Delay: packet arrival rate > output link capacity; packet queue/wait for turn
Delay=*node-lv processing* (check bit error/determine output link/ <1ms) + *queueing delay* (time waiting at output link for transmission/depends on congestion lv of router) + *transmission delay* (packet length(L)÷link bandwidth(R)) + *propagation delay* (physical link / propagation speed in medium)
Queueing delay: La(avg packet arrive rate)/R ~ 0: avg queueing delay is small; 1: large ; >1 : work > serviced, avg delay infinite
Packet loss: queue/buffer has finite capacity; packet arriving to full queue is lost(dropped); may be retransmitted by previous node, or not at all
Throughput: rate at which bits transferred between sender/receiver; instantaneous: rate at given point in time; average: rate over a period of time.

CH 2.1
Layering: Dealing with complex system; explicit structure allows identification, relationship of complex system's pieces; modularization eases maintenance, updating of system
Internet protocol stack: application: supporting network applications (FTP, SMTP, HTTP); transport: process-process data transfer (TCP UDP); network: routing of datagrams form source to destination (IP, routing protocols); link: data transfer between neighboring network elements (PPP, Ethernet); physical: bit "on the wire"
ISO/OSI reference model: presentation: allow applications to interpret meaning of data (encryption, compression, machine-specific conventions) (now: application layer); session: synchronization, checkpointing, recovery of data exchange (now: transport layer)
Encapsulation: source-> message (application)-> segment (transport)-> datagram (network)-> frame (link)
Packets: header + payload; may broke into pieces when sent
Applications/ application-layer protocols: Application – communication, distributed processes (running in network hosts in "user space"; exchange messages to implement app); Application-layer protocols (Specifications of messages exchanged by apps and actions taken; using services provided by lower layer protocols)
Application layer communication: 1.Specification of protocol messages; 2.Interface to network (socket); 3.parameters of low-layer protocols; Implementation of the communication in the low layers are treated as a black box; it is analogous to inter-process communication in the OS; may create many processes for communication with different hosts; three models: 1. Client-server model; 2. Pure P2P model; 3. Hybrid model
Client-Server Model: Client (initiates contact with server; typically requests service form server; for web, client is implemented in browser) Server (provide requested service); Typical properties of client (communicate with server; may be intermittently connected; may have dynamic IP address; do not communicate directly with each other); properties of server (Always on host; permanent IP address; server farms for scaling)
P2P model: Properties (node can be a client or a server; no always-on server; arbitrary end system directly communicate; peers are intermittently connected and change IP address; not too many practical applications built on pure P2P model because of difficult management)
Hybrid of client-server and P2P: eg: skype (voice-over-IP P2P application; centralized server: finding address of remote party; client-client connection: direct), BT
Implement network application: need "door" for network app can send/receive message; door (socket) appear between user space and kernel space (hind details of kernel); socket is host-local, application-created, OS-controlled interface into which application process can both send and receive messages to/from another application process; one socket is tied to one application process, but an application can create many process
Socket programming TCP: TCP is reliable transfer of bytes from one process to another; it could be view as in-order pipe; Before client contacts server, server process must first be running and created socket that welcomes client's contact (need to specify identifies of socket and connection mode, TCP or UDP); Client contacts server by creating client-local TCP socket, specifying IP address and port number of server process, when client creates socket, client TCP establishes connection to server TCP; When contacted by client, server creates new TCP socket for server process to communicate with client, allows server to talk with mult clients, source port no used to distinguish clients
Addressing processes: server can run many process serving clients; use IP address + port no to analogous to identify server process; for connection-oriented socket (TCP), when server receives a connection request, it create new TCP socket for client, the new socket is identified by [server IP, server port, client IP, client port]; for connectionless socket (UDP), as no connection, server socket is identified by [server IP, server port] only, but server can tell from the received packet what client IP and port are; server usually has fixed port no so that clients know where to connect; however client port no is allocated on demand when socket is created
Web and HTTP: web page consists of objects; web page consists of base HTML-file which includes several referenced objects; each object is addressable by URL
HTTP: hypertext transfer protocol; web's application layer protocol; client/server model, client browser that requests, receives, displays web objects, server sends objects in response to requests; Uses TCP (1. client initiates TCP connection to server, 2. server accepts TCP connection form client, 3. HTTP messages exchanged between browser and web server, 4. TCP connection closed); HTTP is stateless, HTTP message exchanges do not store any states about requests, server returns HTTP reply based on request
Nonpersistent HTTP: at most one object is sent over a TCP connection; eg: www.sss.con/file/index.html step: 1a(C): HTTP client initiates TCP connection to HTTP server at "sss"; 1b(S): HTTP server at host "sss" waiting for TCP connection at port 80. "accept" connection, notifying client; 2(C): HTTP client sends HTTP request message into TCP connection socket, message indicates that client wants object "file/index.html"; 3(S): HTTP server receives request message, forms response message containing requested object, and sends message into its socket; 4(S): HTTP server closes TCP connection; 5(C): HTTP client receives response message containing html file, displays html. Parsing html file, finds 10 ref jpeg objects; 6(C): steps 1-5 repeated for each of 10 jpeg objects; Response time is time for a small packet to travel form client to server and back, one RTT to initiate TCP connection + one RTT for HTTP request and first few bytes of HTTP response to return + file transmission time; total = 2RTT + tran time; Problem: OS overhead for each TCP connection, browsers often open parallel TCP connections to fetch referenced objects
Persistent HTTP: Mult object can be sent over single TCP connection; Default for HTTP/1.1; 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
Non-persistent vs persistent: Suppose a client req a web page with 10 images, Non need 22RTT (2 for the web page and each image), per need 12RTT only (1RTT for TCP connection setup, 1RTT for webpage and each image)
HTTP request message: types: GET POST HEAD (in 1.0 and 1.1) PUT DELETE (in 1.1); GET: input is uploaded in URL: field of request line; POST: web page form input, input is uploaded to server in entity body; HEAD: client requests for the header of the response message only; message have 3 parts, status line + header lines + data; status code: 200(OK), 310(Moved Permanently), 400(Bad Request), 404(Not found), 505(HTTP Version Not Supported)
Cookies: four components: 1. cookie header line of HTTP response message, 2. cookie header line in HTTP request message, 3. cookie file kept on user's host, managed by browser, 4. Back-end database at web site; server sends 'cookie' to client in response -> client presents cookie in later requests -> server matches presented cookie with server stored info (authentication, remembering user setting); cookie usage: authorization, user session state

CH2.2
Web Caches: goal is satisfy client request without involving origin server; user browser could accesses web via cache, browser sends all HTTP requests to cache, if object is in cache, cache returns object, else cache requests object form origin server; reason for caching is cache is in same network with client, it have a smaller response time, decrease traffic to distant server; to avoid send object if client has up-to-date stored cached version, client will specify data pf cached copy in http request (If-modified-since:<date>), server will response contains no object if cached copy FTD (304 Not Modified)
FTP: file transfer protocol; transfer file to/from remote host; composed of two connections (data and control); client (side that initiates transfer)/server (remote host) model; FTP client contacts FTP server at port 21, client authorized over control connection, client browses remote directory by sending commands over control connection, when server receives file transfer command, server opens 2nd TCP connection for file, after transferring one file, server closes data connection, FTP server maintains "state": current dir, earlier auth;
DNS Domain Name System: IP address are identifiers for addressing datagrams, and domain name are used by humans; DNS is a name-address mapping service; application-layer protocol: host, routers, name servers to communicate to resolve names; distributed db implemented is used to prevent single point of failure / hunge traffic volume / distant centralized db / difficult for maintenance; host aliasing, each host is associated with an official hostname, host can have on or more alias names, host aliasing refers an alias name to the right canonical hostname; load distribution, set of IP address is associated with one hostname, when clients make DNS query, DNS server responds with the entire set of ID, rotates the ordering of the addresses within each reply; DNS is distributed hierarchical db, eg www.amazon.com, client queries a root ever to find com DNS server -> queries com CNS server to get amazon.com DNS server -> queries form am(ry DNS server to get IP Taaddress for www(ry; root name servers, contacted by local name server that cannot resolve name, root name server contacts authoritative name server if name mapping not known, gets mapping and returns mapping to local name server; server include top-level domain servers, authoritative DNS server; local name server does not strictly belong to hierarchy, each ISP has one, when host makes DNS query, query is sent to its local DNS server
DNS name resolution example
- Host at cis.poly.edu wants IP address for www.umass.edu
- Used iterated query: contacted server replies with name of server to contact, "I don't know this name, but ask this server" (local DNS -> root DNS -> local -> TLD(.edu) -> local -> dns.umass.edu -> local -> client)
- Used recursive query: puts burden of name resolution on contacted name server (local DNS => root DNS => TLD => authoritative => TLD => root => local => client)

DNS: caching and updating records
- once name server learns mapping, it caches mapping
  = cache entries timeout (disappear) after some time
  = TLD servers typically cached in local name servers
    -Thus root name servers not often visited

-update/notify mechanisms under design by IETF (RFC 2136)

DNS records
- distributed db storing resource records (RR)
- RR format: (name, value, type, ttl)
- Type = A: name is hostname, value is IP address
- Type = NS: name is domain(foo.com), value is hostname of authoritative name server for this domain
- Type = CNAME: name is alias name for some "canonical" (the real) name, value is canonical name
- Type = MX: value is name of mailserver associated with name

DNS protocol, messages
- query and reply messages, both with same message format
- msg header
  = identification: 16 bit # for query, reply to query uses same #
  = flags (16b): query or reply, recursion desired, recursion available, reply is authoritative
  = number of questions, number of answer RRs, number of authority RRs, number of additional RRs (16 bit each)
- msg body
  = questions: Name, type fields for a query
  = answers: RRs in response to query
  =authority: records for authoritative servers
  = additional information: additional "helpful" info that may be used

Performance notes of DNS
- Sluggish DNS access can slow all communications
  = But address IP could be cached in client
- Mis-configured DNS server can cause failures
- DNS server can become a communication bottleneck
  = eg: DNS server resides behind a congested router

Security notes of DNS
- DNS helps authenticate IP addresses
  = eg: Rlogin servers recognized trusted hosts through name in .rhost file
  = Server can use RTP query to validate an IP address (match with .rhost)
- Security considerations
  = Queries can be used to compromise info about domain resources
  = Responses can be used to corrupt cache (eg: falsify identity)
  = An intruder can pretend to be a primary and update secondary db
  = Brute force: denial of service attacks on the root servers
- Extensions for dynamic changes increase risks

Additional notes of DNS
- Mobile/dynamic computing environment
  = Dynamic address changes
  =Dynamic (vs. persistent) resource availability
- DNS management
- Using DNS for load balancing
- Expanding the name space
- Generalizing resource discovery:
  = Search "find any server that provides xxx"
  = P2P networks; search engines ….

DNS Summary
- Name/directory services are of central importance
- DNS provides a name service that is
  = Scalable & distributed
  = Supports multi-domain federated management of name space
  = Permits performance tuning and control
  = Somewhat extensible
- The challenges:
  = Secure sharing of resource data
  = handling dynamic environments
  = Generalizing to broad resource discovery
  = Decentralization

Packet Capture
- Why do learn packet capture?
  = Batter understanding on how different protocol layers work
  = Useful to debug your Programming Assignments
- Idea:
  = Capture raw traffic, including Ethernet header, IP header, TCP/UDP header, application payload
- Packet Capture Tools:
  = tcpdump is command-line version of packet capture tool on Li/Unux
  = Wireshark is a graphical version of tcpdump

Using tcpdump
- need root access for traffic on a network interface, opening a file with captured traffic doesn't require root access
- Capture traffic on eth0:
  = # tcpdump –i eth0 –s 0
  = "-s 0" means to capture the whole packet (don't truncate)
- Capture traffic on eth0, and save traffic to a file out.pcap:
  = # tcpdump –i eth0 –w out.pcap –s 0
- Filtering rule could apply for focusing on a subset of packets
- Capture traffic on HTTP
  = # tcpdump –i eth0 –w http.pcap –s 0 'port 80'
- Capture traffic on DNS
  = # tcpdump –i eth0 –w dns.pcap –s 0 'port 53'
- Capture traffic on FTP
  = # tcpdump –i eth0 –w ftp.pcap –s 0 'port 20 and port 21'

Using wireshark
- Once you save the traffic to a file using tcpdump, you can open the file on wireshark
- Wireshark provides a very nice breakdown on every captured packet

CH 3.1
Transport services and 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
  = rcv side: reassembles segments into messages, passes to app layer
- more than one transport protocol available to apps
  = reliable: TCP
  = unreliable: UDP

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

Transport services and protocols
Communication Inside a Protocol Stack:
Application (Service) –Messages→ Transport (Process Communication) –Segments→Network(Machine Communication) –Packets→Network …
- Terminology
  = Message (data unit in application layer)
  = Segment (data unit in transport layer)
  = Packet (data unit in network layer, also call datagram)

Transport vs network layer
- network layer: logical communication between hosts
  = identifies a remote end host by IP address
- Transport layer: logical communication between processes
  = identifies a remote process by both IP address and port number

How is Transport Layer Implemented?
- Lecture's goal: learn the design of transport layer
- Questions that we want to answer:
  = How to pass data to the right process in a single machine?
    - Multiplexing / Demeltiplexing
  = How to send data at best?
    - UDP: best-effort (try our best) approach
  = How to send data reliably?
    - Principles of reliable data transfer
    - TCP: Internet's reliable data transfer
  = How to improve performance of transport layer?
    - Go-Back-N, selective repeat, sliding window
    - Flow control, congestion control

Review of Socket
- A socket is an abstract representation of a communication endpoint
  = Door between application process and the transport layer in kernel
- Sockets work with Unix I/O services like file, pipes & FIFOs
- Sockets (obviously) have special needs:
  = establishing a connection
  = specifying communication endpoint address

Socket Descriptor Data Structure
- Descriptor Table
- in each cell: family, service, local IP, remote IP, local port, remote port

Multiplexing/demultiplexing
- Multiplexing: gathering data from multiple sockets, enveloping data with header (later used for demultiplexing)
- Demultiplexing: delivering received segments to correct socket

How multiplexing works
- Key idea: Encapsulates a transport layer header over the app message
- Source port number: 16 bits, identify the process on sender's machine, provides a "return address" for receiver
- Destination port no: 16 bits, identify the process on receiver's machine
- Range of port numbers: 0 ~ 65535
- port no 0~1023 are called well-known port no, reserved for use by well-known application

How demultiplexing works
- Host receives IP datagrams
  = each datagram has source IP address, destination IP address
  = each datagram carries 1 transport-layer segment
  = each segment has source, destination port no (each 16 bits)
- host uses IP address & port no to direct segment to appropriate socket

Connectionless demultiplexing
- Connectionless demultiplexing is for UDP
- UDP socket identified by two-tuple: [dest IP address, dest port no]
- When host receives UDP segment:
  = checks destination port no in segment
  = directs UDP segment to socket with that port no
- IP datagrams with different source IP address and/or source port no directed to same socket

Connection-oriented demux
- Connection-oriented socket is for TCP
- TCP socket identified by 4-tuple
  = source IP address, source port no, dest IP, dest port
- receiving host uses all four values to direct segment to appropriate socket
- Server host may support many simultaneous TCP sockets
  = each socket 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

I/O Multiplexing
- So far, we discuss multiplexing of socket descriptors
- We often need to be able to monitor multiple descriptors:
  = a generic TCP client (like telnet)
  = a server that handles both TCP and UDP
  = Client that can make multiples concurrent requests
- Can we monitor multiple descriptors within one process (or thread)?
  = use select()

Blocking IO
- Suppose block read() is used, read socket first
  = Socket waits for input from keyboard
  = No response when user enter via keyboard
- read STDIN first
  = need to type something first before the message from the other side are received via socket and displayed on the screen

Select()
- select is a sensor tells which file descriptor has data ready
- select() system call allows use blocking IO on a set of descriptors
- select() is used to detect data availability
- set a timeout on select() to make it non-blocking

```
int select (int nfds, fd_set *read_fds,
  fd_set *write_fds, fd_set *except_fds,
  const struct timeval *timeout);
```
  = nfds: highest no assigned to a descriptor
  = read_fds: set of descriptor read from
  = write_fds: set of descriptors write to
  = except_fds: set of descriptors to watch for exceptions
  = timeout: max time select should wait

Timeout in select()
```
struct timeval { /*second*/ long tv_sec; /*ms*/ long
  tv_usec;}
```
select() – What is fd_set
- fd_set variable is an array of file descriptors
- void FD_CLR(int fd, fd_set *set); clears the given fd that is set
- int FD_ISSET(int fd, fd_set * set); check if given fd is set, returns 0 when FALSE
- void FD_SET(int fd, fd_set * set); set the given fd in fd_set
- void FD_ZERO(fd_set * set); clears all the fds in fd_set structure

Using select()
Pseudo-code of using select()
```
While(1){
  Use FD_ZERO to reset descriptors;
  Use FD_SET() to set the monitored descriptors;
  Call select();
  Foreach(descriptor){
    Use FD_ISSET() to check if data available;
    Process descriptor
  }
}
```
- when select() is called, set of descriptors will be changed. Need to call FD_ZERO and FD_SET to reset them again. So, they are called in while

UDP: User Datagram Protocol
- "no frills", "bare bones" Internet transport protocol
- "best effort" (try-or-best) service, UDP segments may be
  = lost
  = out of order delivery (pkts can routed via different path in network)
- Connectionless:
  = no handshaking between UDP sender and receiver
  = each UDP segment handled independently of others
- Why UDP
  = no connection establishment (reduce delay)
  = simple, no connection state at sender and receiver
  = small segment header
  = no congestion control: UDP can blast away as fast as desired
- often used for streaming multimedia apps
  = loss tolerant, rate sensitive
- other UDP uses: DNS, SNMP
- reliable transfer over UDP: add reliability at application layer
- segment format: source port #, dest port#, length, checksum (@16bit), rest are application data

Checksum: overview
- Even though UDP doesn't guarantee reliable delivery, some minimal fault tolerance is wanted
  = if packet is corrupted receiver should detect the error and drop packet
- checksum is used, a fixed-size value mapped form a chunk of data units

Checksum algo
- Goal: detect errors in transmitted segment
- sender:
  = treat segment contents as sequence 16bit int
  = checksum: addition (1's complement sum) of segment contents
  = 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 error detected.
- On the sender side:
  = add pseudoheader to the UDP segment
  = pseudoheader: to verify message is actually delivered between two end points
  = set checksum field = 0, then fill in the answer to checksum field
- pseudoheader formation:
  = source IP address, dest IP address (@32 bit)
  = 8zero, protocol (8bit), UDP total length(16 bits)

Checksum algo
```
Unsigned short in_cksum(unsigned short *addr, int len){
  int nleft = len; unsigned short *w = addr;
  int sum = 0; unsigned short answer = 0;
  // add to sum in 16-dit units
  while(nleft > 1){ sum += *w++; nleft –= 2;}
  // handle of odd len, add padding byte 0 at the end
  if(nleft == 1){ *(u_char*)(&answer) = *(u_char*)w; sum+=answer;}
  //reduce the 32 bit no to 16 bit
  sum = (sum>>16) + (sum & 0xffff);
  // wrap around overflow bits
  sum += (sum>>16);
  //1s complement
  answer = ~sum;
  // return the last 16 bits
  return (unsigned short)answer;
}
```
- on receiver side:
  = compare if checksum at the sender equals the checksum vale
  = Equivalently, receiver can run in_cksum() on received packet (including the checksum) and see if it returns zero
- The checksum algo can detect errors if one bit is flipped
- if many bits are flipped, error may not be detected, but prob is small
- algo is good enough in practice, as majority or errors are picked up by stronger error detection algorithms, such as Cycle Redundancy code at link level
- Both TCP and UCP use the same checksum algo
  = add pseudoheader, computer checksum on the whole pseu + seg
- IP (network layer) header also has checksum field computed from the same checksum algo, but only on IP header (no pseudoheader is added)
- Why computing checksum on both transport layers and network layers
  = TCP/UDP and IP may not be on the same protocol stack
  = TCP/UCP and IP compute checksums on different parts

CH 3.2
Principles of Reliable data transfer
- unreliable transmission channel: corrupted/lost/reordered packet
- reliable data transfer is having all packets correctly delivered in order
- important in app., transport, link layers
- Reliable data transfer (RDT) principle is used in:
  = app layer: end-to-end reliability (could be over a single or mult path)
  = transport layer: end-to-end reliability over a connection
  = link layer: end-to-end reliability over a single link
- RDT protocol: the interface between app layer and unreliable channel
- characteristics of unreliable channel will determine complexity of RDT protocol
- send side:
  = rdt_send(): call form above(eg: app), passed data to deliver to receiver upper layer
  = udt_send(): call by RDT, to transfer packet over unreliable channel to receiver
- receive side
  = rdt_rcv(): call when packet arrives on rev-side of channel
  = deliver_data(): called by RDT to deliver data to upper

Overview of RDT in Textbook
- Textbook consider different RDT versions under diff assumptions on channel
  = rdt1.0: channel is perfectly reliable
  = rdt2.*: packets can have bit errors
    - 2.0: bit errors appear in data packets only, using +/-acknowledgements (ACKs/NAKs)
    - 2.1 bits error can appears in on both data and control packets, using ACKs/NAKs
    - 2.2 extension of 2.1 – use only ACKs, but not NAKs
  = rdt3.0: packets can have bit errors and can be lost, use ACKs only

RDT 1.0: reliable transfer over a reliable channel
- goal: to learn how to use FSM to present RDT
- underlying channel perfectly reliable (no bit error/loss of packets)
- separate FSMs for sender, receiver
  = sender sends data into underlying channel
  = receiver read data from underlying channel

RDT2.0: channel with bits errors
- underlying channels may flip bits in packets (use checksum to detect)
- how to recover form errors:
  = ACKs: receiver explicitly tells sender that pkt received OK
  = NAKs: receiver explicitly tells sender that pkt had errors
  = sender retransmits pky on receipt of NAK
    - automatic Repeat reQuest (ARQ) refers to the reliable transfer protocols that are based on retransmissions
- new mechanisms in rdt 2.0
  = error detection
  = receiver feedback: ctrl. msgs (ACK,NAK) revr to sender
- stop-and-wait (fro rdt2.* and 3.0)
  = sender will not send a new packet until the receiver acknowledges the correct receipt of the current packet
- FSM specification:
  = sender:
    - nodeA: wait for call from above
      = red_send(data) → snkpkt = make_pkt(data, checksum); udt_send(sndpkt) → nodeB
    - node B: wait for ACK or NAK
      = rdt_rct(rcvpkt)&&isNAK(rcvpkt) → udt_send(sndpkt) → nB
      = r dt_rct(rcvpkt)&&isACK(rcvpkt) → nA
  = receiver
    - nodaA: wait for call from below
      =ret_rcv(rcvpkt)&&corrupt(rcvpkt) → udt_send(NAK) → nA
      = ret_rcv(rcvpkt)&&notcorrupt(rcvpkt) → extrack(rcvpkt,data); deliver_data(data); udt_send(ACK) → nA
- Packet Flow
  = (S) –pkt—X→ (R)pkt corrupted –NAK→ (S)retransmit pkt –pkt→pkt good –ACK→(S) send a new pkt –pkt→ …
- has a fatal flaw: what happens if ACK/NAK corrupted?
  = sender doesn't know what happened at receiver
  = cannot just retransmit: possible duplicate
- handling deplicates:
  = sender retransmits current pkt if ACK/NAK garbled
  = sender adds sequence no to each pkt
  = receiver discards duplicate pkt

RDT2.1
- FSM sender:
  = nodeA: wait for call 0 from above
    - rdt_send(data) →sndpkt=make+pkt(0,data,checksum); udt_send(sndpkt) => nodeB
  = nodeB: wait for ACK or NAK 0
    - rdt_rcv(rcvpkt)&(corrupt|| isNAK(rcvpkt) → udt_send(sndpkt) =>nB
    - rdt_rcv&notcorrupt&isACK(rcvpkt) => nC
  = nodeC: wait for call 1 from above
    - rdt_send(data) -> sndpkt=make_pkt(1,data,checksum); udt_send(sndpkt) => nD
  = noda D: wait for ACK or NCK 1
    - rdt_rcv(rcvpkt)&(corrupt|| isNAK(rcvpkt) → udt_send(sndpkt) =>nD
    - rdt_rcv&notcorrupt&isACK(rcvpkt) => nA
- FSM receiver:
  = nodeA: wait for 0 from below
    - rdt_rcv&corrupt(rcvpkt) -> sndpky=make_pkt(NAK,chksum); udt_send(sndpkt) => nA
    - rdt_rcv&notCorrupt%has_seq1(rcvpkt) -> sndpkt=make_pkt(ACK,chksum); udt_send(sndpkt) => nA
    - rdt_rcv&notCorrupt%has_seq0(rcvpkt) -> extrack(rcvpkt,data); deliver_data(data); sndpkt=make_pkt(ACK,chksum); udt_send(sndpkt) => nB
  = nodeB: wait for 1 from below
    - same as node A but flip transit state
- Packet Flow
(s) send pkt0 –pkt0–X-> (r) pkt0 corrupted, NAK, discard pkt0 –NAK-> (s) retransmit pkt0 –pkt0-> (r) pkt0 good, ACK, delivered to app. Layer –ACK–X-> (s) ACK0 corrupted, resend pkt0 –pkt0-> * pkt0 good, ACK, but delivered already, discard –ACK-> (s)send pkt1

RDT2.1 discussion
- sender:
  = seq# added to pkt, two seq#'s (0,1) will suffice, as picket are in order
  = muck check if received ACK/NAK corrupted
  = twice as many states, as state must remember current pkts has 0 or 1 seq#
- receiver:
  = must check if received packet is duplicate, state indicates 0 or 1 is expected
  = note: receiver cannot know if its last ACK/NAK received OK at sender

RDT2.2: a NAK-free protocol
- same functionality as rdt2.1, using ACKs only
- instead of NAK, receiver sends ACK for last pkt received OK
  = receiver must explicitly include seq# of pkt in ACK
- Duplicate ACk at sender results in same action as NAK: retransmit current pkt
- FSM changes
  = sender: at nodeB
    - rdt_rev(rcvpkt) & (corrupt(rcvpkt) || isACK(rcvpkt,1)) -> udt_send(sndpkt) => nB
    - rdt_rcv & notcorrupt(rcvpkt) & isACK(rcvpkt,0) => nC
  = receiver: at nodeA
    - rdt_rcv(rcvpkt) & (corrupt(rcvpkt) || has_seq1(rcvpkt)) -> udt_send(sndpkt) => nA
    - rdt_rcv & notcorrupt & has_seq0(rcvpkt) -> extract(rcvpkt,data); deliver_data(data); sndpkt=make_pkt(ACK0, shksum); unt_send(sndpkt) => nB
- Packet Flow
(s) send pkt0 –pkt0-> (r) pkt0 good, ack0, delivered to app layer –ACK0-> send pkt1 –PKT1X-> pkt1 corrupted, ack0m discard pkt1 –ACK0-> ack0 received, meaning pkt1 corrupted, resend pkt1 –PKT1-> pkt1 good, ACK1, delivered to app. Layer –ACK1X-> ACK corrupted, resend pkt1 –pkt1-> pkt1 good, ack1, but already delivered, discard –ACK1-> send pkt0 …

RDT3.0: Channels with errors and loss
- new assumption:
  = underlying channel can also lose packets (data or ACKs)
    - checksum, seq#, ACKs retransmissions will be of help, but not enough
- Approach: sender waits "reasonable" timeout for ACK
  = retransmits if no ACK received in this time
  = if pkt (or ACK) just delayed (not lost):
    - retransmission will be duplicate but use of seq#'s already handles this
    - receiver must specify seq# of pkt being ACKed
- sender requires countdown timer to trigger the retransmission
- Note that rdt3.0 is a stop-and-wait protocol
- FSM of rdt3.0 sender
  = changes:
    - added start_timer after send pkt
    - added timeout at wait, which resend pkt and start times
    - stop_timer when reserve expected ACK
- FSM of receiver is as same as 2.2 receiver
- Summary
  = sender: timer installed, timeout→retransmit pkt0/1 if NOT receive ack0/1
  = receiver: no timer, send ACK0/1 if receive pkt0/1

Summary RDT
- key elements in RDT
  = Checksum: to verify if a packet is corrupted
  = Sequence no: to identify a packet
  = Timer: to trigger retransmissions
  = ACK: for receiver to acknowledge the correct receipt of a packet

Performance of RDT 3.0
- recall: end-to-end delay = processing + queueing + transmission + propagation
- rdt 3.0 works but performance stinks
- example: 1Gbps link, 15ms propagation delay, 8000 bit packet:
  = transmission delay = L/R = 8000bits/10^9bps = 8ms
  =utilization of sender: fraction of time sender busy sending
    - U_sender = (L/R)/(RTT + L/R) = .008/30.008 = 0.00027
  = RTT here denotes round-trip propagation delay
  = 1KB pkt every 30ms -> 33kB.s thruput over 1 Gbps link
  = network protocol limits use of physical resources

Pipelined protocols
-Pipelining: send allows mult, "in-flight", yet-to-be-acknowledged pkts
  = range of sequence no must be increased
  = buffering at sender and/or receiver
- Two generic forms of pipelined protocols: go-Back-N, selective repeat
- increased utilization
  = after first pkt last bit transmitted, immediately send second pkt, and so on
  = if max pkt send at same time is 3, U_sender=(3*L/R)/(RTT+L/R) = 0.0008
- Go-back-N overview:
  = sender: up to N unACKed pkts in pipeline
  = receiver: only send cumulative ACKs, doesn't ACK pkt if there's a gap
  = sender: has timer for oldest unACKed pkt, if times up, retransmit all unACK pkt
- Selective Repeat: overview
  = sender: up to N unACKed packets in pipeline
  = receiver: ACKs individual pkts
  = sender: maintains timer for each unACKed pkt, if times up, retransmit only unACK pkt
- N is called window size

GO-Back-N
- main idea:
  = Receiver replies ACK whenever it receives a packet
    - similar to RDT 3.0, but go-back-n puts a different seq # in the replied ACk
  = Cumulative acknowledgment
    - If receiver acknowledges pktN, it means that all pervious packets are correct
- Sender:
  = k-bit seq# in pkt header, "window" of up to Nm consecutive unACK pkt allowed
  = Originally, sending window = [send_base, send_base+N-1]
  = if pkt(sned_base) is acked, then change the sending window to [+=1, +=1]
  = Go-back-N is a sliding window protocol
  = ACK-only: always send ACK for correctly-received pkt with highest in-order seq
    - may generate duplicate ACKs, need only remember expectedSeqNum
  =out-of-order pkt: discard, with no receiver buffering, Re-ACK pkt with highest in-order seq#
- Receiver:
  = keeps a variable currentSeqNum, Discard packet if receives a packet ≤ M

Selective Repeat
- Improvement over Go-Back-N: buffering out-of-order packets
  = go-back-N discards out-of-order packets
- Main idea:
  = receiver individually acknowledges all correctly received pkts
    - buffers pkts, as needed, for eventual in-order delivery to upper layer
  = sender only resends pkts(in window) for which ACK not received
    - sender timer for each unACKed pkt
  = sender window
    - N consecutive seq#'s, again limits seq#s of sent, unACKed pkts
- sender
  = data from above: if next available seq # in window send pkt
  = timeout(n): resend pkt n, restart timer
  = ACK(n) in [sendbase, sendbase+N]: mark pkt n as received, if n smallest unACKed pkt, advance window base to next unACKed seq #
- receiver
  = pkt n in [rcvnase, rcvbase+N-1]: send ACK(n), out-of-order->buffer, in-order-> deliver (also deliver bufferend, in-order pkts), advance window to next not-yet received pkt
  = pkt n in [rcvbase-N, rcvbase-1]: ACK(n)
  = otherwise: ignore
- dilemma
  = eg: seq#'s:0,1,2,3; window size=3;
  = receiver sees no difference in two scenarios
  = incorrectly passes duplicate data as new in (a)
  = occurs when window size is too large compared to seq #'s size

Go-back-N vs Selective Repeat
- Goals: improves RDT 3.0's performance using window size > 1
- Main ideas: (G) cumulative acks vs (S) Ack. Individual packets
- Drawbacks: (G) cannot handle out of order packets vs (S) need timer for individual ack'ed packets

CH 3.3
TCP: putting RDT into practice
- Is TCP go-back-n or SR?
  = traditional TCP is an improved version of GBN, today's TCP is more like SR
- How does TCP trigger retransmissions? Timeout + duplicate ACKs
- How does TCP implement N? It change dynamically with the network status.

TCP overview
- TCP is connection-oriented
  = A sender process and a receiver process must first "handshake: to agree on parameters for data transfer
  = A connection is uniquely identified by [source IP, source port, destination IP, destination port]
- TCP is point to point: A single sender process, a single receiver process
- TCP is a reliable, in order byte stream, byte is the basic unit (not packet/segment)
- TCP is full-duplex: bi-directional data flow in same connection
  = Process A can send data to process B, and process B can send data to process A in same connection
- TCP specifies a Maximum Segment Size (MSS)
  = maximum size of segment's application=layer data (excluding TCP header) that TCP will send to network
  = to ensure that a TCP segment, when encapsulated with TCP and IP headers, can fit into a single link-layer frame
  = largest link-layer frame that can be sent is called maximum transmission unit (MTU), size of MTU in Ethernet is 1500B, excluding Ethernet header
  = Typical values for the MSS are 1460, 536, 512 bytes
- TCP is pipelined, window size is determined by flow control and congestion control
- TCP has both sender and receive buffers
- TCP is flow controlled: sender will not overwhelm receiver

-TCP is congestion controlled
= sender will not overwhelm network
= There are different TCP variants, each of which implements a different congestion control algo

TCP segment structure
- source port #, dest port # (each 16 bits)
- sequence no, acknowledgment no. (each 32 bit): counting by bytes of data
- header length field (4 bit)
= length of TCP header in 32-bit words, eg: if =5, header = 20B
- not used field (6 bit)
- Flag field: each flag is 1 bit
= URG: there's urgent data in segment (not used in practice)
= ACK: equals 1 if a segment contains acknowledgment
= PSH: tell the receiver to pass to upper layer immediately (然並卵 now)
= RST: Used for connection teardown (immediately)
= SYN: used for connection establishment
= FIN: used for connection teardown (gracefully)
- receive window(16bit): # bytes receiver willing to accept, for flow control
- checksum (16 bit): internet checksum, as in UDP
- Urg data pointer (16 bit): use only when URG flag is set
- optional field: additional info (eg: MSS to be used) (size will be 32 * n)
- application data (variable length)

TCP sequence number / ACK numbers
- TCP views data as unstructured, but ordered, stream of bytes
- Sequence no for segment: byte-stream # of the first byte in segment
- ACK no: sequence # of next byte expected from other side
- Rule #1: seq no must follow ACK no reported by other side
- Rule #2: ACK no = sequence no of received segment + data length in bytes
- Note that one segment can have both sequence no and ACK no
= why? TCP is full–duplex and data can be sent form either side
- Piggybacking acknowledgment
= Acknowledge the receipt of bytes and send my data at the same time
= Use ACK bit to indicate that acknowledgment exist
- how does receiver handle out-of-order segments?
= TCP spec doesn't say, up to implementer, can be based on cumulative / selective
- what if the sender does not have data? Do I need to acknowledge?
= No, don't acknowledge the pure ACK. Otherwise there will be endless ACKs

TCP Round Trip Time and Timeout
- Recall that RDT 3.0, FBN, SR use "timeout" to trigger retransmissions
- Timeout should be based on round-trip-time (RTT)
- Challenges:
= need to be longer than RTT, but RTT varies
= too short: premature timeout, unnecessary retransmissions
= too long: slow reaction to segment loss
- Estimate timeout in TCP: sample a few timeouts of recent segments, take the average of samples, put more weight on recent samples, add some safe margin
- SampleRTT: measured time from segment transmission until ACK receipt
- sampleRTT will vary, want estimated RTT "smoother", average several recent measurements, not just current sampleRTT
- EstimatedRTT = (1- α) * EstimatedRTT + α * sampleRTT
= EstimatedRTT: exponential weighted moving average
= influence of past sample decreases exponentially fast, typical value: α = 0.125
- timeout is set to EstimtedRTT plus "safety margin", large variation in EstimatedRTT -> larger safety margin
- first estimate of how much SampleRTT deviates from EstimatedRTT:
= DevRTT = (1- β) *DevRTT + β * |SampleRTT – Estimated RTT|, typically, β = 0.25
- TimeoutInterval = EstimatedRTT + 4*DevRTT

TCP reliable data transfer
- TCP creates pipelined RDT on top of IP's unreliable service
- Different TCP variants to implement RDT
= Cumulative ACKs (eg: TCP Reno)
 - Using only a single retransmission timer for the oldest unacked segment
 - Receiver has a receive buffer to store out-of-order packets
 - Trigger retransmission if timeout/duplicate ACKs
= An extended version of Go-Back-N
- selective acknowledgment (eg: TCP-SACK)
= similar to selective repeat

TCP simplified sender
- Initially, consider a TCP simplified sender
= using cumulative ACK, ignore flow control congestion control
= don't trigger retransmissions if duplicate ACKs received (only retran@timeout)

Algo:
```
NextSeqNum = InitialSeqNum; SendBase = InitialSeqNum;
While(1){
  Switch(event){
    Case data_received_from_application_above:
      Create TCP segment with sequence number NextSeqNum;
      If (Timer currently not running) start times;
      Pass segment to IP;
      NextSeqNum = NextSeqNum + length(data);

    Case timer_timeout
      Retransmit no-yet-acknowledged segment with smallest seq no;
      Start timer;

    Case ACK_received_with_ACK_field_value_of_y
      If ( y > SendBase){
        SendBase = y;
        If (there are currently not-yet-acknowledged segments) start timer;
      }
  }
} // loop never end
```

TCP retransmission scenarios]
Case 1 - Lost ACK scenario
(A) –Seq=92, 8B data->(B) –ACK=100 X->(A)…timeout for seq92 –Seq=92, 8B data >(A) discard –ACK=100-> (B)SendBase = 100
- A retransmits the same segment, B discards the 2nd segment
Case 2 - Premature timeout scenario
- Host A retransmits segment 'seq 92' upon timeout, host a may choose to send 'seq 100' as well, depending on implementation
- When Host A receives 'ACK 100', it updates Sendbase = 100 and restarts timer if there are outstanding unacked segments
- When Host A receives 1st 'ACK 120', it updates sendbase = 120
- When Host A receives 2nd ACK120, ignore
Case 3 – Cumulative ACK scenario
(A) –Seq=92, 8B data-> (B); (A) –Seq=100, 20B data->(B), (B) –ACK=100 X-> (A), (A) –ACK=120-> (B) sendBase = 120
- if Host A receives 'ack120' before timeout, it updates directly sendbase=120, host A doesn't resend both segments 'seq92' and 'seq 100'
- GBN uses the same principle

TCP Receiver
- Base case:
= Immediately reply ACK whenever ACK whenever a segment is received
= Applicable to both Go-Back-N and Selective Repeat
- TCP improves the base case using delayed ACK:
= Wait a short time, to grad the chance of acking more than one segment

TCP ACK generation
Event at receiver / TCP Receiver action
- Arrival of in-order segment with exp seq#. All data up to exp seq# already acked / Delayed ACK. Wait up to 0.5s for next segment. If no next segment, send ACK
- Arrival of in-order segment with exp seq#. One other segment has ACK pending / Immediately send single cumulative ACK, ACKing both in-order segments (ie, ACKing every two segments)
-Arrival of out-of-order segment higher-than-exp seq#. GAP detected / immediately send duplicate ACK, indicating seq# of next expected byte
- Arrive of segment that partially or completely fills gap/ Immediate send ACK, provided that segment starts at lower end of gap

Fast Retransmit
- Tcp simplified sender only uses "timeout" to trigger retransmissions
- time-out period often relatively long: long delay before resending lost pkt
- detect lost segments via duplicate ACKs
= sender often sends many segments back-to-back
= if segment is lost, there will likely be many duplicate ACKs for that segment
- if sender receives 3 ACKs for same data, it assumes that segment after ACKed data will lost: fast retransmit, resend the missing segment before timer expires
- Algo:
```
Event: ACK received, with ACK field value of y
  If (y > sendBase){
    sendBase = y;
    if (exist currently not yet acked segments) start timer
    else {
      increment count of dup ACKs received for y;
      if(count of dup ACKs received for y=3) resent segment with seq no y;
    }
  }
```

TCP-SACK
- in default TCP implementation today, SACK is info embedded in TCP option field
- The basic idea is to include Bytes which is welly reserved at the RCP option field
- for example, for header SEQ = 1, ACK = 6, datalen=0, TCP Option: [[11,15],[21,25]]
= The consecutive byte expected is 6, Received 11 ~ 15 byte and 21~25 byte
= sender knows that only 6~10 bytes and 16~20 bytes are required to retransmit
= after retransmission 6~10, the ACK header will be like this:
 - SEQ = 1, ACK = 16, datalen=0, TCP Option: [21,25]
 - Cumulative acknowledgement, all bytes up to 16 has been received

TCP Flow Control
- sender won't overflow receiver's buffer by transmitting too much too fast
- receive side of TCP connection has a receive buffer
- app process may be slow at reading from buffer
- speed-matching service: might send rate to receiving application's drain rate
- receiver: advertises unused buffer space by including rwnd value in segment header, recall that there's a field for rwnd in the TCP header
- sender: limits# of unACKed B to rwnd, guarantees receiver's buffer not overflow
- unused buffer space = rwnd = RcvBuffer – [LastByteRcvd – LastByteRead]

TCP Connection Management
- What is connection management? Set up / tear down connection
- Connection Setup
= Recall that TCP sender, receiver establish 'connection' before exchanging data
= necessary to initialize TCP variables: seq.#s, buffers, flow control info
= Achieved with 3-way handshake

Three-way handshake
- Step 1: Client host send TCP SYN segment to server
= specifies initial seq#, no data, TCP flag: SYN bit is set to 1
- Step 2: Server host receives SYN, replies with SYNACK segment
= server allocates buffers, specifies server init seq#, TCP flag: SYN & ACK set to 1
- Step 3: Client receives SYNACK, replies with ACK segment, which may contain data
= TCP flags: ACK bit is set to 1

TCP Connection Teardown
- TCP implements 4-way handshake in closing a connection
- Not 3-way as the other side may not be ready to close connection immediately
= eg: other side is calling a blocking system call

Four-way handshake
- Step 1: Client and system sends TCP FIN control segment to server
- Step 2: Server receives FIN, replies with ACK. Closes connection, sends FIN
- Step 3: Client receives FIN, replies with ACK; enter 'timed wait', will respond with ACK to received FINs, in case server resend FIN (lost ACK replay)
- Step 4: Server, receives ACK. Connection closed
- Server will close after timeout even if server never receives ACK

State Machine for 3 and 4 way handshake
- Client:
= State A: CLOSED
 - client application initiates a TCP connection, send SYN => sB
= State B: SYN_SENT
 - receive SYN and ACK, send ACK => sC
= State C: ESTABLISHED
 - client application initiates close connection, send FIN => sD
= State D: FIN_WAIT_1
 - receive ACK, send nothing => sE
= State E: FIN_WAIT_2
 - receive FIN, send ACK => sF
= State F: TIME_WAIT
 - wait for 30s -> sA
- Server:
= State A: CLOSED
 - server application creates a listen socket => sB
= State B: LISTEN
 - receive SYN, send SYN & ACK => sC
= State C: SYN_RCVD
 - receive ACK, send nothing => sD
= State D: ESTABLISHED
 - receive FIN, send ACK = > sE
= State E: CLOSE_WAIT
 - send FIN => sF
= State F: LAST_ACK
 - receive ACK, send nothing => sA

CH 3.4

Application VS Transport – socket()
- socket() system call: creates a set of data for the TCP structure. A file structure is selected to represent the entire TCP socket structure
- bind() system call: assigns a port number to a socket. Bind() can be executed by both the client and server
- listen() system call: set the state of TCP FSM to LISTEN on the PASSIVE side
- before listen(), netstat will return nothing
- after listen():
```
# netstat -a -t -n | grep 12345
 Tcp ... 0.0.0.0:12345 ... 0.0.0.0:* ... LISTEN
```
Where local address means program is running on port 12345, foreign address means program welcome from any IP address and any ports, the TCP FSM state is LISTEN
- connect() system call: sets the state of TCP FSM to SYN_SENT of the active side
- accept() system call does not involve in TCP control, user can have a better control over an accepted TCP connection, by using a file descriptor

Congestion
- TCP uses both flow control and congestion control to limit the sending rate
= Flow control: sender won't overflow receiver's buffer by transmitting too much, too fast, Receiver fills in the receive window in TCP header to inform the sender what's the right sending rate
= Congestion control: sender won't overflow the network, but how does the network inform the sender what's the right sending rate?

Principles of Congestion Control
- Congestion: too many sources sending too much data too fast for network to handle, which congestion control is different from flow control, resulting lost packets (buffer overflow at routers) or long delays (queueing in router buffers)

Approaches towards congestion control
-two board approaches towards congestion control
= end-end congestion control:
 - no explicit feedback form network
 - congestion inferred form end-system observed loss, delay
 - approach taken by TCP
= network-assisted congestion control:
 - router provide feedback to end systems
 - single bit indicating congestion (SNA, DECbit, TCP/IP ECN, ATM)
 - explicit rate sender should send at

TCP congestion control
- End-to-end congestion control
- Main goal: TCP sender should transmit as fast as possible, but without congestion network
- sender limits rate by limiting number of unACKed bytes "in pipeline"
= LastByteSend – LastByteAcked ≤ min(cwnd,rwnd)
= rwnd: window size for flow control; cwnd: window size for congestion control
- roughly, wate = W/RTT (bytes / sec), W = min(cwnd, rwnd)
- cwnd is dynamic, function of perceived network congestion
- decentralized: each TCP sender sets its own rate, based on implicit feedback
= ACK: segment received, network not congested, so increase sending rate
= lost segment: assume loss due to congested network, so decrease sending rate
- "probing for bandwidth": increase transmission rate on receipt of ACK, until eventually loss occurs, then decrease transmission rate
= continue to increase on ACK, decrease on loss (since available bandwidth is changing, depending on other connections in network)
- Three major comonents:
= slow start, when ACK received, increase cwnd exponentially fast at connection start, or following timeout
= Congestion avoidance, when ACK received, increase cwnd linearly
= Fast recovery (optional), if a segment is lost, decrease cwnd less aggressively such that slow start is avoided
- segment loss event: reducing cwnd
= timeout: no response from receiver, cut cwnd to 1
= 3 duplicate ACKs: at least some segments getting through, cut cwnd in half, less aggressively that on timeout
- ACK received: increase cwnd
= slowstart phase, increase exponentially fast at connection start, or following timeout
= congestion avoidance: increase linearly

TCP: slow Start
- when connection begins, cwnd = 1 MSS, eg: MSS = 500B, RTT = 200ms
- available bandwidth may be >> MSS/RTT, desirable to quickly ramp up to respectable rate
- increase rate exponentially until first loss event or when slow-start threshold reached, double cwnd every RTT, done by incrementing cwnd by 1 for every ACK

Transitioning into/out of Slow start
- The exponential growth end cannot last forever
= Slow-start threshold ssthresh: cwnd threshold maintained by TCP
- on loss event: set ssthresh to cwnd/2
= remember (half of) TCP rate when congestion last occurred
- when cwnd >= sstresh: transition from slowstart to congestion avoidance phase

TCP congestion avoidance
- when cwnd > ssthresh grow cwnd linearly
= increase cwnd by 1 MSS per RTT
= approach possible congestion slower than in slowstart
= implementation: cwnd = cwnd + MSS/cwnd for each ACk received
 - assume that each ACK acknowledges one MSS for data, in one RTT, we expect to receive ACKS for all cwnd bytes, so cwnd is incremented by 1 after 1 RTT

TCP Fast Recovery
- when a lost segment is received, cut cwnd
- TCP tahoe (no fast recovery), Always set cwnd = 1 upon loss
- TCP Reno (with FR), use with fast retransmit, if 3 dup ACKs received, cut cwnd by half (dup ACKs imply that at least some segments get through), if timeout, still set cwnd = 1 (network is really congested)

TCP FSM
Start: cwnd=1MSS, ssthresh = 64kB, dupACKcount = 0
- node A: slow start
= duplicate ACK -> dupACKcount++ => nA
= new ACK -> cwnd = cwnd+MSS, dupACKcount = 0 => nA
= timeout -> ssthresh = cwnd/2, cwnd = 1MSS, dupACKcoun t =0 => nA
= cwnd >= ssthresh => nB
= dupACKcount == 3 -> ssthresh = cwnd/2, cwnd = ssthresh + 3 => nC
- nodeB: Congestion avoidance
= new ACK -> cwnd = cwnd+MSS*(MSS/cwnd), dupACKcount = 0 => nB
= duplicate ACK => dupACKcount ++
= dupACKcount == 3 -> ssthresh = cwnd/2, cwnd = ssthresh + 3 (account for 3 ACKs)
- nodeC: fast recovery
= duplicate ACK -> cwnd=cwnd+MSS => nC
= timeput -> ssthresh = cwnd / 2, cwnd = 1, dupACKcount = 0 => nA
= New ACK -> cwnd = ssthresh, cupACKcount = 0 => nB

TCP: Retrospective
- TCP congestion control is referred to as an Additive Increase, Multplicative Decrease (AIMD) algo
= Ignore slow start and assume losses are indicated by dup ACKs, not timeouts
= ACKs: increase cwnd by 1 MSS per RTT, additive increase
= loss: cut cwnd in half (non-timeout-detected loss): mult decrease
= cwnd exhibits sawtooth behaviour
- The AIMD mechanism serves as a distributed asynchronous-optimization algo

Summary: TCP Congestion Control
- when cwnd < ssthresh, sender in slow-start phase, window grows exponentially
- when cwnd >= ssthresh, sender is in congestion-avoidance phase, grows linearly
- when triple duplicate ACK occurs, ssthresh set to cwnd/2, cwnd set to ~ssthresh
- when timeout, ssthresh set to cwnd/, cwnd set to 1MSS
- average throughout of TCP as function of window size, RTT: Ignoring slow start
- let W be window size when loss occurs, when window is W, throughtput is W/RTT, after loss, window drops to W/2, throughput to W/2RTT, average throughput: 75

TCP futures: TCP over "long, fat pipes"
- eg: 1500byte segments, 100ms RTT, want 10Gbps throughput
- requires window size W = 83333 in-flight segments
- throughput in terms of loss rate: 1.22MSS / (RTT*L^1/2), L = 2*10^-10
- new versions of TCP for high-speed

TCP Fairness
- fairness goal: if K TCP sessions share same bottleneck link of bandwidth R, each should have average rate of R/K
- TCP is fair, in two competing sessions:
= Additive increase gives slope of 1, as throughout increases
= mult decrease decreases throughput proportionally
- fairness and UDP
= multimedia apps often not use TCP, as don't want rate throttled by congestion control, use UDP for pump audio/video at constant rate, tolerate packet loss
- Fairness and parallel TCP connections
= nothing prevents app for opening parallel, connections between 2 hosts
= web browsers do this, eg: link of rate R supporting 9 connections: new app ask for 1 TCP, gets rate R/10, new app asks for 11TCPs, get R/2

Conclusions on Transport Layer
- Transport layer deals with end-to-end communication between two communication processes
- Part 1:
Multplexing / demultiplexing
= Multiplexing: encapsulate data chunks with transport layer header, and pass the chunks to the network
= Demultiplexing: analyse the header, and pass the chunks to the right socket
- UDP: a best-effort transport layer implementation
Part 2:
- Principles of reliable data transfer
= rdt3.0: how to deliver one packet reliably to the other end?
= Go-back-N, Selective Repeat: pipelined versions of rdt3.0
Part 3:
- TCP: implementation of reliable data transfer
= Sequence numbers / acknowledgment numbers, timeout estimation, flow control, connection management: 3-way/4-wat handshakes
Part 4:
- Congestion control: Slow start, congestion avoidance, fast recovery

CH 4.1

Network layer
- transport layer: process-process communication
- network layer: comm of hosts (called routers), Think network as a graph, each node is a router
- router examines header fields in all IP datagrams passing through it

Two key Network-Layer Functions
- forwarding: move packets form router's input to appropriate router output
- routing: determine route taken by packets form source to dest, routing algo
-analogy:
= routing: process of planning trip from source to dest
= forwarding: process of getting through single interchange

The Internet Network Layer
- Network layer is between Transport layer and Link layer
- In network layer, there are three protocols
= Routing protocols: path selection, RIP, OSPF, BGP, work with forwarding table
= ICMP protocol: error reporting, router "signalling"
= IP protocol: addressing conventions, datagram format, packet handing conventions

IP datagram format
- IP header length = 20B + length of options
- Version: 4b, current version is 4, refers to IPv4, latest version is 6, refers to IPv6
- IHL: 4b, header length, length is measured in 32-bit words, eg: 20Byte, field = 5
- Type of service: 8b, also called differentiated service fiddle, ask router to provice different classes of service (minimize delay, maximize throughput, minimize reliability, minimize monetary cost), many app nowadays ignore this and set 0
- Total Length: 16b, length of entire IP packet: header + payload, measures number of bytes, max = 65535 bytes
- Identification(16b), Flags(3b), Fragment Offset(13b): belong to IP Fragmentation
- Time to Live(TTL): 8b, number of hops that IP packets is allowed to visit, gen Time exceed error when TTL = 0
- Protocol(8b), Header Checksum(16b), 32bit source IP, 32bit destination IP, Options (if any), data (variable length, typically TCP or UDP segment, with header)

IP Fragmentation and Reassembly
- network links have MTU (max transfer size) – largest possible link-level frame
= different link types have different MTUs
- large IP datagram divided (fragmented), within net
= one datagram becomes several datagrams, reassembled only at final destination (not in routers)
= IP header bits used to identify, order related fragments
- Three fields to identify IP fragments:
= Identifier: 16b, identify the original IP packet
= Flags: 3b, =1 : there are more following fragments, =0 : this is the last fragment
= Fragmentation offset: 13b, start position of the fragment (in units of 8B chunks)
- eg: 4000B datagram, MTU = 1500B, include header + data
= length = 4000; ID = x; fragflag = 0; offset = 0 ➔ length = 1500; ID = x; fragflag = 1/1/0; offset = 0/185/370 (1480 bytes in data field, offset = 1480/8 * 1 or 2)

IP Addressing: introduction (focus on IPv4)
- IP address: 32-bit identifies for a host or an interface
- IP addresses are hierarchical and composed of:
= network part: identifier of a network (which may contain mult sub-networks)
= host part: identifier of a host/interface within a network
- Why hierarchical?
= Facilitates routing: a packet can be first routed to a large network, then to a sub-network, and finally to a host. Flat addressing scheme (eg: Ethernet address) makes routing difficult
- interface: connection between host/router and physical link
= router's typically have mult interfaces, host typically has one interface, IP add associated with each interface

Subnets
- what's a subnet
= device interfaces with same subnet part of IP address
= can physically reach each other without intervening router
- IP address: subnet part (high order bits), host part (low order bits)
- how do I know which subset is used from an IP address?
= IP addressing uses a subnet mask, s.t. the leftmost bits from the network address (i.e. subnet part). Eg: 137.189.0.0/16, /16 is a subnet mask, and the leftmost 16 bit form a network, may represent as /255.255.0.0

Classful addressing
- Traditional addressing scheme constrains network portions of an IP address to be 8/16/24 bits, correspond to class A,B,C respectively
- Class A, first bit is 0 (0.0.0.0 ~ 127.255.255.255)
- Class B, first two bits are 10 (128.0.0.0 ~ 191.255.255.255)
- Class C, first three bits are 110 (192.0.0.0 ~ 223.255.255.255)
- Class D, Multicast address (224.0.0.0 ~ 239.255.255.255)
- Class E, reserved, not used (240.0.0.0 ~)
- Usable address
= Host parts with last bits 0 and 255 are reserved, ~0 usually to represent a network address, ~255 is reserved for broadcast
= no of unable host address in a class C network: 2^8 – 2 = 254
- More on usable addresses:
= @class A, 0.*.*.* / 127.*.*.* reserved, 127~ is used to identify local interfaces
= Private address: not used to identify hosts global, only recognized by hosts within a subnet (10.*.*.* / 172.16.0.0. ~ 172.31.255.255 / 192.168.*.*)
= Link local address: locally assigned by a host and not routable (169.254.*.*)

IP addressing: CIDR
- Classful addressing is not efficient, eg: class C may be too small for an organization but C may be too large
- CIDR: Classless InterDomain Routing
= a.k.a. prefix routing, subnet portion of address of arbitrary length
= address format: a.b.c.d/x, where x is # bits in subnet portion of address (the x most significant bits are called the network prefix)
- Example: CSE department is given the address range 137.189.88.0/22
- What are the valid addresses in that range
= create a subnet mask: /22 -> 11111111.11111111.11111100.000000000
= Apply bitwise AND operations on an IP address and the subnet mask
= if it's equal to the subnet, them the address is in the subnet
= You can show that the range is 137.189.88.0 ~ 137.189.91.255

IP addressing: additional notes
- Note that a subnet must defined with a network prefix
= If the subnet mask in not available, then 137.189.88.0 cannot be properly defined

IP address: how to get one
- how does a host get Ip address?
= static approach: hard-coded by system admin in a file
= DHCP: Dynamic Host Configuration Protocol: dynamically get address from as server, "plug and play"

Dynamic Host Configuration Protocol (DHCP)
- goal: allow host to dynamically obtain its IP address from network server when it joins network
= Can renew its lease on address in use
= allow reuse of addresses (only hold address while connected an "on")
= Support for mobile users who want to join network (more shortly)
- DHCP overview:
= host broadcasts "DHCP discover" msg → DHCP server responds with "DHCP offer" msg → host requests IP address: "DHCP request" msg → DHCP server sends address: "DHCP ack" msg

DHCP client-server scenario
- Assume DHCP server has IP address: 223.1.2.5
- Arriving client: send DHCP discover (src:0.0.0.0,68/ dest:255.255.255.255,67/ yiaddr:0.0.0.0/ transaction ID: 654) here, transaction ID is used to identity
- DHCP server: send DHCP offer (src:223.1.2.5,67/ dest:255.255.255.255,68/ yiaddr:223.1.2.4/ transaction IP: 654/ Lifetime: 3600s) server say 223.1.2.4n in non-use, with lifetime 1hour
- Arriving client: send DHCP request (src:0.0.0.0,68/ dest: 255.255.255.255,67/ yiaddrr: 223.1.2.4/ transaction ID:655/ lifetime:3600s) client ask if it can use 223.1.2.4 as its IP, with lifetime 1 hour
- DHCP server: send DHCP ACK (src: 223.1.2.5,67/ dest:255.255.255.255,68/ yiaddrr: 223.1.2.4/ transaction ID: 655/ Lifetime 3600s) server accept request

DHCP: more than IP address
- DHCP clients periodically renew the DHCP leases (ie, resend DHCP request)
- DHCP can return more than just allocated IP address on subset
= address of first-hop router for client
= name and IP address of DNS sever
= network mast (including network versus host portion of address)

IP address: how to get one?
- How does network get subnet part of IP address?
= get allocated portion of its provider ISP's address space
= eg: CSE dept get the block 137.189.88.0/22 from the CUHK address space 137.189.0.0/16
- How does an ISP get block of address
= ICANN: Internet Corporation for Assigned Names and Numbers
 - allocates addresses/ manages DNS/ assigns domain names, resolves disputes

NAT: Network Address Translation
- eg: ←rest of internet→(138.76.29.7)Routor(10.0.0/24)←local network→ 10.0.0.1, 10.0.0.2, 10.0.0.3
- in rest of internet, all datagrams leaving local network have same single source NAT IP address: 138.76.29.7, but different source port numbers
- In local network, datagrams with source or destination in the network have 10.0.0/24, address for source, destination (as usual)
- Motivation: local network uses 1 IP address as far as outside world is connected
= range of addresses not needed from ISP: just one IP address for all devices
= can change address of devices in local network without notifying outside world
= can change ISP without changing addresses of devices in local network
= devices inside local net not explicitly addressable, visible by outside world
- Implementation: NAT router must
= outgoing datagrams: replace [source IP, port #] of every outgoing datagram to [NAT IP, new port #], remote clients/servers will respond using [NAT IP, new port#] as destination addr
= remember (in NAT translation table) ever [source IP, port#] to [NAT IP, port#] translation pair
= incoming datagrams: replace [NAT IP, new port#] in dest fields of every incoming datagram with corresponding [source IP, port#] stored in NAT table
- working example: take the network in the pervious example
= 1. Host 10.0.0.1 sends datagram to 128.119.40.186,80
= 2. NAT router changes datagram source addr from 10.0.0.1, 3345 to 138.76.29.7,5001, update table
= 3. Reply arrives dest address: 138.76.29.7,5001
= 4. NAT router changes datagram dest addr 138.76.29.7,5001 to 10.0.0.1, 3345
- 16-bit port-number field
= 60000 simultaneous connections with a single LAN-side address
- NAT is controversial
= routers should only process up to layer 3
= violates end-to-end argument, NAT possibility must be taken into account by app designers, eg, P2P applications
= address shortage should instead be solved by IPv6

NAT traversal problem
- client wants to connect to server with address 10.0.0.1
= server address 10.0.0.1 local to LAN (client can't use it as destination addr)
= only one externally visible NATted address: 138.76.29.7
- solution: statically configure NAT to forward incoming connection requests at given port to server, eg: [123.76.29.7, 2500] always forwarded to 10.0.0.1, 25000

Internet Control Message Protocol
- ICMP is used by hosts and routers to communicate network-layer information to each other
= mainly used for error reporting: to diagnose if a host is running, or if a route exists
- ICMP is considered part of IP since it reports IP-layer information, but in fact it architecturally lies above IP, ICMP messages are carried as IP payload

Structure of an ICMP packet
- under 20 Byte IP header
- Type (8bit), Code(8bit), checksum(16bit), message body (depending on type and code)
- Type and code together define the meaning of an ICMP packet

Definitions of Type and Code
- Type/Code (description)

- 0/0 echo reply (ping)	- 3/0 dest.network unreachable
- 3/1 dest host unreachable	- 3/2 dest protocol unreachable
- 3/3 dest port unreachable	- 3/6 dest network unknown
- 3/7 dest host unknown	- 8/0 echo request (ping)
- 4/0 source quench (congestion control – not used)	
- 9/0 route advertisement	- 10/0 router discovery
- 11/0 TTL expired	- 12/0 bad IP header

Uses of ICMP
- two main use
= PING: test if a host is reachable / find the round-trip-time of a host
= TRACEROUTE: show the route taken by packets across an IP network

PING
- how PING works?
= sender sends a ICMP ECHO REQUEST, [type = 8; code = 0]
= receiver sends a ICMP ECHO REPLY, [type=0; code=0]
= after type, code, checksum, add Identifier (identify which "IP flow" this ICMP packet belongs to.), sequence number (number of trials)
- Someone suggests to use ping to detect whether a machine is reachable or not
- Some networks has deployed firewalls to stop ICMP packets from going in and out
= But, not filtering UCP and TCP packets, therefore, in addition to the PING program, we need more powerful tools such as the nmap
- If you want to reach server because of HTTP service, PING absolutely can't help you, so, the usefulness of PING depends on how you use it

Traceroute
- source sends series of UDP segments to dest, first has TTL = 1. Second has TTL=2…
- When nth datagram arrives to nth router:
= Router discards datagram, sends to source an ICMP message [type11, code0]
= message includes name of router and IP address
- When ICMP arrives, source also calculates the RTT, RTT is calculated 3 times for each intermediate node
- When does traceroute stop sending UDP packets?
= UDP segment eventually arrives at destination host
= Destination returns ICMP "host unreachable" packet [type3, code3]
= when source gets this ICMP, stops
- Command (for Linux): traceroute www.cse.cuhk.edu.hk
- in windows, command is tracert

Some notes on ICMP
- how can I create and send a ICMP message in socket programming?
= use RAW socket

CH 4.2

IPv6 Background
- Initial motivation: 32-bit address space soon to be completely allocated
- Additional motivation:
= header format helps speed processing / forwarding
= header changes to facilitate QoS
= fixed-length 40 byte header
= no fragmentation allowed

IPv6 Header
- Version (4b)
- Priority (Traffic Class): 8b, identify priority among datagrams in flow
- flow label: 24b, identify datagrams in same "flow"
- Payload Length (16b)
- Next Header: 8b, identify upper layer protocol for data
- Hop Limit (8b)
- Source Address (16 Bytes), Destination Address (16 Bytes)

Other Changes form IPv4
- Checksum: removed entirely to reduce processing time at each hop
- Options: allowed, but outside of header, indicated by "Next Header" field
- ICMPv6: new version of ICMP, additional message types, multicast group management functions

IPv6 addressing
- Allocation is classless
= Prefixed specify different uses (unicast, multicast, anycast)
 - Anycast: send packets to nearest member of a group
= Lots of flexibility with 128 bits
 - ~1500 address/sqft of the earth surface, every grain of sand has it own IP
- Standard representation is et of eight 16-bit values separated by colons
= if there are large number of zeros, they can be omitted with series of colons
 - eg: 47CD:1234:3200::4325:B792:0428, eight zeros is omitted
= Address prefixes (slash notation) are the same as v4
 - EG: FEDC:BA98:7600::/40 describes a 40bit prefix

Unicast Assignment in v6
- Unicast address assignment is similar to CIDR
= Unicast addresses start with 001
= Host interfaces belong to subsets
= Addresses are composed of a subnet prefix and a host identifier
= Subnet prefix structure provides for aggregation into larger networks
- Anycast addresses are treated just like unicast addresses
= It's up to the routing system to determine which server is "closest"

Transition From IPv4 to IPv6
- Not all routers can be upgraded simultaneous, no "flag days"
- Two approaches for network operate with mixed IPv4 and IPv6 routers
= Dual-stack: A router implements both IPv4 and IPv6 protocols and translates between format, if either one router is only IPv4-capable, only IPv4 can be used, two IPv6 routers may communicate with IPv4
= Tunneling: IPv6 carried as payload in IPv4 datagram among IPv4 routers
- in network transitional performance, dual-stack is faster than tunnelling as tunnelling header has one additional header. However, dual stack will miss out IPv6-specific fields missing, eg: flow table
- to enable IPv6, in cmd of Win, type "ipv6 if", need to install (ipv6 install) first

Virtual Private Network (VPN)
- Why VPN?
= Deploy a stand-alone network, including routers/links, that is separated from the public Internet, which have better management and security
= Challenges of setting up a physical private network
= VPN is a private network that is overlaid on the top of the public network
= Communication within the private network will remain inaccessible to the public network, even though the VPN packets are transmitted over the public
= Data is securely protected

VPN Security
- VPN security is based on the IPSec protocol
- IPSec provides end-to-end protection of IP datagrams between any two network-layer entities
- IPSec services
= Data integrity: Data cannot be changed during transmissions
= Origin authentication: Data must be sent by the origin (not by anyone else)
= Replay attack prevention: Data is not replayed
= Confidentiality: Data is encrypted

VPN Tunneling
- VPN is built upon tunnelling, encrypt the payload and encapsulate a new IP header from local, decrypt the payload and decapsulate the IP header at VPN provider
- IPSec supports two mode
= Tunnel mode (default), Entire IP packet (including data and IP header) is encrypted, for network-to-network communication for VPN
= Transport mode: Only the IP payload is encrypted, for host-to-host communication
- VPN uses tunnel mode
- to make VPN compatible with NAT, UDP encapsulation is usually used
= eg: after applying UDP encapsulation to a VPN packet
- Typical port no for UDP-encapulated packets: 500 or 4500
- Downsides of tunnelling
= Increase the length of packets, increase the chance of fragmentation
= Increase the overhead at routers
= Configurations of the VPN routers need to be well managed

Definitions – network to graph
- each node (router) usually maintains a forwarding table and a routing table
- A forwarding table stores the next hop for each destination
- A routing table stores the next hop and the cost of the path for each destination
- Depending on the context, they are used interchangeably

Graph abstraction
Graph: G = (N, E)   N = set of routers (nodes) = {u, v, w, x, y, z}
E = set of links = {(u, v), (u, x), (v, x), (v, w), (x, w), (x, y), (w, y), (w, z), (y, z)}
C (x, x') = cost or link (x, x'), cost could always be 1, or inversely related to bandwidth or related to congestion
Cost of path (x1, x2, xp) = c(x1, x2) + c(x2, x3) + … + c(xp-1, xp)
Routing algorithm: algorithm that finds least-cost path

Routing Algorithm classification
Global or decentralized information?
- Global: all routers have complete topology link cost info, "link state" algo
- Decentralized: router knows physically-connected neighbours, link costs to neighbours. Iterative process of computation, exchange of info with neighbours, "distance vector" algorithms
Static or Dynamic?
- Static: routes change slowly over time
- Dynamic: routes change quickly. Periodic update, in response to link cost changes
Load-sensitive or load-insensitive?
- Load-sensitive: link costs vary with current level of congestion
- Load-insensitive: link costs do not explicitly reflect current congestion. More stable, uses by today's internet routing algorithms

<u>A Link-State Routing Algorithm</u>
- Dijkstra's algorithm
 = Net topology. Link costs known to all nodes, accomplished via 'link state broadcast', all nodes have same info
 = computes least cost paths from one node (source) to all other nodes, gives forwarding table for that node
 = iterative: after k iterations, know least cost path to k dest.'s
- Notation:
 = c(x,y): link cost from node x to y, =inf if not direct neighbours
 = D(v): current value of cost of path from source to dest. v
 = p(v): predecessor node along path from source to v
 = N': set of nodes whose least cost path definitively known

<u>Dijkstra's Algorithm</u>
```
Initialization:
   N' = {u}
   for all nodes v
      if v adjacent to u
         then D(V) = c (u,v)
      else D(v) = inf

Loop
   find w not in N' such that D(w) is a minimum
   add w to N'
   update D(v) for all v adjacent to w and not in N' :
      D(v) = min ( D(v), D(w) + c(w,v))
      /* new coust to v is either old cost to v or known
        shortest path cost to w plus cost from w to v */
until all nodes in N
```

<u>Dijkstra's algorithm: example</u>

t	N'	D(v), p(v)	D(w), p(w)	D(x), p(x)	D(y), p(y)	D(z), p(z)
0	u	2,u	5,u	1,u	Inf	Inf
1	ux	2,u	4,x		2,x	Inf
2	uxy	2,u	3,y			4,y
3	uxyv		3,y			4,y
4	uxyvw					4,y
5	uxyvwz					

Resulting forwarding table in u:
Destination / next hop
v/v; x/x; y/x; w/x; z/x

Resulting routing table in u:
destination / next hop / cost
v/v/2; x/x/1; y/x/2; w/x/3; z/x/4

<u>Dijkstra's algorithm, discussion</u>
- Algorithm complexity: n node, m edges
 = each iteration: need to check all nodes, w, not in N
 = $O(n^2)$ comparisons performed by each node
 = more efficient implementations possible using heap: O(nlogn + m)
- Oscillations possible:
 = eg: link cost = amount of carried traffic

<u>Distance Vector Algorithm</u>
Bellman-Ford Equation (dynamic programming)
Define d_x(y) := cost of least-cost path form x to y
Then d_x(y) = min_v{c(x,y) + d_v(y)}
When min is taken over all neighbours v of x

<u>Bellman-Ford example</u>
By walking along the paths, we found d_v(z) = 5, d_x(z) = 3, d_w(z) = 3
B-F equation: d_u(z) = min{c(u,v) + d_v(z), c(u,x) + d_x(z), c(u,w) + d_w(z)} = 4
Bellman-Ford ensures that after the B-F equation is called a finite number of times, the algorithm will converge.

<u>Distance Vector Algorithm</u>
- D_x(y) = estimate of least cost from x to y
- Node x knows cost to each neighbour v: c(x,v)
- Node x maintains distance vector D_x = [D_x(y): y in N]
- Node x also maintains its neighbours' distance vectors
 = For each neighbour v, x maintains D_v = [D_v(y): y in N]
- Each node x maintains a distance table with DV D_x and the DVs of all neighbors
- Base idea:
 = from time-to-time, each sends its own distance vector estimate to neighbours
 = Asynchronous
 = When a node x receives new DV estimate form neighbour, it updates its own DV using B-F equation: D_x(y) ←min_v{c(x,v) + D_v(y)} for each node y in N
 = Under minor, natural conditions, the estimate D_x(y) converge to the actual least cast d_x(y)
- Iterative, asynchronous, each local iteration caused by:
 = local link cost change, DV update message from neighbour
- Distributed: each node notifies neighbours only when its DV changes
 = neighbours then notify their neighbours if necessary
- In each node:
 = wait for (change in local link cost or msg form neighbour)
 = recompute estimates
 = if DV to any dest has changed, notify neighbours

<u>Example:</u>

node x table				node y table				node x table			
cost to	x	y	z	cost to	x	y	z	cost to	x	y	z
from x:	0	2	7	from y:	2	0	1	from z:	7	1	0
x:	0	2	3	x:	0	2	7	x:	0	2	7
y:	2	0	1	y:	2	0	1	y:	2	0	1
z:	7	1	0	z:	7	1	0	z:	3	1	0
x:	0	2	3								
y:	2	0	1								
z:	3	1	0								

in step 2, x,y in x table, D_x(y) = min{c(x,y)+Dy(y), c(x,z)+Dx(y)} =2
Graph: x—2—y—1—z—7—x

<u>Distance Vector: link cost changes</u>
- Link cost changes:
 = node detects local link cost change, updates routing info, recalculates distance vector, If DV changes, notify neighbours
At time t0, y detects the link-cost change, updates its DV, and informs its neighbors
At time t1, z receives the update from y and updates its table. It computes a new least cost to x and sends its neighbours its DV
At time t2, receives z's update and updates its distance table. Y's least costs do not change and hence y does not send any message to z
- good news travels fast, bac news travels slow – count to infinity problem
Eg: x—4 change to 60—y—1—z—50—x
- y thinks Dz(x) = 5 and so chooses z as its new next hop, but doesn't know that z's next hop is y itself
- main problem is that y doesn't have the knowledge of the topology
- 44 iterations before algorithm stabilizes
- y thinks Dz(x) = 5
- Step 1: Dy(x) = min{c(y,x) + Dx(x), c(y,z)+Dz(x)} = min {60+0, 1+5} = 6
- Step 2: y informs z that Dy(x) = 6, then z computes Dz(x) = min {50+0, 6+1} = 7
- Step 3: z informs y that Dz(x) = 7. Then y computes Dy(x) = min {60+0, 1+7} = 8

<u>Poisoned reverse</u>
- If Z routes through Y to get to X:
 = Z tells Y its (Z's) distance to X is infinite (so Y won't route to X via Z)
 = Then Y won't choose Z as its next hop, since Z's next hop (or Y next hop's next hop) will be U itself
- Similar approach: split horizon
 = Z simply doesn't advertise route to Y
- It doesn't work in general if the loop has 3 or more nodes
- When link z-w breaks, z tells x and y that w is unreachable. How does poisoned reverse happen?
- Graph: z—x—y—z—broke here—w
- When link z-w breaks, z tells x and y that w is unreachable
- Suppose x learns it first
 = x new thinks the best path to w is through y
 = x reports to y that w is unreachable through x itself (poisoned reverse)
 = x reports to z that it has a route to w with route cost 3
 = z thinks w is reachable through x with route cost =4 and reports it to y
 = y thinks w is reachable through z with route cost = 5 ……

<u>Comparison of LS and DV algorithms</u>
- Message complexity
 = LS: with n nodes, E links O(nE) msgs sent
 = DV: exchange between neighbours only, convergence time varies
- Speed of convergence
 = LS: O(n^2) algorithm requires O(nE) msgs, may convergence time varies
 = DV: convergence times varies, mau be routing loops, count-to-inf problem
- Robustness: what happens if router malfunctions?
 = LS: node can advertise incorrect link cost, each node computers only its own table
 =DV: DV node can advertise incorrect path cost, each node's table used by others, error propagate thru network

CH 4.3
<u>Why Software Routers?</u>
- Software routers (aka Virtual routers) are the computers/PCs that have routing functionalities
- more flexible control over hardware routers
- Larger storage than hardware routers, good for network traffic monitoring / management
- Limitations of software routers: speed is generally slower than hardware ones

<u>How to make a PC become a router?</u>
- A router has to connect to at least two different networks
 = So, it is a requirement that a router contains at least two network interfaces

<u>IP routing in Linux router</u>
- The soul of a router is the routing table
 = The routing table is stored inside the OS kernel memory, no matter is is a windows or a linux machine
 = the routing table is a set of rules that determine how the machine should forward a packet

```
[root@Linux]# route –n
Destination    Gateway          Genmast            Flags
172.16.66.200  0.0.0.0          255.255.255.255    UH
172.16.66.0    0.0.0.0          255.255.252.0      U
0.0.0.0        137.189.91.254   0.0.0.0            UG
```
"-n" means without DNS lookups
- Destination: the destination is presented in terms of a host IP address or a network address, 255.255.255.255 is telling you that this is host. Other are telling you that they are networks
- Flags: U: route is up; H: the route is to host; G: the route is to a gateway
- If the destination of the UP packet is the host "172.16.66.200", then
 = This packet should go the interface vmnet8 to look for the host "172.16.66.200"
 = there is no intermediate gateway, just find the host through vment8
- if the destination of the IP packet belongs to the network "172.16.66.0/24", then
 = this packet should go to the interface vmnet8 to look for the destination
 = there is no intermediate gateway, just find the host through vmnet8
- If the destination of the IP packet belongs to the network "0.0.0.0/0", them
 = This packet should go to the <u>interface eht0</u> to look for the destination;
 = This packet should <u>go to the intermediate router "137.189.91.254"</u> through eth0
 = Every IP address belongs to network 0.0.0.0/0, as any address AND 0 = 0

<u>How to modify routing table?</u>
- Adding and removing a network:
 = route add –net 10.10.10.0/24 dev eth0
 = route del –net 10.10.10.0/24
- Adding and removing a host:
 = route add –host 10.10.10.45 dev eth0
 = route del –howt 10.10.10.45
- Adding a default gateway:
 = route add default gw 192.168.0.1
 = route del default gw 192.168.0.1

<u>Issue #1: Routing</u>
- Suppose I want to route packets to destination 137.189.91.23, but the routing table has two entries
- both 137.189.88.0/22 and 137.189.91.24 match, which entry should I use?
- Longest prefix match: choose the matching entry with the highest subnet mask
- Thus 137.189.91.0/24 is used
- how to find the matching entry?
 = Linear search: Sort the routing entries by subnet mask, from highest to lowest, fine the first one that matches
- A faster search based on the data structure trie is usually used

<u>Issue #2: Routing</u>
- How to send packets to the default gateway?
 = Should I change the destination address to gateway's address?
- Instead, use ARP (address resolution protocol)
- Main idea: to look for the Ethernet address for an IP address
- Will talk about it when we talk about link layer

<u>Iptables</u>
- The tool iptables is about too many things
 = Packet Filtering / Packet Forwarding / Network address Translation (NAT) / Connection Tracking / etc…
- To enable packet forwarding in Linux
 = echo 1 > proc/sys/net/ipv4/up_forward
- iptables is a user-level protram that control the kernel-level network module call netfilter

<u>Iptables – Tables and Chains</u>
- each function provided by the netfilter architecture is presented as a table
 = filter: this table is in charge of filtering packets
 = nat: this table is in charge of translating IP addresses of the packets
 = mangle: this table is in charge of changing packet contest
- under each table, there are a set of chains, under each chain, rules are assigned
 = filter: INPUT / OUTPUT / FORWARD
 = nat: PREROUTING / POSTROUTING / OUTPUT
 = mangle: INPUT / OUTPUT / FORWARD / PREROUTING / POSTROUTING

```
[root@linux]# iptables –t filter –L
Chain INPUT (policy ACCEPT)
Target    port    opt    source      destination
DROP      icmp    --     anywhere    anywhere

Chain FORWARD (policy ACCEPT)
Target    port    opt    source      destination

Chain OUTPUT (policy ACCEPT)
Target    port    opt    source      destination
[root@linux]#_
```
- there is one rule set in the INPUT chain, the outer two chain is empty
- above rule: when a packet with ICMP payload passes through the INPUT hook, DROP that packets, no matter it is from anywhere and to anywhere

<u>IP tables – packet flow</u>
1. Incoming packets goes to [PREROUTING], filtering can be done before routing at the [PREROUTING] hook, then go to routing rules
2. If the destination of the packet is this machine, incoming packet will go to [INPUT], filtering can be done at [INPUT] before the packet arrives [local process]
3. If the destination of the packet is not this machine, and this machine knows where the packet should be send, the incoming packets will go to [FORWARD] and [POSTROUTING], there are two places to filter the traffic that has to be forward
4. [Local processes] can send packets out. To go though [OUTPUT] before touting, then it will be sent to [Routing Rules]
5. if the destination of the outgoing packets is another [local process], the packet will be sent to [INPUT] from [Routing Rules], means it going through a loop. The device handles this kind of traffic is called loopback device
6. If the packet from the [local packet] is set to leave, it will go through the [POSTROUTING] hook from [Routing Rules]

<u>NAT Rules Explained</u>
- IP Masquerading
 = it is a special cast of the NAT, automatically associate private [srcIP, port] to public [srcIP, port]
 = to translate any outgoing packet:
  - from any source IP address to the my IP address
  - from any source port number to the port number assigned by me
 = The MASQUERADE target only valid for the POSTROUTING chain of the nat table
- Examples
```
iptable -t nat -A POSTROUTING -s 172.16.1.0/24 -d 137.189.0.0/16 -j MASQUERADE
```
 = your private network can "access" CUHK network and itself only
```
iptable -t nat -A POSTROUTING -p tcp -d ! 172.16.1.0/24 --dport 22 -j MASQUERADE
```
 = your private network can only use SSH to reach the outside world
```
iptable -t nat -A POSTROUTING -s 172.16.1.0/24 -j SNAT --to 137.189.91.187
```
 = Hardcode the public source IP assignment to 137.189.91.187 (-j SNAT should appear before --to)
- Can I map the traffic form outside to inside?
 = eg: I set up a web server in my private network, I want the public to access the server
 = USE DNAT
```
iptable -t nat -A PREROUTING -i eth0 -j DNAT --to 172.168.1.1
```
 =change dest address to 172.168.1.1
```
iptable -t nat -A PREROUTING -p tcp --dport 80 -i eth0 -j DNAT --to 172.168.1.1:8080
```
 = Direct all web traffic to 172.168.1.1,port 8080 (port forwarding)
 = The gateway computer redirects connections to the specified ports to the designated internal computer and ports and arranges for return traffic to go back to the original address outside the network

<u>More rules on iptables</u>
- List all entries in the NAT table: iptables –t nat –L
- Flush all entries in the NAT table: iptables –t nat –F
- You can also do packet filtering with iptables
 = Drop incoming ICMP packets: iptables –A INPUT –p icmp –s 0/0 –j DROP
 = DROP all incoming TCP traffic destined from ports 0 to 1023 : iptable –A INPUT –p tcp –s 0/0 –d 0/0 --drop 0:1023 –j DROP

<u>Summary on iptables (NAT part)</u>
- NAT can change the source addresses and the destination address on IP packets
- MASQUARADE target
 = changes the source address to be the gateway's address before the packet leaves the gateway at POSTROUTING hook, and
 = changes the destination address automatically back to the original source address (and is done at PREROUTING hook quietly)
- SNAT target
 = focuses on changing the source address of the packet at the POSTROUTING hook to any address specified by the rule
 = Therefore, MASQUARADW is a special case of SNAT
- DNAT target
 = focuses on changing the source address of the packet at the PREROUTING hook to any address specified by the rule

<u>Can we do more with netfilter</u>
- use libnetfilter_queue
- capture packets into your user-space programs
- Do whatever you want on the captured packets, drop them, modify them… Implement your own routing algo and apply the algo to captured packets
- Improve prior version IPQUEUE with multiple queue support

<u>The target</u>
- Review: we add a target to each chain
 = iptables -A input –j [target name]
- eg: we can attach INPUT chain with one of the targets: ACCEPT, REJECT, DROP. Every matched packet will go to the target
- Goal: trap a rule-matched packet inside netfilter
Iptables –A INPUT –p tcp --dport 80 –j NFQUEUE --queue-num 0
 = specify which queue you trap packets in --queue-num

<u>The NFQUEUE target</u>
- Write a user-program to read the target queue, and process the queued packets in any way you want:
iptables -A INPUT –p tcp --dport 80 –j NFQUEUE --queue-num 0
 = A program that is able to manipulate queued packets

<u>libnetfilter_queue</u>
- The libnetfilter-queue library provides an interface for a user-space program to access the netfilter queue (which is in kernel)
- How to install?
 = $ sudo apt-get install libnetfilter-queue*
 = install both library and header code

<u>Program flow of using NRQUEUE</u>
(bootstrap phase) nfq_open() → nfq_unbind_pf()/nfq_bind_pf() → nfq_create_queue() → nfq_set_mode() → nfq_nfnlh() → (main phase) recv() – packets → Callback() → nfq_set_verdict() → recv() –queue closed→ nfq_destroy_queue() → nfq_close()

<u>- Bootstrap Phase</u>
 = nfq_open(): Obtains a netfilter queue header
 = nfq_bind() / nfq_unbind(): Binds the handler to process IP packets, not sure why we need to unbind first (if unbind() is commented, bind will fail)
 = nfq_create_queue(): install a callback function on queue number, eg, nfq_create_queue(nfqHandle, 0, &Callback, NULL) will install Callback() on queue 0
 = nfq_set_mode(): Sqecifies how to capture a packet
  - NFQNL_COPY_NONE: Do not copy any data
  - NFQNL_COPY_META: Copy only packet metadata
  - NFQNL_COPY_PACKET: Copy entire packet
 = at the end, bind the NFQUEUE handle with the socket handle
  - struct nfnl_handle* netlinkHandle = nfq_nfnlh(nfqHandle); int fd = nfnl_fd(netlinkHandle)

<u>- Main Phase</u>
 = Process matched packets
 while((res = recv(fd, buf, sizeof(buf), 0)) && res >= 0)
  { nfq_handle_packet(nfHandle, buf, res); }
 = We may simply process buf directly rather than in a callback function, but using a callback is more like a convention
 = Get the packet ID, which you need later…
  header = nfq_get_msg_packet_hdr(pkt));
  if (header != NULL) id = ntohl(header->packet_id);
 = you can work on the payload (assume you enable NFQNL_COPY_PACKET mode)
  int len = nfq_get_payload(pkt, &pktData);
  if (len > 0) for(int i=0; i<len; i++) printf("%02x ", (unsigned char)pktData[i];
- where does the payload start?
 = The payload starts with the IP header
 = Use casting to manipulate headers:
  struct ip* ip_hdr = (struct ip*)pktData;
  struct tcphdr* tcp_hdr = (struct rcphdr*)((unsigned char*)pktData + (ip_hdr->ip_hl << 2));
 = Don't just add 20 Bytes (or sizeof(struct ip)), IP header could have varying length
 = Headers are defined in /usr/include/netinet/ip.h AND usr/include/netinet/tcp.h
 = Find out UDP and other headers in usr/include/netinet/*

<u>- Main Phase – set verdict</u>
 = After you are done processing a packet, you need to decide its fate
 = nfq_set_verdict()
  - NF_ACCEPT: accept packet   OR   NF_DROP: drop the packet
 = eg: nfq_set_verdict(myQueue, id, NF_ACCEPT, 0, NULL);
  - The id is the value obtained before
 = Format of nfq_set_verdict()
 Int nfq_set_verdict(struct nfq_q_handle *qh, u_int32_t id, u_int32_t verdict, u_int32_t data_len, unsigned char *buf)
 = if data_len = 0, buf = NULL, it refers to the original packet
 = You can put new payload
 Nfq_set_verdict(myQueue, id, NF_ACCEPT, newDataLen, newBuf);

<u>- End phase</u>
 = called when you delete the rule from iptables
```
[root@Linux]# iptables -D INPUT -p tcp --dport 80 -j NFQUEUE
  --queue-num 0
```
 = Destroy the queue and close the NFQUEUE handle
  - nfq_destroy_queue(myQUEUE);   AND   nfq_close(nfqHandle);

<u>Modifying Packets</u>
- if you need to modify a packet, you need to pass it to the mangle table
- Example: to modify a packet in the POSTROUTING chain
```
iptables -t mangle -A POSTROUTING -j NFQUEUE --queue-num 0
```

<u>Checksum</u>
- NOTE: if you have modified the IP header (eg: in NAT operations), make sure the IP header checksum is computed correctly

<u>IP Checksum Calculation</u> // "addr" is pointing to the start of IP header, len is length
```
unsigned short in_cksum(unsigned short *addr, int len){
  register int nleft = len;      register unsigned short *w = addr;
  register int sum = 0;          unsigned short answer = 0;
  while (nleft > 1){   // for every 2 bytes, add 2 bytes to the variable "sum"
     sum += *w++;        nlefr -= 2;
  }
  if(nleft == 1){       // What if the length is an odd number?
     *(u_char *) (&answer) = *(uchar *) w;     sum += answer;
  }
  sum = (sum >> 16) + (sum & 0xffff);   // Reduce the 32-bit number to 16-bit
  sum += (sum>>16);       answer = ~sum;       return(answer);
}
```
- See 3.1 CheckSum part

<u>Other approaches</u>
- There is a library called libcap
 = core of Wireshark, can capture packets at data link layer only, not stopping them
- There is a library called libnet
 = can generate whatever packets you need, at what ever layer of communication
- The iptables can stop any packets, at the network layer level
- Say, a gateway can be made
 = Capture incoming packet by "libpcap" at Data Link Layer and send a copy of the original packet to application, drop incoming packet by "iptables" in Network layer, regenerate a new packet using the original copy by "libnet" in App layer and send back to Physical layer

CU 4.4
<u>Why Scheduling?</u>
- Packet flow are multiplexed in routers
- Share results in contention resources, such as packet queues and output link capacity
- A scheduling discipline resolves contention
- Key to fair sharing resources and provide quality-of-service(QoS) performance guarantees

<u>Components of Scheduling</u>
- A scheduling discipline does two things: Decides service order AND manages queue of service requests
- Example:
 = Let us consider queues awaiting web server, scheduling discipline decides service order, scheduling discipline decides if some query should be dropped

<u>Where do we implement scheduling?</u>
- Anywhere where contention may occur, at every layer of protocol stack
- Usually studied at network layer, at output queues of routers/switches
- Scheduling disciplines can allocate: Bandwidth, Delay and Loss
- Scheduling disciplines also determine how fair the network is

<u>Simplest Scheduling: FIFO Queueing</u>
- First-i-first-out (FIFO) queueing is probably the simplest scheduling approach
 = Dispatches the packet that first arrives
 = Drops packets when the queue is full (tail drop)
- Limitations of FIFO: favors "greedy" flow that have most packets / hard to control the delay of packets

<u>Fairness</u>
EG: A in 10Mb/s to R, B in 100Mb/s to R, R out 1.1Mb/s to C
- what is a fair allocation? [0.55Mb/s, 0.55Mb/s] or [0.1Mb/s, 1Mb/s]?
EG: add D 0.2Mb/s to R
- What is fair now?

<u>Max-Min Fairness</u>
- Resources are allocated in the order of increasing demand
- No source gets a resource share larger than its demand
- Sources with unsatisfied demands get an equal share of the resources
- Intuition:
 = each connection gets no more that what it wants
 = The excess, if any, is equally shared
- N sources and source i has rate (demand) xi. Let x1 ≤x2 ≤…≤xn. Outgoing capacity is C. Let the actual output rate of i by yi
- Algo:
  N' = N
  for(i=1; i<N; i++){ yi = min(xi, C/N'); C=C-yi; N'-- }

<u>Example of Man-Min Fairness</u>
- Four sources with demands 2, 2.6, 4 and 5. The capacity is C = 10;
- Step 1: y1 = min(2, 10/4) = 2. Remaining C = 8
- Step 2: y2 = min(2.6, 8/3) = 2.6. Remaining C = 5.4
- Step 3: y3 = min(4, 5.4/2) = 2.7. Remaining C = 2.7
- Step 4: y4 = 2.7
- The max-min fair allocation is (2, 2.6, 2.7, 2.7)

<u>Fair Queueing</u>
- Packets belonging to a flow are placed in a FIFO, this is called "per-flow queueing"
- FIFOs are scheduled one bit at a time, in a round-robin fashion
- This is called Bit-by-Bit Fair Queueing (or bit-by-bit round-robin)

<u>Weighted Bit-by-Bit Fair Queueing</u>
- Likewise, flows can be allocated different rates by servicing a different no of bits for each slow during each round
- For example, serve 3 flows with weight 1,2,2: f1,f2,f2,f3,f3,f1,f2,f2,f3,f3,…
- Also called Generalized Processor Sharing (GPS)

<u>Packetized Weighted Fair Queueing (WFQ)</u>
- problem: we need to serve a whole packet at a time
- Solution: Determine what time a packet, p, would complete if we served flows bit-by-bit. Call this the packet's finish time, F. Serve packets in the order of increasing finishing time.
- Also called <u>Packetized Generalized Processor Sharing (PGPS)</u>

<u>Intuition behind Packetized WFQ</u>
- Suppose, in each round, the server served one bit from each active connection
- Round no is the number of rounds already completed, which can be fractional
- If a packet of length p arrives to any empty queue when the round no is R, it will complete service when the round no is R+p => finish number is R+p, which is independent of the no of other connections
- If a packet arrives to a non-empty queue, and the previous packet has a finish no of f, them the packet's finish no is f+p
- Serve packets in order of finish no
- to sum up, assuming we know that current round number R
- Finish number of packet of length p
 = if arriving to active connection = previous finish number + p
 = if arriving to an inactive connection = R + p
- there are hundreds to millions of flows; the line card needs to manage a FIFO per flow
- The finish time must be calculated for each arriving packet
- For each arrival packet, packets must be sorted by their departure time. Naively, with m packets, the sorting time is O(logm)

<u>Deficit Round Robin (DRR)</u>
- An O(1) approximation to WFQ
- For each flow i, keeps a quantum size Qi and a deficit counter Di. A flow has more share with larger Qi
- For each round-robin scan
 = serves as many packets as possible for flow I with size less than Qi + Di
 = if packets remain in flow i's queue, stores the deficit in Di
- Fair in the long term for any combination of packet sizes

<u>Example of DRR</u>
- Consider 4 flows. Let Qi = 400, Initially, set all Di = 0
F1[20,750,200]/Deficit counter 0; F2[500, 500]/0; F3[200, 600, 100]/0; F4[50,700,180]/0
- Step 1: available credit for F1 = Q1 + D1 = 400 + 0 = 400
- Can send the first packet (size 200), D1 = 400-200 = 200
F1 change to F1[20,750,<u>200</u>]/200
- Step 2: available credit for F2 = Q2 + D2 = 400
- Cannot send the first packet of size 500, add quantum size to D2, D2 = 400
F2 change to F2[500,500]/400
- After Step 3 and 4, F3[200,600,<u>100</u>]/300; F4[50,700,<u>180</u>]/220
- After Step 5, 6, 7, 8, F1[20,750,<u>200</u>]/600; F2[500,<u>500</u>]/300; F3[200,<u>600,100</u>]/100; F4[50,700,<u>180</u>]/620

<u>Implementation of DRR</u>
- DRR is a credit-based approach
 = Accumulates credits. And use the credits to send packets
- Provides excellent bandwidth guarantees
- One major problem (trade off): poor delay bounds
- Implementation complexity
 = Need to skip a lot of queues to find next active queue
 = We can use an active list for maintaining this
 = However, it can lead to inactive queues not accumulating their fair share

<u>Packet dropping</u>
- Packets that cannot be served immediately are buffered
- Full buffers => packet drop strategy
- Packet losses happen almost always from best-effort connections
 = eg: UDP-based streaming
- Shouldn't drop packets unless imperative, packet drop wastes resources

<u>Classification of drop strategies</u>
1. Degree of aggregation   2. Drop priorities
3. Early or late   4. Drop position

<u>1. Degree of aggregation</u>
- Degree of discrimination in selecting a packet to drop
- Eg: in vanilla FIFO, all packets are in the same class
- Instead, can classify packets and drop packets selectively
- The finer the classification the better the protection
- Max-min fair allocation of buffers to classes, drop the packet form class with the longest queue

<u>2. Drop priorities</u>
- Drop lower-priority packets first
- How to choose?
 = endpoint marks packets / regular marks packets / congestion loss priority (CLP) bit in packet header (if bit is marked, high priority. Not good to be dropped)
- Pros: if network has spare capacity all traffic is carried. During congestion, load is automatically shed
- Cons: separating priorities within a single connection is hard. What prevents all packets being marked as high priority?
- Drop packets form 'nearly' hosts first, because they have used to least network resources. Can't do it on Internet because hop count (TTL) decreases

<u>3. Early vs late drop</u>
- Early drop => drop even if space is available
 = signals endpoints to reduce rate, cooperative sources get lower overall delays, uncooperative sources get severe packet loss
- Early random drop
 = drop arriving packet with fixed drop prob if queue length exceeds threshold
 = intuition: misbehaving sources more likely to send packets and see packet losses
 = DOESN'T WORK!!!! 然而並沒有什麼卵用，因為可能會浪費空位！
- Random early detection (RED) makes three improvements
 = Metric is moving average of queue lengths
  - Small bursts pass through unharmed, only affects sustained overloads
 = packet drop prob is a function of mean queue length, prevents severe reaction to mild overload
 = Can mark packets instead of dropping them, allows sources to detect network state without losses

<u>Random Early Detection (RED)</u>
- Input parameters:
 = AverageQ: average queue length   = MinThreshold: minimum threshold
 = MaxThreshold: Maximum threshold
- AverageQ is estimated via exponential weighted average of different sample queue lengths
 = AverageQ = (1-W) * AverageQ + W * SampleQSize
- Algorithm
```
For each newly arriving packet
  If(AverageQ < MinThreshold)
     Admit the packet
  Else if (MinThreshold < AverageQ < MaxThreshold)
     Drop the packet with probability
     p = (AVerageQ – MinThreshold) / (MaxThreshold – MinThreshold)
  else
     drop the packet
```

<u>4. Drop position</u>
- Can drop a packet form head, tail, or random position in the queue
- Tail: easy, default approach
- Head: harder, let source detect loss earlier
- Random: hardest, if no aggregation, hurts hogs most, unlikely to make it to real
- Drop entire longest queue: easy, almost as effective as drop tail from longest queue

<u>Traffic Shaping</u>
- Traffic shaping (or <u>rate limiting</u>): control the outgoing rate of the router
- Goal: to avoid overloading downstream routers
- Two approaches: Leaky bucket / Token bucket

<u>Leaky Bucket</u>
- Across a single link, only allow packets across at a constant rate
- Packets may be generated in a bursty manner, but after they pass through the leaky bucket, they enter the network evenly spaced
- If all inputs enforce a leaky bucket, it's easy to reason about the total resource demand on the rest of the system
- It is a "traffic shaper": It changes the characteristics of a packet stream
- Traffic shaping makes the network more manageable and predictable
- Usually the network tells the leaky bucket the rate at which it may send packets when a connection is established
- Can also be configured by network administrators
- Doesn't allow bursty Transmissions
 = In some cases, we may want to allow short bursts of packets to enter the network without smoothing them out
 = For this purpose we use a token bucket, which is a modified leaky bucket

<u>Token Bucket</u>
- The bucket holds logical tokens instead of packets
- Tokens are generated and placed into the token bucket at a constant rate
- When a packet arrives at the token bucket, it is transmitted is there is a token available. Otherwise it is buffered until a token becomes available
- The token bucket holds a fixed number of tokens, so when it becomes full, sudsequently generated tokens are discarded
- can still reason about total possible demand

<u>Token Bucket vs Leaky Bucket</u>
Case 1: Shout burst arrivals
Arrival time at bucket: $\boxed{0}\boxed{1}\boxed{2}$ 3 4 $\boxed{5}$ 6
Departure time form a leaky bucket, Leaky bucket rate = 1 packet/2 time units
$\boxed{0}$ 1 $\boxed{2}$ 3 $\boxed{4}$ 5 $\boxed{6}$ 每兩秒比一個過
Departure time from a token bucket, Token bucket rate = 1 tokens/2 time units, Token bucket size = 2 tokens $\boxed{0}\boxed{1}\boxed{2}$ 3 4 $\boxed{5}$ 6 一開始有兩個 token，兩秒多一
Case 2: Large Burst arrivals
Arrive time at bucket: $\boxed{0}\boxed{1}\boxed{2}\boxed{3}\boxed{4}$ 5 6
Departure time form a leaky bucket, Leaky bucket rate = 1packet/2 time units
$\boxed{0}$ 1 $\boxed{2}$ 3 $\boxed{4}$ 5 $\boxed{6}$
Departure time form a token bucket, Token bucket rate = 1 token / 2 time units, token bucket size = 2 tokens $\boxed{0}\boxed{1}\boxed{2}$ 3 $\boxed{4}$ 5 $\boxed{6}$

CH 4.5
<u>Hierarchical Routing</u>
- Our routing study thus far – idealization
 = all routers identical, network "flat"
 Are not true in practice
- Scale: with 200 million destinations:
 = can't store all dest's in routing tables!
 = Routing table exchange would swamp links!
- Administrative autonomy
 = Internet = network of networks
 = each network admin may want to control routing in its own network
- aggregate routers into regions, "autonomous systems" (AS)
- routers in same AS run same routing protocol
 = "intra-AS" routing protocol
 = routers in different AS can run different intra-AS routing protocol
- Gateway router
 = Direct link to router in another AS

<u>Interconnects ASes</u>
EG: AS1 have 4 routers, 1c connect to AS3, 1b connect to AS2
- forwarding table configured by both intra- and inter-AS routing algorithm
 = intra-AS sets entries for internal dests
 = inter-AS & intra-As sets entries for external dests

<u>Inter-AS tasks</u>
- suppose router in AS1 receives datagram destined outside of AS1:
 = router should forward packet to gateway router, but which one?
- AS1 must:
 = learn which dests are reachable through AS2, which through AS3
 = propagate this reachability info to all routers in AS1
- Job of inter-AS routing!

<u>Example: Setting forwarding table in router 1d</u>
- suppose AS1 learns (via inter-AS protocol) that subnet x reachable via AS3 (gateway 1c) but not via AS2
- inter-AS protocol propagates reachability info to all internal routers
- router 1d determines for intra-AS routing info that its interface I is one the least cost path to 1c, installs forwarding table entry (x,l)

<u>Example: Choosing among multiple ASes</u>
- now suppose AS1 learns from inter-AS protocol that subnet x is reachable form AS3 and form AS2
- to configure forwarding table, router 1b must determine towards which gateway it should forward packets for dest x, this is also job of inter-AS routing protocol
- hot potato routing: send packet towards closest of two routers
 = Learn from inter-AS protocol that subnet x is reachable via multiple gateways
→ Use routing info from intra-AS protocol to determine costs of least-cost paths to each of the gateways → how potato routing: Choose the gateway that has the smallest least cost → Determine from forwarding table the interface I thatleads to least-cost gateway. Enter (x,l) in forwarding table

<u>Intra-AS Routing</u>
- also known as <u>Interior Gateway Protocols (IGP)</u>
- most common Intra-AS routing protocols:
 = RIP: Routing Information Protocol
 = OSPF: Open Shortest Path First
 = IGRP: Interior Gateway Routing Protocol (Cisco proprietary)

<u>RIP (Routing Information Protocol)</u>
EG: A—B—D—C—A [A connect u] [B: v w] [C: z] [D: x y]
- distance vector algorithm   - included in BSD-UXIX Distribution in 1982
- distance metric: # of hops (max = 15 hops)
From router A to subnets: (destination/hops)
u/1, v/2, w/2, x/3, y/3, z/2

<u>RIP advertisements</u>
- Distance vectors: exchanged among neighbours every 30 sec via Response Message (also called advertisement)
- each advertisement: list of up to 25 destination subnets within AS

<u>RIP Example</u>
w—[A]—x—[D]—–[B]—y—[…]…[…]—Z; [A]—–[C]
Routing/Forwarding table in D [Destination Network/NextRouter/No of hops to dest] [w/A/2] [y/B/2] [z/B/7] [x/-/1]
- if advertisement from A to D, have [z/C/4], table in D will change to [z/A/5]

<u>RIP: Link Failure and Recovery</u>
- if no advertisement head after 180 sec → neighbour/link declared dead
 = routes via neighbour invalidated
 = new advertisements sent to neighbours
 = neighbours in turn send out new advertisements (if tables changed)
 = link failure info quickly propagates to entire net
 = poison reverse used to prevent ping-pong loops (inf distance = 16 hops)

<u>RIP Table processing</u>
- RIP routing tables managed by application-level process called route-d (daemon)
- advertisements sent in UDP packets, periodically repeated

<u>OSPF (Open Shortest Path First)</u>
-"open": publicly available
- uses Link State algorithm: LS packet dissemination / topology map at each node / route computation using Dijkstra's algorithm
- OSPG advertisement carries one entry per neighbour router
- advertisements disseminated to entire AS (via flooding)
 = carried in OSPF messages directly over IP (rather than TCP or UDP)

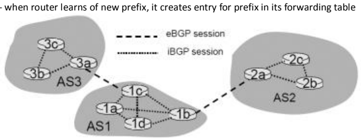
<u>OSPF "advanced" features (not in RIP</u>
- security: all OSPF messages authenticated (to prevent malicious intrusion)
- multiple same-cost paths allowed (only one path in RIP)
- for each link, multiple cost metrics for different TOS (eg: satellite link cost set "low for best effort; high for real time)

- integrated uni- and multicast support:
 = Multicast OSPF (MOSPF) uses same topology data base on OSPF
- hierarchical OSPF in large domains

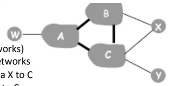
Internet inter-AS routing: BGP
- BGP (Border Gateway Protocol): the de facto standard
- BGP provides each AS a means to:
 = Obtain subnet reachability information from neighbouring Ass
 = Propagate reachability information ro all AS-internal routers
 = Determine "good" routes to subnets based on reachability information and policy
- Allows subnet to advertise its existence to rest of Internet: "I am here"

BGP basics
- pairs of routers (BGP peers) exchange routing info over semi-permanent TCP connections: BGP sessions
 = BGP sessions need not correspond to physical links
- when AS2 advertises a prefix to AS1:
 = AS2 promises it will forward datagrams towards that prefix
 = AS2 can aggregate prefixes in its advertisement

Distributing reachability info
- using eBGP session between 3a and 1c, AS3 sends prefix reachability info to AS1
 = 1c can then use iBGP do distribute new prefix info to all routers in AS1
 = 1b can then re-advertise new reachability info to AS2 over 1b-to-2a eBGP session
- when router learns of new prefix, it creates entry for prefix in its forwarding table



BGP routing policy



- ABC are provider networks
- XWY are customer (or provider networks)
- X is dual-homed: attached to two networks
 = X does not want to route from B via X to C
 = so X will not advertise to B a route to C
- A advertises path AW to B, B advertises path BAW to X
- Should B advertise path BAW to C
 = No, B gets no "revenue" for routing CBAW since neither W nor C are B's customers
 = B wants to force C to route to w via A
 = B wants to route only to/from its customers

Why different Intra- and inter-AS routing?
- Policy:
 = Inter-AS: admin wants control over how its traffic routed, who routes through its net
 = Intra-AS: single admin, so no policy decisions needed
- Scale: hierarchical routing saves table size, reduce update traffic
- Performance:
 = Intra-AS: can focus on performance
 = Inter-ASL policy may dominate over performance

Pure P2P applications
- Can we implement a large-scale network? (as a student or software engineer)
- Challenge: cannot access and configure low-level routers
- Solution: build an application-layer network → P2P

Pure P2P architecture
- no always-on server, arbitrary end systems directly communicate
- peers are intermittently connected and change IP address
- Three topics: file distribution / searching for information / case study: skype

File Distribution: server-Client vs P2P
- how much time to distribute file form one server to N peers?
 = us: server upload bandwidth, ui: peer I upload bandwidth
 = di: peer I download bandwidth
- server sequentially send N copies: NF/us time
 = client i takes F/di time to download
 = Time to distribute F to N clients using client/server approach = dcs
 = max { NF/us, F/min(di)}, increases linearly in N for large N
- in P2p
 = server must send one copy: F/us time
 = client i takes F/di time to download
 = NF bits must be downloaded (aggregate)
 - fastest possible upload rate: us + sum of ui, for all i
 = dP2P = max {F/us, F/min(di), NF/(us + sum of ui)}
- Client upload rate = u, F/u = 1 hour, us = 10u, dmin >= us

File distribution: BitTorrent
- P2P file distribution
 = tracker: tracks peers participation in torrent
 = torrent: group of peers exchanging chunks of a file
- file divided into 256kB chunk
- Peer joining torrent:
 = has no chunks, but will accumulate them over time
 = registers with tracker to get list of peers, connects to subset of peers
- while downloading, peer uploads chunks to other peers
- peers may come and go
- once peer has entire file, it may (selfishly) leave or (altruistically) remain
-Pulling Chunks
 = at any given time, different peers have different subsets of file chunks
 = periodically, a peer (Alice) asks each neighbour for list of chunks that they have
 = Alice sends requests for her missing chunks, rarest first
- Sending Chunks: tit-for-tat
 = Alice sends chunks to four neighbours currently sending her chunks at the highest rate
 - re-evaluate top 4 every 10 secs
 = every 30 secs: randomly select another peer, starts sending chunks
 - newly chosen peer may join top 4, "optimistically unchoke"

BitTorrent: Tit-for-tat
(1) Alice "optimistically unchokes" Bob
(2) Alice becomes one of Bob's top-four providers; Bob reciprocates
(3) Bob becomes one of Alice's top-four providers
 = With higher upload rate, can find better trading partners & get file faster

Distributed Hash Table (DHT)
- DHT = distributed P2P database
- Database has (key, value) pairs; key: content type; value: IP address
- Peers query DB with key, DB returns values that match the key
- Peers can also insert (key, value) peers

DHT Identifiers
- Assign integer identifier to each peer in range 0 ~ 2^n-1
 = each identifier can be represented by n bits
- Require each key to be a n integer in same range
- To get integer keys, hash original key
 = eg: key = h("Led Zeppelin IV")
 = This is why they call it a distributed "hash" table

How to assign keys to peers
- Central issue: Assigning (key, value) pairs to peers
- Rule: assign key to the peer that has the closest ID.
- Convention in lecture: closest is the immediate successor of the key
- EX: n=4; peers: 1, 3, 4, 5, 8, 10, 12, 14
 = key = 13, the successor peer = 14; key = 15, successor peer = 1

Circular DHT
Eg: [1]→[3]→[4]→[5]→[8]→[10]→[12]→[15]→[1]
- Each peer only aware of immediate success or and predecessor
- "Overlay network"
- Eg: [3] want to find [14], [3] will ask who's resp for key 1110, and send to [4], [4] send to [5] … then to [15], [15] answer "I am" and send back to [3]
- O(N) messages on avg to resolve query, when there are N peers

Circular DHT with Shortcuts
Eg: Add [1] → [5] → [12] → [3] → [8] → [15] → [4] → [10] → [1]
- Each peer keep track of IP addresses or predecessor, successor, short cuts
- reduced from 6 to messages ([3] short to [8] then [15])
- Possible to design shortcuts so O(log N) neighbours, O(log N) messages in query

Peer churn
- to handle peer churn, require each peer to know the IP address of its two successors
- Each peer periodically pings its two successors to see if they are still alive
- Eg: peer 5 abruptly leaves
 = peer 4 detects; makes 8 its immediate successor; ask 8 who its immediate successor is; makes 8's immediate successor its second successor

P2P case study:
- inherently P2P: pairs of users communicate
- Proprietary application-layer protocol (inferred via reverse engineering)
- hierarchical overlay with SNs
- Index maps usernames to IP addresses; distributed over SNs

Peers as relays
- Problem when both Alice and Bob are behind NATs
 = NAT prevents an outside peer from initiating a call to insider peer
- Solution:
 = using Alice's and Bob's SNs, Relay is chosen
 = Each peer initiates session with relay
 = Peers can now communicate through NATs via relay

CH 5

Link Layer: Introduction
- Some terminology:
 = Hosts and routers are nodes
 = communication channels that connect adjacent nodes along communication path are links. Eg: wired links, wireless links, LANs
 = layer-2 packet is a frame, encapsulates datagram
- Data-link layer has responsibility of transferring datagram from one node to adjacent node over a link

Where is the link layer implemented?
- In each and every host
- Link layer implemented in "adaptor" (aka network interface card NIC)
 = Ethernet card, PCMCI card, 802.11 card
 = implements link, physical layer
- Attaches into host system buses
- combination of hardware, software, firmware

Adaptors Communication
- sending side:
 = encapsulates datagram in frame
 = add error checking nits, rdt, flow control, etc
- receiving side:
 = looks for errors, rdt, flow control, etc
 = extracts datagram, passes to upper layer at receiving side

MAC addresses and ARP
- 32-bit IP address
 = network-layer address, used to get datagram to destination IP subnet
- MAC (or LAN or physical or Ethernet) address:
 = function: get frame from one interface to another physically-connected interface (same network)
 = 48 bit MAC address (for most LANs), burned in NIC ROM< also sometimes software settable)

LAN Addresses and ARP
- each adapter on LAN has unique LAN address
- Broadcast address = FF-FF-FF-FF-FF-FF
EG: four computer connected to LAN, [1A-2F-BB-76-09-AD], [58-23-D7-FA-20-B0], [0C-C4-11-6F-E3-98], [71-65-F7-2B-08-53]
- MAC address allocation administered by IEEE
- Manufacturer buts portion of MAC address space (to assure uniqueness)
- Analogy: (a) MAC address HKID, (b) IP address: like postal address
- MAC flat address →portability, can move LAN card from one LAN to another
- IP hierarchical address NOT portable
 = address depends on IP subnet to which node is attached
- Question: how to determine MAC address of B knowing B's IP address?
 = Each IP node (host, router) on LAN has ARP table
 = ARP table: IP/<AC address mappings for same LAN nodes
 - <IP address; MAC address RRL>
 = TTL (Time To Live): time after which address mapping will be forgotten (typically 20 min)

ARP protocol: Same LAN (network)
- A wants to send datagram to B and B's MAC address not in A's ARP table
- A broadcasts ARP query packet, containing B's IP address
 = dest MAC address = FF-FF-FF-FF-FF-FF
 = all machines on LAN receive ARP query
- B receives ARP packet, replies to A with its (B's) MAC address
 = frame sent to A's MAC address (unicast)
- A caches (saves) IP-to-MAC address pair in its ARP table until information becomes old (times out)
 = soft state: information that times out (goes away) unless refreshed
- ARP is "plug-and-play"
 = nodes create their ARP tables without intervention net administrator

Addressing: routing to another LAN
EG: A[IP1a, mac1a] and B[IP1b, mac1b] connects to [IP1r, mac1r] net card router R
The same router with another net card [IP2r, mac2r] connect to two computer C&D
- walkthrough: send datagram from A to B via R, assume A knows B's IP address
- two APR tables in router R, one for each IP network (LAN)
- A creates IP datagram with source Am destination B
- A uses ARP to get R's MAC address for IP1r
- A creates link-layer frame with R's MAC address as dest, frame contains A-to-B IP datagram
- A's NIC sends frame, R's NIC receives frame
- R removes IP datagram from Ethernet frame, sees its destined to B
- R uses ARP to get B's MAC address
- R creates frame containing A-to-B IP datagram sends to B

Ethernet Frame Structure
- Ethernet: dominant wired LAN technology
- sending adapter encapsulates IP datagram (or other network layer protocol packet) in Ethernet frame
- Preamble, Dest. Address, Source Address, Type, Data, CRO
- Preamble
 = 7 bytes with pattern 10101010 followed by one byte with pattern 10101011
 = used to synchronize receiver, sender clock rates
- Addresses: 6 Bytes
 = if adapter receives frame with matching destination address, or with broadcast address (eg ARP packet), it passes data in frame to network layer protocol
 = otherwise: adapter discards frame
- Type: indicates higher layer protocol (mostly IP but others possible, eg. Novel IPX)
- CRC: checked at receiver, if error is detected, frame is dropped

Ethernet: unreliable, connectionless
- Connectionless: no handshaking between sending and receiving NICs
- unreliable: receiving NIC doesn't send acks or nacks to sending NIC
 = stream of datagrams passed to network layer can have gaps (missing datagrams)
 = gaps will be filled if app is using TCP
 = otherwise, app will see gaps
- Ethernet's MAC protocol: unslotted CSMA/CD
- Ethernet uses Cyclic Redundancy Check (CRC) code to detect if there is an error
 = Similar to checksum, but much more robust

Hubs
Physical-layer ("dump") repeaters:
- bits coming in one link go out all other links at same rate
- all nodes connected to bub can collide with one another
- no frame buffering, no CSMA/CD at hub: host NICs detect collisions
- Limitations:
 = Insecure (as broadcast all bits to all connected host, receiving all frame, even it is not mine)
 = slow forwarding (host cannot send/rec data when other is sending)

Switch
- link-layer device: smarter than hubs, take active role
 = store, forward Ethernet frames
 = examine incoming frame's MAC address, selectively forward frame to one-or-more outgoing links when frame is to be forwarded on segment, uses CSMA/CD to access segment
- transparent
 = hosts are unaware of presence of switches
- Plug-and-play
 = switches do not need to be configured
- allows multiple simultaneous transmissions
- host have dedicated, direct connection to switch, switches buffer packets
- Ethernet protocol used on each incoming link, but no collisions; cull duplex
 = each link is its own collision domain
- Switching: A-to-A', B-to-B' simultaneously, without collisions
 = not possible with dumb hub

Switch Table
- Q: how does switch know that A' reachable via interface 4, B' reachable via interface 5?
- A: each switch has a switch table, each entry: [MAC address of host, interface to reach host, time stamp], which looks like a routing table
- Q: how are entries created, maintained in switch table?
- Switch learns which hosts can be reached through with interfaces
 = when frame received, switch "learns" location of sender: incoming LAN segment
 = records sender/location pair in switch table

Switch: frame filtering/forwarding
- when frame received:
1. record link associated with sending host (including MAC address, port connected, TTL)
2. index switch table using MAC dest address
3. if entry found for destination then{
  If dest on segment from which frame arrived
   Then drop the frame
   Else forward the frame on interface indicated
  }
 Else flood ← send to all host connected
- forward on all but the interface on which the frame arrived
- frame destination unknown: flood, then target will return to switch and record
- destination A location known: selective send

Interconnecting switches
- switches can be connected together
EG: ABC to S1, DEF to S2, GHI to S3, S123 to S4
- Q: Sending from A to G, how does S1 know to forward frame destined to F via S4 and S3
- A: self learning, work exactly the same as in single-switch case

Switches vs Routers
- both store-and-forward devices
 = routers: network layer devices (examine network layer header)
 = switches maintain switch tables, implement filtering, learning algorithms

Summary
- Link layer: deal with point to point communication over a physical link (could be a wire or wireless medium)
- Link layer addressing, ARP
- Link layer technologies: Ethernet, hub, switch

Past paper 2014-15
Q1: two segments send form sender, first segment is delivered, first ack is lost, no further segments would be lost. Timeout is 4 time units, GBN and SR windows size is 2.

In RDT 3.0

Sender		Receiver
	Segment 0	
	X	
	Segment 0	
	ACK 0	
	Segment 1	
	ACK 1	

In GBN (Go back N)

Sender	Seg 0	Receiver		Sender
	Seg 1		ACK 0(X)	
			ACK 1	

In SR (Selective Repeat)

Sender	Seg 0	Receiver		Sender
	Seg 1		ACK 0(X)	
			ACK 1	
	Seg 0			
			ACK 0	

Q2:
```
1  //client.c
2  int main(void){
3     int fd = socket (…); // create TCP socket
4     char buf[100];
5     …   //set up socket
6     connect(fd,…);   //connect to the server
7     printf("\%d\n", read(fd,buff,100));
8     getchar();
9     close(fd);
10    return 0;
11  }
```

```
1  //server.c
2  int main(void){
3     int afd;
4     int fd=socket(…);
5     …
6     listen(fd, …);
7     afd=accept(fd,…);
8     close(afd);
9     getchar();
10    return 0;
11  }
```

(a) Q: TCp packets are sent before both the users of the client and the server program hits the enter key. Please write the Order, type and the Direction of the packets ///// A: 1 / SYN / Client to server ; 2 / SYN-ACK / Server to client ; 3 / ACK / Client to Server ; 4 / FIN / Server to client ; 5 / ACK / Client to Server
(b) Q: What is the printout in line 7 of client? A: 0, because server send nothing.
(c) Q: How many packet will be send by server if enter press? A: 0 packets
(d) Q: What about client? A: 1 packet, Order:6, Type: FIN, order: client to server

Q3:
(a)Q: Explain why the threshold is set to large in init. A: If the threshold is set too small, TCP sending window side will increase slowly, which needs lots of time to find the real threshold.
(b) Flow starts at FRT, congestion window is 1 MSS, slow start threshold starts with the number 1000, if a packet is loss, it will reduce to 1 MSS. 1,2,4,8,16(loss)
(i) Q: which of the TCP have no fast recovery? A: TCP Tahoe
(ii) If no further packet loss or fast retransmission detected, the value of the congestion window is 8, the slow start threshold in the 10 round trip is 9
(iii) if use another TCP (Reno), 1,2,4,8,16(loss),8,9,10,11,12; slow start th:8

Q4: Private network with four computer C1,2,3,4, IP: 192.168.127.21,22,23,24
(a) The subnet address cover all C1 to 4 with the smallest possible size is 192.168.127.16/28 (samebit / size of same bit)
(b) subnet address cover C1 to C3 , with largest possible size: 192.168.127.16/29

Q5:
A ←1→ C ←3→D←8→B←2→A
(a) Step 1

A B C D	A B C D	A B C D	A B C D
A 0 2 1 i	A i i i i	A i i i i	A i i i i
B i i i i	B 2 0 i 8	B i i i i	B i i i i
C i i i i	C i i i i	C 1 i 0 3	C i i i i
D i i i i	D i i i i	D i i i i	D i 8 3 0

Step 2

A B C D	A B C D	A B C D	A B C D
A 0 2 1 4	A 0 2 1 i	A i i i i	A 0 2 1 i
B 2 0 i 8	B 2 0 3 8	B 2 0 i 8	B i i i i
C 1 i 0 3	C i i i i	C 1 3 0 3	C 1 i 0 3
D i i i i	D i 8 3 0	D i 8 3 0	D 4 8 3 0

Step 2

A B C D	A B C D	A B C D	A B C D
A 0 2 1 4	A 0 2 1 4	A i i i i	A 0 2 1 4
B 2 0 3 8	B 2 0 3 8	B 2 0 3 8	B i i i i
C 1 3 0 3	C i i i i	C 1 3 0 3	C 1 3 0 3
D i i i i	D 4 8 3 0	D 4 8 3 0	D 4 8 3 0

(b) Distance vector algo is "asynchronous" as each local iteration caused by local link cost change and DV update message form neighbour
(c) How poisoned reserve solve this problem? Ref to 4.2

Past paper 2013-14
Question 1
(a) UDP is best-effort, as it is not reliable and no flow control.
(b) Link-state algo is centralized routing algo, as it needs to know the whole network. Distance-vector is decentralized.
(c) UDP to port 53 is used to search DNS, and the 3 packets using TCP is 3-way-hand-shack
(e) When the program is not in a passive state and SYN received RST will be sent