**+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++Test 1 Review**

**+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++**

**CHP 1 - Basics:**

* Protocols: Define format and order of messages sent and received among network entities, and actions taken on message transmission or message receipt
  + Human Protocols: Specific messages sent, and specific actions being taken when messages are received
  + Network Protocols: Involves machines rather than humans; all communication activity in the Internet is governed by protocols
* Network Edge: Computers and other devices that are connected to the Internet
* Access Networks: Physically connects an end system to the first router (“edge router”) on a path from the end system to any other distant end system
* Network Core: Mesh of packet switches and links that interconnects the Internet’s end systems
* How to connect end systems to edge router?
  + Residential access nets, institutional access networks (school, company), and mobile access networks
* Host Sending Function: Takes application message, breaks it into smaller chunks (known as packets) of length L bits, and transmits those packets into access network at transmission rate R
  + Packet Transmission Delay: L (bits) / R (bits/sec)
  + The transmission rate is otherwise known as the link capacity / link bandwidth
* Bit: Propagates between transmitter/receiver pairs
* Physical Link: What lies between the transmitter and the receiver
* Guided (Physical) Media: Signals propagate in solid media (copper, fiber, coax)
  + Twisted Pair (TP): Two insulated copper wires
  + Coaxial Cable: Two concentric copper conductors; bidirectional; broadband (multiple channels on the cable)
  + Fiber Optic Cable: Glass fiber carrying light pulses, each pulse being a bit; high-speed point-to-point transmission; low error rate due to repeaters being spaced apart; immune to electromagnetic noise (interference)
* Unguided Media: Signals propagate freely (e.g., radio)
  + Radio: Signal carried in electromagnetic spectrum; bidirectional; affected by reflection, obstruction by objects, and interference
    - Terrestrial Microwave: Up to 45 Mbps channels
    - LAN (e.g., WiFi): 11 Mbps, 54 Mbps
    - Wide-area (e.g., Cellular): 3G cellular; few Mbps
    - Satellite: Kbps to 45 Mbps channel (or multiple smaller channels); 270 msec end-to-end delay; geosynchronous versus low altitude
* Packet-Switching: Hosts break application-layer messages into packets; forward packets from one router to the next, across links on path from source to destination; each packet transmitted at *full* transmission rate of the link; allows more users to use a network, while giving the same performance as circuit switching with minimal simultaneous users
  + Takes L/R seconds to transmit (push out) L-bit packet into link at R bps
  + Store and Forward: Entire packet must arrive at router before it can be transmitted on next link
  + End-to-end Delay: NL/R, where N = number of links, R = the transmission rate for all the links, and L = length of the packet
    - For calculations involving more than one packet, multiply the formula above by the number of packets
  + It is great for bursty data and resource sharing; much simpler to implement, due to not having a call setup
  + Bandwidth guarantees are needed for audio/video apps, so it remains easier to use circuit switching for those applications
  + Excessive congestion is possible, due to packet delay / loss
    - Protocols are needed for reliable data transfer and congestion control
  + Loss/delay occurs when packets queue in router buffers; packet arrival rate to link (temporarily) exceeds output link capacity, so packets queue and wait for turn
    - Packet Loss: Queue (aka buffer) preceding link in buffer has finite capacity; packet arriving to the full queue is dropped (aka lost); the lost packet may be retransmitted by previous node, by source end system, or not at all
    - Nodal Processing Delay: Check bit errors, determine output link; typically < msec
    - Queuing Delay: Time waiting at output link for transmission; depends on congestion level of router
      * La/R (traffic intensity) can determine the queuing delay (where L = packet length in bits, R = link bandwidth in bps, and a = average packet arrival rate)
        + If La/R ~ 0: the average queuing delay is small
        + If La/R ~ 1: the average queuing delay is large
        + If La/R > 1: there is more “work” arriving than can be serviced, so the average delay is infinite!
    - Transmission Delay: L/R (where L = packet length in bits and R = link bandwidth in bps)
    - Propagation Delay: d/s (where d = length of physical link and s = propagation speed in medium); s is in the range of 2 \* 108 m/s to 3 \* 108 m/s
    - Total Nodal Delay: The sum of all four previous delays
    - End-to-End Delay: N(dproc + dtrans + dprop), where N = the number of routers + the destination host, dproc = the processing delay at each router and at the source host, dtrans = L/R (L = packet size, R = transmission rate), and dprop = the propagation on each link.
  + Throughput: Rate (bits/time unit) at which bits transferred between sender/receiver
    - Instantaneous: Rate at a given point in time (in bits/sec) at which Host B is receiving the file.
    - Average: Rate over a longer period of time; if the file consists of F bits and the transfer takes T seconds for Host B to receive all F bits, then it is F/T bits/sec.
    - Bottleneck Link: Link on end-to-end path that constrains end-to-end throughput
      * If RS < RC, then the bits pumped by the server will “flow” right through the router and arrive at the client at a rate of RS bps, giving a throughput of RS bps.
      * If RC < RS, then the router will not be able to forward bits as quickly as it receives them; bits leave the router at rate RC, giving an end-to-end throughput of RC bps.
      * For a simple two-link network, the throughput is min{RC, RS} (where RC = transmission rate of the link between the router and the client and RS = transmission rate of the link between the server and the router).
* Circuit Switching: End-to-end resources are allocated and reserved for “call” between source and destination; implemented with either FDM or TDM
  + Frequency-Division Multiplexing (FDM): The link dedicates a frequency band to each connection for the duration of the connection; the width of the band is called the bandwidth
  + Time-Division Multiplexing (TDM): Time is divided into frames of fixed duration, and each frame is divided into a fixed number of time slots; when the network establishes a connection across a link, the network dedicates one time slot in every frame to this connection
  + No sharing resources, giving a circuit-like (guaranteed) performance
  + The circuit segment remains idle if not being used by a call (no sharing); referred to as silent periods
* Routing Protocols: Determines source-destination route taken by packets via routing algorithms that automatically set the forwarding tables
* Forwarding Table: Maps destination addresses (or portions of the destination addresses) to that router’s outbound links; when a packet arrives at a router, the router examines the address and searches its forwarding table, using this destination address to find the appropriate outbound link
* Access Networks: End systems are connected to edge routers through residential access nets, institutional access networks (school, company), and mobile access networks.
  + PoP: A group of one or more routers (at the same location) in the provider’s network where customer ISPs can connect into the provider ISP
  + Peering: When a pair of nearby ISPs at the same level of the hierarchy directly connect their networks together so that all the traffic between them passes over the direct connection rather than through upstream intermediaries
  + IXP: A meeting point where multiple ISPs can peer together.
  + Regional ISP: An ISP in a region to which access ISPs in the region connect
  + Tier-1 Commercial ISPs: National and international coverage (e.g., Level 3, Sprint, AT&TT, and NTT); there are approximately 12 of these in the world
  + Content Provider Network: Private network that connects its data centers to Internet, often bypassing tier-1 and regional ISPs (e.g., Google)
  + Multi-Home: When an ISP (any ISP, other than a tier-1 ISP) connects to two or more provider ISPs
* Protocol Layering: Deals with complex systems; an explicit structure allows identification and a relationship of complex system pieces; can be implemented in software, hardware, or in a combination of the two.
  + Modularization: Eases maintenance and updating of system
    - A change of the implementation of a layer’s service is transparent to the rest of the system (e.g., a change in gate procedure doesn’t affect the rest of the system)
* Internet Protocol Stack:
  + Application: Supporting network applications (e.g., FTP, SMTP, HTTP)
  + Transport: Process-to-process data transfer (e.g., TCP, UDP)
  + Network: Routing of datagrams from source to destination (e.g., IP, routing protocols)
  + Link: Data transfer between neighboring network elements (e.g., Ethernet, 802.11 (WiFi), PPP); packets in this layer are referred to as frames
  + Physical: Bits “on the wire”; moves the *individual bits* within the frame from one node to the next
  + The transport layer transports application-layer messages (packets of information in the application layer) between application endpoints; the transport-layer protocol in a source host passes a transport-layer segment and destination address to the network layer, which then transports it to the transport-layer in the destination host; the link layer is responsible for receiving the datagram from the network layer and passing it to the next node along the route, where it passes it back up to the network layer; the physical layer moves the individual bits within each frame from one node to the next.
    - In this way, each layer relies on the services of the layer below it for data transportation to be possible.
  + ISO/OSI Reference Model: Has a presentation and session layer (in that order) between the application and transport layer
    - Presentation: Allows applications to interpret the meaning of data (e.g., encryption, compression, machine-specific conventions)
    - Session: Synchronization, checkpointing, and recovery of data exchange
    - The internet stack is “missing” these layers; if these services are needed, they must be implemented in the application
* Internet History:
  + 1961: Queuing theory shows effectiveness of packet switching
  + 1964: Packet-switching implemented in military nets
  + 1967: ARPAnet is conceived by Advanced Research Projects Agency
  + 1969: First ARPAnet node operational
  + 1970: ALOHAnet satellite network is released in Hawaii
  + 1972: ARPAnet public demo is released, NCP (Network Control Protocol) has first host-to-host protocol, the first e-mail program is released, and ARPAnet has 15 nodes
  + 1974: Cerf and Kahn - Architecture for interconnecting networks
  + 1976: Ethernet at Xerox PARC
  + 1979: ARPAnet has 200 nodes
  + Late 70’s: Proprietary architectures (DECnet, SNA, XNA) released; switching fixed length packets (ATM precursor)
  + 1982: SMTP e-mail protocol defined
  + 1983: Deployment of TCP/IP, and DNS defined for name-to-IP-address translation
  + 1985: FTP protocol defined
  + 1988: TCP congestion control
  + Early 1990’s: ARPAnet decommissioned, web hypertext released (HTML, HTTP)
  + 1991: NSF lifts restrictions on commercial use of NSFnet (decommissioned in 1995)
  + 1994: Mosaic released (later changed to Netscape)
  + Late 1990’s – 2000’s: Commercialization of the web, network security at a forefront, backbone links running at Gbps, 50 million hosts, 100 million+ users, and new apps released (P2P file sharing, instant messaging, etc)

**CHP 2 – Application Layer:**

* Network apps: E-mail, web, text messaging, remote login, P2P file sharing, multi-user network games, streaming stored video (YouTube, Hulu, Netflix), Voice over IP (VoIP, e.g., Skype), real-time video conferencing, social networking, searching, etc
* Creating a Network App:
  + Write programs that run on (different) end systems and communicate over a network (e.g., web server software communicates with browser software)
  + No need to write software for network-core devices – they do not run user applications
  + Applications on end systems allows for rapid app development and propagation
* Application Architectures: Can be structured as client-server or peer-to-peer (P2P)
  + Client-Server Architecture:
    - Server: Always-on host; permanent IP address; data centers for scaling
    - Clients: Communicate with the server; may be intermittently connected; may have dynamic IP addresses; do not communicate directly with each other
  + P2P Architecture: No always-on server; arbitrary end systems directly communicate with each other; peers request service from other peers, provide service in return to other peers
    - Self Scalability: New peers bring new service capacity, as well as new service demands
    - Peers are intermittently connected and change IP addresses – complex management
* Process: Program running within a host; within same host, two processes communicate using inter-process communication (defined by OS); processes in different hosts communicate by exchanging messages; applications with P2P architectures have client processes and server processes
  + Client Process: Process that initiates communication
  + Server Process: Process that waits to be connected
* Sockets: Process sends/receives messages to/from its socket; analogous to a door
  + Sending a process shoves a message out of the “door”; relies on transport infrastructure on other side of “door” to deliver message to socket at receiving process
* Addressing Processes:
  + To receive messages, process must have identifier; host-device has unique 32-bit IP address
  + Does the IP address of the host on which the process runs suffice for identifying the process?
    - No, *any* processes can be running on the same host
  + Identifier: Includes both IP address and port numbers associated with process on host (HTTP server port is 80, mail server port is 25, etc)
* The app-layer protocol defines the types of messages exchanged (e.g., request, response), the message syntax and semantics, and the rules for when and how processes send and respond to messages
  + Message Syntax: What fields in message and how fields are delineated
  + Message Semantics: Meaning of information in the fields
  + Open Protocols: Defined in RFCs; allows for interoperability (e.g., HTTP, SMTP)
  + Proprietary Protocols: Skype, etc
* What transport service does an app need?
  + Data Integrity: Some apps (e.g., file transfer, web transactions) require 100% reliable data transfer; other apps (e.g., audio) can tolerate some loss
  + Timing: Some apps (e.g., Internet telephony, interactive games) require low delay to be “effective”
  + Throughput: Some apps (e.g., multimedia) require minimum amount of throughput to be “effective”; other apps (“elastic apps”) make use of whatever throughput they get
  + Security: Encryption, data integrity, etc
* TCP Service:
  + Reliable Transport between sending and receiving process
  + Flow Control: Sender won’t overwhelm receiver
  + Congestion Control: Throttle sender when network overloaded
  + Does Not Provide: Timing, minimum throughput guarantee, or security
  + Connection-Oriented: Setup required between client and server processes
  + Usually used for e-mail, file transfer, remote terminal access, and websites
* UDP Service:
  + Unreliable Data Transfer between sending and receiving process
  + Does Not Provide: Reliability, flow control, congestion control, timing, throughput guarantee, security, or connection setup
  + Usually used for multimedia streaming or internet telephony
* Securing TCP:
  + TCP and UDP: No encryption; cleartext passwords sent into socket, traverse internet in cleartext
  + SSL: Provides encrypted TCP connection; data integrity; end-point authentication; apps use SSL libraries, which “talk” to TCP
    - The socket API takes any cleartext passwords sent into the socket and traverses them across the internet after encryption
* A web page consists of objects; each object can be an HTML file, JPEG image, Java applet, audio file, etc.
  + Consists of base HTML-file which includes several referenced objects; each object is addressable by a URL
* HTTP (Hypertext Transfer Protocol): Web’s application layer protocol; is a client/server model
  + Client: Browser that requests, receives (using HTTP protocol), and “displays” web objects
  + Server: Web server sends (using HTTP protocol) objects in response to requests
  + Uses TCP: Client initiates TCP connection (creates socket) to server, port 80; server accepts TCP connection from client
    - HTTP messages (application-layer protocol messages) exchanged between browser (HTTP client) and Web server (HTTP server); TCP connection closed
  + HTTP is “Stateless”: Server maintains no information about past client requests
    - Protocols that maintain “state” are complex; past history (state) must be maintained; if server/client crashes, their views of “state” may be inconsistent, must be reconciled
  + Response Time (RTT): Time for a small packet to travel from client to server and back
  + Non-Persistent HTTP: At most one object sent over TCP connection, connection then closed; downloading multiple objects required multiple connections
    - HTTP Response Time: One RTT to initiate TCP connection, one RTT for HTTP request and first few bytes of HTTP response to return; file transmission time; response time = 2RTT + file transmission time
  + Persistent HTTP: Multiple objects can be sent over single TCP connection between client and server
* Suppose user enters URL [www.someschool.edu/somedepartment/home.index](http://www.someschool.edu/somedepartment/home.index) (contains text, references to 10 jpeg images)
  + HTTP client initiates TCP connection to HTTP server (process) at [www.someschool.edu](http://www.someschool.edu) on port 80
  + HTTP server at host [www.someschool.edu](http://www.someschool.edu) waiting for TCP connection at port 80. “accepts” connection, notifying client
  + HTTP client sends HTTP request message (containing URL) into TCP connection socket. Message indicates that client wants object somedepartment/home.index
  + HTTP server receives request message, forms response message containing requested object, and sends message into its socket
  + HTTP server closes TCP connection
  + HTTP client receives response message containing HTML file, displays HTML. Parsing HTML file, finds 10 referenced jpeg objects
  + Above steps repeated for each of 10 jpeg objects

**Other Important Information:**

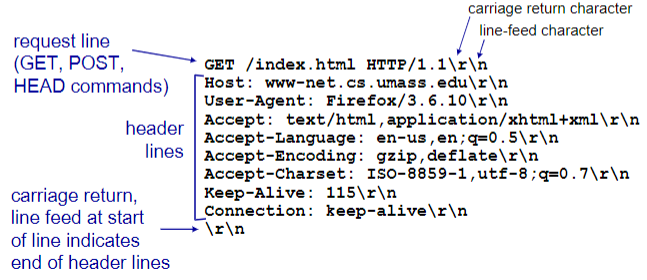
* BYTE \* 8 = BIT
* 1 KBPS = 10^3 BPS
* 1 MBPS = 10^6 BPS
* Binomial Distribution: C(n, x) \* px \* (1 – p)n – x, where p = probability of a success on an individual trial, n = number of trials, x = total number of “successes” (pass or fail, heads or tails, etc.), and C(n, x) = n!/((n – x)!(x!)).

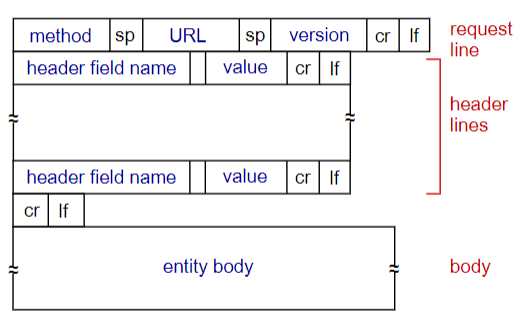
**+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++Test 2 Review**

**+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++**

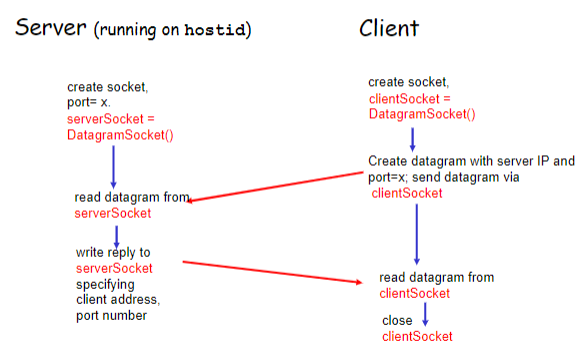
**CHP 2 – Application Layer:**

* Non-Persistent HTTP Issues: Requires 2 RTTs per object; OS overhead for each TCP connection; Browsers often open parallel TCP connections to fetch referenced objects; each request/response pair is sent over a separate TCP connection
* Persistent HTTP: Corresponding responses will be sent over the same TCP connection
  + Server leaves connection open after sending response
  + Subsequent HTTP messages between same client/server sent over open connection
  + Client sends requests as soon as it encounters a referenced object
  + As little as one RTT for all the referenced objects
* HTTP Request Message:

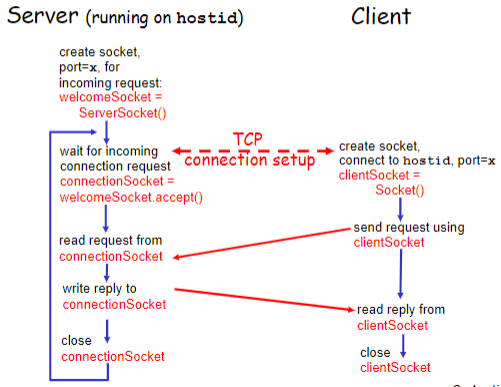




* + POST Method: Web page often includes form input; input is uploaded to server in entity body
  + URL Method: Uses GET method; input is uploaded in URL field of request line: [www.somesite.com/animalsearch?monkeys&banana](http://www.somesite.com/animalsearch?monkeys&banana)
  + Method Types:
    - HTTP/1.0: GET, POST, HEAD
      * HEAD: Asks server to leave requested object out of response
    - HTTP/1.1: GET, POST, HEAD, PUT, DELETE
      * PUT: Uploads file in entity body to path specified in URL field
      * DELETE: Deletes file specified in the URL field
  + HTTP Response Status Codes:
    - Status code appears in 1st line in server-to-client response message
    - Some sample codes:
      * 200 OK: Request succeeded, requested object later in this msg
      * 301 Moved Permanently: Requested object moved, new location specified later in this msg (Location: )
      * 400 Bad Request: Request msg not understood by server
      * 404 Not Found: Requested document not found on this server
      * 505 HTTP Version Not Supported
* User-Server State: Cookies
  + Four Components:
    - 1.) Cookie header line of HTTP response message
    - 2.) Cookie header line in next HTTP request message
    - 3.) Cookie file kept on user’s host, managed by user’s browser
    - 4.) Back-end database at Web site
  + Example: Susan always accesses the Internet from her PC, but this time she visits a specific e-commerce site for the first time
    - When initial HTTP request arrives at site, site creates a unique ID and an entry in the backend database for the ID
  + What Cookies Can Be Used For: Authorization, shopping carts, recommendations, and user session state (Web e-mail)
  + How To Keep “State”:
    - Protocol Endpoints: Maintain state at sender/receiver over multiple transactions
    - Cookies: HTTP messages carry state
  + Cookies & Privacy: Cookies permit sites to learn a lot about you; you may supply name and e-mail to sites
* Web Caches (Proxy Server):
  + Goal: Satisfy client request without involving origin server
    - User Sets Browser: Web accesses via cache
    - Browser sends all HTTP requests to cache
      * Object In Cache: Cache returns object
      * Else cache requests object from origin server, then returns object to client
  + Cache acts as both client and server - Server for original requesting client, client to origin server
  + Typically cache is installed by ISP (university, company, residential ISP)
  + Why Web Caching:
    - Reduce response time for client request
    - Reduce traffic on an institution’s access link
    - Internet dense with caches: enables “poor” content providers to effectively deliver content (so too does P2P file sharing)
  + Caching Example:
    - Assumptions:
      * Avg Object Size: 100K bits
      * Avg Request Rate From Browsers To Origin Servers: 15/sec
      * Avg Data Rate To Browsers: 1.50 Mbps
      * RTT From Institutional Router To Any Origin Server: 2 sec
      * Access Link Rate: 1.54 Mbps
    - Consequences:
      * LAN Utilization: 0.15%
      * Access Link Utilization: 99% (problem!)
      * Total Delay: Internet delay + access delay + LAN delay = 2 sec + minutes + usecs
    - If 1.54 Mbps access link increased to 154 Mbps, LAN utilization increases to 0.99% and there is an increased access link speed (which is not cheap!)
  + Caching Example: Install Local Cache
    - Calculating Access Link Utilization & Delay With Cache:
      * Suppose cache hit rate is 0.4 - 40% requests satisfied at cache, 60% requests satisfied at origin
      * Access Link Utilization - 60% of requests use access link
      * Data rate to browsers over access link = 0.6 \* 1.50 Mbps = .9 Mbps; Utilization = 0.9/1.54 = .58
      * Total Delay = 0.6 \* (delay from origin servers) + 0.4 \* (delay when satisfied at cache) = 0.6 (2.01) + 0.4 (~msecs) = ~1.2 secs; less than with 154 Mbps link (and cheaper too!)
* Conditional GET:
  + Goal: Don’t send object if cache has up-to-date cached version; results in no object transmission delay and lower link utilization
  + Cache: Specify date of cached copy in HTTP request – If-modified-since: <date>
  + Server: Response contains no object if cached copy is up-to-date – HTTP/1.0 304 Not Modified
* FTP (File Transfer Protocol):
  + Transfer file to/from remote host
  + Client/server model
    - Client: Side that initiates transfer (either to/from remote)
    - Server: Remote host
  + FTP: RFC 959
  + FTP Server: Port 21
  + FTP: Separate Control & Data Connections:
    - FTP client contacts FTP server at port 21, using TCP
    - Client authorized over control connection
    - Client browses remote directory, sends commands over control connection
    - When server receives file transfer command, server opens 2nd TCP data connection (for file) to client
    - After transferring one file, server closes data connection
    - Server opens another TCP data connection to transfer another file
    - Control connection: “out of band”
    - FTP server maintains “state”: current directory, earlier authentication
  + Sample Commands:
    - Sent as ASCII text over control channel
    - **USER username**
    - **PASS password**
    - **LIST**: return list of files in current directory
    - **RETR filename:** retrieves (gets) file
    - **STOR filename:** stores (puts) file onto remote host
  + Sample Return Codes:
    - Status code and phrase (as in HTTP)
    - **331 Username OK, password required**
    - **125 data connection already open; transfer starting**
    - **425 Can’t open data connection**
    - **452 Error writing file**
* Socket Programming:
  + Goal: Learn how to build client/server applications that communicate using sockets
  + Socket: Door between application process and end-end-transport protocol
  + Sockets exists between the application process and the start of the 4 link layers; application process is controlled by app developer and the 4 link layers are controlled by the OS
  + Two socket types for two transport services:
    - UDP: Unreliable datagram
    - TCP: Reliable, byte stream-oriented
  + Application Example:
    - 1.) Client reads a line of characters (data) from its keyboard and sends the data to the server
    - 2.) The server receives the data and converts characters to uppercase
    - 3.) The server sends the modified data to the client
    - 4.) The client receives the modified data and displays the line on its screen
  + Socket Programming With UDP:
    - No “connection” between client & server
      * No handshaking before sending data
      * Sender explicitly attaches IP destination address and port # to each packet
      * Receiver extracts sender IP address and port # from received packet
    - Transmitted data may be lost or received out-of-order
    - Application Viewpoint: UDP provides unreliable transfer of groups of bytes (“datagrams”) between client and server
  + Running Example:
    - Client: User types line of text, client program sends line to server
    - Server: Server receives line of text, capitalizes all the letters and sends modified line to client
    - Client: Receives line of text and displays it
  + Client/Server Socket Interaction (UDP):



* UDP Observations & Questions:
  + Both client and server use DatagramSocket
  + Destination IP and port are explicitly attached to segment
* 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 TCP socket, specifying IP address, 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 for server process to communicate with that particular client
      * Allows server to talk with multiple clients
      * Source port numbers used to distinguish clients
  + Application Viewpoint: TCP provides reliable, in-order byte-stream transfer (“pipe”) between client and server
  + Client/Server Socket Interaction (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)
* TCP Observations & Questions:
  + Server has two types of sockets – ServerSocket and Socket
  + When client knocks on serverSocket’s “door”, server creates connectionSocket and completes TCP connection.
  + Destination IP and port are not explicitly attached to segment
* Stream: A sequence of characters that flow into or out of a process
* Input Stream: Attached to some input source for the process, e.g., keyboard or socket
* Output Stream: Attached to an output source, e.g., monitor or socket

**C/C++ Socket Programming:**

* A Client-Server Transaction:
  + Most network applications are based on the client-server model:
    - A server process and one or more of client processes
    - Server manages some resource (ex: Database)
    - Server provides service by manipulating resource for clients
* The Socket Interface:
  + Socket: A descriptor that lets an application read/write from/to the network
    - Similar abstraction for network I/O as file I/O
    - Clients and servers communicate by reading/writing from/to socket descriptors
* Socket API:
  + Originated with the 4.2 BSD system released in 1983
  + Sockets: A way to speak to other programs using UNIX file descriptors.
  + File Descriptor: An integer associated with an open file. This can be a network connection
  + Many Types of Sockets: DARPA Internet addresses (Internet Sockets), Unix Sockets, X.25 Sockets, etc.
  + Types of Internet Sockets:
    - SOCK\_STREAM uses TCP (Transmission Control Protocol); connection oriented and reliable
    - SOCK\_DGRAM uses UDP (User Datagram Protocol); connectionless and unreliable
* Internet Connections:
  + Clients and servers communicate by sending streams of bytes over connections.
  + Socket address is identified by an IPaddress:port pair
  + A port is a 16-bit unsigned integer identifying a process (ephemeral, not physical port)
  + Ports below 1024 are reserved for well-known services (HTTP:80, FTP:21, SSH:22, Telnet:23)
  + Ports 1024-49151 are registered ports managed by IANA
  + Dynamic/Private ports are those from 49152-65535
* Key Data Structures – IP Address:
  + 32-bit IP addresses are stored in an IP address struct in <netinet/in.h>
    - IP addresses are always stored in memory in network byte order (big-endian byte order)
    - True in general for any integer transferred in a packet header from one machine to another (e.g., the port number used to identify an Internet connection)
  + Handy Network Byte-Order Conversion Functions:
    - htonl: Convert ling int from host to network byte order
    - htons: Convert short int from host to network byte order
    - ntohl: Convert long int from network to host byte order
    - ntohs: Convert short int from network to host byte order
* Byte-Ordering:
  + Consider a hexadecimal 4A3B2C1D at address 100. The bytes could be stored within the address range 100 through 103 in the following order:
  + Big-endian (“big end first”)
    - The most significant byte (msb, 4A) is stored at the lowest address
    - Used by Motorola/SPARC and network devices
  + Little-endian (“little end first”)
    - The least significant byte (lsb, 1D) is stored at the lowest address
    - Used by Intel x86, DEC VAX
  + Endianness does not denote what the value ends with when stored in memory, but rather which end it begins with
* Difference Between TCP & UDP:
  + TCP: Connection-oriented
    - First, connection is established between two ends
    - Then, they communicate through the connection
  + UDP: Connectionless (datagram)
    - Whenever one sends a message to the other, the destination address is included (no connection)
    - Packet could be lost or delivered out of order
* Transmission Control Protocol (TCP):
  + Allow networked computers to reliable transfer of byte stream
  + Connection-oriented provides error recovery and flow control
  + Data received in correct order

**CHP 3 – Transport Layer:**

* Transport Services & Protocols:
  + Provide logical communication between app processes running on different hosts
  + Transport protocols run in end systems
    - Send Side: Breaks app messages into segments, passes to network layer
    - Receiver Side: Reassembles segments into messages, passes to app layer
  + More than one transport protocol available to apps
* Internet Transport-Layer Protocols:
  + Reliable, in-order delivery (TCP) – congestion control, flow control, connection setup
  + Unreliable, unordered delivery: UDP – no-frills extension of “best-effort” IP
  + Services not available: delay guarantees, bandwidth guarantees
* Multiplexing at Sender: Handle data from multiple sockets, add transport header (later used for demultiplexing)
* Demultiplexing at Receiver: Use header info to deliver received segments to correct socket
* How Demultiplexing Works:
  + Host receives IP datagrams
    - Each datagram has source IP address and destination IP address
    - Each datagram carries one transport-layer segment
    - Each segment has source and destination port number
  + Host uses IP addresses & port numbers to direct segment to appropriate socket
* Connection-Oriented Demux (TCP):
  + TCP socket identified by 4-tuple: source IP address, source port number, destination IP address, and destination port number
  + Receiver uses all four values to direct segment to appropriate socket
  + Server host may support many simultaneous TCP sockets – each socket is identified by its own 4-tuple
  + Web servers have different sockets for each connecting client – non-persistent HTTP will have different socket for each request
* UDP – User Datagram Protocol:
  + “Bare bones” Internet transport protocol
  + “Best effort” service, UDP segments may be lost or delivered out-of-order to an app
  + Connectionless: No handshaking between UDP sender and receiver
    - Each UDP segment handled independently of others
  + Usage: Streaming multimedia apps (loss tolerant, rate sensitive), DNS, & SNMP
  + Add reliability at application layer
  + Why is there a UDP?
    - No connection establishment (which can add delay)
    - No connection state at sender and receiver (makes it simple!)
    - Small header size
    - No congestion control – UDP can blast away as fast as desired
  + Checksum: Addition (one’s complement sum) of segment contents
    - Sender:
      * Treat segment contents, including header fields, as sequence of 16-bit integers
      * Sender puts checksum value into UDP checksum field
    - Receiver:
      * Compute checksum of received segment
      * Check if computed checksum equals checksum field value:
        + NO – error detected
        + YES – no errors detected (but maybe errors nonetheless)
* Reliable Data Transfer (RDT):
  + rdt\_send(): Called from above (e.g., by app). Passed data to deliver to receiver upper layer
  + udt\_send(): Called by rdt to transfer packet over unreliable channel to receiver
  + rdt\_rcv(): Called when packet arrives on rcv-side of channel
  + deliver\_data(): Called by rdt to deliver data to upper
  + 1.0:
    - Underlying channel is perfectly reliable – no bit errors or loss of packets
    - Sender sends data into underlying channel, receiver reads data from underlying channel
  + 2.0:
    - Underlying channel may flip bits in packet – uses checksum to detect bit errors
    - Acknowledgements (ACKs): Receiver explicitly tells sender that the packet was received OK
    - Negative Acknowledgements (NAKs): Receiver explicitly tells sender that the packet had errors
    - Sender retransmits packet on receipt of NAK
    - New Mechanisms: Error detection and control messages (ACK, NAK) from receiver to sender
    - What happens if ACK/NAK is corrupted?
      * Sender doesn’t know what happened at receiver!
      * Can’t just retransmit: possible duplicate
    - Handling Duplicates:
      * Sender retransmits current packet if ACK/NAK corrupted
      * Sender adds sequence number to each packet
      * Receiver discards (doesn’t deliver up) duplicate packet
    - Stop & Wait: Sender sends one packet then waits for receiver response
  + 2.1:
    - Sender:
      * Sequence number has been added to packet
      * Two sequence numbers (0, 1) will suffice
      * Must check if received ACK/NAK corrupted
      * Twice as many states – state must “remember” whether “expected” packet should have sequence number of 0 or 1
    - Receiver:
      * Must check if received packet is duplicate – state indicates whether 0 or 1 is expected packet sequence number
      * Note: Receiver can *not* know if its last ACK/NAK received OK at sender
  + 2.2:
    - Same functionality as rdt2.1, using ACKs only
    - Instead of NAK, receiver sends ACK for last packet received OK
      * Receiver must explicitly include sequence number of packet being ACKed
    - Duplicate ACK at sender results in same action as NAK: retransmit current paclet
  + 3.0:
    - New Assumption: Underlying channel can also lose packets (data, ACKs)
      * Checksum, sequence numbers, ACKs, and retransmissions will be of help… but not enough
    - Approach: Sender waits “reasonable” amount of time for ACK
      * Retransmits if no ACK received in this time
      * If packet (or ACK) just delayed (not lost):
        + Retransmission will be duplicate, but sequence numbers already handle this
        + Receiver must specify sequence number of packet being ACKed
      * Requires countdown timer
    - This version is correct, but the performance sucks
    - Utilization: Fraction of time the sender is busy sending
    - If RTT = 30 msec, 1KB packet is sent every 30 msec, and we have a 1 Gbps link: 33kB/sec throughput over 1 Gbps link
    - Network protocol limits use of physical resources!
* Pipelined Protocols:
  + Pipelining: Sender allows multiple, “in-flight”, yet-to-be-acknowledged packets
    - Range of sequence numbers must be increased
    - Buffering at sender and/or receiver
  + Two Generic Forms: Go-Back-N & Selective Repeat
  + Go-Back-N:
    - Sender can have up to N unACKed packets in pipeline
    - Receiver only sends cumulative ACK – doesn’t ACK packet if there’s a gap
    - Sender has timer for oldest unACKed packet – when timer expires, retransmit *all* unACKed packets
  + Selective Repeat:
    - Sender can have up to N unACKed packets in pipeline
    - Receiver sends individual ACK for each packet
    - Sender maintains timer for each unACKed packet – when timer expires, retransmit only that unACKed packet