Demultiplexing: Host recieves IP datagrams

→ each datagram has source IP address, destination IP address

→ each datagram carries one transport-layer segment

→ each segment has source, destination port number

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

- TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- demux: receiver uses *all four values (4-tuple)* to direct segment to appropriate socket

- server may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple
  - each socket associated with a different connecting client

when receiving host receives *UDP* segment:

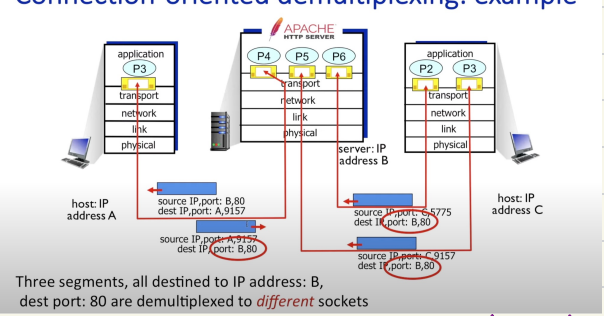
- checks destination port # in segment
- directs UDP segment to socket with that port #

IP/UDP datagrams with *same dest. port #,* but different source IP addresses and/or source port numbers will be directed to *same socket* at receiving host

→ TCP multiplexing

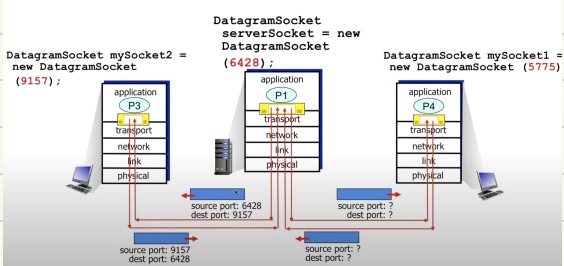
→ UDP multiplexing

Connection-oriented demultiplexing: example



Three segments, all destined to IP address: B, dest port: 80 are demultiplexed to *different* sockets

Connectionless demultiplexing: an example



→ demultiplexing using 4-tuple source, destination IP, port numbers

→ demultiplexing based on destination port number ONLY

* multiplexing & demultiplexing happens at every layer

UDP: User Datagram Protocol

→ Connection-less: no handshaking between users

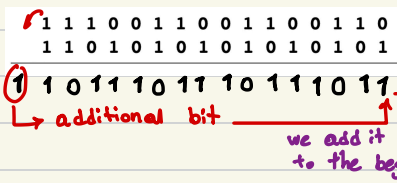
→ each UDP segment handled independently of others

- UDP use:
  - streaming multimedia apps (loss tolerant, rate sensitive)
  - DNS
  - SNMP
  - HTTP/3
- if reliable transfer needed over UDP (e.g., HTTP/3):
  - add needed reliability at application layer
  - acc congestion control at application layer

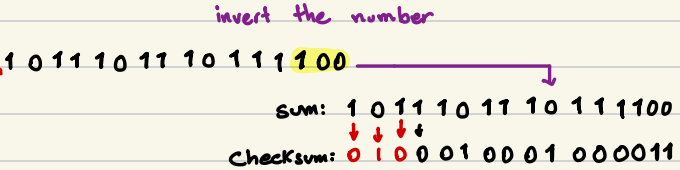
Why is there a UDP?

- no connection establishment (which can add RTT delay)
- simple: no connection state at sender, receiver
- small header size
- no congestion control
  - UDP can blast away as fast as desired!
  - can function in the face of congestion

when sending packets, UDP adds a checksum for error detection: Add these two numbers:

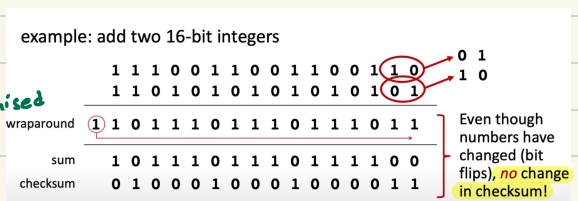


to convert the sum to checksum, we invert the number

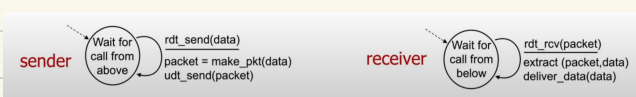


UDP → segments may be lost, delivered out of order, no handshake (no RTT needed) can function when network service is compromised

HTTP/3 built on top of UDP



Reliable data transfer protocol (rdt): It is built on top of an unreliable channel which means data might get lost or corrupted → keep track of them on the reliable channel and get/send confirmations



Problems: bit flips → check sum to find errors. How to recover from errors? using ACK (Acknowledgements) and NAK to inform the sender (Negative ACKs) → Sender retransmits packet (NAKs)

What if ACK, NAK is corrupted? we cannot wait for ACK/NAK before sending the next packet. Also, the sender can't just retransmit the data (possible duplicate)

Add secuence number → handles duplicates, receiver discard duplicates

how to detect duplicates?

→ we don't really need NAKs. Just check if we recieve ACKs.

→ Add wait time for receiving the next ACK after that, we will resend

using secuence numbers, we can check if the ACK is not for the expected sequence, it will resend it

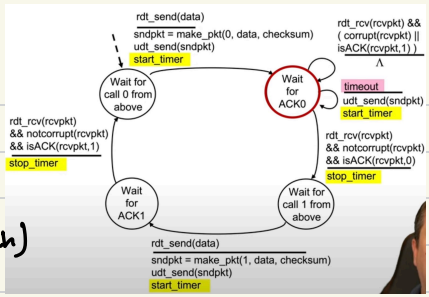
TCP has Cumulative ACKs and Pipelining

↳ can send a window of messages and then wait

Helps with implementing congestion control

sender will not overwhelm the receiver (speed-match)

* The ACK number is one more than the Seq number



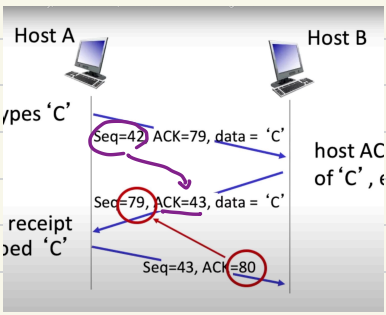
TCP round trip time, timeout

Q: how to set TCP timeout value?

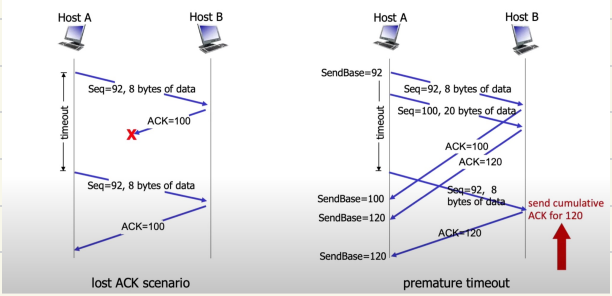
- longer than RTT, but RTT varies!
- *too short:* premature timeout, unnecessary retransmissions
- *too long:* slow reaction to segment loss

Q: how to estimate RTT?

- `SampleRTT`: measured time from segment transmission until ACK receipt
  - ignore retransmissions
- `SampleRTT` will vary, want estimated RTT "smoother"
  - average several *recent* measurements, not just current `SampleRTT`



TCP: retransmission scenarios



Congestion Control: Too many sources sending too much data for network to handle ⟶ long delays (big queue in router)

↳ packet loss (buffer overflow)

In reality, we send duplicates which will get removed

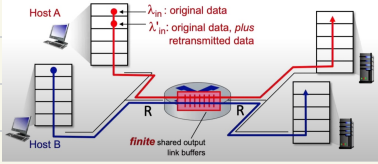
duplicates: $\lambda_{in}' \uparrow$ $\lambda_{out} \downarrow$

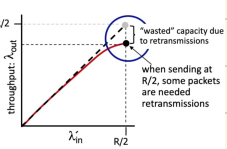
when we retransmit a packet, λ_{in}' increase while λ_{out} ↑ throughput ↓

Causes/costs of congestion: scenario 2

Idealization: *some* perfect knowledge

- packets can be lost (dropped at router) due to full buffers
- sender knows when packet has been dropped: only resends if packet *known* to be lost

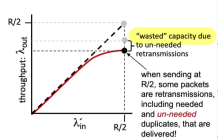




Causes/costs of congestion: scenario 2

Realistic scenario: *un-needed duplicates*

- packets can be lost, dropped at router due to full buffers – requiring retransmissions
- but sender times can time out prematurely, sending *two* copies, *both* of which are delivered



"costs" of congestion:

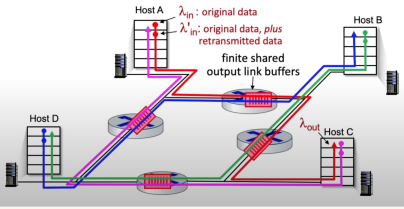
- more work (retransmission) for given receiver throughput
- unneeded retransmissions: link carries multiple copies of a packet
  - decreasing maximum achievable throughput

Causes/costs of congestion: scenario 3

- *four* senders
- *multi-hop* paths
- timeout/retransmit

Q: what happens as λ_{in} and λ_{in}' increase ?

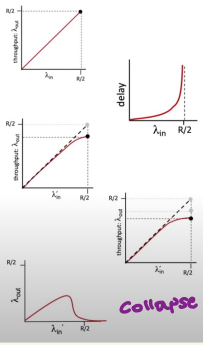
A: as red λ_{in}' increases, all arriving blue pkts at upper queue are dropped, blue throughput → 0



→ when red increase, it makes pkt loss for others happen more. → λ'_{in} for other senders increase because of this → congestion collapse

Causes/costs of congestion: insights

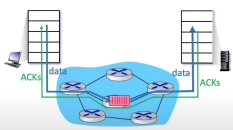
- throughput can never exceed capacity
- delay increases as capacity approached
- loss/retransmission decreases effective throughput
- un-needed duplicates further decreases effective throughput
- upstream transmission capacity / buffering wasted for packets lost downstream



Thus, we need congestion control for tcp. There are two approaches

End-end congestion control:

- no explicit feedback from network
- congestion *inferred* from observed loss, delay



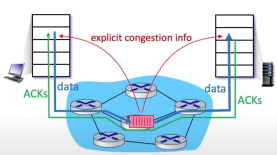
The sender tries to see the amount of lost packets and infer congestion

original approach by Tcp

Approaches towards congestion control

Network-assisted congestion control:

- routers provide *direct* feedback to sending/receiving hosts with flows passing through congested router
- may indicate congestion level or explicitly set sending rate
- TCP ECN, ATM, DECbit protocols



→ in this method, the network layer informs the hosts about congestion before loss

→ new versions of TCP

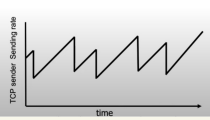
Tcp congestion control: We want to increase incrementaly to avoid losses again decrease multiplicated → we don't want to use losses to find the spot

TCP congestion control: AIMD

- *approach:* senders can increase sending rate until packet loss (congestion) occurs, then decrease sending rate on loss event

Additive Increase
increase sending rate by 1 maximum segment size every RTT until loss detected

Multiplicative Decrease
cut sending rate in half at each loss event



AIMD sawtooth behavior: *probing* for bandwidth

TCP AIMD: more

Multiplicative decrease detail: sending rate is

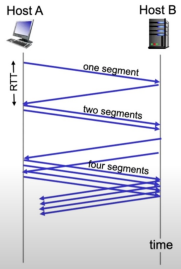
- Cut in half on loss detected by triple duplicate ACK (TCP Reno)
- Cut to 1 MSS (maximum segment size) when loss detected by timeout (TCP Tahoe)

Why AIMD?

- AIMD – a distributed, asynchronous algorithm – has been shown to:
  - optimize congested flow rates network wide!
  - have desirable stability properties

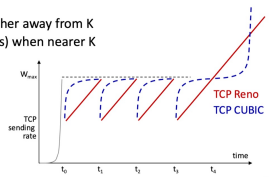
TCP slow start

- when connection begins, increase rate exponentially until first loss event:
  - initially **cwnd** = 1 MSS
  - double **cwnd** every RTT
  - done by incrementing **cwnd** for every ACK received
- *summary:* initial rate is slow, but ramps up exponentially fast

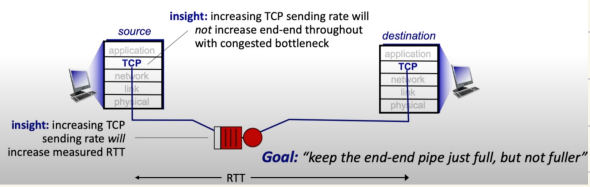


TCP CUBIC

- K: point in time when TCP window size will reach W_{max}
  - K itself is tuneable
- increase W as a function of the *cube* of the distance between current time and K
  - larger increases when further away from K
  - smaller increases (cautious) when nearer K
- TCP CUBIC default in Linux, most popular TCP for popular Web servers

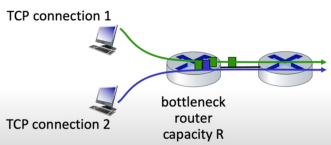


- TCP (classic, CUBIC) increase TCP's sending rate until packet loss occurs at some router's output: the *bottleneck link*
- understanding congestion: useful to focus on congested bottleneck link



→ delay based approach can help

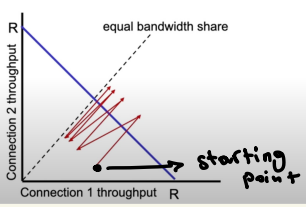
Fairness goal: if *K* TCP sessions share same bottleneck link of bandwidth *R*, each should have average rate of *R/K*



Q: is TCP Fair?

Example: two competing TCP sessions:

- additive increase gives slope of 1, as throughout increases
- multiplicative decrease decreases throughput proportionally



Is TCP fair?

A: Yes, under idealized assumptions:
- same RTT
- fixed number of sessions only in congestion avoidance

From the starting point, both session are not experiencing loss → both would increase → they experience loss → ... → they converge to equilibrium (Fair) no matter where they start

Fairness: must all network apps be "fair"?

Fairness and UDP

- multimedia apps often do not use TCP
  - do not want rate throttled by congestion control
- instead use UDP:
  - send audio/video at constant rate, tolerate packet loss
- there is no "Internet police" policing use of congestion control

Fairness, parallel TCP connections

- application can open *multiple* parallel connections between two hosts
- web browsers do this , e.g., link of rate R with 9 existing connections:
  - new app asks for 1 TCP, gets rate R/10
  - new app asks for 11 TCPs, gets R/2

→ ways of having unfairness

↳ might affect "fair" tcp packets

Network layer: Transports a segment from a sending host to a receiving host.

sender → encapsulates segments into datagrams, passes to link layer

receiver → delivers segments to transport layer protocol

Routers → examines header fields in all IP datagrams passing through it

→ moves datagrams from input ports to output ports to transfer datagrams along end-end path

network-layer functions: (happens in ms)

- *forwarding:* move packets from a router's input link to appropriate router output link
- *routing:* determine route taken by packets from source to destination
  - *routing algorithms*

(happens in seconds)

analogy: taking a trip

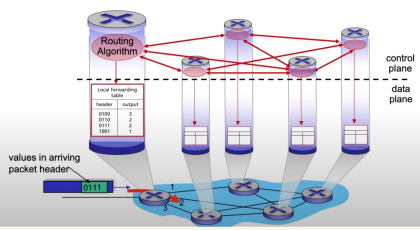
- *forwarding:* process of getting through single interchange
- *routing:* process of planning trip from source to destination

Forwarding is a local action in the network layer while routing is a network-wide action performed by the router

Each router does its own routing

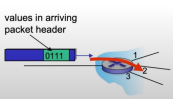
Per-router control plane

Individual routing algorithm components *in each and every router* interact in the control plane



Data plane:

- *local*, per-router function
- determines how datagram arriving on router input port is forwarded to router output port

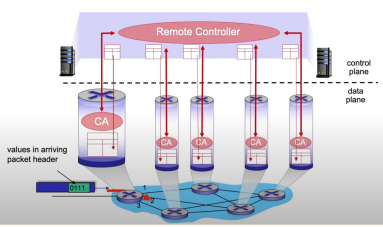


Control plane

- *network-wide* logic
- determines how datagram is routed among routers along end-end path from source host to destination host
- two control-plane approaches:
  - *traditional routing algorithms:* implemented in routers
  - *software-defined networking (SDN):* implemented in (remote) servers

Software-Defined Networking (SDN) control plane

Remote controller computes, installs forwarding tables in routers



In Software-defined networks, routing is done at remote servers while routers do their usual forwarding → Forwarding table gets modified remotely

Internet has a best effort service model

→ no guarantees for bandwidth, loss, order, timing

Network service model

Q: What *service model* for "channel" transporting datagrams from sender to receiver?

example services for *individual* datagrams:

- guaranteed delivery
- guaranteed delivery with less than 40 msec delay

example services for a *flow* of datagrams:

- in-order datagram delivery
- guaranteed minimum bandwidth to flow
- restrictions on changes in inter-packet spacing

Network Architecture	Service Model	Quality of Service (QoS) Guarantees ?			
		Bandwidth	Loss	Order	Timing
Internet	best effort	none	no	no	no
ATM	Constant Bit Rate	Constant rate	yes	yes	yes
ATM	Available Bit Rate	Guaranteed min	no	yes	no
Internet	Intserv Guaranteed (RFC 1633)	yes	yes	yes	yes
Internet	Diffserv (RFC 2475)	possible	possibly	possibly	no

→ not really used. Why?

Reflections on best-effort service:

- simplicity of mechanism has allowed Internet to be widely deployed adopted
- sufficient provisioning of bandwidth allows performance of real-time applications (e.g., interactive voice, video) to be "good enough" for "most of the time"
- replicated, application-layer distributed services (datacenters, content distribution networks) connecting close to clients' networks, allow services to be provided from multiple locations
- congestion control of "elastic" services helps

It's hard to argue with success of best-effort service model

router :

packet → input port → switching fabric → output port

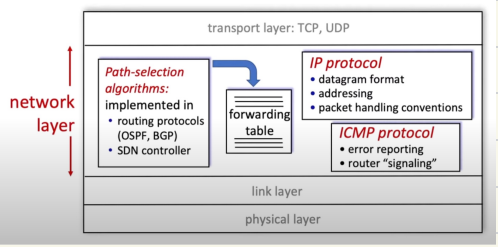
longest prefix match

when looking for forwarding table entry for given destination address, use *longest* address prefix that matches destination address.

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

There are 2^{32} different destinations and routers can't contain a link interface for each → we set them for a range (using *).

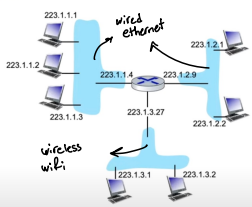
host, router network layer functions:



each with their own IP address

IP addressing: introduction

- IP address: 32-bit identifier associated with each host or router *interface*
- interface: connection between host/router and physical link
  - router's typically have multiple interfaces
  - host typically has one or two interfaces (e.g., wired Ethernet, wireless 802.11)



dotted-decimal IP address notation:

223.1.1.1 = 11011111 00000001 00000001 00000001

223 1 1 1

- *What's a subnet ?*
  - device interfaces that can physically reach each other without passing through an intervening router

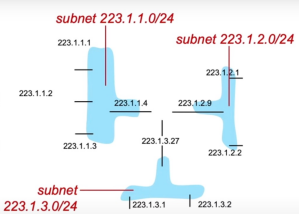
→ 3 subnets in the example

- IP addresses have structure:
  - subnet part: devices in same subnet have common high order bits
  - host part: remaining low order bits

Subnets

Recipe for defining subnets:

- detach each interface from its host or router, creating "islands" of isolated networks
- each isolated network is called a *subnet*

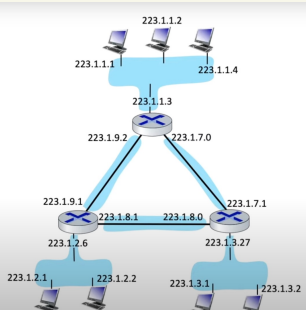


subnet mask: /24

(high-order 24 bits: subnet part of IP address)

Subnets

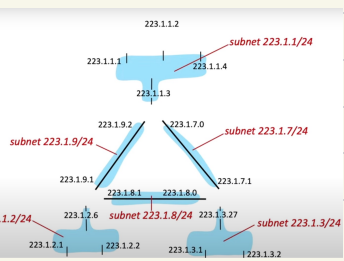
- where are the subnets?
- what are the /24 subnet addresses?



→

Subnets

- where are the subnets?
- what are the /24 subnet addresses?



That's actually two questions:

1. Q: How does a *host* get IP address within its network (host part of address)?
2. Q: How does a *network* get IP address for itself (network part of address)

How does *host* get IP address?

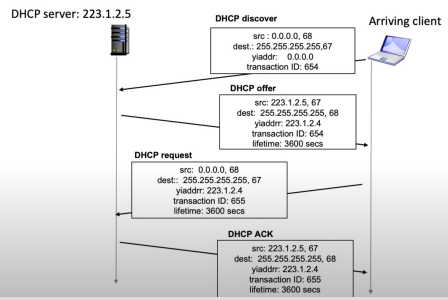
- hard-coded by sysadmin in config file (e.g., /etc/rc.config in UNIX)
- DHCP: Dynamic Host Configuration Protocol: dynamically get address from as server
  - "plug-and-play"

DHCP: Dynamic Host Configuration Protocol

goal: host *dynamically* obtains IP address from network server when it "joins" network

- can renew its lease on address in use
- allows reuse of addresses (only hold address while connected/on)
- support for mobile users who join/leave network

DHCP client-server scenario



DHCP: more than IP addresses

DHCP can return more than just allocated IP address on subnet:

- address of first-hop router for client
- name and IP address of DNS sever → provide access to a DNS server for the user
- network mask (indicating network versus host portion of address)
  - ↳ subnet mask ⇒ helps client determine which devices are in the local network and which are outside

First-hop router: The router that serves as the first point of contact between a client and the larger network, including the internet. Allows the user to communicate outside its local subnet or network

IP addresses: how to get one?

Q: how does *network* get subnet part of IP addr?

A: gets allocated portion of its provider ISP's address space

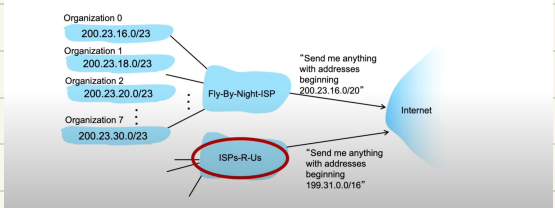
ISP's block	11001000 00010111 00010000	00000000	200.23.16.0/20

ISP can then allocate out its address space in 8 blocks:

Organization 0	11001000 00010111 00010000	00000000	200.23.16.0/23
Organization 1	11001000 00010111 00010010	00000000	200.23.18.0/23
Organization 2	11001000 00010111 00010100	00000000	200.23.20.0/23
...	.....	....	....
Organization 7	11001000 00010111 00011110	00000000	200.23.30.0/23

Hierarchical addressing: route aggregation

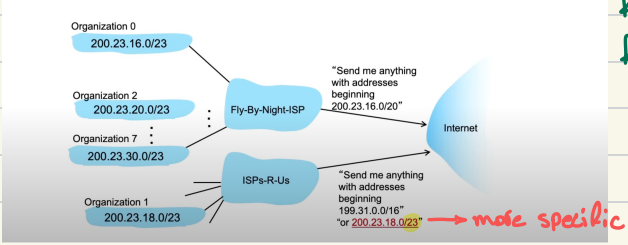
hierarchical addressing allows efficient advertisement of routing information:



what if an org changes their ISP? we add the more-specific address to the new ISP routing. → we know that routers function by longest prefix thus, it will forward correctly.

Hierarchical addressing: more specific routes

- Organization 1 moves from Fly-By-Night-ISP to ISPs-R-Us
- ISPs-R-Us now advertises a more specific route to Organization 1



→ more specific

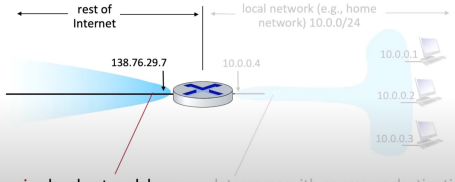
* ISPs get their address range from a corp called ICANN.

IPv4 is not enough → using NAT to help with space exhaustion

IPV6 → 128-bit alternative ✓

NAT: network address translation

NAT: all devices in local network share just one IPv4 address as far as outside world is concerned



all datagrams *leaving* local network have *same* source NAT IP address: 138.76.29.7, but *different* source port numbers

datagrams with source or destination in this network have 10.0.0/24 address for source, destination (as usual)

All devices in local network have 32 bit addresses in a private IP address space. 10/8, 172.16/12, 192.168/16 are the prefixes that can only be used in local nets

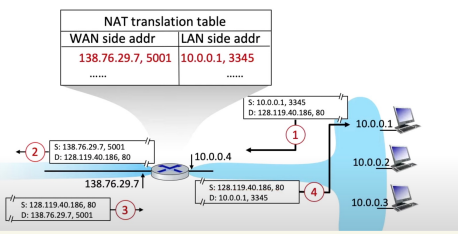
implementation: NAT router must (transparently):

- outgoing datagrams: replace (source IP address, port #) of every outgoing datagram to (NAT IP address, new port #)
- remember (in NAT translation table) every (source IP address, port #) to (NAT IP address, new port #) translation pair
- incoming datagrams: replace (NAT IP address, new port #) in destination fields of every incoming datagram with corresponding (source IP address, port #) stored in NAT table

- advantages:
  - just one IP address needed from provider ISP for *all* devices
  - can change addresses of host in local network without notifying outside world
  - can change ISP without changing addresses of devices in local network
  - security: devices inside local net not directly addressable, visible by outside world

- NAT has been controversial:
  - routers "should" only process up to layer 3
  - address "shortage" should be solved by IPv6
  - violates end-to-end argument (port # manipulation by network-layer device)
  - NAT traversal: what if client wants to connect to server behind NAT?
- but NAT is here to stay:
  - extensively used in home and institutional nets, 4G/5G cellular nets

NAT: network address translation



NAT changes port which is an end host issue

IPv6: motivation

- initial motivation: 32-bit IPv4 address space would be completely allocated
- additional motivation:
  - speed processing/forwarding: 40-byte fixed length header
  - enable different network-layer treatment of "flows"

In IPv6 there is no:

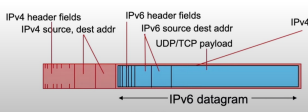
checksum → speed processing at routers

no fragmentation

the header also has a fixed length → allows faster processing

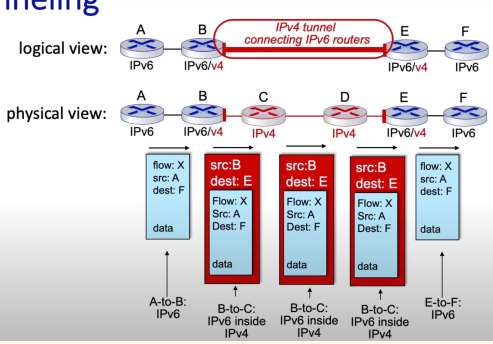
Transition from IPv4 to IPv6

- not all routers can be upgraded simultaneously
  - no "flag days"
  - how will network operate with mixed IPv4 and IPv6 routers?
- tunneling: IPv6 datagram carried as *payload* in IPv4 datagram among IPv4 routers ("packet within a packet")
  - tunneling used extensively in other contexts (4G/5G)

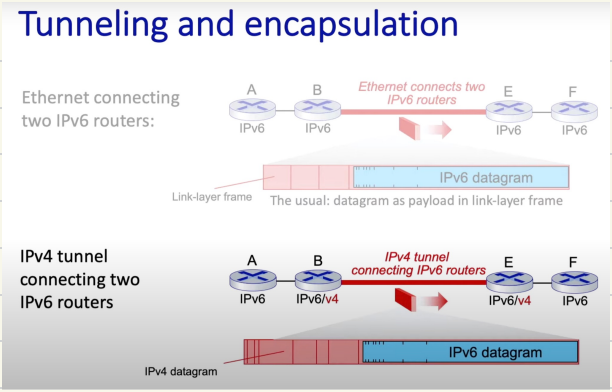


IPv4 is still used in majority of the web

Tunneling



Tunneling and encapsulation



Tunneling → IPv6 datagram sent as payload in IPv4 datagram

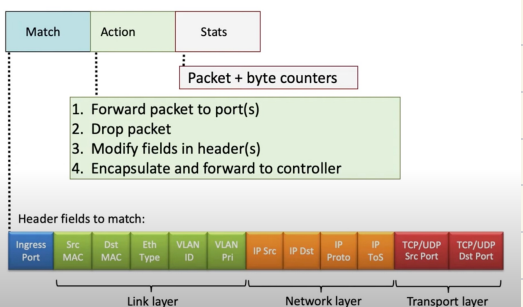
we are putting the ipv6 packet inside the data section of an ipv4 packet like an envelop IPv6 that we put it inside another envelop, IPv4. This will help the packet travel through an IPv4 net

what if IPv6 packet is too big? we break it into smaller pieces, and then send them.

Flow table abstraction

- flow: defined by header fields
- generalized forwarding: simple packet-handling rules
  - match: pattern values in packet header fields
  - actions: for matched packet: drop, forward, modify, matched packet or send matched packet to controller → NAT, ↳ firewall
  - priority: disambiguate overlapping patterns
  - counters: #bytes and #packets

OpenFlow: flow table entries



↳ link layer, network layer, transport

OpenFlow abstraction

- match+action: abstraction unifies different kinds of devices

Router
- *match:* longest destination IP prefix
- *action:* forward out a link

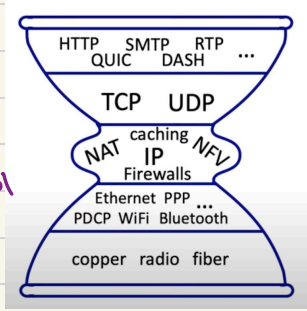
Switch
- *match:* destination MAC address
- *action:* forward or flood

Firewall
- *match:* IP addresses and TCP/UDP port numbers
- *action:* permit or deny

NAT
- *match:* IP address and port
- *action:* rewrite address and port

doing this in the whole network allows us to have a network-wide programability over its behavior

All devices need to have the IP protocol for their network layer protocol unlike other layers

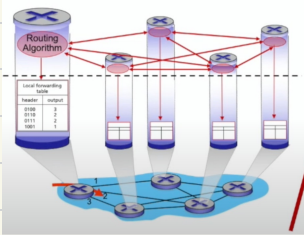


emergence of middleboxes, operating inside the network

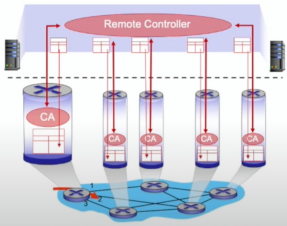
→ we usually want the complexity to be handled on the edge (e.g. rdt, congestion control)

TCP on the edge

Per-router control plane



SDN control plane



Two methods for control plane:
Decentralized (per-router)
Centralized (SDN)

Routing algorithm classification

global: all routers have *complete* topology, link cost info
- "link state" algorithms

How fast do routes change?

static: routes change slowly over time

dynamic: routes change more quickly
- periodic updates or in response to link cost changes

decentralized: iterative process of computation, exchange of info with neighbors
- routers initially only know link costs to attached neighbors
- "distance vector" algorithms

global or decentralized information?

→ Dijkstra is an example of a link-state algorithm

→ Bellman-Ford is more of a "distance vector" algorithm (nodes update their neighbors)

Dijkstra's link-state routing algorithm

- centralized: network topology, link costs known to *all* nodes
  - accomplished via "link state broadcast"
  - all nodes have same info
- computes least cost paths from one node ("source") to all other nodes
  - gives *forwarding table* for that node
- iterative: after *k* iterations, know least cost path to *k* destinations

notation
- $C_{a,b}$: direct link cost from node *a* to *b*; = ∞ if not direct neighbors
- *D(a)*: *current* estimate of cost of least-cost-path from source to destination *a*
- *p(a)*: predecessor node along path from source to *a*
- *N'*: set of nodes whose least-cost-path *definitively* known

For Dijkstra, route oscillation is possible
↳ based on traffic, costs might change

Dijkstra's algorithm: discussion

algorithm complexity: *n* nodes
- each of *n* iteration: need to check all nodes, *w*, not in *N*
- $n(n+1)/2$ comparisons: $O(n^2)$ complexity
- more efficient implementations possible: $O(n\log n)$

message complexity:
- each router must *broadcast* its link state information to other *n* routers
- efficient (and interesting!) broadcast algorithms: $O(n)$ link crossings to disseminate a broadcast message from one source
- each router's message crosses $O(n)$ links: overall message complexity: $O(n^2)$

Distance vector algorithm:

each node:

wait for (change in local link cost or msg from neighbor)

↓

recompute my DV estimates using DV received from neighbor

↓

if my DV to any destination has changed, *send my new DV* my neighbors, else do nothing.

iterative, asynchronous: each local iteration caused by:
- local link cost change
- DV update message from neighbor

distributed, self-stopping: each node notifies neighbors *only* when its DV changes
- neighbors then notify their neighbors – *only if necessary*
- no notification received, no actions taken!

issue: if some nodes weights get updated, it might take some time for the nodes on the other end of the graph to get updated.
↳ like looking at the stars

Comparison of LS and DV algorithms

message complexity

LS: n routers, $O(n^2)$ messages sent

DV: exchange between neighbors; convergence time varies

speed of convergence

LS: $O(n^2)$ algorithm, $O(n^2)$ messages
- may have oscillations

DV: convergence time varies
- may have routing loops
- count-to-infinity problem

robustness: what happens if router malfunctions, or is compromised?

LS:
- router can advertise incorrect *link* cost
- each router computes only its *own* table

DV:
- DV router can advertise incorrect *path* cost ("I have a *really* low-cost path to everywhere"): *black-holing*
- each router's DV is used by others: error propagate thru network

DVs message complexity depends on the graphs diameter. In addition, if it advertises a wrong low-cost path, the error propagates through the server.

In LS, router can advertise incorrect link cost → weight

others are safe of the error due to ← each router computes its own table being calculated independently

Internet approach to scalable routing

aggregate routers into regions known as "autonomous systems" (AS) (a.k.a. "domains")

intra-AS (aka "intra-domain"): routing among routers *within same AS ("network")*
- all routers in AS must run same intra-domain protocol
- routers in different AS can run different intra-domain routing protocols
- gateway router: at "edge" of its own AS, has link(s) to router(s) in other AS'es

inter-AS (aka "inter-domain"): routing *among* AS'es
- gateways perform inter-domain routing (as well as intra-domain routing)

OSPF (Open Shortest Path First) routing

- "open": publicly available
- classic link-state
  - each router floods OSPF link-state advertisements (directly over IP rather than using TCP/UDP) to all other routers in entire AS
  - multiple link costs metrics possible: bandwidth, delay
  - each router has full topology, uses Dijkstra's algorithm to compute forwarding table
- *security:* all OSPF messages authenticated (to prevent malicious intrusion)

Intra-AS

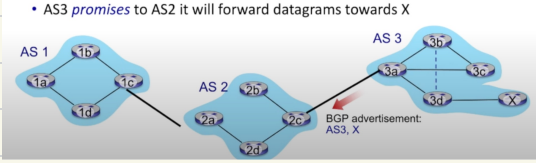
Internet inter-AS routing: BGP

- BGP (Border Gateway Protocol): *the* de facto inter-domain routing protocol
  - "glue that holds the Internet together"
- allows subnet to advertise its existence, and the destinations it can reach, to rest of Internet: *"I am here, here is who I can reach, and how"*
- BGP provides each AS a means to:
  - obtain destination network reachability info from neighboring ASes (eBGP)
  - determine routes to other networks based on reachability information and *policy*
  - propagate reachability information to all AS-internal routers (iBGP)
  - advertise (to neighboring networks) destination reachability info

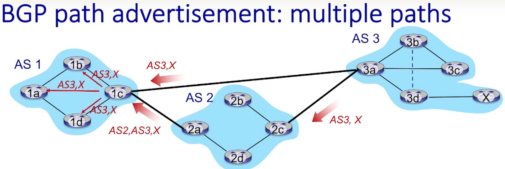
↳ does the router want to route traffic forwarded?

BGP basics

- BGP session: two BGP routers ("peers, speakers") exchange BGP messages over semi-permanent TCP connection:
  - advertising *paths* to different destination network prefixes (e.g., to a destination /16 network)
  - BGP is a "path vector" protocol
- when AS3 gateway 3a advertises path AS3,X to AS2 gateway 2c:
  - AS3 *promises* to AS2 it will forward datagrams towards X



BGP path advertisement: multiple paths

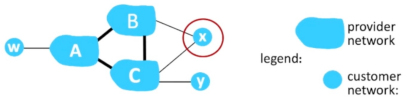


gateway routers may learn about multiple paths to destination:
- AS1 gateway router 1c learns path *AS2,AS3,X* from 2a
- AS1 gateway router 1c learns path *AS3,X* from 3a
- based on *policy,* AS1 gateway router 1c chooses path *AS3,X* and advertises path within AS1 via iBGP

BGP: achieving policy via advertisements

ISP only wants to route traffic to/from its customer networks (does not want to carry transit traffic between other ISPs – a typical "real world" policy)
- A advertises path Aw to B and to C
- B *chooses not to advertise* BAw to C!
  - B gets no "revenue" for routing CBAw, since none of C, A, w are B's customers
  - C does *not* learn about CBAw path
- C will route CAw (not using B) to get to w

BGP: achieving policy via advertisements (more)

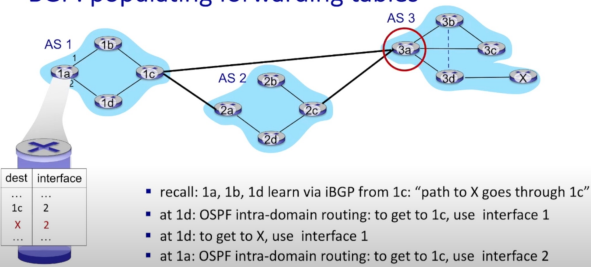


ISP only wants to route traffic to/from its customer networks (does not want to carry transit traffic between other ISPs – a typical "real world" policy)

- A,B,C are provider networks
- x,w,y are customer (of provider networks)
- x is dual-homed: attached to two networks
- *policy to enforce:* x does not want to route from B to C via x
  - .. so x will not advertise to B a route to C

Since x is a customer for B and C, it does not tell them that it has both connections → otherwise, it might carry transit traffic between the ISPs

BGP: populating forwarding tables



- recall: 1a, 1b, 1d learn via iBGP from 1c: "path to X goes through 1c"
- at 1d: OSPF intra-domain routing: to get to 1c, use interface 1
- at 1d: to get to X, use interface 1
- at 1a: OSPF intra-domain routing: to get to 1c, use interface 2
- at 1a: to get to X, use interface 2

Why a logically centralized control plane?

→ easier network management

→ allows "programming routers" ⇒ Centralized programming is easier
distributed programming is harder

By open-sourcing such systems, it allowed for more progress and innovation in a shorter time period.

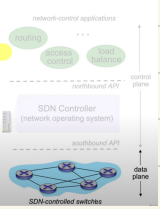
If we want to change path taken in a network or split a path into different paths (load balancing), in a per-router approach, we need to change the link weight → more like knobs than control

* with SDN this will be possible since it's centralized

Software defined networking (SDN)

Data-plane switches:

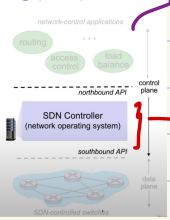
- fast, simple, commodity switches implementing generalized data-plane forwarding (Section 4.4) in hardware
- flow (forwarding) table computed, installed under controller supervision
- API for table-based switch control (e.g., OpenFlow)
  - defines what is controllable, what is not



Software defined networking (SDN)

SDN controller (network OS):

- maintain network state information
- interacts with network control applications "above" via northbound API
- interacts with network switches "below" via southbound API
- implemented as distributed system for performance, scalability, fault-tolerance, robustness



Brains of control: control functions using lower level's API

interface layer to network control apps: abstractions API

network-wide state management : state of networks links, switches, services: a *distributed database*

communication: communicate between SDN controller and controlled switches