University of Southern California

Viterbi School of Engineering

EE450

Computer Networks

Socket Programming

Shahin Nazarian

Spring 2013

Network Applications Development

- To develop a network application, you should write programs (e.g., in C++ or Java) that run on different end systems and communicate with each other over the network

- Client-server architecture

- Peer-to-peer

Network Applications Development (Cont.)

- You won't need to write programs that run on network core devices such as routers or layer 2 switches. We know that they do not have (and do not need) an application layer

- In the jargon of operating systems, it is not the programs but **processes** that communicate with each other.

Process Communication

- Process: Program running within a host
- Within same host, two processes communicate using inter-process communication (defined by the OS)
- Processes in different hosts communicate by exchanging messages across the computer network
  - A network application typically consists of pairs of processes that send messages to each other over a network
- Example: Web application: A web browser client process (in user's host) exchanges messages with a web server process
- Note that any message sent from one process to another must go through the underlying network
- Client process
- Server process

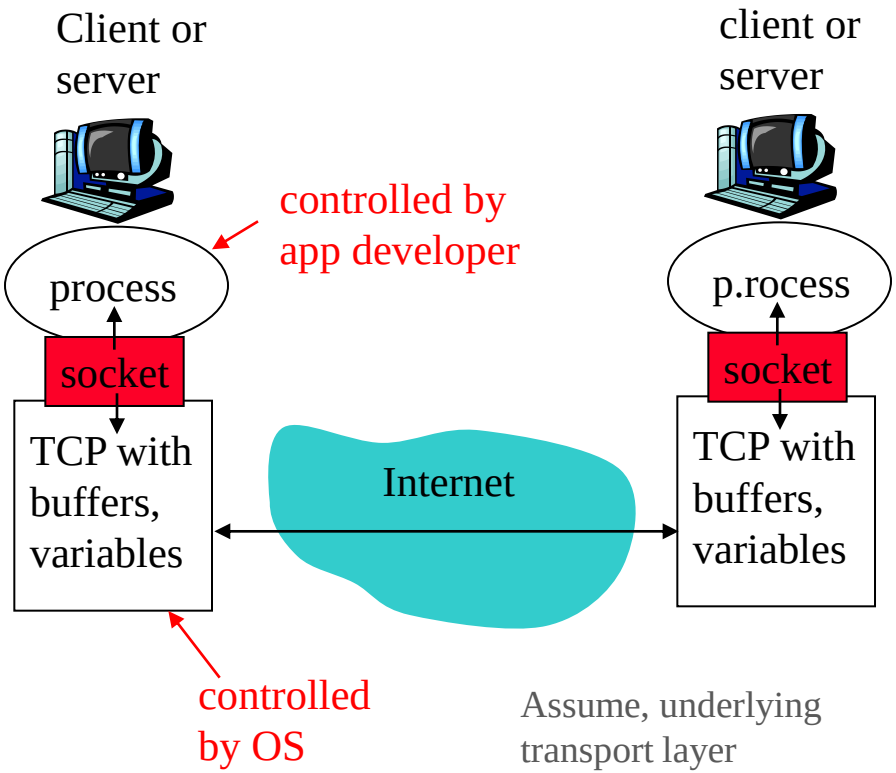
Process Communication in P2P

- In a P2P file-sharing system, a file is transferred from a process in one peer to a process in another peer
  - For each pair of communicating processes we typically label one of the two processes as the client and the other process as the server
- Therefore here a process can be both a client and a server. Nevertheless in the context of any given communication session btn a pair of processes, we can still label one process as the client and the other process as the server

Socket

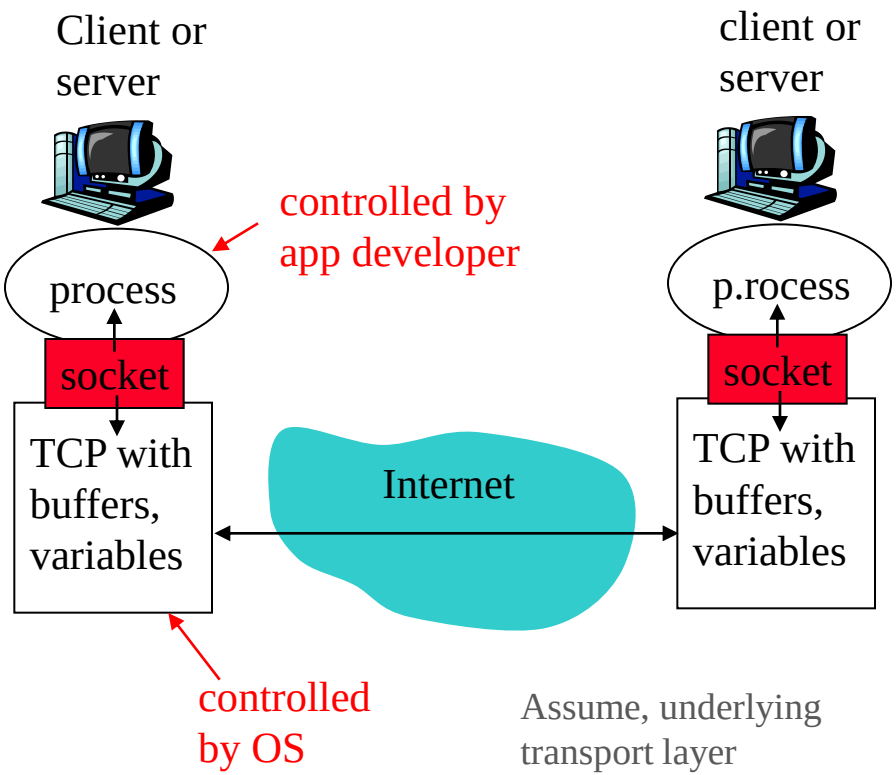
- A process sends messages into, and receives messages from, the network through a software interface called a **socket** or the **network API**

- Socket is an interface (an abstraction, not real connection) through which applications communicate among each other



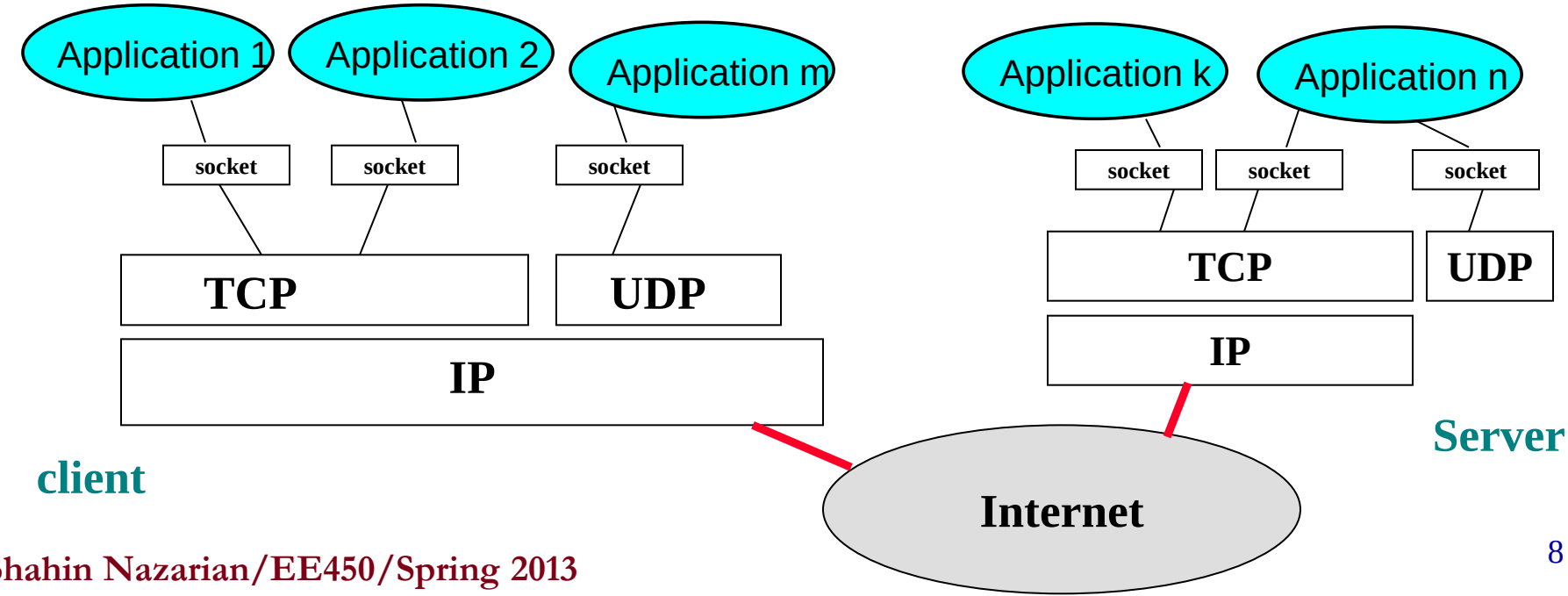
Socket Analogy

- Process is analogous to a house and socket is its door
- Sending process shoves message out the door
- Sending process relies on transport infrastructure on other side of door which brings message to the socket at receiving process
- The application developer has control of everything on the Application-layer side of the socket but has little control on the Transport layer side of the socket



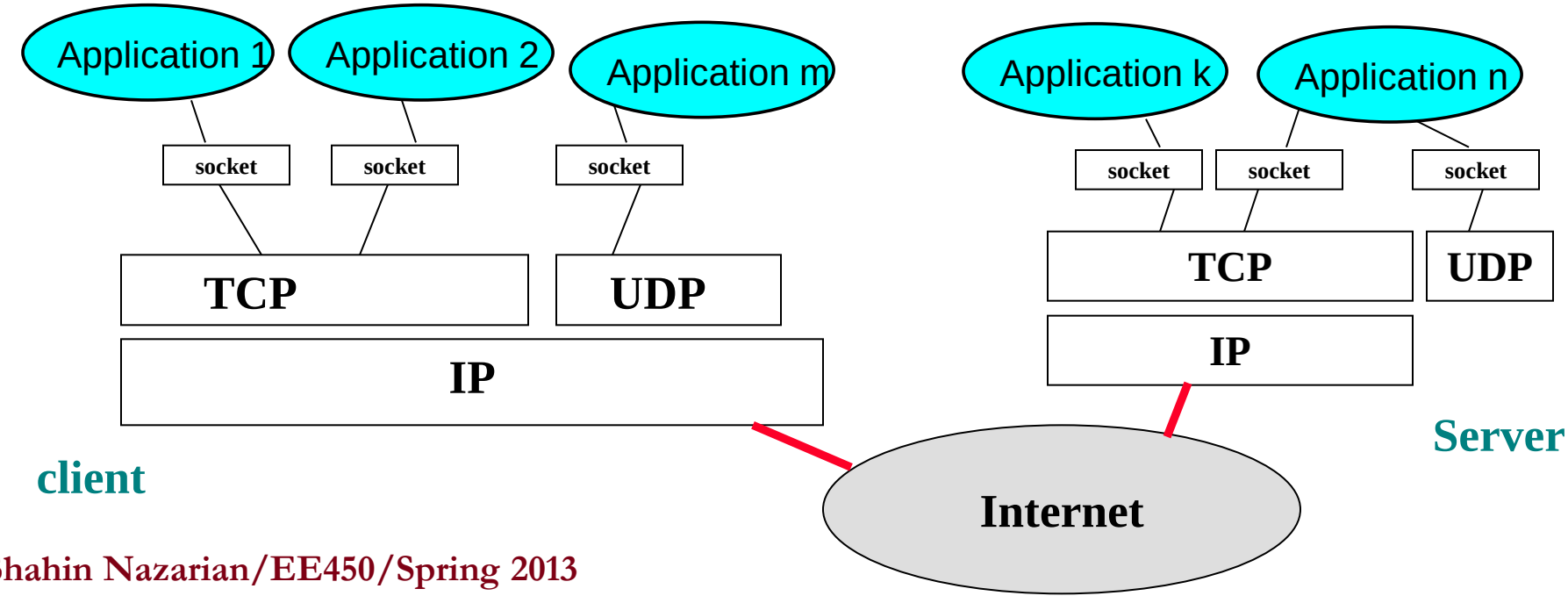
Socket (Cont.)

- The control on the Transport layer side is only
  - the choice of the transport layer protocol, (e.g., UDP or TCP)
  - and perhaps the ability to fix a few Transport layer parameters such as maximum buffer and maximum segment sizes



Socket (Cont.)

- Socket resides in btn Application & Transport layer protocols
- Socket comes in different types, such as UNIX sockets or Windows sockets (WinSocks)
- Important: Socket is independent of TCP/IP, so TCP/IP does not include a definition for API, The network (i.e., routers in the internet) only sees the IP, does not care about Transport and Application layer protocols



RFC-based Network Applications

There are two kinds of network applications:

(1) RFC-based (2) Proprietary

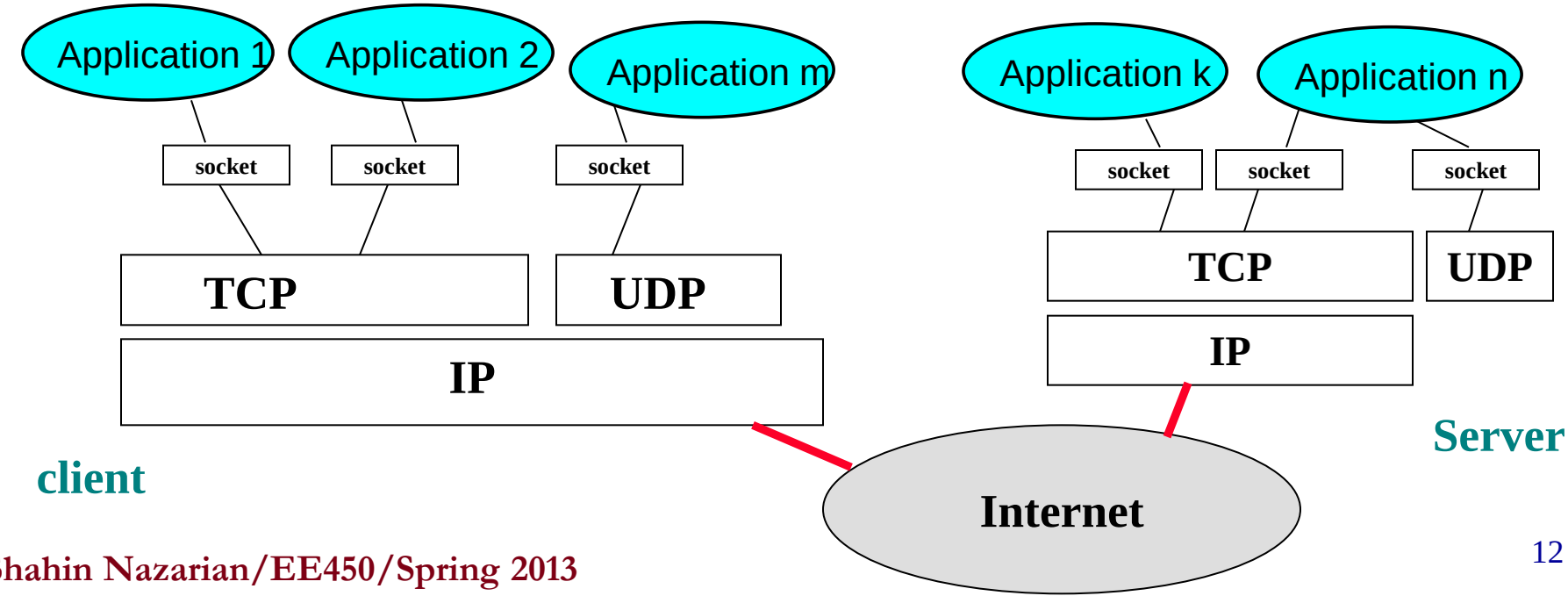
- **RFC-based**: The client and server programs must conform to the rules dictated by the RFC. They will also use port numbers associated with the protocol
  - Example: FTP protocol based on rules explicitly defined in RFC959: Both client and server programs must follow RFC959
- This means two independent developers can write the client and server programs respectively, and the two programs will be able to interoperate
  - Example: Firefox browser communicating with an Apache Web server
  - Example: An FTP client on PC uploading file to a Linux FTP server

Proprietary Network Applications

- **Proprietary**: The application layer protocol used by the client and server programs do not necessarily conform to any existing RFC
- A single development team should create both client and server programs. However since the codes do not implement a public domain protocol, other developers won't be able to develop code that interoperates with the applications
- Also note that the developer of a proprietary application should not use well-known port numbers defined in the RFC
- During development phase of a proprietary application, one of the 1st decisions to make is which one TCP, or UDP should be used as the underlying transport layer

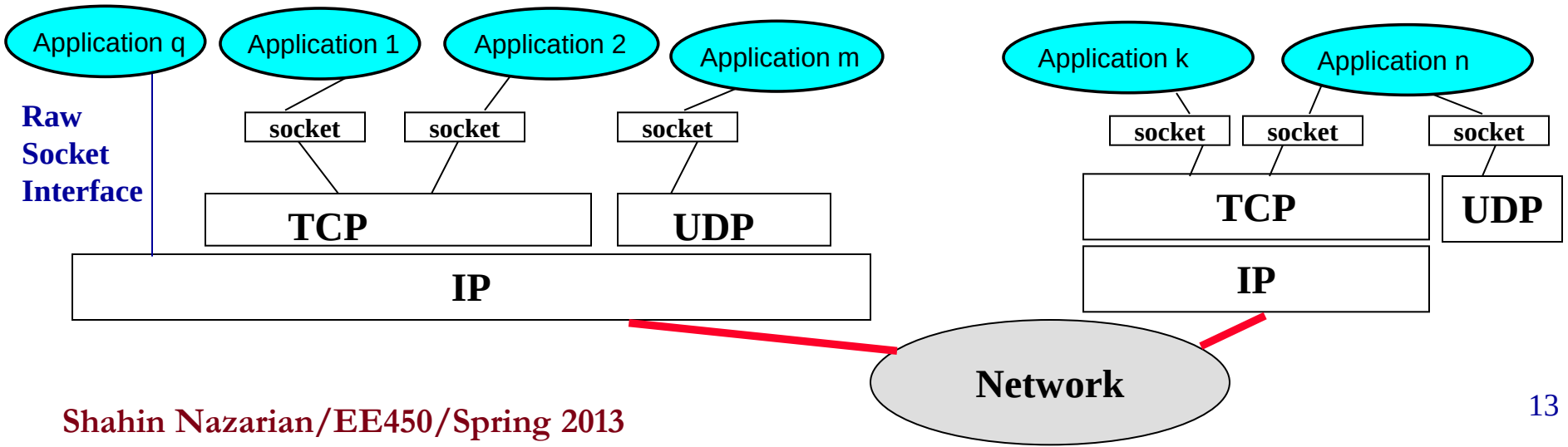
Application, Process, Socket and Transport Layer

- Remember that resembling an end system to a house and an application data to the person living in the house, socket is like the house door, so the person who wants to travel, needs to open the door (i.e., create the socket and open it) the person then sees the TCP or UDP for transportation, therefore TCP can be looked as a means of travel, like a car. Traveling happens all the way across the network through the path to destination house, where the person goes through the door to destination (e.g., server side)



Stream and Datagram Sockets

- Applications create sockets that "plug into" the network
- Applications send/receive to/from sockets. Sockets are implemented in kernel, facilitate the development of network apps and hide details of underlying protocols & mechanisms
- **Stream sockets** run over TCP, a reliable socket, i.e., connection-oriented, or with handshaking, e.g., HTTP, FTP, SMTP, Telnet
- **Datagram sockets** run over **UDP** (User Datagram Protocol) unreliable, connectionless, i.e., no handshaking

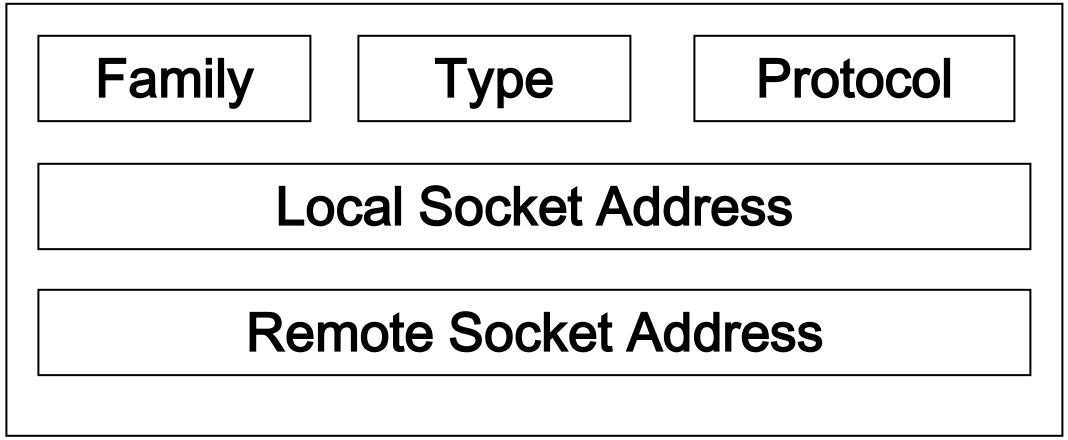


Sever and Client Sockets

- Sockets need to be created in both sides, the client side and the server side
- Server socket: once created it will be always open and serves multiple clients
- Client socket has the job of initiating contact with server. In order for the server to react to the client's initial contact, the server has to be ready
- Commands on the server side include Create, Bind, Listen, Accept, Send/Rev, Close (the Child Socket)
- Commands on the client side include Create, Connect (only for stream sockets,) Send/Rev [Note: in case of Datagram sockets once you create the socket, you can start sending/receiving,] Close Socket

Socket Structure

- Socket domain (family), e.g., the Internet protocol family (PF_INET)
- Socket type, e.g., stream (SOCK_STREAM) or datagram (SOCK_DGRAM)
- Protocol (Transport layer protocol,) e.g, TCP (IPPROTO_TCP) or UDP (IPPROTO_UDP)
  - Note that for the cases we are interested, the socket type would imply the protocol, meaning that the stream type is on top of TCP and the datagram type is on top of the UDP
- IP address, port number + underlying Transport protocol



Create Socket – Server Side

- The application on the server creates the socket by issuing the create command to the NOS (kernel)
- The socket domain, type, and protocol should be specified during creation:

  Int Socket(PF, Type, Protocol)

- Kernel returns to application a non-negative integer number
  - -1 is returned if error occurs
- Remember: this server socket is **passively open**, i.e., once created the socket will always be open for a client request and it serves multiple clients
- When a socket is created using socket(), it is only given a protocol family, but not assigned an address

Bind Socket – Server Side

- The Bind command is used to associate the socket (that was just created) with the local address, i.e., for the server side, the server address
- By this command the application asks the kernel (NOS) to associate the socket (whose descriptor was just returned to the application by kernel) with the well-known port number and the IP address of the server side:

  Int Bind (Int Socket, address, addr length)

- addr length is 16 bits for the port number and 32 bits for the IP address. In response to the application command NOS returns to the application a 0 if everything is fine, and -1 in case of an error, e.g., syntax error
- Note: regardless of the client side, the server side needs to be ready (the application at server side creates and binds the socket)

Listen – Server Side

- After a socket has been associated with an address, listen() prepares it for incoming connections. However, this is only necessary for the stream-oriented (connection-oriented) data modes
- By the Listen command, the application at the server side asks the kernel to listen to all incoming requests to the specific socket (Note again that this socket is passively open)

  Int Listen (Int Socket, Queue Limit)

  where the Queue limit (or backlog) is the maximum number of requests that are allowed to come (from the client side)

  - Note that Listen call allocates space to queue incoming calls for the case several clients try to connect at the same time

Accept – Server Side

- When an application is listening for stream-oriented connections from other hosts, it is notified of such events and must initialize the connection using accept()
- By Accept, the server side application asks the kernel to remove the first request from the Queue [i.e., accept it]

Int Accept (Int Socket, remote address, addr length)

- where Socket is the descriptor of the parent socket. The addr length is 16 bits for the client port number and 32 bits for the client IP address
- After the connection with the client is successfully established, NOS returns a non zero descriptor for a socket called the child socket. The server can then fork off a process or thread to handle connection on new socket and go back to waiting on orig. socket

Parent and Child Sockets

- The socket that was originally created, i.e., the parent socket is going to be used only to listen to the client requests, and it is not going to be used for communication between client and the server

- The child socket (is a duplicate of the parent socket and) is created when the communication with the client is established and going to be used for data exchange between the server and the client. Note that the parent socket is not for data exchange

Parent and Child Sockets (Cont.)

- The parent socket is like a main entrance which is always open, then once a person (client) is accepted to enter the building, then the child socket, a small entrance specific for that client is going to be opened.
- Note: Child sockets that are created for a parent socket have the identical well-known port number at the server side and also identical IP address for the server, but note that each child socket is created for a specific client
- Parent socket is only for listening; every time a client request is accepted, it creates a child socket which is going be used for communication
- Question: Which socket will eventually be closed by server?

Send, Receive & Close – Server Side

Int Send (Int Socket, message, message length)

Int Rev (Int Socket, Buffer, Buffer size)

Int Close (Int Socket)

- where for all, Socket is the descriptor of the child socket
- After a child socket is closed, the server side application cannot send or receive messages using that socket
- Question: can Close be used to close the parent socket? No, the parent socket remains open

Create a Socket – Client Side

Int Socket (PF, Type, Protocol)

- NOS responds by sending a non zero descriptor
- The difference btn creating a socket on the server side and on the client side is that on the client side there is no Bind command, because the client side does not need to ask the kernel to bind the socket that was created with a port number, and an IP address. Note that NOS of the client side would do that automatically, by choosing a random port number and the client IP address
- Remember that the client socket will be actively open, i.e., the client will open it when wants service

Connect – Client Side

- Connect command initiates the "handshaking" process

Int Connect (Int Socket, remote address, addr length)

- Where Socket is the descriptor of the socket
- Remember the Bind command on the server side and see that both Bind at the server side and Connect on the client side need server's IP address and the well-known port number
- Note that the server knows about client's port number and its IP address through the Connect command
- Connect first blocks the caller and actively starts the connection process. When it completes (i.e., appropriate segment received from server,) client process is unblocked and the connection is established
- FDX data send/receive

Send, Receive, & Close – Client Side

- Send, Rev and close commands on the client side are the same as those on the server side:

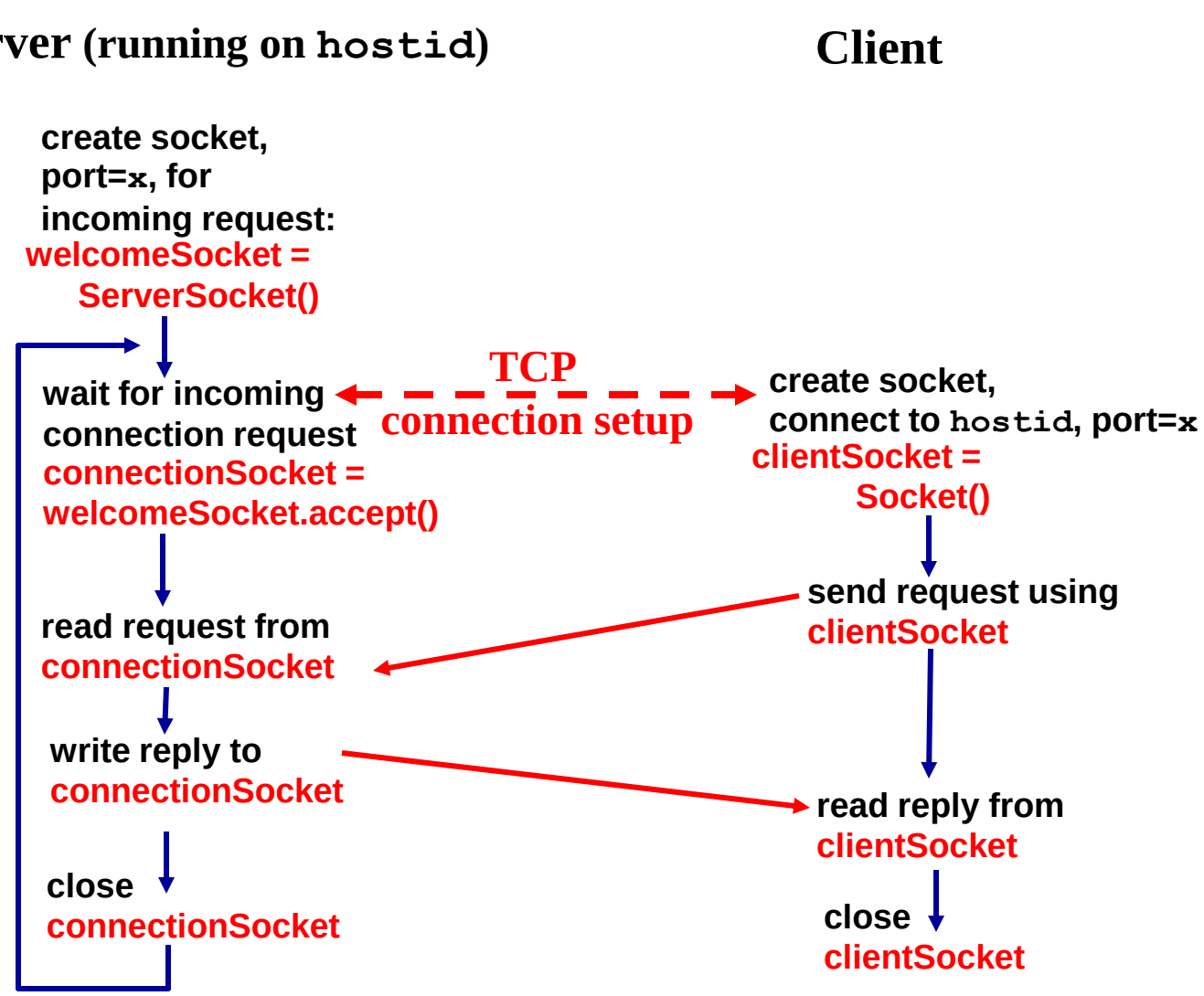
  Int Send (Int Socket, message, message length)

  Int Rev (Int Socket, Buffer, Buffer size)

  Int Close (Int Socket)

- where for all, Socket is the descriptor of client socket
- Question: Can server and client sockets send data when the client terminates the connection? Client can only send acknowledgement to the data that it's receiving from the server
- Connection release with sockets is symmetric. When both sides have executed a Close the connection is released

Client/server Socket Interaction: TCP



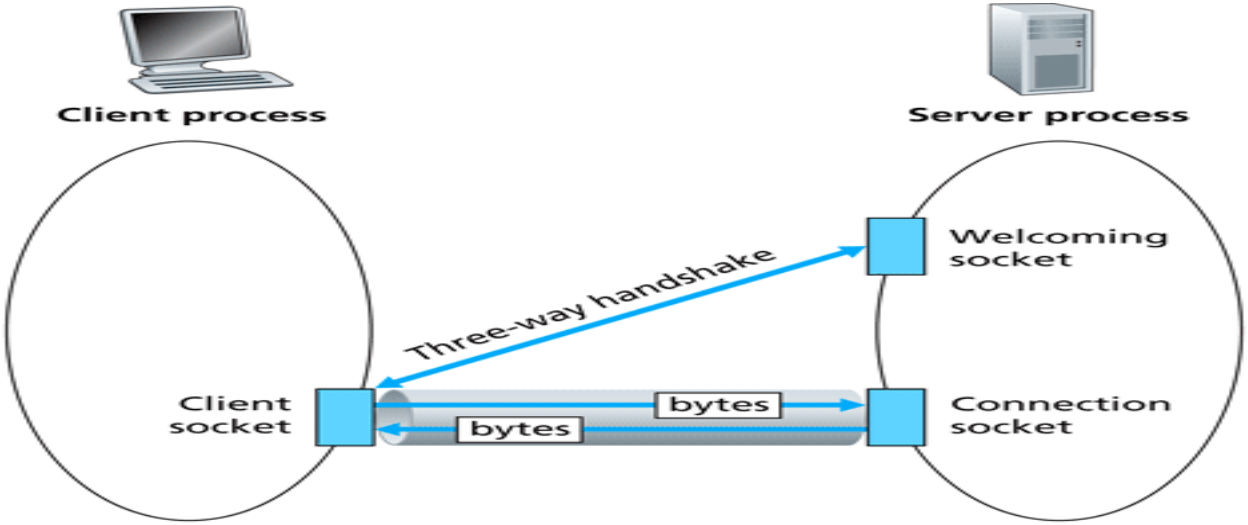
Socket Programming with TCP – Summary

- TCP connection is point-to-point and is FDX
- With the server (parent or welcoming) process running, the client process can initiate TCP connection to the server. This is done in the client program by creating a socket
- Once the client socket has been created in the client program, TCP in the client initiates a 3-way handshake (during which, 3 special segments are sent between the two hosts) and establishes a TCP connection with the server. The 3-way handshake, which takes place at Transport layer, is completely transparent to client & server programs
- During the 3-way handshake, the client process knocks on the welcoming door of the server process. When the server hears the knocking, it creates a new door (child socket) which is dedicated to the particular client. At the end of the handshake phase, a TCP connection exists btn client's socket and server's child socket

TCP Socket Programming Summary (Cont.)

- From application's perspective, the TCP connection is a direct virtual pipe btn the client's socket and the server's child socket

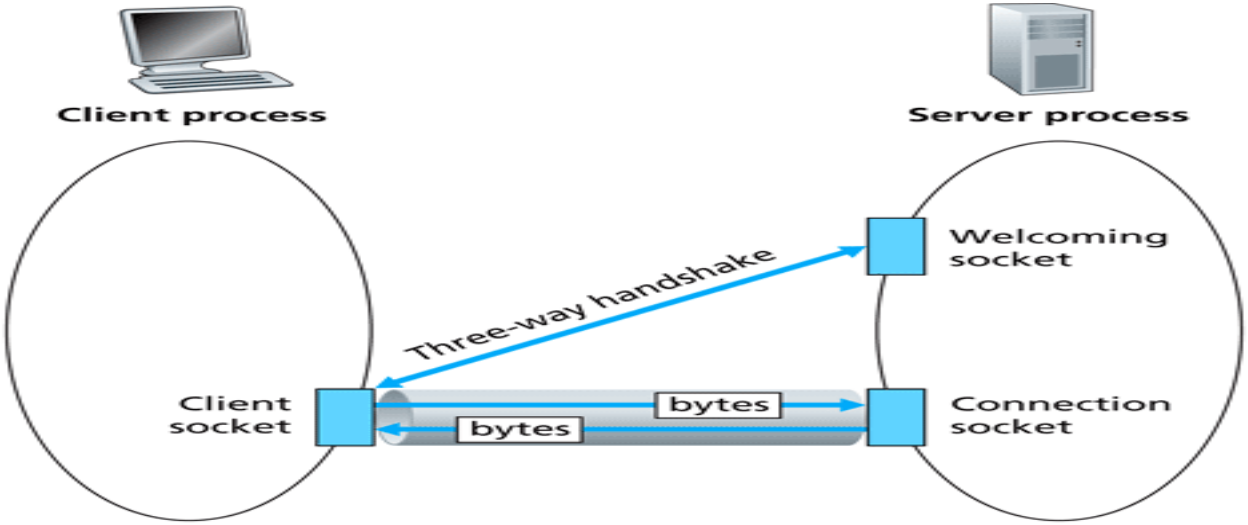
- The client process can send **arbitrary bytes** into its socket and TCP guarantees that the server process **will receive each byte in the order sent**



TCP Socket Programming Summary (Cont.)

- Furthermore, just as people can go in and out the same door, the client process not only sends bytes into but also receives bytes from its socket

- Similarly, the server process not only receives bytes from but also sends bytes into its connection socket



Socket API on Other Transport Protocols

- A socket API can be used by an application for other transport services. E.g., sockets can be used with a connectionless transport service. In this case Connect sets the address of remote transport peer and Send and Receive, send & receive datagrams to & from remote peer
- Sockets can also be used with transport protocols that may or may not have congestion control. E.g., UDP does not, but DCCP (Datagram Congestion Controlled Protocol) is a UDP with congestion control
- SCTP (Stream Control Transmission Protocol) and SST (Structured Stream Transport) protocols slightly modify socket APIs to get the benefits of both connection-oriented and connectionless protocols, e.g., by grouping streams while doing congestion control. However they are not established well yet

Socket Programming with UDP

- When two processes communicate over TCP (connection-oriented service) it is as if there were a pipe btn the two processes & remains open until one of the two closes it
- When one process wants to send some bytes to the other, it simply inserts the bytes into the pipe. The sending process does not have to attach a destination address to the bytes, because the pipe is logically connected to the destination
- Furthermore, the pipe provides a reliable byte-stream channel, meaning the sequence of bytes received by the receiving process is exactly the sequence of bytes that the sender inserted into the pipe

Socket Programming with UDP (Cont.)

- UDP also allows two (or more) processes running on different hosts to communicate, however UDP is a connectionless service
- This means there is no initial handshaking btn the two processes
- Since UDP does not have a pipe, the sending process must attach the destination process's address [a tuple: destination's IP address + port number] to the batch of bytes that the sending process wants to send
- UDP service is best effort

Socket Programming with UDP (Cont.)

- With TCP, the server works with multiple clients, so it cannot use the listening (parent) socket for communications, therefore multiple child sockets are created
- However in connectionless service, the parent socket is used for communication, so child sockets are not required to be created
- In socket programming with UDP, Connect command is not required, because no handshaking is involved
- TCP is a connection-oriented protocol
- UDP is message-oriented

Client/server Socket Interaction: UDP