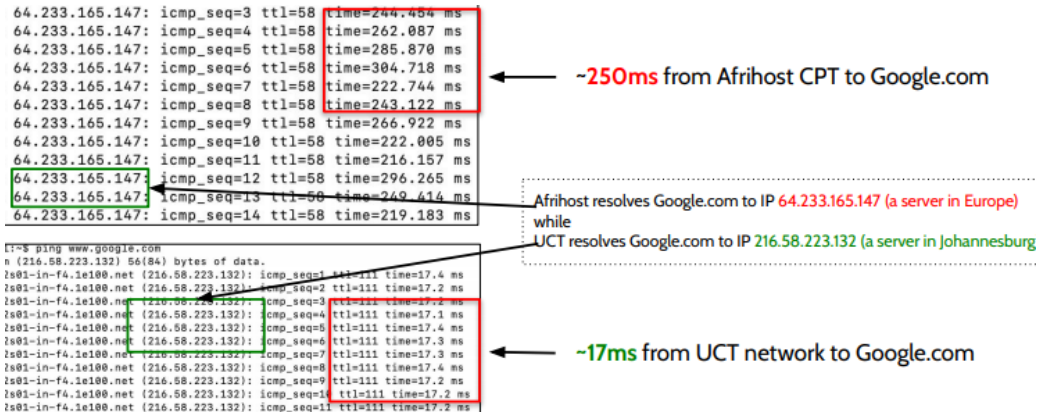
NETWORKS
CSC3002F

Textbook:
Exercises: https://www.computer-networking.info/exercises/html/
MCQs: https://www.computer-networking.info/web/mcq-ex/html/

Introduction

- Key questions:
  - How is info flowing from wherever it is, to a particular host on the internet?

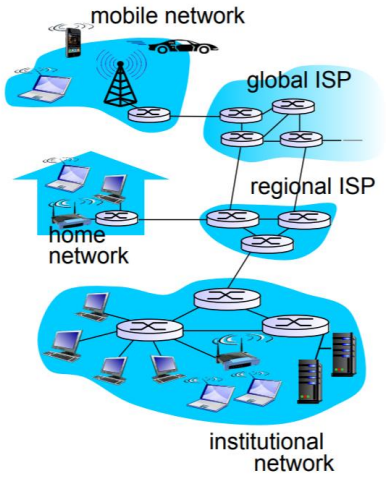


  - Why resolve to a different address when you ping the same server?
  - Systems and protocols allow your machine to change from a server name to an IP addr
    - What mechanisms allow the change e.g. from google.com to an IP addr?
    - How can one get different IP addrs to the same domain?
      - And the impact thereof?
  - How different networks and ISPS interconnecting with each other?
    - And how that affects performance?

The Internet

- Internet is complex strc
  - Many different Internet Service Providers (ISPs)
  - Physical strc underpins it (undersea cables, etc)

"Nuts and Bolts" view

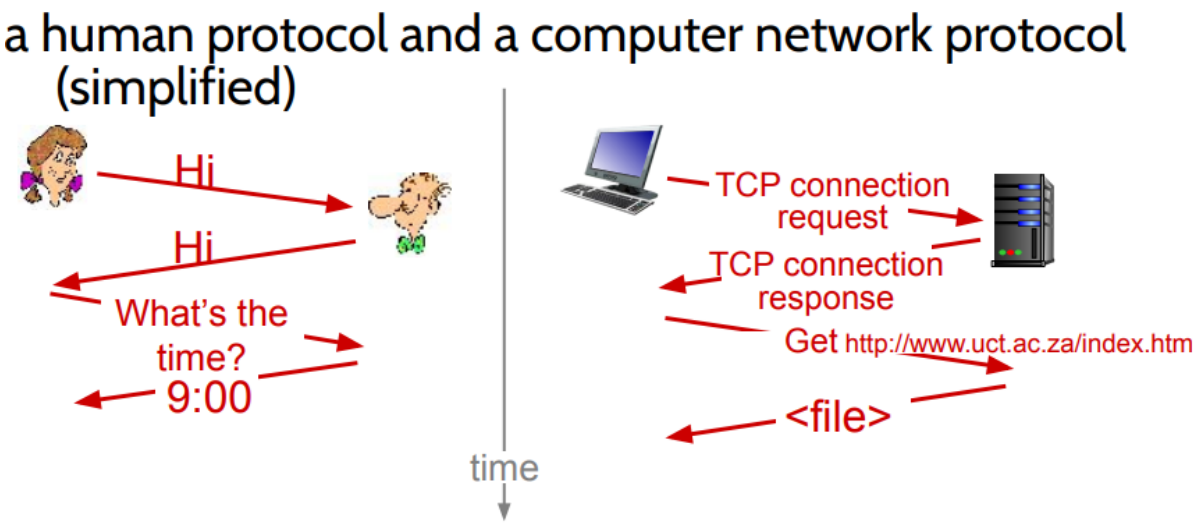


- Components:
  - Hosts
  - Communication links
  - Packet switches
  - These 3 form the internet
- Millions of connected computing devices
  - Where info flows from/to
  - Hosts = end systems
    - End hosts – e.g. like Google; info flows from there
  - Running network apps
    - E.g. your browser (network app) – web server on the other end accepts request and sends content that has been asked for
- Communication links
  - Physical strcs that allow devices to connect to each other
  - Fibre, copper, radio, satellite
  - E.g. connecting your phone to WiFi
  - Bandwidth = transmission rate
    - How much info can flow in that link in a unit of time
    - Some links have more capacity – fibre has higher capacity
- Packet switches
  - Forward packets (chunks of data)

- Routers and switches
- Devices that allow the connection – receive info that is passing from one network to another
- Process that info and forwards it to the nxt part of the network
- Internet is complex – at each point there can be several outgoing links and packet switches determine which is the *correct* path to forward that info to (so that it reaches its destination)

- Internet = "network of networks"
  - Interconnected ISPs
  - Regional ISPs only work in a country/province
    - Can connect it global ISP – allows it to transmit to ISPs in other parts of the world
- Protocols – control sending, receiving of msgs
  - This type of msging allows different devices to talk to each other
  - E.g. TCP, IP, HTTP, Skype
- Internet standards
  - Type of specification that's well understood by the service providers, vendors and device manufacturers
  - These standards allow different types of devices to connect to each other
  - E.g. Router manufactured by Cisco , Huawei – since they follow the same standard, they can communicate with each other and share info between themselves
    - Emails – can send emails via Outlook (email gets formatted in a specific way) to someone and that person will receive it (even if they use a different email client)
  - RFC: request for comments
  - IETF: Internet Engineering Task Force

Protocols



- Host that wants to initiate communication may send a request
  - E.g. TCP – want to connect via TCP connection
    - Receiving machine expects msg to come via a specific port / specific way so that it understands that it this is a TCP request

- ◊ If server machine accepts request, sends a msg back
- ◊ Once initial device (device that sent request) receives acknowledgement, then proceeds to send request for what it actually wants e.g. request webpage
- ◊ Response from server is then that file being transmitted

∇ Have specific msgs sent as requests and specific msgs that can be received as responses

∇ Protocols define format, order of msgs sent and received among network entities, and actions taken on msg transmission receipt

- ◊ All communications activity in internet governed by protocols

LAYERS

∇ Different types of devices and systems that need protocols – hence, need protocols that operate at different layers

∇ Networks are complex, with many "pieces":

- ◊ Hosts
- ◊ Routers
- ◊ Links of various media
- ◊ Apps
- ◊ Protocols
- ◊ Hw, sw

∇ E.g. have one computer (host) trying to speak to a server

- ◊ Between these two hosts, there are switches which also need to be able to understand each other
- ◊ The understanding between the switches may be different to that of the hosts – thus, this is another *layer*
- ◊ Different protocols may be needed

∇ E.g. have browser and web server – these need to understand each other

∇ Protocols need to be understood at different layers – protocols that operate at host level will differ to those of switching devices

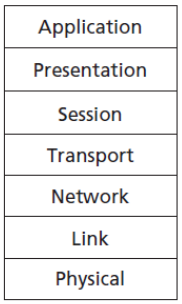
∇ Each layer implements a service

- ◊ Via its own internal-layer actions
- ◊ Relying on services provided by layer below

∇ Need layering since dealing with complex sys:

- ◊ Explicit strc allows identification, relationship of complex system's pieces
  - ♥ Layered reference model
- ◊ Modularisation eases maintenance, updating of sys
  - ♥ Can change one part of the sys without affecting the other parts
  - ♥ Change of implementation of layer's service transparent to rest of system
  - ♥ E.g. if protocol used to connect device to WiFi changes, the protocol for web browsing won't change

ISO/OSI Reference model



b. Seven-layer ISO OSI reference model

- ∇ International Standards Organisation Open Systems Interconnection Reference Model
- ∇ Defines what functions should happen at the different layers of the network
  - ◊ This model is a guide used in network design and doesn't assist with network implementation
- ∇ Initial focus on devices and apps that run on them – what network apps are running on the end points where communication begins
  - ◊ As layers are traversed, deal with what happens in the network to move the signal from one device to the next, etc.
  - ◊ Top – layers focused on E2E / device-to-device and lower layers have more detail
- ∇ (1) Application: supporting nw apps
  - ◊ E.g. web browsing uses HTTP (app layer protocol)
- ∇ (2) Presentation: allow apps to interpret "meaning" of data
  - ◊ Concerned with how info that gets sent between apps is presented to the user
  - ◊ E.g. encryption, compression, machine-specific conventions
- ∇ (3) Session: synchronisation (that happens between two communicating processes/apps), checkpointing, recovery of data exchange
  - ◊ Sending and receiving device have to synch ito how many bits have been sent, what next bit is expected
    - ♥ Needed so that if there's an error in transmission, receiving process can inform the sending process that there's some data missing, or (if network connection disrupted and restarted) how much progress the exchange made.
  - ◊ Need to maintain session between processes that're exchanging info
- ∇ (4) Transport: process-to-process data transfer
  - ◊ Process-to-process communication between processes that're running
  - ◊ App can have several processes that support it
  - ◊ E.g. web browser has many processes to support it sending data
  - ◊ Processes operate at the transport layer ito knowing which specific port the data is sent from (example) and which port it's received from
  - ◊ Specific processes and how they connect to each other
- ∇ App layer to transport layer – everything happens E2E (src host to destination host)
- ∇ (5) Network: routing of datagrams from source to destination
  - ◊ Moving data from src to destination network

- ◊ Routing – determining paths that data must move through – is one of the functions of the nw layer
- ◊ E.g. router operates at the network layer

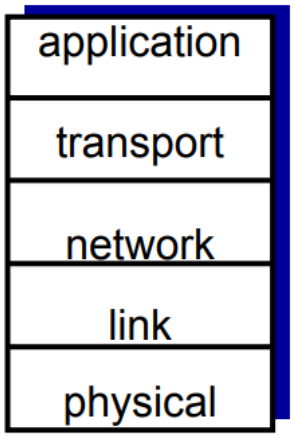
∇ (6) Link: transfer between neighbouring nw elements

- ◊ Within network but unconcerned with E2E movement of data – concerned with movement of data from one device to the next
- ◊ E.g. phone connects to internet via WiFi – link layer concerned with how data is going to move from phone to the router
  - ♥ When data received at router, link layer at the router is concerned with moving data from the router to the next device on the nw which will process that data
- ◊ Concerned with processes that happen at each particular link
- ◊ Different types of links – wireless, fibre optic, etc – each with different protocol supporting it

∇ (7) Physical: encoded bits "on the wire"

- ◊ How data gets encoded on the medium
- ◊ E.g. transmitting on wireless link – data has to be converted into radio freq signal and transmitted; for fibre has to be converted into optic signal
- ◊ Depends on physical media being used – signal converted appropriately

Internet Protocol Stack and TCP/IP suite

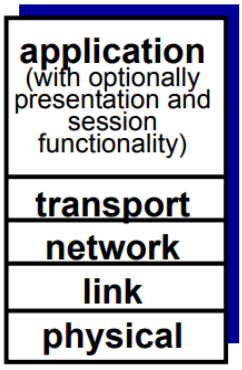


∇ Internet built on TCP/IP (transmission control protocol/internet protocol)

- ◊ These two are the main building blocks of the stack but aren't the only protocols used

∇ Five layers – compared to OSI with seven (for IP stack, some layers embedded in app layer)

- ◊ Missing presentation and session layer
- ◊ If needed, services must be implemented in app layer



- (1) Application: supporting nw apps
  - FTP, SMTP, HTTP (some examples)
- (2) Transport: process-to-process data transfer
  - Actual process that is supporting the app – how these two processes from src to destination are engaging (exchanging control msgs, checking how many msgs have been received, etc)
  - How processes "hook" to each other
    - E.g. client process has to hook to server process and begin msg exchange
  - TCP (connection-oriented protocol) – before start exchanging msgs, there's a connection set up between src and destination processes
  - UDP – src begins to send msgs and receiver begins to process them
- (3) Network: routing of datagrams from src to destination
  - Trying to find "gates" where packets must flow through to reach the destination
  - IP (how networks are addressed), routing protocols
- (4) Link: transfer between neighbouring nw elements
  - Ethernet, 802.111 (supports WiFi to LAN)
- (5) Physical: encoded bits "on the wire"
  - Encodes info into specific signal representation for that type of medium
- NB – focus from app layer to link layer for this course

Encapsulation

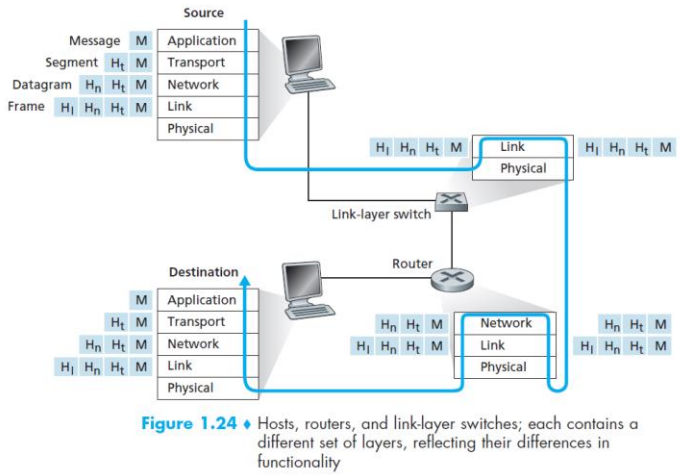


Figure 1.24 ♦ Hosts, routers, and link-layer switches; each contains a different set of layers, reflecting their differences in functionality

- Mnemonic for order – **A**ll **T**igers **N**eed **L**ong **P**aws

- App layer:
  - Have msg generated by app (e.g. email)
  - Once app layer processes that msg, passes it down to layer below
    - NB that layers below provide services to layers above them =>app relies on transport layer
- Transport layer:
  - Receives msg and follow encap process
  - Encaps msg in its own header (in "another envelope")
    - Helps identify msg and where it must go to from the transport layer
    - Msg = segment
    - Header contains e.g. port # of src and at destination; adds more specific info about which ports must be used in order to be able to leave this device and arrive at the destination device
  - Once header is added, pushes msg down to network layer
- Network layer:
  - Encaps msg further – adds its own header which includes info necessary for routing (moving msg from src to dest nw)
  - Examining IP address – pckg comes from a particular network (add that IP address) and needs to go to a particular destination address
  - Once header is added, pushes msg down to link layer
- Link layer:
  - Adds another header – contains physical address (e.g. go to the interface on this router to which it's connected)
- Adding "envelopes" – adding info at each layer
  - Info added at each layer only makes sense for the subsequent layers across the network
    - E.g. header added at transport layer only makes sense once msg is received on the other side by the transport layer on the receiving device at the destination
      - Along network, transport header is ignored (transport is process to process)
  - Process Example:
    - At the router, it will receive the msg as physical signal, pushes through link layer and reach the nw layer. Since router is nw layer device, it will examine nw layer header info – src and dest IP addresses
      - Based on header info, router determines where to send msg to
    - At dest, same process happens in reverse – physical gets converted and passed to link; link decodes and check which port to send to, passes to network; network interface receives msg and gets passed to transport layer; transport layer determine which specific process needs to receives msg using header – passes it to app that is supposed to receive that msg.

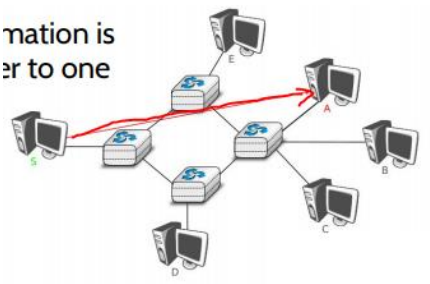
Network Structure

SCALE/SCOPE

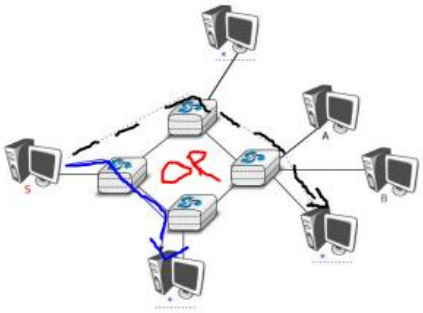
- LAN: local area network typically interconnects hosts that're up to a few/few tens of kms apart
- MAN: metropolitan area network typically interconnects devices that're up to a few hundred kms apart
- WAN: wide-area network interconnects hosts that can be located anywhere on Earth.
  - E.g. Google

TRANSMISSION MODE

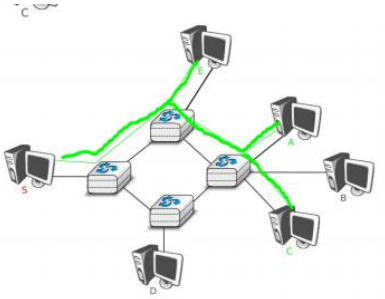
- Unicast: info sent by *one* sender to *one* receiver
  - Most common
  - E.g. downloading a file



- Anycast: info delivered to one receiver that belongs to a set
  - E.g. have several servers with same function – client needs service from servers (any server could perform the function)
    - Anycast puts servers that have same function in the same class – so when client makes request for transmission, data can be received by any one of those servers in the anycast set
  - Anycast protocol determines which to send data to– depends on # of factors



- Multicast: info sent to a set of recipients
  - Receivers part of multicast group
  - Any msgs sent to group is received by all group members – msg is sent from src and replicated so that all receives it
  - Only send **one** msg from src (not several)

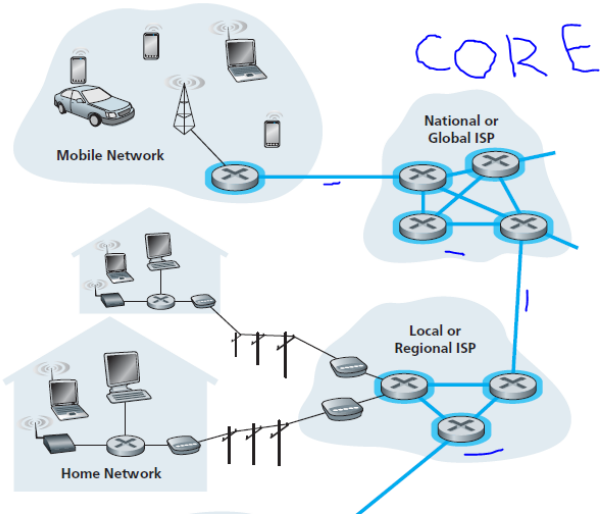


- Broadcast:
  - Msg sent to all nodes that're part of that nw (all devices see msg)

Sections/Parts of the Internet

- Network edge:
  - Where hosts reside; where info flows from/to
  - E.g. campus network; home network
  - Hosts – clients and servers
  - Servers often data centres (where info flows from typically)
- Access networks/physical media:
  - Wired, wireless communication links
  - Networks that allow connection to the internet
  - E.g. for Home network, have ISP that provides link (like fibre) to the internet
- Network core:
  - Interconnected routers/switches which allow different networks to interconnect and traffic to flow from src to dest
  - Allow movement of traffic from one src nw, through several nws in between to the dest nw
  - Have specific protocols and systems which allow it to func
  - Nw of networks

NETWORK CORE



- Mesh of interconnected routers

- ◊ Devices that will receive info and determine which path the data must be pushed to

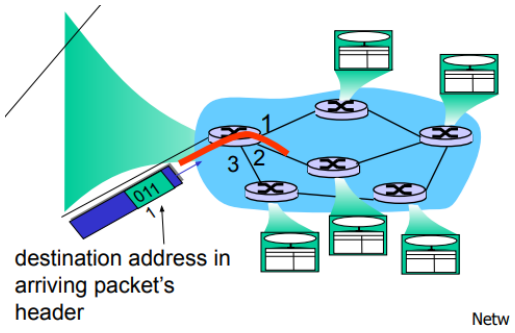
∇ Packet-switching:

- ◊ Hosts break app layer msgs into (smaller units of) packets
  - ♥ As each packet reaches a router, router examines info about dest in header of packet and determines which way to send it out.
  - ♥ Fwd packets from one router to the next, across links on path from src to dest
- ◊ Key – taking msgs from src to dest, looking at address stored in header

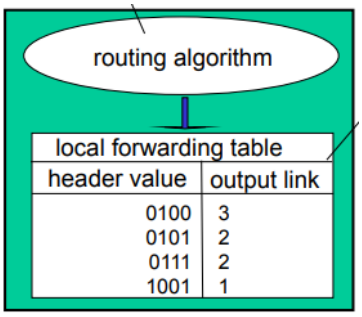
Functions

∇ (1) Routing: (routers) determines src-dest route taken by packets (units of data)

- ◊ Routing algos – before data comes, router examines:
  - ♥ strc of nw, how it's connected it to internet and how it's connected to other dests in the nw
- ◊ Calculates a path for each dest



- ◊ Then it will create **forwarding table** – this indicates that if the dest is x, then need to send packet through gate y
  - ♥ Forwarding table includes header info about dest and outgoing link



∇ (2) Forwarding: Move packets from router's input to appropriate output

- ◊ Router looks at forwarding table and determines destination
- ◊ Forwards packet to appropriate gate

Network of Networks

∇ End systems connect to internet vis ISPs

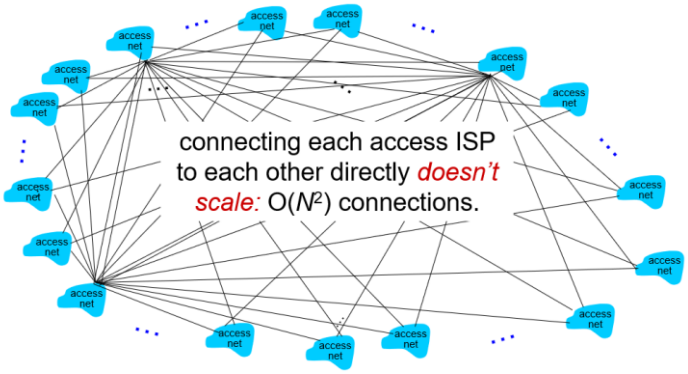
∇ Access ISPs in turn must be interconnected

∇ Result: complex nw of nws

∇ How to connect millions of ISPS together?

◊ Naïve approach: connect each access ISP to each other directly

♥ Doesn't scale well – O(N^2) connections



APPROACHES

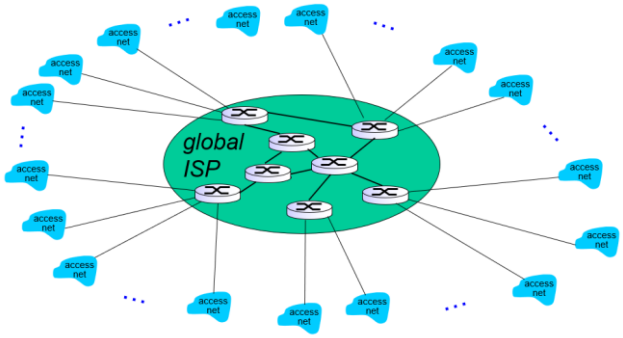
◊ Connect each access ISP to global transit ISP

♥ Customer and providers ISPs have economic agreement – provider ISPs charge customer ISPS for transmitting their data

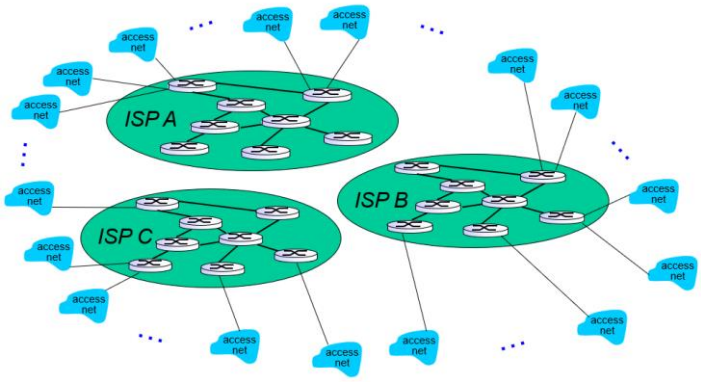
- All access nws then need to pay global ISP for transmission of their data
- With only one global ISP => monopoly and can charge any price
- Global ISP could then cut-off any customer ISP if they so choose – customer ISP then has no other way of connecting to the internet

♥ Single point of control – who will control it?

- Public/private enterprise, owned by a state?... etc



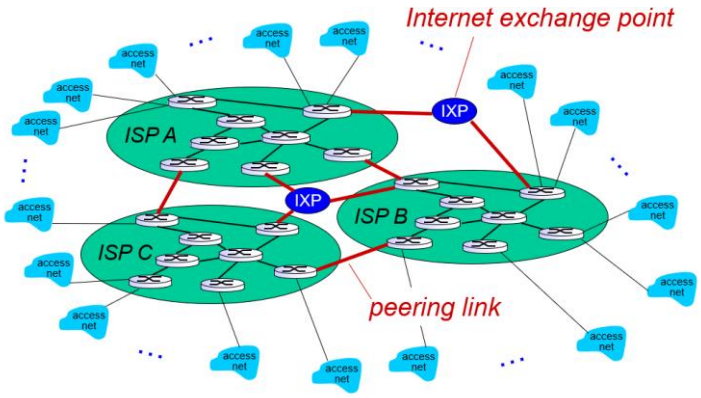
◊ Competitors: If one global ISP viable, there'll be competitors



- ♥ Can have many global ISps, each with its own share of access nws that it's connecting
- ♥ Could then end up with fragmented internets – if e.g. connected to access nw in ISP A, may not be able to communicate with users connected to ISP B

◊ Competitors (interconnected): can have competitors that're interconnected

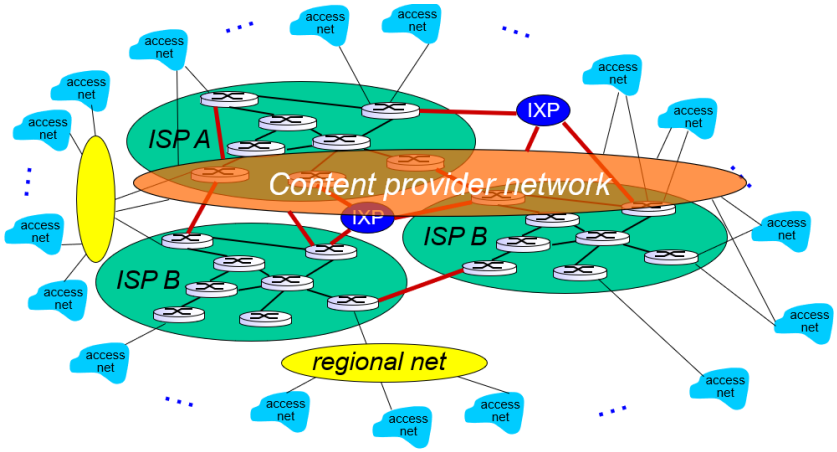
- ♥ Then have **Internet Exchange Point (IXP)** – physical point which allows traffic to flow between the networks
- ♥ ISPs can also be connected directly (without going through IEP) – via **peering link**



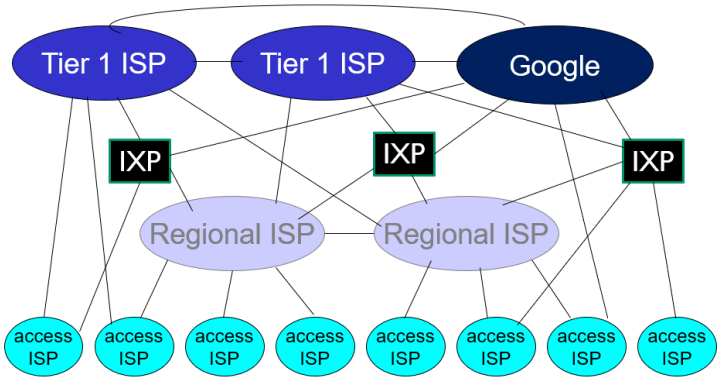
∇ Regional networks: connect access nws to ISPs

- ◊ E.g. regional ISP that only operates within continent or a country
- ◊ Nw connects all different access nws e.g. in a country – that regional nw allows them to exchange traffic within the region
- ◊ Useful because without regional nws e.g. have access nws in JHB and CT – if JHB and CT connect via ISP C (diagram above) then traffic that needs to go from CT to JHB goes through ISP C, through possibly another city like London before travelling back to JHB
  - ♥ Regional nws allow local interconnection (avoids above problem)

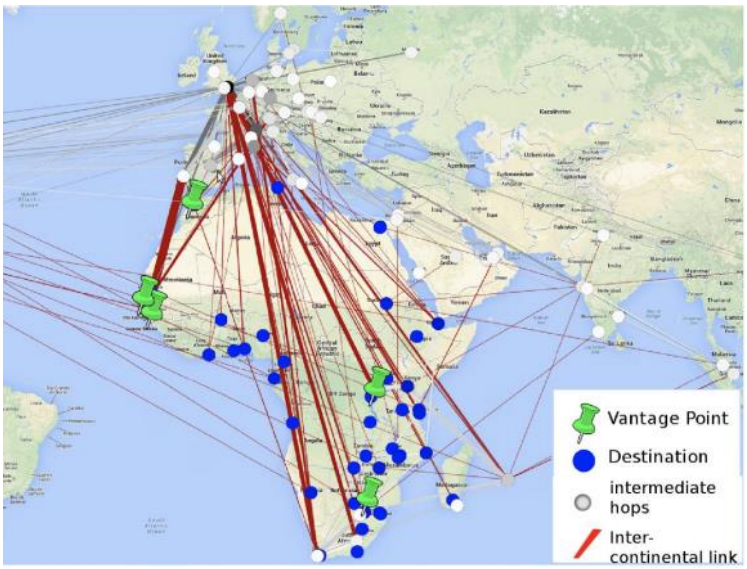
∇ Content Provider networks: e.g. Google, Microsoft, may run their own nw to bring services, content closer to end users



ISP TIERS



- ∇ Tier 1: operators with global reach
  - ◊ Normally have nodes in major world cities
  - ◊ IXPs connect to them
  - ◊ E.g. Level 3, Sprint
  - ◊ At centre – have small # of well-connected large networks
    - ♥ National and International coverage
- ∇ Content provider network:
  - ◊ Private network that connects its data centre to internet
    - ♥ Often bypass tier-1, regional ISPs
- ∇ Local:
  - ◊ Limited local interconnection results in circuitous routes



Application layer

- See the Internet as a service:
  - Infrastructure that provides service to apps
    - Web, games, social networks
    - This is how end users see it – they don't need to understand *how* it works
  - Developer's perspective: provides programming interface to apps
    - Hooks that allow sending and receiving app programs to "connect" to Internet
      - How will you connect apps together so that they can speak to each other online?
      - APIs are like sw intermediary that allows sw apps to communicate with each other
    - Provides service options, analogous to postal service

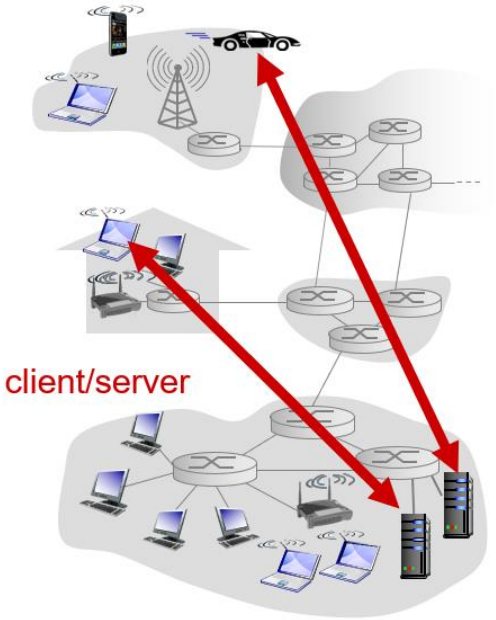
NETWORK APPLICATIONS

- Programs that:
  - Run on different end systems (devices where apps run e.g. PC, mobile device)
  - Communicate over nw
  - E.g. web server sw communicates with browser sw
- No need to write sw for nw-core devices when developing nw apps:
  - Nw-core devices don't run user apps
  - Apps on end systems allows for rapid development and propagation

Application Architectures

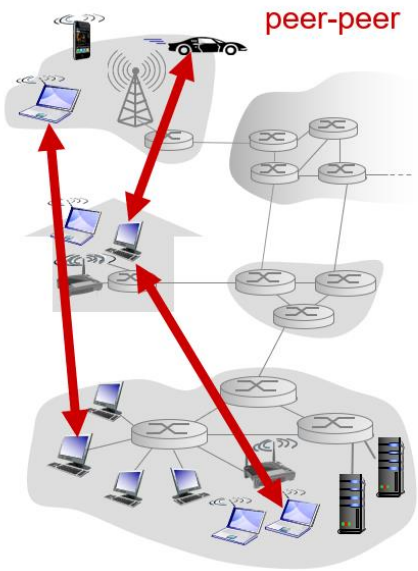
- Possible strc of apps:
  - (1) Client-server: "heavy" or "thin" client
    - Build server which is main system that receives requests, and processes whatever other machines are looking for.
    - Clients then send requests to server for info/processing
    - Clients more limited than server (have fewer capabilities)
  - (2) Peer-to-peer: (P2P)
    - Allows end systems to communicate directly, without having intermediary server to manage/control that communication
    - Direct exchange of info between devices

CLIENT-SERVER ARCHITECTURE



- Server:
  - Always-on host – always ready to receive requests/connection from client
  - Permanent IP address – permanent identifiers on networks
    - If client knows identifier, can send request to the address
  - Data centres for scaling of server capacities
    - E.g. Google services running in data centre – have several servers in same location responding to client requests, providing single service (have several servers responding to one request so one doesn't get overwhelmed).
  - Need to have it located near infrastructure e.g. place with backup power, good connectivity
- Clients:
  - Communicate with server and requests services
  - May be intermittently connected
    - E.g. browser on PC not on all the time, can be turned off and then on – but still able to connect to server
  - May have dynamic IP addresses
    - As device moves, IP address can change but service still maintained
  - Don't communicate directly with each other
    - E.g. when send an e-mail, it doesn't go directly from your email client to the receiver's one – it goes to uct server and then once receiver is online, email is sent to receiver

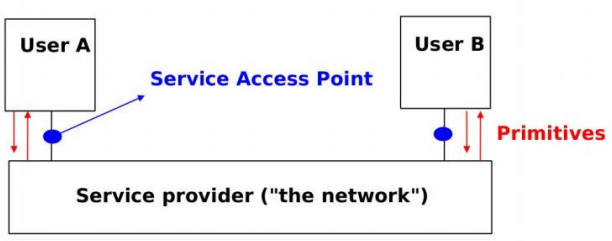
P2P ARCHITECTURE



- ∇ No always-on server
- ∇ Arbitrary end systems directly communicate
- ∇ Peers request service from other peers – also provide service to other peers
  - ◊ Compared to Client-Server arch, where you're just requesting service
  - ◊ Self-scalability: new peers bring new service capacity, as well as new service demands
    - ♥ More nodes on peer network = more capacity, more services can be provided
    - ♥ Example:
      - have 3 nodes in nw and want to share file.
      - File is available on one node and two nodes try and request file at same time.
      - Overall bandwidth available to one node is limited because file is only available in one place with one link connecting into the nw – link gets overwhelmed quickly
      - If more nodes join nw and they also have that file – then can get file from multiple places or can get chunks of file from multiple places
      - Overall available bandwidth is now higher because there are more nodes
      - Every node in P2P nw acts both as a client and as server
- ∇ Advantage: don't rely on any one server – any peer can become a server, change their IP address and receive requests from other online nodes
  - ♥ Mechanism allows for notification for when a peer becomes online
- ∇ Peers intermittently connected and change IP addresses
  - ◊ Complex management needed in order to allow nodes connected to sys to be enabled to announce their presence and for nodes to be able to discover what is available, what services they provide and be able to request such services
- ∇ P2P nw example: Zoom and Skype (real-time communication systems)
  - ◊ Initially connect to server to become part of nw

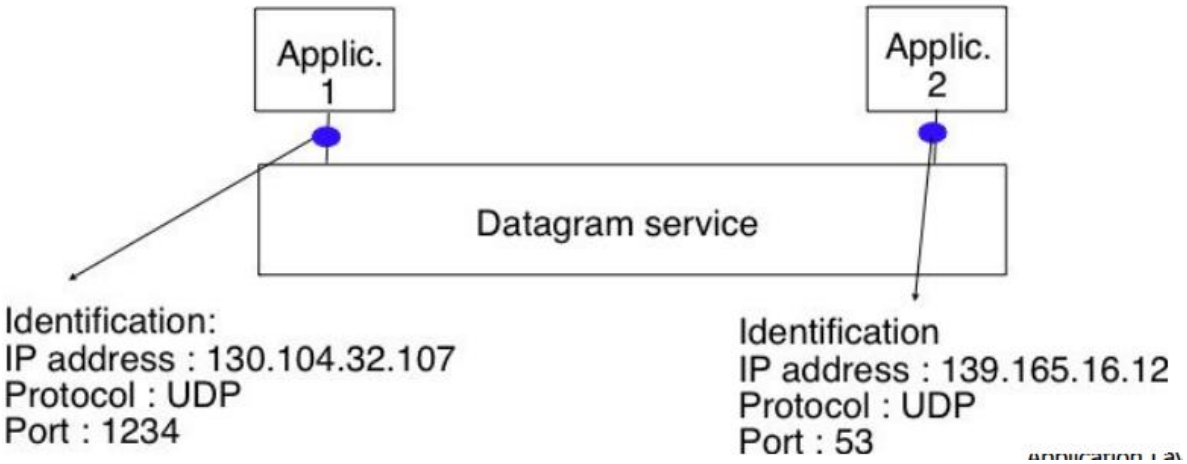
- But way msgs get exchanged between clients that have connected to service is that it's a P2P nw – msgs flow from one machine to another
- Faster because msgs don't have to flow from machine to server and back to machine

Service



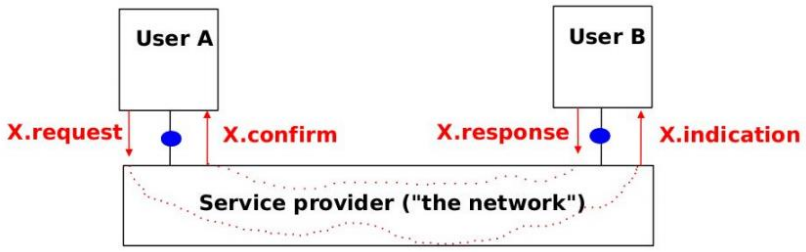
- Capabilities provided by sys to its users
  - Sys is the layer in question i.e. the layer providing the service to the upper layer
- Users are two communicating apps in the app layer (users of the service)
- Service provider e.g. can be nw layer (provides service to transport layer)
- App layer:
  - Two apps require services of transport layer – on each side, they'll use a **service access point** to connect to transport layer service
  - Then apps are able to send and receive msgs from the transport layer
- Port: Logical construct assigned to some nw process so that they can be identified in that sys
- Socket: Instantiation of that port associated with an IP address that identifies that host
- Ports and sockets used as access points (points of attachment) between the transport and app layer service
  - App will have to deposit msg on specific port/socket so that they're taken up by transport service
- Primitives: types of signals/commands exchanged between upper and lower layers for service provision
  - Example – if user (upper layer) wants service i.e. sends a request
    - Primitive is request – request contains msg and destination
    - Transport service does the rest
  - Allow for the setup/connection between user and service provider

TRANSPORT SERVICE



- ∇ Allows networked apps to exchange msgs
- ∇ Msgs transported by lower layer i.e. transport service
- ∇ App identification:
  - ◊ On each side, apps needs to be identified (service access points)
  - ◊ Lower layer needs to be able identify each of the apps at the top that're requesting some service, where the msgs are destined for and coming from
- ∇ Apps identified by:
  - ◊ Host IP address – device that app is running from
  - ◊ Port #: numeric ID used to identify app in local sys
    - ♥ Each integer represents a specific process that's running
    - ♥ Associated with that is port #, through which can exchange msgs from that process
    - ♥ Each nw app will start at particular port # - know that msg coming from a specific port # is coming from this particular app
  - ◊ Protocol:
    - ♥ Identifies the transport layer protocol to be used for this
    - ♥ E.g. TCP, UDP
    - ♥ When each app starts, must identify which protocol it's going to use for transmitting its messages
- ∇ Necessary so that when msgs are going down from app layer to transport layer, that app is clearly and uniquely identified – this app is on this *IP address*, this *port #* and using this *protocol*
  - ◊ Same happens on the other side
- ∇ IP address needed so that msg gets routed to the correct device
  - ◊ When it arrives, look at port # and protocol to know which specific process receives that msg

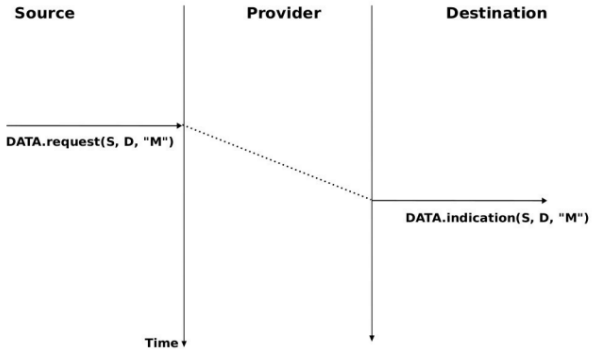
SERVICE PRIMITIVES



- ∇ User and service provider interact using types of primitives through a service access point
- ∇ Example:
  - ◊ Users A and B want to communicate
  - ◊ User A sends request to its transport service – requests service x
  - ◊ Connection request has some identifying info about the app requesting it and destination address of app
  - ◊ Transport service takes request, brings to other side and makes indication on user B side that they have a request for connection
  - ◊ If user B accepts, then response is sent and msg goes back to transport service and confirmation made to initial request (User A)
- ∇ Msgs exchanged through access point – allows this interaction

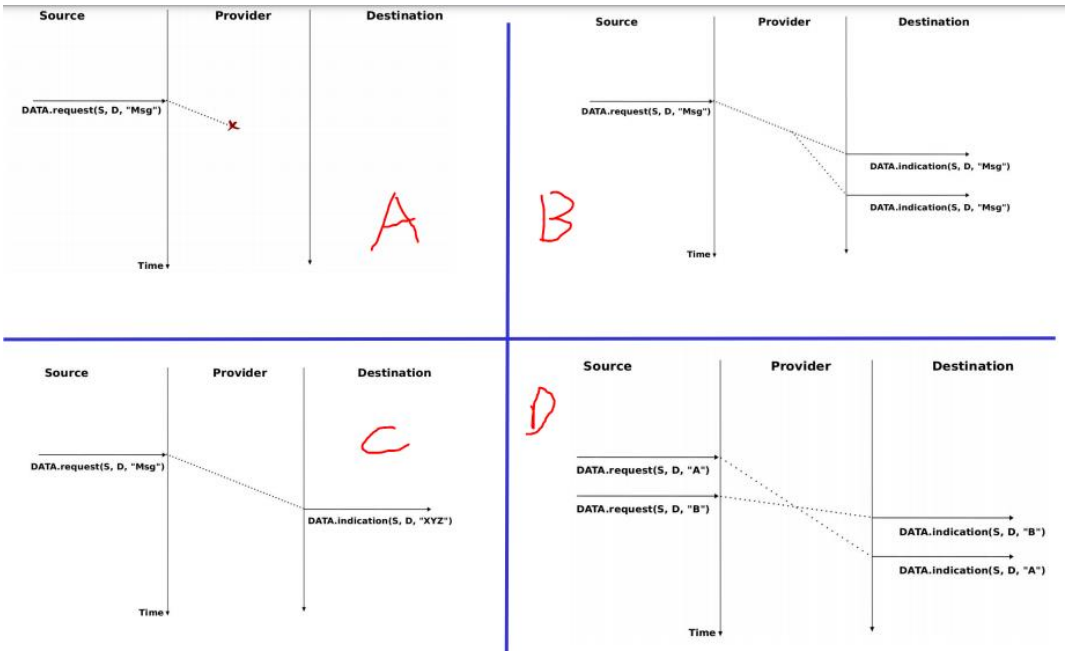
TYPES OF TRANSPORT SERVICES

Connectionless service



- ∇ Enables src/client that wants to send msg to transmit msg without requesting connection with dest (without setting up a circuit)
- ∇ Primitives:
  - ◊ Data.request(src, dest, ServiceDataUnit [SDU])
    - ♥ Dest includes port #s and protocol so that it can be properly identified before it reaches dest
    - ♥ Request for transmitting data
    - ♥ SDU = msg (“M”)
    - ♥ Sent through provider and comes through as data indication
  - ◊ Data.indication(src, dest, SDU)
    - ♥ New data is received by transport layer that must be passed to app layer
- ∇ App indicates to transport layer that it’s trying to send a msg and specify dest of msg
- ∇ Problems:
  - ◊ If msg sent by src to dest without establishing connection = problems

- ♥ Msgs **lost** (A), **duplicated** (B), **not received** (C) or **received in incorrect order** (D)



  - ◊ Msg sent and in nw service, it could be lost – with no connection established, no way of knowing that a msg has been lost
    - ♥ Without mechanism for determining whether msg received/not and only relying on connectionless service, then many msgs could be lost
  - ◊ Send msg and something could happen in network which results in msg being duplicated – send one msg but two received
    - ♥ E.g. if send msg that cmd should be executed, then it could be executed more than what is needed (creates sys problems)
  - ◊ Msg that is sent differs to what is received – errors happen on nw as msg flows and results in msg being transformed
    - ♥ With no connection established, no way of knowing that the msg sent is same as one received
  - ◊ Order of msgs sent could be changed – sent A and then B
    - ♥ Along way some msg gets delayed, and arrive out of order
    - ♥ Depending on what msgs mean to dest, could result in sys problems

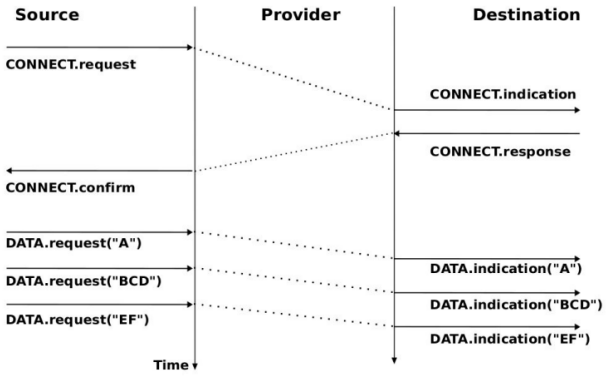
Connection- oriented service

- ∇ First set up connection/circuit between src and dest before sending, receiving msgs
- ∇ Need primitives for establishing connections:
  - ◊ Connect.request
  - ◊ Connect.indication
  - ◊ Connect.response
  - ◊ Connect.confirm
- ∇ Process:
  - ◊ Src indicates to provider that want to connect with particular dest
    - ♥ Connect.request
  - ◊ Msg sent through nw into the dest and it's going to receive connection indication that there is a client that wants to connect

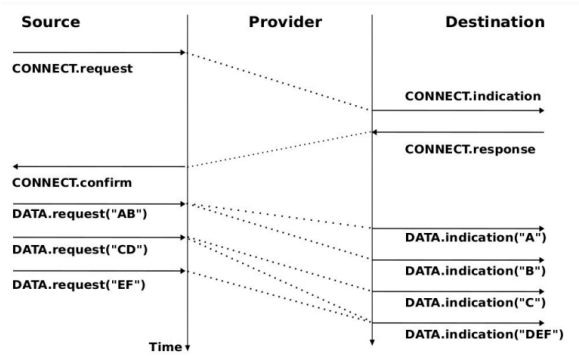
- ♥ Connect.indication
- ◊ If dest accepts connection, it sends +ve response and sends msg back to src
  - ♥ Connect.response
- ◊ Src receives confirmation of that connection
  - ♥ Connect.confirm
- ◊ Once confirmation received, src considers connection open for transmission
- ◊ Dest already opened connection when it confirmed
- ◊ Both sides ready to send msgs

∇ Msg transfer modes:

- ◊ Message-mode: each msg that is sent must be received in the same unit format
  - ♥ E.g. Send msg "A", must be delivered as "A" (same for "BCD")



- ◊ Stream-mode: unconcerned about units of the way msgs transmitted and received
  - ♥ Just want msgs to be received in same order as they were sent
  - ♥ E.g. Send msg "AB" and gets received as "A" and then "B"; send "D" and then "EF" and gets received as "DEF"
  - ♥ Simplifies how msgs transmitted – don't have to maintain boundaries, just need to ensure each unit of msg comes through (in the same order they were sent)



INTERNET TRANSPORT PROTOCOLS SERVICE

∇ Two socket types for transport services

Reliable-Connections Oriented service

∇ Reliable-Connections Oriented service – e.g. TCP Transport Communication Protocol)

- ◊ **Reliable transport** between sending and receiving process – guarantees msg is received in the way it was sent

- ♥ If confirmation of msg receival not retrieved, there are mechanism to correct that – through re-transmission or on the receiver side there is something that corrects any error that has occurred in the transmission
- ♥ Sender aware that msg is correctly received

◊ **Flow control**: sender won't overwhelm receiver
- ♥ Receiver has capacity i.t.o how many msgs it can receiver at once
- ♥ Don't want to fill up buffer and cause other msgs to not be handled
- ♥ Ensures you send just enough to ensure receiver not overwhelmed

◊ **Congestion control**: throttle sender when nw overloaded
- ♥ In the nw there's other msgs coming from other src to other dests
- ♥ Don't want sender to send many msgs such that it severely effects the nw

◊ **Doesn't provide**: timing, minimum throughput guarantee, security
- ♥ **E**.g. can't guarantee can deliver 20 msgs per sec at min
  - Quality of service not guaranteed

◊ **Connection-oriented**: setup required between client and server processes

Unreliable-Connectionless service

∇ e.g. UDP (User Datagram Protocol)

∇ Unreliable data transfer between sending and receiving process
- ◊ Sender has no guarantee that it's msgs have been received/ received correctly
- ◊ If want those guarantees – have to implement in own app
- ◊ E.g. have an app that sends msg X and requires receiver to send back Y in order to confirm receival of msg

∇ Doesn't provide: reliability, flow control, congestion control, timing, throughput guarantee, security, or connection setup
- ◊ Timing = interarrival time
  - ♥ E.g. Msgs sent one per sec but no guarantee msg will be received one every second – circumstances in the nw can change the interarrival time
    - Maybe all msgs arrive within a second or there could be bigger gaps
- ◊ Modern UDP implementation includes a checksum that is used to validate the integrity of the data (segment) that is being received.
  - ♥ If checksum validation fails at the receiver transport layer, then the data unit is discarded
  - ♥ BUT the sender is NOT aware that an error occurred, and the data has been discarded.
    - And the data is not passed on to the application layer.
    - Hence, this service is unreliable.

∇ Why use UDP?
- ◊ Lots of online services require short msgs that don't necessarily need to be confirmed and need to happen quickly
- ◊ In this case, wastes time to setup connection before transport msg – if msg is short and takes a very small amount of time to send, then setting up a connection reduces performance of service
  - ♥ Especially if no guarantee of msg receival is needed – then don't need connectionless server

∇ Example: DNS
- ◊ Machine just wants to query server quickly for IP address of destination server

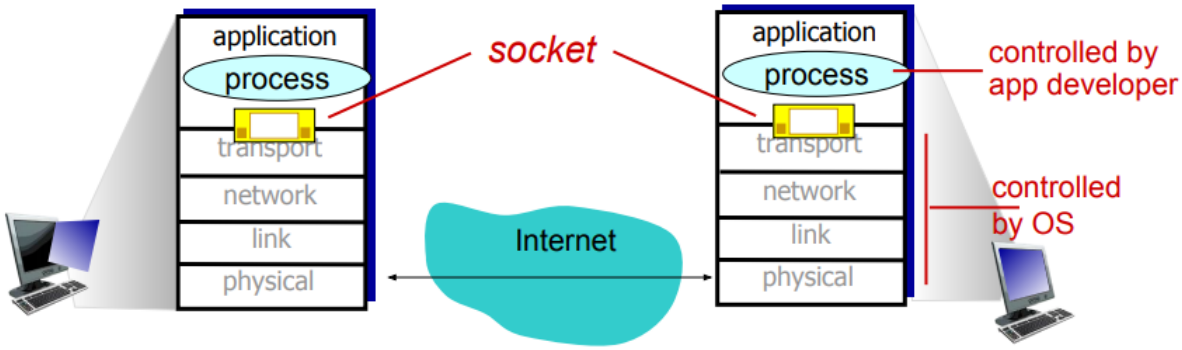
- ◊ Needs to happen quickly – short msg with short answer
- ◊ If connection require before sending, then service will be much slower
- ◊ If msg not received, client will resend msg
- ◊ Thus, UDP used for DNS

∇ Some services do better with unreliable connectionless service because can't afford to waste time setting up connection and because msgs are such that they can be easily transmitted if don't get a response in time

Process Communication

- Process: program running within a host
  - Can have prog running as several processes
    - Within same host, two processes communicate using inter-process communication (defined and governed by OS)
  - Have apps that run on different hosts but still communicate
    - Processes in different hosts communicate by exchanging msgs
- Two types of processes:
  - Host that initiates communication with another process (on remote device) – client
  - Process that is "waiting" to be contacted – server
    - Once it receives request from client, then server process becomes active and stars communicating with client process
- Client process: process that initiates communication
- Server process: process that waits to be contacted
- Apps with P2P archs have both client and server processes
  - Since peers act as both clients and servers at different instances
  - Peer acts as client when initiating communication with remote peer – in other cases, it's a server as it has to wait and receive requests from other peers

Sockets



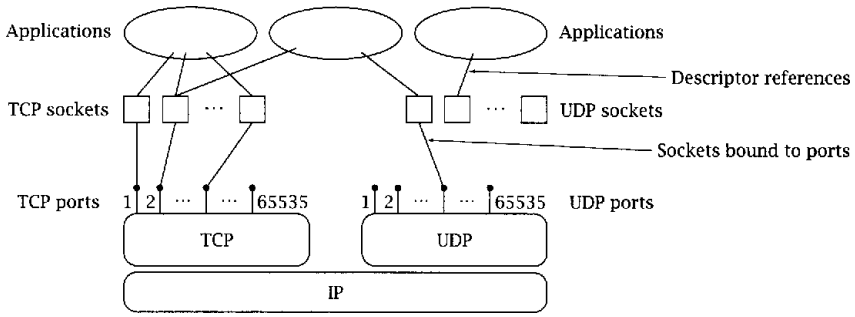
- Where nw processes send/receive msgs
- Socket is service access point
  - If nw app needs to send something, it has to put it into that socket
  - If it needs to receive something, it must open that socket and check if there is any package coming through
- Socket analogous to door
  - Sending process shoves msg out the door
  - Sending process relies on transport infrastructure (courier service) on other side of door to deliver msg to socket receiving process
    - Sending process doesn't have to know what transport infrastructure does – just needs to know that if pckg put "outside door", transport service will pick up pckg and transfer it to dest.
  - Ensures that pckg gets delivered to dest address – placed outside door

- Recipients open door (socket) and place pckg inside
- Nw apps open socket, place pckg there (to be sent)
  - On receiving side, have to open socket and pick up pckg for app
- What happens from transport to physical layer i.t.o. msg getting passed down (from transport down to physical) is controlled by OS
  - This also applies backwards i.e. from physical back to transport on receiving end
- App process controlled by app dev
  - Dev needs to know who to send to and receive msgs from socket

Addressing processes

- Process must have **identifier** to receive msgs
  - Street address in analogy – identifies house to which pckg must be delivered i.e. host to which pckg delivered to (IP address)
  - Host device has unique 32-bit IP address
- IP address on which process runs doesn't suffice for identifying the process – since many processes can be running on same host
  - Analogy: recipient house has many doors – need pckg to be delivered to correct door
    - E.g. deliver to door #2
  - These (doors) are the port #s in nw processes – integer identifier associated with particular process running on host
    - Have this because of several processes running on same host
- Identifier includes both IP address and port # associated with process on host
- Example: accessing web server from browser – browser automatically adds port #
- Example port #s:
  - HHTP server: 80
  - Mail server: 25
- Example: to send HTTP msg to gaisa.cs.umass.edu web server
  - IP address: 128.119.245.12
    Port #: 80

Socket Programming



- Socket: door between app process and E2E transport protocol
- Port #s are 16-bit integers managed by OS
  - When nw process starts, it is associated with one of these #s
  - 16-bit value could represent numbers up to 65535 = range of port #s

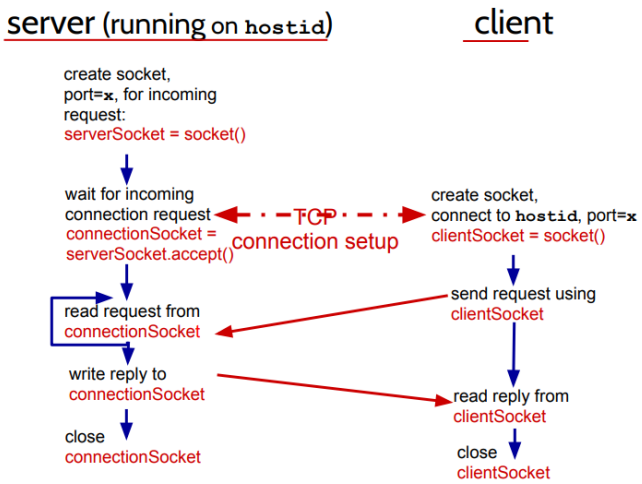
- Have app being developed, and in sockets running in transport layers (sockets associated with port #s)
  - Sockets interface with transport layer
  - Have TCP and UDP sockets – registered with port #s
- TCP is byte stream-oriented

WITH TCP

- *Client must contact server*:
  - Server process must first be running
  - Server must have created socket that welcomes client's contact
- Client contacts server by:
  - Creating TCP socket, specifying IP address of server and port # of server process
  - When client creates socket:
    - Client TCP establishes connection to server TCP
- When contacted by client, server creates new socket for server process to communicate with that particular client
  - Allows server to talk with multiple clients – i.e. server socket used for accepting connections must be available to accept other connections from other clients
  - Src port #s used to distinguish clients
    - If client contacts serverand server maintains communication with that client on that same server socket – makes it impossible for other clients to communicate with that server socket as it is bound to that client
    - Hence why new socket on new port created for each client
- TCP provides reliable, in-order byte-stream transfer ("pipe") between client and server

CLIENT/SERVER SOCKET INTERACTION

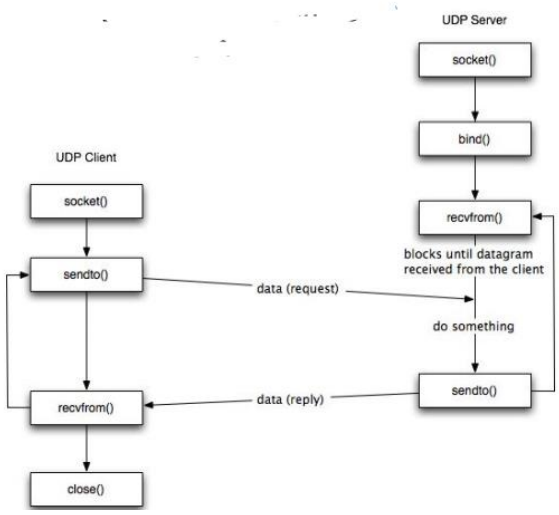
TCP



- Have server and client process
- Server creates socket, specifies server port # (on which socket is running)
  - Client also creates socket – specifies remote host/server ID/IP address and server port # f
    - While it does that, it sets up TCP connection with server
  - In Java it's instantiated as an object of socket class
- If server accepts connection request it creates new connection socket with that client.

- Once connection set up, msgs start being sent
  - Send request through client socket which server receives through connection socket
- At the end, either party can close socket

UDP



- Client doesn't establish connection with server
  - **Send** function requires dest IP address as parameter (because there's no connection)
- Server doesn't accept connection from client – but must be in running state (enabled to receive msgs)
  - Server's **receive** function waits until data arrives from some client.

UDP send example:

```
DatagramPacket request = new DatagramPacket(buffer,
buffer.length, address, port);
socket.send(request);
```

UDP receive example:

```
DatagramPacket request = new DatagramPacket(buffer,
buffer.length);
socket.receive(request);

InetAddress clientAddress = request.getAddress();
int clientPort = request.getPort();
```

- On sender's side:
  - Set up datagram packet – created using dest address and port #
    - Buffer is memory location on local machine where info is that you're trying to create packet for
- On receive side:
  - Packet creation – created without needing an address or port
    - Use receive function to receive data from socket
  - If server needs to know client IP address (msg sender) in case it needs to reply, then client address can be obtained with getAddress() (from java socket class)

- ◊ Define socket where msg being received – receive msg from socket and then get the address and client port with getPort() (where msg came from)
  - ♥ Use this info if need to send reply to client

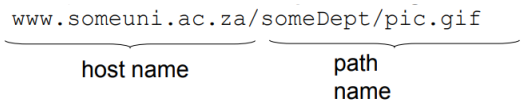
QUESTIONS

- Q: Sockets for UDP vs TCP, server-side. How many sockets will there be when 15 clients connect over UDP? And over TCP?
- A:
- Q: Client-server app over TCP. Why must the server program be executed before the client program?
- A:
- Q: Designing a network application, you want to ensure the transactions with the remote host occurs as fast as possible. What underlying protocol would you use?
  - A) TCP
  - B) UDP

∇ (1) There are as many sockets as there are clients connected BUT the # of ports doesn't necessarily increase
- ◊ Server will use the same port to setup socket
- ◊ Socket ID'd by server-side port, IP and client-side port and IP
- ◊ BUT if client makes several TCP connections, will need to use different ports – can't set up many unique sockets on same port

Web and Http

- ∇ Webpage consists of objects
  - ◊ Object can be HTML files, media files, etc
- ∇ Webpage consists of base HTML-file which includes several referenced objects
- ∇ Each object addressable by URL (universal resource locator)
  - ◊ When asking for resource from web server – specify host name of web server (asking for content from this server)
  - ◊ Specify a path name



HTTP OVERVIEW

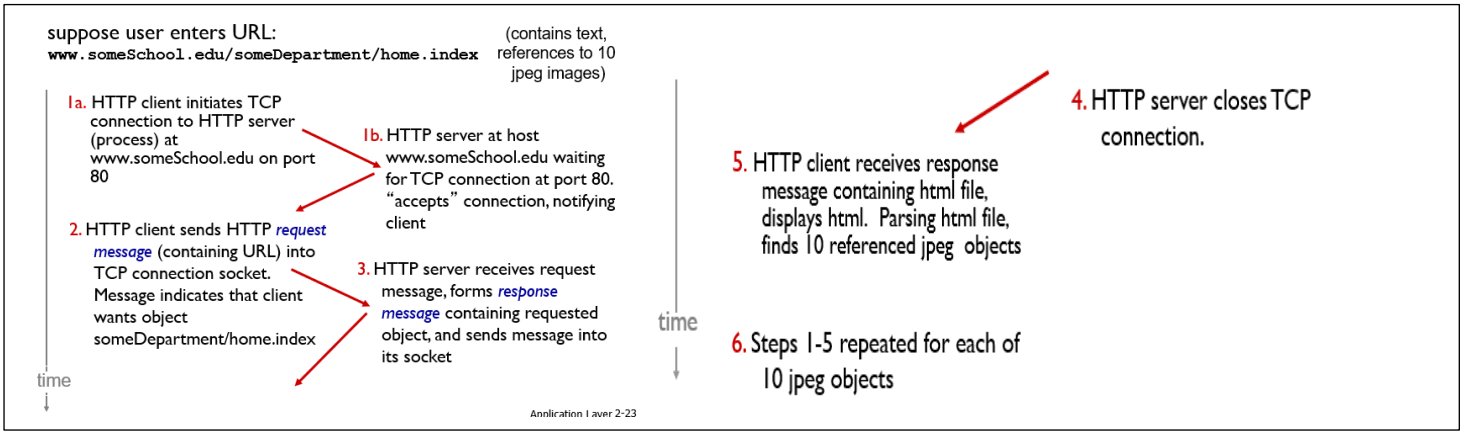
- ∇ Hypertext transfer protocol
- ∇ Web's app layer protocol (operates at the app layer)
- ∇ Client/server model:
  - ◊ Client – browser that requests, receives (using HTTP protocol) and "displays" web objects
  - ◊ Server – web server sends (using http) objects in response to requests
- ∇ Uses TCP at transport layer:
  - ◊ Client initiates connection (creates socket) to server, port 80 (if using http)
    - ♥ Could be other port if using other protocol, such as https
  - ◊ Server accepts TCP connection from client
    - ♥ Once connection established, http msgs begin to flow
  - ◊ HTTP msgs (app-layer protocol msgs) exchanged between browser (http client) and web server (http) server
    - ♥ Client requesting specific url object and web server responding with those objects
  - ◊ Once that's done, TCP connection closed
    - ♥ TCP *can* be kept open for subsequent requests
- ∇ HTTP is "stateless": server maintains no info about past client requests

- ◊ When client make request – only deals with that current request (doesn't need to know about previous requests i.e. maintain some state about its interaction with other clients)
- ◊ If want to maintain state, complex sys needed

Http connections

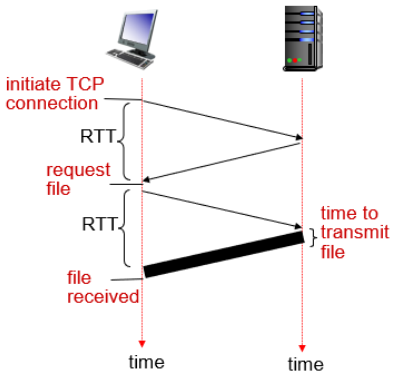
NON-PERSISTENT HTTP

- ∇ TCP connection established
- ∇ At most one object sent over TCP connection
  - ◊ Connection then closed
- ∇ Process:
  - ◊ Client makes connection
  - ◊ Makes request for an object
  - ◊ Server sends object
  - ◊ TCP connection terminated
- ∇ Downloading multiple objects requires multiple connections
  - ◊ E.g. have several imgs and media – need to set up connection for each of these



- ∇ Example:
  - ◊ (1) Initiate TCP connection to server – get confirmation that connection accepted
  - ◊ (2) Client makes request – sends request msg
  - ◊ (3) Server receives request for that object – makes response msg which contains that msg and sends it back on socket that's been established
  - ◊ (4) Http server closes TCP connection
  - ◊ (5) Once client receives it – displays the needed file
  - ◊ (6) Repeat 1-5 for all objects

Response Time



- ∇ Performance is concerned with how much time it takes for a pg to load
- ∇ Round-Trip-Time (RTT): time for small packet to travel from client to server and back
  - ◊ Time for msg to be sent and response to come back
  - ◊ Client sends request and request response comes back RTT
- ∇ HTTP response time:
  - ◊ One RTT to initiate TCP connection
  - ◊ One RTT for HTTP request and first few bytes of HTTP response to return
    - ♥ i.e. time it takes to request file and amount of time it takes to transfer file
  - ◊ File transmission time
  - ◊ Non-persistent HTTP response time = 2RTT + file transmission time
    - ♥ For **each** object

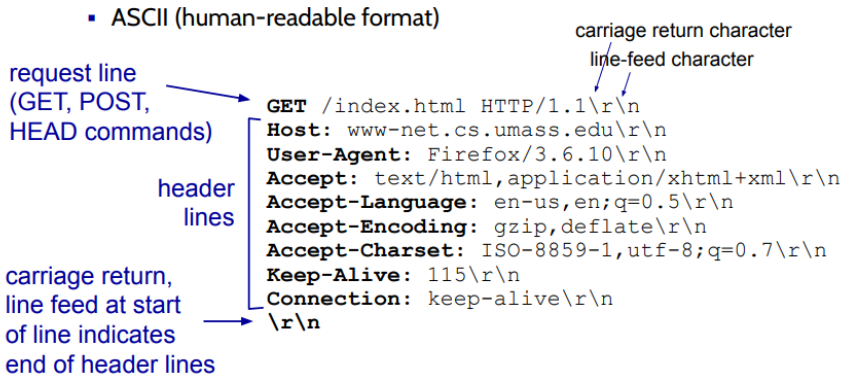
Issues

- ∇ Requires 2RTTs per object
  - ◊ Need one RTT to setup connection, another to request object
- ∇ OS overhead for *each* TCP connection
- ∇ Browsers often open parallel TCP connections to fetch referenced objects

PERSISTENT HTTP

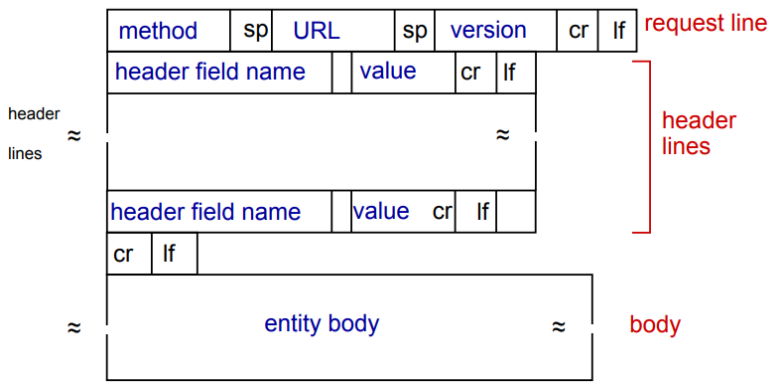
- ∇ Multiple objects can be sent over single http connection between server, client
- ∇ Solution to non-persistent http issues!
- ∇ Server leaves connection open after sending response
- ∇ Subsequent http msgs between same client and server sent over (that same) open connection
- ∇ Client sends requests as soon as it encounters a referenced object
- ∇ As little as one RTT for all referenced objects
- ∇ Have one RTT to establish connection and all subsequent requests take one RTT

Http request message



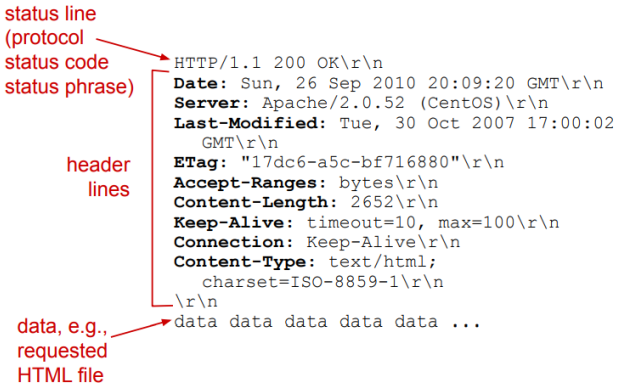
- ∇ Two types http msgs: request, response
  - ◊ Request goes from client to server
  - ◊ Response goes from server to client
- ∇ HTTP request msg:
  - ◊ ASCII (human-readable format)
  - ◊ Msg has keywords and structure so that client and server both know how to read and interpret the msgs that are being exchanged
- ∇ Msg structure:
  - ◊ Starts with initial cmd line – specific cmd is sent e.g. GET, POST etc
    - ♥ In example: get cmd requests specific url "/index.html xxxx"; specifies protocol http version 1.1; has newline char to specify that msg continues on next line
    - ♥ When server reads msg and encounters "\n" – knows that msg has another part on next line
  - ◊ Request (initial) line followed by number of header lines which specify parameters associated with the request being sent
    - ♥ E.g. First line is Host – where you're requesting file from (web server)
      - User agent – what client is requesting the file
      - Connection – i.e. persistent or non-persistent
        - "keep-alive" – client requests that same connection be used for subsequent communication (persistent connection)
  - ◊ After header lines, have another carriage return at start of line "\r\n"

GENERAL FORMAT



- Have request line – method, url (link to what we're looking for – html file, some object etc), version of http protocol being used
- Then have headers and their values
- Lastly there is the body of the msg
  - If request – body may be empty
  - If response – body may contain file being sent to client (img file, html file etc)

Http Response Message



- First line is status line
  - E.g. HTTP responding with "200 OK" (request was received and server able to satisfy request)
- Header lines with keywords follow
  - E.g. Date, server, last modified (when file being sent was last modified), Keep-Alive (server maintains connection)
    - Content-type indicates type of file being sent (e.g. jpeg, text)
- Finally have data requested

HTTP RESPONSE STATUS CODES

- Status code appears in 1st line in server-to-client response msg
- Sample 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 – requested msg not understood by server
    - E.g. url not properly specified
  - 404 Not Found – requested document not found on this server
  - 505 HTTP Version Not Supported

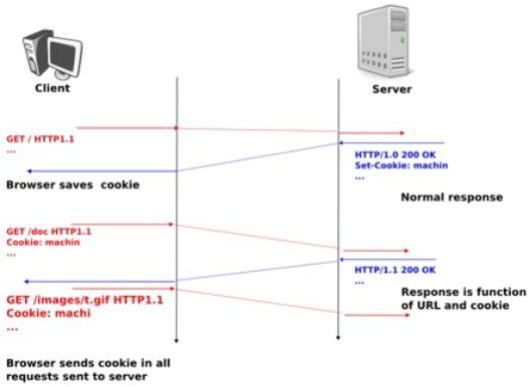
User-server state: cookies

- HTTP stateless protocol by design – so when protocol receives request, it doesn't necessarily know the previous requests sent by that client (just deals with requests as it comes)
  - Not conducive to web browsing – useful to know if the client already visited the site and interacted with it, and know user preferences
- Workaround – server-side cookies

- ◊ Way of identifying and knowing you as a client when client returns to site

∇ 4 Components:

- ◊ (1) Cookie header line of HTTP response msg
  - ♥ When visit site for first time – http response comes back to client
- ◊ (2) Cookie header line in next HTTP request message
  - ♥ Includes in its header – indicates that it has cookie available and asks client if client would like to use
    - Client can deny => no state established between client or server, no way of tracking client
- ◊ (3) Cookie file kept on user's host, managed by user's browser
  - ♥ Or accept => next time client sends request to server it includes cookie in its header to ID itself to server
- ◊ (4) Back-end dtb at site
  - ♥ On server side dtb maintains all cookies – used to ID client



∇ Client makes TCP connection and GET request

∇ Server receives request and sets cookie in HTTP response msg

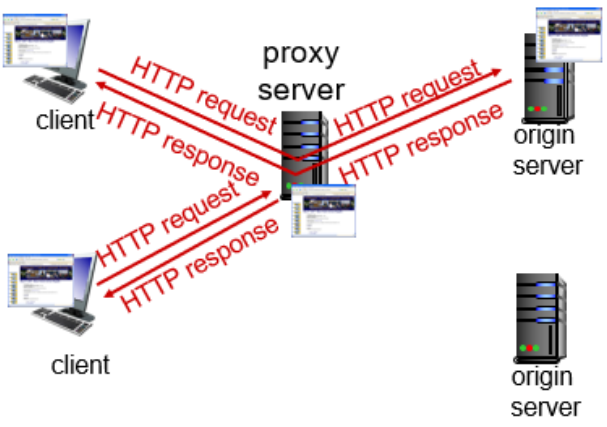
∇ Browser receives cookie and can accept it -saves it locally on the devices

∇ Next time browser makes request to same site – includes cookie in request

∇ Request goes to server, sees cookie in it and checks dtb

∇ Give web response customised based on what you did last time

Web Cache



- ∇ Goal: satisfy client request (for web content) without involving original server (where content came from)
- ∇ Have web cache in nw – most nws with lots of users have web cache
- ∇ User sets browser – web accesses via cache
- ∇ Browser sends all HTTP requests to cache server (managed by nw) and server checks dtb
    - ◊ Object in cache – cache returns object
    - ◊ Else cache requests object from origin server, then returns object to client and cache keeps copy of response
- ∇ When another client makes request for the same page – server responds with page it has in cache
- ∇ Web caches used to improve performance – not all users have to wait for a page (it's available already)
    - ◊ Also used to conserve bandwidth – if multiple clients are making multiple requests for the same content, better to have it cached

Domain Name System

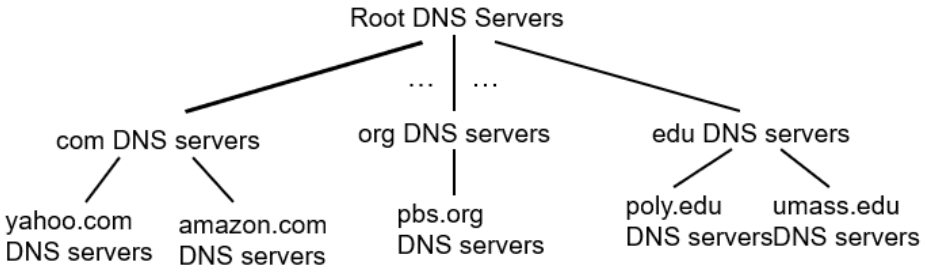
- People use names to identify, whereas computers use numbers
- Internet hosts, routers:
  - 32-Bit IP (Internet Protocol) address used for addressing datagrams
  - Host name e.g. www.uct.ac.za used by humans
- How to map between IP address and host name and vice versa?
  - This is where DNS comes in
- DNS is distributed dtb implemented in hierarchy of many name-servers
  - These servers respond to queries when computers want to know an IP address -i.e. when app gives domain name, computers need to convert this into IP address (use DNS for that)
  - App-layer protocol:
    - Hosts, name servers communicate to resolve names (address/name translation)
      - Note: core internet function implemented as app-layer protocol
        - Means don't need to design routing protocol that supports DNS in core
      - Complexity at network's edge

Dns Services

- DNS service
  - Translation between hostname and IP address
  - Host aliasing
    - Have different names that map to same server/domain
    - Canonical, alias names
    - When user types in any of the domain names, it maps to same place
    - E.g. amazon.com, amazon.co.uk
  - Mail server aliasing – same as with host aliasing
    - E.g. company gets acquired by another but still wants to use original email – use mail server aliasing
      - someone@bethesda.com will still work, even though Bethesda owned by Microsoft
  - Load distribution
    - Replicated Web servers – many IP addresses correspond to one name
    - E.g. have popular website Facebook – when users log in from different locations, don't all access the same server
      - Do this when have x number of servers for website located all over – depending on where client is querying from (like SA), give them an IP address of the nearest server (server located in SA for example)
- Why not centralise DNS?
  - Single point of failure
  - Traffic volume (large)
    - Volume would choke connection

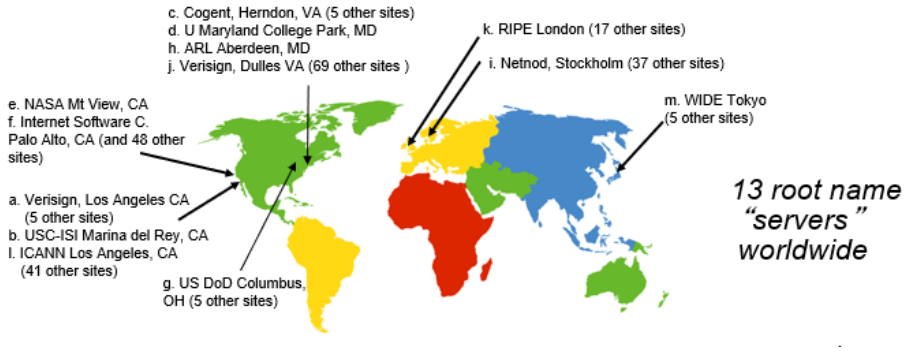
- ◊ Distant centralised dtb
- ◊ Maintenance – easier to e.g. bring down one server and point users to different server
- ◊ Doesn't scale

Dns Structure



- ∇ DNS is distributed, hierarchical dtb
  - ◊ Have zones(layers) that respond to different types of queries
    - ♥ E.g. google.com – all sites that end with .com belong to .com zone
      - Google could also have several sub-domains
- ∇ At top have root servers – primary servers that any client/server will contact if they don't know where to find results for domain query
  - ◊ Each DNS server or client, when setup has a list of IP addresses that it can query when it needs to begin DNS resolution process
  - ◊ Root servers respond to client about another set of servers that can give them final answers
  - ◊ E.g. your PC doesn't know where to get google.com results - so it queries root servers
    - ♥ Root server responds – it cannot answer query fully but can provide list of servers that can give response relating to an .com website
      - Returns list of IP addresses belonging to .com zone
    - ♥ Go to those servers – ask about google.com
      - .com server provides IP address for google.com or provides another server which responds to queries relating to google.com
        - ○ Go to that server – could then say need mail.google.com, and server provides final IP address
  - ◊ Those servers may have own servers which could respond to more specific domains

ROOT NAME SERVERS



- ∇ Main function: contacted by local name server that can't resolve name
  - ◊ E.g. will send client looking for xxxx.com to .com servers
- ∇ Root name server:
  - ◊ Contacts authoritative name server if mapping not known
  - ◊ Gets mapping
  - ◊ Returns mapping to local name server
- ∇ 13 worldwide – but there's much more instances of each because each IP address is now mapped to several other servers with same IP address using anycast system
- ∇ Anycast:
  - ◊ Have IP address associated with several servers
  - ◊ When query/contact IP address, anycast system sends you to server located closest to you
    - ♥ Thus, able to scale – can have many instances of a server
    - ♥ Each IP address can be positioned in different locations – use anycast mapping system to allow users to reach any server within anycast group

TOP LEVEL DOMAINS AND AUTHORITATIVE SERVERS

- ∇ Top-Level domain (TLD) servers:
  - ◊ Servers that respond to the second part of the domain name e.g. the .com
  - ◊ Responsible for com, org, net, edu, aero, jobs, museaums and all top-lvl country domains e.g. uk, za, gh
  - ◊ Network solutions maintains servers for .com TLD
  - ◊ Educause for .edu TLD
  - ◊ E.g. asked for google.com – gives you authoritative DNS server that belongs to google
- ∇ Authoritative DNS Servers:
  - ◊ Org's own DNS server(s) – providing authoritative hostname to IP mappings for org's named hosts
  - ◊ Can be maintained by org or service provider
  - ◊ E.g. can than ask for IP address for mail.google.com and this authoritative DNS server will respond
  - ◊ Using dig -t ANY website.extension
    - ♥ Have name server associated with domain and can get an actual IP address after querying

LOCAL DNS NAME SERVER

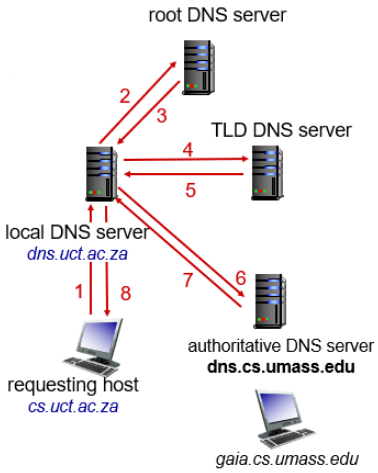
- ∇ Doesn't strictly belong to hierarchy
- ∇ Each ISP (residential, university etc) has one
  - ◊ Also called "default name server"
  - ◊ DNS server that sits within ISP and does the DNS querying/resolution on behalf of clients.
- ∇ Queries can go all the way up to the root server – but wouldn't work in practice
  - ◊ Thus, in many ISPs, have local DNS server
- ∇ When host makes DNS query, query is sent to its local DNS server
  - ◊ Has local cache of recent name-to-address translation pairs (may be outdated)
    - ♥ Can then just provide response
  - ◊ If local cache answer time of validity has expired – acts as proxy, forwards query into hierarchy (root to top lvl, then authoritative name server and send final results)
- ∇ Local DNS works on behalf of its clients in the local nw
- ∇ ASIDE: In many cases, when connected to nw, local setup gives you DNS server for querying BUT could choose to use different name server to handle requests
  - ◊ E.g. 1.1.1.1 – open/public DNS resolvers
  - ◊ Use when you don't trust local organisation server
  - ◊ When client makes request, it's not to your local DNS server – but another external DNS server which will help you resolve

Dns Name Resolution

- ∇ Two main ways in which DNS resolution happens

ITERATED QUERY

- ∇ Contacted server replies with name of server to contact
- ∇ "I don't know this name, but ask this server"

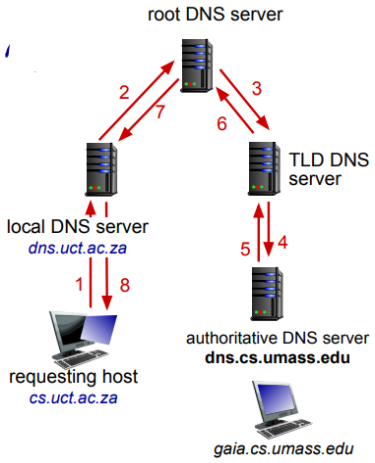


- ∇ Process:
  - ◊ Client looking to reach some site – query based on local DNS config is sent to local DNS server
  - ◊ DNS server makes request query to root server
  - ◊ Root server gives IP address of top-lvl domain server

- Local DNS then sends query to that top lvl server
- Top lvl server provides name server for domain
- Send final query to name server belonging to domain – provides final answer

- Each of upper lvl domain server is responding with IP address of next lvl in an iterated manner

RECURSIVE QUERY



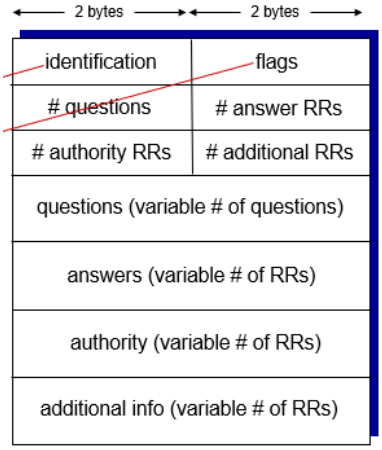
- Puts burden of name resolution on contacted name server
- Upper lvl server takes query upwards until final answer is found
  - Then goes back to client that made request
- Process:
  - Query goes to local DNS server
  - Local queries root – root forwards to TLD server
  - TLD server forwards query to authoritative DNS server
  - Final answer gets passed back to requesting host
- Public [recursive] resolvers – act like a local DNS would to a client
  - As far as client is concerned – everything is recursive (for both iterated and recursive queries)
  - Provides service to anyone – don't need to be connected to the nw
- Local DNS server provides recursive service only to its clients
  - Will not do this for those outside nw

Dns Records

- DNS: distributed db storing resource records (RR)
  - RR format: (name, value, type, ttl)
- type = A
  - **name** = hostname
  - **value** = IP address
  - Gives final IP of host that client is looking for
- type = NS
  - **name** = domain
    - e.g. x.com
  - **value** = hostname of authoritative name server for this domain

- type = CNAME
  - **name** = alias for some canonical (real) name
  - **value** = canonical name
  - e.g. www.ibm.com is really servereast.backup2.ibm.com
  - Gives authoritative server for domain that client requested
- type = MX
  - **value** is name of mailserver associated with **name**

Dns Protcol, Messages



- Two types of msgs:
  - Query – client to server
  - Reply – server to client
- Msg header
  - ID: 16-bit # for query, reply to query uses same #
  - Flags:
    - Query or reply (QR) – value for flag is 0 for query, 1 for reply
    - Type of query (Opcode) – e.g. querying for A record or NS
      - Values to indicate what type
    - Recursion desired (RD) – client wants recursive query (no intermediate answers, just final)
    - Recursion available (RA) – server says that recursion is available
    - Reply is authoritative (AA) – whether result being received is coming from authoritative name server (i.e. the server manages the domain) or another server that has been delegated to operate on behalf of domain
    - Error (RCode) – if there is error in transmission/exchange
  - # questions – how many domains are being queried
  - # answer RR – if response, # of RRs being returned
    - E.g. answer has one A record, two MX records etc
  - # authority RRs – which records client should pay more attention to since they're coming from authoritative server
  - # additional RRs - # of additional RRS which are non-authoritative
- Body:
  - Questions – name, type fields for a query

- ♥ Variable # - matches with # questions indicated in msg header
- ◊ Answers (if response msg) – RRs in response to query
  - ♥ Variable # - matches with # questions indicated in msg header
- ◊ Authority – records for authoritative servers
- ◊ Additional info – additional "helpful" info that may be used

INSERTING RECORDS INTO DNS

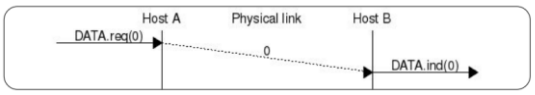
- ∇ Register name at DNS registrar (e.g. Afrihost)
  - ◊ Name "sitename" with sitename.com
  - ◊ Provide names, IP addresses of authoritative name server (primary and secondary)
  - ◊ Registrar inserts RRs into TLD server
    - ♥ E.g. insert two RRs into .com server - (sitename.com, dns1.sitename.com, NS); (dns1.sitename.com; (dns1.sitename.com, 111.111.111.1, A)
    - ♥ Bigger companies have own DNS server in infrastructure – will provide name and IP for that server
      - Or can have ISP host DNS
  - ◊ Create authoritative servers
    - ♥ Type A for www.sitename.com; type MX record for sitename.com
    - ♥ All queries relating to authoritative servers will be sent to name server for resolution.

Network Principles

- Physical Media – connecting two hosts
  - Electrical cables – twisted pair copper cables
    - Use electric signals to transmit data
    - E.g. ethernet
  - Optical fibre – multimode and monomode fibre cables
    - Uses light as a form of signal for sending data
    - Effective for long distances – light signals don't experience interference
    - Monomode – allows single light mode/signal to pass through at a time
    - Multimode – has larger core, which allows it to support more light modes to be transported at a time
      - Allows for higher data rates
      - Caveat: requires transmission be within 400m
  - Wireless:
    - Radio frequency signal – access points + clients
    - Laser and microwave – point-to-point links
      - Longer distance communication
      - Uses higher radio frequencies

Physical Layer

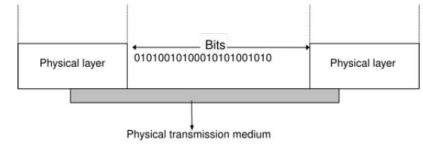
- Have data represented as bits as it moves through the layers
  - At physical layer, code bits as signal that must be moved on physical media

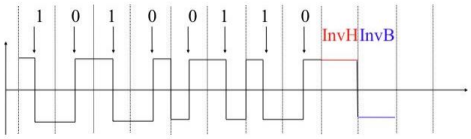


- Bits converted to signal over channel
  - E.g. for fibre optic cable, 0 or 1 can be represented as presence/absence of light pass in the cable
- How many bits can be transmitted through physical media in a second?
  - Bandwidth issue

Bit rate	Bits per second
1 Kbps	10^3
1 Mbps	10^6
1 Gbps	10^9
1 Tbps	10^{12}

- Bit conversion process:



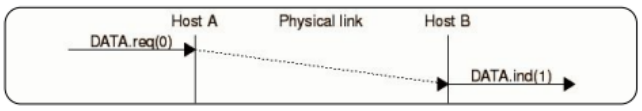


- ◊ Manchester encoding
- ◊ Have electrical cable connecting two devices – create time slots
  - ♥ Each time slot can be used to represent a zero or one
  - ♥ Time divided into some fixed length period and each period divided into two halves
- ◊ First half of time slot has high voltage and second half has low voltage
  - ♥ E.g. first half has voltage = 5 and second half = -5
    - 1 here represented by high to low and conversely 0 represented by low to high
- ◊ Need symbol to indicate beginning and end of piece of info
  - ♥ For Manchester have InvH and InvB
    - InvH – whole time slot has high voltage
    - InvB – both halves have negative voltage
  - ♥ Symbols do NOT happen within actual data

Unreliable service

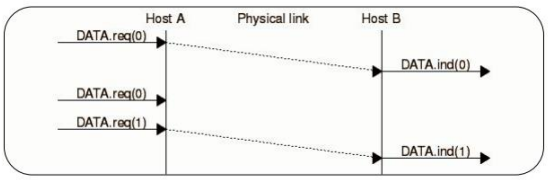
- ∇ Transmission media are themselves unreliable – no guarantee that signal will arrive as has been sent
- ∇ Types of errors:
  - ◊ Signal is *changed* – signal that is sent arrives differently
    - ♥ E.g. send signal as 0 – voltage gets spiked and it becomes a 1

Flipped bits --->>



  - ◊ Bits get lost – some of the bits being transmitted get lost in physical media
    - ♥ E.g. send 001 – but receive 01
      - Voltage not correctly read on receiving side or the timing has shifted for time slots (send 3 time slots but only able to read 2)
        - Time slots shift because of error on physical device ito timing

bits lost --->>



- Bits created –
  - E.g. 01 sent but due to timing issue on receiver side, possible to end up with more bits than were sent 001

- Physical media is NOT itself a reliable service

Datalink layer

- Physical layer interfaces with link layer
- Link layer is where try to account for all bits being sent, as well as indicate beginning and end of data unit
- Frame:
  - Sequence of bits that has a particular syntax or structure
    - E.g. sending byte (8 bits) – representation of some info
  - Basic unit of info exchanged between two directly connected hosts
    - One or several characters – but must be exchanged as single unit
- Framing:
  - How does a sender encode frames so that receiver can efficiently extract them from the stream of bits that it receives from the physical layer?
    - i.e. how to indicate to receiver the beginning and end of that frame?
    - Idling after transmission of each frame – bad solution
      - Losing bandwidth – since being idle = not doing anything = loss of transmission time
      - Need accurate synch of timing between sender and receiver so each knows exactly when the beginning and end of each frame is
      - Better solution – bit and character stuffing

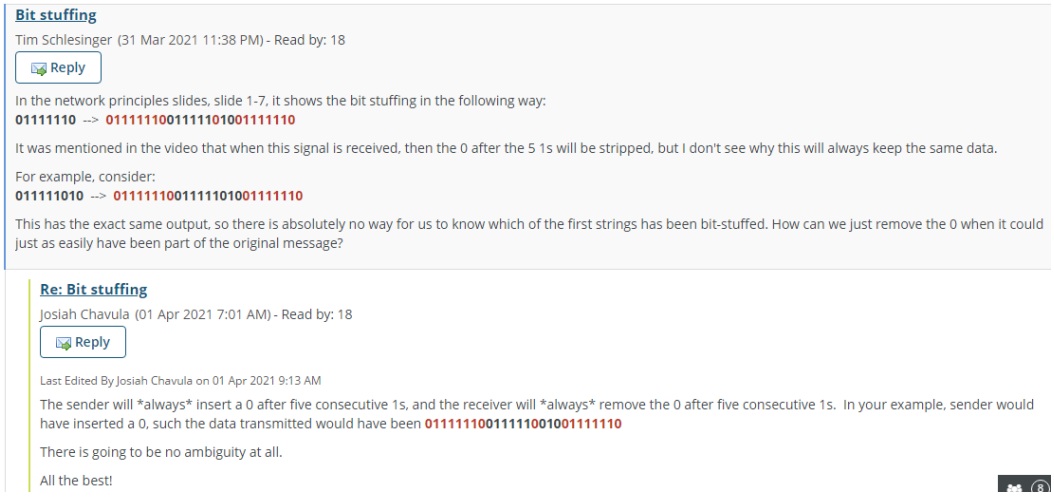
BIT STUFFING

- Have particular sequence of bits that will be used to indicate beginning and end of frame
- E.g. Bit stuffing reserves the 01111110 bit string as the frame boundary marker

Original frame	Transmitted frame
000100100100100100100011	011111100001001001001001000011101111110
011011111111111111111110010	011111100110111111011111011111011001001111110
01111110	011111100111110100111111 0

- Caveat: can't have 01111110 as part of the data/frame itself
  - For 011011111111111111111110010 it can become 011111100110111110111110111110110010001111110 – insert 0 after sequence of 5/more 1s to prevent appearance of bit string
    - *Always* insert 0 after 5 1s
  - Start transmission with bit string, then after each set 5 1s, insert 0, finally insert closing bit string
  - At receiver side – knows the beginning of frame, then anywhere else where it sees 5 1s, it will remove zero that comes after the 5 1s
  - This mechanism ensures there is no scenario where data contains bit string
- If have data that is in the form of the bit string – 01111110

- ◊ Becomes 01111110011111010011111110



Bit stuffing

Tim Schlesinger (31 Mar 2021 11:38 PM) - Read by: 18

Reply

In the network principles slides, slide 1-7, it shows the bit stuffing in the following way:
01111110 --> 011111100111110100111110

It was mentioned in the video that when this signal is received, then the 0 after the 5 1s will be stripped, but I don't see why this will always keep the same data.

For example, consider:
011111010 --> 011111100111110100111110

This has the exact same output, so there is absolutely no way for us to know which of the first strings has been bit-stuffed. How can we just remove the 0 when it could just as easily have been part of the original message?

Re: Bit stuffing

Josiah Chavula (01 Apr 2021 7:01 AM) - Read by: 18

Reply

Last Edited By Josiah Chavula on 01 Apr 2021 9:13 AM

The sender will *always* insert a 0 after five consecutive 1s, and the receiver will *always* remove the 0 after five consecutive 1s. In your example, sender would have inserted a 0, such the data transmitted would have been **011111100111111001001111110**

There is going to be no ambiguity at all.

All the best!

CHARACTER STUFFING

Original frame	Transmitted frame
1 2 3 4	DLE STX **1 2 3 4** DLE ETX
1 2 3 DLE STX 4	DLE STX **1 2 3 DLE** DLE **STX** 4 DLE ETX
DLE STX DLE ETX	DLE STX **DLE** DLE **STX DLE** DLE ETX DLE ETX

- ∇ Characters are used as markers to indicate frame boundaries
- ∇ Uses DLE, STX and ETX characters of the ASCII character set
  - ◊ DLE STX marks beginning of frame
  - ◊ DLE ETX marks end of frame
  - ◊ Sender adds a DLE char before each transmitted DLE char
- ∇ E.g. (1) Send 1234 – becomes DLE STX 1234 DLE ETX
  - ◊ (2) 123 DLE STX 4 – DLE STX 123 DLE DLE STX 4 DLE ETX
  - ◊ (3) DLE STX DLE ETX – becomes DLE STX DLE DLE STX DLE DLE ETX DLE ETX
    - ♥ Don't have to escape ETX or STX – they only become special characters when they appear next to DLE (if DLE is not escaped)

ERROR DETECTION

- ∇ To ensure communication channel is reliable – data shared between adjacent nodes must be correct
- ∇ Datalink layer is responsible for communication between adjacent nodes – error detection is an integral part of this
  - ◊ Ensure that frames being received are error free and be able to detect when frames have been lost or unexpected frames appear in receiving stream
  - ◊ Frames can be corrupted by transmission error, lost
- ∇ Error detection requires that include redundant info together with bits being sent (frames)
  - ◊ Sender converts each string of **N** bits into a string of **N+r** bits
  - ◊ At receiver, error detection code (function) computes the r redundant bits corresponding to each string of **N** bits
- ∇ Parity bits:

- ◊ Have bits string that want to transmit – add extra bit at end either 0 or 1
- ◊ Determined by criteria of making # of 1s in bit stream even/odd – depends on method being used
  - ♥ Can have odd/even parity

3 bits string	Odd parity	Even parity
000	1	0
001	0	1
010	0	1
100	0	1
111	0	1
110	1	0
101	1	0
011	1	0

- ◊ E.g. have 000 bits string – using odd parity system, add 1 = 0001
  - ♥ Now have odd # of 1s
  - ♥ On receiver side – sees # of 1s in stream as odd, determines that data stream has no errors
  - ♥ If have 001 bit stream – will add 0 so that # of 1 bits stays odd

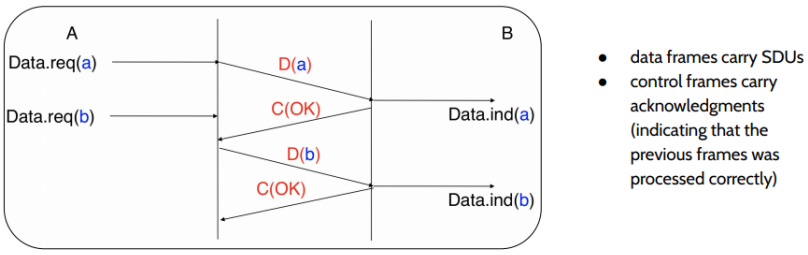
ERROR CORRECTION

Received bits	Decoded bit
000	0
001	0
010	0
100	0
111	1
110	1
101	1
011	1

- ∇ Error correction requires more redundant info in the transmitted frames
  - ◊ To be able to correct for **D** errors, sender must add **2D** redundant bits per data bit
- ∇ E.g. Want to correct for single bit error
  - ◊ For each bit, have to send two extra bits
  - ◊ To transmit 0, have to send 000
  - ◊ When received, 000 will be decoded as zero
  - ◊ Send 000 but 001 received – on sender side, two more bits add to what was sent
    - ♥ Receiver can infer that 0 was what was sent – even if one error occurred
    - ♥ The same would happen in 000 sent but 010 received
- ∇ Drawback of error correction:
  - ◊ Have to send a lot of bits for each that want to correct for – inefficient bandwidth use
  - ◊ If have more than a certain # bits flipped, can make a mis-correction – sent 0 (000) but received 011 and corrected to 1

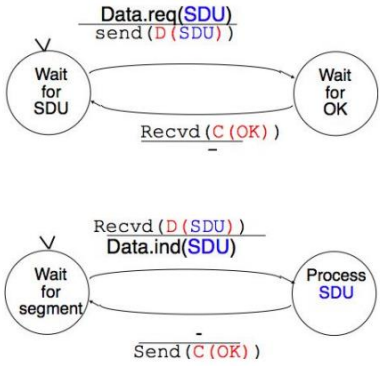
- ∇ Reliable protocols use error detection schemes, not error correction
  - ◊ Use enough bits to determine that error has occurred
- ∇ To detect errors, a data unit is usually divided into two parts
  - ◊ Header that contains fields used by the reliable protocol to ensure reliable delivery
    - ♥ E.g. a checksum + total length of frame
  - ◊ A payload that contains user data

Dealing with errors



- ∇ In a perfect system, all transmitted units are correctly received
  - ◊ Send a for A to B – B acknowledges that A has been received
  - ◊ Need mechanism to notify that data has been *correctly* received
- ∇ Along with data msgs that are sent – have control frames/messages
  - ◊ Control frame could send acknowledgement that message has been received and is correct (OK)
  - ◊ Once OK received, send b
  - ◊ Inefficient solution
- ∇ Data frames carry SDUs (Service Data Unit)
  - ◊ Control frames carry acknowledgements (indicating that the previous frames was processed correctly)
- ∇ Sender side waits for acknowledgement before it sends msg

STOP-AND-WAIT PROTOCOL

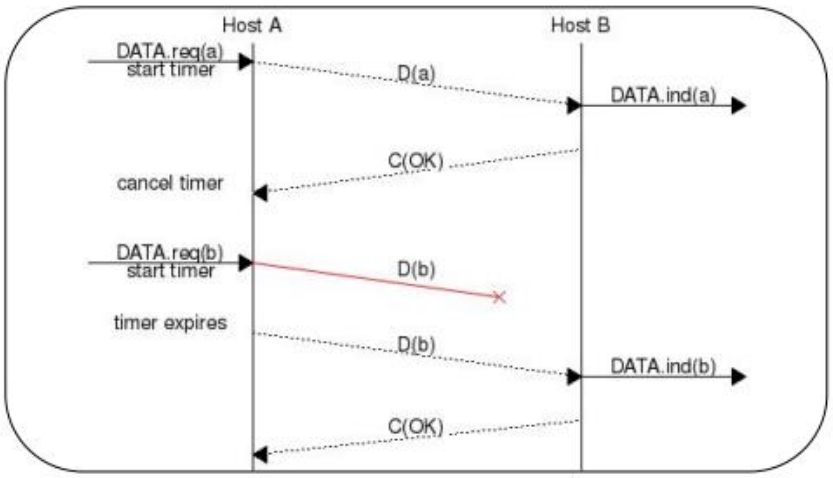


- ∇ Send msg – wait until you received acknowledgement of msg before proceed
  - ◊ Sending side waits for msgs coming from upper layers – once it receives request for msg to be sent, will send msg

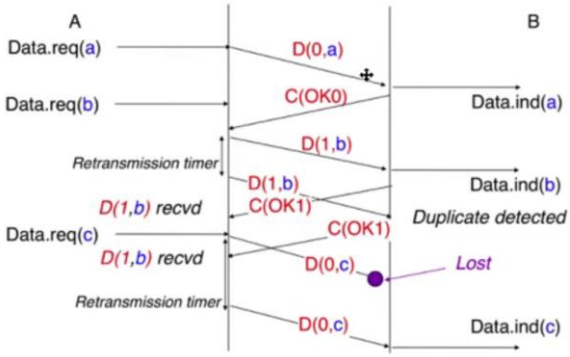
- Once msg sent, goes into waiting state – waits for an OK (acknowledgement that previous msg has been received)
- Once OK received – can now wait for msgs from upper layers
- Receiver side similar – wait for segment, once received, indicate to app that msg has been received
  - Msg processed and acknowledgement sent back

DATA FRAMES LOST/DAMAGED

- Lost data frame is re-transmitted after time-out
- Send msg a and start timer
  - Once acknowledgement received, cancel timer and send nxt msg b (start timer)
- If sent msg lost = no acknowledgement
  - Once timer expires, re-transmit msg b



Sequence Diagram

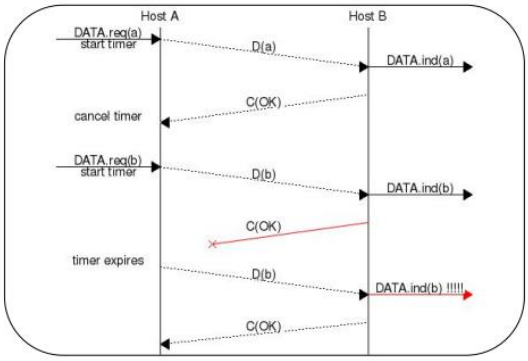


Retransmission timer + alternating bit protocol

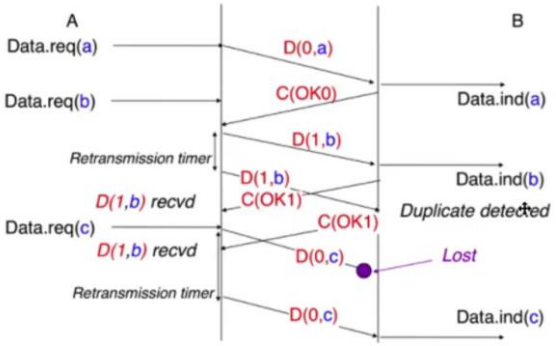
- Msg a sent, received – acknowledgement sent back and received
- Msg b is transmitted and received – gets acknowledged
  - Acknowledgement received late so b is re-transmitted and becomes duplicate
- Add value to data – sequence number
  - E.g. D(0,a) – acknowledgement is C(OK 0)
- Using sequence number, can detect that it already received msg b but receiver still acknowledges and sends OK

- ∇ Sender knows that msg for b has been received
- ∇ Sender sends c which gets lost
  - ◊ Once timer expires, re-send c
- ∇ Alternate between 0 and 1 to determine whether msg is one that has been acknowledged/not
- ∇ **Retransmission timer + alternating bit protcol**

CONTROL FRAMES LOST/DAMAGED

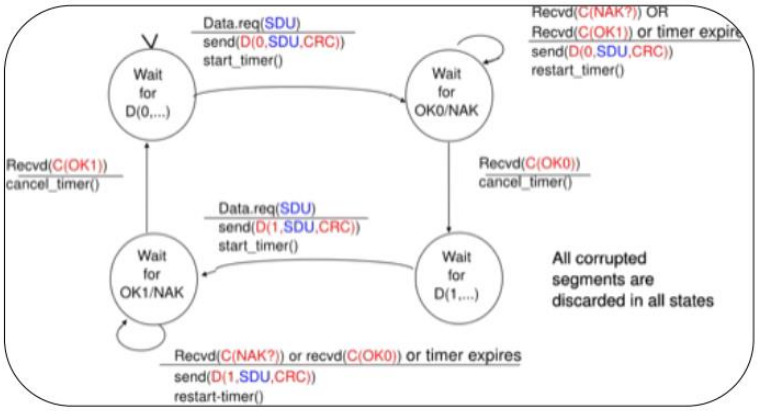


- ∇ Send msg b – msg received and acknowledgement sent
- ∇ Acknowledgement gets lost
- ∇ Re-transmit msg b – now msg received as duplicate
- ∇ Using a timer = not best solution



Retransmission timer + alternating bit protocol

Finite State Machine Diagram (Sender side)



- ∇ Initial state – wait for msg sequence 0

- ◊ Wait for msgs from upper layers ()
- ◊ Once receive msg that needs to be transmitted, send this msg with sequence number 0 and error correction code
  - ♥ Could be hash/checksum added to protocol – used by receiver side to determine whether msg is an error/not
- ◊ Send(sequence #, msg, checksum value)

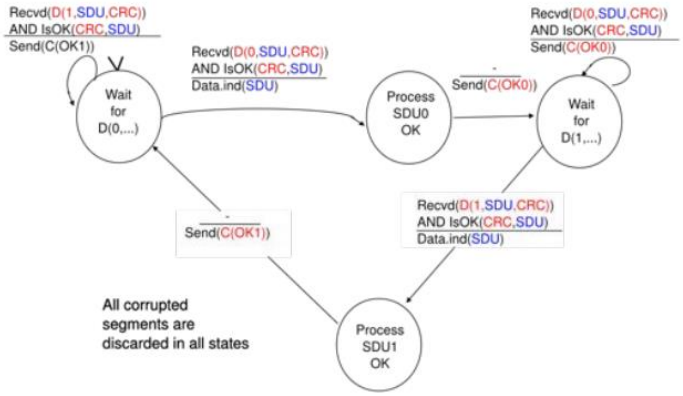
∇ Sender moves into state of waiting for acknowledgement (ack)
- ◊ Stop-and-wait protocol – won't send any msgs until receive ack for msg sequence 0
  - ♥ Waiting for an OK on msg 0 or negative ack
- ◊ Receiver can send negative/positive acknowledgement, or an OK for a msg not sent in 0 state (i.e. a previous msg)
  - ♥ Implies msg 0 not received/something else
- ◊ Re-transmit msg, with same checksum (that was sent in initial state)
  - ♥ Restart timer and stay in same sate
- ◊ Once positive acknowledgement for msg 0 received, cancel timer and move to different state
  - ♥ If negative ack received or receive an okay for the incorrect msg, or timer expires – retransmit msg and stay in same state

∇ Can now send new msg with sequence # 1
- ◊ Same process as initial state
- ◊ Send msg with sequence # 1 + checksum and start timer

∇ Once msg sent – move into acknowledgement state
- ◊ Same process as previous acknowledgement state
- ◊ Waiting for ack of msg 1 – stay in this state until receive ack
- ◊ Once ack received, move into wait state with msg sequence 0 again

∇ Alternating bit protocol ensures always receive acknowledgement for correct msg that waiting for
- ◊ A msg is set with an alternating bit sequence (0/1) – enables us to determine that an ack is for msg that has just been sent/not
  - ♥ If receive an ack for different msg than one that was sent – know that something has gone wrong and should try to re-transmit

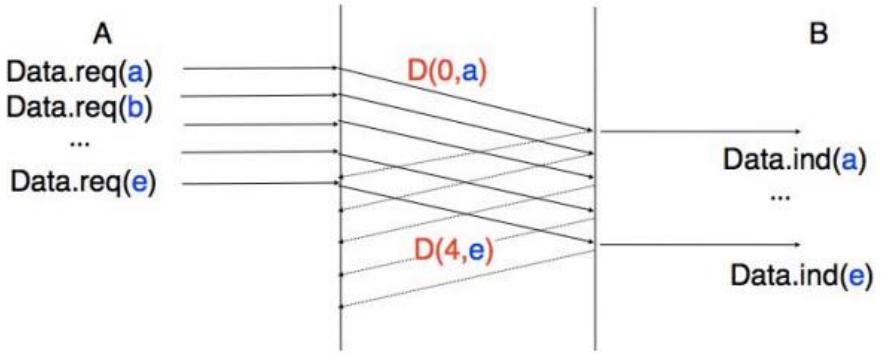
Finite State Machine Diagram (Receiver side)



∇ Receiver is in waiting state – waits for msg with particular sequence number
- o Sender and receiver synched – knows to wait for msg with sequence number 0

- ∇ If receives msg with sequence number 1 – then will ack received msg BUT remain in waiting state
    - Still want to receive msg 0
- ∇ If receive msg 0 – check that msg is okay
    - If okay – indicate to app that new msg has been received
- ∇ Process that msg and send ack for msg 0
- ∇ Go into waiting state for next msg with sequence # 1
    - Remain in this state and until receive msg 1 and it's considered OK (it passes the checksum)
    - If receive msg 0, then ack by sending an OK
        - i.e. we received msg 0 – msg 0 is NOT sent up to app for processing
            - Msg only sent up for processing if received in the correct state (e.g. waiting for msg 1 and receive msg 1)
- ∇ Indicate to app, process msg and send ack for msg 1
- ∇ Move back to state of waiting for msg 0
- ∇ Basic way of achieving reliability:
    - Have receiver ack that it receives a msg and have sender to have timeout that allows it to transmit msgs that are being acknowledged
    - Need some way of knowing that msg being acknowledged is the correct one (i.e the one we're waiting for)
    - This stop-and-wait has limitations – send and wait until receive positive ack of msg that we're waiting for
        - Slows down nw system capacity

Pipelining



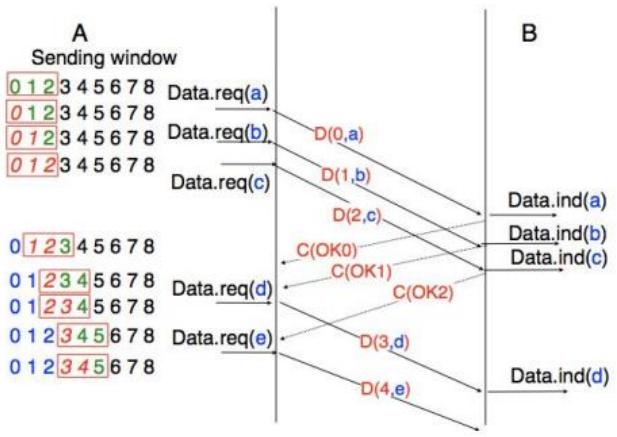
- ∇ With stop-and-wait:
    - Inefficient – sender has to send one msg and wait until msg is acknowledged before can send another
    - Depending on how long it takes between send and receive ack – sender wastes a lot of time being idle
- ∇ Pipelining: sender transmits several consecutive frames before having to wait for an acknowledgement after each frame
- ∇ Each data frame contains a sequence number
    - Response to msg will include sequence number
- ∇ Once get response to first msg, have already sent a # of msgs

- Will continue to get acks for msgs that has been sent

WITH SLIDING WINDOW

- How to ensure that pipelined higher transmission rate doesn't overload the receiver?
  - i.e. How many msgs should the sender be allowed to send before they have to wait for ack?
    - If just keep sending without waiting – receiver gets overwhelmed and doesn't know how process all of them
    - Where do we keep those msgs?
      - Receiver has limited buffer to keep msgs that haven't been processed
- Sliding Window:
  - Set of consecutive sequence numbers that the sender can use when transmitting frames without being forced to wait for an ack
    - Sender can transmit **W** unacknowledged frames before being forced to wait for an ack from the receiving entity
      - i.e. have a fixed # of msgs that allow sender to send before msgs get ack
      - When window is full/used up all the sequence #s that can be sent without ack = now have to *wait*
        - Then can only use as much as has been acknowledged.
      - Can't have more than the window size ito msgs that have yet to be ack

- Example (1) above:
  - Have a window of size 5 – allowed to send 5 msgs without acknowledgement
    - Have sent 2 msgs that aren't yet ack (6,7)
    - Thus have 3 more sequence #s to use before require ack (8,9,10)
  - If ack for 6 and 7 received, window shifts right
    - Can send two more sequence numbers



- ∇ Example (2) above:
  - ◊ Have window size = 3
  - ◊ Sender allowed to send 3 msgs without needing ack
    - ♥ Once 3 sent, have to wait for ack
  - ◊ Start with 0,1,2
    - ♥ Send A uses slot 0
    - ♥ Send B uses slot 1
    - ♥ Send C uses slot 2
  - ◊ Window is full – have to wait until acks received
    - ♥ Receiver sends ack – OK0
      - Shift window right – now have a new sequence # to use (3)
    - ♥ Similarly for Ok1, OK2
    - ♥ Send D – uses slot 3
  - ◊ As receive acks, move window to the right
- ∇ With pipelining – receiver expects particular sequence number at each point
  - ◊ E.g. once it's received 0 (for example (2) – knows it must receive sequence # 1 and then 2
- ∇ Challenges:
  - ◊ Msgs sent are not all received correctly
    - ♥ E.g. received 0 but 1 is lost and 2 is received
    - ♥ What does receiver do?

WITH RETRANSMISSION

- ∇ Solutions to challenge:
  - ◊ Go-Back-N
  - ◊ Sliding window with selective repeat

Go-Back-N

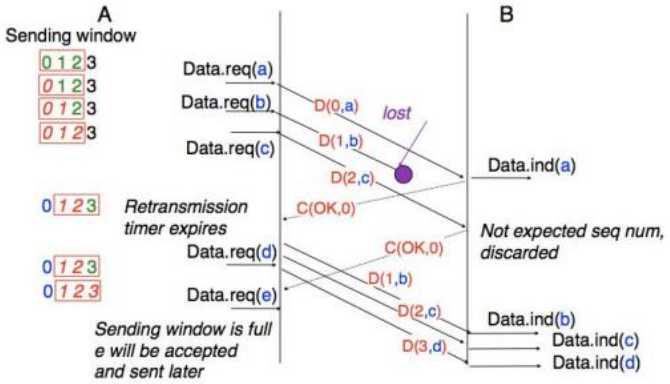
- ∇ Try to detect msgs that have been lost and re-transmit correctly
- ∇ Receiver only accepts frames that arrive in sequence
- ∇ Returns ack containing sequence # of last-in-sequence frame received
  - ◊ **Cumulative**: ack for sequence # X implicitly acks receipt of all frames whose sequence #s are earlier than X
    - ♥ i.e. sender acks sequence # for last frame that was correctly received prior
  - ◊ Sender retransmits <u>all unacknowledged frames</u> once it has detected a loss

- ♥ Even if only one msg lost and others received out of order (see example (1) below) – have to retransmit all msgs

∇ Discard all msgs that arrive out of order

∇ Example (1) below:

- ◊ Send msgs A (sequence #0), B (sequence #1), C (sequence #2)
- ◊ 0 and 2 correctly received, but 1 lost
- ◊ Receiver knows something is wrong
- ◊ Receiver discards 2 since it is out of order – sends an ack for 0 since it was correctly received
  - ♥ Indicates to sender that what they sent has been lost – last thing correctly received was 0
  - ♥ Implies to sender that all msgs after 0 have to be re-sent
    - i.e. re-transmit 1 and 2
- ◊ Receiver could send ack msgs up to sequence # 4
  - ♥ Implies to sender that all msgs from 0 to 4 have been correctly received



Selective Repeat

∇ Receiver accepts out-of-sequence frames

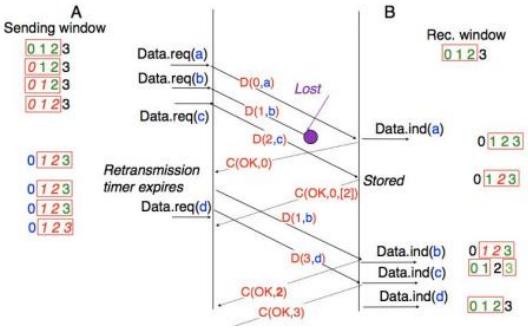
- ◊ Sender retransmits only frames that've been lost
  - ♥ Made possible because receiver indicates which msgs have been received
- ◊ All msgs that haven't been explicitly ack will be retransmitted

∇ Example (1) below:

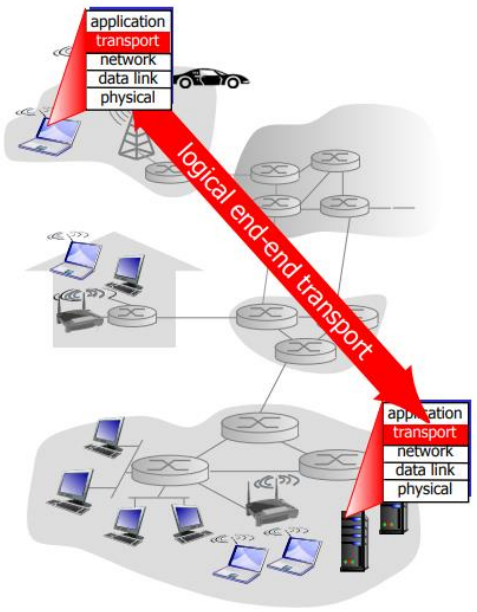
- ◊ Send msgs A (sequence #0), B (sequence #1), C (sequence #2)
- ◊ 0 and 2 correctly received, but 1 lost
- ◊ Receiver will receive 2 and store in a buffer
  - ♥ Receiver also sends ack that last correctly received msg was #0 AND that they've received 2
  - ♥ Tells sender that there's a problem
- ◊ Sender will retransmit msgs for which no ask has been sent
  - ♥ Know that 0 and 2 received
  - ♥ Will retransmit 1 – since retransmission timer expires
- ◊ Once sequence #1 received – msg will be processed and passed to upper layer
  - ♥ Sequence #2, in the buffer will also be processed and sent up.

Selective Repeat Receiver

Selective Repeat operation



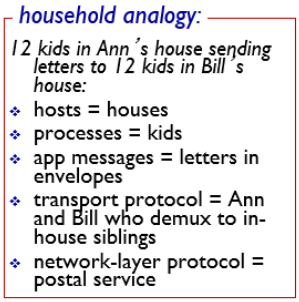
Transport Layer



- Provides logical communication between app process running on different hosts
  - Have process (as an app) running on host and there is another remote one
  - Want to establish connection between two processes – transport layer allows for the establishment of a virtual link
  - Link allows for the exchange of msgs as if they were running on the same host, simply exchanging msgs through some file descriptors (sockets)
- Transport layer is not implemented in any devices along the path – only implemented at source and at destination
  - Source – where traffic transmitted
  - Rcvr – where data is passed on to app
- Transport protocols run in end systems
  - Send side: breaks app msgs into segments, passes to nw layer
    - Large msg to be sent from app layer – at transport layer figure out the right size for the msg and split it up and package into segments (passed to nw layer for transmission)
  - Rcv side: reassembles segments into msgs, passes to app layer
    - Receives multiple segments that belong to the same msg – reassemble at transport layer and pass up to app layer (as single msg)
- More than one transport protocol available to apps:
  - Internet: TCP and UDP
- Transport layer concerned with your msg – how it's passed down from app layer and segmented properly at transport layer
  - Then shipped as those segments to the remote side – the receiving transport layer then putting those segments together and providing them to app layer
  - More than one protocol able to achieve this

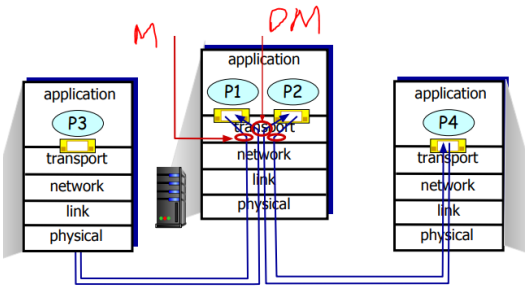
Internet Transport-Layer Protocols

- ∇ TCP: reliable, in-order delivery
  - ◊ Congestion control
  - ◊ Flow control
  - ◊ Connection setup – between src process and destination process
- ∇ UDP: unreliable, unordered delivery
  - ◊ Extension of "best-effort" IP
  - ◊ On rcvr side – within transport layer, receiver is able to check is msgs it's been received have errors or not
- ∇ Transport vs NW layer:



  - ◊ Nw layer: logical communication between hosts
    - ♥ Ensures package is delivered to the right building
  - ◊ Transport layer: logical communication between processes
    - ♥ Ensuring that package gets delivered to the right office
    - ♥ Relies on, enhances, nw layer services
      - Once it has packaged its msgs – pushes to nw layer to ensure it gets sent from src to dest host
      - At dest host – ensures msg is processed and passed onto app layer
    - ♥ Moving from src to dest process
    - ♥ Src/dest host will have several processes running – several sockets open at once
      - Need to ensure that msgs are **sent** from the *correct socket* and **received** at the correct socket

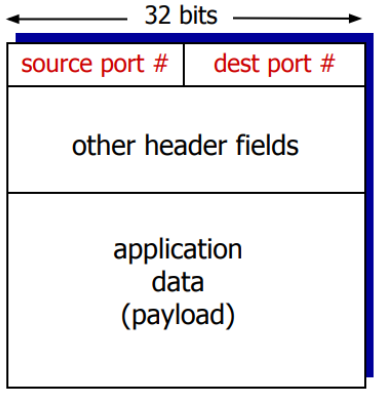
Multiplexing/Demultiplexing



- ∇ All communicating hosts may have several processes running on those devices => multiple sockets
  - ◊ Want to ensure that msgs are directed to the right sockets

- Multiplexing at sender: handle data for multiple sockets, add transport header (later used for demultiplexing)
  - Data coming in from app to different sockets – want to be able to send msgs out to multiple sockets on different hosts, different processes
  - Add appropriate transport headers – including port #s
    - Used to demultiplex (identify) which socket a msg came out of
  - How to handle multiple sockets at sender side whether local host sockets on sender or whether sender dealing with multiple sockets on remote side
- Demultiplexing at receiver: use header info to deliver received segments to correct socket (when msg received)
  - In diagram above – server has 2 sockets to deal with because it's communicating with different apps
    - Thus, need to ensure app msgs are put onto correct socket for outgoing msgs
  - When msgs come in from P3, P4 – want to ensure msgs sent to correct socket (either P1 or P2 at server side)

DEMULTIPLEXING



TCP/UDP segment format

- Host receives IP datagrams (msgs)
  - Each datagram has source IP address, destination IP address
  - Each datagram carries <u>one</u> transport layer segment – host received it as nw layer datagram and within that datagram there's one transport layer segment
    - On sender side, transport layer segment gets passed to nw layer and placed into a datagram
  - Each segment has source, destination port number
- Host uses IP addresses and port #s to direct segment to appropriate socket

Connectionless Demultiplexing

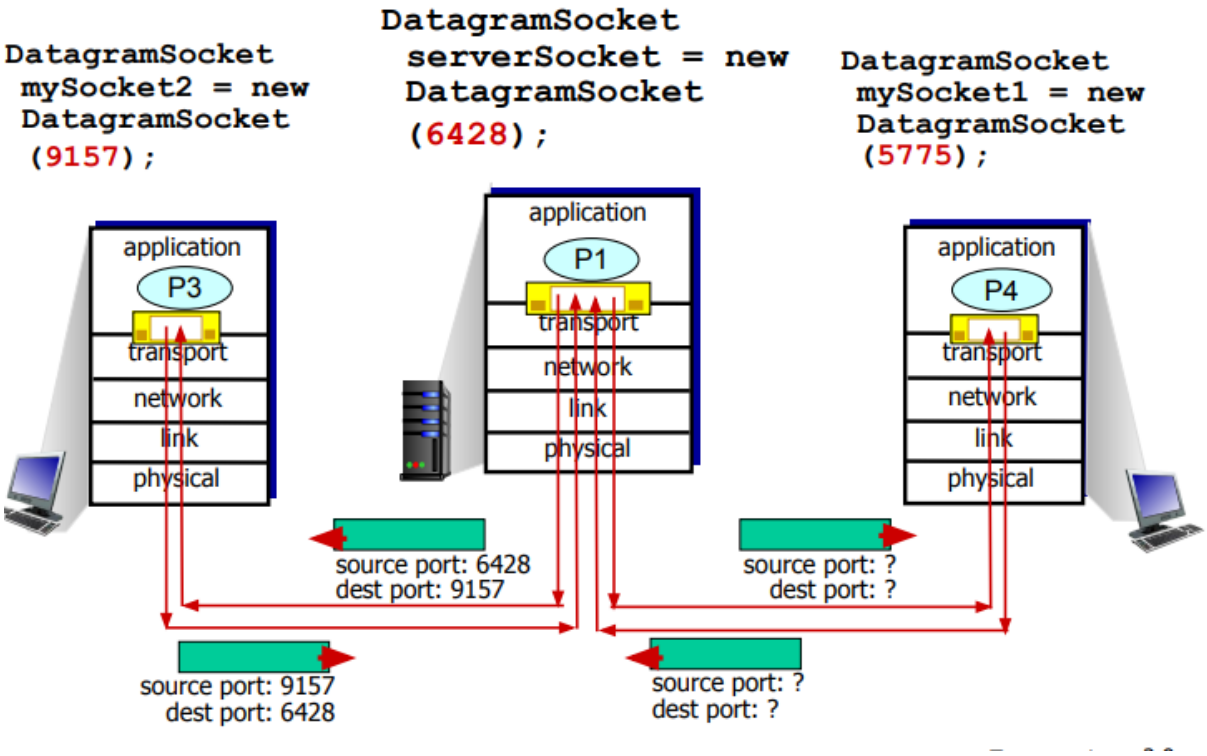
- Create a local socket on local machine:
  - Note: created socket has host-local port #
  - DatagramSocket mySocket1 = new DatagramSocket(12534);
- Creating datagram to send into UDP socket, must specify:
  - Destination IP address
  - Destination port #
- When host receives UDP segment:

- ◊ Checks destination port # in segment
- ◊ Directs UDP segment to socket with that port #

∇ IP datagrams with same dest port #, but different source IP addresses and/or source port numbers will be directed to same socket at dest

- ◊ E.g. many clients can send msgs to the same server socket

∇ Example (1):



- ◊ Have server running on port 6428 – socket established on this number
- ◊ Host running on P3 sends msg to server – sends msg to port # on server
  - ♥ Dest port is 6428
  - ♥ Source port is 9157
- ◊ All msgs coming from different hosts e.g. P3 and P4 will have same destination port
  - ♥ P4 has destination port 6428 and source port 5775

Connection-oriented Demultiplexing

∇ When client and server establish connection, that connection uniquely used by that pair

- ◊ Thus, if server has several clients – has separate connection socket for each of those clients
  - ♥ Msgs that are coming are directed to same port # but must be sent to the correct socket that is assigned to each of those clients

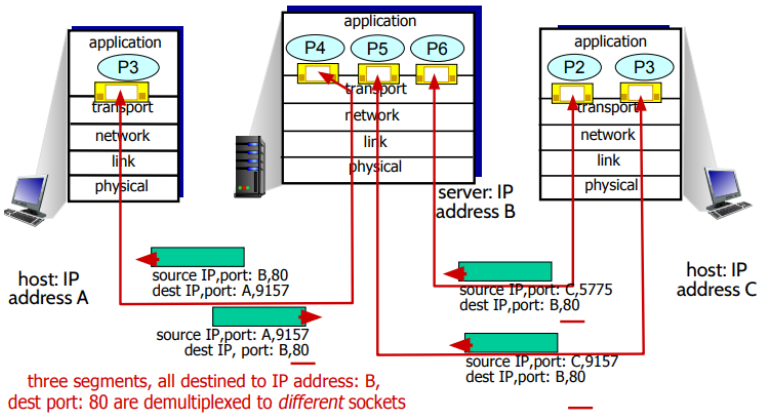
∇ TCP socket identified by 4-tuple:

- ◊ Src IP address
- ◊ Src port #
- ◊ Dest IP address
- ◊ Dest port #

∇ Demultiplexing: receiver uses all 4 values to direct segment to appropriate socket

- ◊ If another client connects to server on same port # - new socket has different src IP and src port #

- Server host may support many simultaneous TCP sockets – running multiple services at the same time and connecting to multiple different sockets (server side):
  - Each socket identified by its own 4 -tuple
- Web servers have different sockets for each connecting client
  - Non-persistent HTTP will have different socket for each request
  - Can have multiple sockets established by the same client IF client connecting on several instances
- Need to be able to distinguish where msgs coming from and going to in connection-oriented service
- Example (1):



  - Have 3 sockets established on server and 3 processes and 2 clients
  - On server side – all sockets established on same port # (port # 80)
  - Each client, as they est connection to server, had a socket created for them
    - Socket involves the 4-tuple – src IP, src port #, dest IP and dest port
  - On client side – src port and IP are the clients'
    - Dest port and dest IP is the server's
  - Msgs from client to server get sent to port 80
    - From server to client A (P3) – gets sent to port 9157
  - For host C
    - Have P3 and P2 processes
    - Can send msg through port 5775 or 9157 – what distinguishes these msgs are the src ports (indicates that they're separate sockets)
  - Three segments, all destined to same IP address: B, dest port: 80 are demultiplexed to different sockets
- In connection-oriented service each socket is matched exactly between one server and client

TCP: OVERVIEW

- Point-to-point: one sender, one receiver
- Reliable, in-order byte stream:
  - No "msg boundaries"
  - Distinguish one byte from another
  - E.g. msg could have been sent as 400 bytes but be received as different chunks as long as bytes are feeding in and they're being read as bytes on rcvr side
- Pipelined:
  - TCP congestion and flow control set to window

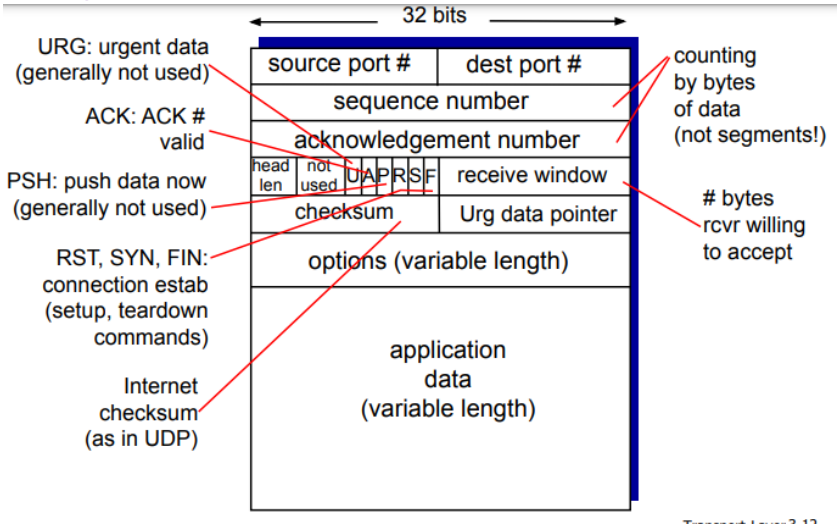
- ◊ Can only send certain # of bytes before have to wait for acks
- ◊ Window shifts as acks received

∇ Full duplex data:
- ◊ Bi-directional data flow in same connection
- ◊ MSS: max segment size

∇ Connection-oriented:
- ◊ Handshaking (exchange of control msgs) inits sender, receiver state before data exchange

∇ Flow-controlled:
- ◊ Sender won't overwhelm receiver
- ◊ Using window once more

TCP Segment Structure



∇ Each "row" is basically 4 bytes (32 bits)

∇ This is the TCP segment inside a nw layer datagram i.e. a packet – BUT this is not the packet itself

∇ Row 1:
- ◊ Src port # (2 bytes); dest port # (2 bytes)

∇ Row 2:
- ◊ Sequence #s – counting by bytes of data (not segments)
  - ♥ When send first msg – msg will have sequence # 0 i.e. byte 0
  - ♥ Can then send another e.g. 9 msgs – when send 10th msg, will have sequence # 10

∇ Row 3:
- ◊ Ack # - based on how many bytes have been received on receiver side
  - ♥ i.e. what is the next byte we're expecting
- ◊ E.g. have sent 9 bytes – sequence number on receiver side thus 9
  - ♥ When receive next msg, should have sequence number 10

∇ Row 4:
- ◊ Head len - indicates how many bytes there are in the header
- ◊ Not used
- ◊ Flags:
  - ♥ U = URG – urgent data (must be processed right away)

- Generally not used

♥ A = ACK – ack # valid
- Msgs coming from receiver send acks will activate this flag

♥ P = PSH – push data now
- Used if msg must be pushed up immediately to app layer
- Generally not used for public internet

♥ R = RST; S = SYN; F = FIN – flags used when connection established or being torn down
- RST – restarting connection
- SYN – msg that is trying to start new connection
- FIN – indication that want to tear down/finish connection

◊ Receive window - # bytes rcvr willing to accept

♥ This value can be adjusted depending on how much capacity rcvr has

∇ Row 5:

◊ Checksum – calculation done on entire TCP segment

◊ Urg data pointer – pointer that associates with URG i.e. if urgent flag indicated, urgent pointer used to ppoint to next msg that transport protocol should focus on

◊ Generally, not used

∇ Row 6:

◊ Options (var length) – server and client could have certain set of options that they've negotiated that facilitate their communication

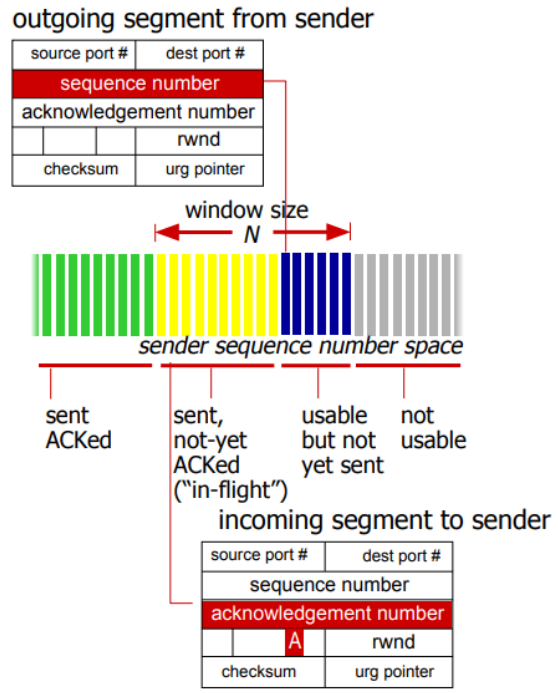
◊ Generally, not used in public internet but has use in internal systems

∇ Row 7:

◊ App data (variable length) – data that was passed down from app layer to transport layer

◊ E.g. msg, html file

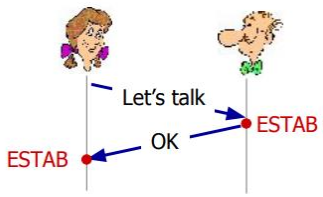
TCP seq numbers, ACKs



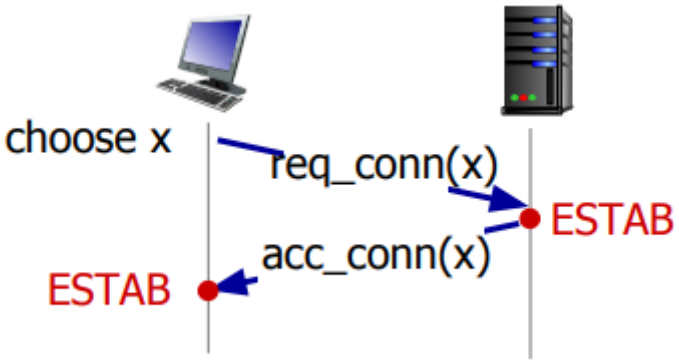
- Sequence numbers: byte stream "number" of first byte in segment's data
  - Relies on window size that have, how many msgs been ack, how many msgs "in flight" (sent, not yet ack), etc
  - E.g. sent 100 bytes, next message will have seq # 100 (msgs go from 0 to 99)
- Acknowledgements: seq # of nxt byte expected from other side
  - Cumulative ack
  - E.g. Received bytes 0 to 120 – sends ack indicating for 121
    - This indicates to sender that bytes 0 to 120 received correctly – and that byte 121 must be sent
- Diagram:
  - Sending window with size N
    - Green = msgs sent and acknowledged
    - Yellow = msgs sent but NOT yet acknowledged
    - Dark blue = msgs that have not yet been sent
  - From sender side – next sequence number will be the next byte that can be sent (1st dark blue bar)
  - On receiver side – ack # will be that of the latest (yellow) msgs that has been received
    - Tells the sender that all msgs have been received up to a specific point – so it's free to send the next byte
- How does rcvr handle out-of-order segments?
  - TCP spec doesn't say – to implementer
    - Go-back-N, selective repeat
- Use byte stream to work on seq # and ack #

Connection Management

Two-way handshake

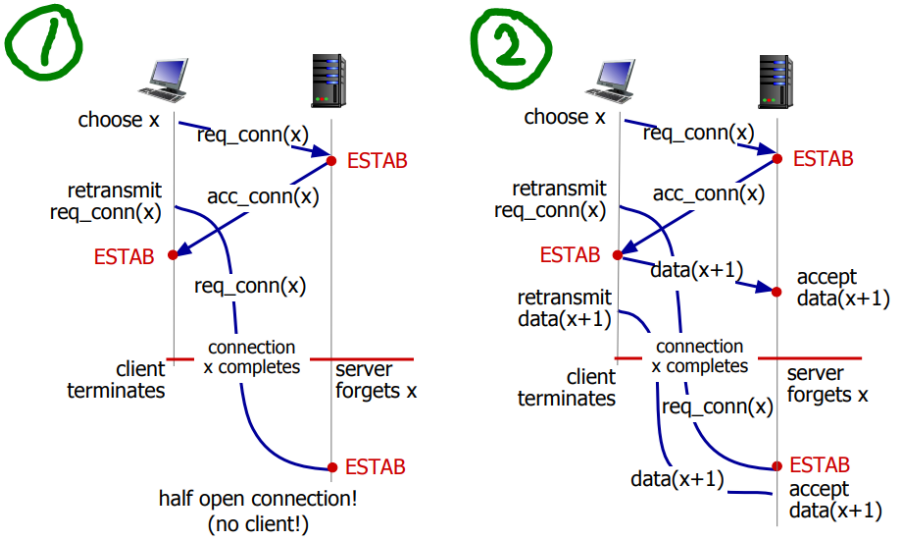


- ∇ Before you talk to someone, have to est a connection – connection is est as soon as you receive msg from other party
  - ◊ Once connection est, can talk.
- ∇ Connection oriented protocols have to setup connection before can transmit data between client and data



  - ◊ Client makes request for connection – server receives that connection
  - ◊ As soon and it receives that, est connection and sends back an ack of connection to client
  - ◊ Once ack received, client assumes connection has been established.
  - ◊ 2-way handshake
- ∇ Will 2-way handshake always work in nw?
  - ◊ Variable delays
  - ◊ Retransmitted msgs
    - ♥ E.g. req_conn(x) due to msg loss
      - Sent multiple connection requests – server recives first one, accepts earlier connection and sends ack to sender
      - Because client sent multiple requests, client doesn't know which connection is valid and being used for communication
  - ◊ Msg reordering
  - ◊ Can't "see" other side

FAILURE SCENARIOS

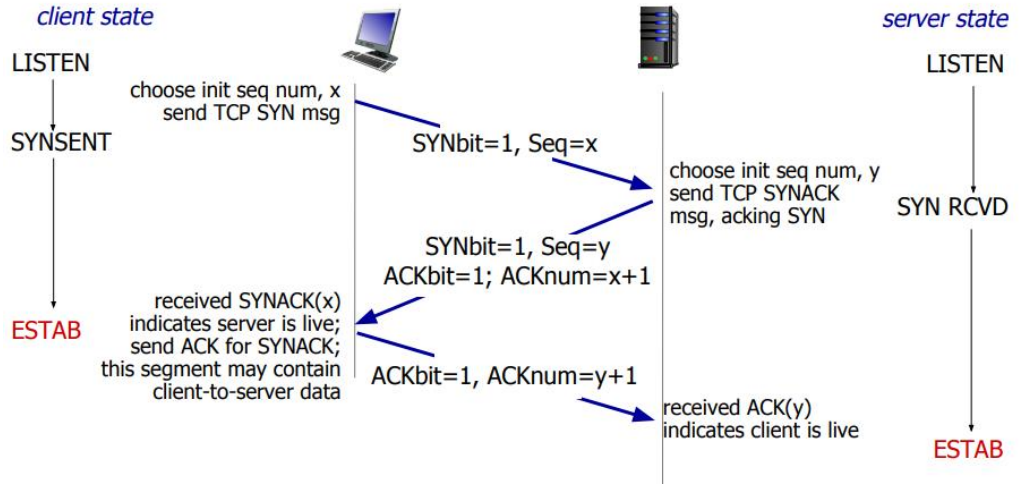


- (1) Half open connection
  - Client sends connection request
  - Server receives connection request and goes into established mode
    - Assumes connection with client is live
    - Sends back ack to confirm connection
  - If ack takes too long – client sends another conn request
  - Client receives initial ack and also goes into established mode
  - Client and server exchange msgs and then terminate connection
  - Earlier re-transmission conn request that client sent is received by server
    - Server assumes this is a new connection, but client hasn't actually tried to est another connection
  - Sends ack to client but client not online – server sits with half open connection
- (2) Data transmission errors
  - Client sends connection request
  - Server receives connection request and goes into established mode
    - Assumes connection with client is live
    - Sends back ack to confirm connection
  - If ack takes too long – client sends another conn request
  - Client receives initial ack and also goes into established mode
  - Client then begins to transmit data – which server receives
    - If client doesn't get ack quick enough for e.g. data(x+1), re-transmits data
  - Client and server exchange other msgs and then terminate connection
  - Earlier re-transmission conn request that client sent is received by server
    - Server assumes this is a new connection, but client hasn't actually tried to est another connection
    - New data that is received after new connection established is not new data, rather a re-transmission of data that client has already sent, and server already worked on.
    - Creates server-side problem – server may take "new data", process it and perform some actions

- Client may not want this since the actions may have already been completed earlier with original transmission of data

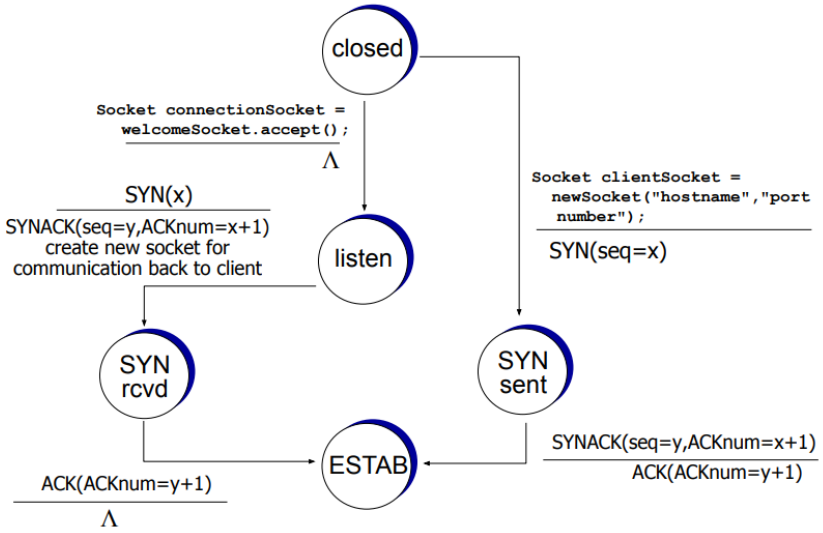
∇ Problem: not confirming on both sides as to whether a connection has been established/not
  ◊ Have to go beyond two-way handshake

TCP Three-Way Handshake



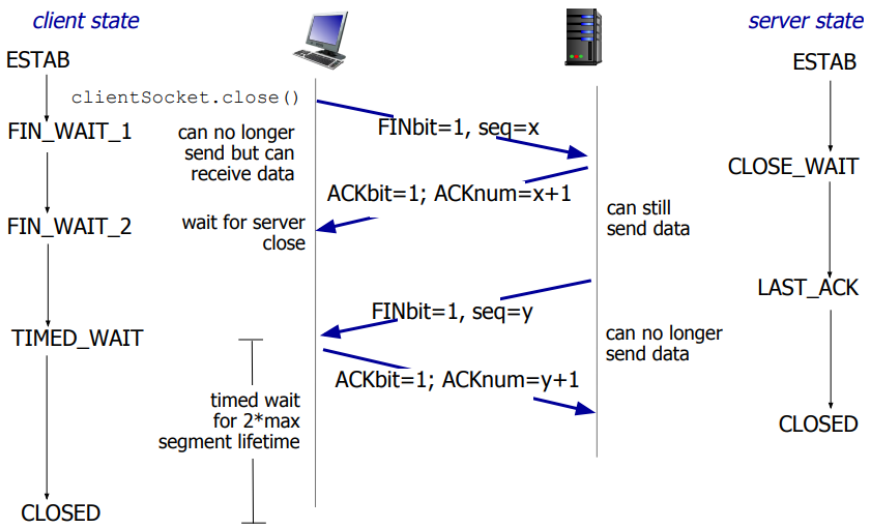
∇ Process:
  ◊ Client sends conn request
    ♥ Set SYN bit to 1 (refer back to TCP segments)
    ♥ Client chooses starting sequence number, x
    ♥ This is sent to server
  ◊ Once server receives conn request msg – chooses its own sequence # and sends back ack for SYN (conn request) msg
    ♥ Acks sequence # that it receives from the client
    ♥ Msg being sent from server includes – SYN bit =1, sequence # y, ACK bit =1, ack # sent back is x+1
      - Have SYN bit set to 1 because server is making its own conn request
      - x+1 indicates that all msgs to x have been received and server expecting msg with seq # x+1
  ◊ Msg received by client
    ♥ Client's conn request acknowledged – indicates to server that connection is live and has been accepted
    ♥ Now client acks server's connection request
    ♥ ACK bit =1; ack # = y+1
    ♥ Could transmit data at this point
  ◊ Server receive ack from client for it's own conn request to client – considers connection complete

FINITE STATE MACHINE DIAGRAM



- ∇ Client and server both in a **Closed** state when they're not waiting or sending connection request(s)
- ∇ Client creates socket to est connection with server
  - ◊ Server has already setup listening socket – has moved to **Listen** state
- ∇ Client sends connection request – indicates server trying to connect to and port #
  - ◊ Send msg with SYN bit = 1 and sequence number x
  - ◊ Client goes into state of **SYN sent** – waiting, until it receives SYNACK i.e. an ack of the SYN that it sent
- ∇ Client receives ack from server – should contain new seq # y, from server, and client's own seq # that it sent (x+1)
  - ◊ Client acks this ack – sends ack # y+1
  - ◊ Client goes into **Established** state
- ∇ Server is in listening mode – receives client's SYN msg with sequence # x
  - ◊ Server responds with conn request i.e. SYNACK with SYN bit =1, ACKnum = x+1 and it's own seq # y
  - ◊ Creates new socket for communication back to client
  - ◊ Goes into **SYN received** state – waits for ack from client
- ∇ Once server receives ack for its conn request with its own sequence #
  - ◊ Moves into **Established** state

TCP: Closing a Connection

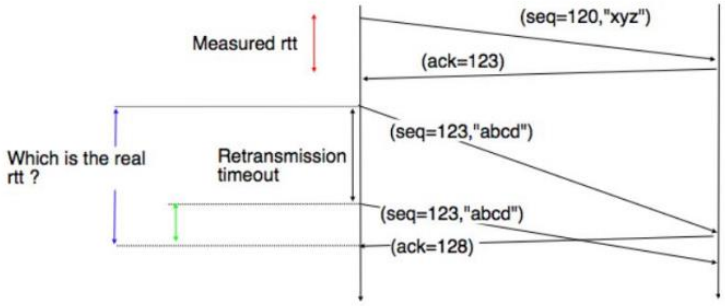


- ∇ Either side can initiate process of closing connection
- ∇ Client, server each close their side of connection
  - ◊ Send TCP segment with FIN bit = 1
- ∇ Respond to received FIN with ACK
  - ◊ On receiving FIN, ACK can be combined with own FIN
- ∇ Simultaneous FIN exchanges can be handled
- ∇ Time Series Diagram:
  - ◊ Client wants to close connection – sends msg with FIN bit = 1 and seq # x
    - ♥ Client closes its socket
    - ♥ Is no longer able to send new msgs but able to receive msgs
  - ◊ Server receives msg from client – acks msg with ACK bit =1 and ACK # = x +1
    - ♥ Server can still send data
  - ◊ Client receives ack – knows now that server received FIN msg
    - ♥ Has to wait – server hasn't closed connection on its side
  - ◊ Once server has no more data to send – sends FIN msg with FIN bit =1 and seq # y
    - ♥ Server can no longer send new data
  - ◊ Msg received by client – sends ack with ACK bit = 1 and ACK # = y + 1
  - ◊ Once ack receive by server – assumes connection closed
    - ♥ Client will remain open for a short while because there may be msgs on route that have been delayed
    - ♥ Client waits for 2 * max segment lifetime i.e. how long it takes for segment to get from server to client
      - i.e. delay between client and server
  - ◊ This system allows both sides to send any remaining msgs before connection fully closed down

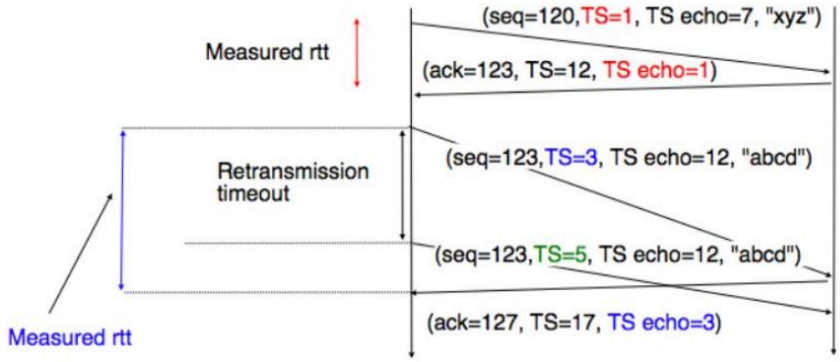
TCP Round Trip Time, Timeout

- ∇ Operational requirement of TCP – needs to be able to determine when a packet that has been transmitted can be considered lost
  - ◊ i.e how long should we wait before we determine that msg segment won't be arriving = timeout period

- Need to use Round Trip Time – amount of time it will take for a packet to be sent to a destination, acknowledged, and for that ack to arrive back to the sender
  - Challenge with use as is – it's variable because packets have to travel through different nw conditions at each time
  - How to measure RTT for TCP purpose?
    - Measuring RTT is difficult already
- Example:



  - Case 1: Send packet to destination – it arrives, and response sent back
    - This is considered measured RTT
  - Case 2: Send another packet – but takes much longer to get there and longer for response to come back
    - So, had to re-transmit before receiving ack
  - Receive response packet just after sending re-transmitted packet, how to know which is real RTT?
    - Is it time form first one to feedback, or from re-transmitted packet to feedback?
- Solution:



  - Disambiguating RTT measurement with timestamp
  - Msgs sent for measurement of RTT have a timestamp included
  - E.g. send 120 with TS (timestamp) = 1, msg = "xyz" and response has ack 123 with TS = 12 and TS echo =1
    - NB that ack is 123 because sent 3 bytes
  - For case 2:
    - Initial msg has TS = 3
    - Need to retransmit – second msg has TS =5

- This is what differentiates two msgs that have been sent

♥ Ack has TS echo = 3 – shows that measure RTT from original msg to ack

TIMEOUT

∇ How to set TCP timeout value?

◊ Longer than RTT but RTT varies

◊ If value too short – premature timeout, unnecessary retransmissions

♥ Might send packets, assume lost and immediately retransmit

♥ Creates congestion

◊ Too long – slow reaction to segment lost

◊ Need goldilocks timeout period – small # of retransmissions, but quick reaction to lost msgs

∇ How to estimate RTT?

◊ Sample RTT – measure time from segment transmission until ACK receipt

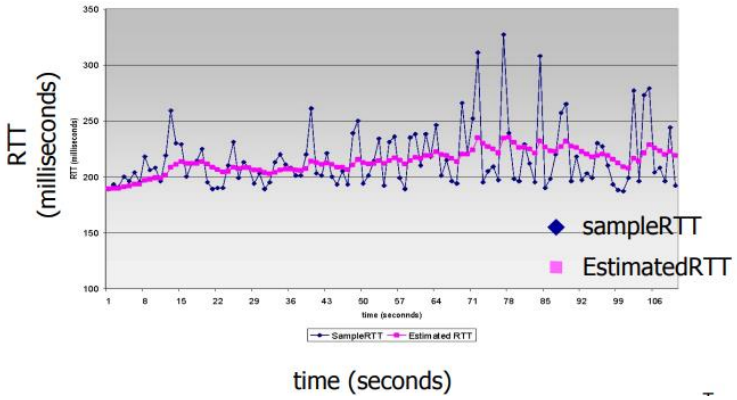
♥ i.e. at each point when we measure RTT, that becomes the sample RTT

◊ Sample RTT will vary – want estimated RTT "smoother"

♥ Average several recent measurements, not just current Sample RTT

- i.e. average (previous measurements) + current sample RTT

Estimated RTT



◊ Estimated RTT = (1- α)*EstimatedRTT + α*SampleRTT

◊ Exponential weighted moving avg

◊ Values of past constantly being averaged, but influence of past sample decreases exponentially as get new values

◊ Typical value for weight α = 0.125

◊ Graph: estimated RTT less variable (smoother) compared to sample RTT

∇ Is estimated RTT viable timeout value?

◊ Estimated timeout must also consider how far it is from actual current value of RTT

◊ Need safety margin – since there is a chance that at any point in time, our estimate is higher or lower than current RTT

♥ i.e. deviations from sample RTT (variation)

◊ Timeout interval – estimated RTT + safety margin

♥ Large variation in estimated RTT => larger safety margin

◊ Estimate sample RTT deviation from estimated RTT

♥ Dev RTT = (1-β)*DevRTT + β*|sample RTT – estimated RTT|

- Use abs value since sometimes sample RTT greater, other times estimated RTT higher (then value would be negative)
- Indication how far our estimate is from reality at each point in time

♥ Typically, β = 0.25

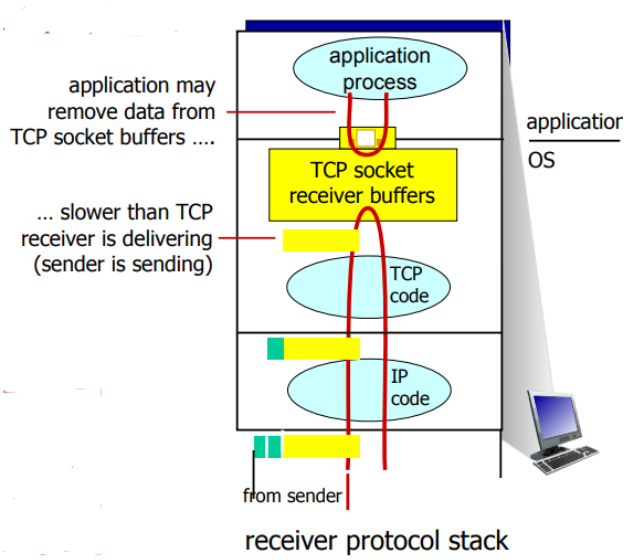
∇ Timeout Interval = estimated RTT + 4*DevRTT

◊ DevRTT is the safety margin

◊ Allows us to wait a sufficient amount of time before timing out

◊ Value used in all circumstances when it has to determine packet loss or whether connection timeout should happen

TCP Flow Control



∇ Flow control: rcvr controls sender, so sender won't overflow rcvr's buffer by transmitting too much too fast.

∇ Need flow control – have cases in which app is retrieving data from tcp socket at rate slower than how quickly we're putting new msgs into tcp buffer

◊ Have buffer on rcvr side -all msgs coming up through the layers are put into the transport buffer where app will collect msgs

◊ If app doing this at slower pace than new msgs are put into buffer, buffer will overflow – leads to scenario where some msgs have to be dropped when coming in due to lack of space in buffer

∇ Flow control – receiver indicates to sender that msgs are coming in faster than can be removed from buffer

◊ Receiver sends info about receive window – includes rcv buffer i.e. total amount of space available at transport layer receiving side

∇ RcvBuffer has some buffer data – data that has already been received but not yet collected by the app

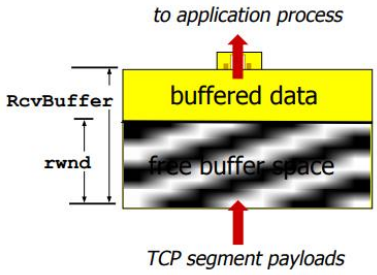
◊ That takes up some space

◊ Receive window being indicated to the sender is the amount of free buffer space still available in there

- ♥ Size of rcv window will change depending on how quickly buffer data is being removed
  - If quickly removing data in the buffer, then receive window will increase – rcvr can indicate to sender the new size of the receive window
  - If data comes into quickly – receive window decreases and alert sender

∇ Rcvr "advertises" free buffer space by including rwnd (receive window) value in TCP header of rcvr-to-sender segments

◊ Included when rcvr sends acks to sender



◊ RcvBuffer size set via socket options

- ♥ Typical default is 4096 bytes
- ♥ Done when TCP connection being set up

◊ Many OSs autoadjust rcvbuffer

- ♥ Once connection in operation

∇ Sender limits amount of unacked data to receive's rwnd value

∇ Guarantees rcv buffer won't overflow – sender constrained by this

Congestion Control

∇ Congestion

◊ Too many srcs sending too much data, too fast for nw to handle

- ♥ Can be one src sending too much data or several
- ♥ Nw refers to links and routers along the way

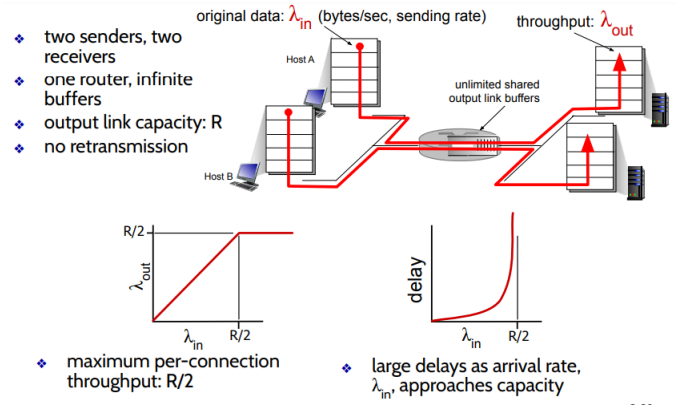
◊ Congestion =/= flow control (purely focuses on rcvr side)

◊ Manifestations:

- ♥ Lost packets – buffer overflow at routers
- ♥ Long delays – queuing in router buffers

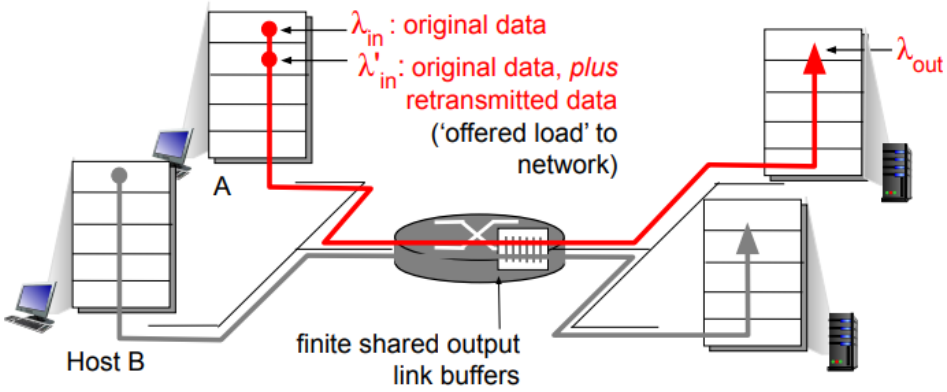
CAUSES/COSTS

SCENARIO 1

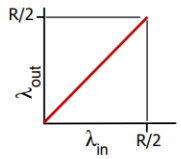


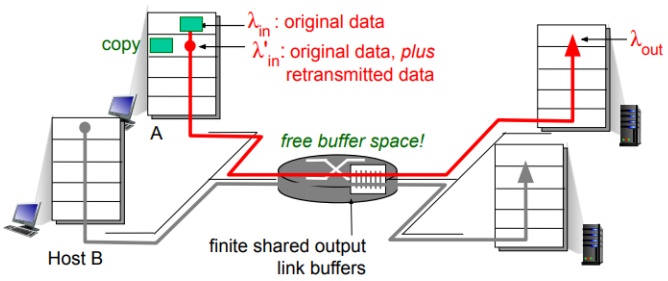
- ∇ Nw with:
  - ◊ Two senders, two rcrvs – hosts A and B are senders with C and D rcvrs
  - ◊ One router, ∞ buffers
- ∇ Output link capacity – R
  - ◊ Each host is able to use half of R
- ∇ No retransmission
- ∇ Ideal – two senders each sending msgs, and router is able to rcv and keep all incoming msgs until they're able to be transmitted
  - ◊ Amount of data sent – λin will equal data that comes out on receiving app side
  - ◊ Input/original data [λin] (bytes/sec, sending rate) = throughput [λout]
    - ♥ When increase input => inc throughput
- ∇ From POV of Host A – as it increases input (data being sent into nw), on rcvr side, throughput increases linearly
  - ◊ Done until reach max capacity – for each node, that is R/2
  - ◊ Once reach capacity – output flattens
- ∇ Above situation is ideal – actual observed events are:
  - ◊ Since buffer unlimited, will observe an increase in delay for all packets coming through the router
  - ◊ More data = longer queue
    - ♥ Each msg that is coming through thus waits longer
  - ◊ As approach link capacity – will experience much higher delays since queue is longer
    - ♥ Each new transmission has to wait longer
  - ◊ Limit to how quickly router can process msgs that are coming in, even with unlimited buffer capacity

SCENARIO 2



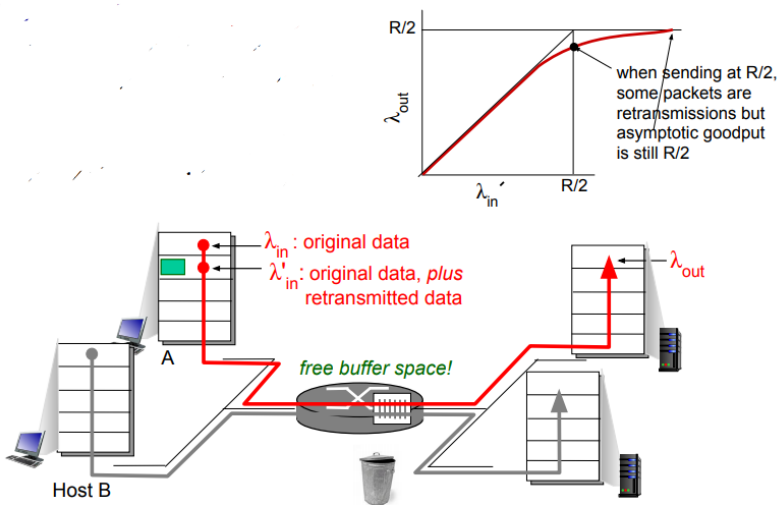
- ∇ Nw with:
  - ◊ Same as scenario 1 EXCEPT – one router, ***finite*** buffers
  - ◊ Finite buffers => once buffers full, some msgs dropped and lost at router
    - ♥ Now need to allow sender to retransmit msgs when they've been lost/take too long to be acked
- ∇ Sender retransmission of timed-out packets
  - ◊ Now actual transmitted data at transport layer is more than app data
    - ♥ App sending data but because some data has to be transmitted because of timeouts, then actual data flowing on input link is higher than input that was coming in from app itself
      - This is where congestion begins
      - BUT data that comes out through to app is the same as data that was sent in by src app – will achieve same max output
  - ◊ App layer input = app layer output
    - ♥ λin – original data
    - ♥ λin = λout
  - ◊ Transport layer input includes retransmissions
    - ♥ λ'in >= λin
    - ♥ λ'in – original data + retransmitted data
  - ◊ How to deal with problems retransmission causes?
- ∇ Idealisation – perfect knowledge





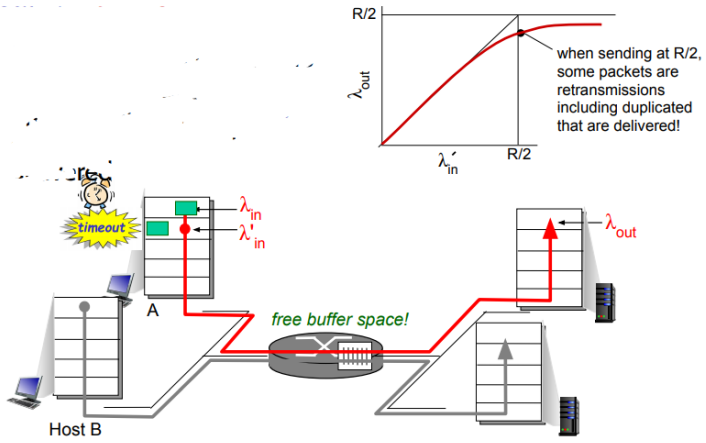
- ◊ Sender sends only when router buffers available – won't overwhelm the router
- ◊ Able to reach point where input = output
- ◊ Basically, only send when there is capacity – eliminates problem of retransmission (no need to retransmit because data won't be lost *because we're not overloading the buffer*)
  - ♥ Not real-life applicable – difficult to know when buffer space available
  - ♥ Even if measure it, by that time, another nw node may have transmitted and buffer pace may not be available anymore

∇ Idealisation 2 – known loss



- ◊ Packets can be lost, dropped at router due to full buffers
- ◊ Sender only resends if packet known to be lost
- ◊ Data that comes from app is copied – copy kept by transport layer buffer on sender side
  - ♥ When transport layer at src is informed that data has been lost, get copy and re-transmit data
- ◊ Still wasting some capacity of input link – but not too much, since only retransmitting what has been lost
- ◊ App layer input = app layer output
  - ♥ λin = λout
- ◊ Transmitted data on input link will be higher due to retransmissions
  - ♥ λ'in >= λin
- ◊ By sending only when buffer space available and only retransmitting when packet is known to be lost
  - ♥ App would have sent a higher amount of data than what can send out on rcvr side – since some data being sent through is retransmitted (see graph in diagram above)
- ◊ Since still minimising amount of the unnecessary retransmission – at higher sending rates, still able to reach capacity limit of output link
  - ♥ App might on rcvr side approach max link capacity
  - ♥ But, at max capacity ending rate – app won't receive actual link capacity, but will be close enough

∇ Realistic – duplicates



◊ Packets can be lost, dropped at router due to full buffers

◊ Aim is to only retransmit when packets are known to be lost – in reality, retransmit packets that haven't yet been lost

♥ Because have long queue – might time out when data still in router buffer (we retransmit prematurely)

◊ Sender times out prematurely, sending two copies, both of which are delivered

♥ More data sent on input link, and more sent on output link – since transmitting duplicates

◊ App on rcvr side – max throughput will not equal max capacity because some data being received is duplicates and that won't be passed onto app

◊ More congestion in the nw = more queues on buffer => more transmissions

♥ Longer queues = more packets timeout = more retransmissions = longer queues (cycle)

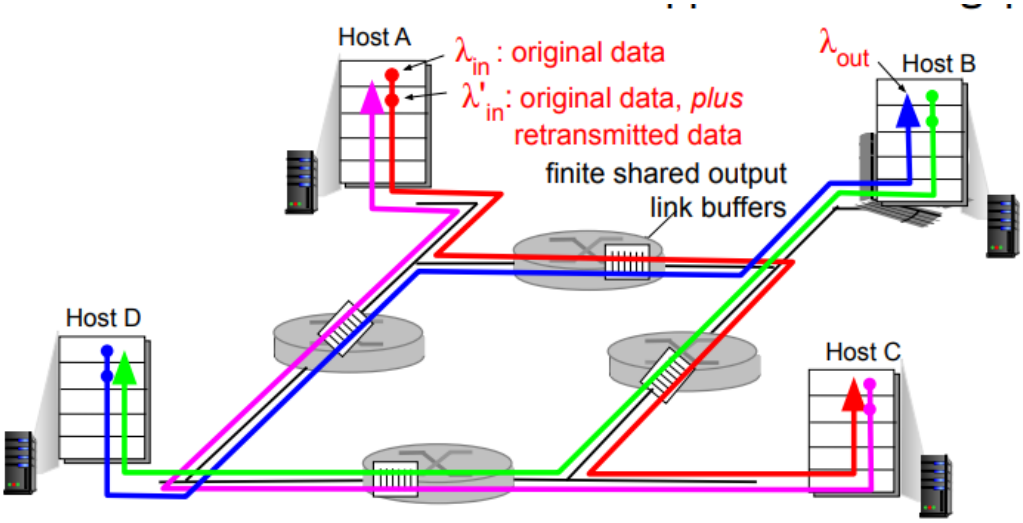
∇ "Costs" of congestion:

◊ More work (retransmissions) for given "goodput" (goodput = data that gets to app layer)

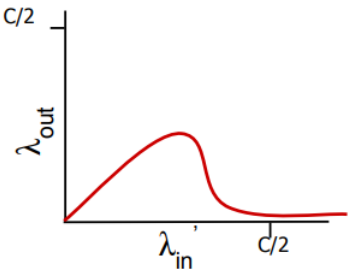
◊ Unneeded retransmissions – link carries multiple copies of pkt

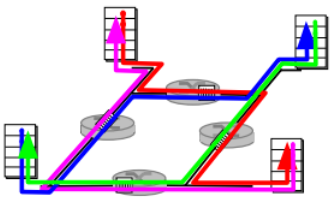
♥ Decreasing goodput

SCENARIO 3



- ∇ Nw with:
  - ◊ Four hosts – A, B, C, D
  - ◊ Multihop paths
  - ◊ Timeout and retransmit
- ∇ Host A:
  - ◊ λin – original data
  - ◊ λ'in – original data + retransmitted data
  - ◊ Sends more data – congests router => experiences timeouts
- ∇ At nw level – what happens as one part of nw becomes congested
  - ◊ Router experience high traffic volumes from A
  - ◊ But some traffic must come through from host D too (blue)
  - ◊ But since router congested, buffers are full
  - ◊ Host D is sending at lower rate – that should not congest nw, but since host A is sending too much data, this results in problem for host D since they're sending to the same router as A
    - ♥ Host D will loose most of its packets
  - ◊ Host A and D are experiencing severe congestion and increased retransmissions, whole nw could begin to completely collapse
    - ♥ This will affect other parts of nw – host D could redirect traffic to another router, but this could then affect hosts C and B
    - ♥ Host A's congestion also affects hosts B and C (red line)





- ∇ When increase input and reach capacity – the output will max at one point and then decrease sharply (see graph above)

- ◊ Because sending lots of retransmission and duplicates because traffic held up on router
- ◊ ISPs try to ensure that capacity is double the need

∇ Congestion "cost":

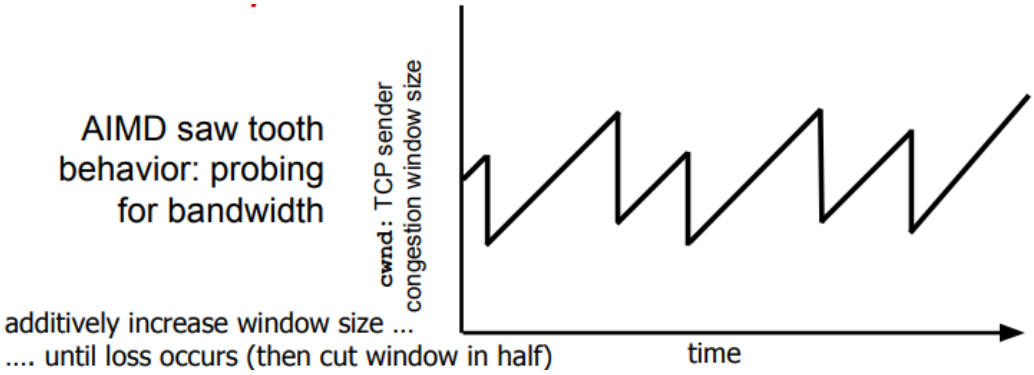
- ◊ When packet dropped, any upstream transmission capacity used for that packet was wasted

Approaches Towards Congestion Control

∇ Two broad approaches

- ◊ End-to-end congestion control
  - ♥ Managed by hosts i.e. devices sending traffic to nw
  - ♥ No explicit feedback from nw
  - ♥ Congestion inferred from end-system observed loss, delay (in packets being exchanged)
    - i.e src seeing acks are taking longer, being lost or timing out – indicates congestion which they need to act on
  - ♥ Approach taken by TCP
- ◊ Network assisted congestion control
  - ♥ Not a transport layer issue – managed at the nw layer
  - ♥ Routers provide feedback to end systems about situation in nw
    - Single bit indicating congestion – TCP/IP ECN (Explicit Congestion Notification), ATM
      - Indicates to hosts that there is congestion in nw and that they need to reduce their transmission rate
    - Explicit rate for sender to send at

TCP CONGESTION CONTROL

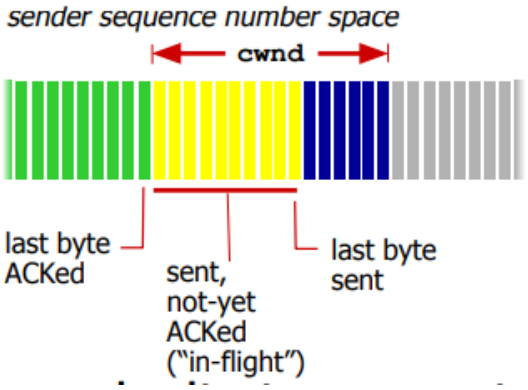


∇ Approach: sender increases transmission rate (window size), probing for usable bandwidth until loss occurs

- ◊ Additive increase - increase cwnd (sending window) by 1 MSS (Max Segment Size, i.e. msg) every RTT until loss detected
  - ♥ Start with small rate of transmission (small sending window) and slowly increase
  - ♥ Sending window – set # of msgs that can be sent by src without being acknowledged
  - ♥ NB – NOT increasing by bytes but rather 1 msg size

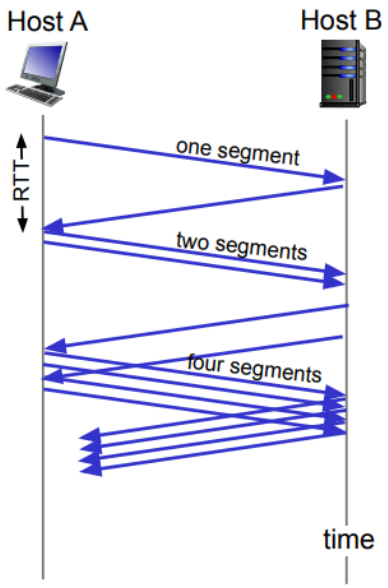
- ◊ Multiplicative decrease – cut cwnd in half after loss
  - ♥ E.g. had window size 12, decrease by half – sending window now size 6
    - Can now be increased to 7,8 etc until loss occurs again
  - ♥ Decrease has to be fast in case there is problem in nw

Details



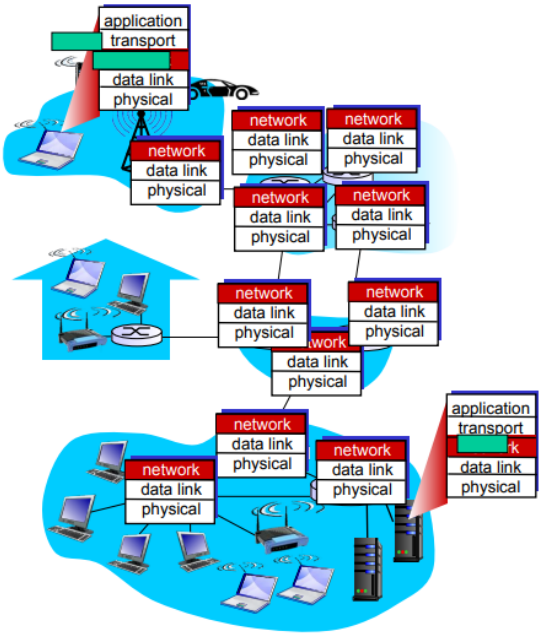
- ∇ Window is indication of how many msgs you can send before you can move on
- ∇ Series of sequence #s that is acceptable – within sending window have msgs that have been sent but not acked (yellow), and msgs that can be sent (blue)
  - ◊ If window full of yellow msgs => can't send anymore msgs, have to wait for ack
- ∇ Sender limits transmission:
  - ◊ Last byte sent – last byte acked <= cwnd
- ∇ Cwnd is dynamic, function of perceived nw congestion
  - ◊ Window can increase in size if there's less congestion, decrease if there's more
- ∇ TCP sending rate:
  - ◊ Limited by sending window
  - ◊ Roughly – send cwnd bytes, wait RTT for ACKS, then send more bytes
  - ◊ Rate approx. = cwnd/TT bytes per sec

TCP Slow Start



- Want beginning of transmission to be as slow as possible – need to determine what is an acceptable transmission rate in nw so don't overwhelm the nw immediately
- When connection begins, increase rate exponentially until first loss event
  - Initially cwnd = 1 MSS
  - Double cwnd every RTT
  - Done by incrementing swnd for every ack received
- Process:
  - Send a msg – wait 1 RTT
  - If ack received, double sending window – second 2 MSS
  - If ack received for both msgs – double to 4
- Ensures reach capacity of nw quickly before there is loss
  - Once reach loss, go back to TCP approach for congestion control

Network Layer



- ∇ Ensures that data units are able to move from src to dest host
  - ◊ Unconcerned about processes running on end system
  - ◊ Host to host process
- ∇ On sending side, data at transport layer (segments) is encapsulated in nw layer data unit i.e. datagram packet
- ∇ On receiving side, nw layer delivers segments (takes packet) and passes up to transport layer
- ∇ Nw layer protocols in every host, router
- ∇ What happens in between from src to dest host – datagram is examined at every router along the way
  - ◊ Router examines header fields in all IP datagrams passing through it – determines where to send it to next
- ∇ What does the nw layer header contain and how does is the info used at the nw layer?
  - ◊ What nw layer does, how it looks at info in header and how it determines where to send the packet so that it reaches the destination
- ∇ Diagram:
  - ◊ Have msg from app to transport layer – gets encapsulated in nw layer msg
  - ◊ At each nw layer device, msg is examined until it reaches destination host

Key Principles

- ∇ Addressing: each nw layer entity is identified by nw layer address
  - ◊ E.g. IP address
- ∇ Link layer Independence: service provided does not depend on internal organisation of the underlying datalink layers
  - ◊ E.g. have computer connected to nw via wireless link – nw layer sitting on top of wifi interface/datalink

- ♥ Another computer connected to nw via ethernet cable – nw layer sitting on top of ethernet datalink protocol
- ◊ Can have an internet of devices that're connected through different datalink protocols – but all devices able to talk to each other because they all have same nw layer (regardless of link layer supporting the nw)

∇ Two planes:

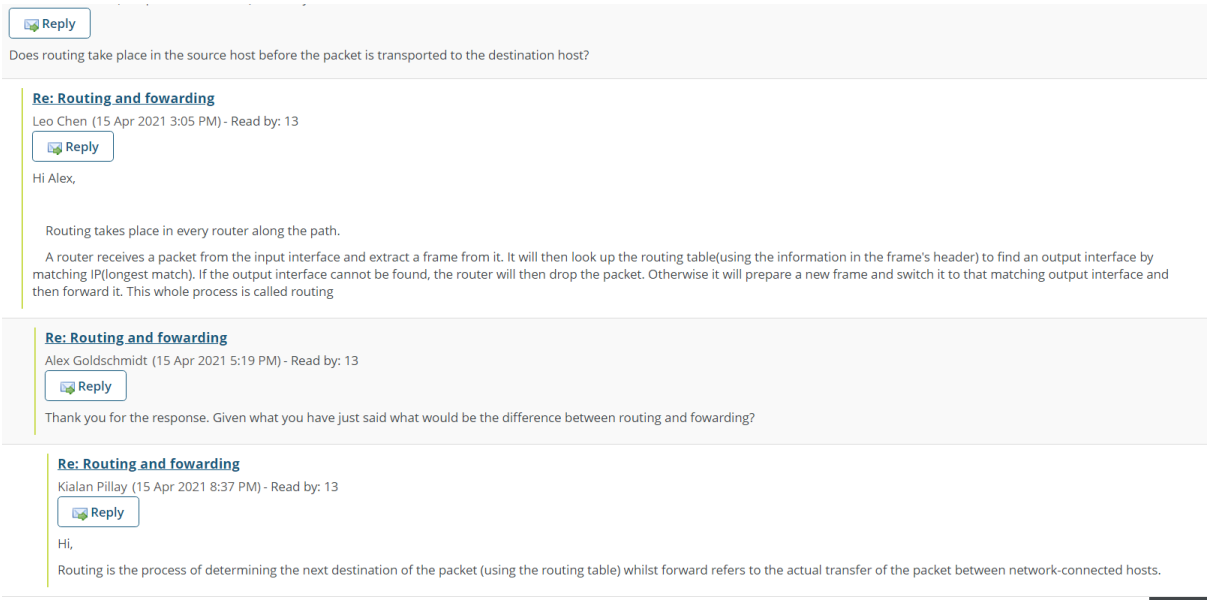
- ◊ Control pane (nw wide logic) – used to compute and install routing tables on routers
  - ♥ i.e. responsible for understanding entire nw for which host is part of
- ◊ Data plane (loca, per-router) – used by hosts and routers to create and process the packets that contain user data
  - ♥ i.e. data plane used to transform host's transport layer segment into nw layer packet – put in all info like src address, dest address etc and then sending into the nw
  - ♥ for router, concerned with receiving packet, inspecting header info and deciding where to send it next and forwarding it

∇ Routing: determine route taken by packets from src to dest

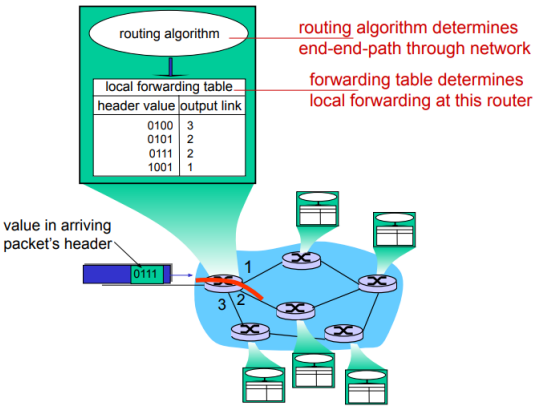
- ◊ Control plane function determining how traffic will flow from src to dest
- ◊ Routing algos
- ◊ Routing must be computed beforehand – before data can be sent
  - ♥ E.g. Use google maps – before begin journey, compute the path and then move – along the way, at each point where there's an intersection, (since everything mapped already), will just tell you whether to go left or right or straight

∇ Forwarding: move packets from router input to appropriate router output

- ◊ Sent data – comes through router
- ◊ Router looks through forwarding table and forwards accordingly using the table and the correct output port
- ◊ FORWARD data, we ***don't*** route the data

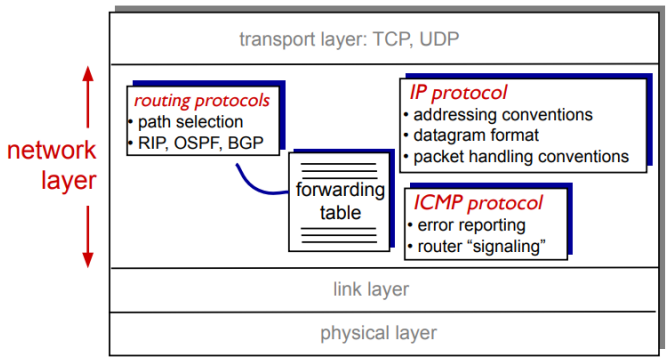


INTERPLAY BETWEEN ROUTING AND FORWARDING



- ∇ Routing and forwarding work together
- ∇ Routing algo determines end-end path through nw
- ∇ Forwarding table determines local forwarding at this router
  - ◊ Has header value column – each header value connected to certain destination address
    - ♥ E.g. value 0100 use output link 3
  - ◊ When packets come – router inspects incoming packet header and sends data out using the corresponding output link
- ∇ Don't always use dest address for lookup table – lookup tables can use different info
  - ◊ Some use nw IDs to decide how traffic should be forwarded
  - ◊ Header value – generic term, can be used to represent IP address of host or some other nw ID

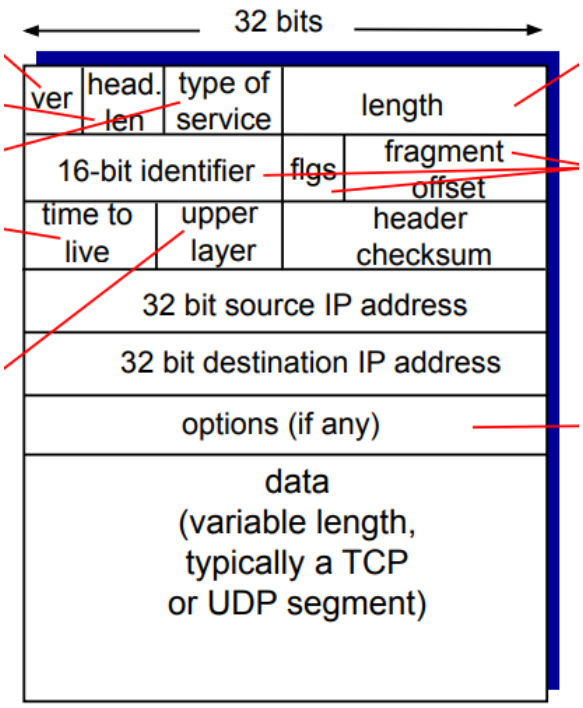
Internet Network Layer



- ∇ Have different protocols performing different functions
- ∇ Routing protocols – used to select path
  - ◊ RIP, BGP etc
  - ◊ Builds forwarding table
- ∇ Within forwarding table
  - ◊ IP protocol
    - ♥ Addressing conventions
    - ♥ Datagram format
    - ♥ Packet handling conventions (when packets come in)

- ICMP protocol
  - Used for nw management
  - Error reporting
    - If packet unable to go throough router, router may send ICMP msg to sender to say there is an error
  - Router "signalling" – used when establishing links between routers

IP DATAGRAM FORMAT



- Header structured in parts of 4 bytes = 32 bits
- Row 1:
  - Version (ver) – IP version # 4 or 6
  - Header length (head. Len) – how many bytes have in the header
    - Used when router receives packet – router needs to know how much info it needs to read in order to understand header info
  - Type of service – used to indicate the different categories of traffic that could be supported by nw
    - E.g. real-time traffic indicated here
    - Most nws ignore this – useful for internal nw when want to differentiate service
    - Sometimes used as congestion notification – explicit congestion notification bit taken from this set of bits
  - Length – total datagram length
    - Used to check whether all data for a packet has been received at each host
- Row 2:
  - 16-bit identifier – used to ID packet
    - Mostly used when need to break up packet into smaller pieces along the way – fragmentation

- ◊ Flags – used if need to fragment packet
  - ♥ Sometimes have big packets (size 1500 bytes) – along the way it goes to a router which can't process a packet that big
    - Packet can be broken down into smaller packets e.g. two packets of 700 and 800 bytes
  - ♥ When receive part of packet, will then know there must be other part(s) needed to complete packet
- ◊ Fragment offset – indicates which part of fragmentation that packet is
  - ♥ E.g. if packet is first fragment, will have offset 0; second has offset value based on # of bytes in first segment
    - First is 0 with 800 bytes, second fragment has value of 801 to indicate is starts from byte 801
- ◊ This row is used for fragmentation re-assembly
- ◊ NOTE: IP v6 doesn't allow for fragmentation

∇ Row 3:

- ◊ Time to live – number of hops this packet is allowed to go on for
  - ♥ i.e. how many routers this packet can go through before it becomes obsolete because it's expired
  - ♥ Decremented at each router
  - ♥ Generally, set to 64
  - ♥ If value goes to 0 before reach destination – packet discarded
    - Mechanism that is put there so that packets don't exist in the nw forever – just in case there is a loop in nw, or other nw problem, want the packet to expire and be thrown away
- ◊ Upper layer – upper layer transport protocol to deliver payload to
  - ♥ E.g. packet carrying UDP data – indicate for UDP
- ◊ Header checksum – used to verify that header is received correctly
  - ♥ Computed at source and placed into packet

∇ Row 4:

- ◊ 32-bit src IP address

∇ Row 5:

- ◊ 32- bit destination IP address

∇ Row 6:

- ◊ Options – different nw layer protcols and implementations may use this
- ◊ E.g. as packet move along, want to include timestamp when it was generated at source; or which routers the packet moved thorough on its trip

∇ Row 7:

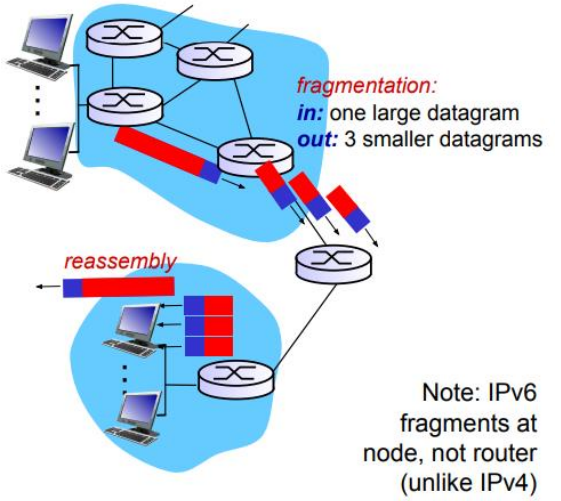
- ◊ Data – variable length
- ◊ Typically, TCP or UDP segment

∇ Overhead:

- ◊ Add 20 extra bytes for nw layer (rows 1 to 5)
- ◊ Min bytes for TCP header in transport layer - IP
- ◊ Have 20 bytes TCP and 20 bytes IP = 40 bytes + app layer overhead
  - ♥ When being transmitted form src – for every unit of data, adding this type of overhead
- ◊ Overhead can seem great – but impact of that depends on size of packet

- ♥ So, if packet is big e.g. 1500 bytes – adding extra 40 is not a big deal
- ♥ But if packet is only e.g. 50 bytes, overhead is much greater

IP FRAGMENTATION AND REASSEMBLY

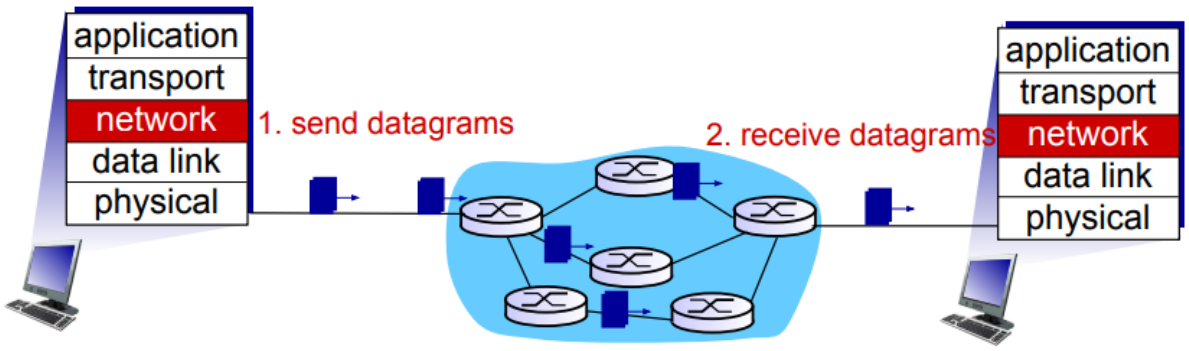


- ∇ Nw links have MTU – max transfer size
  - ◊ Largest possible link lvl frame
  - ◊ Different link types, different MTUs
- ∇ Packet generated from src – goes through nw and each device on nw has MTU
  - ◊ If packet goes through router whose MTU is lower than what packet size currently is – router implements some fragmentation to handle that packet
- ∇ Large IP datagram divided (fragmented) within net
  - ◊ One datagram become several datagrams
  - ◊ Reassembled only at final destination
  - ◊ IP header bits used to identify, order related fragments
- ∇ Diagram shows single packet which gets fragmented into 3 units
  - ◊ Have to copy header into each of those units
  - ◊ Overhead much higher
  - ◊ Packets move along path – get reassembled at destination
  - ◊ Gets reassembled using 16-bit identifier and the fragment offset and the passed up to transport layer

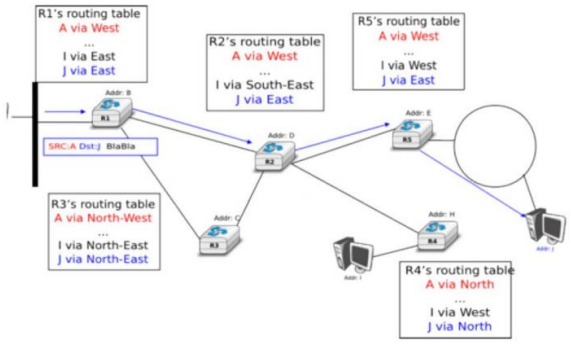
Network Layer Service Models

- ∇ Connection, connectionless service
- ∇ Datagram nw provides nw-layer connectionless service
  - ◊ From nw layer pov – no establishment of circuit/connection from src to dest host
  - ◊ Packets flow in nw from different paths and find their way to the dest – *don't* establish dedicated circuit between src and dest host
- ∇ Virtual-circuit nw provides nw-layer connection service
  - ◊ Establish connection from src to dest
  - ◊ Dedicated circuit which packet flows through from src to dest device
- ∇ Analogous to TCP/UDP connection-oriented connectionless transport-layer services but:
  - ◊ Service – host-to-host (rather than transport layer, which is process to process)
  - ◊ No choice- nw provides one or the other
  - ◊ Implementation – in the nw core

Datagram networks

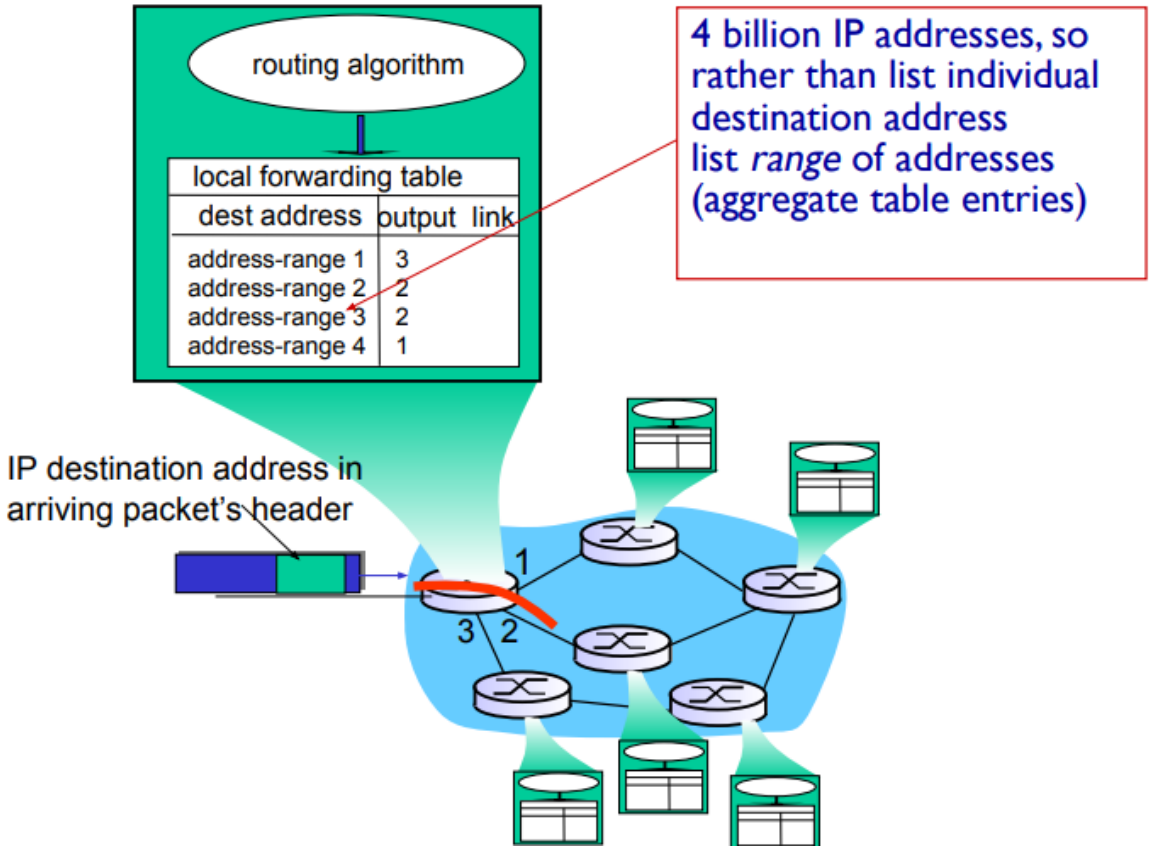


- ∇ No call/circuit setup at nw layer
  - ◊ Because of this, packets must contain src and dest host address, plus data
  - ◊ At each router, need above info to know how to forward it
- ∇ Routers – no state about e2e connections (hop-by-hop forwarding)
  - ◊ Have no awareness about connections that exist between srcs and ests – just examine packet as it goes through router
    - ♥ i.e. won't know if packet is part of particular flow/communication
  - ◊ Deal with each hop as it comes and goes
- ∇ Packets forwarded using dest host address (+ ref forwarding table)
  - ◊ At each router, will look up dest address and determine where to forward packet
- ∇ Diagram:



- ◊ Forwarding table at each router – indicates diff destinations that router aware of and which output links to take when packets must reach those destinations
- ◊ E.g. R1 says to get to A, need to go via West link; I via east, J via West etc
  - ♥ Have packet with src address A and dest J going through R1
  - ♥ R1 looks through its table – forwards packet to East link

DATAGRAM FORWARDING TABLE



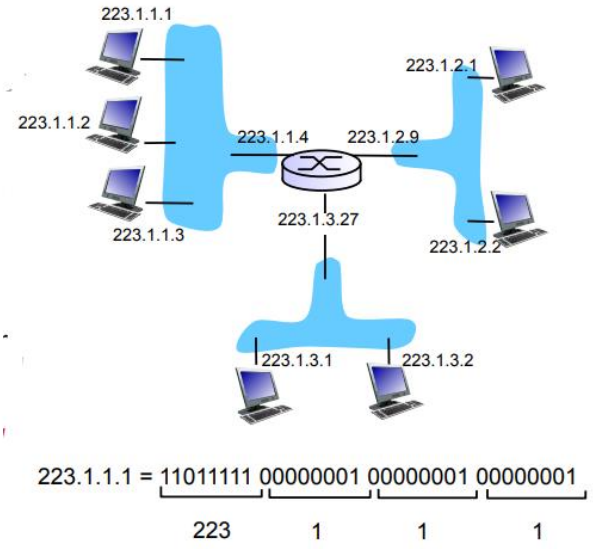
- ∇ Dests are IP addresses (since hosts ID by IP addr)
- ∇ With IPv4, over 4 billion IP addresses – can't list all in each forwarding table
  - ◊ Lookup time would be too long
- ∇ IPv6 has even more Ip addr compared to v4 – impossible to list all in each table
- ∇ Solution: range than list IP addresses, list ranges
  - ◊ Range of IP addresses which indicates a particular nw – first IP and last IP that belong to particular nw belong in that range

- ♥ All packets going to those hosts in nw will have to go through same route – assume those hosts are at the same physical nw
- ♥ Scalable

◊ IP addresses belonging to same nw will be in same range

◊ E.g. all IP addresses that belong to UCT are within a particular range

IP Addressing



∇ IP address: 32-bit identifier for host (binary format) [IPv4; Ipv6 has 128 bits]

- ◊ Assigned to an interface on a host
  - ♥ Includes router interface
- ◊ Device has wireless interface, ethernet interface (or multiple thereof)
  - ♥ Multiple interfaces => multiple IP addresses
- ◊ When sending data, sending it to particular interface on host => have to assign IP to interface

∇ Interface: connection between host/router and physical link

- ◊ Routers typically have multiple interfaces – because they're connecting multiple nws to multiple outgoing dests
- ◊ Host typically has one/two interfaces
  - ♥ E.g. wired ethernet, wireless 802.11

∇ IP addr associated with each interface

- ◊ 32 bits are in 4 groups of 8 bits
- ◊ Take each group of 8 bits – can convert to decimal value

DATAGRAM FORWARDING TABLE

Destination Address Range	Link Interface
11001000 00010111 00010000 00000000 through 11001000 00010111 00010111 11111111	0
11001000 00010111 00011000 00000000 through 11001000 00010111 00011000 11111111	1
11001000 00010111 00011001 00000000 through 11001000 00010111 00011111 11111111	2
otherwise	3

- ∇ Have range of IP addr that use for each entry in forwarding table with corresponding outgoing link
- ∇ Ranges will have upper bits be the same up to a point, differing towards the end for individual hosts
  - ◊ E.g. entry 1 in diagram above has 11001000 00010111 00010000 00000000 through to has 11001000 00010111 00010111 11111111
    - ♥ For an IP address to belong to this range, must have upper bits be the same (as red above)
- ∇ Have another entry within forwarding table called "otherwise" – any entry that doesn't fit into any of the other ranges in the table is sent out on that corresponding link (3 for diagram above)
  - ◊ In practice, should be router that should know more about where precisely to send packet to
- ∇ Above diagram is idealisation – in practice, sometimes within range there's smaller nw with its own IP addresses or could have nw with IP addresses that fall into more than one range

Longest Prefix Matching

Destination Address Range	Link interface
11001000 00010111 00010*** *********	0
11001000 00010111 00011000 *********	1
11001000 00010111 00011*** *********	2
otherwise	3

examples:

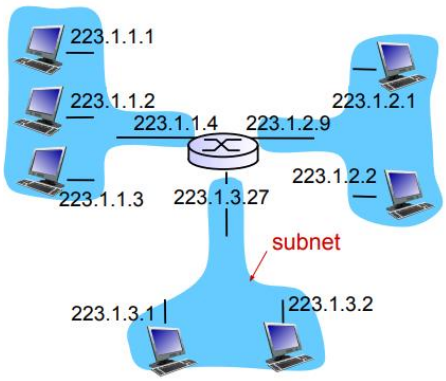
DA: 11001000 00010111 00010110 10100001 which interface?

DA: 11001000 00010111 00011000 10101010 which interface?

- ∇ Longest prefix matching: when looking for forwarding table entry for given dest addrr, use **longest** addr prefix that matches dest addr
- ∇ Lower bits get represented with asterisk
  - ◊ Basically, saying we don't care about these bits – only care as long as the IP addr matches the upper bits, they belong to this range
- ∇ Diagram – possible or one IP to match with multiple entries

- E.g. row 2 and row 3 will match to a point - 11001000 00010111 00011000 (row 2) and 11001000 00010111 00011 (row 3)
- When looking in forwarding table – router matches it with the longest add prefix to decide on how to forward packet
- Examples:
    - (a) matches with row 1
    - (b) matches with row 2 and row 3 – but because row 2 has 24 bits specified (longer than row 3, which has 21 bits), it gets finally matched with row 2

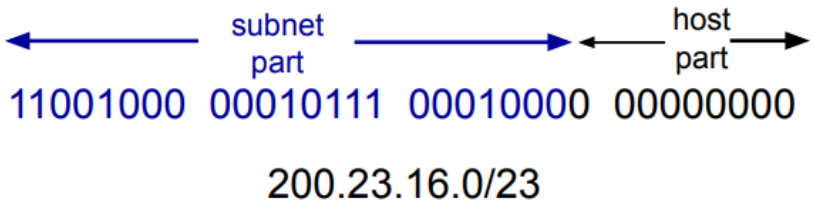
Subnets



network consisting of 3 subnets

- IP address
  - Subnet part – high order bits
    - Indicate what nw an IP addr belongs to
  - Host part – low order bits
    - ID individual interface for that IP addr
- Subnet: device interfaces with same subnet part of IP address
  - Can physically reach each other without intervening router
    - Hosts within same subnet can exchange data without traffic having to go through router
    - If one subnet needs to exchange data with another subnet – then will need to send data through router
  - Diagram – shaded blue section with some # of hosts
    - Top left – 223.1(larger nw).1.* (subnet)
    - Top right – 223.1.2.*
    - Bottom – 223.1.3.*
  - Smaller part of a larger nw – have specific set of bits that ID it as subnet
- Creating subnets i.e. how to divide nw
  - How many bits to borrow from host to create subnets?
    - E.g. to create 4 subnets, need 2 bits (since 2^2 = 4)
  - Generally, divide nws into powers of 2
- E.g. have 200.23.16/21
  - Can ignore 200.23 – only 16 bits and we need 21
  - Convert 16/21 – 200.23.00010|000.00000000
    - Before | is subnet part
    - Everything after is the host – can run from 000.00000000 (lower bound) to 111.11111111 (upper bound)
    - i.e. 200.23.0001|0000.00000000 to 200.23.0001|111.11111111
  - if combine lower bound, and convert to decimal – 0001111 becomes 23 and 11111111 is 255

IP Addressing – CIDR



- ∇ Classes InterDomain Routing
  - ◊ Subnet portion of address of arb length
    - ♥ i.e. can adjust the size of the subnet length to create bigger/smaller nw
    - ♥ Smaller subnet – more bits on host side => can create more addresses
    - ♥ Decrease # of bits on host part – smaller nws because have fewer IP addresses within each subnet; but have more subnets (With smaller # of hosts within them)
  - ◊ Address format – a.b.c.d/x
    - ♥ Where x is # bits in subnet portion of address
- ∇ Example:

Example

Consider the routing table as shown in the table. Suppose packets with the **destination IP address 137.145.128.128** arrives at the router R.
What will be the outgoing link interface (if any) for forwarding these packets?

Destination network	Outgoing link interface
137.158.16.0/20	1
137.156.0.0/14	2
136.0.0.0/6	3
128.0.0.0/1	4

Example

Answer in 3 steps:

1)Write out the binary prefix

Destination network	Outgoing link interface	Binary prefix
137.158.16.0/20	1	10001001.10011110.0001
137.156.0.0/14	2	10001001.100111
136.0.0.0/6	3	100010
128.0.0.0/1	4	1

2)Write out the destination address in binary

Destination address	Binary address (matching prefix underlined)	Outgoing link interface
137.145.128.128	10001001.10010001.10000000.10000000	3

3)Check longest prefix matching

- ◊ NB conversion to binary from decimal

Binary conversion

- ➔ Binary to Decimal, given **binary number** 10100011.
- ➔ $10100011 = (1 \times 2^7) + (0 \times 2^6) + (1 \times 2^5) + (0 \times 2^4) + (0 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0)$
- ➔ $= 128 + 0 + 32 + 0 + 0 + 0 + 2 + 1.$
- ➔ $= 163.$

Given: Decimal Number = 163

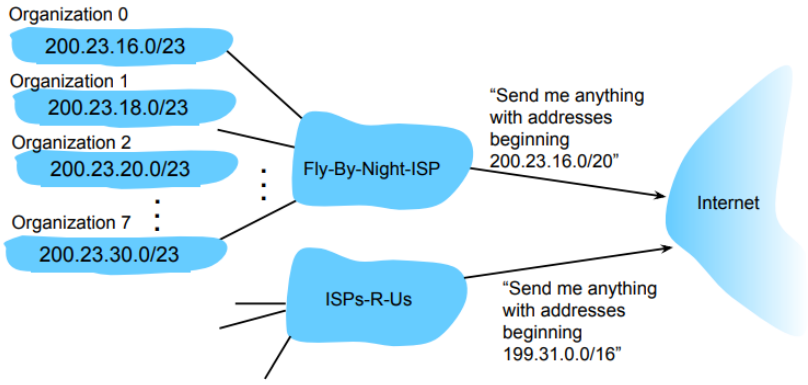
Divide by 2	Result	Remainder	Binary Value
163 ÷ 2	81	1	1 (LSB)
81 ÷ 2	40	1	1
40 ÷ 2	20	0	0
20 ÷ 2	10	0	0
10 ÷ 2	5	0	0
5 ÷ 2	2	1	1
2 ÷ 2	1	0	0
1 ÷ 2	0	1	1 (MSB)

Decimal **160** => binary **10100011**

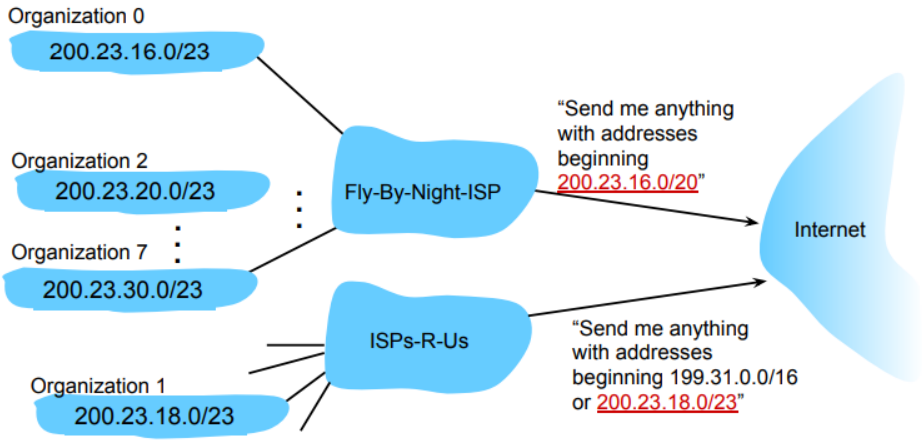
- ◊ For entry 1, since have /20 => means only interested in first 20 bits and drop the rest

Hierarchical addressing: route aggregation

- ∇ Hierarchical addressing allows efficient advertisement of routing info
- ∇ Aggregate many routes into one nw announcement
  - ◊ E.g. have several entries 137.158.18.0/20, 137.156.0.0/14 that are kind of similar
    - ♥ Aggregate so only have one entry
- ∇ Example:
  - ◊ CIDR principles employed here

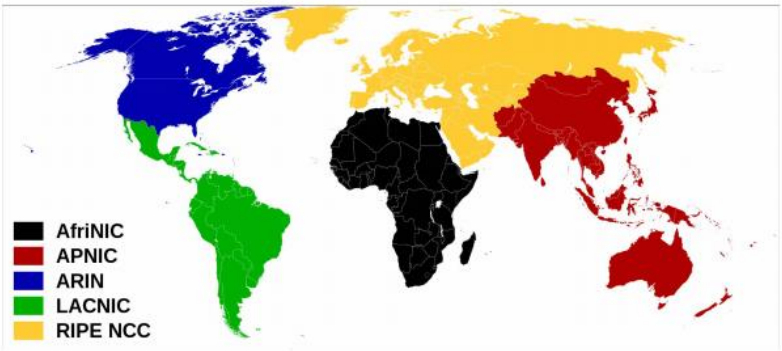


  - ◊ Have ISP with block of addresses assigned to different organisations that are ISP customers
  - ◊ Org 0 has block 22.23.16.0/23; org 1 has 200.23.18.0/23 and so on for orgs 2 to 7
  - ◊ ISP needs to inform other ISPs or nws that any traffic for any of the subsets can be sent to that ISP
    - ♥ ISP needs to inform internet of all these addresses – rather than announce each subnet individually, aggregate
    - ♥ Combine all subnets into single block of addresses and announce that
    - ♥ If expanded IP addr block for each org – will see that first 20 bits will be the same for each (only differ on lower order bits)
  - ◊ ISP announces it as single /20
  - ◊ When traffic being sent by internet to that address, when those packets arrive at ISP, ISP will examine its own forwarding table to determine which one of the orgs the traffic must be forwarded to
- ∇ Example (2):



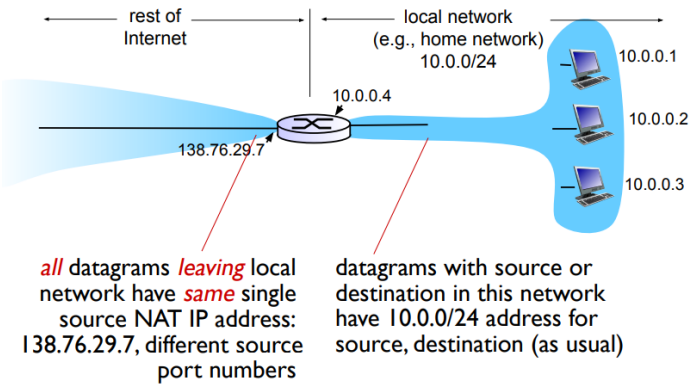
- ◊ Had orgs 0 to 7 with ISP
  - ♥ But org 1 moves to different ISP – moves with IP block
- ◊ First ISP will continue to announce block it originally did – includes all org blocks, including the one that moved
- ◊ Second ISP will need to announce org 1 block more specifically
  - ♥ Announces it as 200.23.18.0/23
  - ♥ By doing that, routers in the internet will have two entries that could match
- ◊ Use longest prefix match – when routers doing matching for dest address, IP address that belongs to org 1 will match more specifically with announcement by second ISP
  - ♥ Traffic will thus be sent there

Origin of IP Addresses

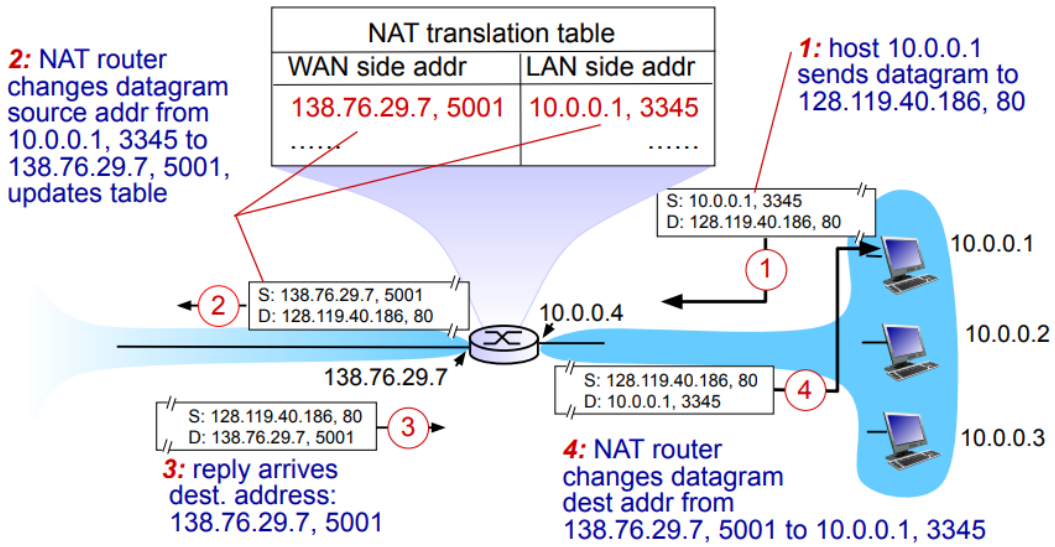


- ∇ Internet Corporation for Assigned Names and Numbers (ICANN)
  - ◊ Allocates addresses (through RiRs)
  - ◊ Manages DNS (assigns domain names, resolves disputes)

Network Address Translation (NAT)



- ICANN reserved certain range of IP addresses to be used a private IPs – used with LAN
  - Can be used within several nws – only visible within the nw itself (so no conflicts)
- Ranges (IPV4):
  - 10.0.0.0 to 10.255.255.255
  - 172.16.0.0 to 172.31.255.255
  - 192.168.0.0 to 168.255.255
- NB public IP needs to be unique globally – can't have every device have unique global IP addr (IPv4 limit)
  - Thus, need to use private IP for LANS – router gets public IP addr
- NAT
  - All datagrams leaving local nw have same single src NAT IP addr, with different port #s
  - Datagrams with src or dest in this nw have private IP addr within above range for src, dest
- Diagram:
  - All datagrams leaving LAN (through router) have src IP 138.76.29.7
    - Internet sees nw in the sense of this one IP
  - All datagrams with src/dest in nw have 10.0.0/24 addr for src, dest
- How to manage multiple connections?
  - May have several devices connected within LAN – but all have their private src IP changed to router public IP
  - How to differentiate which device an incoming packet meant for?
  - Solution – maintain NAT translation table on router
- Example (NAT table):



- ◊ Host 10.0.0.1 sends datagram out to some remote server 128.119.40.186, port 80
  - ♥ Packet will initially have src addr 10.0.0.1
- ◊ When packet reaches router, it puts its own IP as the src and replaces src port # with its own too (138.76.29.7, port # 5001)
  - ♥ Puts all info into table
- ◊ Packet goes through nw until it reaches dest
- ◊ Dest replies – sends packet to 138.76.29.7, port # 5001
- ◊ Response reaches router – looks up in NAT table
  - ♥ Sees it must send response to 10.0.0.1 with port # 80
  - ♥ Replaces the dest IP and port # to above
- ◊ Using 16-bit port #s – can have 2^16 concurrent sessions in LAN (using NAT)
  - ♥ If router runs out of possible port #s to allocate to new sessions – then can't have more sessions accepted on LAN
  - ♥ 60,000 simultaneous connections with a single LAN-side address

∇ NAT = controversial – violates # of nw design principles

- ◊ Routers should only process up to layer 3
  - ♥ Physical -> link -> nw layer
  - ♥ Port #s are for transport layer – by introducing port #s at nw layer, violating separation of processes and services
  - ♥ For incoming packet, must inspect packet up to transport layer header to know the src port and dest port
    - Routers only supposed to check up to nw layer headers i.e. IP addresses, **not** port #s
- ◊ Violates e2e argument
  - ♥ Communication is meant to be from src process to end process – src and dest process must have a link that exchanges traffic directly between them
    - But with NAT, have middleman that does some translation and manipulation of traffic being exchanged between src and dest process
  - ♥ NAT possibility must be taken into account by app designers

- E.g. P2P apps

∇ Addr shortage should instead be solved by IPv6

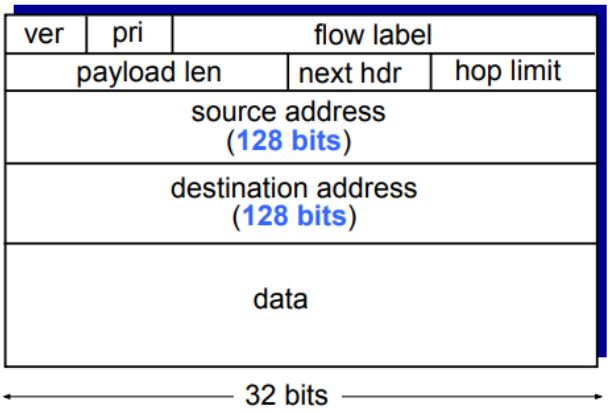
IPv6 Motivation

∇ Initial motivation: 32-bit addresses anticipated to be exhausted by 2008
  ◊ Blocks were exhausted between 2011-2015
  ◊ Though individual ISP have some; some recycling

∇ Additional motivation:
  ◊ When IPv4 designed, security was not considered enough
  ◊ Header format helps speed processing/forwarding
  ◊ Header changes to facilitate QoS – quality of service

∇ IPv6 datagram format
  ◊ Fixed-length 40 byte header – so routers can process quicker, without having to calculate as much
    ♥ IPv4 had dynamic size header – had that "options" component of variable length
  ◊ No fragmentation allowed – to send packet, sending device must check entire route and calculate what is the max transmission unit they can achieve on that path, then send the max
  ◊ 128-bit address space (3.4 x 1038 addresses)

IPV6 DATAGRAM FORMAT



∇ Row 1:
  ◊ Priority: identify priority among datagrams in flow
  ◊ Flow Label: identify datagrams in same "flow"
    ♥ In IPv4 – packets treated as individual, with no aspect of state in router (no consideration as to whether it's part of a particular flow)
    ♥ With Ipv6, have flow label – useful for enhancing QoS
      - E.g. Netflix has agreement with all ISPS – all packets originating from NF has flow label
        - So as packet comes through an ISP, it will be examined as seen as a packet that belongs to NF and gets treated with priority

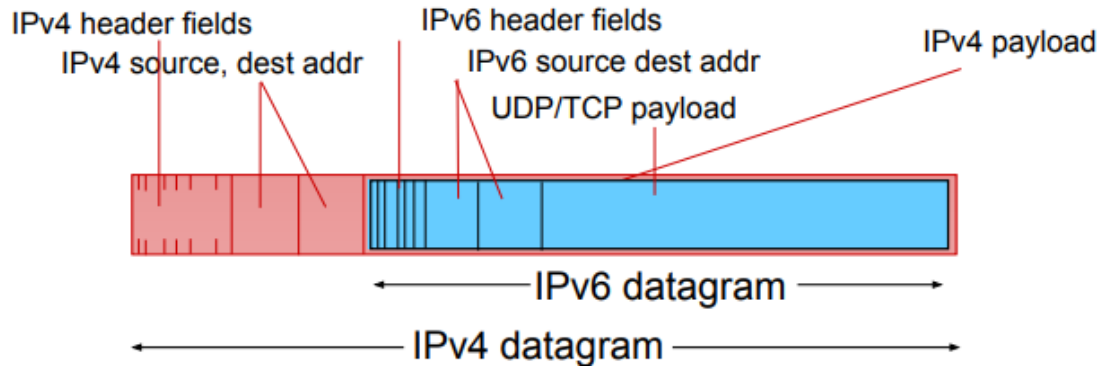
∇ Row 2:

- Next header (next hdr) - identify upper layer protocol for data
  - Header size is fixed, but there is another header that should be examined in conjunction with this one

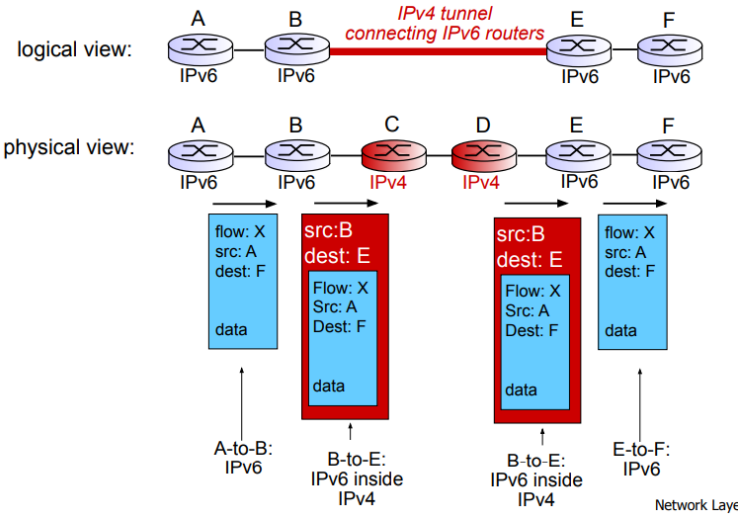
IPV6 OTHER CHANGES FROM IPV4

- Checksum: removed entirely to reduce processing time at each hop
- Options: allowed, but outside of header, indicated by “Next Header” field
- ICMPv6: new version of ICMP
  - Additional message types, e.g. “Packet Too Big”
  - Multicast group management functions

TRANSITION FROM IPV4 TO IPV6



- Not all routers can be upgraded simultaneously
  - How will network operate with mixed IPv4 and IPv6 routers?
- Tunnelling: IPv6 datagram carried as payload in IPv4 datagram among IPv4 routers
  - E.g. coming from Ipv4, coming through IPv6 nw and going to IPv4
    - Create tunnel in between such that the IPv4 gets carried in IPv6
  - Place into payload for corresponding alternate IP version packet
- In diagram above:
  - Coming form IPv6 nw, and going to tunnel through IPv4 nw
  - IPv6 nw gets encapsulated in IPv4 packet
- Additional example:



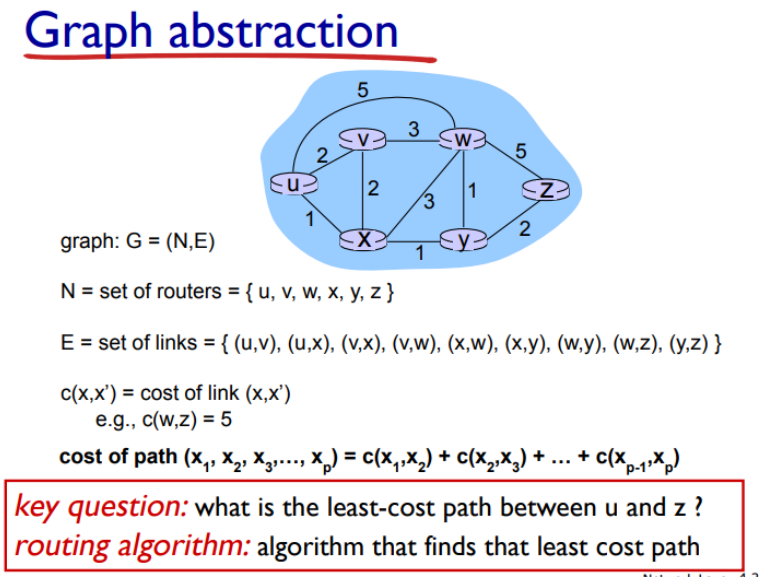
  - Have flow from A with dest F

- ◊ When packet flows to B, IPv6 packet gets encapsulated in IPv4 packet
- ◊ Packets moves from B to C to D
- ◊ When packet reaches E, gets encapsulated into original v6 packet

Routing Algorithms

- Have two planes at nw layer
  - (1) data plane – concerned with forwarding data packets from one router interface to the next
    - Forwarding table consulted
  - (2) control plane – where routing algos compute e2e paths and install necessary rules in each routers' forwarding table

Graphs



- Nw can be represented as graph with nodes (routers) and edges
- Link: physical connection between router and its adjacent neighbours
  - An "edge" in graph
- Within each link, have cost – many factors to decide what cost is
  - Generally, latency/delay when packets moving on that link; throughput i.e. link capacity (how much traffic can be pushed through; packet loss
  - Sometimes financial cost used e.g. when ISP uses another's' cable
- Routing algos try to compute e2e costs – summation of costs of links connecting 2 end points
  - Since there are many paths possible – try to compute shortest path between end points
  - Each router's control plane does this and installs the rules in the forwarding table for the appropriate routers
- Cost of x to x' is the same as x' to x – path cost same in both directions
- Cost of path is concatenation of links from src to dest
- Answer to key question – depends on whether considering cost of concatenating link paths or the # of hops

Routing Algorithms Classification

- Static or dynamic?
  - Static: routes change slowly over time
    - No change anticipated – run calculations to determine best path, then "leave it"
  - Dynamic: routes change more quickly
    - Periodic update
    - In response to link cost changes
- Global or decentralised info?
  - Global
    - All routers have complete topology, link cost info
    - "link state" algos
    - Each router tries to get info about every other router in nw, and how to reach them
  - Decentralised
    - Router knows physically connected neighbours, link costs to neighbours
    - Iterative process of computation, exchange of info with neighbours
    - "distance vector" algos

Link-State Routing Algo – Dijkstra

- Net topology, link costs known to all nodes
  - Accomplished via "link state broadcast"
  - All nodes have same info
- Computes least cost paths from one node (src) to all other nodes
  - Gives forwarding table for that nodes
- Iterative – after k iterations, know least-cost path to k destinations
- Takeway – for all processes in link state, each node periodically broadcasts to entire topology about its links
  - E.g. U broadcasts that is connected to X at cost 5, W at cost 3 etc

NOTATION

- C(x,y) – link cost from node x to y
  - = ∞ if not direct neighbours
- D(v) – current value of cost of path from src to dest vertx
- P(v) – predecessor node along path from src to dest v
- N' – set of nodes who least cost path definitively known

ALGO

Pseudocode

```
1  Initialization:
2   N' = {u}
3   for all nodes v
4     if v adjacent to u
5        then D(v) = c(u,v)
6     else D(v) = ∞
7
8  Loop
9    find w not in N' such that D(w) is a minimum
10   add w to N'
11   update D(v) for all v adjacent to w and not in N' :
12     D(v) = min( D(v), D(w) + c(w,v) )
13   /* new cost to v is either old cost to v or known
14    shortest path cost to w plus cost from w to v */
15 until all nodes in N'
```

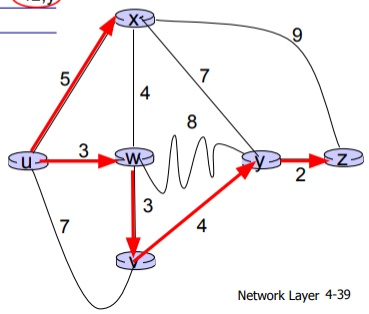
- ∇ Algo complexity O(n^2)
- ∇ Oscillations possible
  - ◊ E.g. support link cost equals amount of carried traffic

Diagram

Step	N'	D(v) p(v)	D(w) p(w)	D(x) p(x)	D(y) p(y)	D(z) p(z)
0	u	7,u	3,u	5,u	∞	∞
1	uw	6,w		5,u	11,w	∞
2	uwx	6,w			11,w	14,x
3	uwxv				10,v	14,x
4	uwxvy					12,y
5	uwxvyz					

notes:

- construct shortest path tree by tracing predecessor nodes
- ties can exist (can be broken arbitrarily)



Network Layer 4-39

- ∇ Process:
  - ◊ Start with src U – can get to Z via X, W, V
  - ◊ Step 0 – visit U
  - ◊ U to X = 5, U to W = 3, U to V = 7
  - ◊ Step 1 – visit W
    - ♥ W to X = 7 so keep current value 5
    - ♥ W to V = 6, 6 < 7 so becomes 6
    - ♥ W to Y = 11, since Y has no previous value, becomes 11
  - ◊ Step 2 – visit X
    - ♥ X to Y = 12, 11 < 12 so no change
    - ♥ X to Z = 14, Z has no value so becomes 14
  - ◊ Step 3 – visit V
    - ♥ V to Y = 10, 10 < 11 so becomes 10
  - ◊ Step 4 – visit Y
    - ♥ Y to Z = 12, 12 < 14 so becomes 12

- ◊ Final shortest path U to Z = UWVYZ

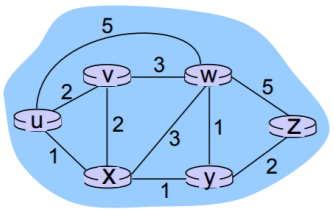
Distance Vector Algorithm

Bellman-Ford equation (dynamic programming)

let
$d_x(y)$:= cost of least-cost path from x to y
then
$d_x(y) = min_v \{c(x,v) + d_v(y)\}$

cost from neighbour v to destination y
cost to neighbour v
min taken over all neighbours v of x

- ∇ Given a host – look at all known neighbours
  - ◊ Each router knows distance to all its neighbours – neighbours regularly exchange reachability info through distance vector
  - ◊ Each neighbour informs its neighbour about all destinations that it knows of and the cost to those dests
- ∇ Dx(y) = estimate of least cost from x to y
  - ◊ x maintains distance vector Dx = [Dx(y):y ϵ N]
- ∇ Node x:
  - ◊ Knows cost to each neighbour v – c(x,v)
  - ◊ Maintains it neighbours distance vectors
  - ◊ For each neighbour v,x receives Dv = [Dvy): y ϵ N]
- ∇ Key idea:
  - ◊ From time-to-time, each node sends its own distance vector estimate to neighbours
    - ♥ For link-state, send knowledge about links to entire topology
  - ◊ When x receives new DV estimate from neighbour, it updates its own DV using B-F equation:
    - ♥ Dx(y) <- minv{C(x,v) + Dv(y)} for each node y ϵ N
  - ◊ Under minor, natural conditions, estimate Dx(y) converge to the actual least cost Dx(y)
- ∇ Cost to any remote node is sum of distance to neighbour + distance from neighbour to dest node
  - ◊ For each remote node, select neighbour that affords the least cost to a particular dest node – taking into consideration the nod's distance to that particular neighbour
  - ◊ Once above has been done for all neighbours that have informed the node of the particular dest – choose the min value
    - ♥ Share this info to all your neighbours
- ∇ Example



Q: What is the cost (u,z)?

$d_v(z) = 5$, $d_x(z) = 3$, $d_w(z) = 3$

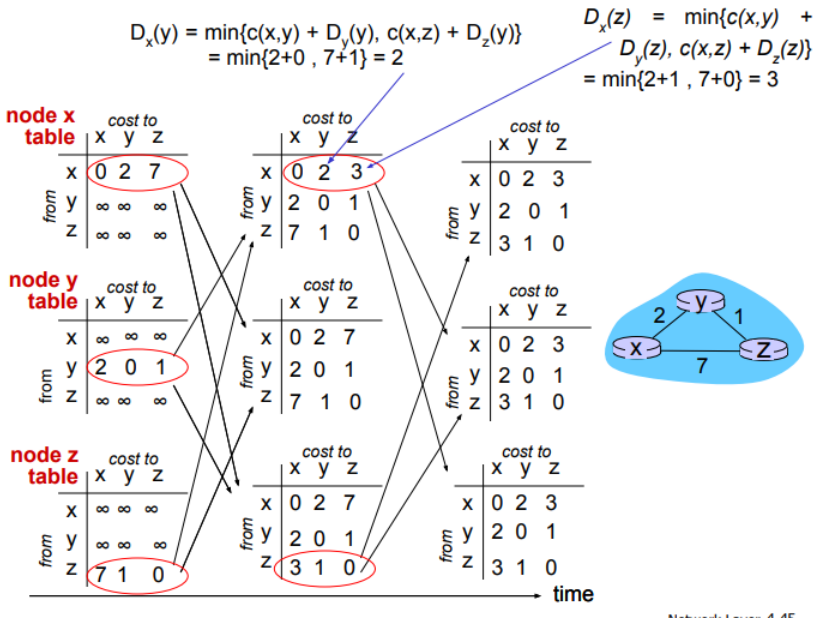
Filling in B-F equation:

$$d_u(z) = \min \{ c(u,v) + d_v(z), \; c(u,x) + d_x(z), \; c(u,w) + d_w(z) \}$$
$$= \min \{2 + 5, \; 1 + 3, \; 5 + 3\} = 4$$

node achieving minimum is next hop in shortest path, used in forwarding table

◊ Calculate cost from u to all its neighbours w,v,x and find min

- ♥ Dv(z) = 5=> implies v can reach z with distance = 5
- ♥ Min is 4 – u to x, then x to z



◊ Each vertex v knows the distance to each of its neighbours

- ♥ E.g. x to x =0; x to y = 2; x to z = 7

◊ At each iteration, get up date from neighbours

- ♥ E.g. x gets updates from y, and will re-compute distances to all its neighbours based on this new info
- ♥ Now, x to z = 3 (since y can reach z at distance of 1)

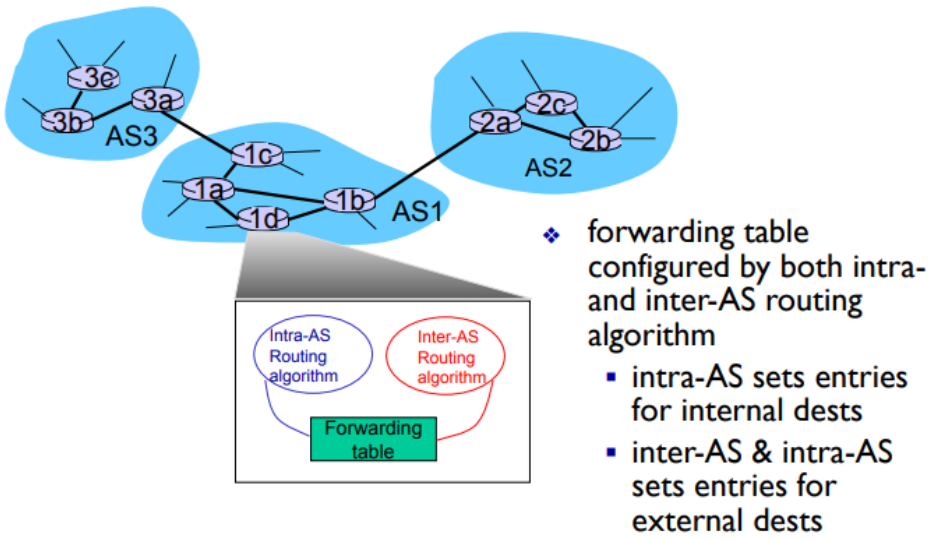
Hierarchical routing

∇ Routing study thus far – idealisation

- ◊ All routers identical – all routers equally participate in routing and forwarding process
- ◊ Nw flat
- ◊ ……not true in practice

- Scale – 600 million destinations
  - Can't store all dests in routing tables
  - Routing table exchange would swamp links
- Admin autonomy
  - Internet = nw of nws
    - Internet divided up into admin zones
    - Only responsible for routing for specific regions – have one gateway that allows it to reach other nws
    - Outside nws don't need to understand details of routing happening inside a particular AS
  - Each nw admin may want to control routing in its own nw
  - Want small # of routers that communicate at each lvl
  - E.g. ISP is an AS
- Aggregate routers into regions – "autonomous systems" (AS)
  - Have unique ID #
- Routers in same AS run same routing protocol
  - Intra-AS routing protocol
  - Routers in different AS can run different intra-as routing protocol
- Gateway router
  - At "edge" of its own AS
  - Has link to router in another AS
  - Have own routing algos to link gateways
  - Each AS has one or a few gateways – one or two gateways are visible to an external AS
    - Seen as one node to external nw, but AS can have multiple ASes

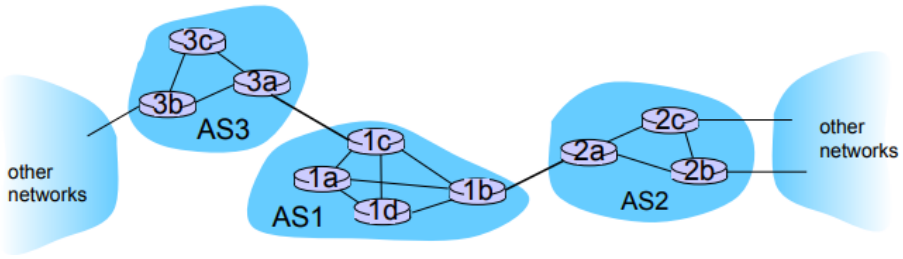
INTERCONNECTED ASES



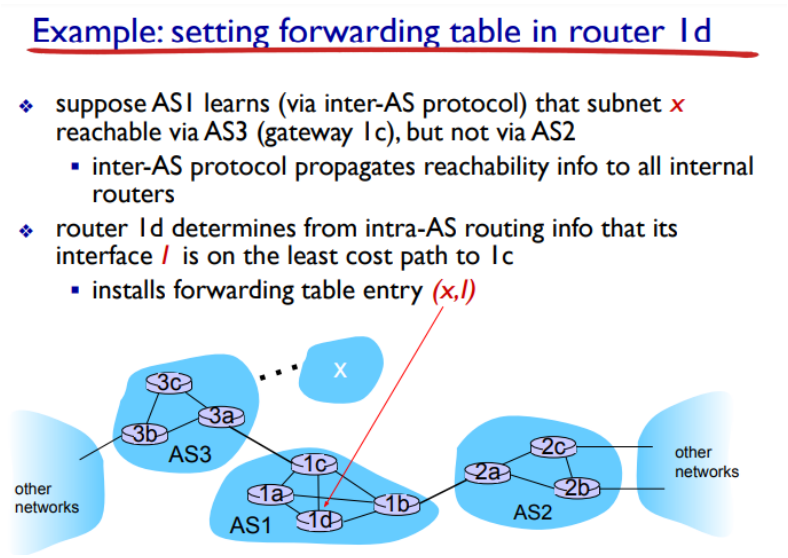
- Communication between different ASes
- Have ISP with different routers and ASes
- Have intra-AS routing algo that runs internally within AS – builds topology for routers, within scope of an AS
- Inter-AS routing algo that will allow gateway routers to send data between different ASes

- Forwarding table configured by both intra and inter-AS routing algo
  - Intra-AS sets entries for internal dests – routing within own domain
    - Need to be able to reach all dests within domain
  - Inter-AS and intra-AS sets entries for external dests – routers must know how to reach a gateway (and *which* gateway to use) in order to send data to external dests

Inter-AS tasks



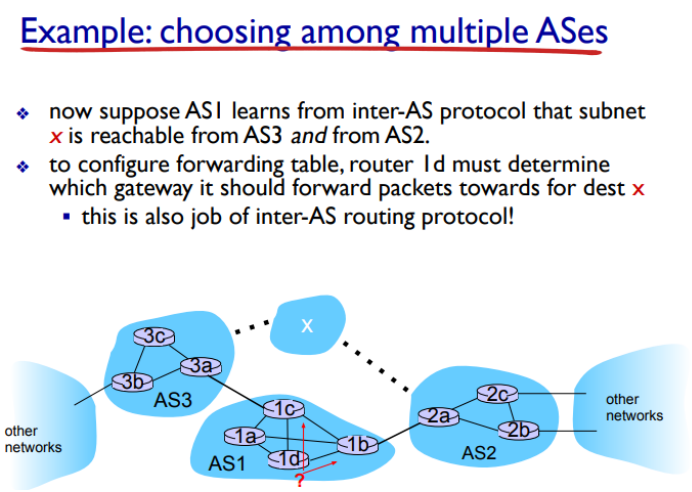
- Gateways are part of inter-AS infrastructure – used to connect nws (reach other nws)
  - Needed when routers need to forward packets to external nws
- Each nw must determine which dests are reachable via its different neighbours i.e. the nws it's connected to
  - BF algo – trying to see which neighbours can help reach remote dests
- Suppose router in AS1 receives datagram destined outside of AS1:
  - router should forward packet to gateway router, but which one?
- AS1 must:
  - Learn which dests (nws) are reachable through AS2, which through AS3
  - Propagate this reachability info to all routers in AS1
- Dealing with dests outside the AS – know where dests are
  - Get info through gateways and propagate info to internal routers
- Example:



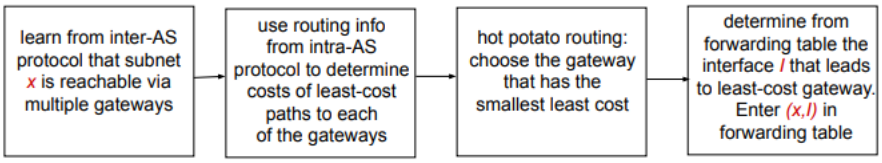
  - Router 1D needs to know which of its interfaces must be used to reach gateway that is associated with that dest
    - Inter-As protocol informs all routers 1a to 1d that x is reachable via gateway 1c

- ♥ Intra-As protocol informs all routers within AS1 how they can reach 1c
- ♥ Inter-routing algo installs forwarding table – to reach external address x, must use interface *L* to go to 1C

∇ Example 2:



- ◊ Have multiple ways of reaching external gateway
- ◊ Router needs to determine which gateway to use before it configures intra-routing
- ◊ Use inter-routing algo to determine shortest path for reaching dest – 1d must choose between 1c and 1b
  - ♥ Intra-routing algo determines which interfaces must be used to reach selected gateways
- ◊ Each nw wants to ensure that traffic moving within nw is moving as quickly
  - ♥ Internal nw wants to ensure that external traffic spends little time within it
    - Use as minimal resources as possible from traffic going outside
  - ♥ Hot-potato routing: send packet towards closest of two routers
    - Forward externally destined traffic to closest gateway
    - From 1d pov – which gateway is closest, so that can get it out of nw as quickly as possible



  - ♥ Try to find gateway that will get data out of nw as quickly as possible

INTRA-AS ROUTING

Intra-AS Routing

- also known as *interior gateway protocols (IGP)*
- most common intra-AS routing protocols:
  - RIP: Routing Information Protocol
  - OSPF: Open Shortest Path First
  - IGRP: Interior Gateway Routing Protocol (Cisco proprietary)

- ∇ Concerned with movement of traffic within AS
  - ◊ Also known as IGP – interior gateway protocol
- ∇ Rip – old, hardly used anymore
  - ◊ Distance vector approach based on hop count
  - ◊ Costs based on hop count i.e. how may hops from neighbours to dest
- ∇ OSPF – link-state algo implementation
  - ◊ More dynamic – costs based on link conditions
  - ◊ Follows link-state approach – involves broadcasting link-state info to all neighbours
    - ♥ Every router in topology responds to this – adjusts routing accordingly
    - ♥

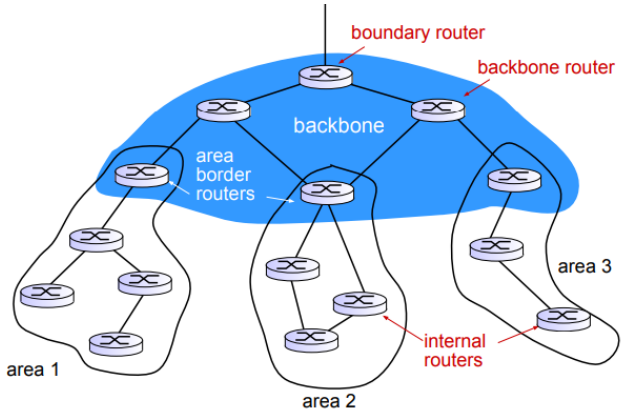
OSPF

OSPF (Open Shortest Path First)

- "open": publicly available
- uses link state algorithm
  - LS packet dissemination
  - topology map at each node
  - route computation using Dijkstra's algorithm
- OSPF advertisement carries one entry per neighbour
- advertisements flooded to *entire* AS
  - carried in OSPF messages directly over IP (rather than TCP or UDP

- ∇ Advertise to within specific AS or specific part of AS
- ∇ Broadcast info about direct links
  - ◊ Carries one link-state entry per neighbour
- ∇ Router with n neighbours – OSPF ad will have n entries
- ∇ Each node in nw has info about entire topology through those ads
  - ◊ Ads flooded to whole AS
- ∇ Ads done at nw layer using IP – no transport layer involved (since not host-to host)
- ∇ Use Dijkstra to compute shortest path from each router to every other in nw

Hierarchical OSPF

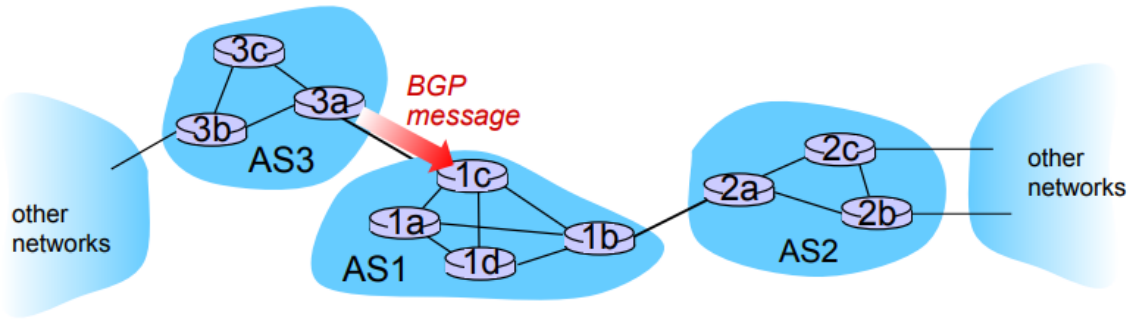


- ∇ Not all routers perform above process – use hierarchical OSPF
  - ◊ # of link-state ads could overwhelm nw if allowed all routers to broadcast to every other in nw
  - ◊ Thus, need to break large nws into smaller routing zones
- ∇ Zones could be based on geographical area (often the case), some virtual criteria (e.g. categories of users), etc
  - ◊ Within each zone – have border router which is used to connect with other zones
- ∇ As will have set of more powerful routers that interconnect all ASes areas through area border routers
  - ◊ Each zone/area will have border router connected to nw's backbone
  - ◊ Area border routers are part of backbone – allows traffic to be moved from one area to another through backbone
- ∇ ISP gets broken down into backbone - the routers that form the main part of ISP, interconnect different parts of ISP
- ∇ Within backbone
  - ◊ Boundary router – allow connection of one AS to external nw
  - ◊ Area border routers responsible for disseminating OSPF info to specific area
- ∇ Two-level hierarchy: local area, backbone
  - ◊ Facilitates scalability by ensuring that ads limited to smaller areas within hierarchy
    - ♥ E.g. link-state ad in area 1 will only be broadcast within area 1
  - ◊ Link-state advertisements only in area
    - ♥ Don't want to broadcast to entire topology – since how routers link to each other e.g. in area 1 may not necessarily affect how the other areas are linked together, as long as you can get to the borders
    - ♥ Info about backbone links only shared to backbone routers
  - ◊ Each node has detailed area topology; only know direction (shortest path) to nws in other areas
- ∇ Area border routers: "summarize" distances to nets in own area, advertise to other Area Border routers
- ∇ Backbone routers: run OSPF routing limited to backbone
- ∇ Boundary routers: connect to other AS's
- ∇ Each area may run different routing
  - ◊ E.g. run OSPF with backbone routers; within area 1 could run different routing protocol

Internet inter-AS routing: BGP

- BGP (Border Gateway Protocol): the de facto inter-domain routing protocol
  - Concerned with connectivity between different ASes
  - Enables internet to operate at scale
  - "glue that holds the Internet together"
  - Any nw connecting with another on the internet likely runs this protocol
- BGP provides each AS a means to:
  - eBGP – obtain external subnet reachability information from neighboring Ases
    - Boarder gateway gets info from other domains
      - "What can we reach from using your nw?"
    - External-facing component
    - "What can we reach via our neighbour(s)?"
    - Each eBGp router chooses routes to other nws based on received reachability info and policy
      - Helps ASes to advertise its IP address ranges (subnets) to the outside world
  - iBGP – propagate reachability information to all AS-internal routers
    - Boarder gateway learns about how to reach entire internet – internal BGP takes it, summarises it and sends it across to internal nw

BGP BASICS



- BGP session: two BGP routers ("peers") exchange BGP msgs
  - Virtual establishment of link between two BGP routers – two routers in two different nws
    - Exchange BGP msgs – prefix announcements or link updates
    - Follows distance vector routing approach – msg contains all nw paths that each neighbour knows to all remote dests and informs neighbours about those dest, and the neighbour's distance to those dests
  - Advertising paths to different destination nw prefixes ("path vector protocol")
    - AS wishes to tell rest of internet that particular prefix is reachable via that AS
    - E.g. as an ISP have set of IP addresses that you've allocated to your customers – want to announce to internet that those IP addresses belong to you and your customers
      - Traffic destined for those IP prefixes must be sent to you – so it can be delivered to your customers

- ♥ AS ad is way of AS saying that it promises to forward packets destined for a particular prefix
- ◊ Exchanged over semi-permanent TCP connections
  - ♥ Establish session in the pairs – permanent but not actively always on
  - ♥ Listening on both sides – if one needs to begin transmit msgs, then send signal on that session and msgs will be exchanged
  - ♥ Always on but not always active
- ◊ Foundation of BGP

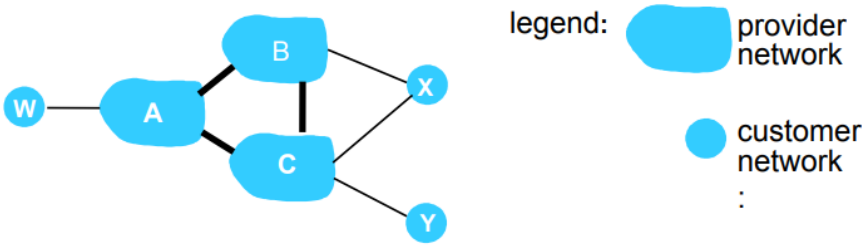
∇ Diagram:

- ◊ When AS3 advertises prefix (a.b.c/) to AS1
  - ♥ AS3 promises it'll forward datagrams towards that prefix
    - i.e. able to receive msgs destined for those IP addresses and deliver them to correct dest host
  - ♥ AS3 can aggregate prefixes in its ad
    - BGP router announces what IP addresses are within its nw or within nws of its customers
    - Ad goes out only to neighbours – neighbours propagate that info to its neighbours and so on
  - ♥ BGP ad may be an aggregate of several prefixes that have been received by that AS

∇ Distributing path info

- ◊ AS3 sends prefix reachability info to AS1 using eBGP session between 3a and 1c
  - ♥ AS3 ests BGP session and makes announcement to AS1
  - ♥ 1c can then use iBGP to distribute new prefix info to all routers in AS1
    - Propogate info received from outside to all routers within AS1
  - ♥ 1b can then re-advertise new reachability info to AS2 over 1b-to-2a eBGP session
    - Info gets propagated to AS2
    - Whether 1b re-advertises info to AS2 depends on business relationship between AS1 and AS2, and between AS1 and AS3, and AS1s policies
    - E.g. AS1 is provider for AS3 – AS1 then obliged to carry and receive all traffic that belongs to AS3
      - ○ AS1 tells AS2 – if have traffic for AS3, send to AS1
- ◊ When router learns of new prefix, it creates entry for prefix in its forwarding table

ROUTING POLICY



∇ Whether an AS re-advertises a learnt prefix depends on business relationship

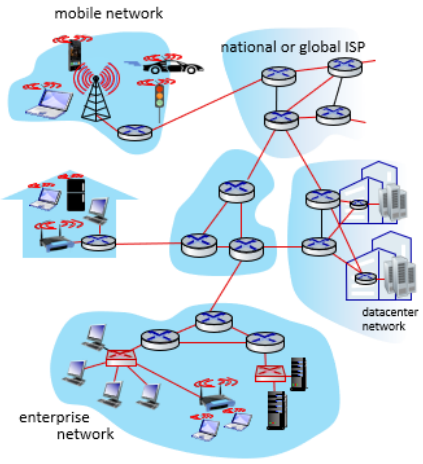
∇ A,B,C are provider networks

- X,W,Y are customer (of provider networks)
- Scenario 1 (diagram):
  - X is dual-homed: attached to two networks (customer of both B and C)
    - X does not want to route from B via X to C
      - i.e. not carry B's traffic to C via itself – if X received prefix info from B
        - if X announces reachability info to C – X could begin to carry traffic meant for C coming from B via itself
      - Whether X advertises reachability info to C and B depends on position of X towards B and C – in this case, X is customer of B, so won't make any announcements about the prefixes it gets from B to C
    - .. so X will not advertise to B a route to C
- Scenario 2 (diagram):
  - A advertises path AW to B
    - Essentially telling B, A will accept and forward traffic to W
  - B advertises path BAW to X – acceptable since X is customer of B
  - Should B advertise path BAW to C?
    - No way! B gets no "revenue" for routing CBAW since neither W nor C are B's customers
    - B wants to force C to route to w via A
    - B wants to route only to/from its customer

DIFFERENT INTRA-, INTER-AS ROUTING

- Policy:
  - inter-AS: admin wants control over how its traffic routed, who routes through its net
  - intra-AS: single admin, so no policy decisions needed
- Scale:
  - hierarchical routing saves table size, reduced update traffic
- Performance:
  - intra-AS: can focus on performance
    - i.e. traffic moves smoothly as possible through nw and external traffic doesn't take up too much internal resources
    - optimal usage of internal resources
  - inter-AS: policy may dominate over performance
    - i.e. what nws give cheaper, cost-effective transmission etc

Link Layer



- Hosts and routers: nodes
- Links: communication channels that connect adjacent nodes along communication path
  - Wired links
  - Wireless links
  - LANs
- Frame: layer-2 packet, encapsulates datagram
  - Datagram comes from nw layer to link layer – gets encapsulated into frame
- Data-link layer has responsibility of transferring datagram from one node to physically adjacent node over a link
  - Concerned with hop-to-hop transfer of data along path to dest

MAC Addresses

- Medium Access Control: controls hw responsible for interaction with medium e.g. wired medium, optical fibre etc
  - Responsible for managing interaction between hw and medium
  - When sending data to another device on the nw, MAC protocol encapsulates higher lvl data into frames appropriate for particular transmission medium
    - Encapsulation for e.g. wireless link differs to that of fibre optic
  - i.e. manging the hw and medium that's there + putting data into correct format
- MAC (or LAN or physical or Ethernet) address:
  - Unique identifier assigned to nw interface controller (NIC card) on device
  - Used for communication within nw segment
  - Commonly used in IEE 802 technologies – ethernet, Wi-Fi, Bluetooth etc (all have MAC address)
  - Used "locally" to get frame from one interface to another physically-connected interface (same network, in IP-addressing sense)
  - 48-bit MAC address coded into NIC ROM, also sometimes software settable
    - Represented in hexadecimal notation (base 16)
    - e.g.: 1A-2F-BB-76-09-AD

- ♥ Used to identify each physical interface in nw – so that can pass msgs to devices within LAN

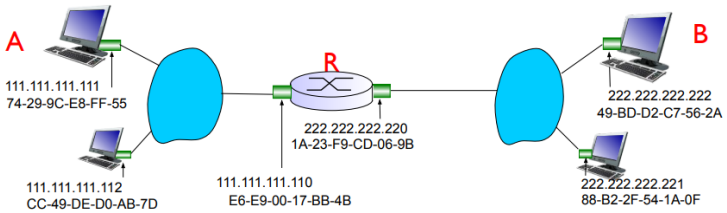
Address Resolution Protocol

- ∇ Protocol used for discovering MAC addresses
- ∇ When need to associate given nw address, such as IP, use this protocol to find MAC address i.e. physical interface associated wih that IP address
- ∇ Datagram comes from nw layer with src and dest IP - datagram pushed down to link layer
- ∇ Link layer must move data to the next node
  - ◊ How to determine interface's MAC address, knowing its IP address? ARP!
- ∇ ARP table: each IP node (host, router) on LAN has table
  - ◊ IP/MAC address mappings for some LAN nodes: < IP address, MAC address, TTL>
    - ♥ IP is logical address assigned at nw layer
    - ♥ MAC address is physical address associated with an interface
      - Hardwired into the device interface (hw)
  - ◊ TTL (Time To Live): time after which address mapping will be forgotten (typically 20 min)
    - ♥ How long assignment remains valid

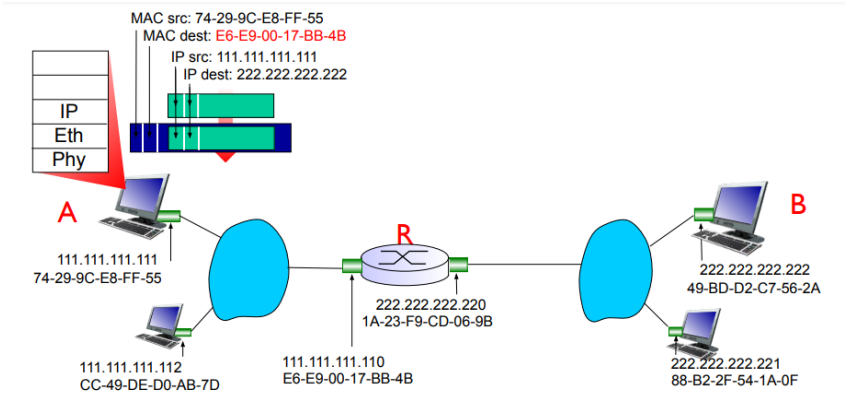
ARP PROTOCOL: SAME LAN

- ∇ A wants to send datagram to B (in same nw)
  - ◊ B's MAC address not in A's ARP table.
- ∇ A broadcasts **ARP query packet**, containing B's IP address
  - ◊ Looking for interface for IP address of B
  - ◊ Destination MAC address = FF-FF-FF-FF-FF-FF
    - ♥ Highest # that can be assigned to MAC addresses
  - ◊ All nodes on LAN receive ARP query
- ∇ B receives ARP packet, replies to A with its (B's) MAC address
  - ◊ Frame sent to A's MAC address (unicast)
  - ◊ Request came with MAC address, so able to respond to it directly
- ∇ A caches (saves) IP-to-MAC address pair in its ARP table until information becomes old (times out)
  - ◊ Soft state: information that times out (goes away) unless refreshed
- ∇ ARP is "plug-and-play":
  - ◊ Nodes create their ARP tables without intervention from net administrator

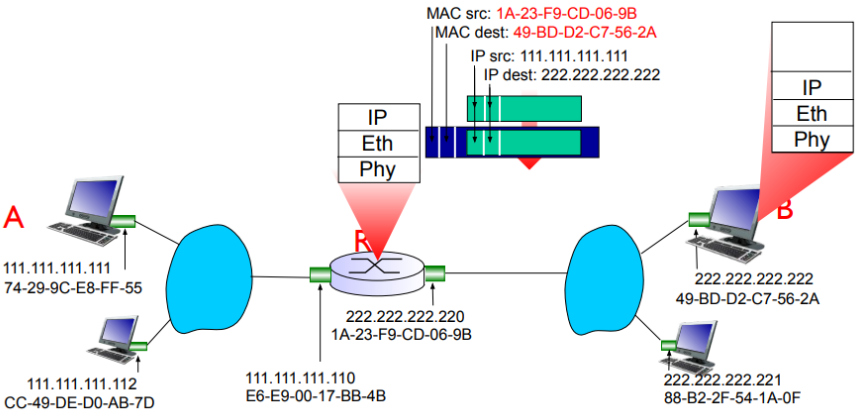
Addressing: Routing to Another LAN



- ∇ Sending msg to another LAN i.e. not within same nw
- ∇ Walkthrough: send datagram from A to B via R
  - ◊ Focus on addressing – at IP (datagram) and MAC layer (frame)
  - ◊ Assume A knows B's IP address
  - ◊ Assume A knows IP address of first hop router, R (how?)
    - ♥ IP of router is default gateway of A -in forwarding table, if dest is not local, have to send through gateway router R
  - ◊ Assume A knows R's MAC address (how?)
    - ♥ Same would have done ARP broadcast to find MAC address of R – R would have send response indicating its MAC address



  - ◊ A creates IP datagram with IP source A, destination B
  - ◊ A creates link-layer frame with R's MAC address as destination, frame contains A-to-B IP datagram
    - ♥ Src MAC address is A
    - ♥ Dest MAC address is R – only concerned with moving from host to next link



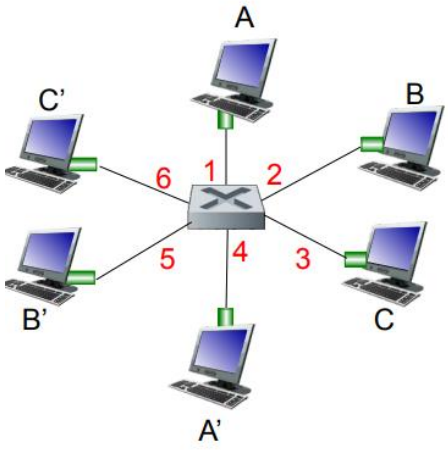
  - ◊ R forwards datagram with IP source A, destination B
    - ♥ R does routing and forwarding process – removes link-layer frame and looks at nw layer datagram
    - ♥ Sees destination is host B with a particular IP address – examines forwarding table to see which interface to send datagram to
    - ♥ Sees dest B is on same nw as itself (R) – forwards it as link-layer frame
  - ◊ R creates link-layer frame with B's MAC address as dest, frame contains A-to-B IP datagram (encapsulated)

        - ♥ IP datagram doesn't change – same src and dest IP
        - ♥ Src MAC address is R's
        - ♥ Dest MAC address is B's
    - ◊ Datagram processed and taken up to nw layer
    - ◊ Key difference (compared to arp protocol in same LAN)
        - ♥ If nodes within same nw – exchange msgs directly
            - Do ARP broadcast – find MAC address of host exchanging msg with
            - Create frame and send
        - ♥ If nodes not within same nw – msg moves hop by hop
            - Between src, router and dest
            - Often several routers between src and dest
            - Use arp lookups procedure until msg reaches dest address

Ethernet Switch

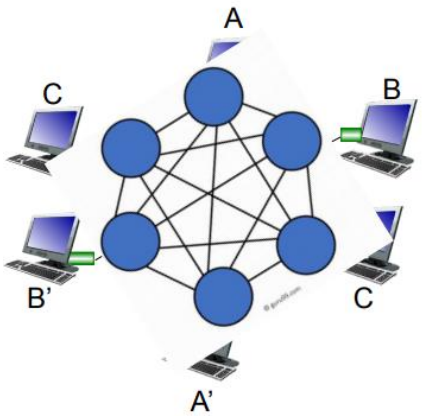
- ∇ Link-layer device: takes an active role
    - ◊ Store, forward Ethernet frames
    - ◊ In LAN, if one device needs to send frames to another, but these devices are setup such that they can't reach each directly – connected through ethernet switch
        - ♥ Switch receives frame from one device, store them – ensure get sufficient frames to able to decide where frames are going
        - ♥ Then forward to device meant to receive them
    - ◊ Active role – examining frames and deciding where they're going, then forwarding to that device
    - ◊ Examine incoming frame's MAC address, selectively forward frame to one-or-more outgoing links when frame is to be forwarded on segment
- ∇ Transparent
    - ◊ Hosts are unaware of presence of switches
    - ◊ As far as hosts are concerned – it appears to them as though they're connected directly
- ∇ Plug-and-play, self-learning
    - ◊ Switches do not need to be configured

MULTIPLE SIMULTANEOUS TRANSMISSIONS



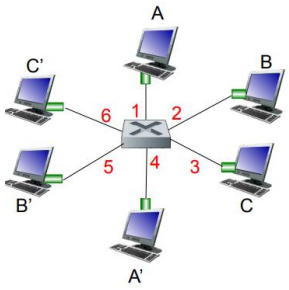
switch with six interfaces
(1,2,3,4,5,6)

- ∇ Switch allows nw to have multiple simultaneous transmission between different hosts connected to the switch
- ∇ Hosts have dedicated, direct connection to switch
  - ◊ Can be wired or wireless
- ∇ Switches buffer packets – receive frames from each host and then decide where to forward packets to
- ∇ Ethernet protocol used on each incoming link
- ∇ Switching: A-to-A' and B-to-B' can transmit simultaneously, without collisions
  - ◊ Each link is its own collision domain
  - ◊ Collision, if any, can only happen within single link – between host and switch
- ∇ Devices in nw see themselves as having direct connection to all other hosts in nw
  - ◊ Switch is "transparent" to all hosts



switch with six interfaces
(1,2,3,4,5,6)

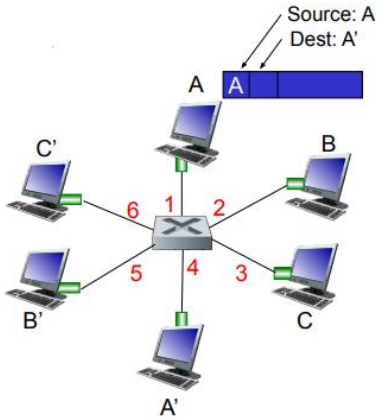
SWITCH FORWARDING TABLE



switch with six interfaces (1,2,3,4,5,6)

- Q: how does switch know A' reachable via interface 4, and B' reachable via interface 5?
- A: each switch has a switch table, each entry:
  - (MAC address of host, interface to reach host, time stamp)
  - Looks like a router forwarding table!
  - Switching table indicates that interface for A' is on interface # 4

SELF-LEARNING



- Switch learns which hosts can be reached through which interfaces
  - When frame received, switch "learns" location of sender – by examining incoming LAN segment
  - Records sender/location pair in switch table
  - Uses responses to frames to continue building table

FRAME FILTERING/FORWARDING

- when frame received at switch:
  - 1. record incoming link, MAC address of sending host
  - 2. index switch table using MAC destination address
  - 3. if entry found for destination then

    {

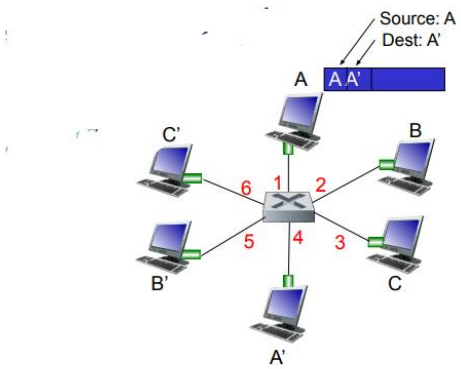
    if destination is on segment from which frame arrived

    then drop frame //filtering – won't forward things that are going to the interface that they came from

    else forward frame on interface indicated by entry

}
//didn't find entry based on MAC address
else flood /* forward on all interfaces except arriving interface */

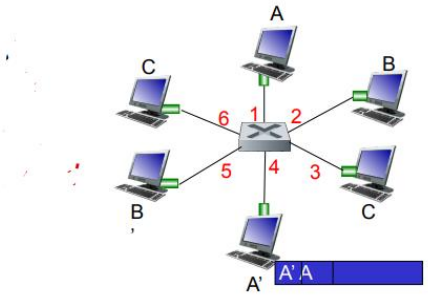
Example: Self-learning, forwarding:



MAC addr	interface	TTL

switch table (initially empty)

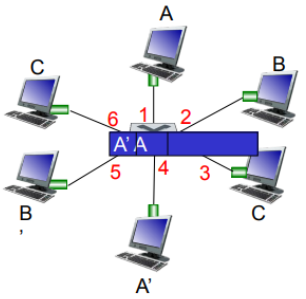
- Frame src – A
  - Frame destination, A',
  - Switch table empty – have no records yet
- Switch receives frame from A for A' – records As MAC address and interface in table
  - Check in switching table for MAC address for A'
  - Location unknown: flood (send to all outgoing links)



MAC addr	interface	TTL
A	*1*	*60*

switch table (initially empty)

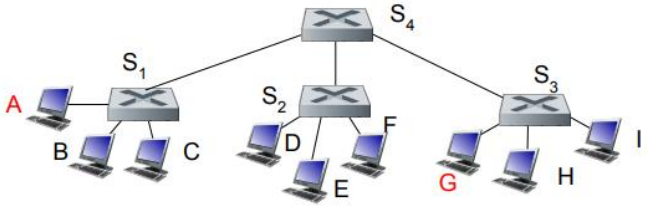
- All hosts receive frame
  - All others besides A' will drop that frame
- A' sends packet to switch
  - Switch records src interface and MAC address of A'
- Send packet from A' to A
  - Switch checks table – sees forwarding info for A
  - Forwards frame to interface 1 for A



MAC addr	interface	TTL
A	*1*	*60*
A'	*4*	*60*

switch table (initially empty)

INTERCONNECTING SWITCHES

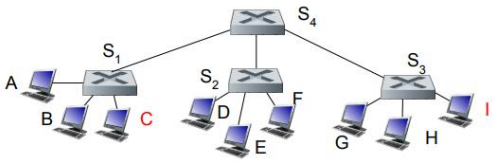


- ∇ Switches can be connected together
- ∇ Q: sending from A to G - how does S1 know to forward frame destined to G via S4 and S3?
- ∇ A: self-learning! (works exactly the same as in single-switch case!)
  - ◊ A sends frame to S1 meant for G – but S1 doesn't know where G is
    - ♥ Frame forwarded to all outgoing links – B,C and S4
  - ◊ S4 receives frame – G not in switching table
    - ♥ Frame forwarded to all outgoing links – S2 and S3
  - ◊ S2 receives – forwards to D, E, F
  - ◊ S3 receives – forwards to G, H, I
- ∇ When connect switches together => increase broadcast domain
  - ◊ E.g. have empty switch table for S1, send msg from A to C – C is unknown, so msg sent to all outgoing links
    - ♥ Msg sent to every part of nw

Practice Question

Self-learning multi-switch example

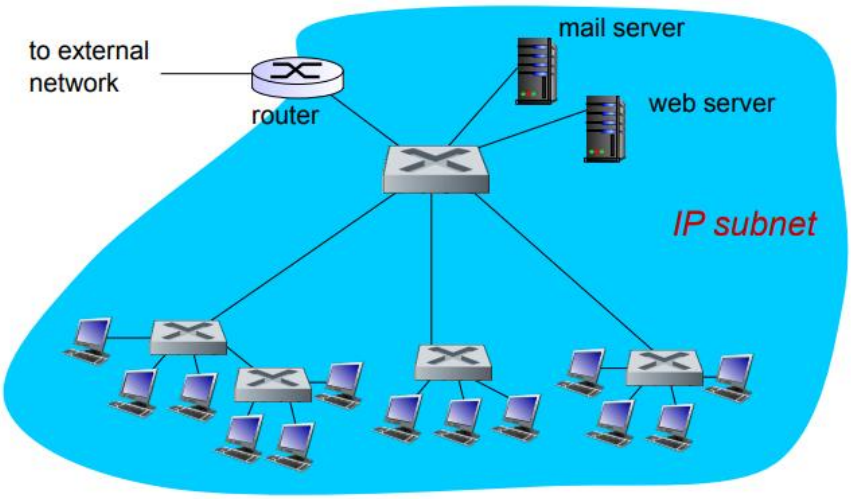
Suppose C sends frame to I, I responds to C



- Q: show switch tables and packet forwarding in S_1, S_2, S_3, S_4

  ∇ Build switching tables for all switches

Institutional Network

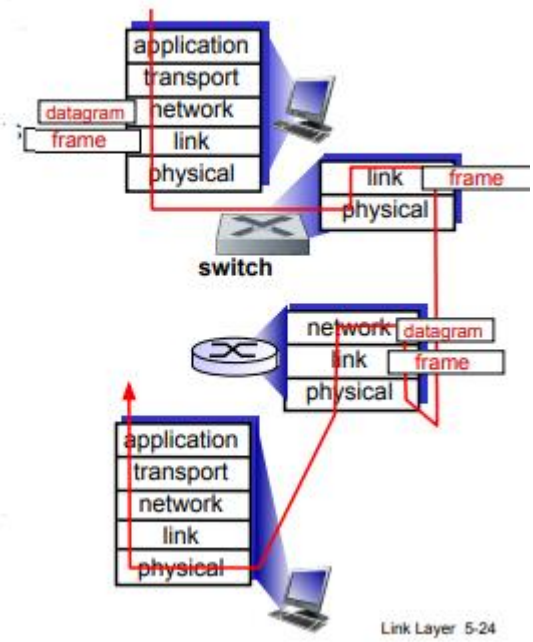


∇ E.g. have campus nw where each department has switch connecting all hosts in that dept

◊ Some depts will have multiple switches – one of those connect to the big switch for the dept (which connects to the biggest switch)

∇ NB: can't connect to external internet using switches – need router

Switches vs. Routers



- Both are store-and-forward:
  - Routers: network-layer devices (examine network-layer headers)
  - Switches: link-layer devices (examine link-layer headers)
- Both have forwarding tables:
  - Routers: compute tables using routing algorithms, IP addresses
  - Switches: learn forwarding table using flooding, learning, MAC addresses

TROUBLE WITH INTERCONNECTING SWITCHES

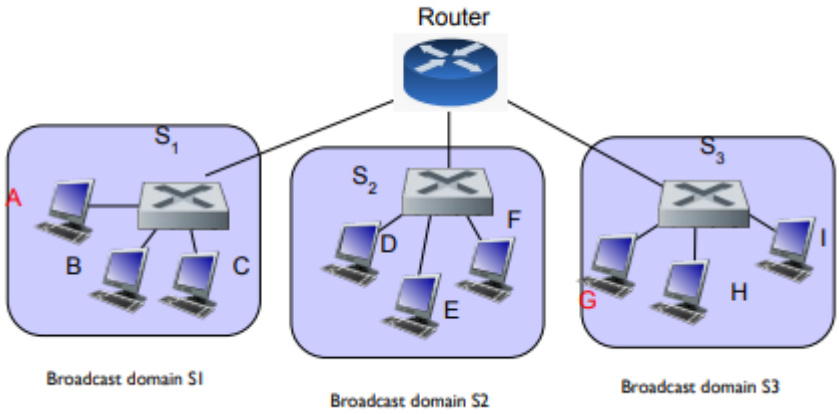
Q: what happens if S4 is replaced by a router?

A: broadcast domain confined to S1, S2, S3, (3 limited broadcast domains)

- Broadcast Domain: when a device sends out a broadcast message, all the devices present in its broadcast domain have to pay attention to it
- Broadcast domains grow when switches are connected together – this takes up more bandwidth, processing power of switches, etc
  - Thus, want to keep broadcast domains as small as possible, so that few/n o messages have to cross different nw segments
- Example (above) – router won't forward layer 2 broadcast msgs coming from s1, s2 or s3

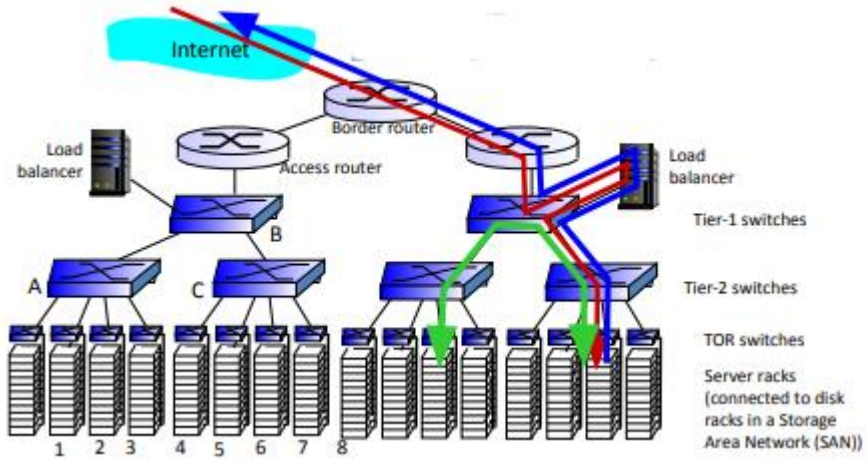
- Router only fwds if destination IP address is on the outgoing link
- Router thus just drops the messages if any layer 2 broacast msgs arrive

BROADCASTS AND COLLISIONS



- If A is trying to reach G, the message will be broadcast to router and other nodes within S1– but it won't forward that msg (since router fwds at layer 3)
- Router will only fwd a msg if it is a particular msg intended for G
  - Router must have in its forwarding tbl, an outgoing link for destination G
- S1, S2 and S3 are essentially separate nws – thus broadcast msgs will be confined to each of these smaller networks
  - Will only fwd these msgs at the IP layer if they are intended for these nws
- Minimise nw flooding – don't build nw just on switches

DATA CENTRE NETWORKS



- Have large number of end hosts connected together by switches – end hosts are usually servers that are accessed when accessing content hosted by these data centres
- Load balancer: 'application-layer routing'
  - Receives external client requests
  - Directs workload within data centre
  - Returns results to external client (hiding data centre internals from client)