Routing #4 and Addressing

Today in CS168

- Finishing up Learning Switches & Spanning Tree Protocol
- Addressing

Learning Switches
&
The Spanning Tree Protocol

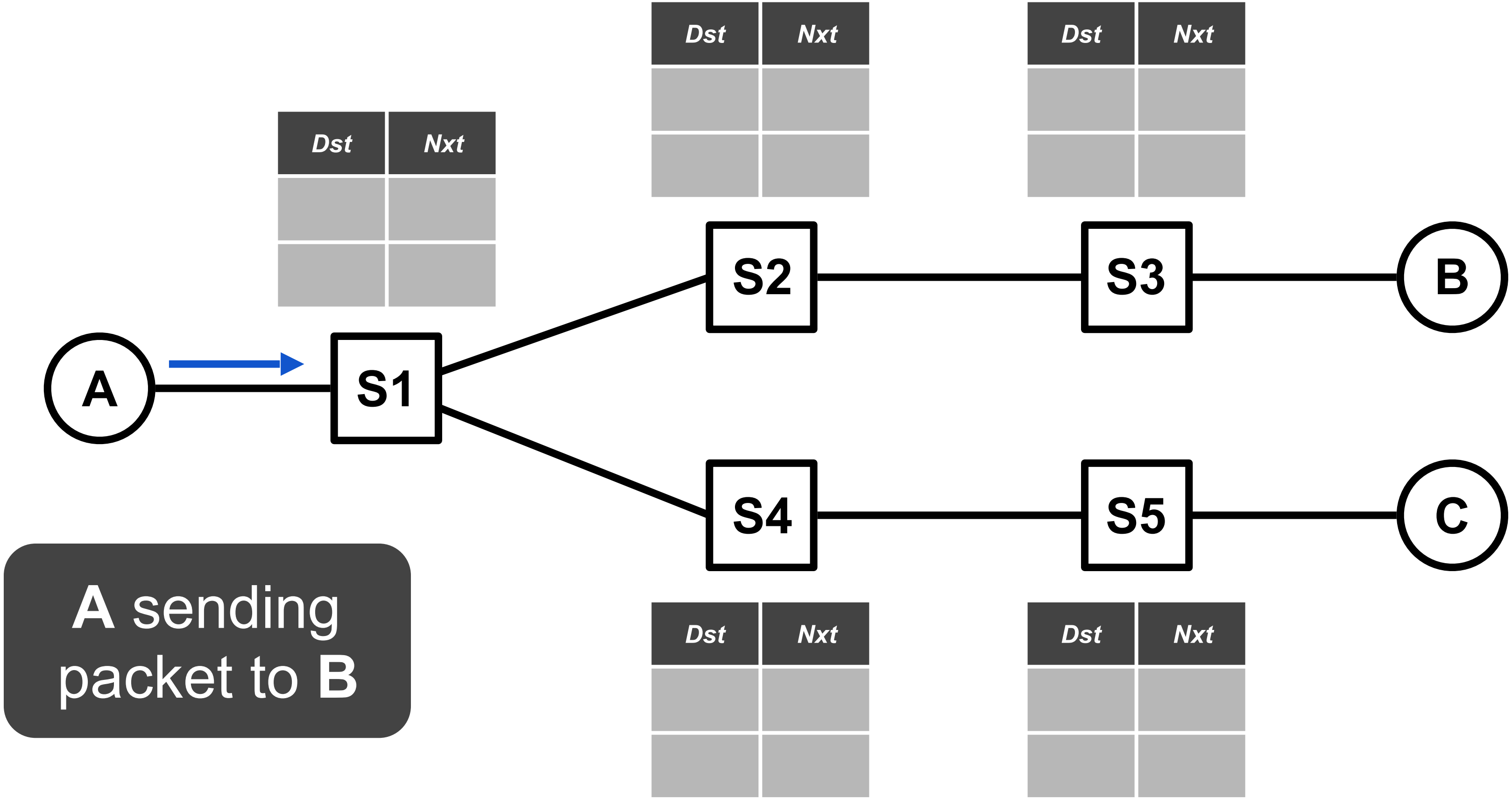
Learning Switches

- We’d been looking at Distance-Vector and Link-State protocols:
    - Tables filled in by ongoing routing process
    - Are “seeded” with static routes for destinations
    - Very common for routing at the network layer (L3)
        - i.e., using IP addresses

- And now a very different approach to filling in our tables!

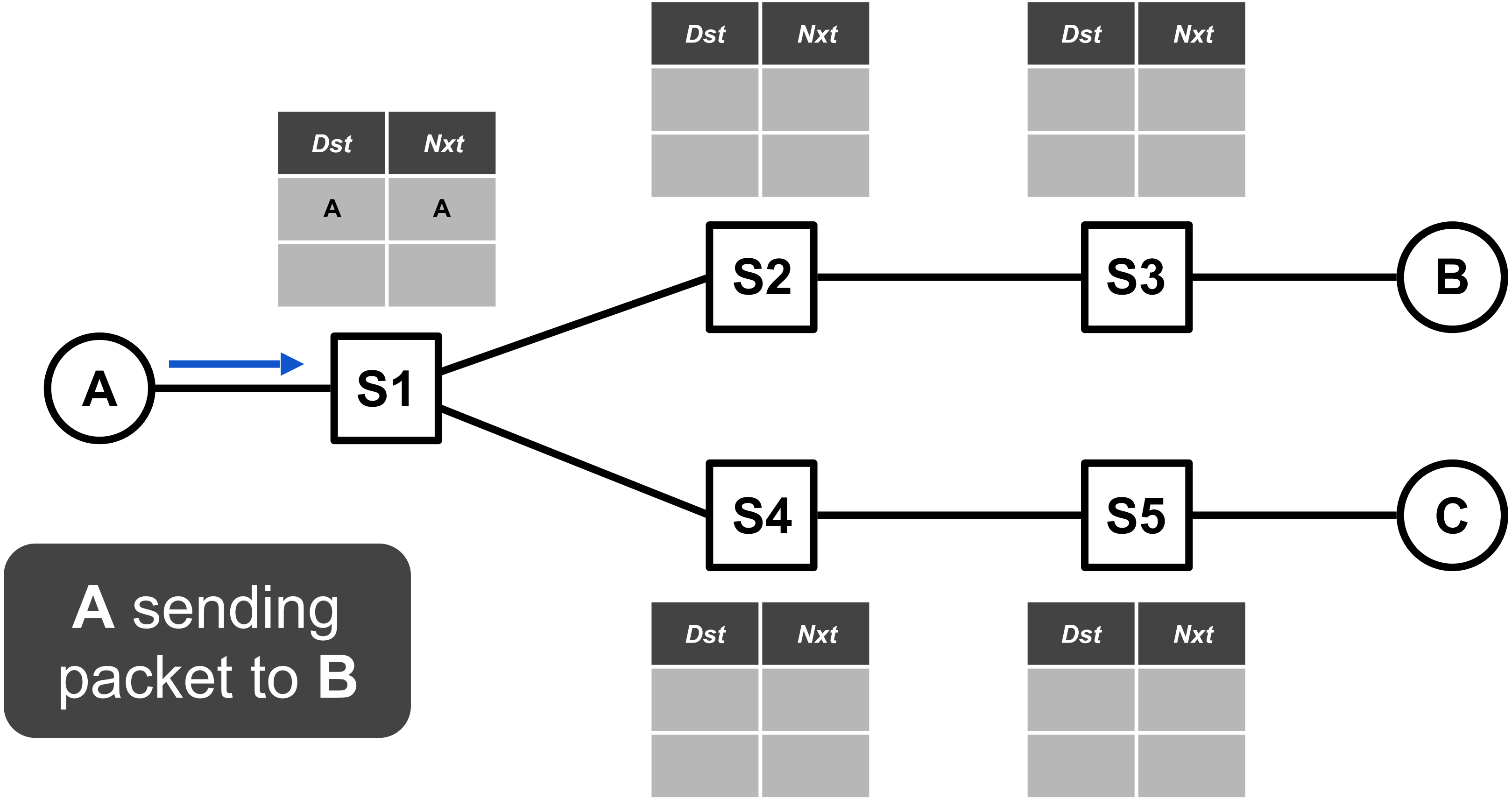
- Learning switches:
    - Tables filled in opportunistically using data packets
    - No “seeding” with static entries required!
    - Very common for routing at the link layer (L2)
        - Many people would say this isn’t routing
        - But it fills in tables to get packets from source to destination, so…

Learning Switches



A sending packet to B

Learning Switches



Dst	Nxt
A	A

Dst	Nxt

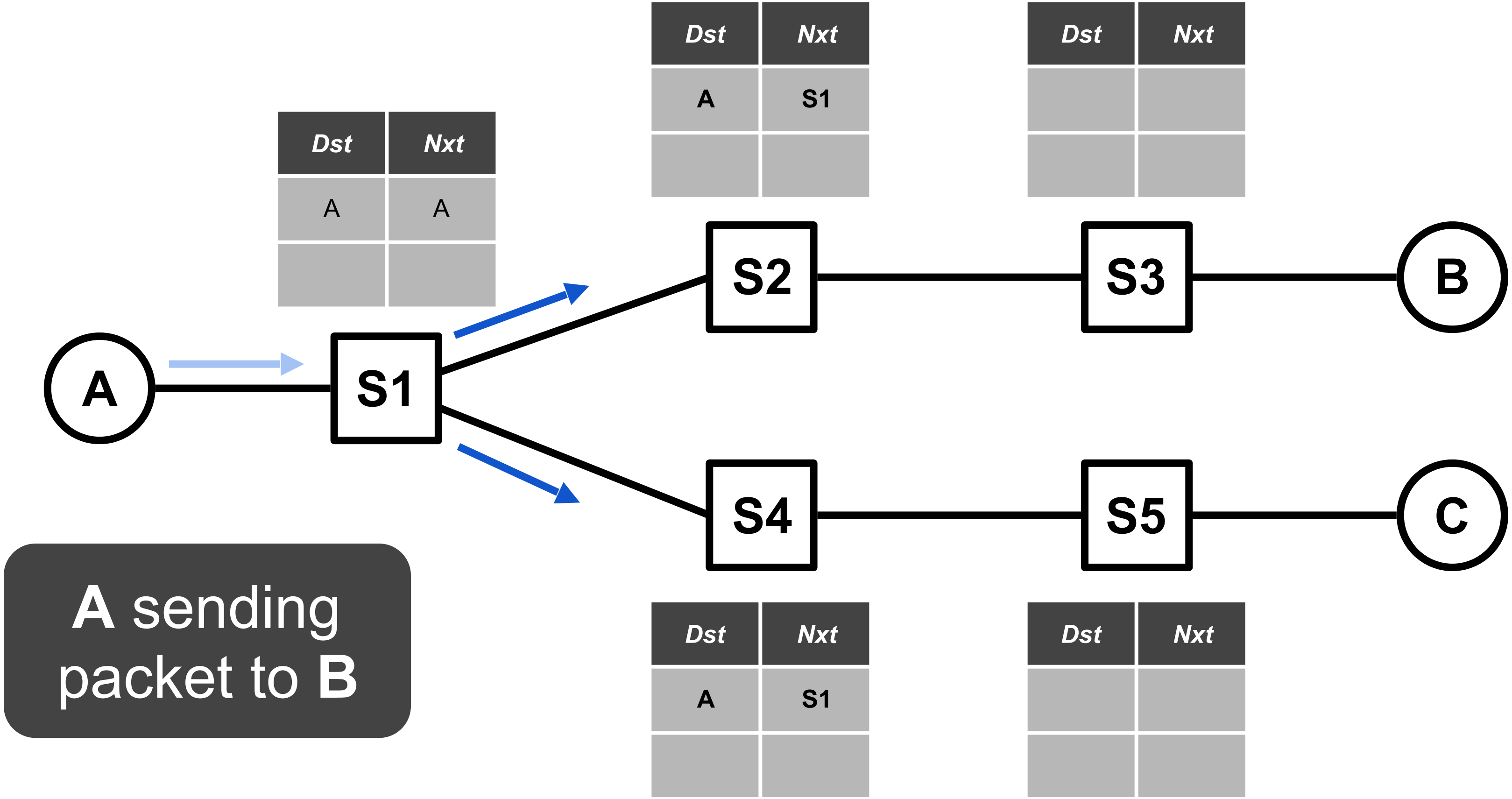
Dst	Nxt

Dst	Nxt

Dst	Nxt

A sending packet to B

Learning Switches



S1

Dst	Nxt
A	A

S2

Dst	Nxt
A	S1

S3

Dst	Nxt

S4

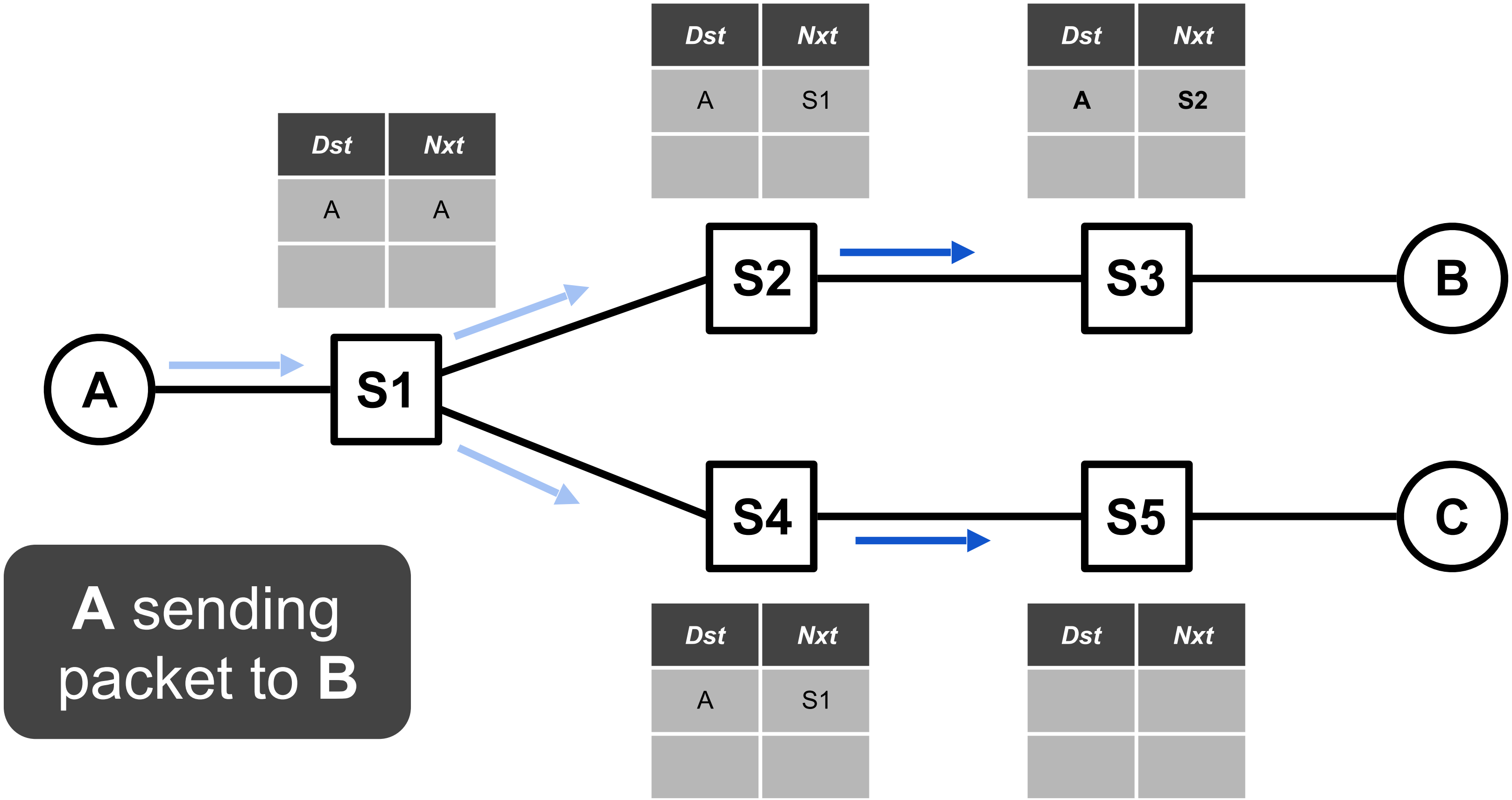
Dst	Nxt
A	S1

S5

Dst	Nxt

A sending packet to B

Learning Switches



S1:

Dst	Nxt
A	A

S2:

Dst	Nxt
A	S1

S3:

Dst	Nxt
A	**S2**

S4:

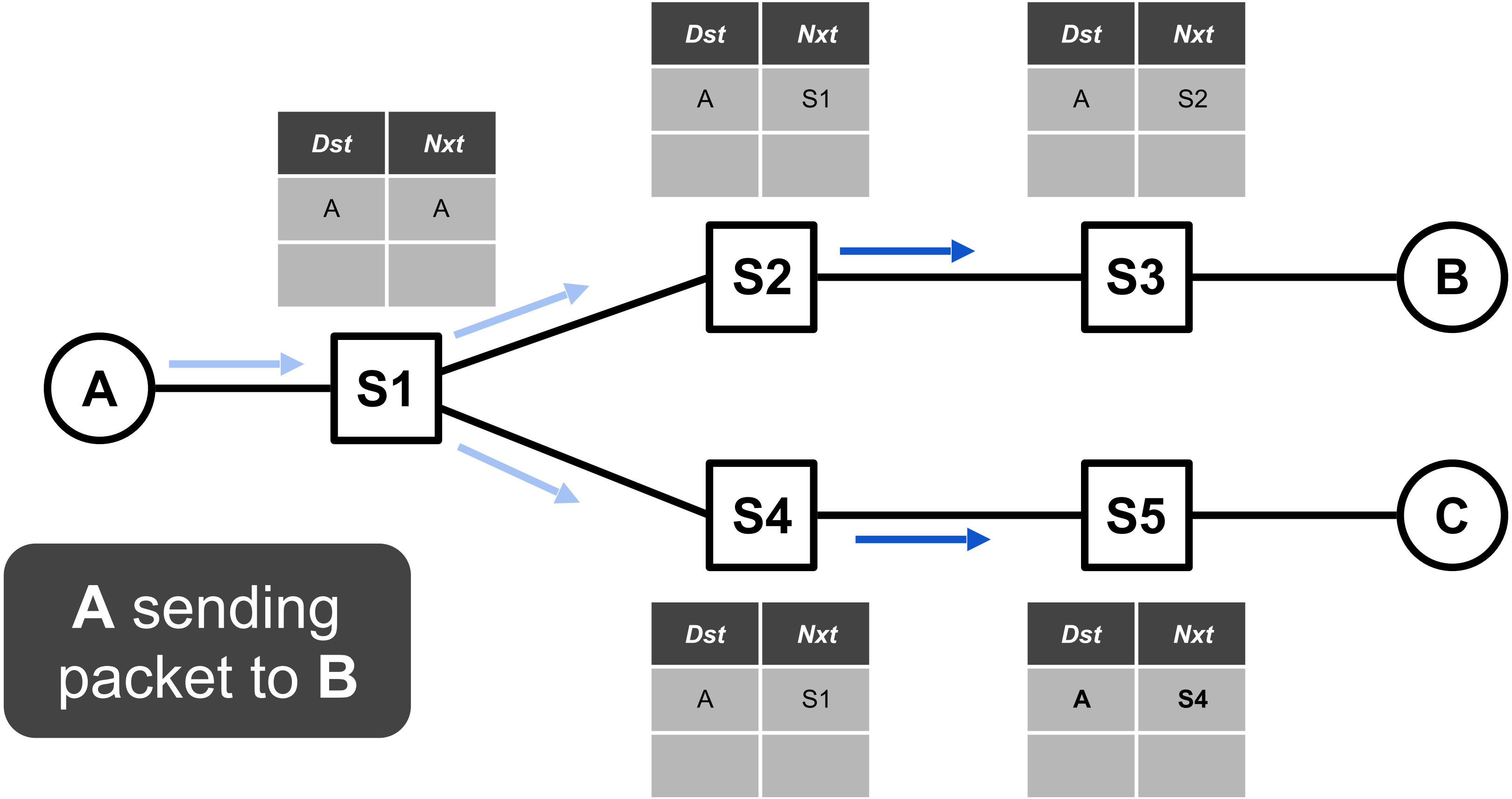
Dst	Nxt
A	S1

S5:

Dst	Nxt

A sending packet to **B**

Learning Switches



Dst	Nxt
A	A

Dst	Nxt
A	S1

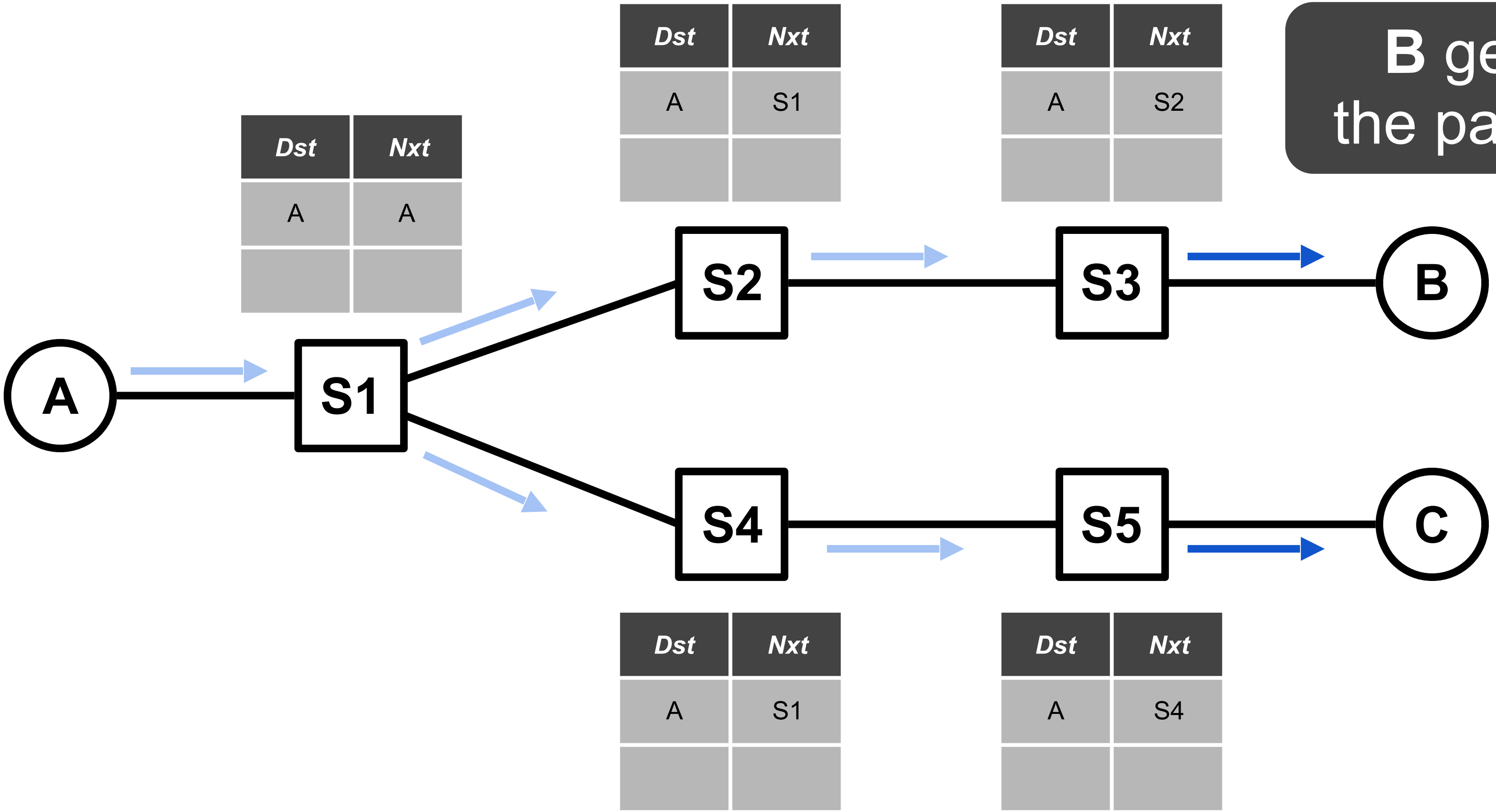
Dst	Nxt
A	S2

Dst	Nxt
A	S1

Dst	Nxt
A	**S4**

A sending packet to **B**

Learning Switches



Dst	Nxt
A	A

Dst	Nxt
A	S1

Dst	Nxt
A	S2

Dst	Nxt
A	S1

Dst	Nxt
A	S4

B gets the packet

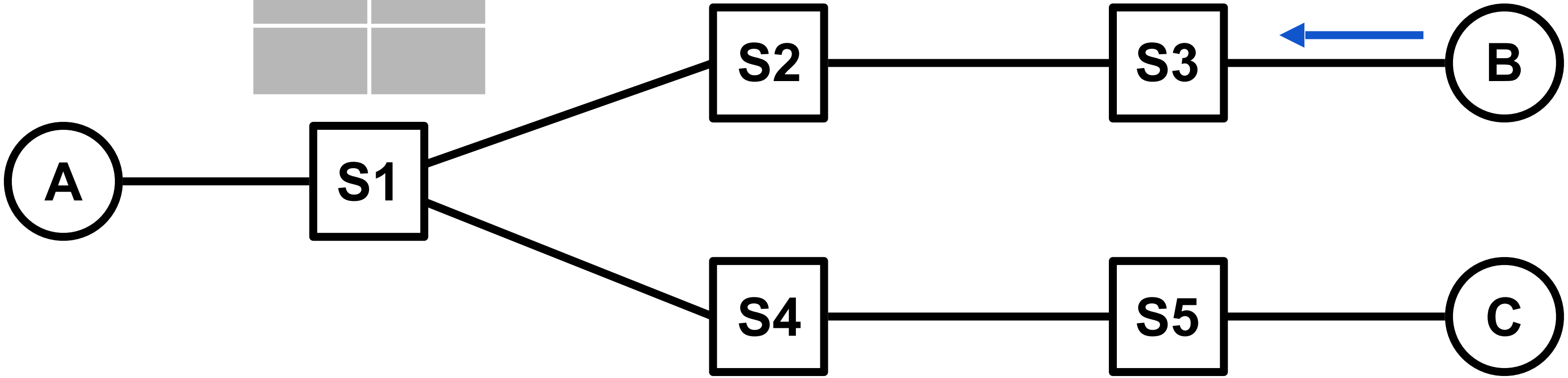
Learning Switches

B replies to A

Dst	Nxt
A	S1

Dst	Nxt
A	S2

Dst	Nxt
A	A

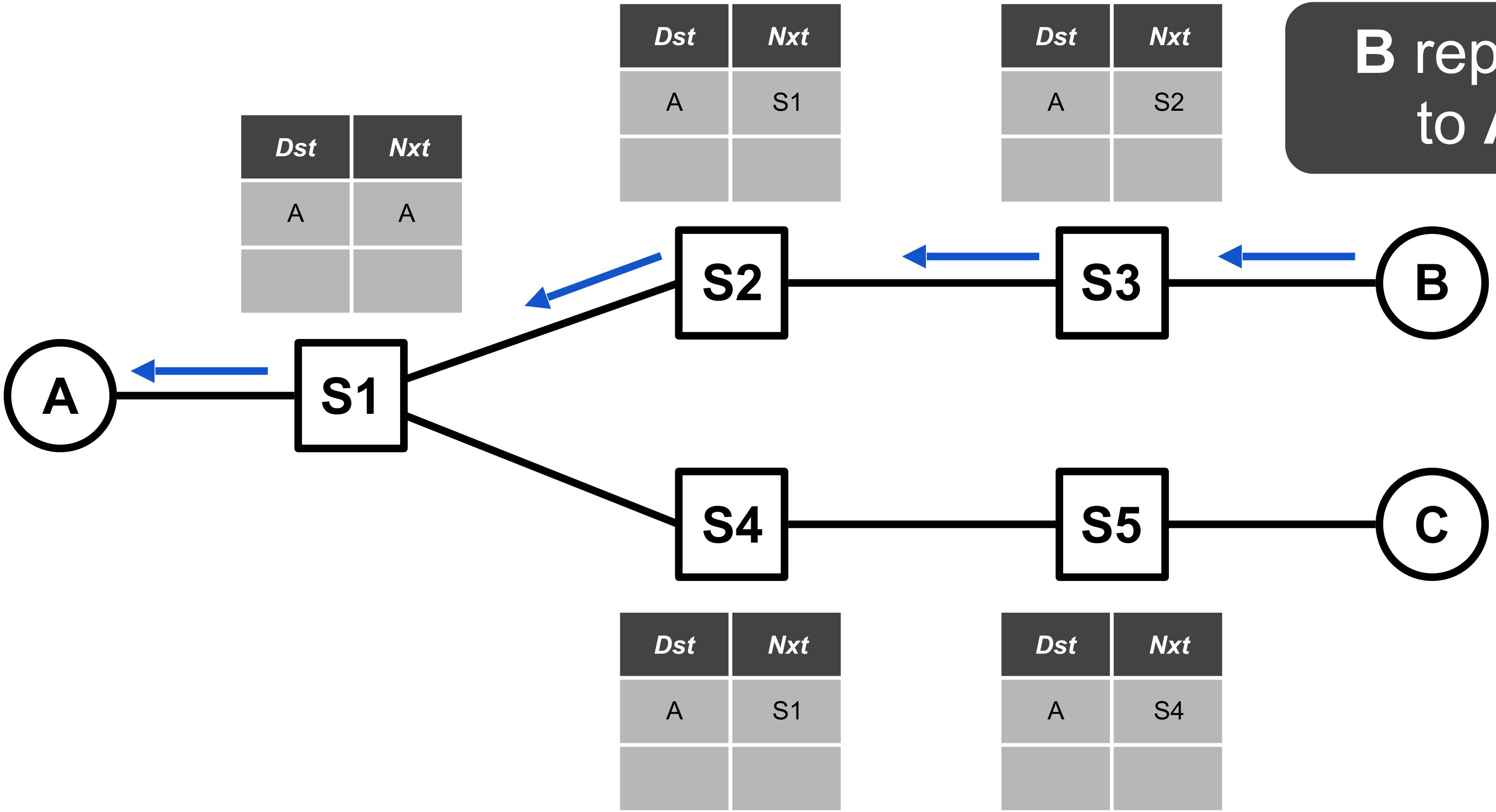


Dst	Nxt
A	S1

Dst	Nxt
A	S4

Learning Switches

B replies to A



Dst	Nxt
A	A

Dst	Nxt
A	S1

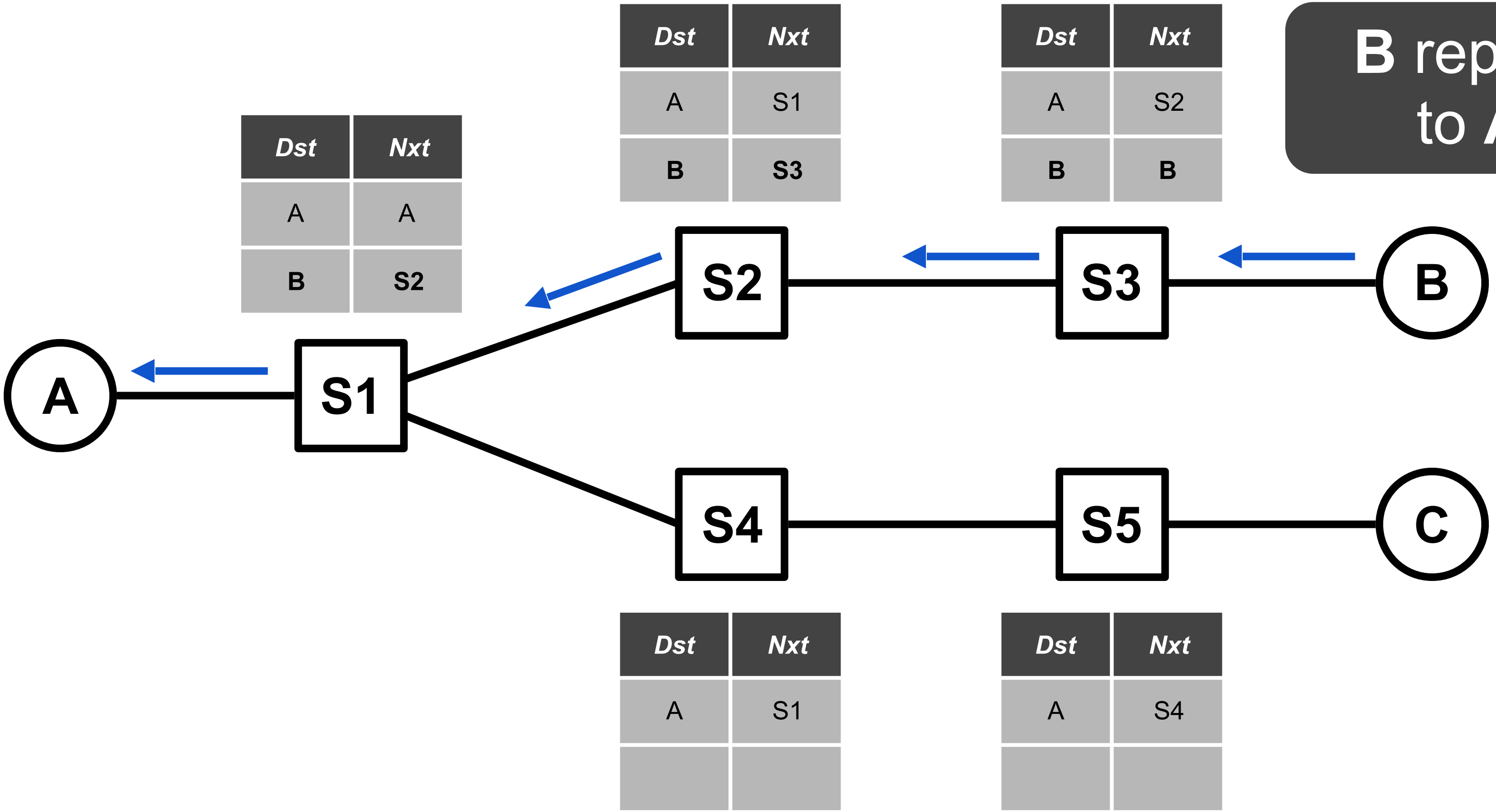
Dst	Nxt
A	S2

Dst	Nxt
A	S1

Dst	Nxt
A	S4

Learning Switches

B replies to A



Dst	Nxt
A	A
B	**S2**

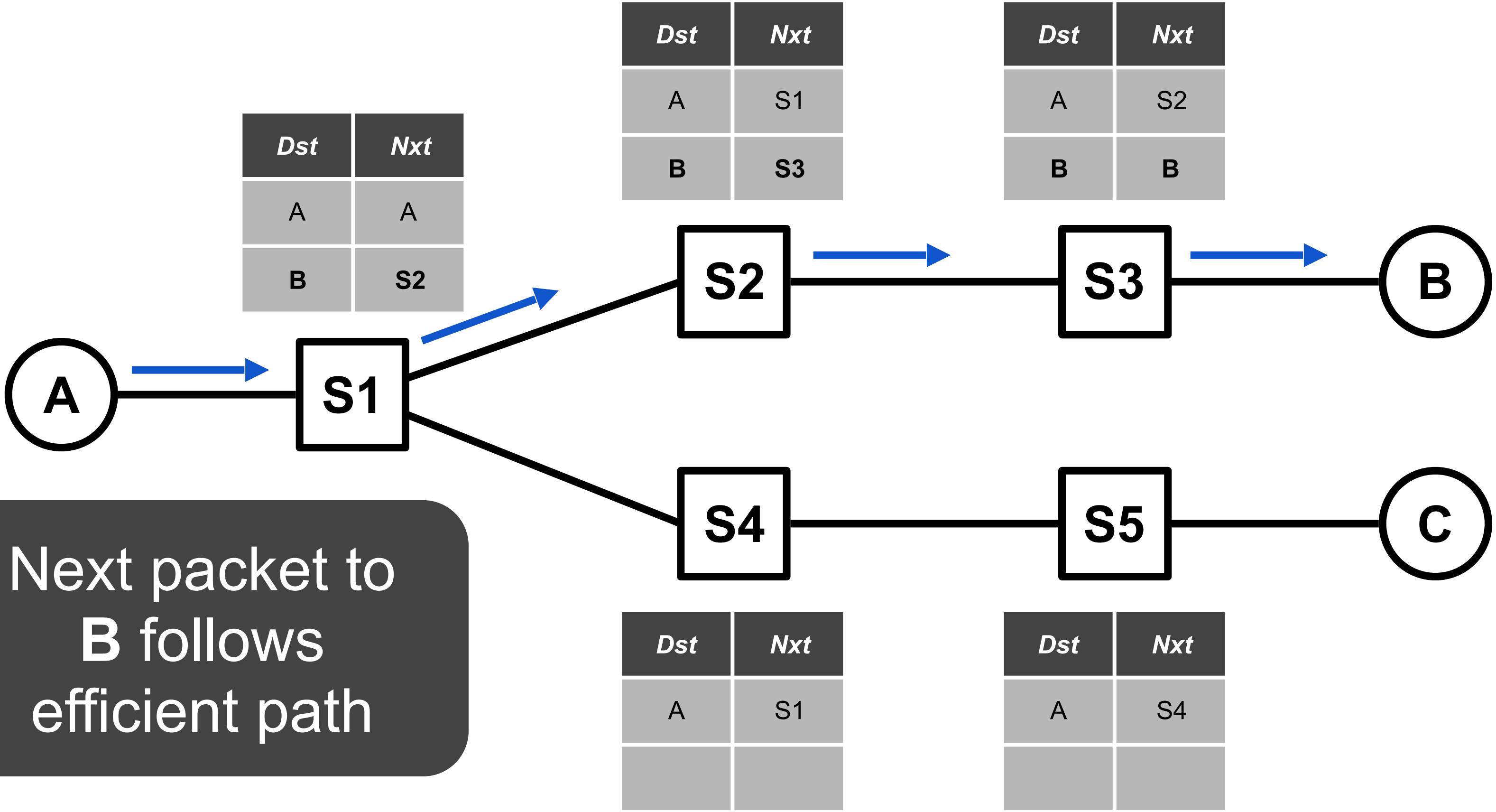
Dst	Nxt
A	S1
B	**S3**

Dst	Nxt
A	S2
B	**B**

Dst	Nxt
A	S1

Dst	Nxt
A	S4

Learning Switches



Dst	Nxt
A	A
B	**S2**

Dst	Nxt
A	S1
B	**S3**

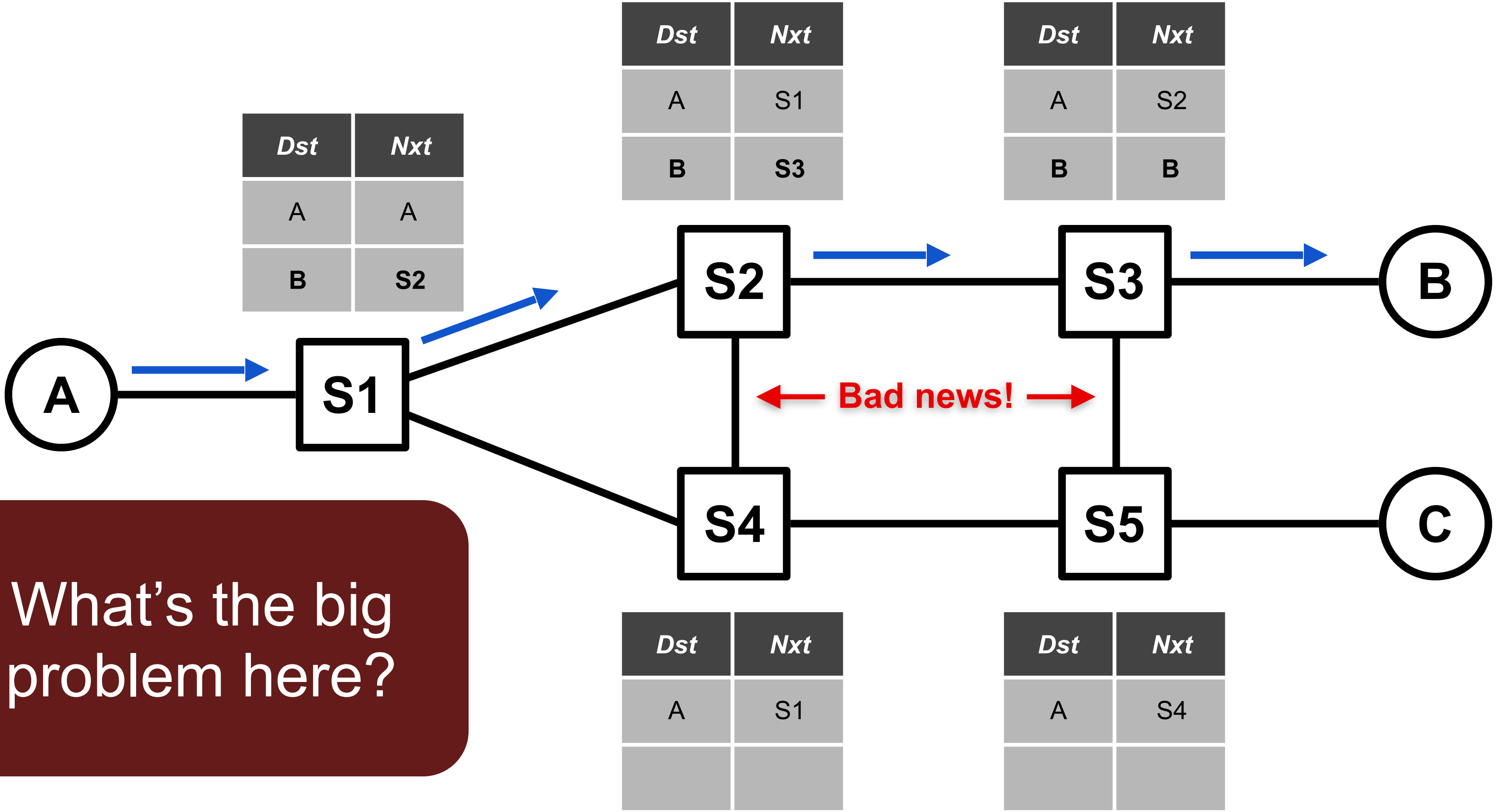
Dst	Nxt
A	S2
B	**B**

Dst	Nxt
A	S1

Dst	Nxt
A	S4

Next packet to **B** follows efficient path

Learning Switches



Dst	Nxt
A	A
B	**S2**

Dst	Nxt
A	S1
B	**S3**

Dst	Nxt
A	S2
B	**B**

Dst	Nxt
A	S1

Dst	Nxt
A	S4

What’s the big problem here?

Learning Switches

- Major problem with learning switches:
    - Floods when destination is unknown
    - .. floods have problems when topology has loops

- Our previous solution doesn’t work in this case
    - .. we’ll come back to this in just a second

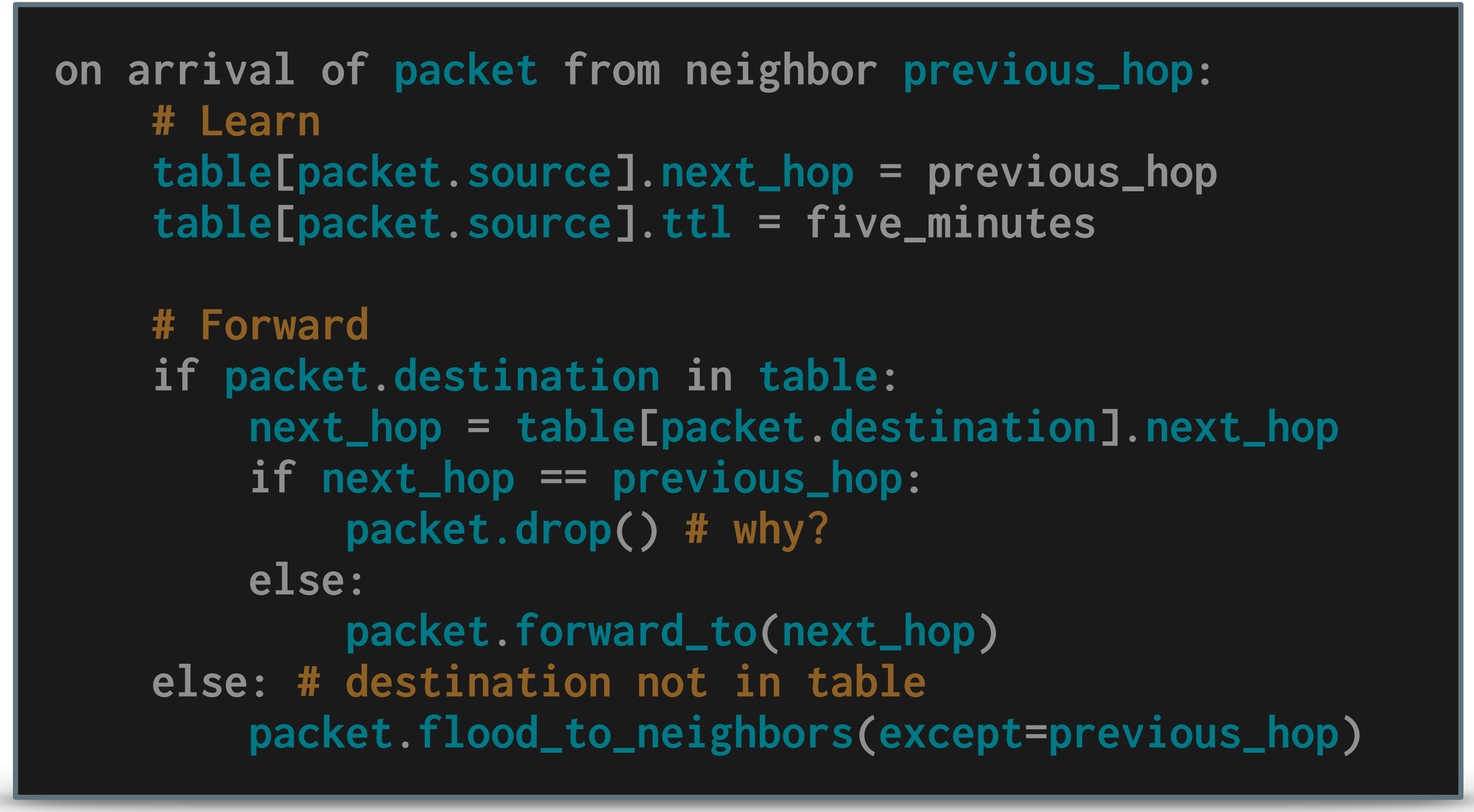
Learning Switches

- **Note: the decision to flood is done on a switch-by-switch basis…**

- Packets are not purely flooded or purely point-to-point throughout their lifetimes

- Instead, at each switch, packets are:
  - Sent out correct port if table entry exists
  - Flooded out all ports (except incoming) if not

Learning Switches: Pseudocode-Style



```
on arrival of packet from neighbor previous_hop:
    # Learn
    table[packet.source].next_hop = previous_hop
    table[packet.source].ttl = five_minutes

    # Forward
    if packet.destination in table:
        next_hop = table[packet.destination].next_hop
        if next_hop == previous_hop:
            packet.drop() # why?
        else:
            packet.forward_to(next_hop)
    else: # destination not in table
        packet.flood_to_neighbors(except=previous_hop)
```

Learning Switches

- Major problem with learning switches:
    - Floods when destination is unknown
    - .. floods have problems when topology has loops

- Our previous solution doesn’t work in this case

Learning Switches

- Major problem with learning switches:
  - Floods when destination is unknown
  - .. floods have problems when topology has loops

- Our previous solution doesn’t work in this case
  - Old solution kept state for each sender (the highest sequence number)
    - Worked okay for number of internal routers in a network…
    - .. but probably does not scale to number of hosts on Internet!
    - .. and data packets don’t necessary have a sequence number anyway!

- New solution:
  - Disable links until there are no loops (make it into a *spanning tree*)!

Spanning Tree Protocol

- How do you make a spanning tree from an arbitrary network?
  - Step 1: Find least cost paths from every switch to the root
  - Step 2: Disable data delivery on every link not on a path to root
  - Step 3: When the tree breaks (a link on it fails), start over

Spanning Tree Protocol: Step 1 (Paths to root)

- Step 1: Find least cost paths from every switch to the root
- Wait; do we already have an algorithm/protocol that does this?
- Spoiler alert: Step 1 of STP is basically D-V with a single table entry/destination
    - No split horizon or poison reverse
    - The “destination” is the switch at the root of the tree
    - Every switch has a unique, orderable ID (based on Ethernet address)
    - We simultaneously work to find:
        - The root (switch with lowest ID)
        - The best path to the root (lowest cost)

Spanning Tree Protocol: Step 1 (Paths to root)

- All switches begin by thinking they are the root
- Advertises “route” to itself (“The root is **`my_id`** and I can reach it in **`zero`** hops”)

- Compare distances like `(distance, next_hop_id)` (i.e., using `id` to break ties)

- On receiving a “route” (STP message) from a neighbor:
    - First, *compare the advertised root ID* to what we think root ID is…
    - If it’s smaller than current, it is a better root: use it as root
    - If it’s larger than current, it is a worse root: ignore it
    - If it’s the same: Basically normal D-V update rules (minimize distance)
        - Except: Break ties by preferring next hop with smaller ID as shown above!
    - .. and send *triggered* update if your own state changes

- Only generate *periodic* advertisements if you think you’re the root
    - Other nodes just forward advertisements to neighbors farther than they are

Spanning Tree Protocol: Step 2 (Disable links)

- Step 2: Disable data delivery on every link not on a shortest path to root
- Remember: A neighbor is either *closer* to root or *farther* from root than you
  - No distance ties — broken using unique IDs
- Each switch:
  - ***Enables*** the link along the best path to the root
  - ***Disables*** every other link to a neighbor closer to the root
  - Lets the further-away neighbors decide the rest!
  - (Also enables all links to hosts!)

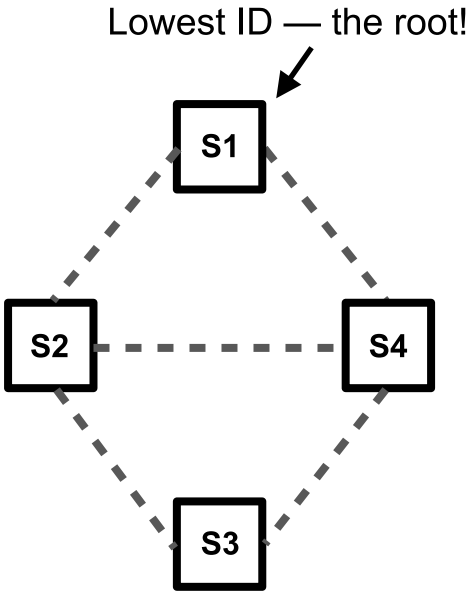
Spanning Tree Protocol: Step 2 (Disable links)

- Step 2: Disable data delivery on every link not on a shortest path to root

- Wait; why is this so complicated?
  - Maybe it's not as easy as you think…

- A switch knows which link is part of *its own* shortest path to the root
  - Definitely enable that one!
- .. but how does it know which of its links are part of *another* switch's path to root?
  - It better not disable those!
  - .. how does S4 know if it is on S3's best path?

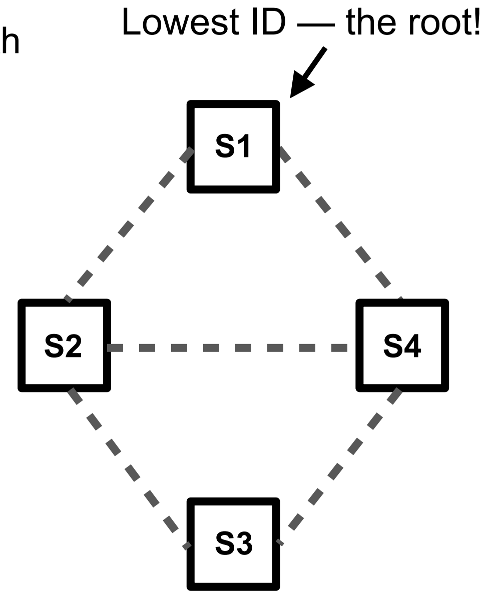
- Observations:
  - If neighbor is closer to root than I am, I can't be on its shortest path
  - If neighbor is farther from root than I am, I ***might*** be on its shortest path
  - You know everyone's distance from the root along the tree because that's what the advertisements tell you!



Spanning Tree Protocol: Step 2 (Disable links)

- Observations:
  - If neighbor is closer to root than I am, I can’t be on its shortest path
  - If neighbor is farther from root than I am, I ***might*** be on its shortest path
    - e.g., again, S4 doesn’t know if it is on S3’s best path
  - You know everyone’s distance from the root along the tree because that’s what the advertisements tell you!

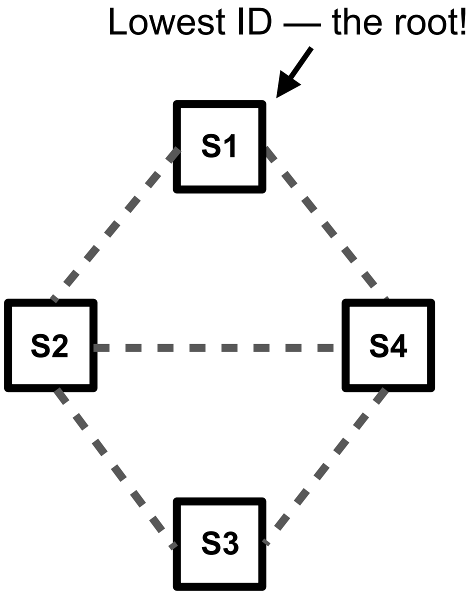
- Strategy:
  - **Enable** link along your best path to root
  - **Disable** other links to switches closer to root than you
    - .. they’re not on your best path
    - .. and you can’t possibly be on theirs (you're father!)
  - Leave other links for other switches to decide
    - .. they’re all farther from root than you are
    - .. so you're closer than they are
    - .. so the above enable/disable rules work for them



Spanning Tree Protocol: Step 2 (Disable links)

- Strategy:
  - **Enable** link along your best path to root
  - **Disable** other links to switches closer to root than you
    - .. they're not on your best path
    - .. and you can't possibly be on theirs
  - Leave other links for other switches to decide
    - .. they're all farther from root than you are
    - .. so you're closer than they are
    - .. so the above enable/disable rules work for them

- .. but what about switches of equal distance? (e.g., S2 & S4)
  - Can't possibly be on each other's shortest paths
  - .. but only one should determine link enable/disable
  - .. so break distance ties using switch ID
  - .. S4 & S2 are both distance 1 from root… break tie with ID… S4 has bigger ID so it's "farther"… so it decides for S2—S4 link

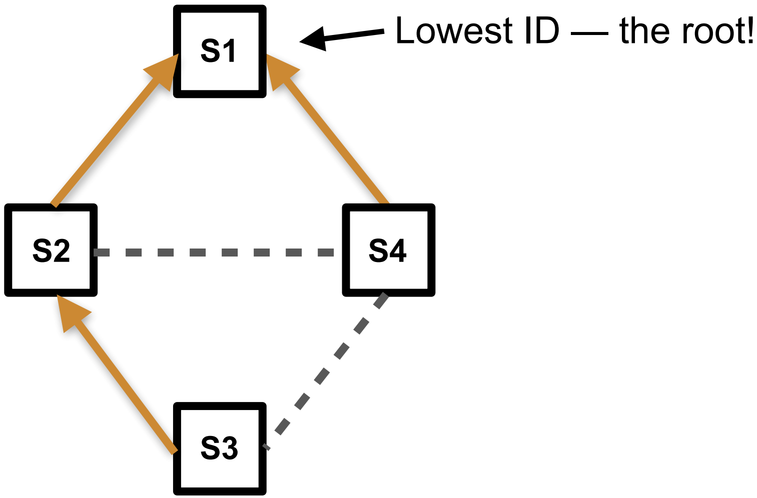


Spanning Tree Protocol: Step 2 Example

- Gray dashed links unknown
- Black links enabled
- Red messy links disabled

- S1 is the root

- Assume all switches have completed step 1 already ("next hops" shown here)

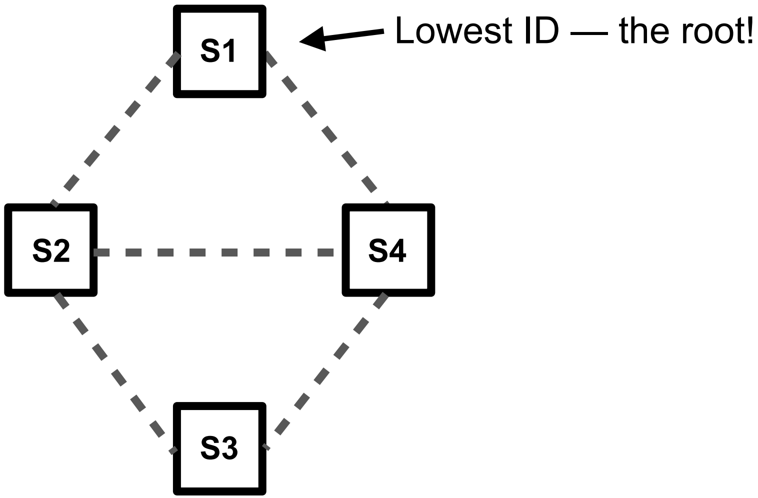


Spanning Tree Protocol: Step 2 Example

- Gray dashed links unknown
- Black links enabled
- Red messy links disabled

- S1 is the root

- Assume all switches have completed step 1 already



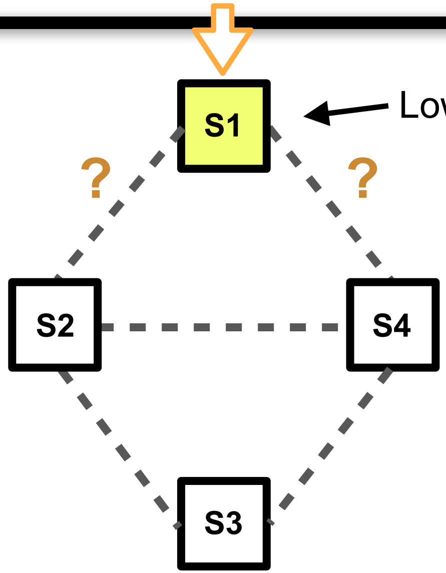
Enabled: Link on best path to root
Disabled: Links to other neighbors “closer” to root
Unknown: Links to neighbors “farther” from root

Remember: Break distance ties using IDs!

Spanning Tree Protocol: Step 2 Example

- **S1's Perspective**

- S1-S2: Unknown
- S1-S4: Unknown



Lowest ID — the root!

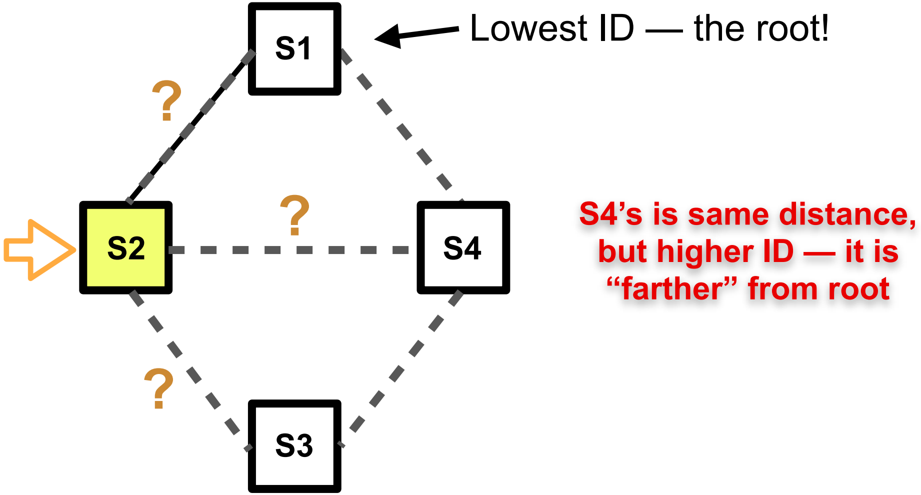
Enabled: Link on best path to root
Disabled: Links to other neighbors "closer" to root
Unknown: Links to neighbors "farther" from root

Remember: Break distance ties using IDs!

Spanning Tree Protocol: Step 2 Example

- **S2's Perspective**

- S2-S1: Enabled
- S2-S3: Unknown
- S2-S4: Unknown



Lowest ID — the root!

S4's is same distance, but higher ID — it is "farther" from root

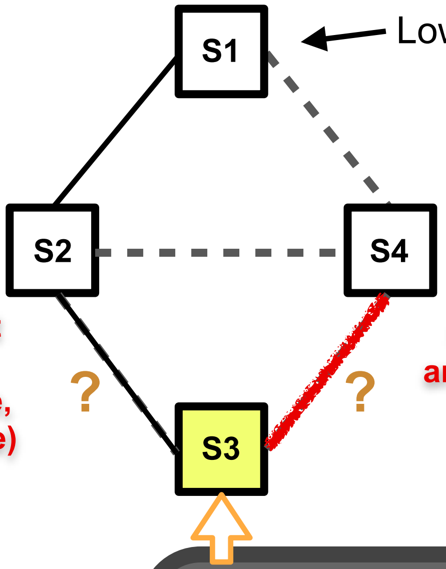
Enabled: Link on best path to root
Disabled: Links to other neighbors "closer" to root
Unknown: Links to neighbors "farther" from root

Remember: Break distance ties using IDs!

Spanning Tree Protocol: Step 2 Example

- **S3's Perspective**

- S3-S2: Enabled
- S2-S4: Disabled



Lowest ID — the root!

S2 is "closer" to root than S4 (same actual distance, but lower ID breaks tie)

S4 is closer than we are, but not on shortest path — disable!

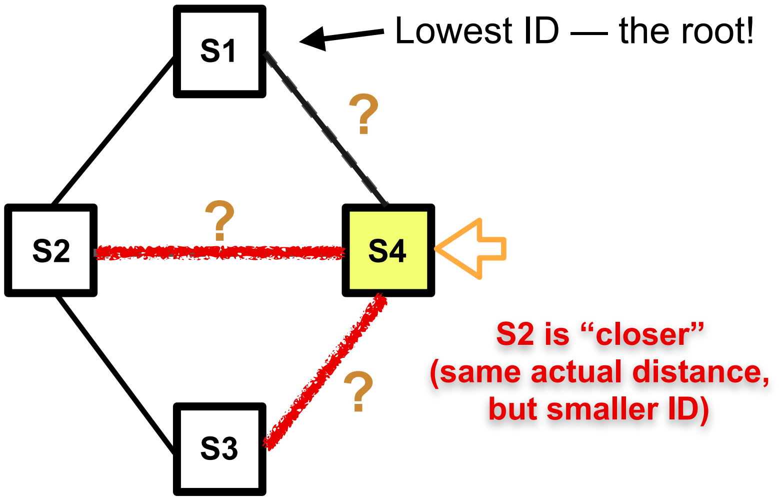
Enabled: Link on best path to root
Disabled: Links to other neighbors "closer" to root
Unknown: Links to neighbors "farther" from root

Remember: Break distance ties using IDs!

Spanning Tree Protocol: Step 2 Example

- **S4's Perspective**

- S4-S1: Enabled
- S4-S3: Unknown (leave alone)
- S4-S2: Disabled

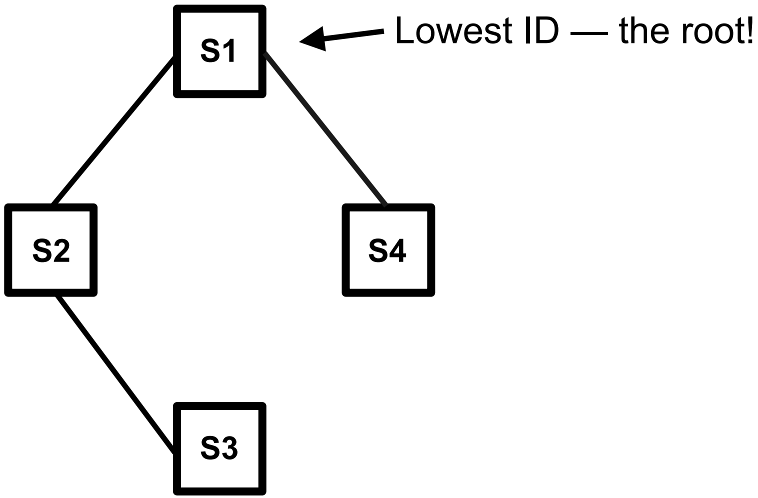


Enabled: Link on best path to root
Disabled: Links to other neighbors "closer" to root
Unknown: Links to neighbors "farther" from root

Remember: Break distance ties using IDs!

Spanning Tree Protocol: Step 2 Example

- We've got a spanning tree!
- .. and it matches the next hops each switch came up with!



Enabled: Link on best path to root
Disabled: Links to other neighbors "closer" to root
Unknown: Links to neighbors "farther" from root

Remember: Break distance ties using IDs!

Spanning Tree Protocol: Step 2 (Disable links)

- Step 2 Recap…

- No ties when comparing distance — break ties using switch IDs

- Each switch:
  - ***Enables*** the link along the best path to the root (and all links to hosts!)
  - ***Disables*** every other link to a neighbor closer to the root
  - Lets the further-away neighbors decide the rest!

- .. in this way, a switch closer doesn’t disable a link needed by a switch that’s farther
  - .. doesn't require explicit coordination (no need to ask, “do you need this link?”)
  - .. exactly one switch responsible for enabling/disabling each link

Spanning Tree Protocol: Step 3

- Step 3: When the tree breaks (a link on it fails), start over
- If “route” expires, pretend you’re the root again
    - You’ll (hopefully) get messages from neighbors
    - You’ll all sort out new links and possibly a new root!

STP & Learning Switches: Summary

- STP is basically distance-vector at its core

- .. except you are always only figuring out the route to the root (lowest ID switch)
  - (A single tree, not a single tree per destination!)
- .. and you don't use the "routes" for forwarding directly
- .. instead, disable links between switches which *aren't* on a shortest path to root

- After disabling links, topology is *logically* a tree
- .. learning switches can flood freely on that tree
- .. and you can learn table entries from data packets moving along tree

STP & Learning Switches: Summary

- Only used in local (layer 2) networks
  - Bandwidth is plentiful, number of nodes relatively small
  - So flooding is feasible

- Flooding lets you reach destinations even without routing information
  - You don’t *need* table entries (static or from routing protocol)
  - (But they’re nice!)

- Flooding can “find” hosts
  - No need for static routes

- Once a switch has seen a packet from a host, it has a table entry for it
  - If all switches see packet from host, no more need to flood when it is destination

Questions?

A Final Thing about STP

Algoryhme by Radia Perlman

I think that I shall never see
A graph more lovely than a tree.

A tree whose crucial property
is loop-free connectivity.

A tree that must be sure to span
so packets can reach every LAN.

First, the root must be selected.
By ID, it is elected.

Least-cost paths from root are traced.
In the tree, these paths are placed.

A mesh is made by folks like me,
Then bridges find a spanning tree.

See Also
"Trees"
by American poet Joyce Kilmer
1913

LAN ≈ L2 network (Local Access Network)

mesh ≈ a graph with high degree of connectivity
bridge ≈ switch

Addressing

(and a bit of IGP/EGP interplay)

Addressing

- How do routing and forwarding scale to the size of the Internet?!

- Can I really have a table entry for every host?
- How long would it take for D-V to converge this distributed algorithm when you have propagation delays brought about by the speed of light?
- Can a L-S router really build/maintain a graph for the entire Internet?

- I’ve mentioned that intradomain & interdomain routing use different protocols
- We’ve mostly talked about intra so far (IGPs); inter next week (BGP the EGP)
- .. maybe the magic of scaling shows up in the interdomain routing protocols?

- Actually, the scaling is mostly about *addressing*

Addressing

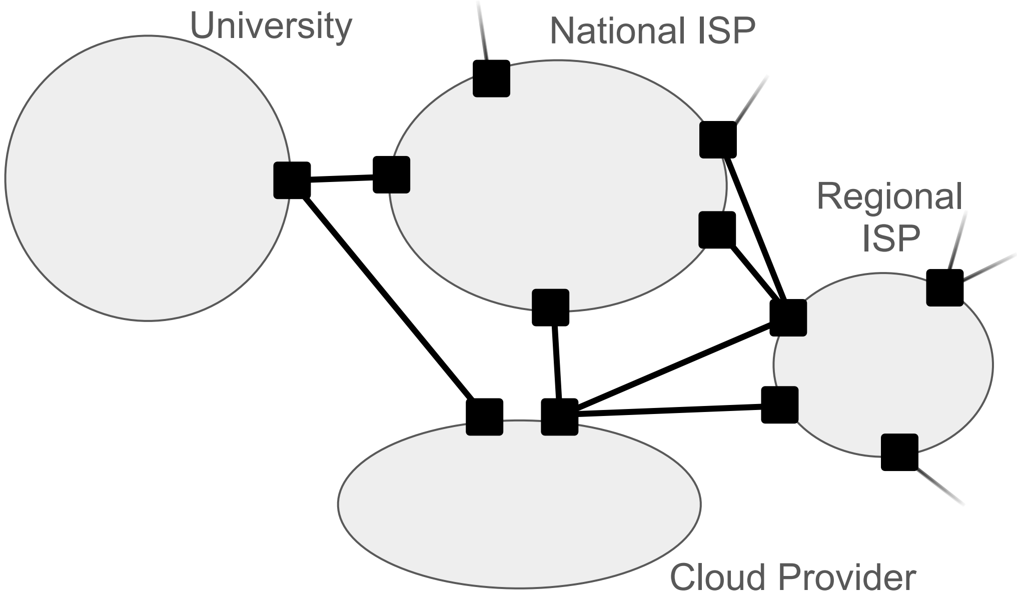
- IP addresses are part of what makes IP scalable
- We'll focus on IPv4 addresses
  - IPv6 is pretty similar; we don't focus too much on it in this class

- Without talking about details of BGP, I will also touch on how intradomain and interdomain routing protocols interact

- I am *not* going to talk about Layer 2 addresses today (Ethernet addresses)
  - They work differently; probably better name would be Ethernet *identifiers*
  - They don't need to scale as much (though bigger than people thought…)
  - They'll probably come up later in the semester

Addressing: Early Internet

- Remember, the Internet is a network or networks



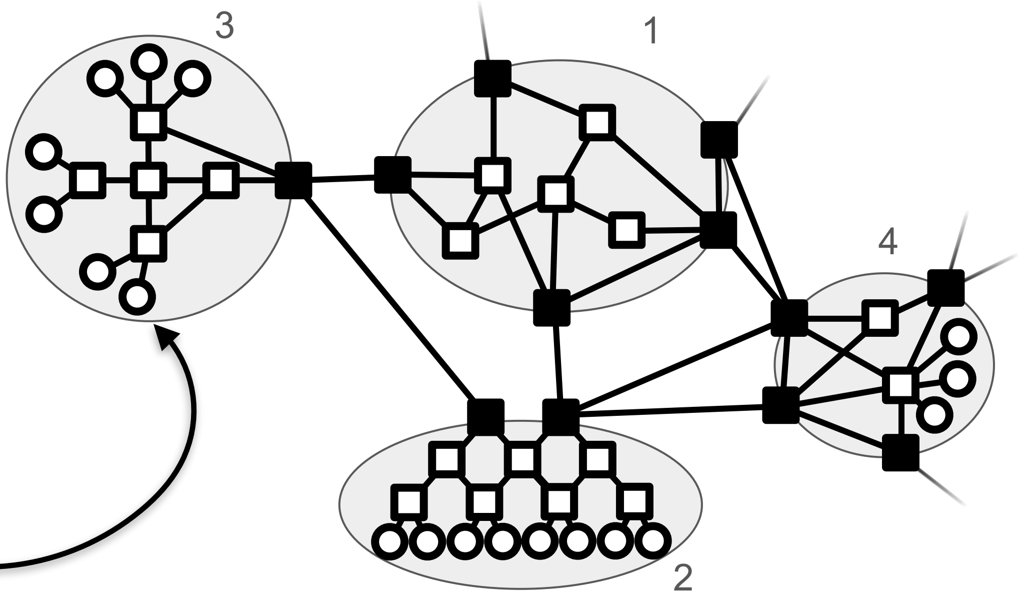
Addressing: Early Internet

- Remember, the Internet is a network or networks
  - Leads naturally to a two level hierarchy
  - .. and hierarchy is one of the major tools to address scaling!

- Could imagine hierarchical addressing scheme…

  - Hosts have identifiers
  - Networks have identifiers

  - Address is like: *Network.Host*
    - This could be 3.7

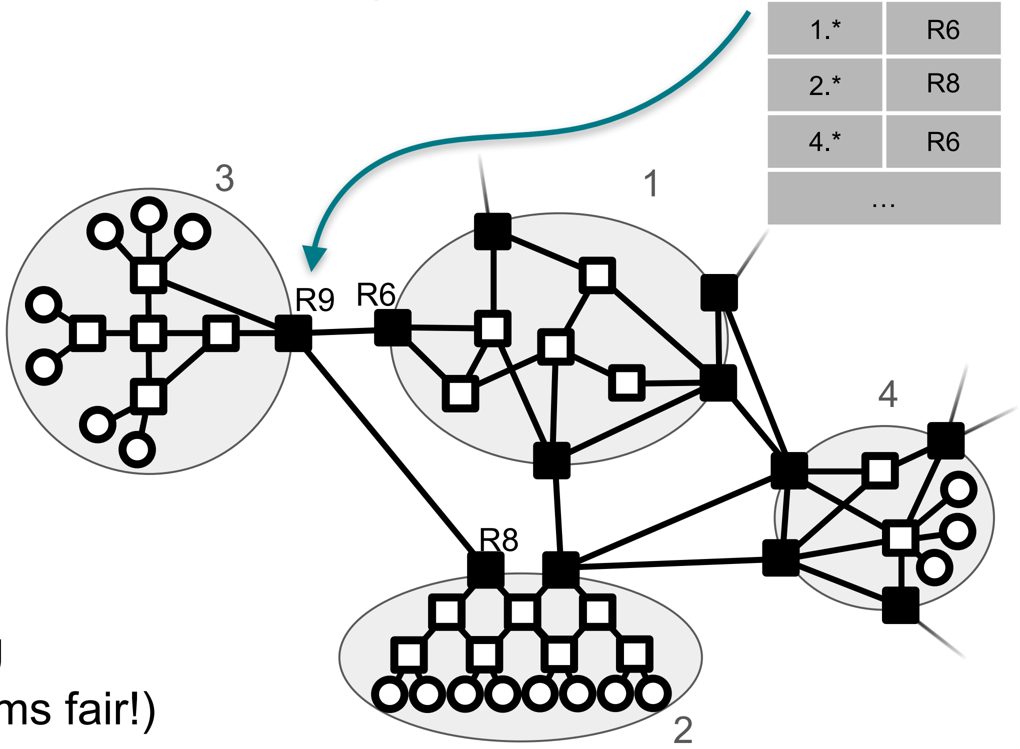


Hierarchical Addressing Implications

□ Internal router
■ Border router

- Routing between domains only concerned with network part
- Interdomain routing protocol only deals with four nodes!
- Limits table size & routing state
- Limits *churn*
  - Links added/failed inside domains generally has no effect; require no messages
- Big scalability improvement assuming many more hosts than networks (seems fair!)

R9's Table	
Dst	***Nxt***
1.*	R6
2.*	R8
4.*	R6
...	



Hierarchical Addressing Implications

□ Internal router
■ Border router

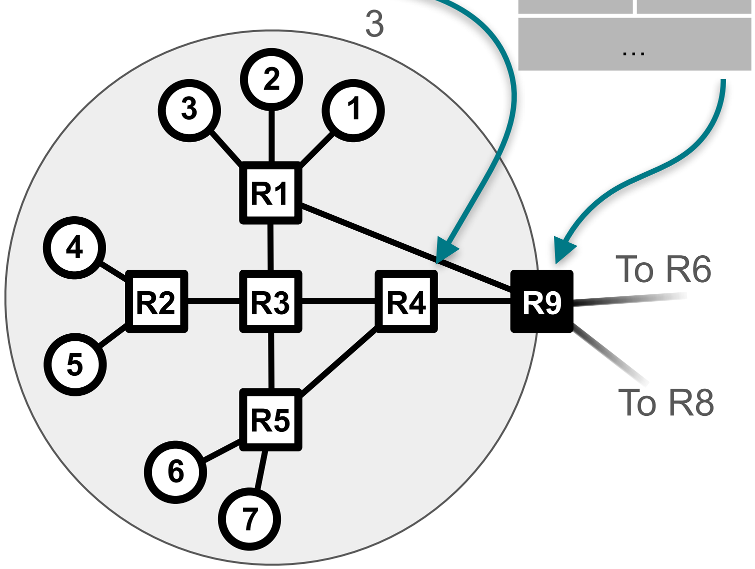
- Internal routers need routes for all hosts in *same* network…
  - Scales with number of hosts in single network

R4's Table

Dst	Nxt
3.1	R3
3.2	R3
3.3	R3
3.4	R3
3.5	R3
3.6	R5
3.7	R5

R9's Table

Dst	Nxt
1.*	R6
2.*	R8
4.*	R6
…	



Hierarchical Addressing Implications

□ Internal router
■ Border router

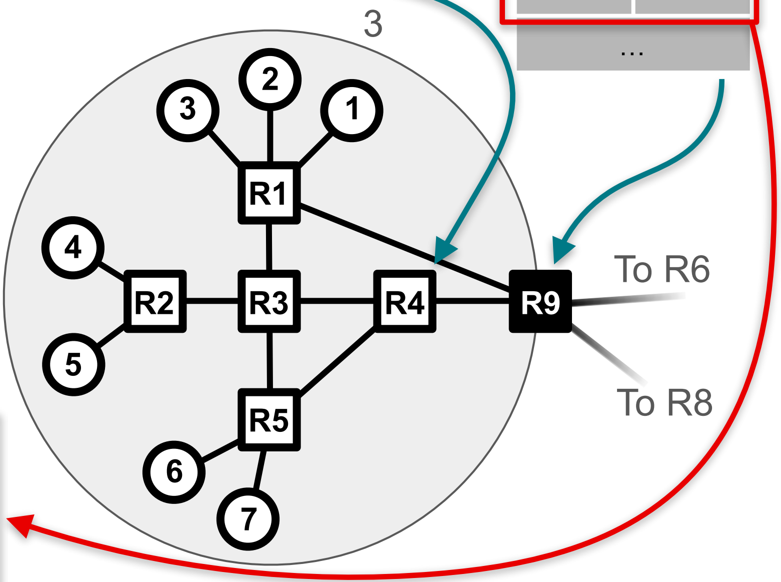
- Internal routers need routes for all hosts in *same* network…
  - Scales with number of hosts in single network
- .. *and* routes for other networks

R4's Table

Dst	*Nxt*
3.1	R3
3.2	R3
3.3	R3
3.4	R3
3.5	R3
3.6	R5
3.7	R5
1.*	**?**
2.*	**?**
4.*	**?**

R9's Table

Dst	*Nxt*
1.*	R6
2.*	R8
4.*	R6
…	

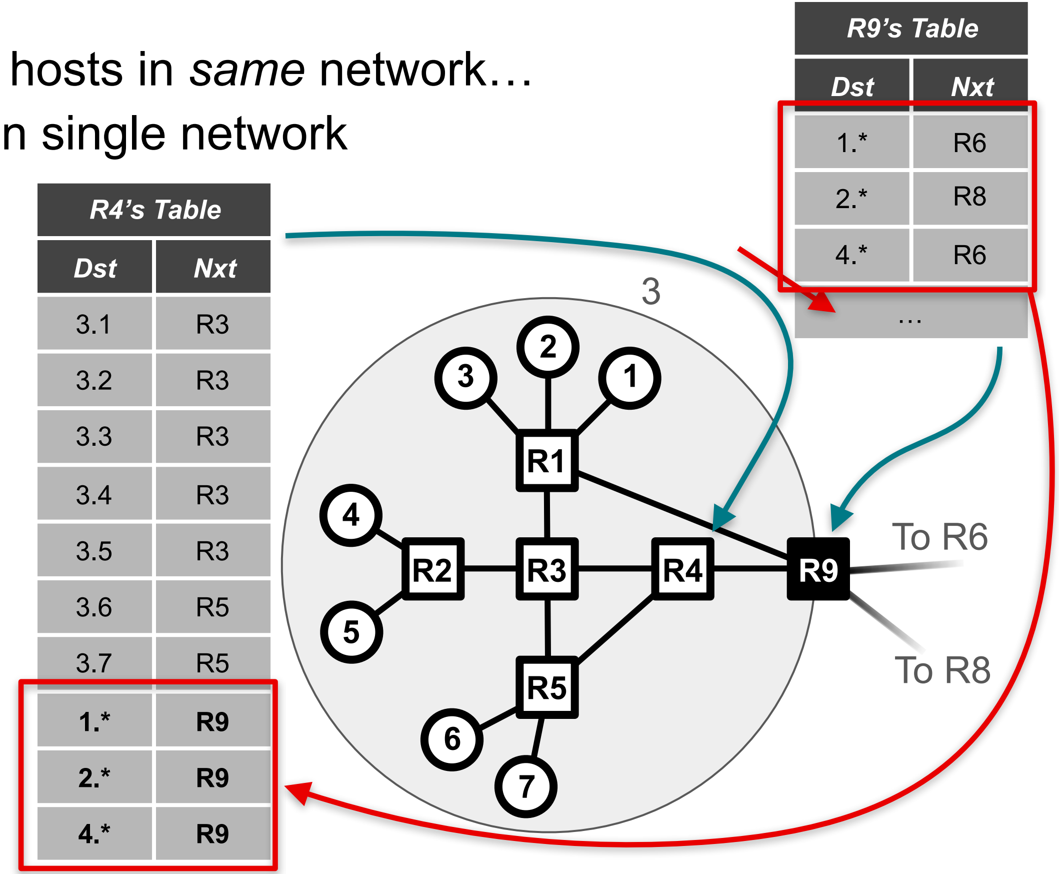


Hierarchical Addressing Implications

□ Internal router
■ Border router

- Internal routers need routes for all hosts in *same* network…
  - Scales with number of hosts in single network

- .. *and* routes for other networks



R4's Table	
Dst	***Nxt***
3.1	R3
3.2	R3
3.3	R3
3.4	R3
3.5	R3
3.6	R5
3.7	R5
1.*	**R9**
2.*	**R9**
4.*	**R9**

R9's Table	
Dst	***Nxt***
1.*	R6
2.*	R8
4.*	R6
…	

Hierarchical Addressing Implications

□ Internal router
■ Border router

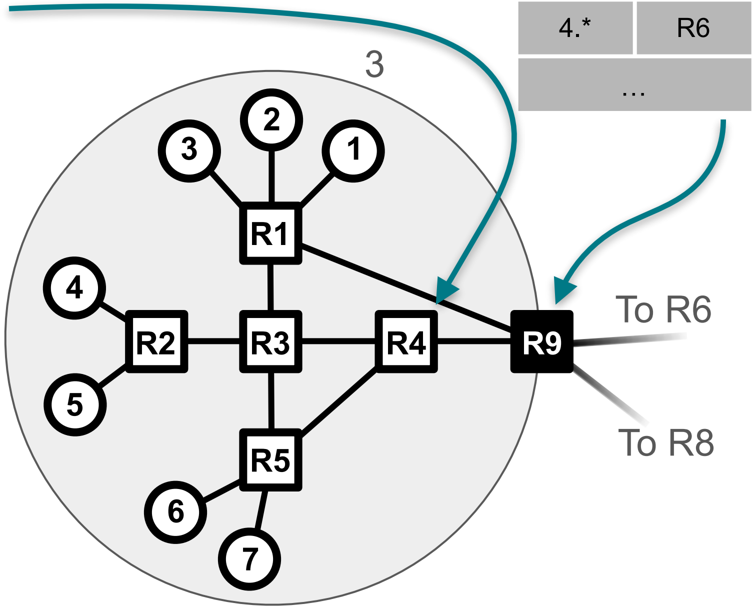
- Internal routers need routes for all hosts in *same* network…
  - Scales with number of hosts in single network
- .. *and* routes for other networks
- So total state scales with number of *hosts in this network* plus number of *other networks*
- Again: big scalability improvement assuming many more hosts than networks!

R4's Table

Dst	*Nxt*
3.1	R3
3.2	R3
3.3	R3
3.4	R3
3.5	R3
3.6	R5
3.7	R5
1.*	**R9**
2.*	**R9**
4.*	**R9**

R9's Table

Dst	*Nxt*
1.*	R6
2.*	R8
4.*	R6
…	

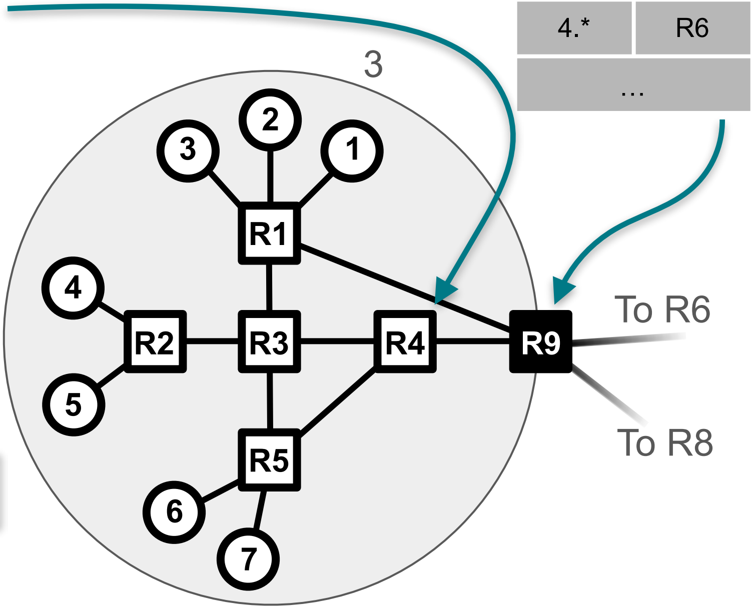


Hierarchical Addressing Implications

□ Internal router
■ Border router

R9's Table	
Dst	Nxt
1.*	R6
2.*	R8
4.*	R6
…	

R4's Table	
Dst	Nxt
3.1	R3
3.2	R3
3.3	R3
3.4	R3
3.5	R3
3.6	R5
3.7	R5
.	**R9**



Sidenote: You don't even *need* individual network routes in all the internal routers.

Since we only have one way to get to anywhere else in this network, we could just have a *default route*.

Hierarchical Addressing Implications

- Note that addresses aren’t assigned randomly!
- Hosts that are “close to each other” (in some sense) share part of their address
- We leverage this structure to make routing (and forwarding) scale better

- We use structured addresses like this all the time!
    - Soda Hall #417 is much easier to work with than if we just numbered every office in the world uniquely…

- This also explains why hosts don’t generally participate in routing protocols…
    - A human decided how to divide up the network in a way that makes sense
    - Your computer doesn’t have its own IP address wherever it goes…
    - .. it changes it address depending on where it is
    - .. it “moves in” to the network where it’s attached (and gets a new address there)

Hierarchical Addressing Implications Recap

- Assuming addresses have two parts: Network.Host

- **Border routers** running EGPs figure out routes between networks

- **Internal routers** running IGPs figure out host routes for hosts *in that network*
  .. and *may* propagate the network routes from the EGP (it's one way to do it)

- Scales much better than “flat” routing:
  - Border routers don’t see churn inside networks
  - Internal routers don’t see churn in other networks
  - Routers only need state for:
    - Hosts in *their network*
    - And *other networks* themselves

Addressing: Early Internet

- So that's basically how addresses worked on early Internet

Still true

- An IPv4 address is 32 bits long
- Each host gets a unique one (or more than one, and with caveats)

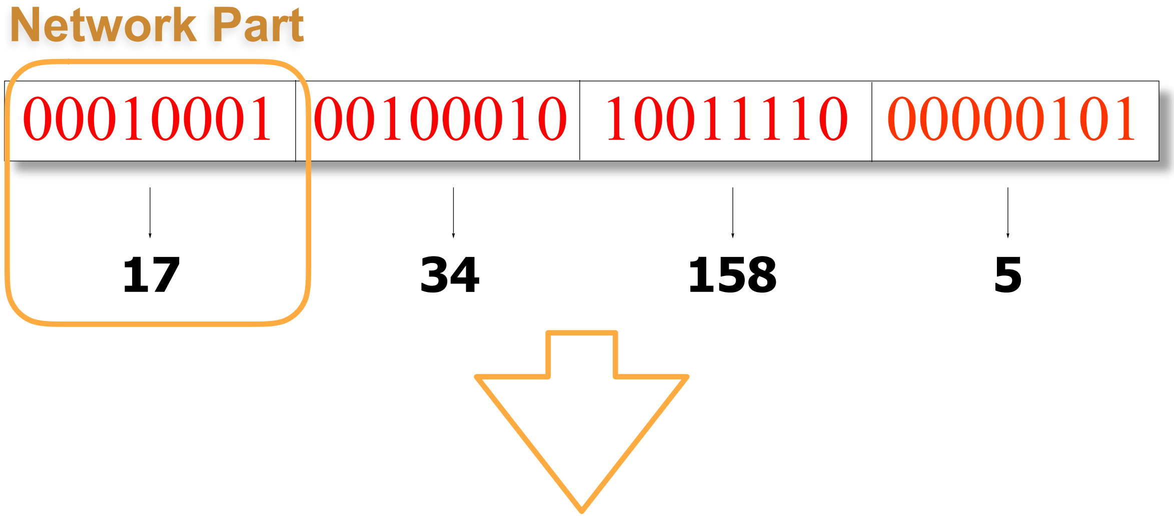
- Was broken into:
  - Network part (8 bits)
  - Host part (24 bits)

- When an organization wanted to get on the Internet, they'd get their own network part.
  - e.g., Apple was (and is still) 17…

Different today; we'll discuss…

IPv4 Addresses

- You could just represent an IPv4 address as a single big integer
- But far more common is a *dotted quad* or *dot quad*



Network Part

00010001 00100010 10011110 00000101

17 34 158 5

17.34.158.5 (or 17.2268677)

IPv4 Address Evolution

- 8 bit network part

- .. at most 256 networks

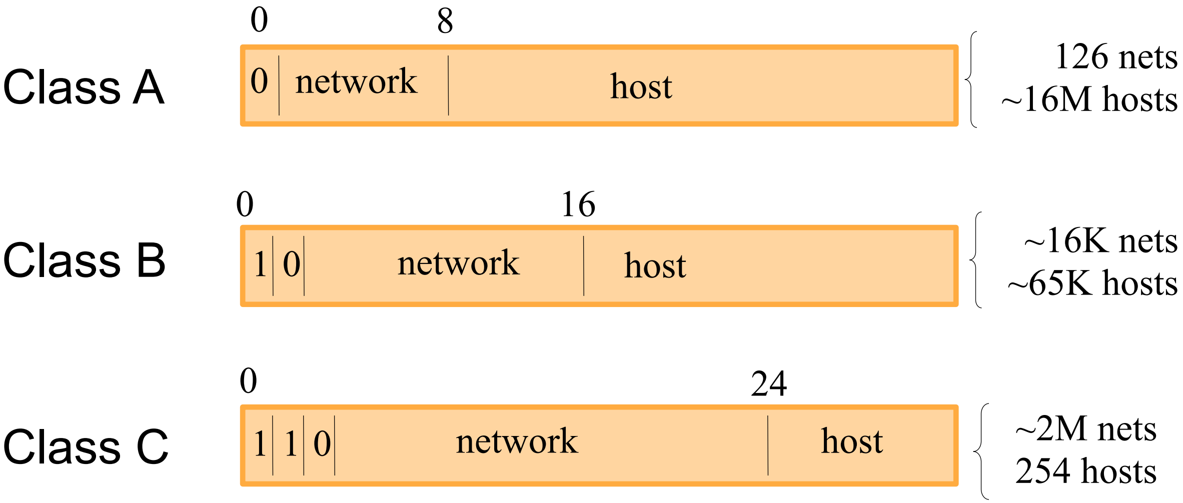
- .. this probably seemed like enough at the time

- .. boy were they ever wrong

- Became clear we needed more networks
- Solution:
    - “Classful” addressing

Classful Addressing

- Three main classes of network



Class A

0 8

0 | network | host

126 nets
~16M hosts

Class B

0 16

1 | 0 | network | host

~16K nets
~65K hosts

Class C

0 24

1 | 1 | 0 | network | host

~2M nets
254 hosts

Classful Addressing

- Ran into problems of its own!

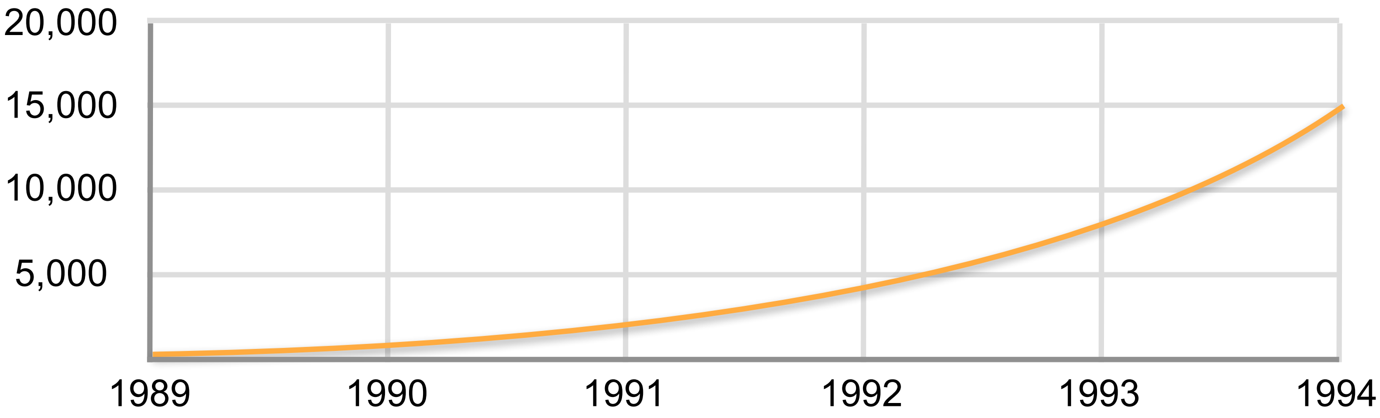
- The sizes of the classes weren’t that useful
  - Class A far too big for most organizations!
  - Class C far too small for many organizations!
  - Class B is best option for many
    - Still too big for many organizations
    - Not that many of them!

- Running out of Class B? That’s a lot of routes…
  - Number of interdomain routes was going up!

Class A	126 nets ~16M hosts
Class B	~16K nets ~65K hosts
Class C	~2M nets 254 hosts

Classful Addressing

- Number of interdomain routes by year (approximate)



CIDR: Classless Inter-Domain Routing

- So they needed a new solution: CIDR
  - Classless Inter-Domain Routing
  - Still what we use today
  - In a nutshell:
    - Introduces a hierarchical process for assignment of addresses
    - Gives up simple notion of “network part” and “host part” of fixed sizes

CIDR: Hierarchical address assignment

- ICANN (Internet Corporation for Assigned Names and Numbers)
    - .. gives out large contiguous blocks of the old Class C addresses to …

- RIRs (Regional Internet Registries)
    - (ARIN, AFRINIC, APNIC, LACNIC, RIPE NCC)
    - .. who give out portions of those blocks to …

- Large organizations
    - (e.g., ISPs like AT&T)
    - .. who give our portions of those blocks to …

- Smaller organizations and individuals
    - (e.g., UC Berkeley)

CIDR: Hierarchical assignment example (Fake!)

- **ICANN** wants ARIN to have 500M addresses
  - Requires 28 bits — Prefix
  - ICANN picks 4 bit *prefix* — `1101`
  - Assigns it to ARIN (4 + 28 = 32)
- **ARIN** allocates 8M of its addresses to AT&T
  - Requires 23 bits
  - ARIN picks next 5 bits of prefix — `110111001`
  - Assigns it to AT&T (4 + 5 + 23 = 32)
- **AT&T** allocates 16K addresses to UC Berkeley
  - Requires 14 bits
  - AT&T picks next 9 bits of prefix — `110111001110100010`
  - Assigns it to UCB (4 + 5 + 9 + 14 = 32)
- **UCB** …
  - Now has its own block with prefix of 18 bits
  - Remaining 14 bits are for its hosts — `110111001110100010xxxxxxxxxxxxxx`

CIDR: Hierarchical assignment example (Fake!)

- **ICANN** wants ARIN to have 500M addresses
  - Requires 28 bits
  - ICANN picks 4 bit *prefix*
  - Assigns it to ARIN (4 + 28 = 32)
- **ARIN** allocates 8M of its addresses to AT&T
  - Requires 23 bits
  - ARIN picks next 5 bits of prefix
  - Assigns it to AT&T (4 + 5 + 23 = 32)
- **AT&T** allocates 16K addresses to UC Berkeley
  - Requires 14 bits
  - AT&T picks next 9 bits of prefix
  - Assigns it to UCB (4 + 5 + 9 + 14 = 32)
- **UCB** …
  - Now has its own block with prefix of 18 bits
  - Remaining 14 bits are for its hosts

Prefix

```
11010000000000000000000000000000 = 208.0.0.0

11011100100000000000000000000000 = 220.128.0.0

11011100111010001000000000000000 = 220.232.128.0

110111001110100010xxxxxxxxxxxxxx
```

CIDR: Hierarchical assignment example (Fake!)

CIDR “slash notation”

- **ICANN** wants ARIN to have 500M addresses
  - Requires 28 bits
  - ICANN picks 4 bit *prefix*
  - Assigns it to ARIN (4 + 28 = 32)
- **ARIN** allocates 8M of its addresses to AT&T
  - Requires 23 bits
  - ARIN picks next 5 bits of prefix
  - Assigns it to AT&T (4 + 5 + 23 = 32)
- **AT&T** allocates 16K addresses to UC Berkeley
  - Requires 14 bits
  - AT&T picks next 9 bits of prefix
  - Assigns it to UCB (4 + 5 + 9 + 14 = 32)
- **UCB** …
  - Now has its own block with prefix of 18 bits
  - Remaining 14 bits are for its hosts

Prefix

```
11010000000000000000000000000000 = 208.0.0.0/4
11011100100000000000000000000000 = 220.128.0.0/9
11011100111010001000000000000000 = 220.232.128.0/18
110111001110100010xxxxxxxxxxxxxx
```

Netmasks: Another representation of prefixes

- Besides “slash notation”, there is *netmask* notation
- **Totally equivalent, just a different way of writing it**
- A bitmask of the prefix bits
- Just turn the prefix bits to 1 and convert to dot quad

```
11010000000000000000000000000000 = 208.0.0.0/240.0.0.0
11110000000000000000000000000000 = 240.0.0.0


11011111100000000000000000000000 = 220.128.0.0/255.128.0.0
11111111100000000000000000000000 = 255.128.0.0


11011100111010001000000000000000 = 220.232.128.0/255.255.192.0
11111111111111111100000000000000 = 255.255.192.0
```

CIDR: Classless Inter-Domain Routing

- Back to the problems CIDR was trying to solve…

- #1: Classful was wasteful

- Like our example, Berkeley wanted ~16K addresses
- Would have needed a Class B, which has ~65K address
- .. the other ~50K addresses wasted!

- With CIDR, blocks are at worst about twice as big as needed
    - .. if you want 254 addresses, you can get a /8 — no waste
    - .. if you want 255 addresses, you need a /9 — wastes 255!
    - (the first last address in a block is reserved, hence 254, not 256)

CIDR: Classless Inter-Domain Routing

- Back to the problems CIDR was trying to solve…
- #2: Number of interdomain routes was going up

To Be Continued…

Attributions

Radia Perlman, Public Domain

https://commons.wikimedia.org/wiki/File:Radia_Perlman_2009.jpg