Chapter1: Network: edge- hosts, access network – wired/wireless link, core – network of network(routers); HFC: hybrid fiber coax; frequency division multiplexing: different channels transmitted in different frequency bands; **Packet Switching:** store and forward - entire packet must arrive at router before it can be transmitted on next link; **routing** – determines source dest route taken by packets accd2 routing alg; f**orwarding**: move packets from router's input to appr router output; high bandwidth utilization, easy to implement, less reliable, no bandwidth guarantee; **Circuit Switching:** FDM vs TDM; bandwidth guarantee, highly reliable, low bandwidth utilization, requires proprietary mux**; Delays**: dproc: nodal processing - check bit errors, determine output link, typically < msec; dqueue: queueing delay - time waiting at output link - depends on congestion level of router; dtrans: transmission delay: L: packet length (bits) R: link bandwidth (bps) = L/R | dprop: propagation delay: d: length of physical link s: propagation speed (~2x108 m/sec) dprop = d/s; traffic intensity = La/R; a: average packet arrival rate – La/R ~ 0: small delay, ->1 big delay, >1 infinite delay | **traceroute**: test delay, send 3 pkts to router i, take round trip time; **throughput**: rate (bits/time unit) transferred between sender and receiver; **bottleneck:** link on end-end path that constrains end-end throughput; **Layering:** app – support network app (SMTP, HTTP, FTP); transport – process to process data transfer (UDP, TCP); network – routing of datagrams from source to dest (IP, routing protocols); link – data transfer between neighbor network elements (Ethernet, 802.111 (WiFi), PPP); physical: bits "on the wire"; **ISO/OSI reference model**: two more layers between app and transport, must be implemented in apps if needed, can be combined with app layer; presentation: allow apps to interpret meaning of data (eg. Encryption, compression, machine-specific conventions); session: synchronization, checkpointing, recovery of data exchange; **Encapsulation**: each layer generates specific type of header and attach it to the msg being transferred; header contains layer specific info only corresponding layers can decrypt; app: message; transport: segment (packet); network: datagram; link: frame; headers can be changed while transferred between diff layers; transport layer info never changed; the dest end user decrypt the packet encapsulated w/ layers of headers; **Chapter2: client-server: S**: always on host, permanent public IP, have datacenter; **C**: cumm w/ server, may be intermittently connected, dynamic IP addr, does not comm directly w/ other c; c initiates comm w/ server; **P2P**: no always on server, end sys comm w/ each other directly, slef scalability: both new cap and demand by new peers; peers intermittently connected, change IP addr; two processes in same host comm w/ inter-process comm defined by OS, in diff host by exchanging msg, app w/ p2p architecture have client process and server process; **Sockets**: process comm w/ socket, sendin process shoves msg out socket, relies on transport infras on other side to deliver msg to socket at rcving process; **Addressin process**: need **identifier**: 32 IP addr & port# (HTTP 80, mail 25)**; HTTP:** hypertext transfer protocol: web page consists of objects (addressed by URL): HTML file, JEPG; **stateless**: server maintains no info about past client requests; **communication process**: client initiate TCP connection to server, port 80 -> server accepts TCP connection -> HTTP msg exchange between browser(HTTP client) and web server (HTTP server) -> TCP closed; **RTT**: round trip time, time for a small packet to travel from client to server and back; **non-persistent HTTP**: at most one object sent over TCP connection, TCP then close; 2RTT + file trans time per object; **persistent:** multiple object sent over single TCP connection; 1 RTT for all referenced objects; **HTTP msg:** request & response; **request msg**: ASCII; format: request line (GET/POST/PUT/DELETE/HEAD) & URL & version + header lines + body; 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; **response msg**: status line(status code, phrase) + header lines + data (eg. Requested HTML file); **HTML response status code:** 200 OK: req succeeded, reqd object later in msg; 301 Moved Permanently: new location specified later in this msg; 400 Bad Request: req msg not understood by server; 404 Not Found: reqd document not found on this server; 505 HTTP Version Not Supported; **User-server state: cookies:** keeping states, **four components**: cookie header line of HTTP response msg, cookie header line in next HTTP req msg, cookie file kept on user's host and managed by user's browser, backend database at website; **client:** usual HTTP msg -> server: HTTP response + set cookies: xx (create data entry for cookie # at backend db)-> client: HTTP msg + cookies: xx -> server: response (access cookie specification from backend db); **cookies in use:** authorization, shopping carts, recommendations, user session state (Web e-mail); **Web caches (proxy server):** satisfy client request without involving origin server; browser sends all HTTP requests to cache, cache returns if in cache, else cache request object from origin server and return to client**; Conditional GET:** don't send if up-to-date; HTTP req msg if-modified-since<date> -> 304 not modified (nothing attached)/200 OK (data)**; DNS** domain name system: distributed database implemented in hierarchy of many name servers; app-layer protocol – hosts, name servers communicate to resolve names (addr/name translation); **DNS services**: hostname to IP address translation, host aliasing (canonical, alias names), mail server aliasing, load distribution (replicated web servers: many IP address correspond to one name); **why not centralize** DNS: single point of failure, traffic volume, distant centralized database, maintenance, poor scalability; **DNS structure:** root DNS -> top-level domain (TLD) servers: com, org, net, edu & country domains: uk, fr -> authoritative DNS servers (organizations own DNS servers, providing authoritative hostname to IP mappings for org's named hosts, can be maintained by org or service provider. eg: amazon.com); **Local DNS name server**: (aka default name server) not strictly belong to hierarchy, each ISP (resid/company/univ) has one; when host makes DNS query, query is sent to its local DNS server; LDNSNS has local cache of recent name-to-addr translation pairs (can be outofdate), acts like proxy; **DNS name resolve**: iterated: local DNS server in charge of querying root/TLD/authoritative DNS server iteratively if it doesn't know the addr; recursive: recurse up the DNS tree to find the answer, put burden on contacted name server, especially upper levels of hierarchy; **DNS caching:** caches mapping once learn new mapping, cache entries timeout after some time (TTL), TLD servers cached in local name servers; **DNS records:** distributed database storing resource records **(RR)**: format: name, value, type, ttl; type A: name = hostname, value = IP address; type NS: name = domain(foo.com), value=hostname of authoritative NS for this domain; type CNAME: name = alias name for canonical (real) name, value = canonical name; type MX: value = name of mail server associated with name; **DNS msg:** query and reply msg same format: msg header (identification(16bits), flags (query/reply, recursion desired/available, reply authoritative?)), question (name, type fields for query), answers (RRs in response to query), authority (records for authoritative server), additional info**; Insert records to DNS:** register name (gentsk.com) at DNS registrar, provide names, IP addresses of authoritative name server, registrar inserts two RRs into .com TLD server; auth sever w/ type A (www.gentsk.com), type MX for mail server (gentsk.com); **DNS Attack**: DDoS attacks: bombard root & TLD servers with traffic; redirect attacks: intercept queries, DNS poisoning (send fake reply to DNS server and cached); exploit DNS for DDoS: send queries w/ spoofed source addr, requires amplification; **Chapter3: Transport protocol:** logical comm between app processes on diff hosts, run in end systems, sender break app msg into segments & pass to network layer, rcvr reassembles segments into msg & pass to app layer; **Mux**: gather data chunks, add headers create segments, pass to network layer; **Demux**: receive segments, read header, send to correct socket by using IP addr and port numbers; **Connectionless demux**: (UDP) specify dest IP addr & port # in segment, direct UDP segment to socket with that port #, IP datagram w/ same dest port # but diff source IP addr and/or source port # directed to same socket at dest; **Connection-oriented demux**: (TCP) socket identified by 4-tuple (source IP, source port #, dest IP, dest port #); rcvr uses all four values to direct segment to appropriate socket; web server has diff sockets for each connecting client, non-pers HTTP has diff socket for each request; **UDP:** no connection/handshaking between client and server; sender attaches IP dest addr and port# to each packet; receiver extracts sender IP addr and port# from rcvd packet; transfer unreliable datagram; **Why UDP**: no connection establishment, simple, small header size, no congestion control: fast, used for multimedia apps, DNS, SNMP; **UDP segment header:** source port#, dest port#, length (in bytes of UDP segment, including header), checksum; **UDP checksum:** detect errors in transmitted segment; sender add segment header fields and contents up as sequence of 16bit ints, do 1s complement, store in checksum fields; receiver add everything including checksum up, shud be all 1, if not then error; **RDT1.0**: perfectly reliable assumption, no bit errors/loss of packets; **RDT2.0:** can have bit errors, use checksum, applies ACK/NAK; flaw: sender unaware if ACK/NAK corrupt, cant just retransmit duplicates; **RDT2.1**: if ACK, NAK corrupted: add sequence # (0, 1). Resend current packet if NAK or corrupt ACK/NAK, receiver discard duplicate; **RDT2.2**: No NAK, receiver sends ACK for last pkt received ok (receiver include # of pkt being ACKed), duplicate ACK = NAK at sender: resend; **RDT3.0**: channel can lose data/ACK; sender waits for a while, retransmit if no ACK, seq# handles duplicate, receiver include seq# for pkt being ACKed, sender needs countdown timer; **rdt3.0 performance**: Dtrans = L packet length/R link cap; U: sender utilization, fraction of time sender busy sending, U = (L/R)/(RTT +L/R); limit utilization for long RTT; **Pipelined protocol:** sender allows multiple inflight (to be acked) pkts; range of seq# shud be increased; need buffer at sender rcvr; N: windowsize; k:

seq# bits; **GBN:** sender: has up to N unacked pkts in pipeline; has timer for oldest unacked pkt, when timer expires retransmit all unacked packets; rcvr: sends cumulative acks, only ack correct rcvd pkt w/ highest inorder seq#, remember only expectedseq#; discard outoforder pkt, reack highest correct pkt; **N < 2^k – 1**; **Selective Repeat:** sender: has up to N unacked pkts; has timer for each unacked pkt, when expires retransmit only that pkt; if next avail seq# in window, send pkt; if receive smallest unacked pkt n, advance window base to next unacked seq# (window sliding); rcvr: sends indvid ack for each pkt; buffer outoforder pkts; also window slide to next unrcvd pkt; if rcv pkt w/ seq# in prev max window size, ack that pkt; **N < 2^k/2; TCP:** reliable inorder byte stream, connection-oriented transport**;** flow control**,** congestion control, full duplex data; no timing/min thruput/security; **TCP segment structure**: source & dest port#; seq#, byte stream # of 1st byte in seg's data; ack#, seq# of next byte expected from other side, TCP use cumlative ACK; receive window: # bytes rcvr willing to accept; header length field: length of TCP header; **TCP RTT**: EstimatedRTT = (1- a)*EstimatedRTT + a*SampleRTT (a typically = 0.125); **TCP timeout interval**: EstimatedRTT plus safety margin, large variation in EstimatedRTT -> larger safety margin; DevRTT = (1-b)*DevRTT + b*|SampleRTT-EstimatedRTT| (b typically = 0.25); TimeoutInterval = EstimatedRTT + 4*DevRTT; **TCP reliable data tranfer**: create rdt service on IP's unreliable service; pipelined segs, cuml acks, single timer; retrans triggered by timeout and dup acks; sender: rcvd data from app -> make segm with seq# (byte stream # of first data) -> start timer -> if timeout, resend, restart timer; if ACK, update known ACKed, start timer if exist unacked seg; rcvr: if inorder seg w/ expected seq#, and all gud, delay ACK, if no next seg then send ACK; if inorder seg w/ expected seq# and one other seg ACK pendin, then send single ack immediately for both; if higher than expected seq#, send dup ACK for next expected seq#; if seg that partially fills gap, send ACK immediately; **TCP fast retransmit**: (timeout often long, detect loss through dup ACKs) if sender receives 3 ACKs for same data, resend unacked segment with smallest seq# right away; **TCP flow control:** rcvr controls sender, so sender wont overflow receivers buffer by transmitting too much; if rcvr buffer(set via socket option, typical 4096) space available, including rwnd value in TCP header sent to sender -> sender limits # of unacked (in transit)data to rwnd value; **TCP Connection Management:** handshake before exchange data: estab connection (agree on parameter); 2-way handshake problem: delay -> resend req_connectiont -> server open another one without client -> retransmitted data to the half open connection; 3-way: client send SYNbit=1, Seq=x -> server send SYNbit=1, Seq=y ACKbit=1; ACKnum=x+1 -> client – knows server live: ACKbit=1, ACKnum=y+1(may have data) -> server knows client live; Closing**:** Client: FINbit=1, seq=x -> Server: ACKbit=1; ACKnum=x+1 ->(server can still send data)-> Server: FINbit=1, seq=y (server cant send data anymore) -> Client: ACKbit=1; ACKnum=y+1, wait for 2*max segm lifetime, close; **Principles of congestion control:** too many sources sending too much data too fast for network to handle -> lost package, long delay; **Cost of congestion**: long delays, unneeded retrans hurting gudput (data not retrans/total transd data), fairness(farther host lost packet); **TCP congestion control:** sender increases transmission rate (window size), probing for usable bandwidth, until loss occurs; **additive increase**: increase cwnd by 1 MSS every RTT until loss detected; **multiplicative decrease**: cut cwnd in half after loss; **sender limits transmission**: LastByteSent- LastByteAcked <= cwnd; **TCP sending rate**: send cwnd bytes, wait RTT for ACK, then inc or dec: rate = cwnd/RTT (byte/sec); **TCP Slow Start**: connection begin with cwnd = 1MSS, increase rate exponentially for every 1RTT (every ACK rcvd) until reach threshold (half of prev lost), change to linear until next lost, decr to 1MSS again, slow start, repeat; **Loss indicators**: timeout: more severe indicator since rcvr can not recv at all; 3 dup ACK: stable comm, just a few gaps, congestion not too bad; **TCP reno**: performs fast recovery on 3dup (linear incr from half of prev lost), and slow start on timeout; **TCP Tahoe**: slow start for both cases; **TCP thruput:** W: window size (measured in bytes) where loss occurs, avg cwnd = avg TCP thruput = 3W/4RTT bytes/sec; **TCP fairness:** linear incr w/ 45` slope; multiplicative decr decrs thruput proportionally, fair with parallel TCP connections; **Chapter4: Network layer:** routing protocol <-> forwarding table, IP, ICMP(error reporting, router) protocol; **Data plane**: local per-router func, determines how datagram arrive on router input port is forwarded to router output port, use forwardin func; **Control plane:** network-wide logic, determine how datagram is routed among routers along end2end path from source host to dest host; traditional routin alg(in routers) & sw-defined networkin(SDN, in remote server); **IPv4 datagram format:** version number(of datagram); header length; datagram length; identifier(same frag same val), flags(last: 0, otherwise 1), fragmentation offset(prev total leng / 8, determine reassemble seq)(for fragmentation & reassembly); time-to-live (decre at each router); header checksum; source and dest IP addresses; **Overhead**: TCP = IP = 20bytes; **MTU** (max transfer size): large IP datagram fragmented within net to smaller datagram, reassembled at final destination only; **IP address:** 32-bit identifier for host, router interface; **interface**: connection between host/router and physical link, routers typically have multiple interfaces, host typically has one or two interfaces, IP addresses associated with each interface; **Subnet:** device interfaces with same subnet part of IP address can physically reach each other without intervening router; subnet high order bits IP address, host low order bits, leng accd2 subnet mask; detach each interface from its host/router, each isolated network is called a subnet; **IP addressing: CIDR,** Classless Inter Domain Routing: subnet portion of address (length vary), format: a.b.c.d/x, x= # of bits for subnet; host's IP: hard-coded by system admin in a file or DHCP; network IP: gets allocated portion of its provider ISPs address space; **DHCP(app layer):** Dynamic Host Configuration Protocol(configured host IP automaticly): allow host(subnet) to dynamically obtain its IP address from network server when upon connecting (also allow reuse of address); **DHCP client join network**: DHCP discover- find address -> server: DHCP offer – offer IP -> client: DHCP request- take IP -> Server: DHCP ACK; **DHCP other func**: address of first-hop router for client, name and IP address of DNS sever, network mask (indicating network versus host portion of address); **NAT**: routerish, network address translation – local network has one NAT IP address to the world, can chang addr in local network w/o notifyin outside world, change ISP w/o changing addr of devices in local network, local devices not addressable by outside; **NAT practice**: replace source IP for outgoing pkt, remember every IP within, replace dest for incoming; use port# (16bits) on outside to represent diff local IPs; **IPv6:** exist cuz 32-bit address is running out; **Datagram format**: fixed-length 40byte header, no fragmentation; priority (priority in datagram flow), flow label (identify same flow), next header (upper layer protocol for data); **IPv6 VS IPv4**: checksum gone, option allowed outside header, ICMPv6: new version of ICMP – additional msg type); **IPv6 to IPv4**: **tunneling**: IPv6 datagram carried as payload in IPv4 datagram with v4 headers among IPv4 routers