1 Computer Networks and the Internet

1.1 What is the Internet?

- Internet is a computer network that connects billions of computing devices.
- Devices connected to the internet are called **hosts** or **end systems**.
  - Hosts can be further split into **clients** and **servers**.
- Hosts are connected through **communication links** (e.g. fiber, copper, radio, satellite) or **packet switches** (e.g. routers and switches)

1.2 Network Edge

1.2.1 Access Network

- Network that physically connects an end system to the first router
- They are what end-users connect to if they want to access the Internet

1.3 Network Core

Network Core is the mesh of interconnected routers that links the Internet's end systems (Runs low level protocols)

1.3.1 Packet Switching

Data sent through the net in discrete chunks where each **packet** is transmitted at full link capacity (transmission rate / bandwidth)

- **Transmission rate (R)** = # of bits transmitted into the link per second
- **Transmission delay** = Time needed to transmit L-bit packet into link = $\frac{L(bits)}{R(bits/sec)}$
- **Store-and-forward**: entire packet must arrive at a router before it can be transmitted to the next link.
- **Queueing and loss:**
  - packets enters a queue if arrival rate > transmission rate
  - packets can be dropped if buffer (queue) fills up
- **Forwarding Tables and Routing Protocols:** Routers use these to determine the link it should forward the packet to.
- **Benefits:** Message segmentation **reduces delay**, **allow for retransmission** of smaller parts in case of corruption, and are **easier to buffer + queue** behind (more fair for other packets).
- **Drawbacks**: Packet needs to be re-arranged at destination and there is more overhead

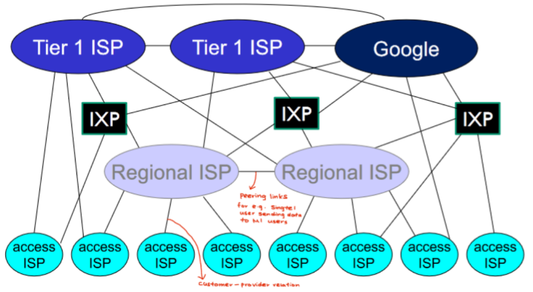
1.3.2 Circuit switching

Dedicated circuit per call

- end-end resources reserved for call between source & dest
- call setup required
- circuit-like (**guaranteed**) performance
- circuit segment idle if not used (no sharing)
- commonly used in telephone networks

1.3.3 Internet Structure

- End systems access the Internet through **access Internet Service Providers (ISPs)**
- Access ISPs must be interconnected so that any 2 hosts can send packet to each other
- Results in a network of network:
  - **Regional ISP:** ISPs that access nets in the region connect to.
  - **Tier-1 commercial ISP:** Provides national and international coverage. Regional ISPs connect to Tier-1 ISPs (e.g. Sprint, AT&T)
  - **Internet Exchange Point (IXP):** Allows multiple Tier-1 ISPs to interconnect with each other
    - ⋆ Peering: bilateral. more expensive but much faster
    - ⋆ IXP: multilateral, cheaper but slower
  - **Content-Provider Networks:** Private network own by company that connects their own data centres to the Internet, often bypassing tier-1 and regional ISP (e.g. Google)
- If we were to connect all N devices in a network, we would need $N \times (N-1)/2$ links


Tier 1 ISP
Tier 1 ISP
Google
IXP
IXP
IXP
Regional ISP
Regional ISP
Peering links for e.g. Singtel user sending data to M1 users
access ISP
customer – provider relation

1.4 Delay, Loss and Throughput in Networks

1.4.1 Packet Loss

As the queue for router has finite capacity, a router will **drop** a packet if the queue is full and the packet will be lost. It **can be retransmitted** by previous node, source end system or not at all.

1.4.2 4 Sources of Packet Delay

- **Nodal Processing (d_{proc}):** check bit errors, determine output link, typically in microseconds or less
- **Queueing (d_{queue}):** time waiting in output link for transmission, depends on congestion level of router. Typically in microseconds to milliseconds.
- **Transmission ($d_{trans} = \frac{L}{R}$):** L = packet length (bits), R = link bandwidth (bps), time taken to push data onto link
- **Propagation ($d_{prop} = \frac{d}{s}$):** d = length of physical link, s = propagation speed in medium ($2 \times 10^8 m/sec$), time taken to travel through physical link
- For $N-1$ routers between source host and destination host, we have:

$$d_{end-to-end} = N(= d_{proc} + d_{queue} + d_{trans} + d_{prop})$$

1.4.3 Throughput

- Rate (bits/unit time) at which bits are transferred between sender/receiver
  - **average:** rate over longer period of time (note that size of file does not matter in this case!)
  - **instantaneous:** rate at given point in time

1.5 Protocol Layers and Service Models

Protocols: Define format, order of messages sent and received among network entities, and actions taken on message transmission, receipt

To provide structure to the design of network protocols, network designers organise protocols in layers. We are interested in the services that a layer offers to the **layer above**, i.e. the **service model** of a layer.

1.5.1 Why layer?

- explicit structure **allows identification**
- modularization **eases maintenance**, updating of system
  - change in implementation of a layer's service is transparent to the rest of the system, i.e. changing one layer **does not** affect the rest of the layers

1.5.2 5 Layers

- **Application**: Where network applications and their application-layer protocols reside. A packet of information at this layer is called a **message**. Examples: FTP, SMTP, HTTP
- **Transport**: Transports application-layer messages between application endpoints. A transport-layer packet is called a **segment**. Examples: TCP, UDP
- **Network**: routing of **datagrams** from source to destination. Examples: IP, routing protocols
- **Link**: Data transfer of **frames** between neighbouring network elements (nodes). Examples: Ethernet, 802.11 (WiFi), PPP
- **Physical**: bits "on the wire". Link dependent and transmission medium-dependent

2 Application Layer

Goal: Write programs that run on different end systems, communicate over the network (e.g. web server software ↔ browser software)
Note that network-core devices (routers) have **no application or even transport layer** which allows for rapid application development.

2.1 Principles of Network Applications

2.1.1 Client-Server Architecture

- **Server**: always-on, waits for incoming requests, provides requested service to client, data centres for scaling, has **permanent IP address**
- **Client**: initiates contact with server, typically requests service from server, may have **dynamic IP addresses**, do not communicate with each other, may be intermittently connected

2.1.2 Peer-to-Peer (P2P)

- No always-on server
- Arbitrary **end systems directly communicate**
- Peers request and provide services to other peers
- **Self-scalability**: new peers bring new service capacity, as well as new service demands
- Peers are intermittently connected and change IP addresses (lead to complex management)

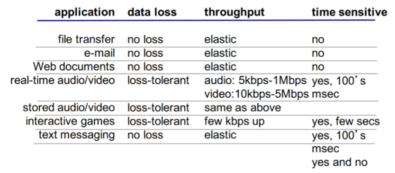
2.2 Process Communication

2.2.1 Sockets

- A **software interface** (between application & transport layer) that a process uses to send messages into, and receive messages from the network. Uses **IP address and a port number** as identifier.
- **IP Address:** 32-bit Integer that uniquely identifies a host
- **Port Number:** 16-bit integer that identify what services a process is using (e.g. HTTP Server: 80, mail server: 25, DNS: 53)

2.2.2 Transport Services

- **Data integrity**: 100% reliable (e.g. file transfers, web transactions) vs tolerant to data loss (e.g. audio, video stream, multimedia)
- **Timing**: some apps require low delay (latency) to be "effective" (e.g. interactive games, zoom)
- **Throughput:** require minimum amount of throughput to be "effective" (e.g. gaming)
- **Security:** encryption scheme, data integrity

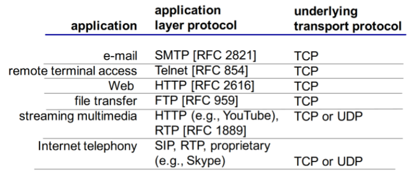


application	data loss	throughput	time sensitive
file transfer	no loss	elastic	no
e-mail	no loss	elastic	no
Web documents	no loss	elastic	no
real-time audio/video	loss-tolerant	audio: 5kbps-1Mbps video:10kbps-5Mbps	yes, 100's msec
stored audio/video	loss-tolerant	same as above	
interactive games	loss-tolerant	few kbps up	yes, few secs
text messaging	no loss	elastic	yes, 100's msec yes and no

2.2.3 Transport-Layer Protocols

TCP	UDP
Reliable data transfer	**Unreliable** data transfer
Flow control: sender won't overwhelm receiver	**No flow control**
Congestion control: throttle sender when network is overloaded	**No congestion control**
Connection-oriented: setup required between client and server	**No setup** needed
Does not provide: timing, minimum throughput guarantee, security	**Does not provide**: timing, throughput guarantee, security

- Why UDP?
  - UDP has less overhead and complexity and has its uses in multimedia where data integrity is not as important and delay might be shorter



application	application layer protocol	underlying transport protocol
e-mail	SMTP [RFC 2821]	TCP
remote terminal access	Telnet [RFC 854]	TCP
Web	HTTP [RFC 2616]	TCP
file transfer	FTP [RFC 959]	TCP
streaming multimedia	HTTP (e.g., YouTube), RTP [RFC 1889]	TCP or UDP
Internet telephony	SIP, RTP, proprietary (e.g., Skype)	TCP or UDP

2.2.4 Definition of App-layer Protocols

- **Types of Messages exchanged,** e.g. request, response
- **Message syntax**, e.g. message fields and how they are delineated
- **Message semantics**: meaning of information in fields
- **Rules** for when and how application send and respond to messages

There are also open protocols defined in RFCs (Request for Comments) such as HTTP (allows for interoperability) and some proprietary protocols (e.g. Skype)

2.3 Web and HTTP

2.3.1 Web Page

- consists of a base HTML file and several referenced objects
- objects can be HTML file, JPEG image, Java applet, audio file etc.
- object is addressable by a URL:
  - **Hostname**: www.comp.nus.edu.sg
  - **Path name**: / ~cs2105/img/lect1.pdf

2.3.2 HTTP

- **HyperText Transfer Protocol** is the Web's **application layer** protocol
- **Client/server model**: Client is the browser that requets, receives and "displays" web objects whereas the server is a Web Server that sends objects in response to requests
- **Over TCP** (Reliable!)
- **Stateless**: server maintains no information about past client requests

2.3.3 Persistent HTTP

- **Multiple objects** can be sent **over single TCP** connection
- TCP connection is left open after sending response
- **Persistent with pipelining**: client may send requests as soon as it encounters a referenced object – as little as 1RTT for all referenced objects.

2.3.4 Non-Persistent HTTP

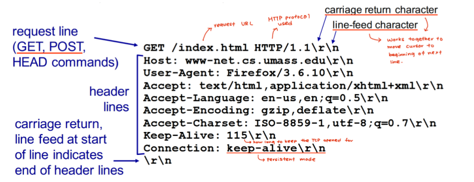
- At most 1 object sent over a TCP connection (TCP connection is closed after each request)
- downloading multiple objects requires multiple connections
- Requires 2 RTTs per object (1 to initiate TCP connection, 1 to request the file), and 2 RTTs get the initial HTML file
- Response time = $(2 \times RTT +$ file transmission time) / object
- incurs **overhead** for each TCP connection → solution is to **open parallel TCP connections**

* **RTT**: time for a small packet to travel from client to server and back (does not include transmission delay!)

$$RTT = d_{proc} + d_{queue} + d_{prop}$$

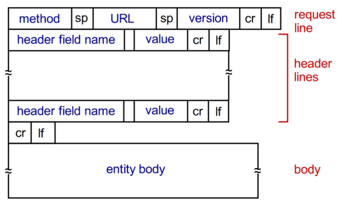
2.3.5 HTTP Request Format

* Note that it is **not possible to find IP address** of Host in the HTTP header



```
GET /index.html HTTP/1.1\r\n
Host: www-net.cs.umass.edu\r\n
User-Agent: Firefox/3.6.10\r\n
Accept: text/html,application/xhtml+xml\r\n
Accept-Language: en-us,en;q=0.5\r\n
Accept-Encoding: gzip,deflate\r\n
Accept-Charset: ISO-8859-1,utf-8;q=0.7\r\n
Keep-Alive: 115\r\n
Connection: keep-alive\r\n
\r\n
```

request line (GET, POST, HEAD commands)
header lines
carriage return, line feed at start of line indicates end of header lines
request URL
HTTP protocol used
carriage return character
line-feed character
works together to move cursor to beginning of next line.
how long to keep the TCP opened for
persistent mode



2.3.6 HTTP Method Types

- **HTTP/1.0** (Default to Non-Persistent):
  - GET: Gets an object
  - POST: Requests web server to put data in body of message, typically used for sending form data
  - HEAD: asks server to leave requested object out of response (mostly for debugging)
- **HTTP/1.1** (Default to Persistent):
  - GET, POST, HEAD
  - PUT: uploads file in entity body to path specified in URL
  - DELETE: deletes file specified in URL

2.3.7 HTTP Response Status Codes

- `200 OK`: Request succeeded, requested object later in this msg
- `301 Moved Permanently`: requested object moved, new location specified later in this msg (Location:)
- `400 Bad Request`: request msg not understood by server
- `404 Not Found`: requested document not found on this server
- `500 HTTP Version Not Supported`

2.4 Cookies

As HTTP is stateless, we need cookies to help carry states. Cookies has four components:

1. cookie header line of HTTP response message
2. cookie header line in next HTTP request message
3. cookie file is stored on user's host and is managed by user's browser
4. back-end database at website

Cookies are typically used for: authorization, shopping carts, recommendations, user session state (Web email). The state is maintained at sender/receiver over **multiple transactions**

2.4.1 Conditional GET

Web heavily **utilises caching to reduce response time** by caching objects in a proxy server (typically much closer to clients). If a request object is in the proxy server, cache returns the object else it will request object from origin server. (lowers link utilisation)

Since there is a cache, we must handle and prevent transferring stale objects. This is done through a `If-Modified-Since: <data>` header line which returns a `200 OK` and returns new object if data has been modified, else `304 Not Modified` and no object if data is not modified.

2.5 DNS: The Internet's Directory Service

The **Domain Name System** helps to map hostnames (e.g. google.com) to their IP address and vice versa so we as humans do not have to remember individual IP addresses for websites. (Default port 53)

- **Distributed, Hierarchical Database** in the Application Layer
- **Root Server**: answers requests for records in the root zone by returning a list of the authoritative name servers for the appropriate TLD
- **DNS Caching**:
  - cache entries timeout after some time (**TTL** - Time to live)
  - TLD servers are typically cached in local name servers → root name servers are often not visited
  - cached-entries **may be out of date** → if a host changes IP address, it may not be known until TTL expires
- Runs over **UDP**

2.5.1 Local DNS name server

- when host makes DNS query, first sent to the Local DNS server
- contains local cache of recent name-to-address translation pairs (but **might be out of date**)
- acts as proxy to forward query into hierarchy
- **does not strictly belong to hierarchy**

2.5.2 Root name server

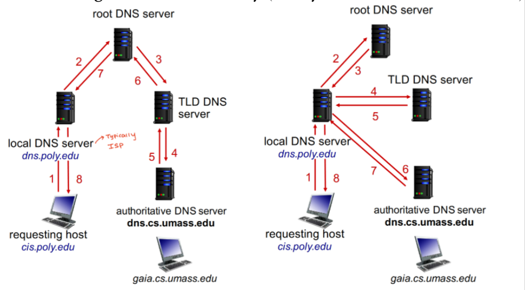
- contacted by local name server that cannot resolve name
- provides IP address of TLD (Top Level Domain e.g. .com, .gov, .sg) servers

2.5.3 Authoritative servers

- organization's owned DNS servers that provides authoritative hostname to IP mappings for organization's named hosts
- maintained either by organization or service providers

2.5.4 DNS Name Resolution

- **Iterative query**: Local DNS server makes DNS requests one by one in the hierarchy
- **Recursive query** (rarely used): each server in the hierarchy asks one server higher in the hierarchy (heavy load on root DNS server)



(a) Recursive (b) Iterative

3 Socket Programming

Socket: the software interface between app processes and transport layer. Allows for communication between processes over the Internet.

Relies on:
1. **Port Number**: 16-bit unsigned integer (0-1023 are reserved)
2. **IP Address**: 32-bit unsigned integer

3.1 UDP (User Datagram Protocol)

- **unreliable** datagram, **connectionless**
- no handshaking before sending data
- sender (client) explicitly attaches destination IP address and port number to each packet
- receiver (server) extracts sender IP address and port number from the received packet
- transmitted data **may be lost** or **received out-of-order**

* For clients, port number can be random whereas server port number must be assigned

3.2 TCP (Transmission Control Protocol)

- reliable **byte stream**, connection-oriented
- When client creates socket, client TCP establishes a connection to server TCP. (Server must have "welcome socket")
- When contacted by client, server TCP creates a new socket (with same port number) for server process to communicate with that client
- Allows server to talk with multiple clients individually.
- Communicates as if there is a pipe between 2 processes, sending process doesn't need to attach a destination IP address and port number in each sending attempt.

UDP	TCP
Server uses one socket to serve all clients (n clients -> 1 socket)	Server creates a new socket for each client (n clients -> n+1 sockets)
No connection is established before sending data	Client establishes connection to server
Sender explicitly attaches destination IP address + port#	Server uses connection to identify client
Unreliable datagram: Data may be lost, received out-of-order	Reliable stream pipe: data guaranteed to be received in order

4 UDP, Reliable Data Transfer

4.1 Transport-layer Services and Protocols

- provides **logical communication** (allows processes to feel like they are directly connected to each other) between app processes running on different hosts
- **Sender**: Breaks app messages into segments, passes them to network layer
- **Receiver**: Reassembles segments into message, passes it to app layer
- Primarily uses 2 protocols:
  1. TCP (reliable, more overhead, slower)
  2. UDP (unreliable, less overhead, faster)

4.2 Transport vs Network layer

- Transport layer: logical communication between processes
  - relies on and enhances network layer services
- network layer: logical communications between hosts/interfaces
  - unreliable "best-effort" (relies on TCP for reliability)

4.3 Connectionless Transport: UDP

- UDP adds very little on top of IP
  - **Multiplexing** at sender (aggregate packets that needs to be transported out)
  - **Demultiplexing** at receiver (determines which packets are for which processes)
  - **Checksum**
- UDP transmission is unreliable, often used by streaming multimedia app (loss tolerant & rate sensitive apps)

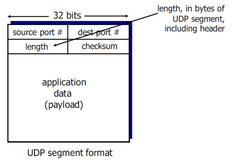
4.3.1 Why UDP?

No connection establishment (less delay), simple (no connection state at sender, receiver), small header size, no congestion control (UDP can send data without worrying about data loss)

4.3.2 Connectionless De-multiplexing

When UDP receiver receives a UDP segment:
- Check destination port number in segment, and direct that segment to the socket with that port number.
- basically same dest. port # → same socket at destination

4.3.3 UDP Header


32 bits
length, in bytes of UDP segment, including header
source port #
dest port #
length
checksum
application data (payload)
UDP segment format

* Note that the minimum length of UDP segment is 8 bytes which is just the header itself

4.3.4 UDP Checksum

1. Treat UDP segment as sequence of 16-bit integers
2. Apply binary addition
3. Add carry to result (if any)
4. Compute 1's complement to get the UDP checksum

Result of adding checksum and the original sum should give all 1's. Also note that UDP checksum is done in complement with other checksums (in network layer) to ensure reliable data transfer (passing UDP checksum alone is not guaranteed that no bit error occurred)

4.4 Principles of Reliable Data Transport

To build a reliable transport layer protocol, we have to handle:
- Packet corruption
- Packet loss
- Packet reordering
- Packet delay (arbitrarily long)

4.4.1 RDT 1.0

Assumption: Underlying channel is perfectly reliable (no bit errors and no loss of packets).
1. (Sender) Make packet and send packet to receiver using network layer.
2. (Receiver) Extract packet from network layer and deliver data to application.

4.4.2 RDT2.0

Assumption: Underlying channel may flip bits i.e. corruption. We use the stop and wait protocol (server send one packet at a time then wait for recv response).
1. (Sender) Make packet and send packet to receiver.
2. (Receiver) Check if packet is corrupt. If not corrupt, deliver data to application and send ACK to sender, else just send NAK to sender.
3. (Sender) If received NAK, resend the same packet, else go to step 1 for next packet.

Problems: If ACK or NACK are corrupted, there is no guaranteed way to recover. If we simply resend the packet, the receiver will not know it's a duplicate.

4.4.3 RDT2.1

Assumption: Same as rdt 2.0 - corruption. In addition to what is covered in rdt 2.0, we now add a sequence number to the packet. This number alternates between 1 and 0. Duplicates are detected using sequence number.
1. (Sender) Sends pkt0.
2. (Receiver) Sends ACK or NAK.
3. (Sender) If NAK or corrupted message received, retransmit message. Else send pkt1.
4. (Receiver) If duplicate pkt0 is received, drop it and reply ACK. Else continue as per usual.

4.4.4 RDT2.2

Assumption: Same as rdt 2.0 and 2.1 - corruption. In addition to what is covered in rdt 2.1, we now explicitly include the sequence number of the packet being acknowledged, removing the need for a NAK. From the sender's perspective, we basically resend current packet if a duplicate ACK is received.
1. (Sender) Sends pkt0.
2. (Receiver) Sends ack1 if corrupted, else ack0.
3. (Sender) So long as ack# is different from packet seq# sent earlier or corrupted ACK, keep resending that packet.
4. (Receiver) If duplicate pkt0 is received, drop it and reply ack. Else continue as per usual.

4.4.5 RDT3.0

Assumption: Corruption, packet loss and packet delays, but no reordering (i.e. the order sent is the order received). We need to add sender **timeouts** to rdt 2.2 to handle packet loss and delays. One important difference is that duplicate ACKs are ignored. To detect packet loss, the timer is used.
1. (Sender) Sends pkt0.
2. (Receiver) If received and pkt# correct, send ack0. Else if received and is corrupted, send ack1.
3. (Sender) If no response is heard by a certain time (either due to delay or loss on either side), resend pkt0. Else if ack1 received, also resend pkt0. Only if ack0 is received do we move on to next packet. (Corrupt ACKs ignored)
4. (Receiver) Same as step 2, except if packet 1 has been received before and it's a duplicate, we also send ACK1.
5. (Sender) If repeated ack is received, it will ignore the ack (i.e. it will not send pkt again)

4.4.6 Performance of RDT3.0

Performance of stop-and-wait protocol is very bad!
e.g. if 1 Gbps link, $D_{prop} = 15ms$, 8000 bit packet:

$$D_{trans} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits / s}} = 8 \text{ microseconds}$$

$$U_{sender} = \frac{L/R}{RTT + L/R} = \frac{.008}{30.008} = 0.0027\%$$

if RTT=30 msec, 1KB pkt every 30 msec: 33kB/sec throughput over 1 Gbps link → network protocol limits resources!

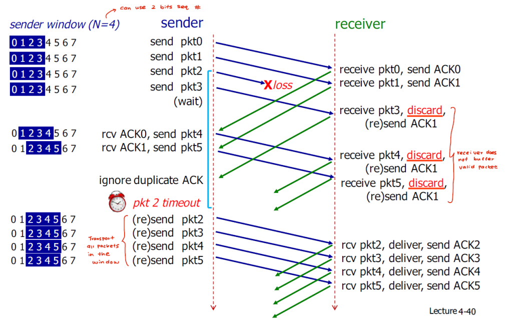
4.5 Pipelining

Generally, N-packet pipe-lining will increase utilization by a factor of N. To allow for pipelining, we have to: increase range of sequence number and buffer at sender and/or receiver

4.5.1 Go-back-N

Pros: only 1 timer needed for oldest in-flight pkt, only need to remember `ExpectedSeqNum`, cumulativeACK → if ACK lost, just take highest ACK# and carry on from there

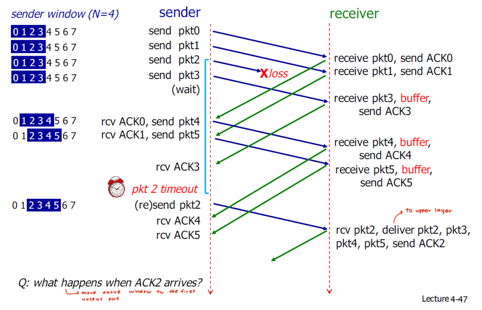
Cons: If one packet loss halfway and the rest is received, everything that is received will be discarded → not efficient use of resources



4.5.2 Selective Repeat

Pros: Only retransmits unACKed pkts and buffer out-of-order pkts → more efficient use of resources, no need to retransmit so many pkt

Cons: much more complex since need to maintain multiple timers for all the packets in the window



5 Connection-oriented Transport: TCP

- **Point-to-point**: 1 sender, 1 receiver
- **Connection-oriented**: handshake before data exchange
- **Full duplex data**: bi-directional data flow in same connection
- **Reliable, in-order byte stream**: no "message boundaries" (i.e. format of data sent does not matter → whether its a .jpg or .txt file, both is seen as stream of bytes)
- **Pipelined**: **dynamic** window size set by congestion/flow control

5.1 TCP: buffers and Segments

- two buffers: send and receive, created after handshaking at both sides. Buffers are regarded as a stream of bytes
- **Max Segment Size (MSS)**: typically 1460 bytes, max app-layer data one TCP segment can carry.
- Limited by maximum transmission unit (MTU), largest link-layer frame e.g. 1500 bytes for Ethernet

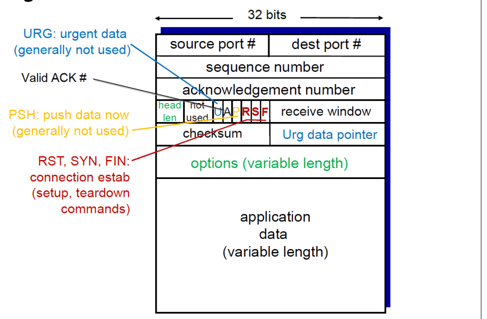
* Why 1460 Bytes specifically? Max = 1500 bytes, 20 bytes for IP header, 20 bytes for TCP header

5.2 Connection-oriented de-multiplexing

A TCP connection/socket is identified by 4-tuple (`srcIPAddr, srcPort, destIPAddr, destPort`). If there are multiple connections to same destination port and address number, with different source IP and/or source port#, server demultiplex connections to different sockets.

* Note that server is usually threaded so that it can handle multiple connections

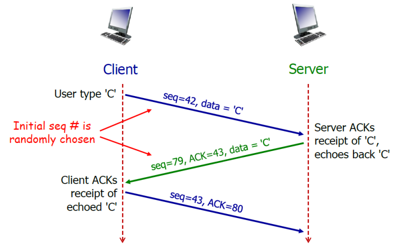
5.3 TCP Segment


32 bits
URG: urgent data (generally not used)
Valid ACK #
PSH: push data now (generally not used)
RST, SYN, FIN: connection estab (setup, teardown commands)
source port #
dest port #
sequence number
acknowledgement number
head len
not used
C E U A P R S F
receive window
checksum
Urg data pointer
options (variable length)
application data (variable length)

- **Sequence Number**: byte number of the first byte of data in a segment (if it exceeds $2^{32}-1$ then it will wrap around back to 0).
  - Seq# of sender and receiver is different and is randomly picked at the start of the connection (to prevent confusion during packet reorder)
  - if we have a file of 500,000 bytes, MSS = 1000 bytes, then SEQ# will be 1000, 2000, ...
- **ACK number**: sequence number of the next byte of data expected by the receiver. Uses cumulative ACK (similar to go-back-N). ACK# = client_seq# + len(data)
  - Allows for **piggy-backing**: can send data along with ACK. ACK will be processed as long as ACK bit is 1.
  - **Delayed ACK**: Receiver can wait up to 500ms for the next segment before sending an ACK (saves 1 cycle of ACK)
- ACK bit: indicates whether segment includes an ACK and whether acknowledgement # is valid. If ACK bit is 0, indicates that segment is a pure data segment
- SYN bit: used to signal a request for connection setup
- FIN bit: used to signal connection tear down
- **Receive Window**: # of bytes rcvr willing to accept

* Note that TCP specifications doesn't say how to handle out-of-order segments, but it is usually buffered in practice (similar to selective repeat)

5.3.1 Example of transmission of TCP segment


Client
Server
User type 'C'
seq=42, data = 'C'
Initial seq # is randomly chosen
Server ACKs receipt of 'C', echoes back 'C'
seq=79, ACK=43, data = 'C'
Client ACKs receipt of echoed 'C'
seq=43, ACK=80
TCP Echo Server

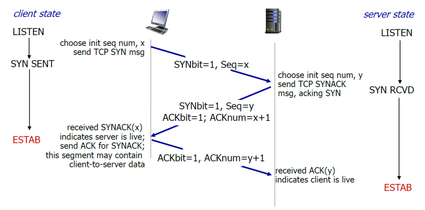
5.4 TCP Flow Control

Flow control allows rcvr to control sender, making sender dynamically change the window size so that rcvr's buffer won't overflow.
Steps involved:

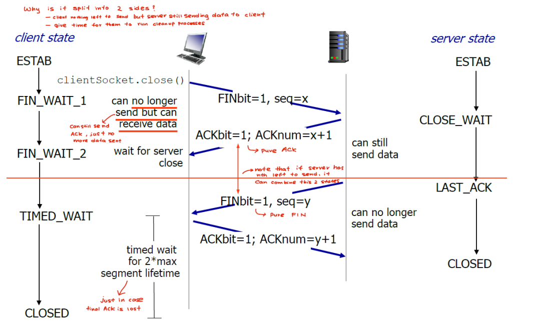
1. receiver "advertises" free buffer space (receive window in header (rwnd), typically defaults to 4096 bytes)
2. sender limits amount of unacked data to rcvr's rwnd value
3. this mechanism will hence guarantee rcvr buffer will not overflow and prevents data loss due to buffer overflow

5.5 TCP 3-way handshake

5.5.1 TCP Set-up

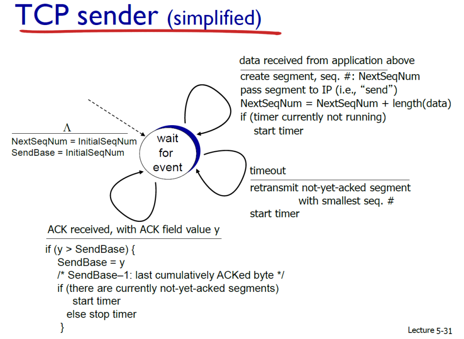

client state
server state
LISTEN
LISTEN
choose init seq num, x send TCP SYN msg
SYN SENT
SYNbit=1, Seq=x
choose init seq num, y send TCP SYNACK msg, acking SYN
SYN RCVD
SYNbit=1, Seq=y ACKbit=1; ACKnum=x+1
ESTAB
received SYNACK(x) indicates server is live; send ACK for SYNACK; this segment may contain client-to-server data
ACKbit=1, ACKnum=y+1
received ACK(y) indicates client is live
ESTAB

5.5.2 TCP Tear-down


Why is it split into 2 sides?
– client nothing left to send but server still sending data to client
– give time for them to run cleanup processes
client state
server state
ESTAB
ESTAB
clientSocket.close()
FIN_WAIT_1
can no longer send but can receive data
can still send ACK, just no more data sent
FINbit=1, seq=x
CLOSE_WAIT
ACKbit=1; ACKnum=x+1
pure ACK
FIN_WAIT_2
wait for server close
can still send data
note that if server has nth left to send, it can combine this 2 states
LAST_ACK
FINbit=1, seq=y
pure FIN
TIMED_WAIT
can no longer send data
ACKbit=1; ACKnum=y+1
timed wait for 2*max segment lifetime
just in case final Ack is lost
CLOSED
CLOSED

5.6 TCP RDT

- TCP creates rdt service on top of IP's udt service: pipelined segments, cumulative acks, single retransmission timer, checksums
- Retransmissions triggered by: timeouts, duplicate ACKs


TCP sender (simplified)
wait for event
Λ
NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum
data received from application above
timeout
ACK received, with ACK field value y
Lecture 5-31

```
data received from application above
create segment, seq. #: NextSeqNum
pass segment to IP (i.e., "send")
NextSeqNum = NextSeqNum + length(data)
if (timer currently not running)
    start timer

timeout
retransmit not-yet-acked segment
          with smallest seq. #
start timer

ACK received, with ACK field value y
if (y > SendBase) {
   SendBase = y
   /* SendBase–1: last cumulatively ACKed byte */
   if (there are currently not-yet-acked segments)
      start timer
    else stop timer
  }
```

5.6.1 TCP ACK Generation

event at receiver	*TCP receiver action*	3 4 5
arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed	delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK	🟦 ⬜ ⬜
arrival of in-order segment with expected seq #. One other segment has ACK pending	immediately send single cumulative ACK, ACKing both in-order segments	🟥 🟦 ⬜
arrival of out-of-order segment higher-than-expect seq. # . Gap detected	immediately send *duplicate ACK,* indicating seq. # of next expected byte	⬜ ⬜ 🟦
arrival of segment that partially or completely fills gap	immediate send ACK, provided that segment starts at lower end of gap	🟦 ⬜ 🟥

5.7 TCP Timeout

Considerations:

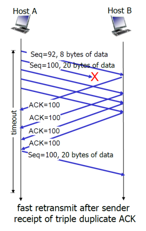
- Must be longer than RTT but RTT varies
- too short: premature timeout, unnecessary retransmissions
- too long: slow reaction to segment loss

Important equations:

$$EstimatedRTT = (1-\alpha) \times EstimatedRTT + \alpha \times SampleRTT$$
$$DevRTT = (1-\beta) * DevRTT + \beta * |SampleRTT - EstimatedRTT|$$
$$TimeoutInterval = EstimatedRTT + 4 * DevRTT$$

- exponential weighted moving average
- influence of past sample decreases exponentially fast
- typical value of $\alpha = 1/8$
- typically $\beta = 1/4$

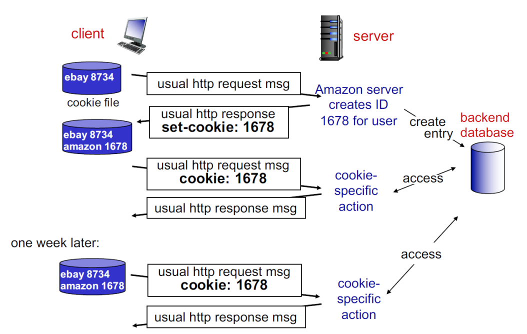
5.8 TCP fast retransmit



fast retransmit after sender receipt of triple duplicate ACK

6 Miscellaneous

Cookies Illustration

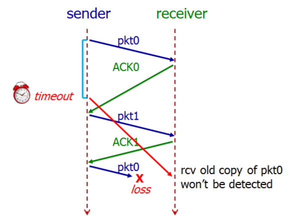


RDT Summary

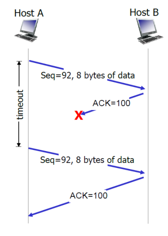
rdt	What gets corrupted?	Checksum?	Sequence Number?	Can packets be lost?
1.0	Perfectly Reliable	No	No	No
2.0	Packet	Yes	No	No
2.1	Packet, ACK and NAK	Yes	Yes	No
2.2	Packet, ACK	Yes	Yes	No
3.0	Packets, ACK	Yes	Yes	Yes

Packet Reordering Issues using RDT3.0

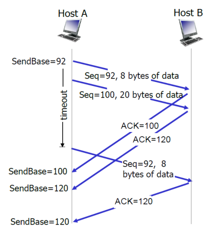
Solution is to increase seq# to hold more than 2 bits; use TTL to limit the lifespan of packets so that by the time seq# is reused, TTL of old packet expires; Use random init seq#



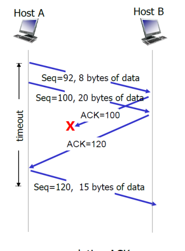
TCP Retransmission Scenarios



lost ACK scenario



premature timeout



cumulative ACK

Pipe-lining Summary

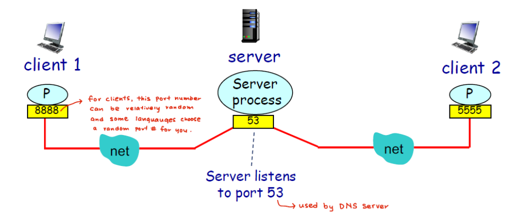
Go-back-N:

- Receiver
  - doesn't buffer or ack out-of-order pkts
  - only sends *cumulative ack* for the most-recent pkt
- Sender
  - has timer for *oldest* unacked packet
  - when timer expires, retransmit all unacked pkts

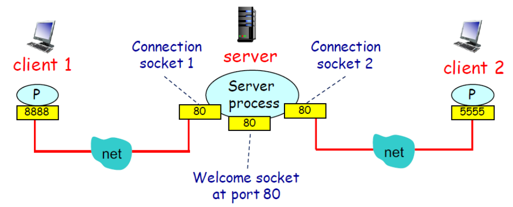
Selective Repeat:

- Receiver
  - maintain a *sliding window to buffer* out-of-order pkts
  - sends individual ack for each packet
- Sender
  - maintains timer for each unacked packet
  - when timer expires, retransmit *only* that unacked pkt

UDP Socket Illustration


client 1
server
client 2
P
8888
for clients, this port number can be relatively random and some languages choose a random port # for you.
Server process
53
P
5555
net
net
Server listens to port 53
used by DNS server

TCP Socket Illustration



Throughput vs Latency

- **Latency**: how much time needed for information to be sent across to end user. (e.g. if someone on zoom speaks, how long does it take for other people to hear it?)
- Low latency important for live streaming service (e.g. Zoom, Twitch)
- **Throughput**: amount of data received in a given time period (e.g. When streaming Netflix, you need a throughput of say 5Mbps)
- Throughput does not account for delay (e.g. when streaming Netflix, you can allow for it to buffer for a few minutes, don't have to be live)