EECS325 Computer Networks

Yue Shu
Spring 2019

        - Packet Switching VS Circuit Switching
            - Circuit Switching
            - Packet Switching
    - Internet Structure: network of networks
    - Delay, loss, and throughput in networks
        - Process Delay
        - Transmission Delay
        - Queueing Delay
        - Propogation Delay
        - Real internet delays and routes
        - Packet Loss
        - Throughput
- January 22, Tuesday
    - Protocol Layers
        - Internet Protocol Stack: End to End Protocol
        - ISO/OSI reference model
        - Process of data transfer between end users
    - Network Security
        - Malware
        - Denial of Service (DoS)
        - Packet "Sniffing"
        - IP spoofing
- Jan 24, Thursday
- Chapter Two: Application Layer
    - Application Architecture
        - Client-Server Architecture
        - P2P architecture
        - Sockets
        - Addressing processes
        - App-layer protocol defines
        - What transport services the app required
        - Internet transport protocols services
    - Web and HTTP
        - HTTP connections
            - Non persistant HTTP connections
            - Persistant HTTP connections
        - HTTP request message
        - HTTP response message
            - Response status codes
- Jan 29, Tuesday

    - User-server state: stateful cookies
    - Web Caches (proxy server)
    - Reverse Proxy
    - Caching Example
        - Without web cache mechanism
        - Increase access link capacity without web cache
        - Install local web cache server
    - Conditional GET
    - Electronic Mail
        - SMTP protocol
        - Mail message format
        - Message access Protocols
            - POP3 protocol
            - IMAP
            - HTTP
- Feb 5, Tuesday
    - DNS (Domain Name System)
        - DNS services
        - DNS structure
        - DNS: distributed, hierarchical database
            - TLD, authoritative servers
            - Local DNS name server
        - DNS name resolution example
            - Iterated query
            - Recursive query
        - DNS: caching, updating records
        - DNS records
        - DNS protocol, messages
        - Inserting recoreds into DNS
        - Attackin DNS
            - DDoS attacks
- Feb 7, Thursday
    - Socket programming
        - Socket with UDP
            - UDP communication process
        - Socket with TCP
            - TCP ommunication process
- Chapter 3: Transport Layer
    - Transport layer services and protocols
        - Transport vs. network layer

Jan 15, Tuesday

Refer to the syllbus available on canvas.

Chapter One: Introduction

Internet as an example of network systems

- `Routers` and `switches` are pretty much the same now days.
- Devices not automatically connected. Need the `ISPs`.
- Need `protocol` (the standards) to have a formatted signals.
- Also need internet standards to define the entity

Protocol

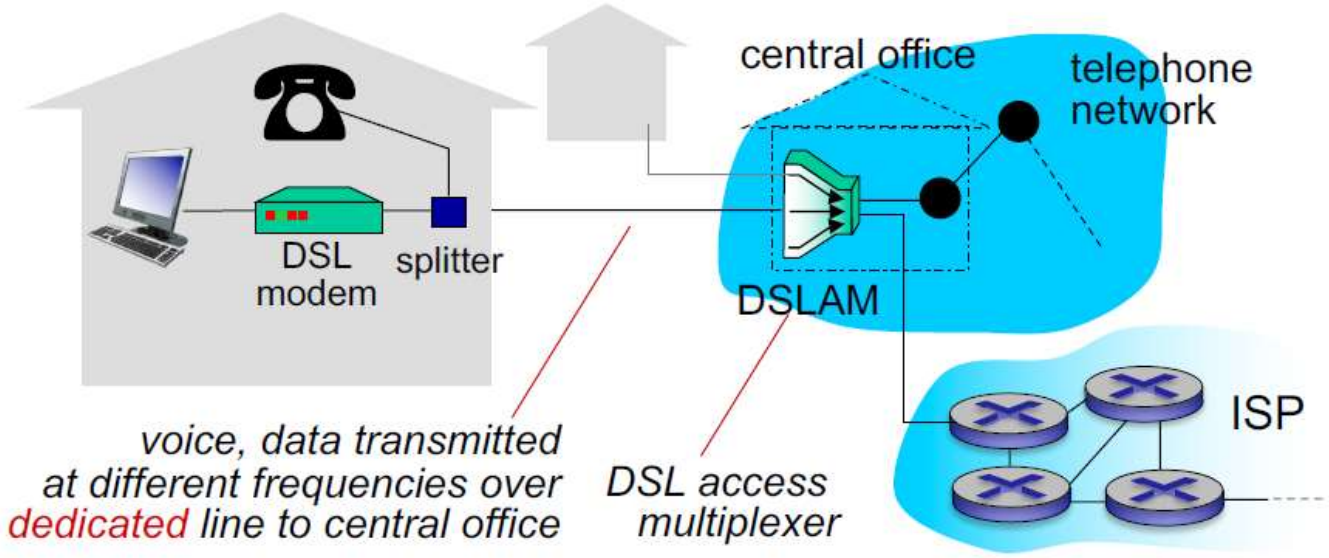
- `Protocal`: define the interactions between two end systems or entities (message to be sent and received etc.)
- `Internet protocol`: defines the standard of internetworks communication
- Set up connection with internet protocol request (eg: TCP) between your pc and the server. Once the connection is setup, you are allowed to get to the file corresponded to the link
- `TCP request` checks if you are authorized to receive the file.

Network Structure

- Different network system on the edge, connected by different ISPs.
- `Routers`: routers within different ISPs relay/forward traffic from your pc/desktop. Routers transmit data, provide scalability, security, etc.
- `Network edge`: the end of the communication
- `Bandwidth`: determines the performance(speed) of your access network

Access Network Types

Digital subscriber line (DSL)

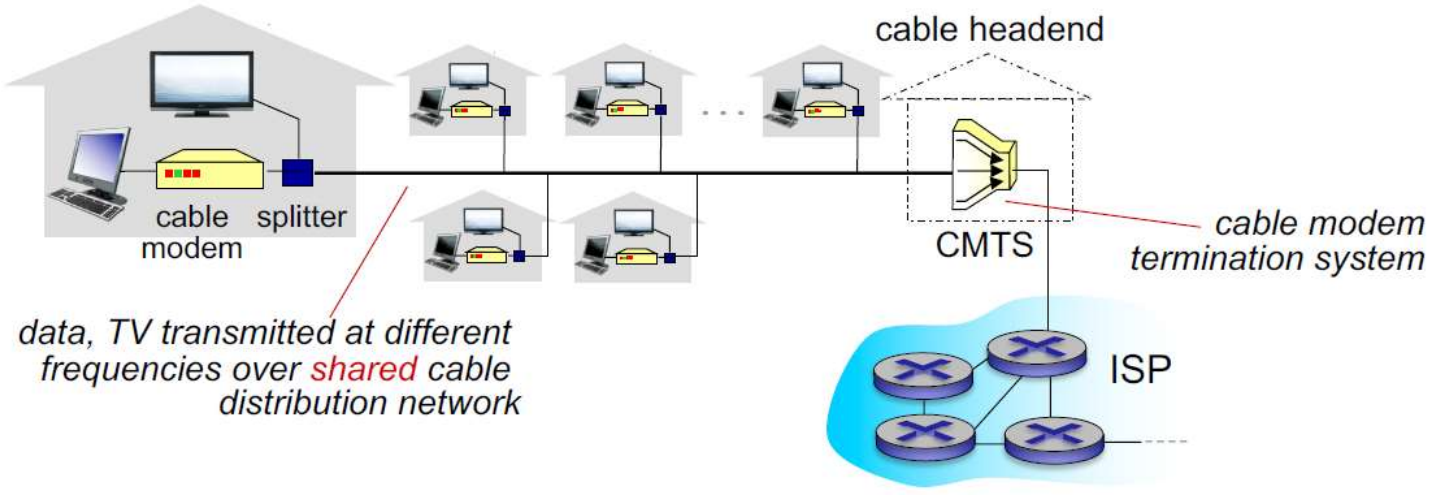


`DSL modem` -CONNECT- `DSLAM` (receive all different signals with different frequencies)

`telephone network` -CONNECT- `DSLAM` (receive all different signals with different frequencies)

- `DSLAM` : DSL access multiplexer

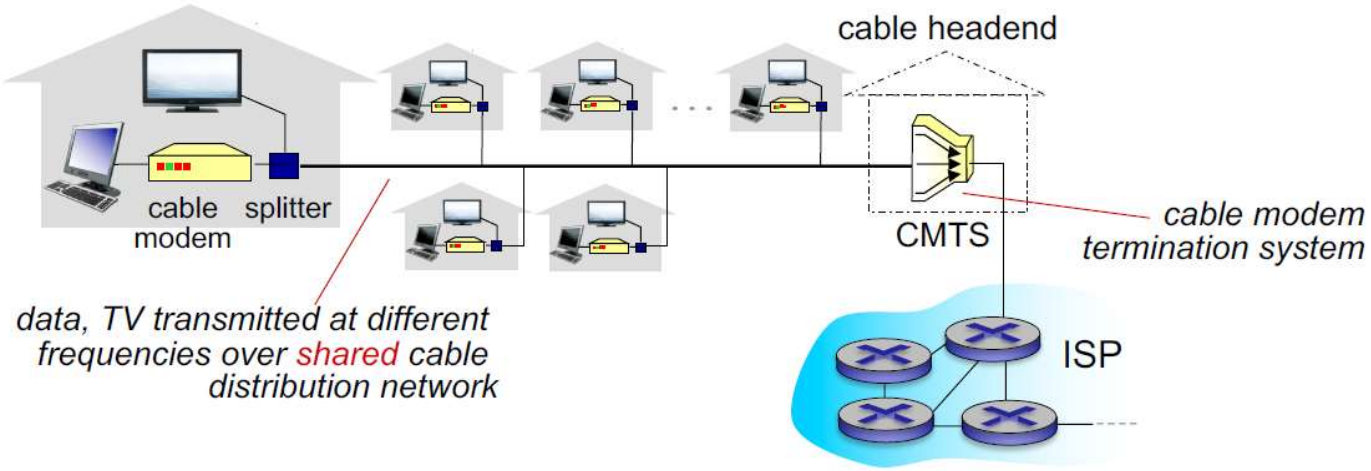
Cable network



your cable modem -- cable headend

Difference:

- `DSL` : dedicated link, transmission



rate slow

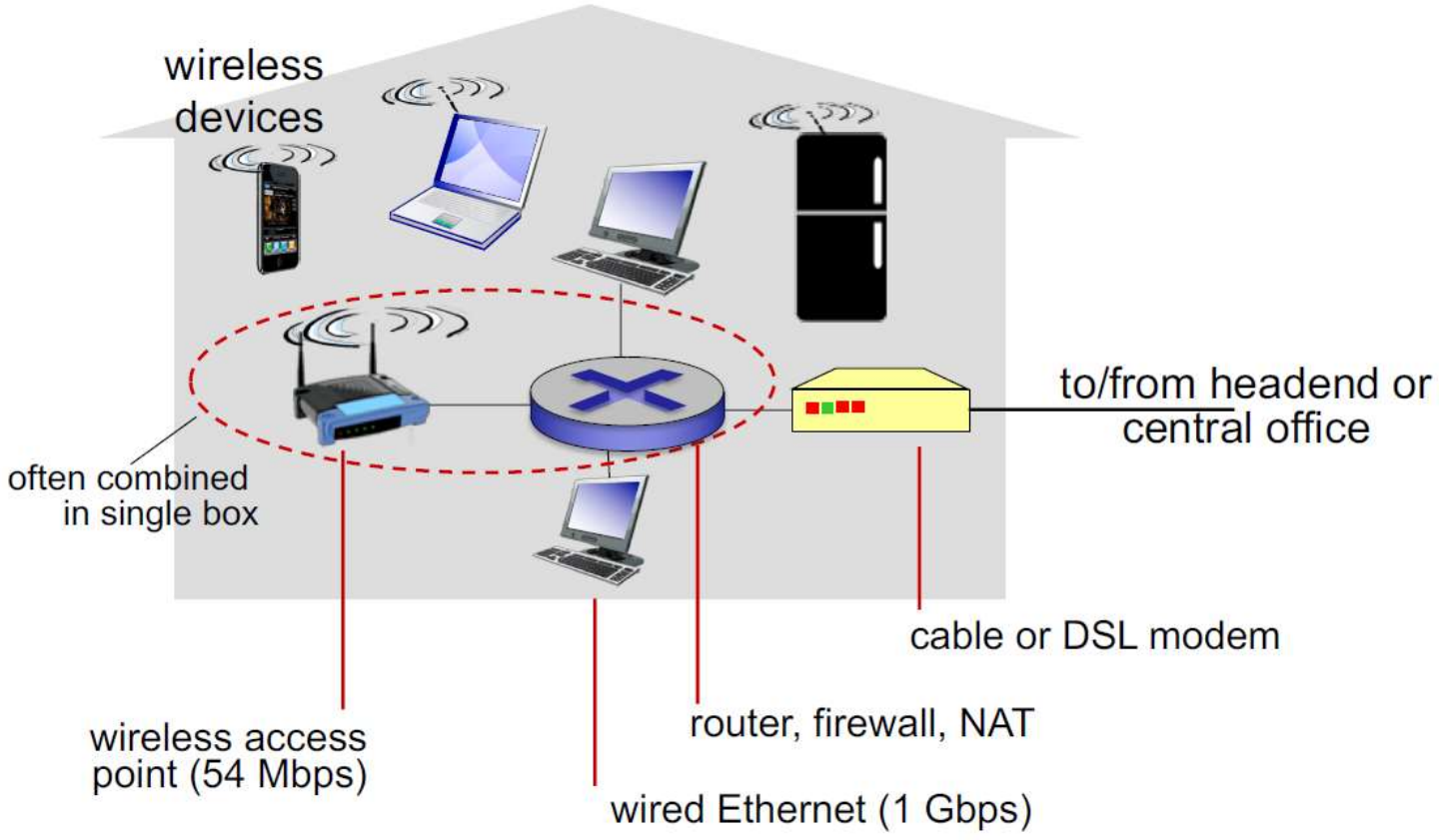
- `Cable`: link shared by multiple users; transmission rate faster
- Both DSL and Cable network carry data generated by different signals, thus need `frequency division multiplexing` to differentiate channels

Frequency Division Multiplexing (FDM)

Since all signals(video, data, control messages) are mixed together, need to differentiate them with FDM.

- `Hyrid Fiber Coax` (HFC): Provide asymmetric up/downstream services

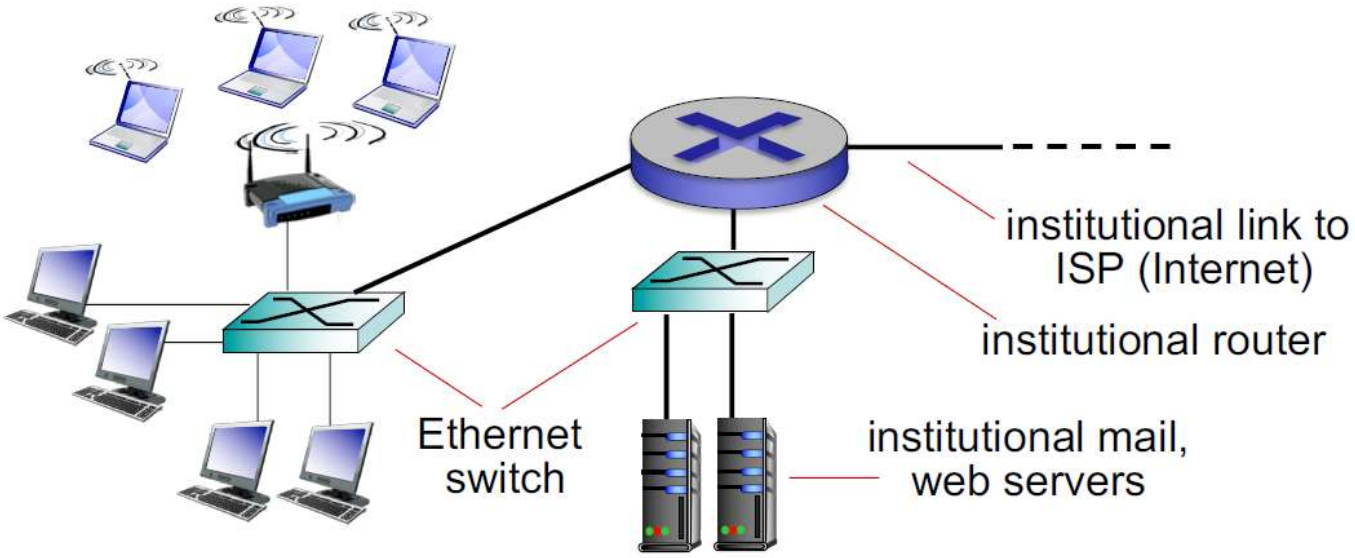
Access network: Your home network



[Wired devices with ethernet interface (eg: laptop, desktop)] -- [router(transmit data, provide scalability, security, etc.)] -- [cable/DSL modem] -- ....

[Wireless devices] -- [wireless access] -- [router] -- [cable/DSL modem] -ethernet switch- [institutional modem (-- institutional mail, web servers, etc.)] -institutional link to ISP- ...

Enterprise network (Ethernet)



Has institutional mail, web servers; high transmission rate

Jan 17, Thursday

Idle Layer

Wireless access networks

shared wireless access network connects end system to router via base station aka "access point"

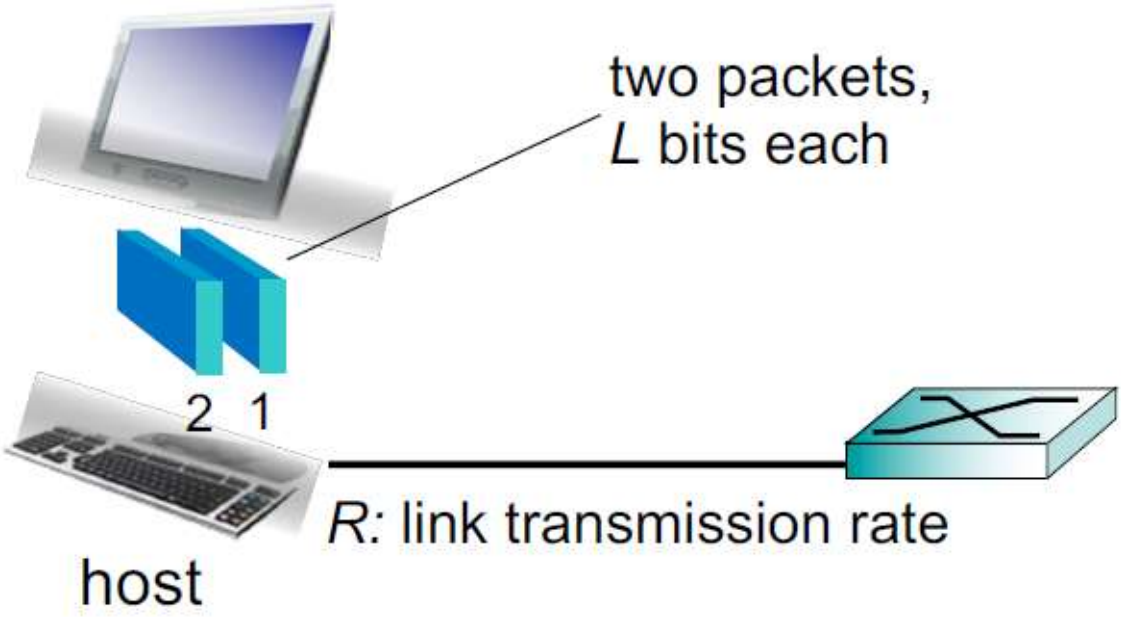
Wireless LANs

- Cover smaller area. Has three different protocols that support different transmission rate
- `Bandwidth`: limited by actual channel (wireless/wired, wired > wireless), different frequencies for wireless channels, cable network determined by the material of the cable

Wide-area Wireless Access

eg: your cellular data. smaller capacity, cover larger area. provided by different carriers

Process of Sending Data Packets



- link transmission rate `R` : aka link capacity or link `bandwidth`
- `L` : length of total bits of the packet
- `packet` : smaller chunks of application message
- [App message] -break into smaller packets of L bits, with transmission rate R- [router]

$$packet\ transmission\ delay = time\ needed\ to\ transmit\ L-bit\ packet\ into\ link = \frac{L}{R}$$

Physical Media

- `bits` : propogates between transmitter and receiver pairs
- `physical link` : what lies between transmitter and receiver
- `guided media` : signals propagate in solid media: copper, fiber, coax
- `unguided media` : signals propagate freely, eg: radio
- `twisted pair` : two insulated copper wires

Coaxial Cable

- two concentric copper conductors
- Bidirectional: support bidirectional data transmission

- Broadband: support multiple channels on cable; HFC

Fiber optic cable

- Higher speed, lower error rate

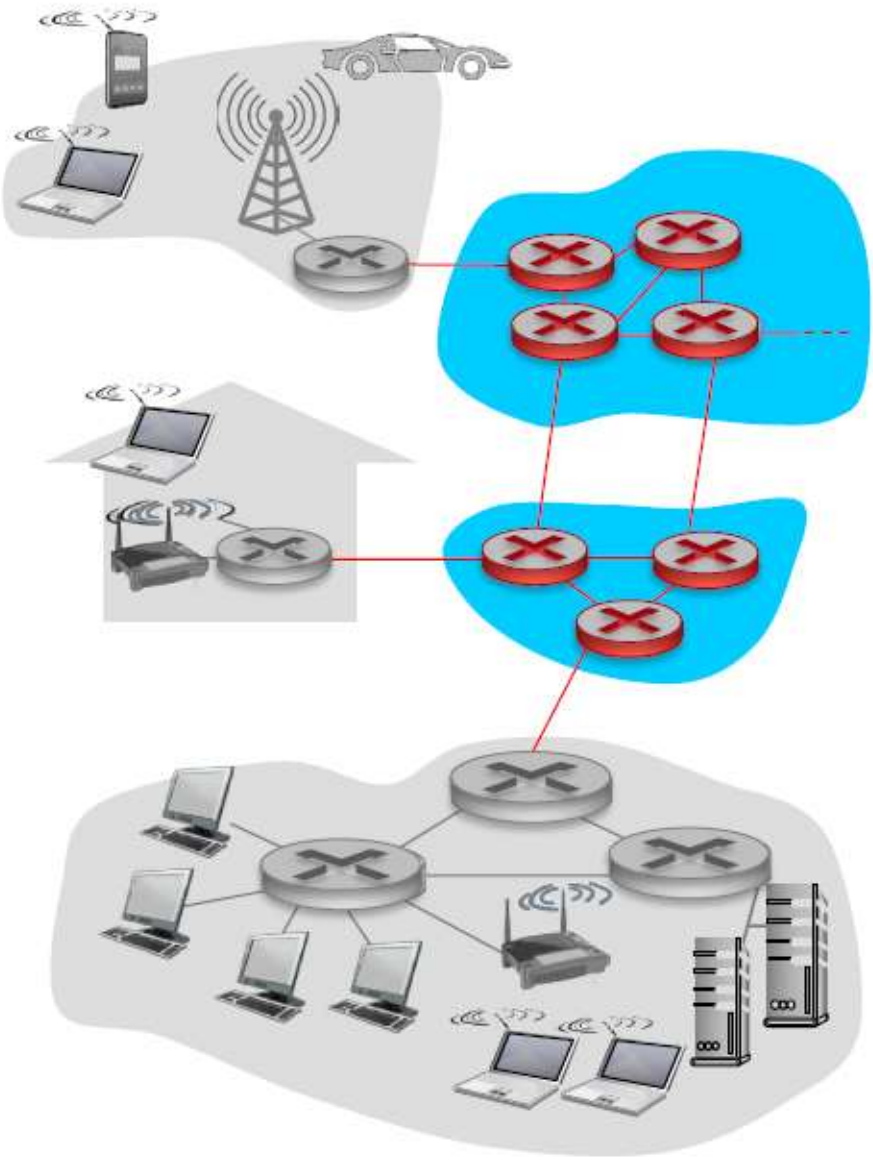
Radio

- signal carried in electromagntic spectrum
- "wireless"
- bidirectional
- propogation environment effects: easily affected by the environment: reflection, interference, obstruction by objects etc.
- `Terrestrial microwave`
- `LAN`
- `Wide area`
- `Satellite`

Network Core

= The network core is a mesh of interconnected routers

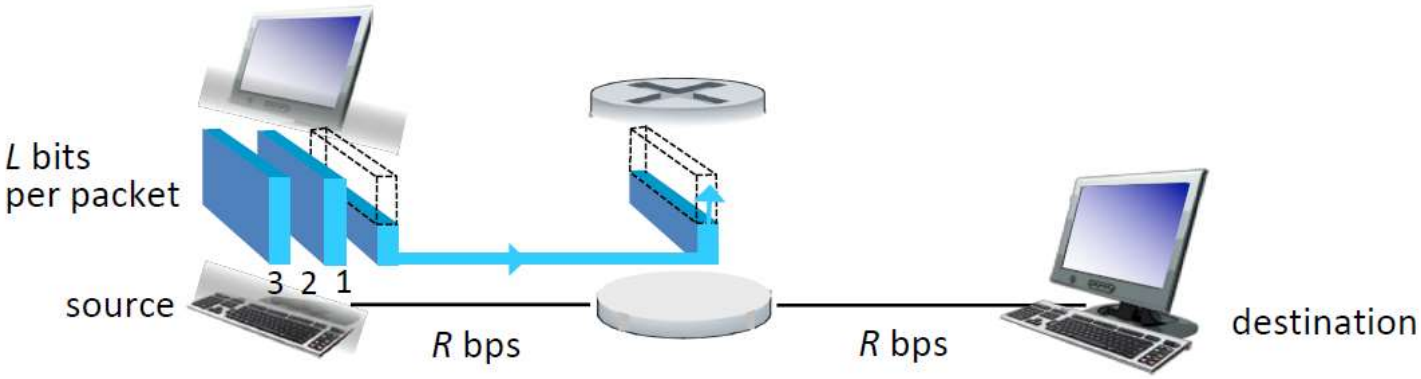
- Ther are two network cores:
  - `packet switching`
  - `circuit switching`



Packet switching:

- hosts break application-layer messages into packets
- then forward packets from one router to the next, across links on path from source to destination
- each packet transmitted at full link capacity

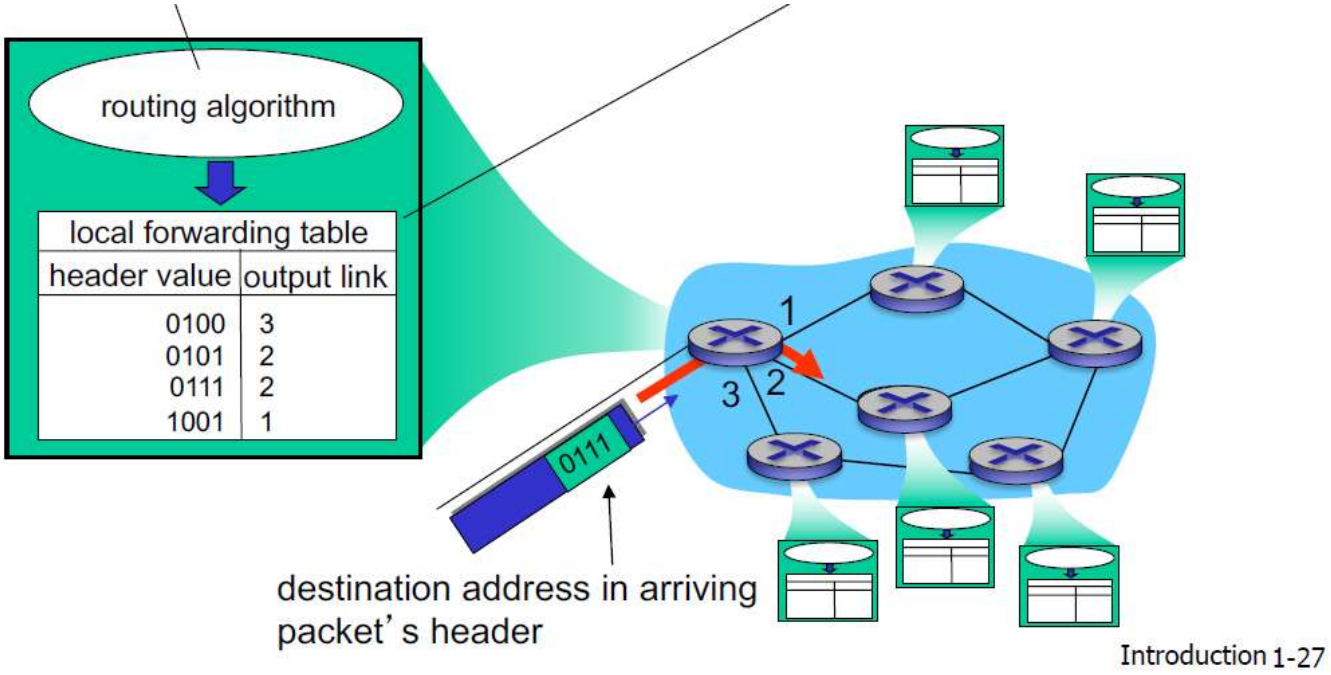
store and forward



- `packet` : consist of bits (basic units)
- packets transmitted in the link as 0/1 signals bit by bit.
- transmission time: length of packet / transmission rate $= \frac{L}{R}$
- **store and forward**:
  - some bits arrive at the routers faster than the rest, it will wait until all the bits arrive in the router before the next transmission
- **queueing delay, loss**:
  - if arrival rate (in bits) to link exceeds transmission rate of link for a period of time, the packets will queue and wait to be transmitted on link
  - if the memory(buffer) fills up, the packets can be dropped(lost)

Routing

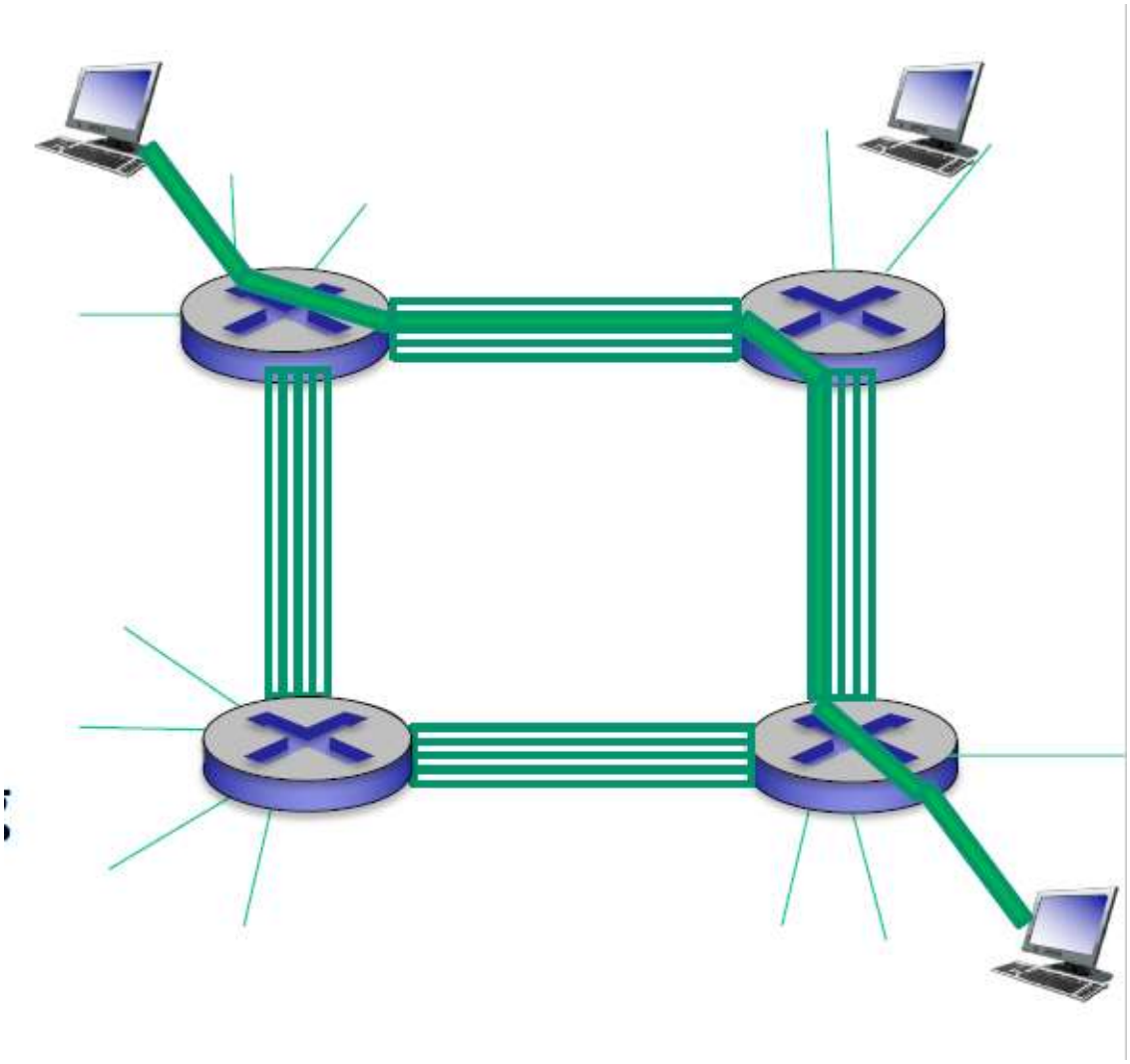
- `Routing` : determines where the packets should be heading to (making decision) by the `routing algorithm` according to a `local forwarding table`
- `local forwarding table` a mapped table contains the following
  - `header value` : the address of the destination, does not change, is shared by the whole system
  - `output link` : the link the packet should be transmitted to, local to each router
- `Routing algorithm` : distributed algorithm that only knows the condition of neighbouring routers.
- Routers need to exchange information to determine the destination.
- Routers do not know the buffer condition of each other.
- The routing process is completed by software of routers.



Forwarding

- The action the router performs according to local forwarding table
- The forwarding process is completed by the hardware of routers.

Circuit switching



- dedicated resource for the user, there is no sharing for a particular circuit
- circuit segment remains idle if not in use.
- commonly used in traditional telephone networks
- circuit switching is no longer in use right now. Mostly replaced by packet switching.

FDM (Frequency Devision Multiplexing)

- Equally devide the bandwidth for the users so that all the users can use the channels all the time

TDM (Time Devision Multiplexing)

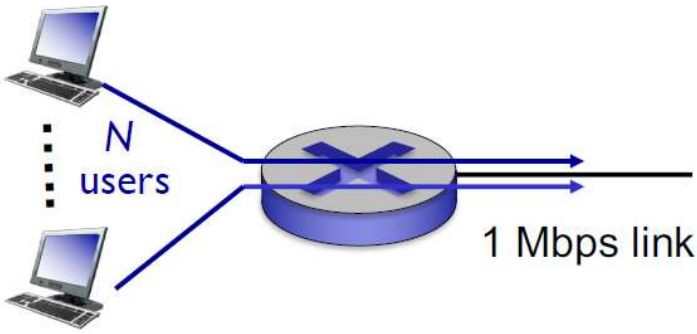
- Each user use the entire bandwidth exclusively for a very small amount of time periodically, so the users cannot realize that they are sharing the bandwidth with other users
- Widely applied in the Operating System, eg: Linux (reserve different time intervals for multiple users)

Packet Switching VS Circuit Switching

packet switching allows more users to use network!

example:

- 1 Mb/s link
- each user:
  - 100 kb/s when "active"
  - active 10% of time



- *circuit-switching:*
  - 10 users
- *packet switching:*
  - with 35 users, probability > 10 active at same time is less than .0004 *

Q: how did we get value 0.0004?

Q: what happens if > 35 users ?

$$\binom{n}{0}0.1^{0}0.9^{n-0}+\binom{n}{1}0.1^{1}0.9^{n-1}+\cdots+\binom{n}{10}0.1^{10}0.9^{n-10}>1-0.0004$$

Circuit Switching

Only allow limited amount of users. Reserve resources for limited users.

Packet Switching

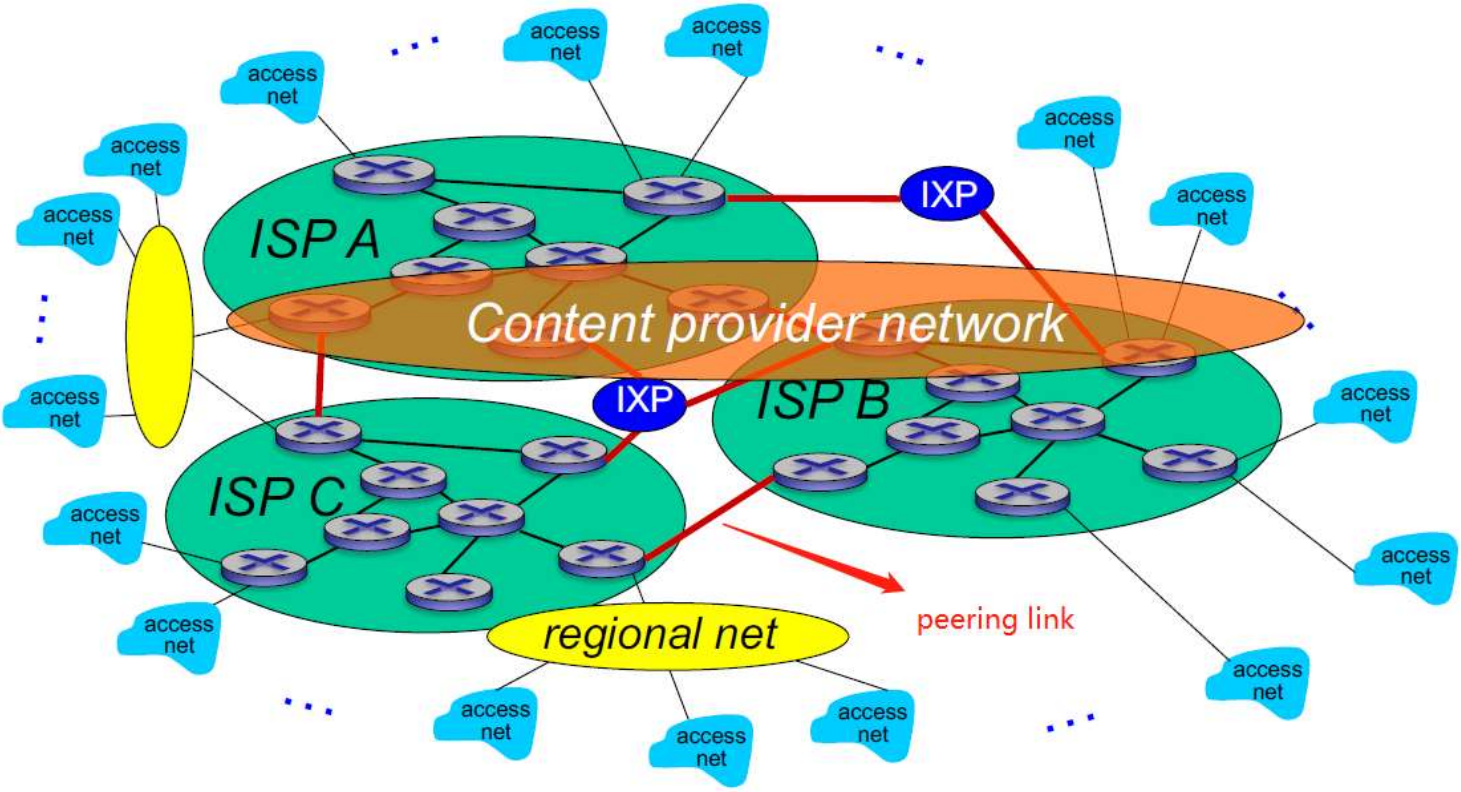
- Allow resource to be shared. Thus could accomodate more users.
- Could have `congestions` : `packet lost/delay`
- need good protocol for reliable data transfer, `congestion control` .

Internet Structure: network of networks

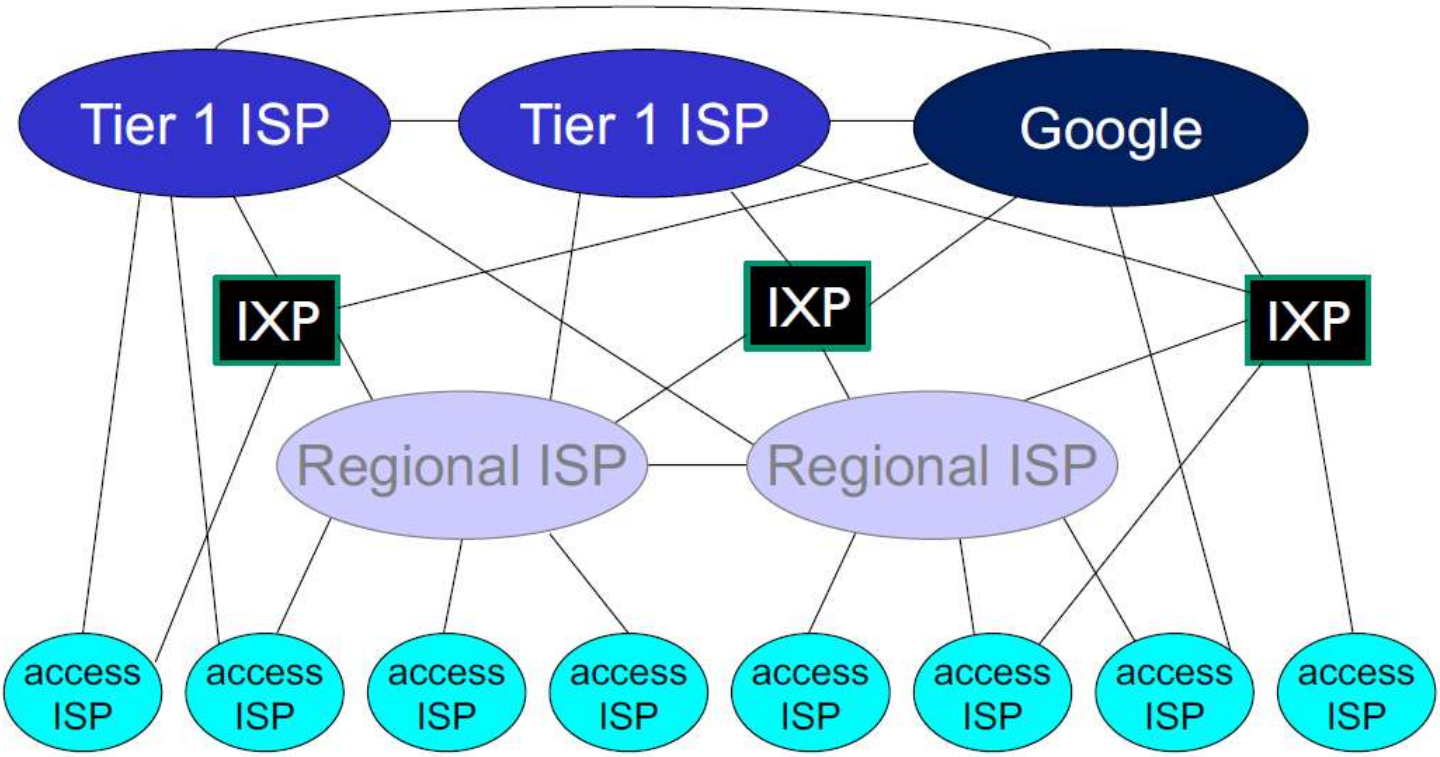
We need to connect different access networks with each other. But how?

With a global ISP? NO, need the same agreement

With **different core ISPs interconnected to each other**? YES!



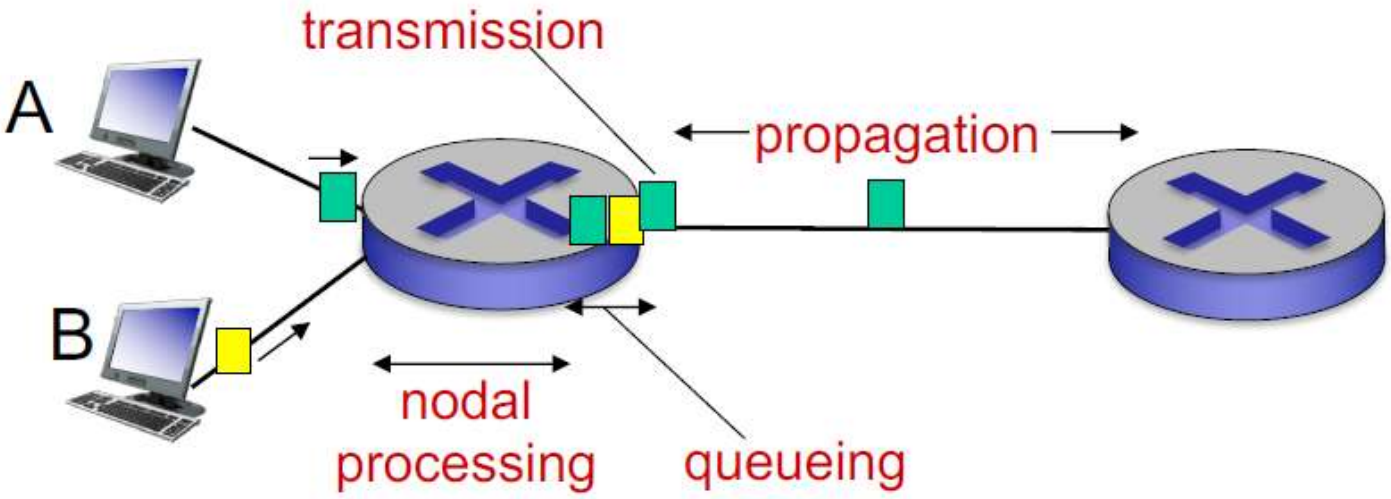
- `ISP`: internet service providers
- End systems connect to Internet via access ISPs
  - residential, company and university ISPs
- Access ISPs in turn must be interconnected.
- Different `core ISPs` connected by `peering links` and `IXP hubs` (internet exchanging point)
- Core ISPs connected to `regional ISPs`



- at center: small number of well-connected large networks
- tier-1 commercial ISPs (e.g., Level 3, Sprint, AT&T, NTT), national & international coverage
- `Content providers`: private network (eg: Google) with multiple `data centers` connecting to IXP, tier-1 and regional ISPs directly
- `POP`: point of presence. Frequently used by content providers to get closer to their end users

---

Delay, loss, and throughput in networks



$$d_{nodal} = d_{proc} + d_{queue} + d_{trans} + d_{prop}$$

The sequence is: arrive at router - `process` - `queueing` - `transmission` - leave router - `propogation` - arrive at next router

Process Delay

- d_{proc}: the time the router takes to process the entire packet upon arrival
- also called `nodal processing`
- check bit error
- determine output link
- typically < msec

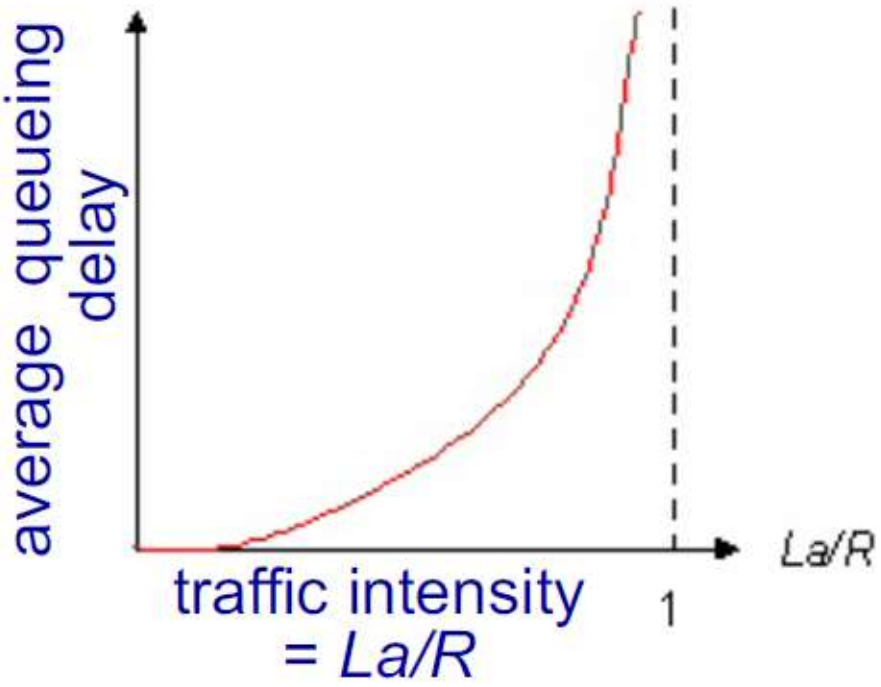
Transmission Delay

- d_{trans}: the time the router takes to process and direct the packet bits by bits in the router *(The time the toll station takes to serve the entire caravan car by car)*
- `L`: packet length (bits)
- `R`: link bandwidth (bps)
- $d_{trans} = L/R$

Queueing Delay

- d_{queue}: time interval between arrival and being sent again while waiting at the output link for transmission. *(The time one caravan waits while the former caravan is being serviced)*
- depends on congestion level of router
- calculate length of queueing delay by determining the value of average queueing delay

- can lead to data loss

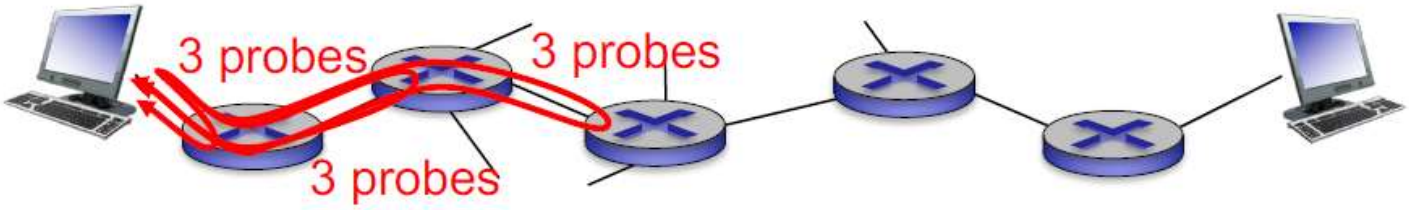


- `R`: link bandwidth (bps)
- `L`: packet length (bits)
- `a`: average packet arrival rate
- traffic intensity $= La/R$
  - $La/R \approx 0$: average queueing delay small
  - $La/R \to 1$: average queueing delay large
  - $La/R > 1$: more packets arriving than can be serviced, average delay infinite

Propogation Delay

- d_{prop}: the time it takes the packets to travel in the link from one router to the next. *(The time the caravan takes to travel to the next toll station)*
- Usually very fast
- `d`: length of physical link
- `s`: propagation speed (~$2 \times 10^8\ m/sec$)
- $d_{prop} = d/s$

Real internet delays and routes

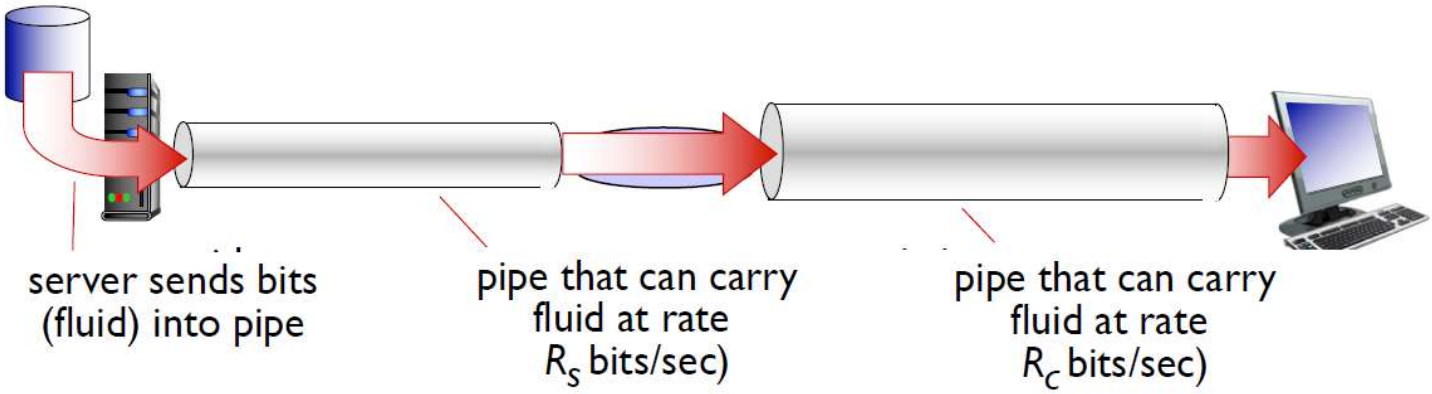


- `traceroute` program: provides delay measurement from source to router along end-to-end internet path towards destination
  - for all i:
    - sends three packets that will reach router i on path towards destination
    - router i will return packets to sender
    - sender records the time interval between sending and receiving of the packet

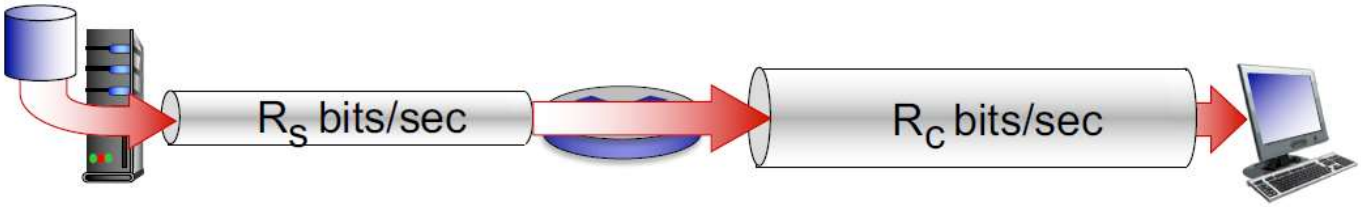
Packet Loss

- `Packet loss`: packets can be dropped due to insufficient `buffer size`
- `Buffer size`: the storage size of packets that could be stored in the router

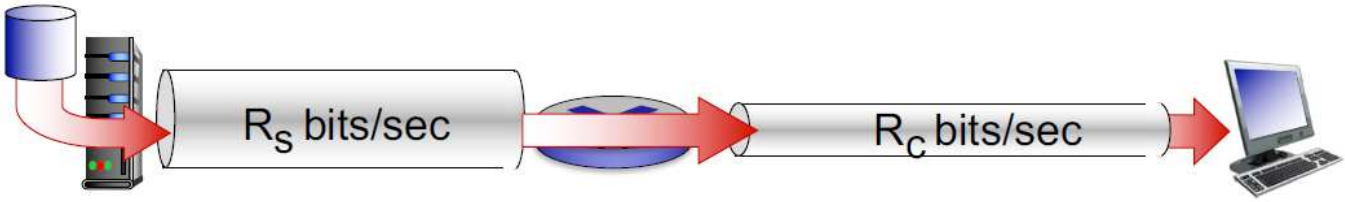
Throughput



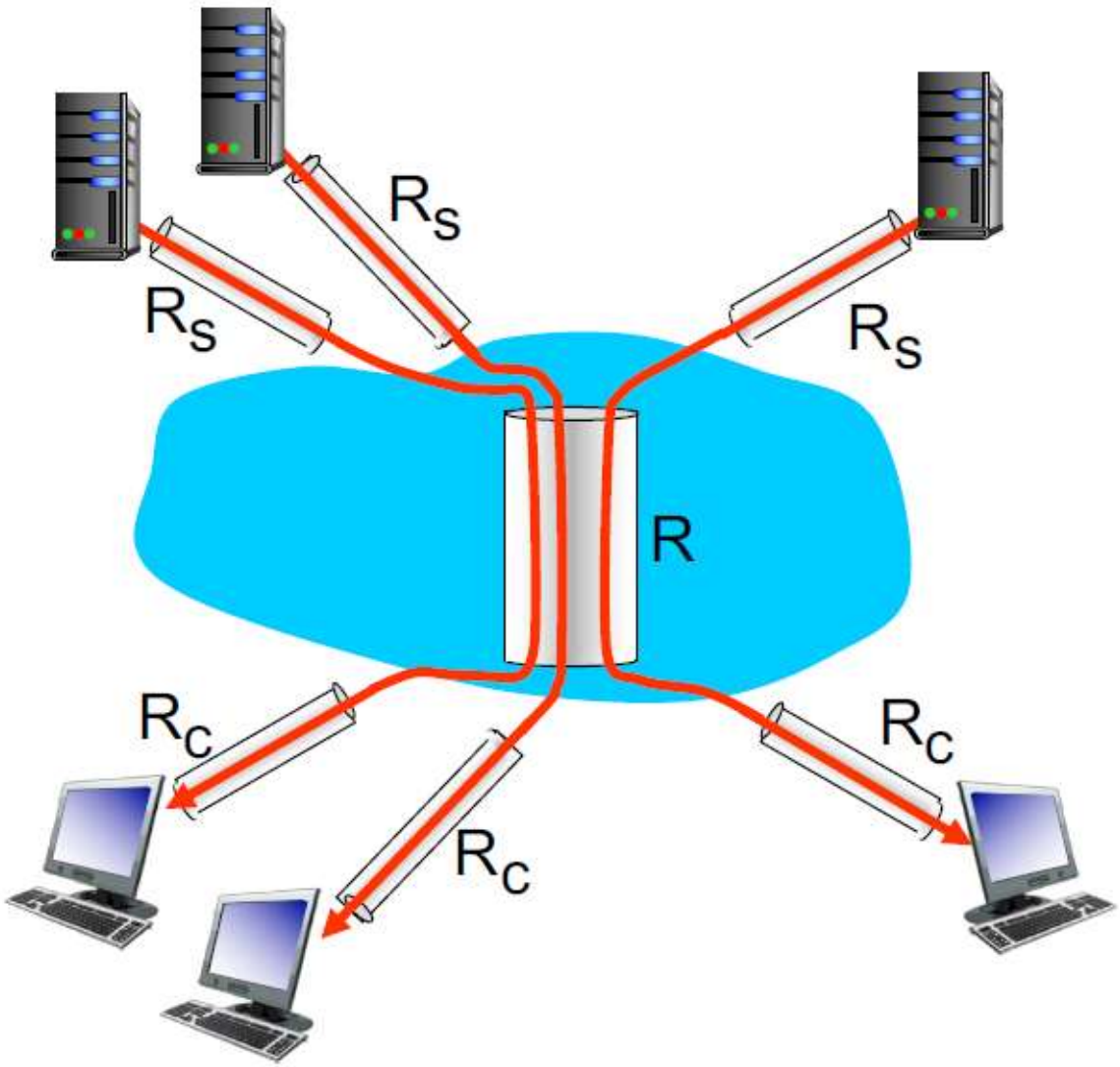
- `throughput`: rate ($bits/time\ unit$) at which bits transferred between sender and receiver
- `instantaneous`: rate at given point in time
- `average`: rate over longer period of time
- $R_s < R_c$: average end to end throughput $= R_s$



- $R_s > R_c$: average end to end throughput $= R_c$



- `bottleneck link`: link on end to end path that has **smaller** transmission rate
- **internet scenario:**



10 connections (fairly) share backbone bottleneck link R bits/sec

  - per-connection end to end throughput: $min(R_c, R_s, R/10)$

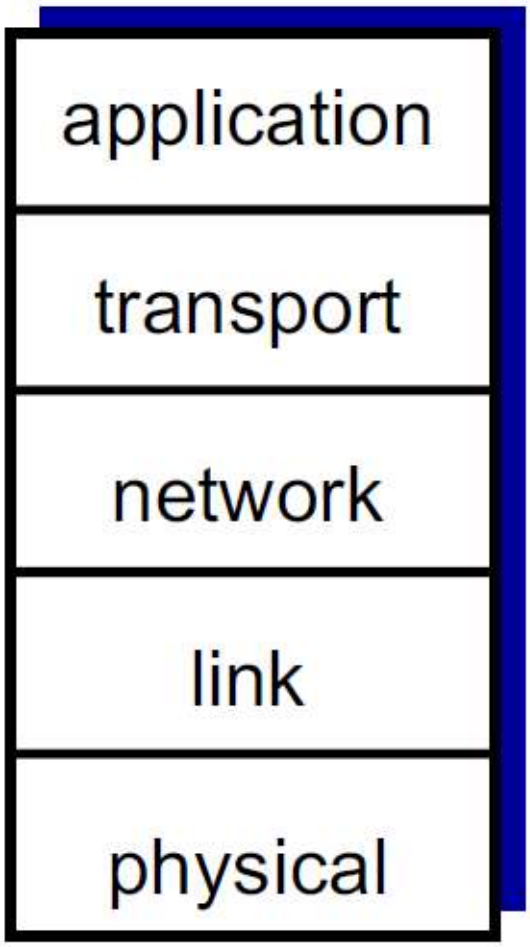
- in practice, R_c or R_s is often bottleneck

January 22, Tuesday

Protocol Layers

- Used to organize all the pieces of `network structure`: host, routers, etc.
- `Moduralization` ensures that `individual layers` would not affect the functions of other layers
- The layers can work completely independently by themselves, but also able to communicate information
- But we may not always divide all the functions into independant layers.

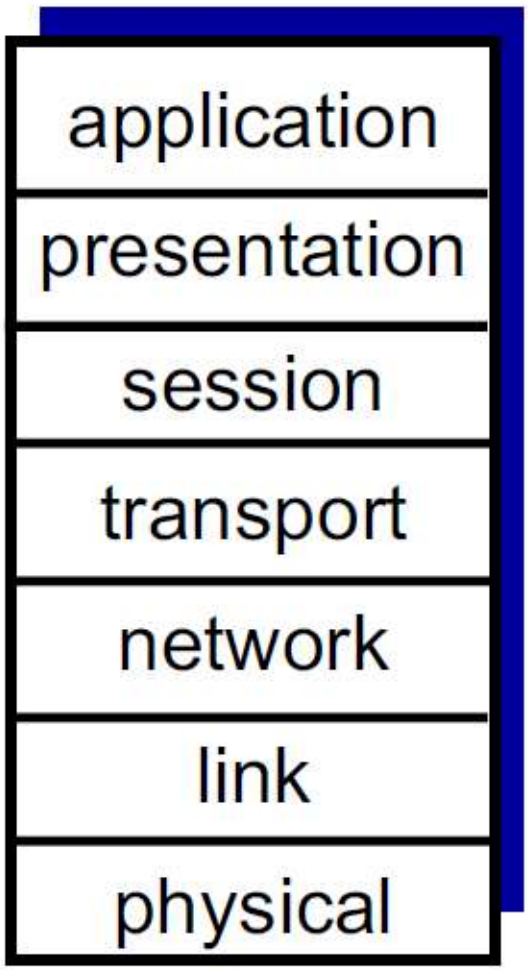
Internet Protocol Stack: End to End Protocol



- **application layer**
  - Supporting the network applications.

    - eg: FTP, HTTP, SMTP
    - HTTP is developed based on TCP
- **transport layer**
    - Interprocess data transfer
    - Only happens between two end systems to let two processes communicate with each other
    - eg: TCP (guarantee reliability of data transfer, contains congestion control), UDP (unreliable data transfer)
    - TCP is slower than UDP. Most game or streaming applications use UDP rather than TCP. But TCP contains reliability, whereas UDP does not guarantee
    - Not necessary for the routers/switches to support the transport layer protocols, although most devices now days can also support such layer.
- **network layer**
    - Both routing and forwarding are in this layer.
    - Routing of datagrams from source to destination.
    - Only IP protocol
    - `IP protocol`: consists of different routing protocols
- **link layer**
    - Only used to transfer data within two neighbouring network elements (router to router, switches to switches, etc.)
    - Do not connect two end users
    - eg: Ethernet (wired connection), 802.11 (WiFi, wireless connection), PPP
- **physical layer**
    - Basically are just bits on the wire

ISO/OSI reference model

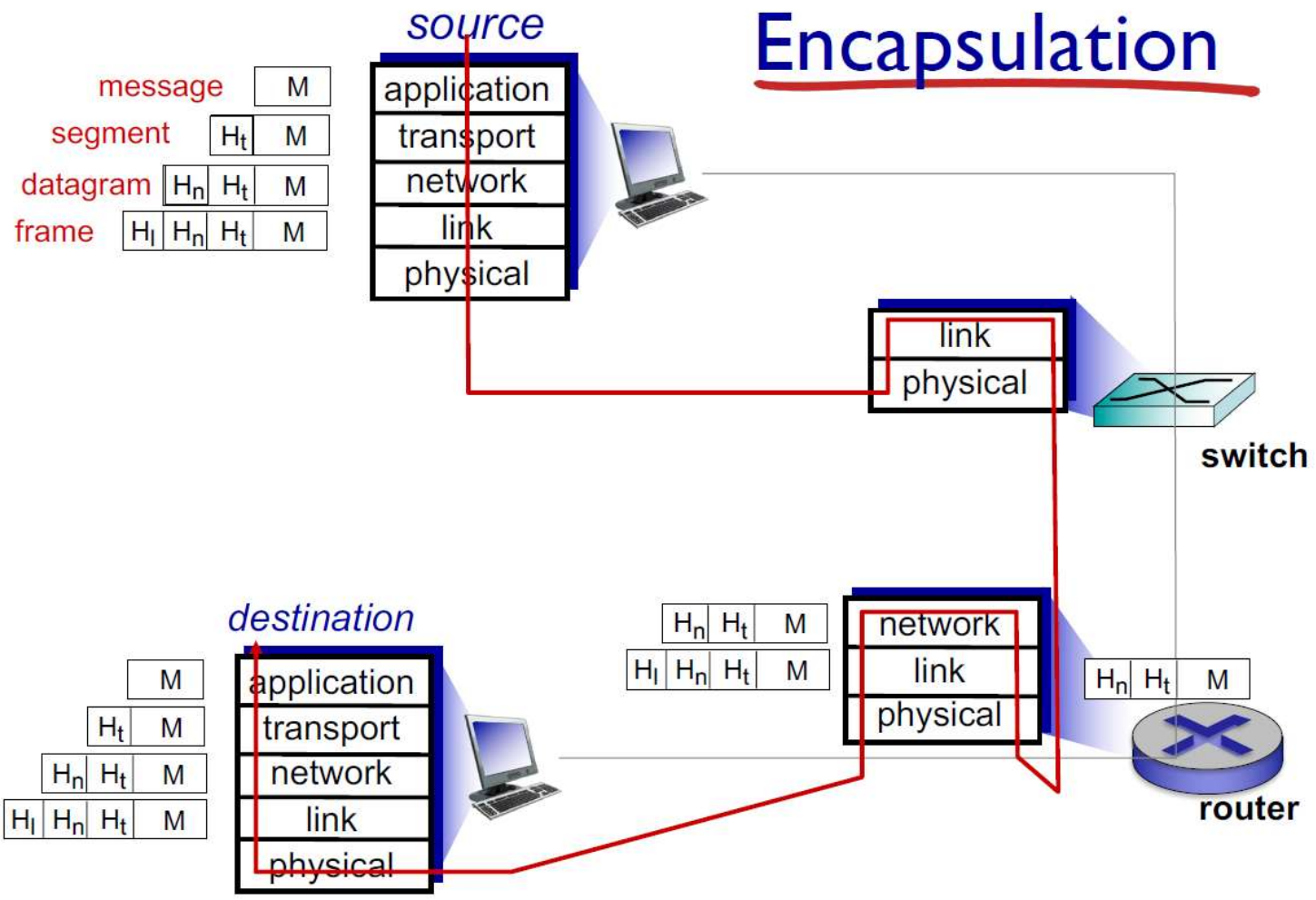


- Includes two extra layers that internet stack does not share
- These services must be implemented in application if needed
- Not necessary for all applications, thus can be combined with application layer
- **presentation layer**
  - allows application to describe/interpret the meaning of data: encryption, compression, machine-specific conventions etc.
- **session layer**
  - synchronization, checkpointing, recovery of data exchange

Process of data transfer between end users

- Data are transferred with `encapsulation`
- Each layer generates a specific type of `header` and attach it to the `message` being tranferred.
- The header contains *layer specific information* (eg: ip address by the network layer), and **only corresponding layers** can decrypt it.

- While the headers are being transferred between different layers, some of the headers might be changed, such as the data link layer, and the network layer for specific applications
- Tranport layer information (header) will never be changed



- The process: [application layer] - message -> [transport layer] - segment -> [network layer] - datagram -> [link layer] - frame -> [physical layer]
- Each layer can only see the correct corresponded header
- The layers try to interpret/decapsulate the packets even though they might not understand
- [source] - encapsulated packets -> [router]
- Router drop the network header and check how to head the packet to the next link
- A new network layer header is attached to the packet
- [router] - encapsulated packets -> [destination]
- The destination end user decrpt the packet encapsulated with layers of headers

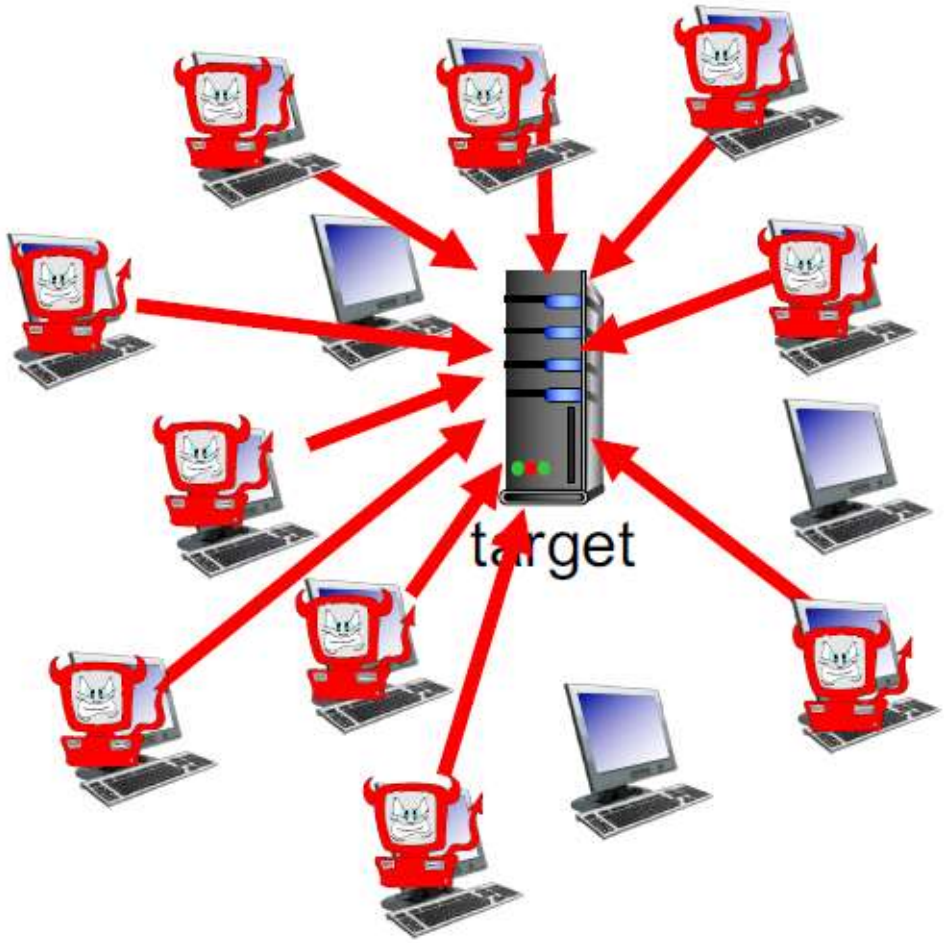
---

Network Security

Malware

- Hide behind your system/application.
- can get in host from:
  - `virus`: need to be exectued to infect your devices
  - `worm`: if your pc is infected, your mobile device can also get infected
- `spywre malware`: record your keystrokes. Once infected, the host can be enrolled in `botnet`, used by the hacker to attack servers illegally, such as spam and DDoS attacks.
- By hacking other machines, the hackers utilize others' devices and hide their identities.

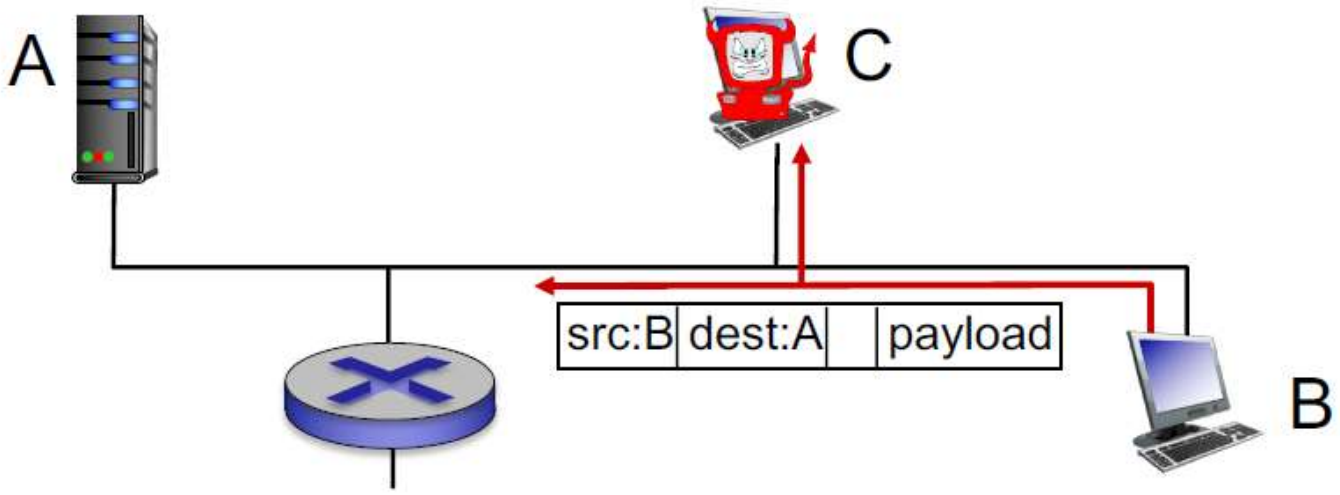
Denial of Service (DoS)



Attackers make resources (server, bandwidth) unavailable to legitimate traffic by overwhelming resource with bogus traffic

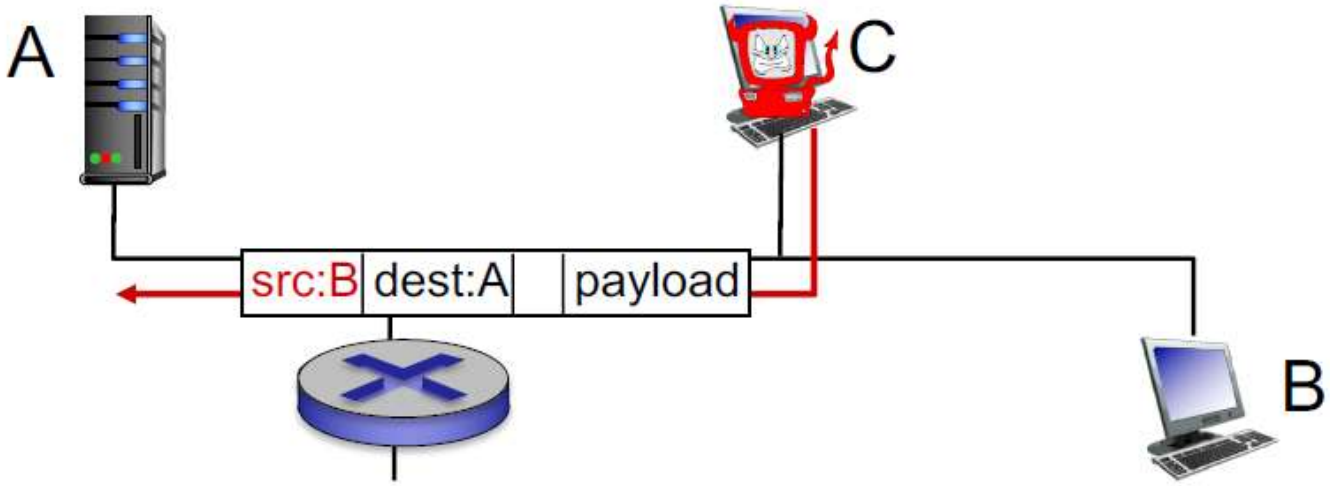
1. select target
2. break into hosts around the network (see botnet)
3. send packets to target from compromised hosts

Packet "Sniffing"



- broadcast media (shared Ethernet, wireless)
- promiscuous network interface reads/records all packets (eg: including passwords) passing by
- wireshark software used for end-of-chapter labs is a (free) packet-sniffer

IP spoofing



send packet with false source address

Jan 24, Thursday

Chapter Two: Application Layer

Application Architecture

Client-Server Architecture

- **server**
  - Always-on host
  - Have a permanent public IP address in most cases
  - Have more servers to cater more requests: have data centers
  - data center: a building stores only servers, routers, switches etc. An approach for server providers to scale up?
- **client**
  - communicate with the server
  - may be intermittently connected
  - Dynamic IP adress depanding on server
  - Does not communicate directly with each other.
  - [Client] - message -> [Server] - message -> [Client]
- **communication process**
  - `process`: program running within a host
  - client process: initiate communication
  - server process: waits to be contracted by clients

P2P architecture

- no always-on server
- end systems can directly communicate with each other
- peers request and provide serivce to each other, which pertains **self scalability**:
  - both new capacity and demands are brought to the architecture when new peers show up
- peers are intermittently connected and change IP addreesses
  - more complexed management
- **communication process**:
  - within same host, two processes communicate using inter-process communication defined by OS
  - processes in different host communicate by exchanging messages
  - aside: applications with P2P architectures have client processes & server processes

Sockets

Addressing processes

- Processes need identifier to receive messages
- Identifier: containing IP address and prot number associated with the process on host

- Port number eg: HTTP (80)

App-layer protocol defines

- REFER TO THE SLIDES DUDE

What transport services the app required

- Interlayer requirements that should be done by layer collaboration
- Also see the slides

Internet transport protocols services

- Reliable transport: between sending and receiving sides
- Flow control: by receiving side(?) to make sure receivers do not get overwhelmed by sender messages
- Congestion control: by sending side to make sure internet resources are also shared by other end systems

---

Web and HTTP

- Internet objects are addressable by URL which consists of a host name + a path name
- `HTTP`: hypertext transfer protocol
  - web's application layer protocol
  - applies the **client/server model**
    - **client**: browser that requests, receives (with HTTP protocol), and displays web objects
    - **server**: web server sends objects (with HTTP protocol) in response to requests

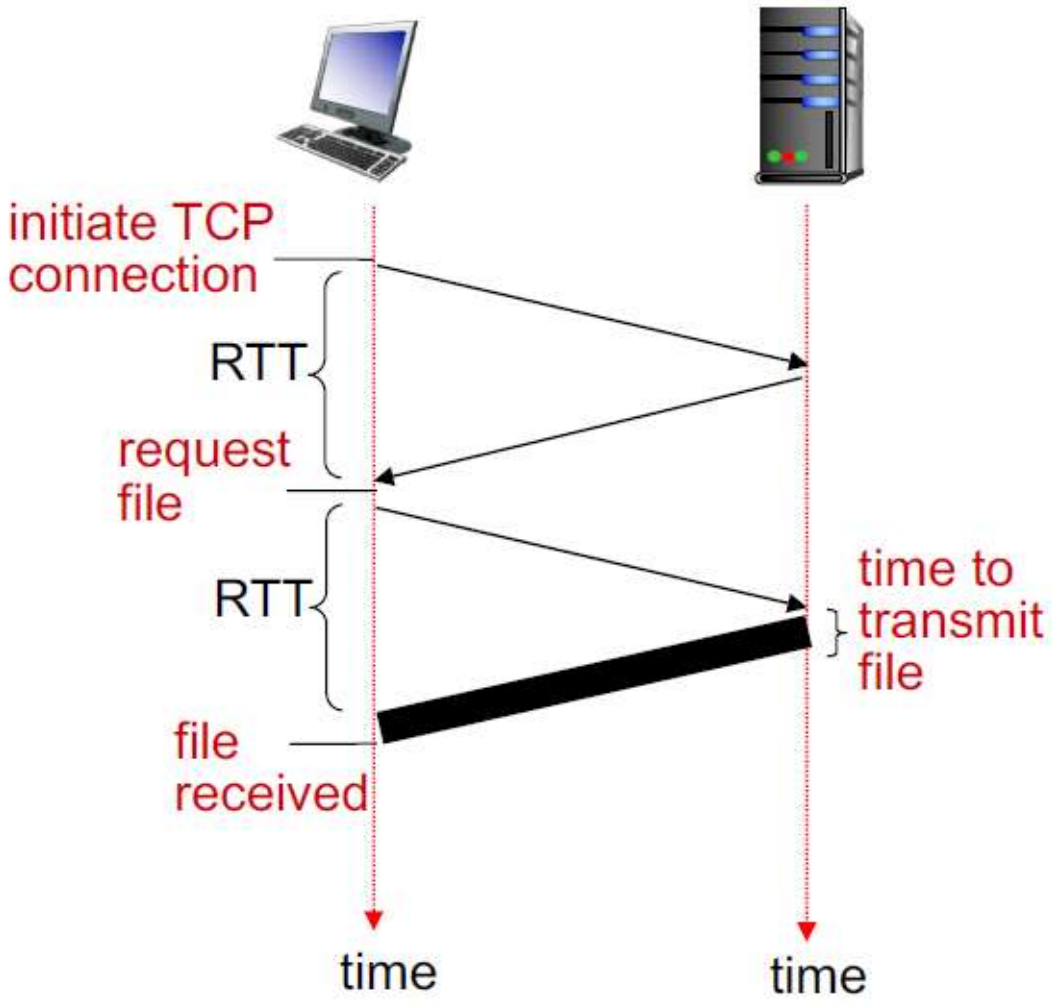
HTTP connections

- HTTP is using `TCP` protocal
- **Client initiates TCP connection to server** using `port 80`
- Server accpets the TCP connection from client
- HTTP messages (application layer protocol messages) exchanged between browser (HTTP client) and Web server
- The server closes the TCP connection (sometimes the client can also close the connetion by accident, in which case the server might not konw, and the server will close the connection after a certain amount of time -- connection timeout)
- HTTP is stateless, servers do not need to maintain a certain state of past client requests
- SSL: provides secured library for TCP. TCP does not use SSL by default

- HTTPs is TCP using SSL?
- `RTT`: **run trip time**, the time it takes the small packet to travel from the client to the server and then back from the server to the packet. Cannot avoid this time no matter what.

Non persistant HTTP connections

- **At most one object** can be sent over TCP connection, which will be closed by the server after one request message and resoponse
- To download multiple objects, the HTTP client should build up multiple connections
- After downloading the first object receiving from the server, in order to download a second object, the HTTP client should establish the TCP connection with the server again, and repeat sending out requests etc.
- **response time**:



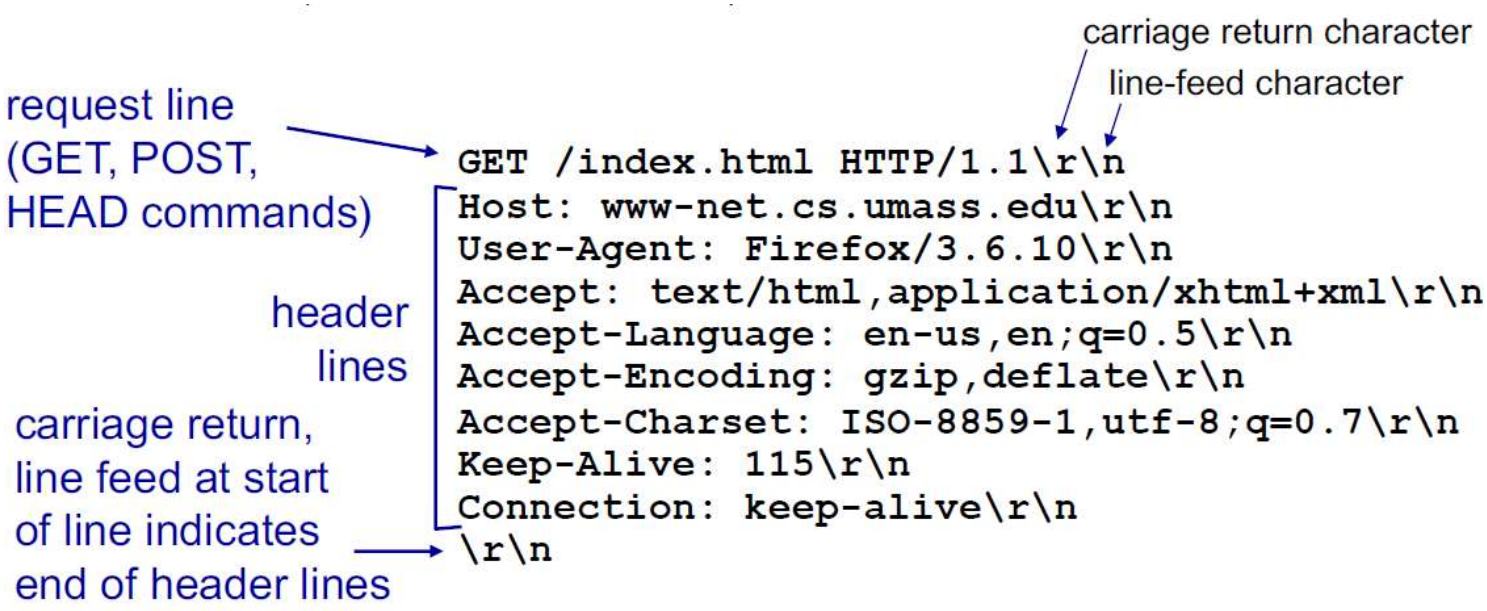
  - Initiate TCP connection: 1 RTT (must happen before sending HTTP request)
  - Send HTTP request and receive file back from server: 1 RTT + some time to transmit the file
  - Cost $2RTT + file\ transmission\ time$ to request and receive in total
- issues:
  - requeires $2RTT$ per object

- OS overhead for each TCP connection
- browsers often open parallel TCP connections to featch referenced objects

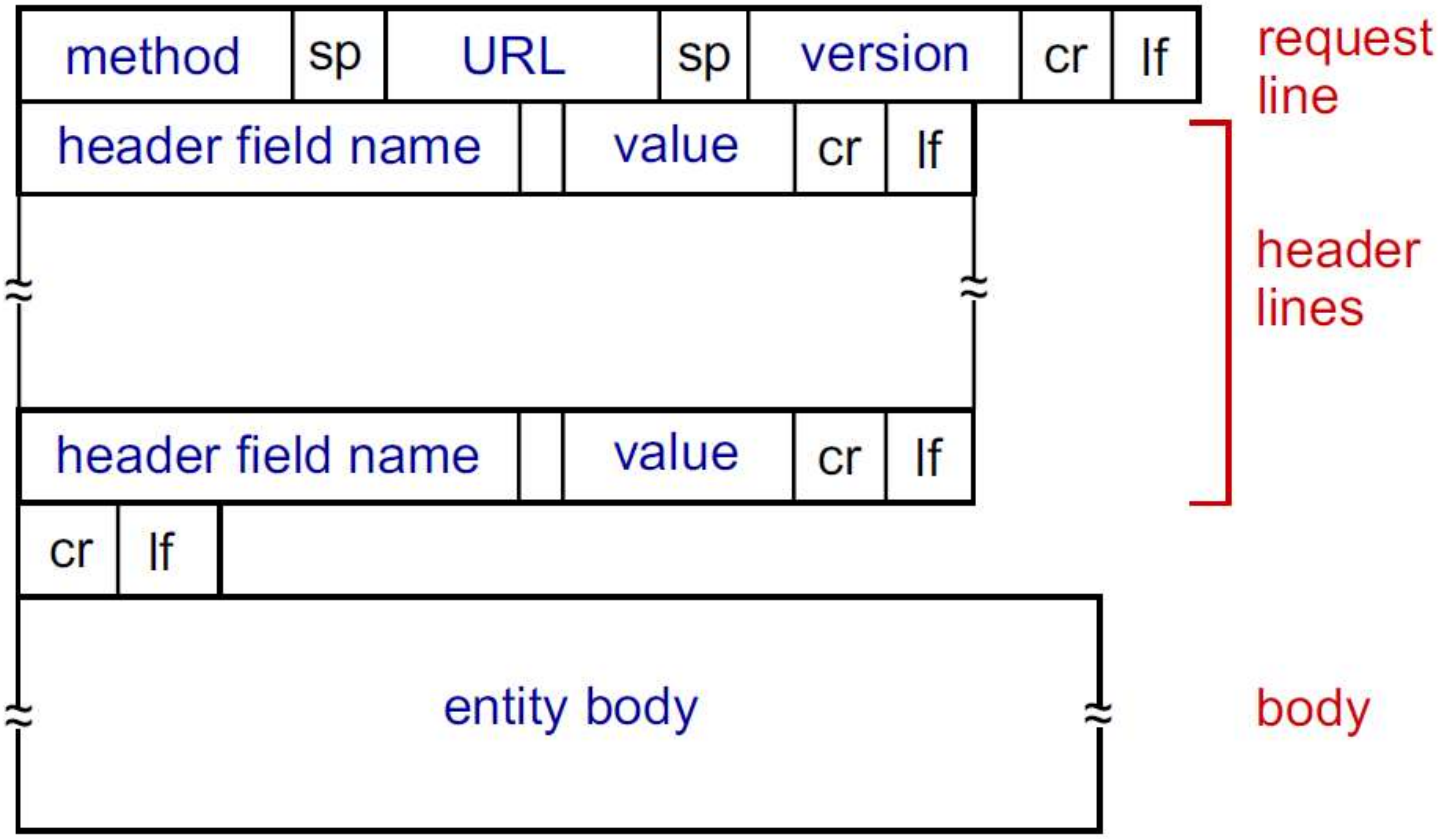
Persistant HTTP connections

- only need to set up the TCP connection once
- server leaves connection open after sending response
- all later on HTTPmessages between same client/server sent over open connection
- client sends requests as soon as it encounters a referenced object
- Cost only $1RTT$ roughly

HTTP request message

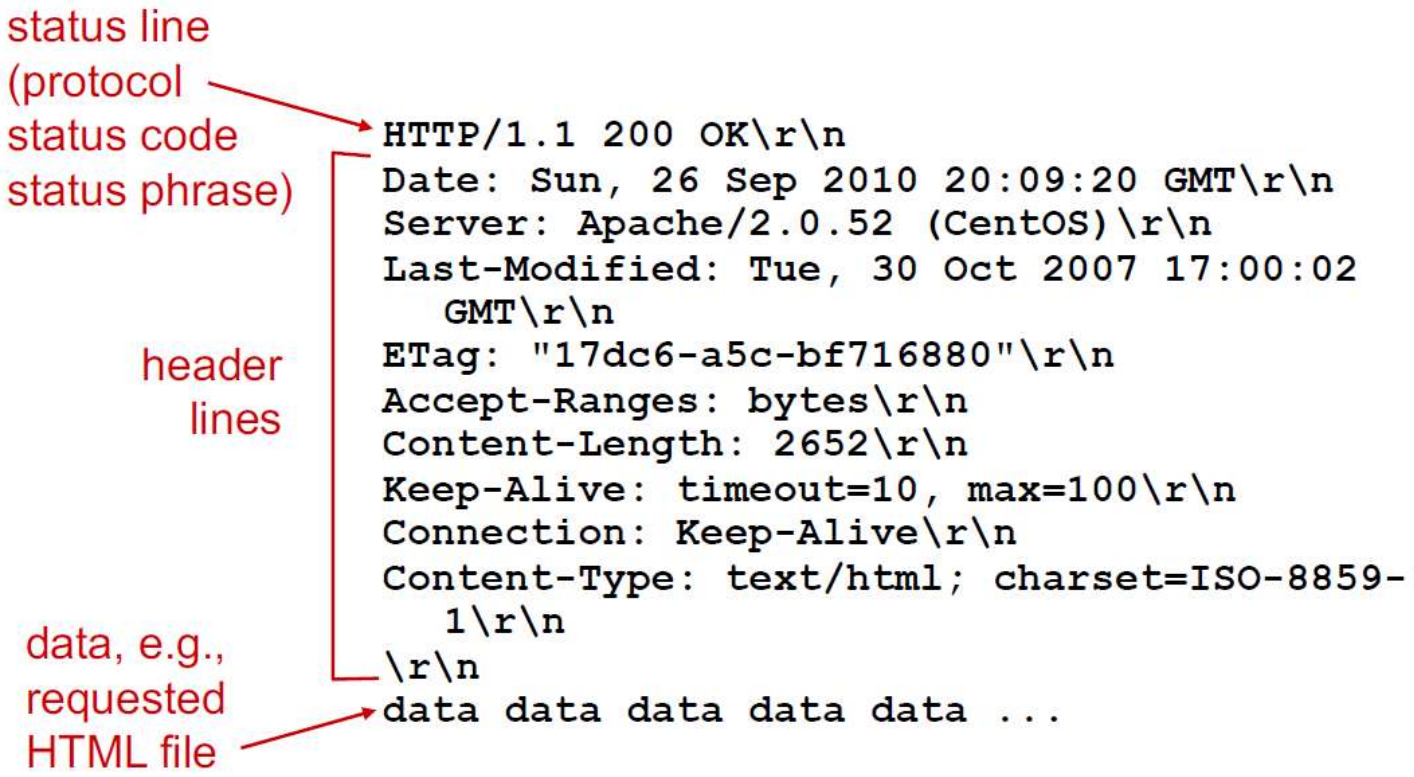


- Two types: request and response
- Applying the ASCII format, which is a human readable format
- **general format**



- Reuest line
- Header lines
- Body: contains the user input if the user is using post method
  - eg: user input typed in the search box
  - `POST` method
    - web page often includes form input
    - input is uploaded to server in entity body
  - `URL` method
    - use GET method
    - input is uploaded in URL field of request line

HTTP response message



- Status line: versions of HTTP protocol the server is running
- Header lines:
- Data objects requested by the client

Response status codes

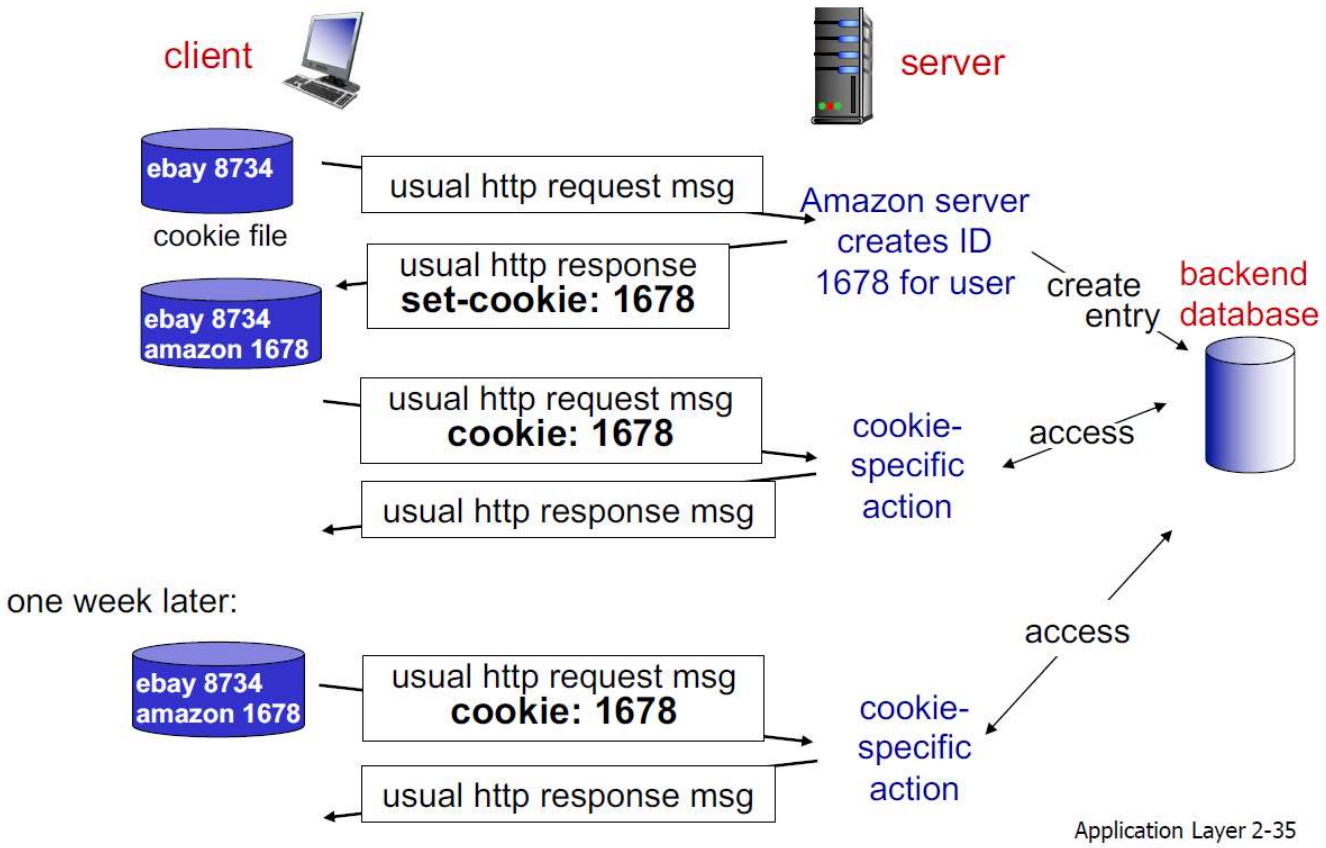
- 200: ok
- 301: requested object moved permanently, new location will be specified in this message
- 400: bad request, probably due to mistyping in the URL. Can also be due to something wrong on the server side
  - The API exists, but you called the wrong API request
- 404: requested file not provided by the server side ☹((((((
  - Also means API error. The API does not even exist
- 505: some internal errors in the server ☹

Jan 29, Tuesday

User-server state: stateful cookies

- Servers do not maintain information for clients
- Not so convenient, we want to keep track of the user data and let the server identify specific user

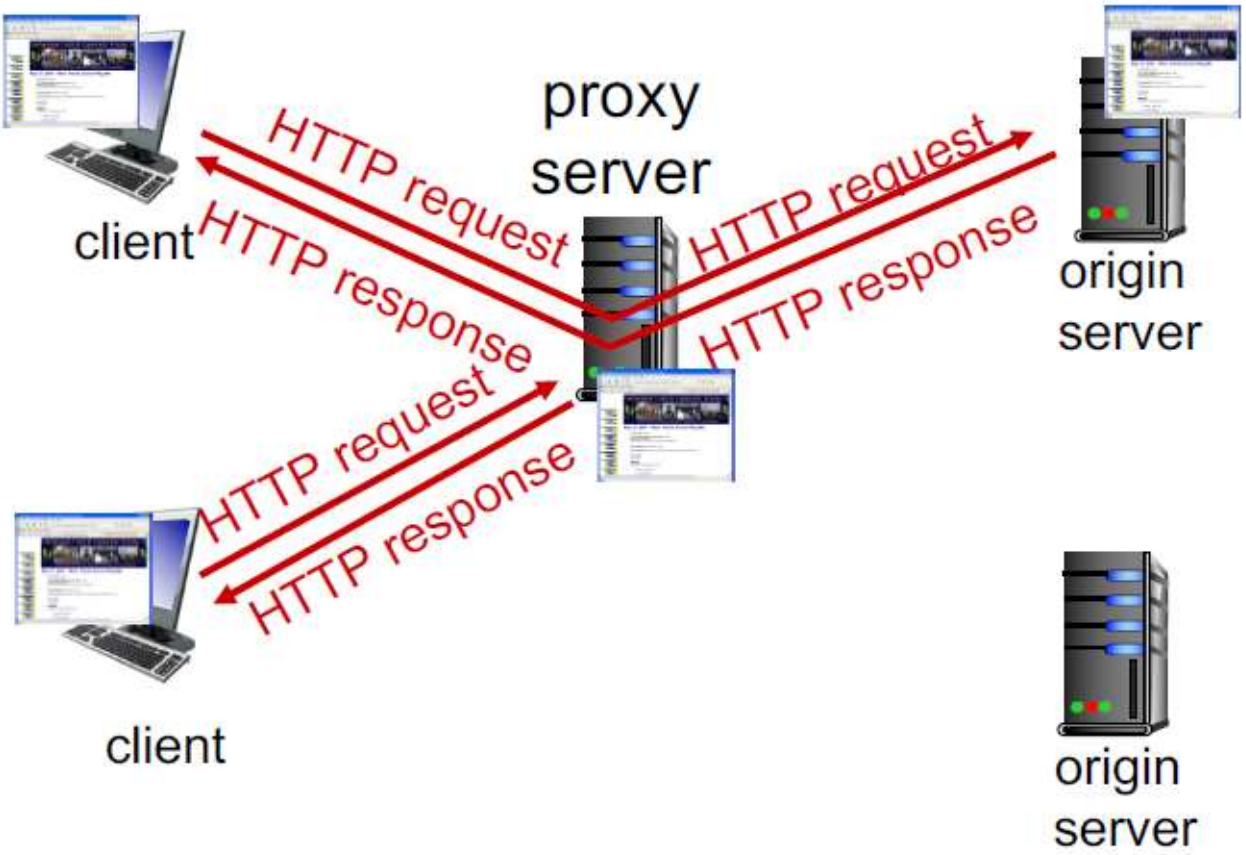
- eg: unique user ID on a website
  - your browser will use that ID upon the HTTP request and the server would know who you are



- For HTTP cookies, stored in the browser instead of devices
- **cookies can be used for**
  - authorization
  - shopping carts
  - recommendations
  - can also violate privacy, can be shut down
- how to keep state:
  - protocol endpoints: maintain state at sender/receiver over multiple transactions
  - cookies: http messages carry state

Web Caches (proxy server)

- Satisfying user request without involving origin server
- **Forward Proxy** (we are talking about this one):



- Closer to the client side
- Client first sends all HTTP request to cache for information
  - Client send HTTP request to proxy server
  - Proxy server process data
    - If information available, send HTTP response back to the client directly
    - Else request information from origin server with HTTP request, receive HTTP response from origin server
- cache acts as both client and server
- typically, cache is installed by ISP
- Reduces traffic through the public internet, saves client time
- Through HTTP protocol and origin server, the proxy server would know what to keep and how long to keep

Reverse Proxy

- Closer to the server side

Caching Example

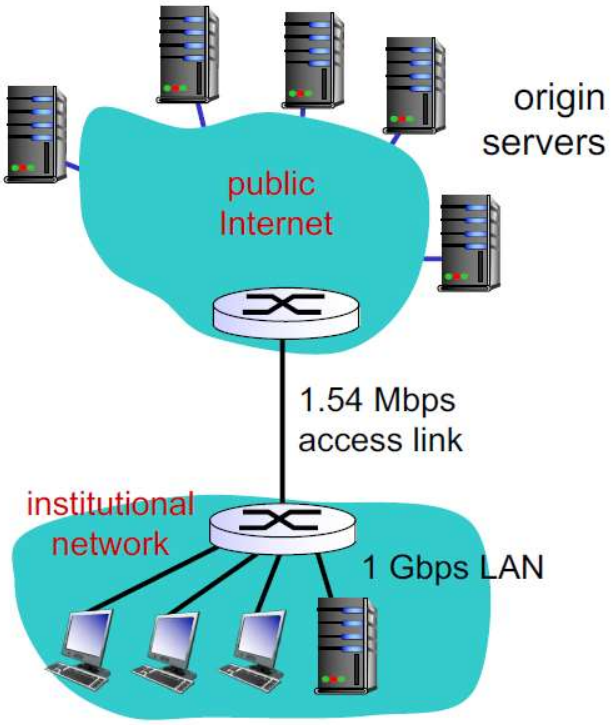
- original without proxy:

assumptions:

- avg object size: 100K bits
- avg request rate from browsers to origin servers: 15/sec
- avg data rate to browsers: 1.50 Mbps
- RTT from institutional router to any origin server: 2 sec
- access link rate: 1.54 Mbps

consequences:

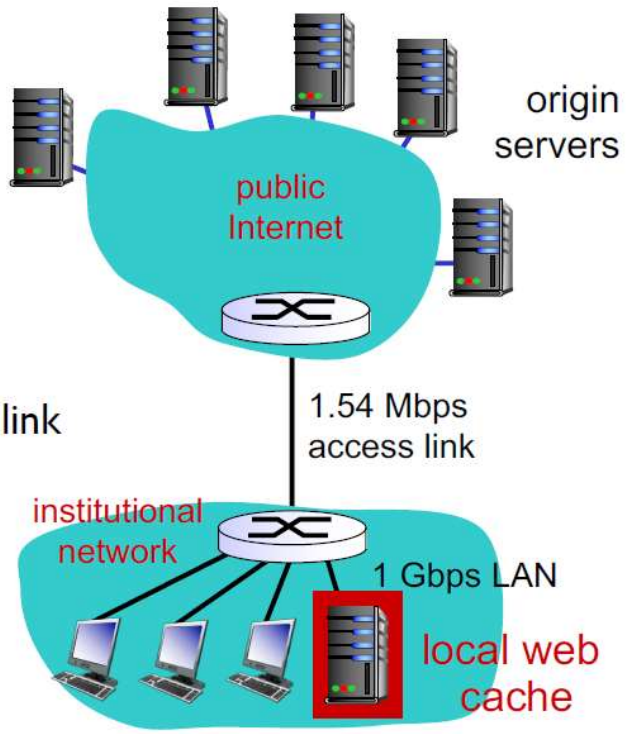
- LAN utilization: 15% *problem!*
- access link utilization = 99%
- total delay = Internet delay + access delay + LAN delay
  = 2 sec + minutes + usecs



- install local cache:

Calculating access link utilization, delay with cache:

- suppose cache hit rate is 0.4
  - 40% requests satisfied at cache, 60% requests satisfied at origin
- access link utilization:
  - 60% of requests use access link
- data rate to browsers over access link
  = 0.6*1.50 Mbps = .9 Mbps
  - utilization = 0.9/1.54 = .58
- total delay
  - = 0.6 * (delay from origin servers) +0.4 * (delay when satisfied at cache)
  - = 0.6 (2.01) + 0.4 (~msecs) = ~ 1.2 secs
  - less than with 154 Mbps link (and cheaper too!)



Without web cache mechanism

- Without using caching, access link becomes most utilized by the browsers to go directly to the origin server
- Total delay = internet delay + access delay + LAN delay

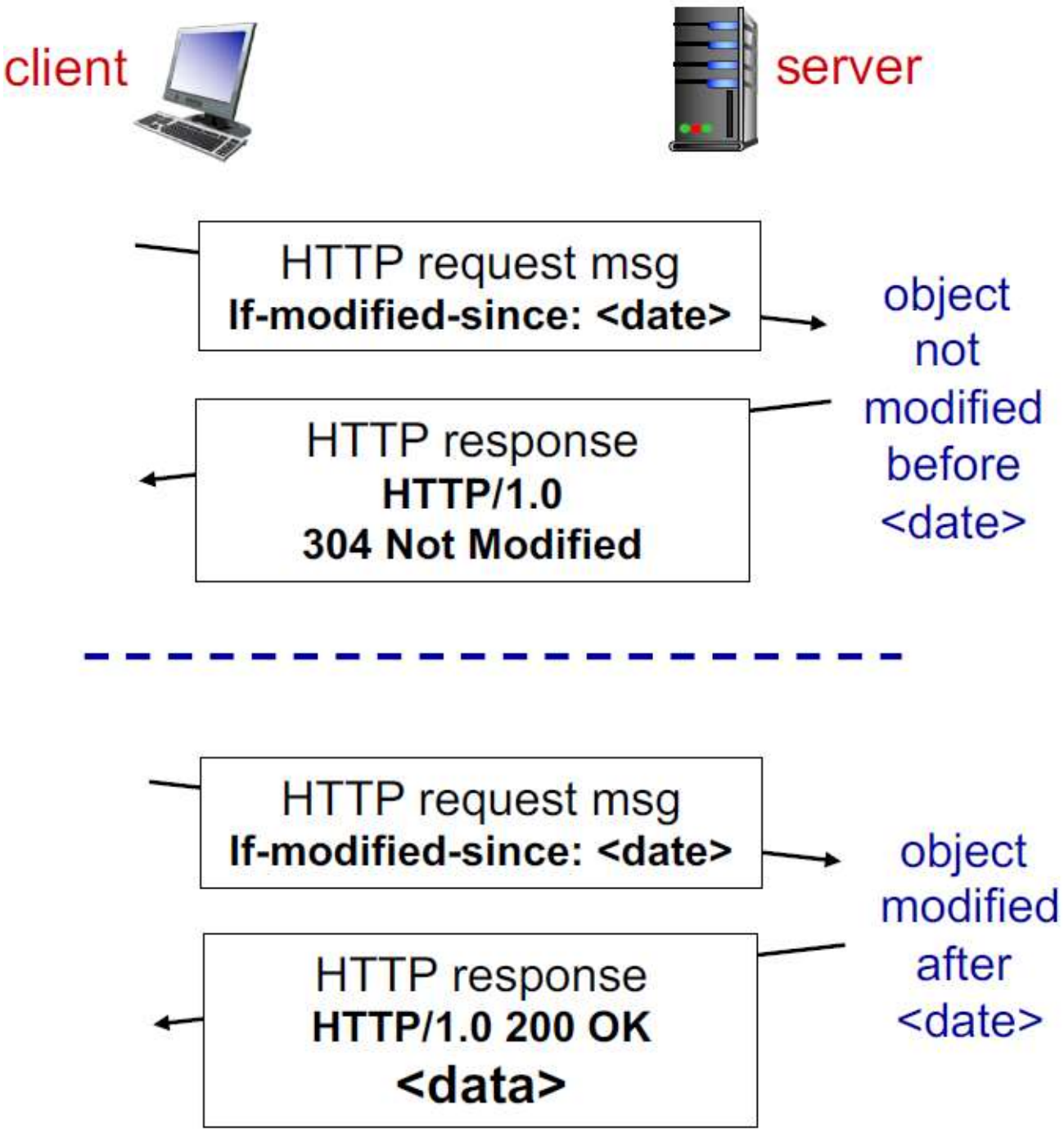
Increase access link capacity without web cache

- Time it takes to go through access link decreased to 10%!

Install local web cache server

- Attach a cache server to the local institutional server
- How to calculate total delay
    - Total delay that could be processed by the local web cache
    - Total delay that could only be processed by the origin server

Conditional GET

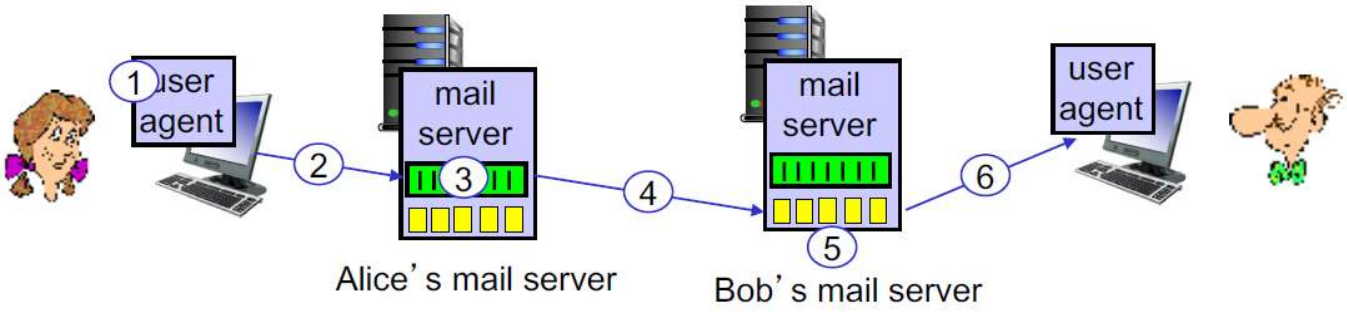


- Goal: the server won't send object if cache has up-to-date cached version in the web cache server
  - No object transmission delay
  - Lower link utilization
- Proxy (cache server) request if the data has been modified since a specific time from the origin server
- Server send back a message indicating if the data has been modified
  - If the object has not been modified since the date specified
    - HTTP 304 Not modified
    - No object attached
  - If the object has been modified
    - HTTP 200 OK

    - Object included

Electronic Mail

- Three major components
    - `user agents`
        - the mail readers
        - composing, editing, reading mail messages
        - eg: Outlook, iPhone mail client
    - `mail server`
        - mailbox contains incoming messages for user
        - message queue of outgoing messages to be sent
    - `SMTP` (simple mail transfer protocol)
        - protocol between mail servers for exchanging mail messages
        - **client side** of the mail server *send* message to **server side** of the mail server with `SMTP protocol`
- **Steps of sending emails**



1. User sender compose email on UA
2. UA transfer message to mail server of user sender with protocol depending on the specific user agent application
3. Client side of SMTP (the sender) set up `TCP connection` with the receiver's mail server to **allow** SMTP connection between two end servers (the hand shaking process with SMTP)
4. SMTP client transfers the mail message to recepeint's mail server with `SMTP protocol` over TCP connection
5. Closure of the connection
6. Recepient's UA retrieve the message from mail server using `mail retrieving protocols` (IMAP, POP3)

SMTP protocol

- SMTP uses **persistent** connections

- SMTP requires message (header and body) to be in 7-bit ASCII
- uses CRLF.CRLF to determine end of message
- SMTP client side `push` information to other mail server
  - unlike HTTP which `pull` information
- **multiple** objects sent in **multipart** message
  - unlike HTTP: each object encapsulatd in its own response message

Mail message format

- SMTP: protocol for exchanging email messages
- RFC 822: standard for text message format
  - header lines: To, From, Subject
    - Note: different from SMTP commands MAIL FROM, RCPT TO, etc.
  - body: the message body content, ASCII character only

Message access Protocols

POP3 protocol

- `POP`: post office protocol
- authorization, download
  - **authorization phase**:
    - `user:` decaler user name
    - `pass:` password
  - **transaction phase**:
    - `list:` list message numbers
    - `retr:` retrieve message by number
    - `dele:` delete
    - `quit`
- Two different modes
  - Download and delete
  - Download and keep

IMAP

- `IMAP`: internet mail access protocol
- more features, including manipulation of stored messages on server
- Keep all messages at server
- Allow user toorganize meesgae in folders
- keeps user state across sessions
  - names of folders and mappings between message IDs and folder name

HTTP

- used by gmail, hotmail, yahoo! mail, etc.

Feb 5, Tuesday

DNS (Domain Name System)

- Indentifier for internet hosts, routers:
  - IP address (32 bit): used for addessing datagrams
- Distributed database
- application-layer protocol
- Used to interprete host name to IP addresses

DNS services

- Translate hostname to IP address
- Host aliasing: canonical, alias names
  - `Canonical name`: Used to point one main domain to multiple subdomians
    - makes sure you don't need to change multiple records if you want to change your IP address
    - Structure: Alias name - IP address - canonoical name - subdomains?
  - Mail server aliasing: you simply use @case.edu instead of the full domian name
  - Load distribution: linking many IP addresses corresponding to one name

DNS structure

- Do not centralize DNS for the sake of scalability
- Centralized service cannot support all the flow?

DNS: distributed, hierarchical database

Take Amazon as an example?

- root servers: multiple root servers accross the globe managed by 13 organizations

TLD, authoritative servers

- top-level domain (TLD) servers:
  - eg: .edu, com, etc.

    - Responsible for com, org, net, edu, etc. and all top-level country domains
- authorative DNS servers:
    - organizations own DNS server, providing authorative hostnames to IP mappings for organizations named hosts
    - maintained by organization or service provider

Local DNS name server

- Does not striclty belong to hierarchy
- each residential ISP, company, university, has one "default name server"
    - also called default name server
- when host makes DNS query, query is sent to local DNS server
    - has local cache of recent name-to-address translation pairs (but may be out of date!)
    - acts as proxy, forwards query into hierarchy

DNS name resolution example

Iterated query

- Local DNS server is majorly in charge of generating all the requests to different servers
- Requesting host sending request to the local DNS server
- Local DNS server keeps querying other level of servers to search for the IP address if it does not have the cache of the IP address of a certain layer following the sequence below
    - root DNS server
    - TLD DNS server
    - authorative DNS server
- eg: the local DNS server queries the TLD DNS server directly if it has cached the TLD address, so it skips the root DNS server
- more secured because when attacked by hacker, only one way of information is hacked, and the user will be able to tell soemthing is wrong, unlike in the recursive query, where the amplification effect will work

Recursive query

- The contacted name servers are in charge of looking for the IP address and sending queries to each other
- Heavy load at upper level layers: root server, TLD DNS server, etc.

DNS: caching, updating records

- once (any) name server learns mapping, it caches mapping
    - cache entries timeout (disappear) after some time (TTL)

- TLD servers typically cached in local name servers
  - thus root name server not often visited
- cached entries may be out-of-date (best effort name-to-address translation!)
- update/notify mechanisms proposed IETF standard

DNS records

- DNS: distributed database storing resource records (RR)
- RR format: (name, value, type, ttl)
  - `type=A`
    - name is hostname
    - value is IP address
  - `tpe=NS`
    - name is domain
    - value is hostname of the authorative name server for this domain
  - `type=CNAME`
    - name is alias name for some "canonical" name
    - value is canonical name
    - eg: www.ibm.com = servereast.backup2.ibm.com
  - `type=MX`
    - value is name of mailserver associated with name

DNS protocol, messages

both query and reply messages have the sae message format

- message header
  - identification: 16 bit # for query, reply to query uses same #
  - flags:
    - query or reply
    - recursion desired
    - recursion available
    - reply is authoritative

Inserting recoreds into DNS

- register name gentsk.com at DNS registrar lmao
  - provide names, IP addresses of the authoritative name server (primary and secondary)
  - registrar inserts two RRs into .com TLS server
- Create authoritative server type A record for www.gentsk.com, type MX record for gentsk.com

Attackin DNS

- root servers are relatively safe
- domain level servers are more dangerous since more user information goes thru this level
- DNS applies no encryption machanism for the sake of efficiency, all messages are plain text

DDoS attacks

see slides for more details

Feb 7, Thursday

Socket programming

- goal: build client-server
- `socket`: door between application process and end- end-transport protocol
  - two socket types:
    - `UDP`
    - `TCP`

Socket with UDP

- no "connection" between client and server
- no handshaking before sending data
- sender explicitly attaches IP address and port number to each packet
- receiver extract sender `IP address` and `port number` from the packet
- transmitted data may be lost or received out-of-order
- provides unreliable transfer of groups of bytes (□datagrams□) between client and server

UDP communication process

- server run on serverIP
- server creates socket, port = x
- client create socket
- client create datagram with server IP and port=x, send datagram via clientSocket
- server read datagram from serverSocket
- server write reply to server socket specifying client address and port number
- client read datagra from client socket
- client close client socket

Socket with TCP

- server process must run first
- server must create socket that welcomes client's contact
- client must contact server
- client contacts server by creating TCP socket, specifying IP address, port number of server process
- when client creates socket, client TCP establishes connectiont to server TCP
- when server contacted by client, server TCP creates new socket for server process to communicate with client
- reliable communication between server and client

TCP ommunication process

- server create socket, port=x, for incoming request
- server wait for incoming connection request
- client create socket, connect to hostid, port= x
- client send request using clientSocket
- server read request from connectionSocket
- server write reply to connetionSocket
- client read request from client socket
- server closes connectionSocket
- client closes client socket

Chapter 3: Transport Layer

Transport layer services and protocols

- provide logical communication between app processes running on different hosts
- end to end system protocols
- Data units transferred within the unit: segments

Transport vs. network layer

- network layer: logical communication between hosts
- transport layer: logical communication between two end processes, relies on and enhances network layer
- processes communicate based on the communication between hosts

Reliable In-order delivery (TCP)

Unreliable unordered delivery (UDP)

Multiplexing and demultiplexing

- `multiplexing`: happens on sender end. Handles data from multiple sockets and transport header (later used for demultiplexing)
- `demultiplexing`: on receiver end. Use header info to deliver received segments to correct socket.

How demultiplexing works

host uses IP addresses & port numbers to direct segment to appropriate socket
32 bits

Connectionless demultiplexing

see slides, wait for review :x
used by UDP

Connection-oriented demultiplexing

- used by TCP, identified by 4-tuple
  - source IP address
  - source port number • dest IP address
  - dest port number
- demux: receiver 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

UDP: User Datagram Protocol

- Connectionless: no handshaking between UDP sender, receiver
- reliable transfer over UDP

UDP: segment header

- source port number
- destination portnumber
- IP address
- `checksum`: a signature computing the data you transmit. Used to detect if there are errors in the header.

    - the receiver compute the checksum of the received message and compares that with the checksum included in the header to see if there is an error
        - if doesnt match, there is an error
        - else we still dont know if its correct
    - checksum cannot be all zero since receiver will consider it as checksum not computed. we use all ones instead

Principle of reliable data transfer

- important in application, transport, link layers
- Most important API:
    - `rdt_send()`: called from above (by the app). Passed data to deliver to receiver upper layer
    - `rdt_rcv()`: called when packet arrives on rcv-side of channel
- use finite state machines (FSM) to specify sender, receiver

rdt1.0: Reliable transfer over a reliable channel

- underlying chennel perfectly reliable
- separate FSMs for sender, receiver:
    - sender sends data into underlying channel
    - receiver reads data from underlying channel
- very simple behaviors for both sender and receiver since both underlying layers are completely reliable
- Both sender and receiver have only one state. There are still state transitions, but always end up with the original state
- **Sender**:
    - wait for call from above
    - rdt_send(data)
    - packet = make_pkt(data)
    - udt_send(packet)
- **Receiver**:
    - Wait call from below
    - rdt_rcv(packet)
    - extract (packet,data)
    - deliver_data(data)

rdt2.0: Channel with bit errors

- underlying channels may flip bits in the packet
- still no packet loss
- `acknowledgements (ACKs)`: receiver explicitly tells sender that packet received OK

- `negative acknowledgements`: receiver explicitly tells sender that pkt had errors
  - sender retransmits pkt on receipt of NAK
- **Sender** has two states:
  1. Wait for call from above
  - rdt_send(data)
  - sndpkt = make_pkt(data, checksum)
  - udt_send(sndpkt)
  - transitions to state `2`
  2. Wait for ACK or NAK
  - rdt_rcv(rcvpkt) && isNAK(rcvpkt)
    - udt_send(sndpkt)
    - transitions to state `2`
  - rdt_rcv(rcvpkt) && isACK(rcvpkt)
    - transitons to state `1`
- **Receiver** has one state with two transitions:
  1. Wait for call from below
  - rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
    - extract(rcvpkt,data)
    - deliver_data(data)
    - udt_send(ACK)
  - rdt_rcv(rcvpkt) && corrupt(rcvpkt)
    - udt_send(NAK)
  - Go back to state `1`

Fatal Flaw

- The acknowledgement message can also be corrupted
- sender doesn't know what happened at receiver
- can't just retransmit: possible duplicate at receiver end
- **Solutions**:
  - sender retransmits current pkt if ACK/NAK corrupted
  - sender adds sequence number to each pkt
  - receiver discards duplicate pkt
- Sender use the stop and wait model
  - it will not send another packet if it does not receive any feedback message from the receiver
  - This guarantees the sender always sends only one packet at a time
  - Receiver gets to differentiate the duplicate packet

Feb 12, Tuesday

(Continue with rdt)

rdt2.1: handles garbled ACK/NAKs

- Upgrade the rdt2.0 model
- Doubles the number of states
  Refer to the slides, need revision
- **Sender**:
  - seq number added to pkt
  - two seq. #s (0,1) will suffice
  - must check if received ACK/NAK corrupted
  - state must remember whether expected pkt should have seq # of 0 or 1
  1. wait for call 0 from above
     - rdt_send(data))
- **Receiver**:
  - must check if received packet is duplicate
  - state indicates whether 0 or 1 is expected pkt seq #
  1. Wait for 0 from below
     [Attach images here later]

rdt2.2: NAK-free protocol

Alter from the rdt2.1 by dropping NAK message and have one more check message function
[Attach images here later]

rdt3.0: channels with errors and loss

- new assumption: underlying channel can also lose packets (data, ACKs)
- approach: sender waits reasonable amount of time for ACK
- retransmits if no ACK received in this time
- if pkt (or ACK) just delayed (not lost):
  - retransmission will be duplicate, but seq. #'s already handles this
  - receiver must specify seq # of pkt being ACKed
- requires countdown timer

[Attach images here later]

rdt3.0 in action

[Attach images here later]

Performance of rdt3.0

- rdt3.0 is correct, but with low performance
- [See slides for datails on the example]
- want to calculate utilization of the sender
  - ratio between transmission time and total amount of time:

$$U_{sender} = \frac{L/R}{RTT + L/R}$$

- network protocol limits use of physical resources

Pipelined protocols

- `pipelining`: sender allows multiple, in-flight, yet- to-be-acknowledged pkts
- two generic forms of pipelined protocols:
  - `go-Back-N`
  - `selective repeat`
- increase utilization by sending multiple pkts together:
  - eg: three packets, then utilization of sender becomes three times as large:

$$U_{sender} = \frac{3L/R}{RTT + L/R}$$

Go-back-N

- `sender`: can have up to N unacked packetes in pipeline
- `receiver`: only sends cumulative ack
  - does not ack packet if there's a gap
- Sender has timer for *oldest unacked packet*
  - when timer expires, retransimit all unacked packets
  - original design: only one timer. but now can have multiple timers?

The logic is:

- **Sender**:
  - k-bit seq number in pkt header
  - window of up to N, consecutive unack'ed pkts allowed
  - [Attach images here later]
  - two `pointers` in the window to point to the `sender base` and the `nextseqnum`
  - ACK(n): ACKs all pkts up to and including seq # n: cumulative ACK
    - may receive duplicate ACKs (see receiver)
  - there is only one timer, and it is for oldest in-flight pkt
  - timeout(n): retransmit packet n and all higher seq # pkts in window
  - [Attach images here later for sender FSM]

- **Receiver**:
  - applies cumulative acknowledgement
  - drop all successive packets if one intermediate packet was loss
  - resend ack for the last packet received in correct order until the loss packet was received
  - ACK-only: always send ACK for correctly-received pkt with highest in-order seq #
    - may generate duplicate ACKs
    - need only remember `expectedseqnum`
  - the receiver does not buffer out-of-order pkt, instead, the receiver discards all packets untill receiving the packet in the correct order
  - re-ACK pkt with highest in-order seq #
  - [Attach images here later for receiver FSM]
- [Attach images here later for GBN in action]

Feb 14, Thursday

- **GBN window size and sequence number**
  - [Attach images here later for GBN erroneous example]
    - `window size` $= 2^k = 4$
    - `k` : length of sequence number
    - would not work because of the oversize window size
  - [attach image for correct example]
    - `window size` $= 2^k - 1 = 3$
  - Correct window size should be at least 1 less than the 2's exponential of sequence number
- **real life case in TCP**
  - the size of increment in receiver's ACK value is related to the lenght of the data sent to the receiver

Selective Repeat:

- sender can have up to N unack'ed packets in pipeline
  - sender only resends pkts for which ACK not received
- receiver sends individual ack for each packet
  - receiver buffers pkts, as needed, for eventual in-order delivery to upper layer
- sender maintains different timers for each unacked packets
  - when timer expires, retransmit only that unacked packet

The logic is:

- **sender**:
  - data from above:
    - if next available seq number 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 number
- **receiver**
    - pkt n in `[rcvbase,rcvbase+N-1]`:
        - send ACK(n)
        - buffer out of order pkt
        - deliver buffered in order pkts, advance window to next not yet received pkt
    - pkt n in `[rcvbase,rcvbase-1]`:
        - ACK(n)
    - else:
        - ignore
- **Selective repeat dilemma:**
    - see slides, attach image here

Connection oriented transport: TCP

TCP segment structure

- acknowledgement number: used by receiver to indicate what seq number receiving and what seq number expecting
- sequence number: counting by bytes of data instead of segments
    - so is acknowledgement number
- URG: urgent data
- ACK: ACK # valid
- receive window: how many more bytes of data the receiver is willing to accept

TCP seq. numbers, ACKs

- TCP protocol is very similar to GBN
- [attach image of segement sample]
- `sequence number`: first byte of sent segment data
- `acknowledgements`:
    - next byte expected by the receiver
    - cumulative ACK
- both numbers incremented according to the length of data segments
- TCP doesn't specify how to handle out of order segments, addressed by implementor
- [attach example]

    - two hosts tranferring data, both sender and receiver

TCP round trip time, timeout

wait for reviewing....

TCP reliable data transfer

- TCP creates rdt service on top of IP's unreliable service
    - pipelined segments
    - cumulative acks
    - single retransmission timer
    - allows out of order packets
- retransmission is triggered by two factors:
    - timeout events
    - duplicate acks
- **sender events**:
    - data rcvd fomapp:
        - create segment with seq #
        - seq # is first byte of sent segment data
        - start timer if not already running
            - think of timer as for oldest unacked segment
            - expiration interval: `TimeOutInterval`
    - time out:
        - only retransmit segment that caused timeout
            - unlike GBN which transmitted all successive packets
        - restart timer
    - ack rcvd:
        - if ack acknowledges previously unacked segments
            - update what is known to be ACKed
            - start timer if there are still unacked segments
    - [attach simplified FSM for TCP sender here]
- **receiver events**
    - event at receiver
        - [image]
    - TCP receiver action
        - [image]
- **TCP retransmission scenarios**
    - [attach image]
- TCP fast retransmit
    - time-outperiod often relatively long: long delay before resending lost packet

    - 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.

Flow control

- receiver controls sender, so sender won't overflow receiver's buffer by transmitting too much, too fast
- receiver advertises free buffer space by including rwnd value in TCP header of receiver-to-sender segments

Connection management

- sender and receiver handshakes before exchanging data
    - agree to establish connection (each knowing the other willing to establish connection)
        - `req_conn(x)` + `acc_conn(x)`
    - agree on connection parameters