Lecture 12: BGP wrapup, Ports intro

Today
- BGP: Prefix hijacking
- Transport layer intro
- Ports and socket tables

- IP: now due Friday
    - You can make bugfixes after the deadline without using late days
    - Look for meeting signups (will send announcement on Edstem)

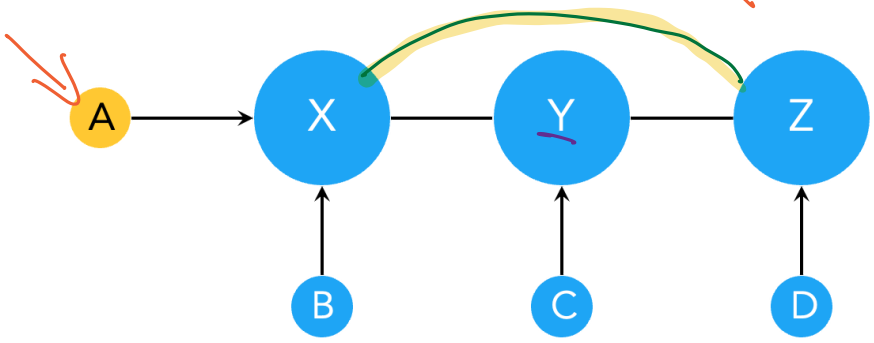
- TCP partner form: due Friday 11:59pm
    - Please please fill it out!!!
    - TCP: officially out Tuesday
    - TCP gearup I: Thursday next week 5pm

- HW2: as soon as I can get there (due in >1wk)

Lecture 12: BGP wrapup, ports

Advertised by…	Export to…
Customer	Everyone
Peer	Customers only
Provider	Customers only

Warmup: Given the following AS relationships, which ASes will A know about?



Customer ("A is customer of X")

Peer

CAN A REACH...

B ✓

X ✓

Y ✓

C ✓

Z NO!

D NO!

WAYS TO FIX?

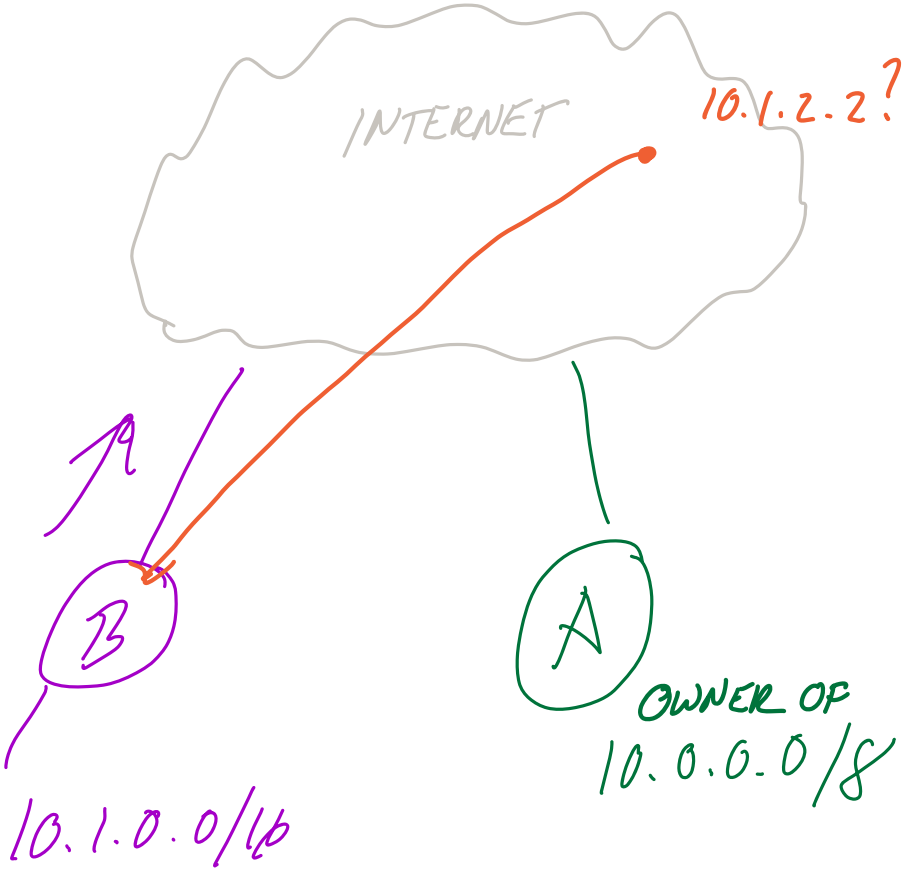
EG. - PEER X,Z

- X BECOMES CUST. OF Z
- X " " Y

Prefix hijacking

What if B decides to start advertising A's prefix?

What if it starts advertising a *more specific* prefix?



Prefix hijacking: malicious router can advertise prefix it doesn't own
=> will get traffic for that prefix

If advertised prefix is more specific than the original, other routers will prefer the more specific prefix!

Design problem: Who "owns" a prefix? Who is allowed to *originate* a prefix?
=> BGP by default **does not verify** announce messages match the network that owns them.
=> ASes have their own security polices (and they are being more widely adopted), but they are not unified

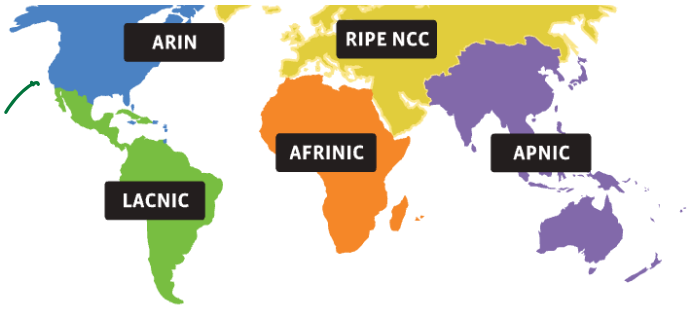
If you can hijack a prefix, what can you do?
- Intercept or redirect packets for some IP range
- Snooping
- Modify/slow down traffic

=> Prefix hijacking is hard to debug, because it may only be visible from certain parts of a network. (Though this is easier to see for companies that have visibility from very large networks.)

Who "owns" an IP prefix?

Allocated by internet authorities, hierarchically:

- Top-level: Internet Assigned Numbers Authority (IANA)
- Regional Internet Registries
- Internet Service providers



Officially, prefixes are allocated by these organizations: they allocate ranges of IP prefixes to ISPs or other entities (like Brown).

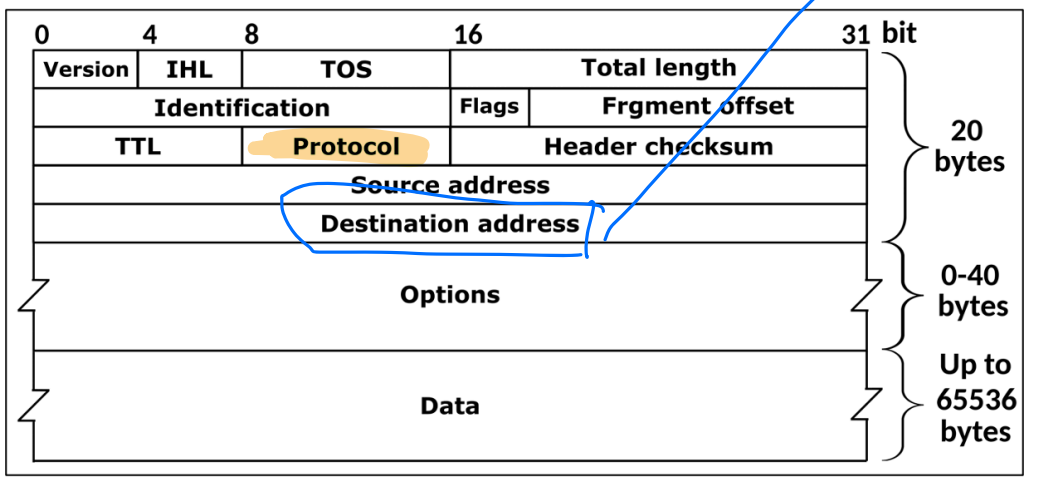
There is a lot of information available about current allocations, but BGP doesn't use this to verify routes by default. In recent years, network engineers have developed mechanisms for AS owners to write policies about how their routes should be advertised, and ways for other routers to check them.

For more on this, see the notes on RPKI at the end of this document--we didn't cover RPKI in class this year, but feel free to take a look for reference.

Next unit: Transport layer

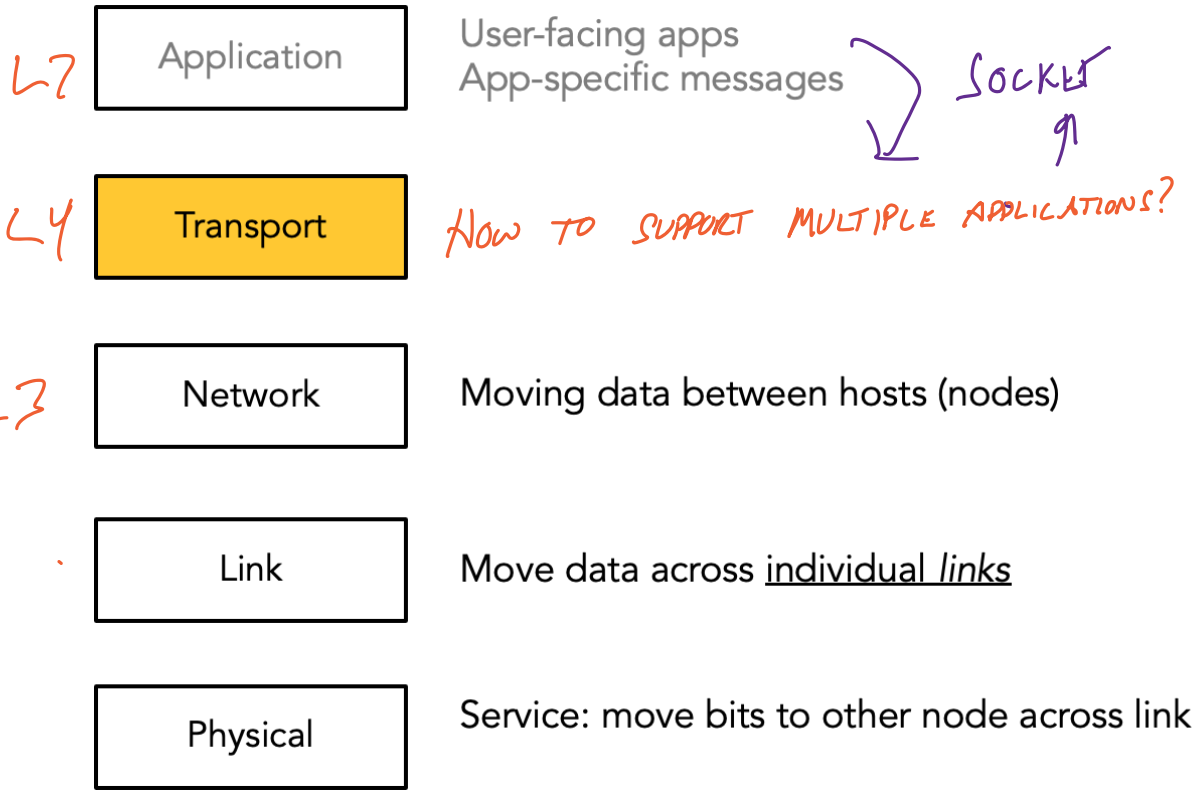
The story so far: network layer

The IP header:



The story so far: the network layer (L3): moving packets between hosts, anywhere on the internet

Where do we go from here?



Two major types of transport protocols:

TCP: "reliable, connection-oriented byte stream"

FOCUS ON THIS PART TODAY

UDP: "unreliable datagram service"

Major challenges

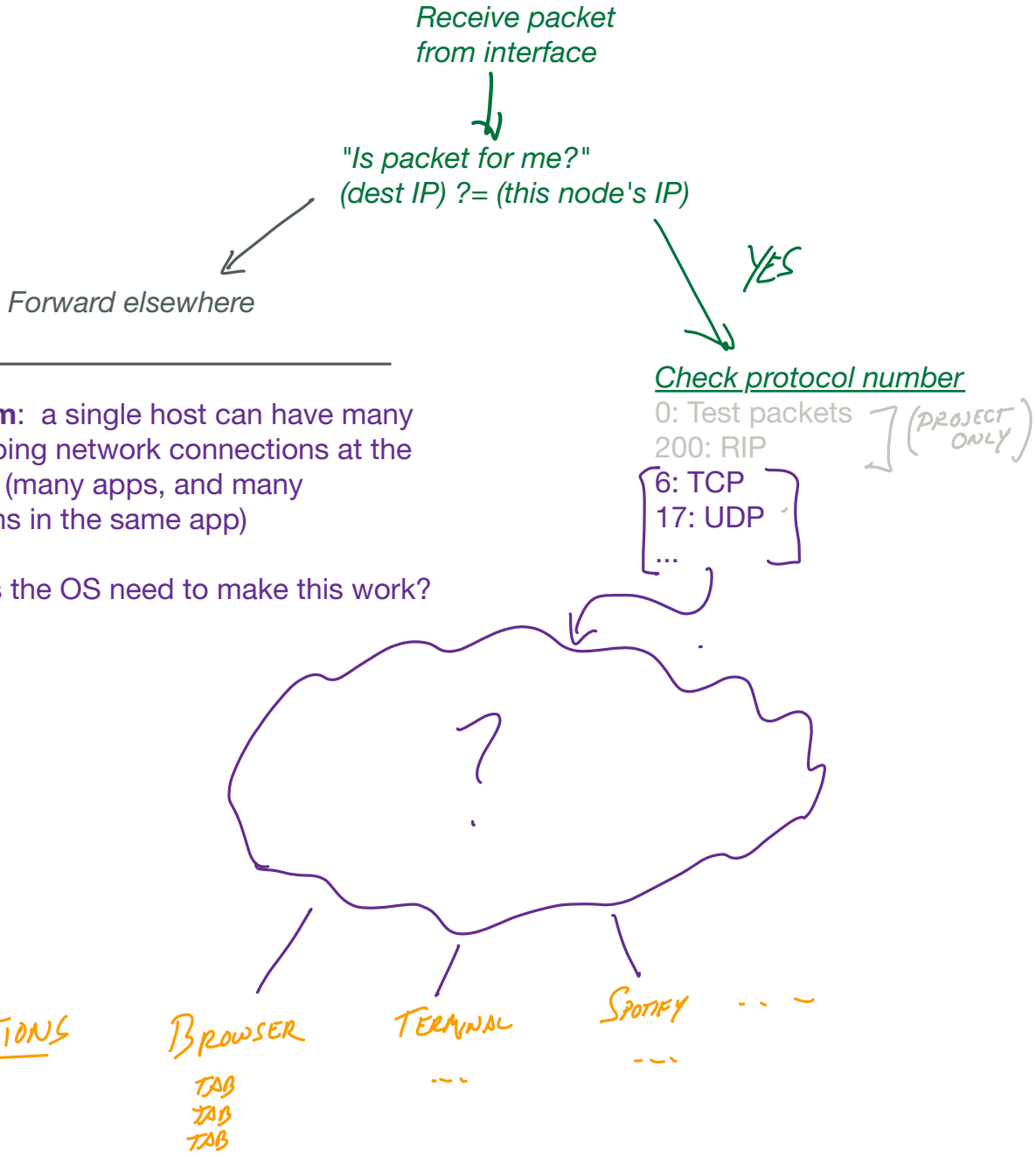
- (today) Multiplexing: multiple connections at same IP (e.g., different apps, multiple connections for the same app

  => Realized by: **sockets** OS abstraction for a network connection, like a file dedscriptor

- (next lectures) Messaging: how to turn user messages into packets (and back again)

Key concept for today: port numbers => CONNECTIONS

Sketch: how to support multiple applications?



Brainstorm: a single host can have many many ongoing network connections at the same time (many apps, and many connections in the same app)

What does the OS need to make this work?

Things we need in the OS:
- Some way to identify different connections
  => Port numbers (part of packet header, used by OS)
- Table to keep track of open connections
- Need a "pipe" to store data and deliver it to each application
  => Need to store some state about each open connection

Each connection represented by a socket in OS, which has some state about the connection
=> OS kernel receives all packets => needs to map each packet to a socket to deliver to app

Transport layer: the goal

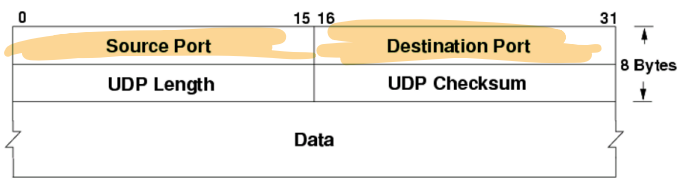
Use port numbers to multiplex connections using the same IP

What's a port number?

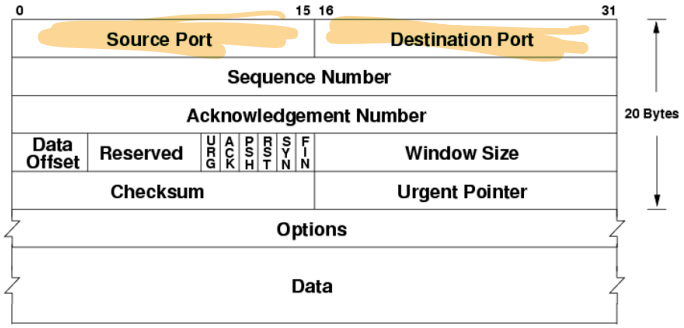
- 16-bit unsigned integer, 0-65535
- port < 1024: "Well known port numbers"
- port > 20000 (usually): "ephemeral ports", for general app use

MOST COMMON APPS

UDP header



TCP header



Port numbers are part of the transport layer (not part of IP!)
=> First two fields of TCP/UDP headers

This might seem familiar...

We've worked with sockets before, when writing applications...
Remember snowcast? (or, in lecture, the guessing game?)

Client

Client-side: create a connection to a specific IP:port
=> Returns socket representing this connection

```
func main() {
     conn, err := net.Dial("tcp", "127.0.0.1:5000")
     // . . .
     conn.Write(...)

}
```

Server

Server-side: listen on a port => wait for clients to connect
=> Also called "binding" on a port
=> Returns a listen socket

```
func main() {
     listenConn, err := net.Listen("tcp", "127.0.0.1:5000")
     for {
        clientConn, err := listenConn.Accept()

        go handleClient(clientConn)
     }
}
```

When client connects, new socket created!
(demo today, much more info later)

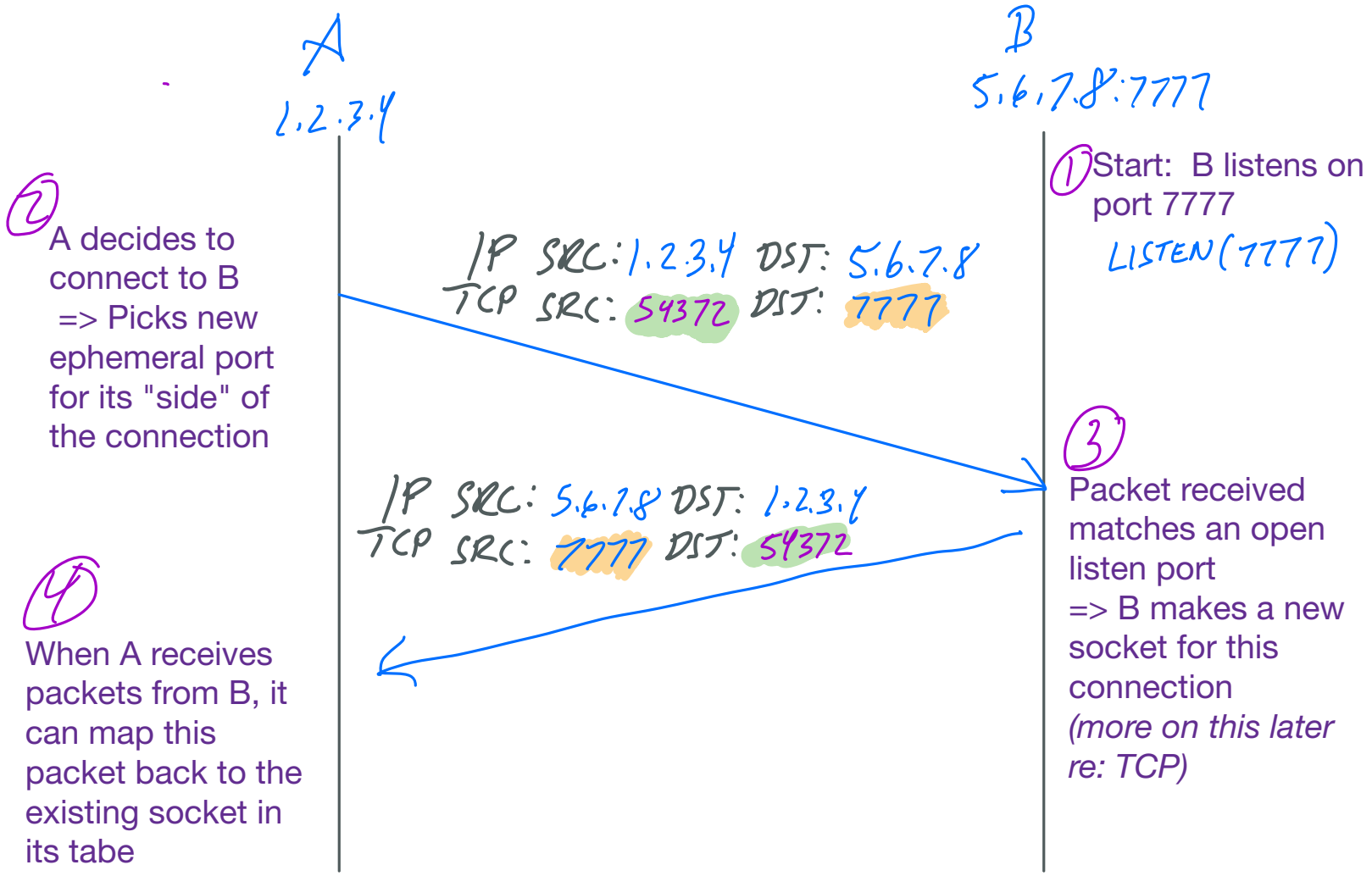
OS decides how to map packets to sockets, based on port numbers
=> Uses source/dest port numbers, and also source/dest IPs,

The 5-tuple of these values:
(protocol number, source IP, source port, destination IP, destination port)

... identifies one unique connection.

Sketch: how ports map to connections

(we'll learn more TCP-specific details soon!)



A's Socket TABLE

LOCAL		REMOTE		STATE
IP	PORT	IP	PORT	
1.2.3.4	54372	5.6.7.8	7777	

B's SOCKET TABLE

LOCAL		REMOTE		STATE
IP	PORT	IP	PORT	
*	7777	*	*	—
5.6.7.8	7777	1.2.3.4	54372	

Tuple of (local IP, local port, remote IP, remote port)
create a unique identifier to look up each individual connection
=> Connections to different hosts will have different remote IPs
=> Connections to different apps on same system will have different remote ports
=> Multiple connections from same host will have different local ports

We'll learn much more about this--only need an overview for now!

Demo: netstat: linux tool to show socket tables
Run with: `netstat` or `ss`

Show listen sockets: `ss -lnp`
Show all sockets: `ss -anp`

Socket tables: Running the guessing game

Server-side

```
Proto  State   Local              Remote               Process
tcp    LISTEN  0.0.0.0:7777       0.0.0.0:*            users:(("guessing-game-s",pid=3125128,fd=3))
tcp    ESTAB   66.228.43.75:7777  128.148.194.11:52878 users:(("guessing-game-s",pid=3125128,fd=4))
tcp    ESTAB   66.228.43.75:7777  128.148.140.184:54100 users:(("guessing-game-s",pid=3125128,fd=8))
tcp    ESTAB   66.228.43.75:7777  128.148.140.184:36642 users:(("guessing-game-s",pid=3125128,fd=7))
                      [ ... sockets for other apps omitted ... ]
```

Things to note:
=> Server has two clients connected from one host (below), one client from a different host
=> All sockets belong to same process (the server!), but map to different FDs!

Socket table from one client (host with IP 128.148.140.184)

```
Proto  State  Local                 Remote              Process
tcp    ESTAB  128.148.140.184:36642 66.228.43.75:7777  users:(("guessing-game-c",pid=172356,fd=3))
tcp    ESTAB  128.148.140.184:54100 66.228.43.75:7777  users:(("guessing-game-c",pid=172402,fd=3))
```

This system had two clients running (each as their own process)

(Third client was on a different system, not shown here)

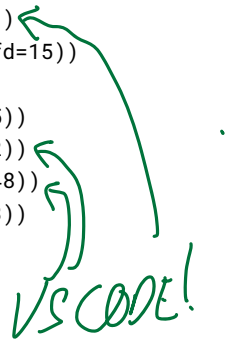
Here's a socket table for a real system (running many different applications)...

Example: Socket table on my laptop (listen sockets)

```
(On linux:  netstat or ss)
# ss -lnp
```

Apps that are listening for connections on my laptop:

```
Netid State  Recv-Q Send-Q Local Address:Port  Peer Address:PortProcess
tcp   LISTEN 0      4096         0.0.0.0:12949      0.0.0.0:*    users:(("docker-proxy",pid=2437235,fd=7))
tcp   LISTEN 0      4096         0.0.0.0:9095       0.0.0.0:*    users:(("docker-proxy",pid=2437219,fd=7))
tcp   LISTEN 0      128          0.0.0.0:22         0.0.0.0:*    users:(("sshd",pid=1838,fd=6))
tcp   LISTEN 0      511        127.0.0.1:35411      0.0.0.0:*    users:(("code",pid=16137,fd=53))
tcp   LISTEN 0      4096       127.0.0.1:36191      0.0.0.0:*    users:(("containerd",pid=1844,fd=15))
tcp   LISTEN 0      4096       127.0.0.1:631        0.0.0.0:*    users:(("cupsd",pid=1837,fd=8))
tcp   LISTEN 0      32         127.0.0.1:53         0.0.0.0:*    users:(("dnsmasq",pid=1504,fd=5))
tcp   LISTEN 0      511        127.0.0.1:32995      0.0.0.0:*    users:(("code",pid=16137,fd=122))
tcp   LISTEN 0      511        127.0.0.1:33981      0.0.0.0:*    users:(("code",pid=2926088,fd=48))
tcp   LISTEN 0      4096       127.0.0.1:1313       0.0.0.0:*    users:(("hugo",pid=1962756,fd=3))
tcp   LISTEN 0      5          127.0.0.1:6600       0.0.0.0:*    users:(("mpd",pid=3036,fd=9))
```



"Normal" (non-listen) sockets on my laptop

```
tcp   ESTAB      0      0              10.3.146.52:42858   128.148.36.21:443   users:(("chromium",pid=5946,fd=732))
tcp   ESTAB      0      0              10.3.146.52:42864   128.148.36.21:443   users:(("chromium",pid=5946,fd=760))
tcp   ESTAB      0      0              10.3.146.52:42870   128.148.36.21:443   users:(("chromium",pid=5946,fd=885))
tcp   ESTAB      0      0              10.3.146.52:42874   128.148.36.21:443   users:(("chromium",pid=5946,fd=844))
tcp   ESTAB      0      0              10.3.146.52:42888   128.148.36.21:443   users:(("chromium",pid=5946,fd=336))
tcp   ESTAB      0      0              10.3.146.52:34916    66.228.43.75:22    users:(("ssh",pid=2947753,fd=3))
tcp   ESTAB      0      0              10.3.146.52:60678    3.171.117.50:443   users:(("firefox",pid=61456,fd=64))
tcp   ESTAB      0      0              10.3.146.52:52896    66.228.43.75:22    users:(("ssh",pid=2992231,fd=3))
tcp   ESTAB      0      0              10.3.146.52:58078    66.228.43.75:22    users:(("ssh",pid=3033479,fd=3))
tcp   ESTAB      0      0              10.3.146.52:35580   34.107.243.93:443   users:(("firefox",pid=3198286,fd=100))
tcp   ESTAB      0      0              10.3.146.52:55738   34.107.243.93:443   users:(("firefox",pid=2134356,fd=67))
tcp   ESTAB      0      0              10.3.146.52:55750   34.107.243.93:443   users:(("firefox",pid=1004388,fd=69))
tcp   ESTAB      0      0              10.3.146.52:55766   34.107.243.93:443   users:(("firefox",pid=2464385,fd=149))
tcp   ESTAB      0      0              10.3.146.52:55770   34.107.243.93:443   users:(("firefox",pid=3447626,fd=62))
tcp   ESTAB      0      0              10.3.146.52:55776   34.107.243.93:443   users:(("firefox",pid=389114,fd=64))
tcp   ESTAB      0      0              10.3.146.52:55796   34.107.243.93:443   users:(("firefox",pid=12949,fd=65))
tcp   ESTAB      0      0              10.3.146.52:55810   34.107.243.93:443   users:(("firefox",pid=61456,fd=135))
tcp   ESTAB      0      0              10.3.146.52:55824   34.107.243.93:443   users:(("firefox",pid=2850132,fd=67))
tcp   ESTAB      0      0              10.3.146.52:58850   34.107.243.93:443   users:(("firefox",pid=1531947,fd=45))
tcp   ESTAB      0      0                127.0.0.1:35411       127.0.0.1:56646 users:(("code",pid=16137,fd=43))
tcp   ESTAB      0      0              10.3.146.52:42132   52.44.223.164:443   users:(("slack",pid=3165264,fd=68))
tcp   ESTAB      0      0              10.3.146.52:52766   52.44.223.164:443   users:(("slack",pid=3165264,fd=46))
tcp   ESTAB      0      0              10.3.146.52:52782   52.44.223.164:443   users:(("slack",pid=3165264,fd=64))
tcp   ESTAB      0      0              10.3.146.52:57734   52.44.223.164:443   users:(("slack",pid=3165264,fd=45))
tcp   ESTAB      0      0              10.3.146.52:57738   52.44.223.164:443   users:(("slack",pid=3165264,fd=20))
tcp   ESTAB      0      0              10.3.146.52:59622   52.44.223.164:443   users:(("slack",pid=3165264,fd=51))
```

Port scanning

What can we learn if we just start connecting to well-known ports?

- Applications have common port numbers
- Network protocols use well-defined patterns

```
deemer@vesta ~/Development % nc <IP addr> 22
SSH-2.0-OpenSSH_9.1
```

⇒ Can discover things about the network
⇒ Can learn about open (vulnerable) systems

Port scanners: try to connect to lots of ports, determine available services, find vulnerable services...

Large-scale port scanning

- Can reveal lots of open/insecure systems!
- Examples:
  - shodan.io
  - VNC roulette
  - Open webcam viewers…
  - …

Disclaimer

(We'll talk a bit more about this in the next lecture)

- Network scanning is easy to detect
- Unless you are the owner of the network, it's seen as malicious activity
- If you scan the whole Internet, the whole Internet will get mad at you (unless done *very* politely)

Do NOT try this on the Brown network. I warned you.

After this page are extra notes on RPKI, a modern way to help secure BGP

We didn't talk about RPKI this year, but read on if you want to know more!

Recap: Prefix hijacking

By default, BGP doesn't verify that advertised routes match their owners

Update: *"I can reach prefix 128.148.0.0/16 through ASes 44444 3356 14325 11078"*

BGP router should ask:

"*Should* AS11078 be originating 138.16.161.0/24?"

Recap: Prefix hijacking

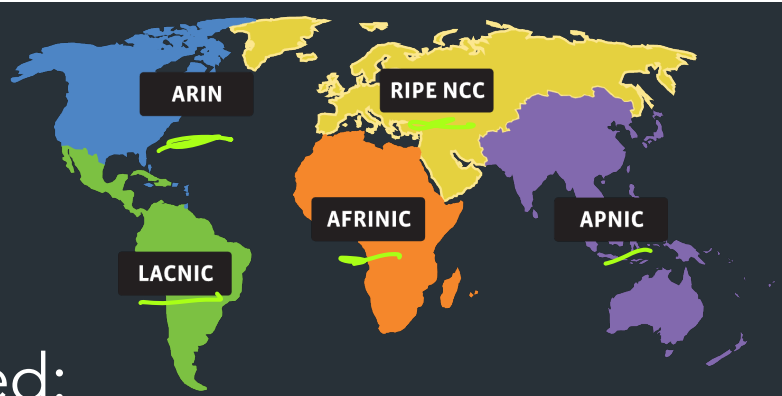
By default, BGP doesn't verify that advertised routes match their owners

Update: *"I can reach prefix 128.148.0.0/16 hrough ASes 44444 3356 14325 11078"*

BGP router should ask:
"*Should* AS11078 be originating 138.16.161.0/24?"

=> Not part of BGP by default. Standards have evolved to help, but adoption is limited.

A modern way: RPKI



Leverages hierarchy of how IPs are allocated:

- Every AS adds a *signature* of its route info in database, signed by authority that allocates addresses
  => ROA (Route Origin Authorization)

- Other ASes can verify ROA signature using cryptography, making it hard to forge

BGP ADVERTISEMENT + SIGNATURE

A modern way: RPKI

Leverages hierarchy of how IPs are allocated:

- Every AS adds a *signature* of its route info in database, signed by authority that allocates addresses
  => ROA (Route Origin Authorization)

- Other ASes can verify ROA signature using cryptography, making it hard to forge

- Can avoid
  - Prefix hijacking
  - Addition, removal, or reordering of intermediate ASes

ROAs for OSHEAN (Brown's provider)

Found *4* ROAs and *9* certificates

ROAs

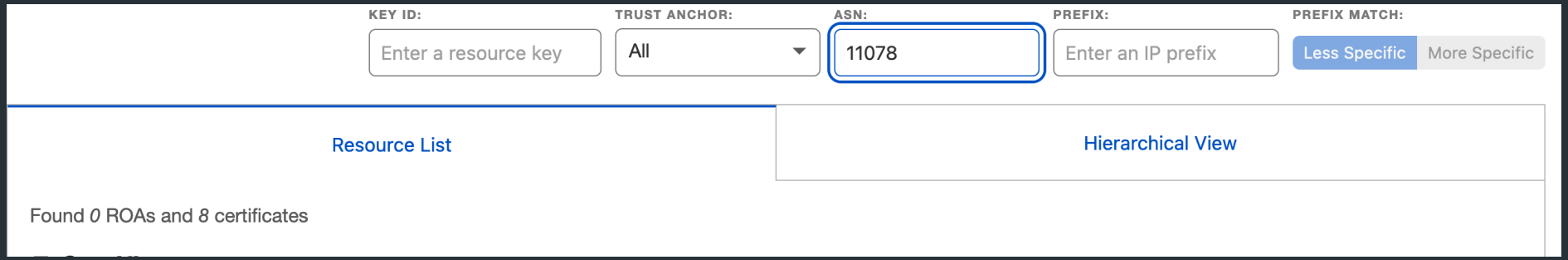
ASN	Prefix	Max Length	IP Family	Trust Anchor	Emitted	Expiration
AS14325	2607:d00::/32	/64	IPv6	ARIN	8/28/2024	in a month
AS14325	131.109.0.0/16	/24	IPv4	ARIN	8/24/2024	in a month

ARIN ATTESTS THAT AS14325 IS ALLOWED TO ADVERTISE 131.109.0.0/16

Prefix: 131.109.0.0/16
Max Length: /24
ASN: 14325
Emitted: Sat, 24 Aug 2024 13:00:41 GMT
Validity: Sat, 24 Aug 2024 13:00:41 GMT - Fri, 22 Nov 2024 14:00:41 GMT
Trust Anchor: ARIN
Name: 5f622e78-c575-449a-836d-8c3f5e873fd4
Key: e852ddd445bb44252099dd8b7c39d39388bea544
Parent Key: 6b3fadc67835654f184fbbaa8e7b2408de32dabb
Path: rsync://rpki.arin.net/repository/arin-rpki-ta/5e4a23ea-e80a-403e-b08c-2171da2157d3/a73420cb-b3cc-4b03-bda7-1be204933ae5/f3e09673-5c6e-4340-ad11-4da8dfb8c777/08efbae2-5477-331b-b473-38399917c289.roa

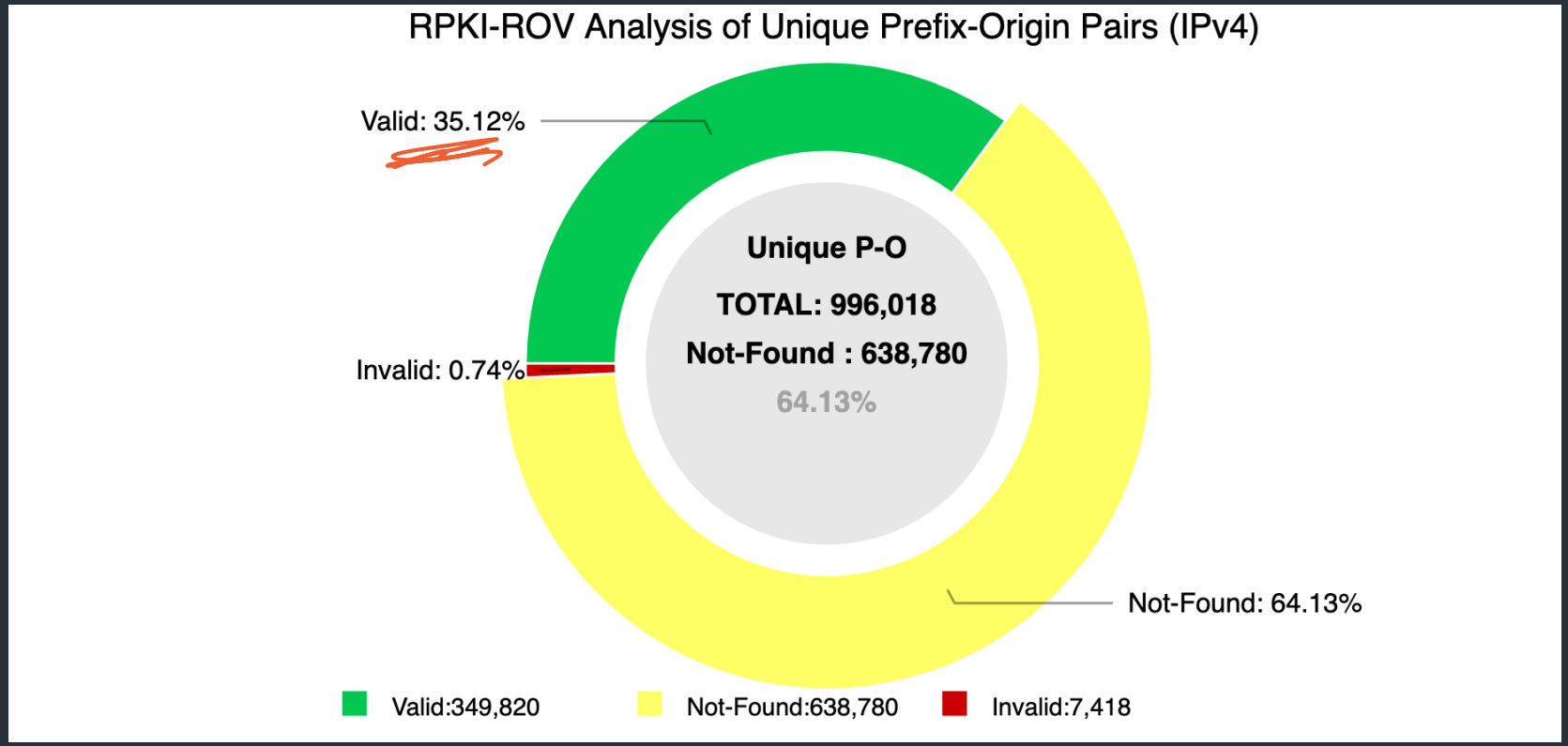
(For us, cryptography details aren't important, but the idea is that the signature makes it difficult for a rogue AS to forge this info, so long as the AS receiving the update verifies it)

ROAs for Brown



Not all networks sign their routes!
(And it seems like Brown does not.)

RPKI deployment (2022)



RPKI deployment (2024)