Chpt1: hybrid fiber coax & fiber optic cable; **Packet Switching: store and forward** - entire packet must arrive at router before it can be transmitted on next link, high bandwidth utilization, easy to implement, less reliable, no bandwidth guarantee; **Circuit Switching:** FDM vs TDM; dedicated rsrc, 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; **thruput**: rate (bits/time unit) transferred between sender and receiver; **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 model**: two more layers between app and transport: presentation & session; **Encapsulation**: app: **message**; transport: **segment**; network: **datagram**; link: **frame**;

Chpt2: **ser**: always on host, permanent public IP, have datacenter; **cli**: 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, self scalability: both new cap and demand by new peers; peers intermittently connected, change IP addr; **Sockets**: process comm w/ socket; **HTTP:** hypertext transfer protocol; **stateless**: server maintains no info about past client requests; **non-persistent HTTP**: 2RTT + file trans time per object; **persistent:** 1 RTT for all referenced objects; **cookies:** keep states, 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); **Web caches (proxy server):** satisfy client request w/o involving origin server; **Conditional GET:** don't send if up-to-date**; DNS** domain name system (app layer): distributed database implemented in hierarchy, hosts, name servers communicate to resolve names (addr/ name translation), **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); **DNS structure:** root DNS -> top-level domain (TLD) servers: com, org, edu etc. -> authoritative DNS servers (organizations own DNS servers); **Local DNS name server**: (default name server) not belong to hierarchy, each ISP (resid/company/univ) has one; when host makes DNS query, query is sent to its local DNS server; **DNS iterated**: local DNS server in charge of querying root/TLD/authoritative DNS server iteratively; **recursive**: recurse up the DNS tree to find the answer, put burden on contacted upper name server; **DNS caching:** caches mapping once learn new mapping, TLD cached in local name servers; **DNS records (RR)**: 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; **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;

Chpt3: **Transport:** logical comm btw 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 skt:** 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, no connection estab, simple, small header size, fast, used for multimedia apps, DNS, SNMP; **checksum:** sender add segment header fields and contents up as sequence of 16bit, do 1s complement, store in checksum fields; receiver add everything including checksum up, shud be all 1; **TCP skt**: client must contact server, server must create skt to welcome client, client contact server and set up TCP con, then send/read msg; **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; **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; **sender utilization U = (L/R)/(RTT +L/R)**; **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 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 rdt**: 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; **Connection mgmt:** handshake before exchange data: estab connection (agree on parameter); 2-way handshake: delay -> resend req_connectiont -> server open another one without client -> retransmitted data to the half open connection; **3-way**: see **EXAMPLE**; **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 ctrl: additive increase**: incr cwnd by 1 MSS every RTT until loss detected; **multiplicative decrease**: cut cwnd in half after loss; send cwnd bytes, wait RTT for ACK, then inc or dec: rate = cwnd/RTT (byte/sec); **Slow Start**: begin w/ cwnd = 1MSS, incr 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; **Loss indicators**: timeout&3 dup ACK; **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;

Chpt4: **Network layer**: in every host&router; **forwarding**: move pkt from router's input to appropriate router output; **routing**: determine route taken by pkt from src~dest; **data plane**: **local, per-router** func, determine forwarding func; **control plane**: ntwk-wide logic, determine datagram routing, 2 approaches: trad routin alg (in routers) & SDN(software-defined-ntwk, in remote server); **router** architecture: routing processor (management control plane) <-> input ports + swichin fabric + output ports (forwardin data plane); router **INPUT** port: line termination(phys)->link layer prot(data link)->lookup, fwdin, queuein(decentz **swichin**: use header fields, lookup output port w/ fwdin table in input port memory, 2types); **destn-based fwdin**: fwd only based on dest IP addr, eg: longest

prefix matchin; **generalized fwdin**: use OpenFlow prot, fwd based on any header fields; **switchin fabric:** transfer pkt from input buffer to appr output buffer, 3 types: s via **Memory** (pkt copied to sys), **Bus**(via shared bus), **interconnection ntwk** (overcome bandwidth lim, allow parallel fwd if pkt fwd to diff output port); **HOL blocking**: head of the line blockin, in input port, queued datagram at front of queue prevents following ones in queue from movin fwd, queuein delay and loss due to input buffer overflow; **OUTPUT port**: also buffer, can have pkt lost, **output port contention** (one pkt switched to one port at one time), schedule datagrams to be trans; **scheduling**: choose next pkt to send on link, diff types: **FIFO**, **priority** (send highest priority), **RR** (round robin, cyclically scan class queue and send 1 pkt), **WFQ** (weighted fair queuein, generalized RR where each class weighted amount instead of just 1);**IP fragmentation**: large IP datagram fraged, only reassemble at final dest, use **(ID, fragflag=1(pre)/0(last), offset=predata/8)** fields to determine reassembly, IP overhead=20; IP addr: 32bit, **interface**: conn btw host/router and phys link (router can have mult interfaces), **1 IP per interface, switch need none**; **Subnet**: high order bit in IP addr, use **subnet mask** (/x) to determine length; **CIDR: addr format = a.b.c.d/x**; ; **DHCP(app layer):** Dynamic Host Configuration Protocol in subnet, allow host to dynamically obtain its IP addr from ntwk server when connect, allow reuse of address, **STEP**: DHCP discover- find address -> server: DHCP offer – offer IP -> client: DHCP request- take IP -> Server: DHCP ACK; **DHCP other func**: addr of 1st hop router, name&IP addr of DNS sever, network mask of subnet; **NAT**: routerish, network addr 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; **IPv6:** exist cuz 32-bit addr run out, **format**= fixed-length 40byte header, no frag, flow label (identify same flow), next header (upper layer protocol for data); **IPv6 VS IPv4**: checksum gone, option allowed outside header; from v4 to v6: **tunneling**, wrap IPv4 header outside entire IPv6 pkt**; Generalized fwdin & SDN**: each router has flowtable from logically centralized routing controller, use **OpenFlow** prot to achieve and communicate (set of header fields to det how to match, counters, actions);

Chpt5: **routing prot:** det good path from send~rcv host; Link State (Dijkstra): global, link cost known to all nodes thru link state broadcast, from single src to all other nodes, O(n^2) comp, oscillation (rerouting due to traffic), can advertise wrong link cost; Distance Vector (BellmanFord): dx(y)=min_v{c(x,v)+dv(y)}, distributed, all nodes kno only self DV and neighbors states, dynamic programming, update to neighbors only when detect self DV change, update self when local link cost change/DV update msg from nbr, can advertise wrong path cost; Count to inf: link cost incr -> bad news slow; poisoned reverse: Z tells Y its dis to X = inf if Z routes thru Y to X, can't solve completely; AS(autonomous system): intraAS (in same AS, RIP w/ DV, OSPF w/ LS, IGRP) vs interAS(among diff ASes), all routers in same AS use same prot, in diff AS can have diff prot, gateway routers at edge of AS, peer link to other AS, gateway perform both inter & intra routing; OSPF: open shortest path first, public avail, LS so topology map at each node, only kno direction to nets in other areas, router floods OSPF LS ads to all other routers in entire AS, =ISIS, security, mult same cost path allowed, hierarchical OSPF in large domain, carry OSPF msg over IP; Hierarchical OSPF: boundary(connect other AS) – back bone (run OSPF routing limited to backbone) – area border(sum distances to nets in own area, ad to other area border router) (backbone)| (local area, LS ads only in here) – internal router; BGP (border gateway prot): support reachability information, eBPG(gateway obtain subnet reachability from nbr AS) vs iBGP(prop info to all internal AS router), support policy, exchange over TCP connection, (AS-PATH, NEXT-HOP), prefer shortest AS-PATH and closest next-hop router(within AS); BGP and OSPF: once router learn about dest via iBGP, set up forwardin table entry w/ OSPF intra-domain routing (eg: dest=X, interface=1);

Chapt6: nodes: host&router; link: comm channels connect adj nodes along comm path; diff link can diff prot; **MAC addr**: in frame header, identify src/dest; **link layer service**: framing, link access, rd between adj nodes(maynot have), flow ctrl, error det&corr, full-dup, half-duplex: nodes at both ends of link can transmit (not at same time); **NIC**: network interface card, aka adaptor, **implements link layer** (eg: Ethernet card 802.11 card), **attaches to host's system bus**; **EDC**: error detection and correction bit, not 100% reliable, larger better; **CRC**: cyclic redundancy check**, D, G(r+1), R(r) = Dx^r/G, F= x^rD+R, F'=F+E, F'%G!=0: error!**; single bit always detect: **x^j%G!=0**; if x+1 factor of G, then all odd number bit errors detected**: assume E odd = (x+1)T, let x=1, then E(x)=E(1)=1 since odd number of terms, and x+1=1+1=0, then (x+1)T=(1+1)T=0!=1, therefore E cannot be factor of x+1, error always detected**; CRC12=12+11+3+2+1+0, CRC16=16+15+2+1, CRCCCITT=16+12+5+1, 16&CCITT catch(single/double error, odd number of bit, burst <=16, 99.997% 17&18 bit; **2 types of links**: point2point vs broadcast (shared wire or medium, eg: upstream HFC, 802.11 wireless LAN); **multiple access protocol**: distrb alg determines how nodes share channel, the comm about channel also use channel itself; **MAC protocols**: channel partitioning (**TDMA vs FDMA**), random access (slotted/pure ALOHA, CSMA(/CD)), taking turns; **slotted ALOHA**: all frames same size, equal time slots, nodes transmit at beginning of slot, synch, >=2 nodes in slot: all detect collision, if no collision->transmit in next slot/else->retransmit with prob P until success, **p(1succ)=p(1-p)^(N-1), p(any succ)=N*p(1succ)**, max eff=.37; **pure ALOHA**: simpler, no synch, transmit frames immediately once arvs, p(1succ)=p(1-p)^(N-1)(1-p)^(N-1), max eff=.18; **CSMA**: carrier sense multiple access, listen before trans, if sensed idle->trans entire frame/else->defer trans, prop delay still cause collision; **CSMA/CD**: abort colliding trans to reduce channel wastage, ez detect collision in wired LAN: measure signal str, compare tsmtd rcvd signals, tugh in wireless LAN: rcvd signal str overwhelmed by local trms str; ethnet CSMA/CD alg: once abort, NIC enters binary backoff: after Mth colls, **choose K at random from [0,2^(M-1)],** wait **K*512 bit** times, then keep monitoring channel; **polling**: master node invites slaves (dumb devices), cons: polling overhead, latency, single point master fails; **token passing**: pass token msg seqly, cons: token overhead, latency, single point token fails; **MAC (LAN/physical/ethnet) addr**: used locally to get frame from one interface to another physically connected interface (same network in IP-addrsing sense), 48 bits (eg: 1A-2F-BB-76-09-AD), each adaptor/LAN has unique LAN addr, admined by IEEE, better portability->move LAN card from one LAN to another; **ARP**: addr resoluting prot, plug&play, each IP node on LAN has ARP table <IP addr, MAC addr, TTL(20min)>, **in SAME LAN**-> A broadcas ARP query pkt w/ B's IP->only B rcv&rply to A w/ B's MAC addr->A saves into ARP table until timeout; **DIFF LAN routing**: A kno 1st hop router IP&MAC(DHCP&ARP), A: destIP=B, destMAC=RL, R: srcIP=A, srcMAC=RR, destIP=B, destMAC=B; **Ethnet topology**: bus(all nodes same collision domain) vs star(switch center, no collision w/ sep Eth prot); **Ethnet**: connectionless(no handshaking), unreliable(rcvin NIC no ack/nack), use unslotted CSMA/CD w/ binary backoff as MAC prot, diff phys layer media: fiber/cable; **ethnet switch**: store&forward frames, use CSMA/CD to access link, **transparent**, **self-learnin** (no config needed), buffer pkt, **allows mult simultaneous trans**, **full duplex; SWITCH vs ROUTER**: both store&fwd, have fwdin table, **router**(network layer exm network header, compute table w/ routin alg and IP addr), **switch** (link layer, learn fwdin table w/ floodin, learn MAC addr); Port-based **VLAN**: single phys switch operates as mult virtual switch, traffic isolation (only spec ports reachability), dynamically assigned ports, fwdin between VLANs done w/ routing as normal switch; **VLAN spanning** mult switch: use **trunk port** to carry frames btw VLANs defined over mult phy switches, sim to double-linklist nodes; **MPLS**(multiprotc label swichin): fast lookup w/ **fix length identifier**, not prefix match, fwd pkts **only based on label** val, diff fwdin table from IP fwdin table, more flex->both src&dest addr|quick re-route backup path if link fails; **datacenter ntwk**: internet->border router->access router->load balancer(app layer routing, direct workload, hide internal datacenter and return result to ex client)| ->T1 switch ->T2 switch->TOR switch->server racks, rich interconn w/in swich&rack to incr thruput btw racks & reliability via redundancy;

EXAMPLE: 1. PC get own IP addr, 1st hop router IP, DNS IP: use **DHCP**, ethnet frame broadcast (802.3(IP(UDP(DHCP) on LAN, rcvd by DHCP server, demux, DHCP ACK w/ client + 1stHop IP + DNS server **name&IP**, encap, frame fwd thru **LAN** (**switch learnin**) back to client, demux, DHCP client **ACK reply**; 2. Get IP of google.com: use **DNS**, encap DNS query, need MAC addr of router interface -> use **ARP** query **broadcast**, router reply w/ MAC, client rcv MAC addr of 1st hop router, send encapd frame w/ DNS query to 1st hop router; 3. IP datagram w/ DNS query fwd via **LAN switch** from client to 1st hop router, then from campus ntwk to Comcast ntwk, w/ tables created by **RIP, OSPF, IS-IS &/ BGP routing prot**, routed to DNS server, dmx, DNS rply w/ IP addr of google.com; 4. Client open **TCP socket to web server**, send **TCP SYN segment** (1st step 3way hdsk) inter-domain routed to web server, web server **rply TCP SYNACK** (2nd), setup TCP connection; 5. Client **send HTTP req** into TCP socket, IP datagram w/ HTTP req routed to google.com, webserver responds w/ HTTP reply (web page), IP datagram routed back to client.