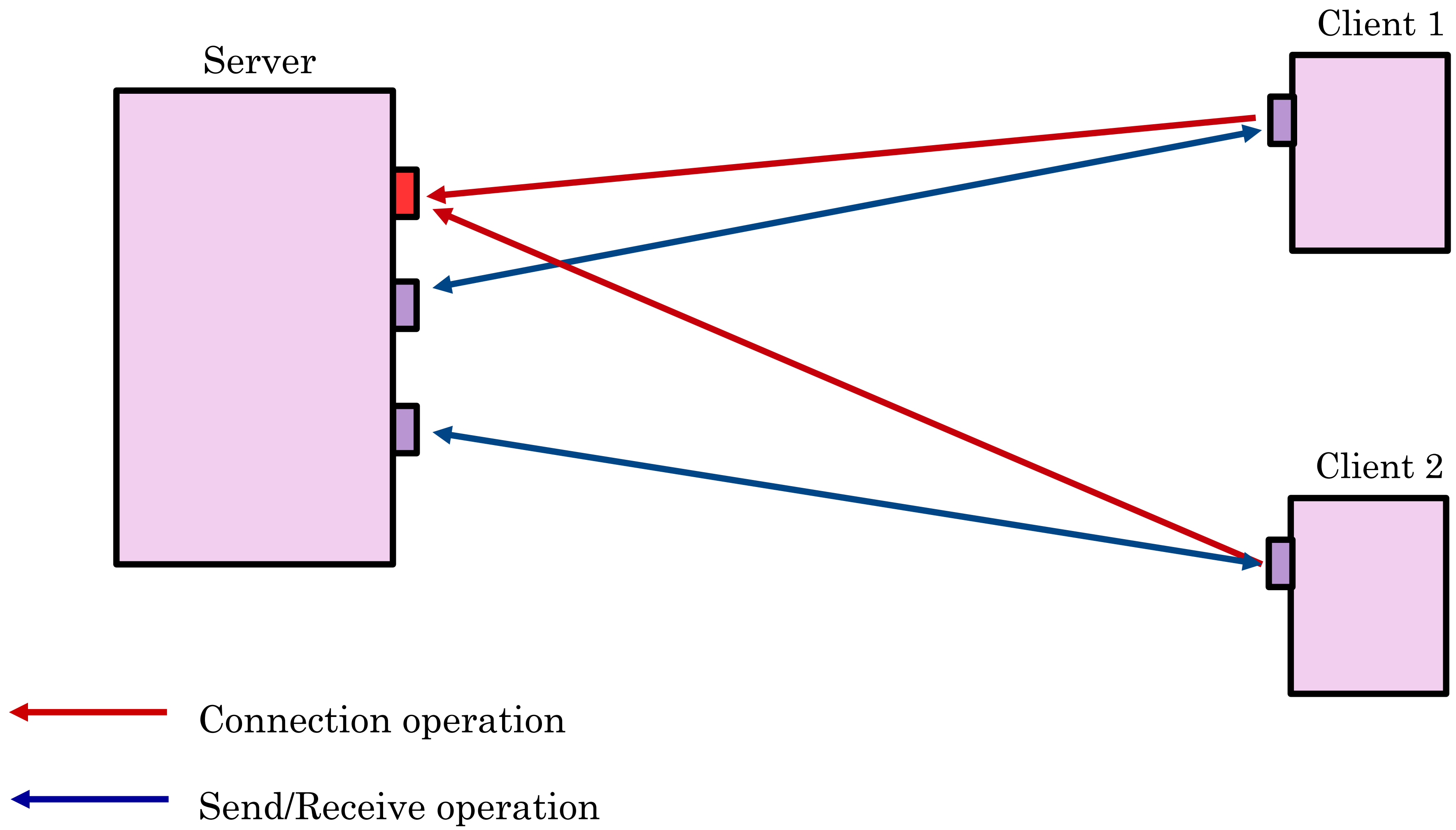
How Stream Sockets Work?
Behind the curtain

Instructor: Muhammad Arif Butt, PhD

System Call Graph: TCP Sockets



Pictorial Representation of TCP Socket



POSIX Socket API For TCP Client

Instructor: Muhammad Arif Butt, PhD

Pseudocode: TCP Sockets

SERVER

```
socket()
bind()
listen()
while(1) {
        accept()
        while(client writes) {
            Read a request
            Perform requested action
            Send a reply
        }
        close client socket
    }
close passive socket
```

CLIENT

```
socket()
connect()
while(x) {
        write()
        read()
}
close()
```

socket()

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

- **`socket()`** creates an endpoint for communication
- On success, a file descriptor for the new socket is returned
- On failure, -1 is returned and `errno` is set appropriately
- The first argument `domain` specifies a communication domain under which the communication between a pair of sockets will take place. Communication may only take place between a pair of sockets of the same type
- These families are defined in **`/usr/include/x86.../bits/socket.h`**

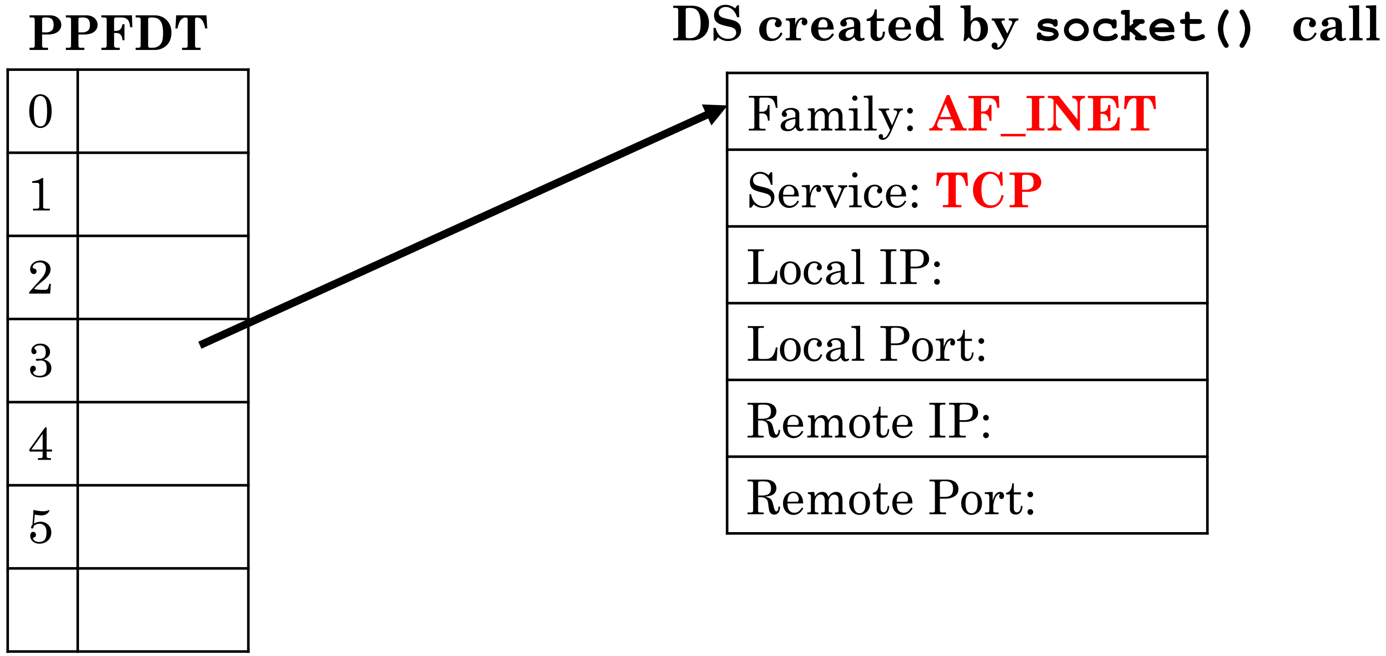
Domain	Comm Performed	Comm between applications	Address format	Address structure
AF_UNIX	Within kernel	On same host	pathname	sockaddr_un
AF_INET	Via IPv4	On hosts connected via an IPv4 network	32-bt IPv4 addr + 16-bit port#	sockaddr_in
AF_INET6	Via IPv6	On hosts connected via IPv6 network	128-bit IPv6 addr + 16-bit port#	sockaddr_in6

socket() (...)

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

- The second argument `type` specifies the communication semantics. These types are defined in the header file `/usr/include/x86.../bits/socket_type.h`. Most common types are SOCK_STREAM and SOCK_DGRAM
- The 3rd argument specifies the protocol to be used within the network code inside the kernel, not the protocol between the client and server. Just set this argument to "0" to have `socket()` choose the correct protocol based on the `type`. You may use constants, like IPPROTO_TCP, IPPROTO_UDP. You may use `getprotobyname()` function to get the official protocol name (discussed later).You may look at `/etc/protocols` file for details
- To view more details about these constants visit following man pages:
  - **`$man 7 tcp, udp, raw, unix, ip, socket`**
  - **`$man 5 protocols`**

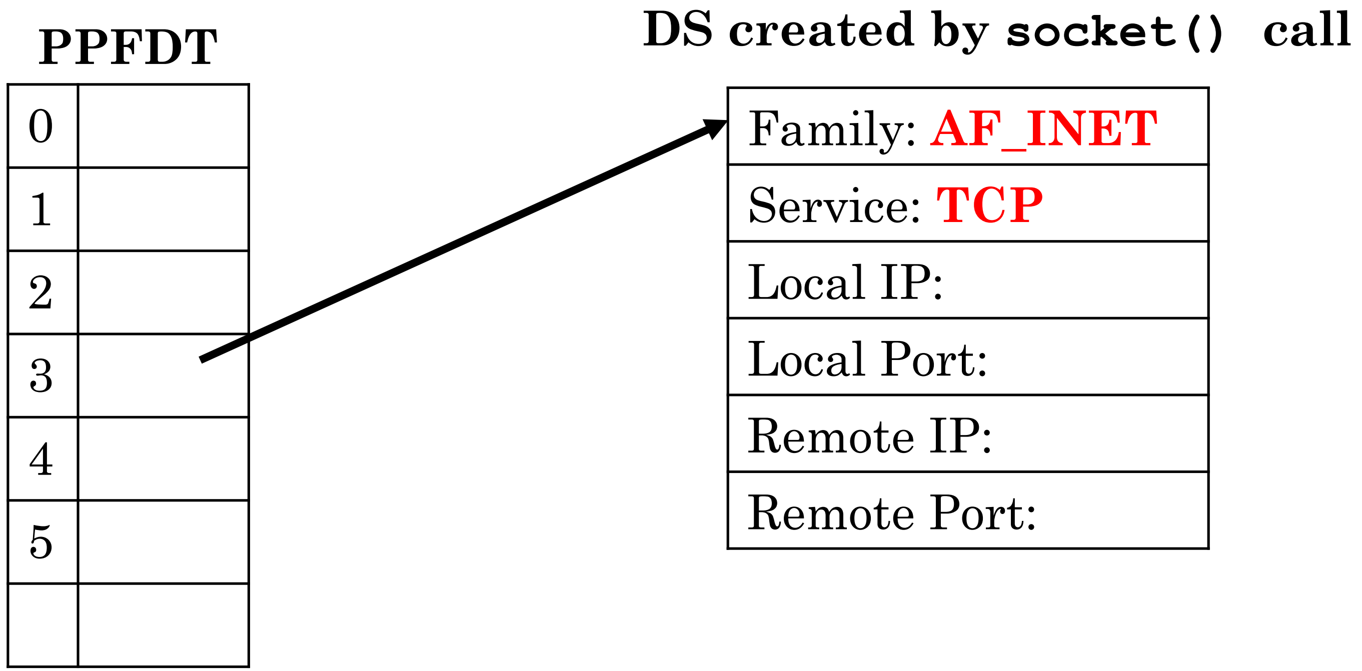
socket() (...)




```
int sockfd = socket(AF_INET, SOCK_STREAM, 0);
```

- The socket data structure contains several pieces of information for the expected style of IPC, including family/domain, service type, local IP, local port, remote IP, and remote port
- UNIX kernel initializes the first two fields when a socket is created
- When the local address is stored in socket data structure we say that the socket is **half associated**
- When both local and remote addresses are stored in socket data structure, we say that socket is **fully associated**

socket() (...)




```
int sockfd = socket(AF_INET, SOCK_STREAM, 0);
```

<u>How addresses in socket data structure are populated</u>

For Client

- Remote endpoint address is populated by `connect()`
- Local endpoint address is automatically populated by TCP/IP s/w when client calls `connect()`

For Server

- Local endpoint addresses are populated by `bind()`
- Remote endpoint addresses are populated by `accept()`

connect()

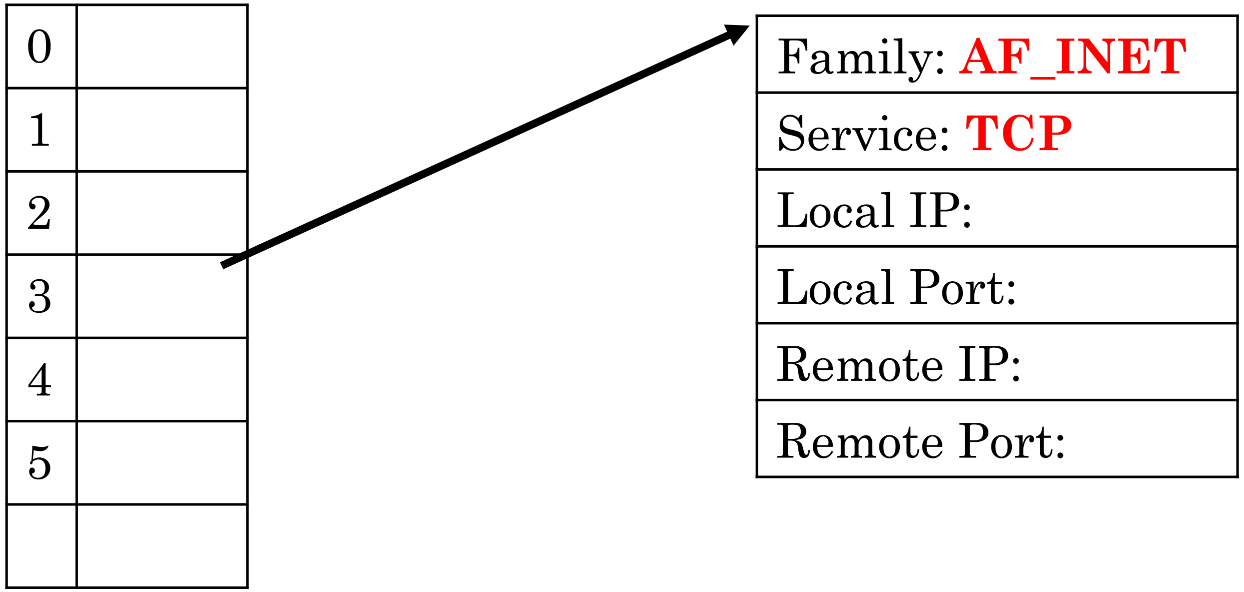
```
int connect(int sockfd, const struct sockaddr *svr_addr,int addrlen);
```

- The `connect()` system call connects the socket referred to by the descriptor `sockfd` to the remote server (specified by `svr_addr`)
- If we haven't call `bind()`, (which we normally don't in client), it automatically chooses a local endpoint address for you
- On success, zero is returned, and the `sockfd` is now a valid file descriptor open for reading and writing. Data written into this file descriptor is sent to the socket at the other end of the connection, and data written into the other end may be read from this file descriptor
- TCP sockets may successfully connect only once. UDP sockets normally do not use connect(), however, connected UDP sockets may use `connect()` multiple times to change their association
- When used with `SOCK_DGRAM` type of socket, the `connect()` call simply stores the address of the remote socket in the local socket's data structure, and it may communicate with the other side using `read()` and `write()` instead of using `recvfrom()` and `sendto()` calls

`connect()` (Cont.)

`connect()` performs four tasks

1. Ensure that the specified `sockfd` is valid and that it has not already been connected
2. Fills in the remote end point address in the (client) socket from the second argument
3. Automatically chooses a local endpoint address by calling TCP/IP software
4. Initiates a TCP connection (3 way handshake) and returns a value to tell the caller whether the connection succeeded



Internet Socket Address Structure

Generic Socket Address structure: This is a basic template on which other address data structures of different domains are based. When `sa_family` is AF_UNIX the `sa_data` field is supposed to contain a pathname as the socket's address. When `sa_family` is AF_INET the `sa_data` field contains both an IP address and a port number

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

Internet Socket Address Structure:

```
struct sockaddr_in{
   u_short        sin_family;
   u_short        sin_port;
   struct in_addr sin_addr;
   char           sin_zero[8];
}
```

```
struct in_addr{
   in_addr_t s_addr;
}
```

UNIX Domain Socket Address Structure:

```
struct sockaddr_un{
   short  sun_family;
   char   sun_path; }
```

Populating Address Structure

- **Example:** We normally need to populate the address structure and then pass it to `connect()`. Following is the code snippet that do the task:

```
struct sockaddr_in svr_addr;
svr_addr.sin_family = AF_INET;
svr_addr.sin_port = htons(54154);
inet_aton("127.0.0.1", &svr_addr.sin_addr);
memset(&(svr_addr.sin_zero), '\0', sizeof(svr_addr.sin_zero))
connect(sockfd,(struct sockaddr*)&svr_addr,sizeof(svr_addr));
```

- **Question:** Why we need to cast the `sockaddr_in` to generic socket address structure `sockaddr`?
- **Answer:** Address structures (of all families) need to be passed to `bind()`, `connect()`, `accept()`, `sendto()`, `recvfrom()`. In 1982, there was no concept of `void*`, so the designers defined a generic socket address structure

Little Endian vs Big Endian

- Byte order is the attribute of a processor that indicates whether integers are represented from left to right or right to left in the memory
- In **Little Endian Byte Order**, the low-order byte of the number is stored in memory at the *lowest address* and the high-order byte of the same number is stored at the highest address
- In **Big Endian Byte Order**, the low-order byte of the number is stored in memory at the *highest address* and the high-order byte of the same number is stored at the lowest address

```
short int var = 0x0001;
char *byte = (char*)&var;
if (byte[0] == 1)
   printf("Little Endian");
else
   printf("Big Endian");
```

00000001	00000000	00000000	00000000
00000000	00000000	00000000	00000001
0x2000	**0x2001**	**0x2002**	**0x2003**

Byte Order and Byte Ordering Functions

```
uint16_t htons(uint16_t host16bitvalue);
uint16_t htonl(uint32_t host32bitvalue);
```

Returns: value of arg passed is converted to NBO

```
uint16_t ntohs(uint16_t net16bitvalue);
uint16_t htons(uint32_t net32bitvalue);
```

Returns: value of arg passed is converted to HBO

- The API `htons()` is used to convert a 16-bits data from *host byte order to network byte order* such as TCP or UDP port number
- The API `htonl()` is used to convert a 32-bits data from *host byte order to network byte order* such as IPv4 address
- The API `ntohs()` is used to convert a 16-bits data from *network byte order to host byte order* such as TCP or UDP port number
- The API `ntohl()` is used to convert a 32-bits data from *network byte order to host byte order* such as IPv4 address

Address Format Conversion Functions

```
in_addr_t inet_addr(const char* str);
int inet_aton(const char* str,struct in_addr *addr)
```

- Both of these functions are used to convert the IPv4 internet address from dotted decimal C string format pointed to by `str` to 32-bit binary network byte ordered value
- The `inet_addr()` return this value, while `inet_aton()` function stores it through the pointer addr
- The newer function `inet_aton()` works with both IPv4 and IPv6, so one should use this call in the code even if working on IPv4

read() and write()

```
ssize_t read(int fd, void* buf, size_t count);
ssize_t write(int fd, const void* buf, size_t count);
```

- The **read()** and **write()** system calls can be used to read/write from files, devices, sockets, etc. (with any type of sockets stream or datagram)
- The **read()** call **attempts** to read up to count bytes from file descriptor `fd` into the buffer starting at `buf`. If no data is available read blocks. On success returns the number of bytes read and on error returns -1 with `errno` set appropriately
- The **write()** call writes `count` number of bytes starting from memory location pointed to by `buf`, to file descriptor `fd`. On success returns the number of bytes actually written and on error returns -1 with errno set appropriately
- The **send()** and **recv()** calls can be used for communicating over stream sockets or connected datagram sockets. If you want to use regular unconnected datagram sockets (UDP), you need to use the **sendto()** and **recvfrom()**

send()

```
int send(int sockfd, const void* buf, int count,int flags);
```

- The **send()** call writes the `count` number of bytes starting from memory location pointed to by `buf`, to file descriptor `sockfd`
- The argument `flags` is normally set to zero, if you want it to be "normal" data. You can set flag as `MSG_OOB` to send your data as "out of band". It's a way to tell the receiving system that this data has a higher priority than the normal data. The receiver will receive the signal `SIGURG` and in the handler it can then receive this data without first receiving all the rest of the normal data in the queue
- The **send()** call returns the number of bytes actually sent out and this might be less than the number you told it to send. If the value returned by **send()** does not match the value in `count`, it's up to you to send the rest of the string
- If the socket has been closed by any side, the process calling **send()** will get a `SIGPIPE` signal

recv()

```
int recv(int sockfd, void* buf, int count, int flags);
```

- The **recv()** call **attempts** to read up to `count` bytes from file descriptor `sockfd` into the buffer starting at `buf`. If no data is available it blocks
- The argument `flags` is normally set to zero, if you want it to be a regular vanilla `recv()`, you can set flag as `MSG_OOB` to receive out of band data. This is how to get data that has been sent to you with the `MSG_OOB` flag in `send()` As the receiving side, you will have had signal `SIGURG` raised telling you there is urgent data. In your handler for that signal, you could call `recv()` with this `MSG_OOB` flag
- The call returns the number of bytes actually read into the buffer, or -1 on error
- If **recv()** returns 0, this can mean only one thing, i.e., remote side has closed the connection on you

close()

```
int close(int fd);
```

- After a process is done using the socket, it can call `close()` to close it, and it will be freed up, never to be used again by that process
- On success returns zero, or -1 on error and `errno` will be set accordingly
- The remote side can tell if this happens in one of two ways:
    - If the remote side calls `read()`, it will return zero
    - If the remote side calls `write()`, it will receive a signal `SIGPIPE` and `write()` will return -1 and `errno` is set to `EPIPE`
- In practice, Linux implements a reference count mechanism to allow multiple processes to share a socket. If **n** processes share a socket, the reference count will be **n**. `close()` decrements the reference count each time a process calls it. Once the reference count reaches zero (i.e., all processes have called `close()`) the socket will be deallocated

shutdown()

```
int shutdown(int fd, int how);
```

- When you close a socket descriptor, it closes both sides of the socket for reading and writing, and frees the socket descriptor. If you just want to close one side or the other, you can use `shutdown()` call
- The argument `fd` is descriptor of the socket you want to perform this action on, and the action can be specified with the `how` parameter
- `SHUT-RD`(0): Further receives are disallowed
- `SHUT-WR`(1): Further sends are disallowed
- `SHUT-RDWR`(2): Further sends and receives are disallowed

Difference between close() and shutdown():

- `close()` closes the socket ID and frees the descriptor for the calling process only, the connection is still opened if another process shares this socket ID. The connection stays opened for both read and write
- `shutdown()` breaks the connection for all processes sharing the socket ID. It doesn't close the file descriptor or free the socket DS, it just change its usability. To free a socket descriptor, you still have to call `close()`

Demonstration

GitHub Code Repository Link: *https://github.com/arifpucit/OS-Codes*

POSIX Socket API
For TCP Server

Instructor: Muhammad Arif Butt, PhD

bind()

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

- A socket created by a server process must be bound to an address and it must be advertised. Thus any client process can later contact the server using this address
- The `bind()` call assigns the address given in the 2nd argument `myaddr`, to the socket referred to by the `sockfd` given in the 1st argument (obtained from a previous `socket()` call)
- The 2nd argument, `myaddr` is a pointer to a structure specifying the address to which this socket is to be bound. There are different address families and each having its own format. The type of structure passed in this argument depends on the socket domain
- The `addrlen` argument specifies the size in bytes of the address structure pointed to by `myaddr`
- On success, the call returns zero. On failure -1 is returned and `errno` is set appropriately

listen()

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

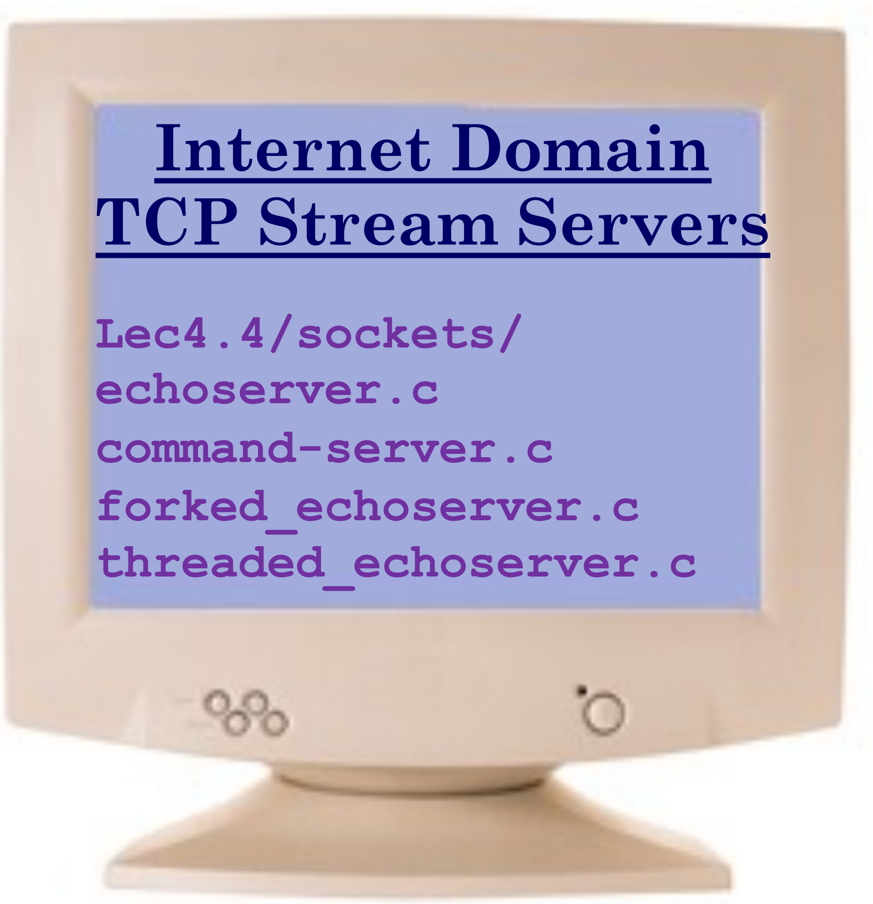
- The `listen()` system call requests the kernel to allow the specified socket mentioned in the 1st argument to receive incoming calls. (Not all types of sockets can receive incoming calls, SOCK_STREAM can)
- This call put a socket in passive mode and associate a queue where incoming connection requests may be placed if the server is busy accommodating a previous request
- The `backlog` argument is the number of connections allowed on the incoming queue. The maximum queue size depends on the socket implementation
- On success it returns zero and on failure -1 is returned and `errno` is set appropriately
- We need to call `bind()` before we call `listen()`, otherwise the kernel will have us listening on a random port

accept()

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

- The **accept()** system call is used by server process and returns a new socket descriptor to use for a new client. After this the server process has two socket descriptors; the original one (master socket) is still listening on the port and new one (slave socket) is ready to be read and written
- It is used with connection-based socket types (SOCK_STREAM)
- The argument `sockfd` is a socket that has been created with `socket()`, bound to a local address with `bind()`, and is listening for connections
- On success, the kernel puts the address of the client into the second argument pointed to by `callerid` and puts the length of that address structure into the third argument pointed to by `addrlen`
- On success return a non-negative integer that is a descriptor for the accepted socket. On failure -1 is returned and `errno` is set appropriately

Demonstration



GitHub Code Repository Link: https://github.com/arifpucit/OS-Codes

To Do

- Watch SP video overview of TCP/IP
  https://youtu.be/p5SrRob-bWg?si=r7fIr9DlyYm4Y4w5

- Watch SP video on Socket Programming Part-I (Internet Domain TCP Sockets)
  https://youtu.be/tk_RpIVbOMQ?si=PlAf7Q_KSmsr9gWX

- Watch SP video on Socket Programming Part-II (Internet Domain UDP Sockets)
  https://youtu.be/yNUFQaSclmM?si=_oI1CsjqIGN1j7Pq

- Watch SP video on Socket Programming Part-III (UNIX Domain Sockets)
  https://youtu.be/TDRIweWXHe4?si=_OARm5gVKrxd2nAO

- Watch SP video on Socket Programming Part-IV (Design of Concurrent Servers)
  https://youtu.be/irRkNrruwxc?si=IV2rX3f5H1kqaXgK

Coming to office hours does NOT mean that you are academically weak!