

# Routing #3

# Today

---

- A look at some self-test questions
- Finishing up Distance-Vector protocols
- Link-State protocols
- Learning Switches

# Self-Test Questions

# Self-Test from Lecture 5

---

- Routing ensures reliable delivery of packets between end hosts.
- True (33%)
- False (67%)

# Self-Test from Lecture 5

---

- If you have valid routing state, does that mean the network will never drop any packets?
- Yes
- No (90%+) ←
  
- Routing ensures reliable delivery of packets between end hosts.
- True (33%)
- False (67%)

**If packets can still get dropped, then the network isn't ensuring they're reliably delivered.**

**Routing may be necessary, but it's not sufficient!**

# Self-Test from Lecture 5

---

- Does the "no loops, no dead ends" theorem only hold true for destination-based forwarding?
- Yes (38%)
- No (63%)

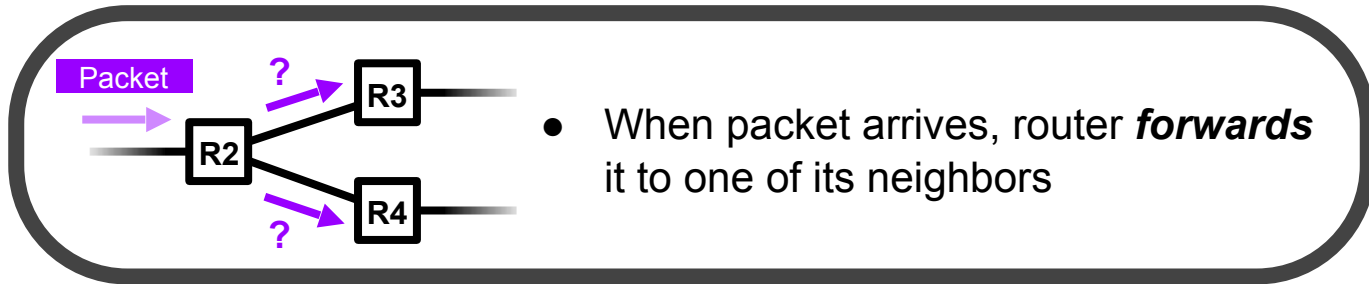
## Lecture 5

“Same basic *no loops or dead ends* condition generalizes to *at least\** any other system that does deterministic forwarding based on fixed packet headers (**that is, it's not *limited* to destination-based routing**)”

# Self-Test from Lecture 5

---

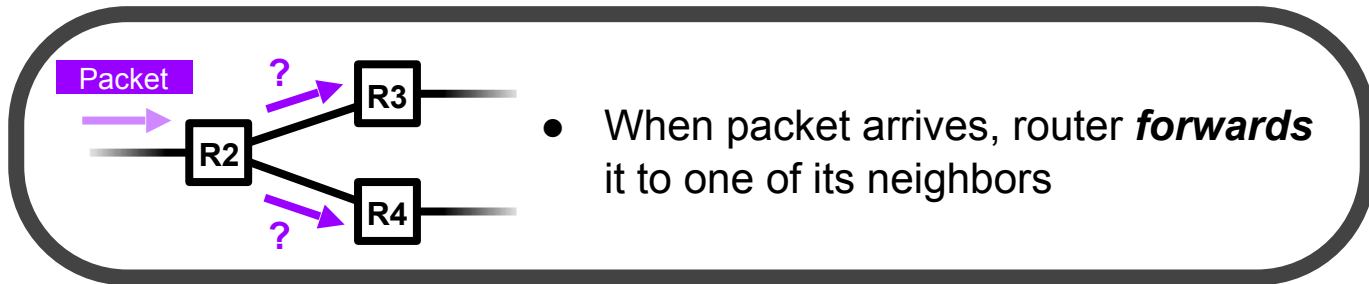
- Routing occurs when a data packet arrives at a router and is sent out another port.
  - True (46%)
  - False (54%)



# Self-Test from Lecture 5

---

- Routing occurs when a data packet arrives at a router and is sent out another port.
  - True (46%)
  - False (54%)



- *Forwarding* occurs when a data packet arrives at a router and is sent out another port.
  - Routing determines which neighbor to forward to (i.e. which port to forward out of).



Questions?

Finishing up D-V

# From B-F to D-V

---

- We refined our update rule
- We resolved some wacky problems with split horizon
- We ensured that we eventually converge instead of counting to infinity
- We made it robust to packet drops/ordering by advertising periodically
- We saw that we can adapt to new links easily
- We can identify failed links and dead routes by missing advertisements

# Distance-Vector

---

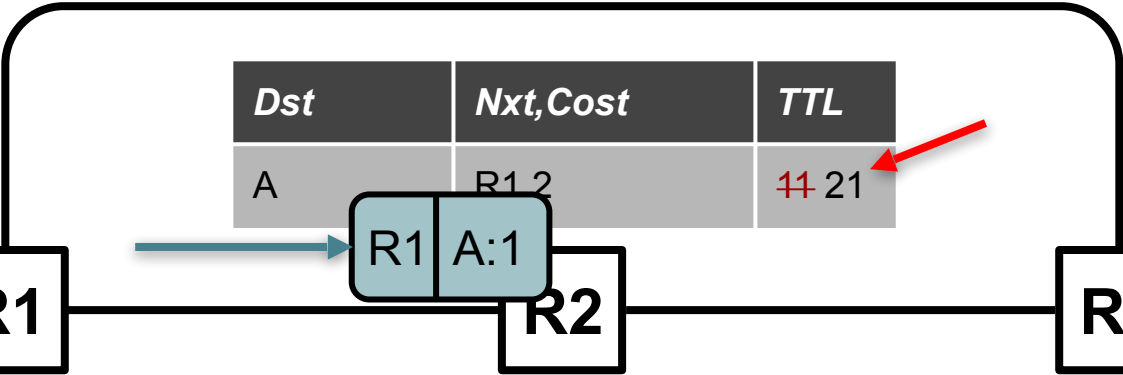
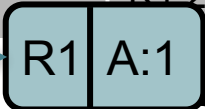
Can it handle failed links?

# Distance-Vector: Failures

**t=10**

4

<i>Dst</i>	<i>Nxt,Cost</i>	<i>TTL</i>
A	R1,2	<del>11</del> 21

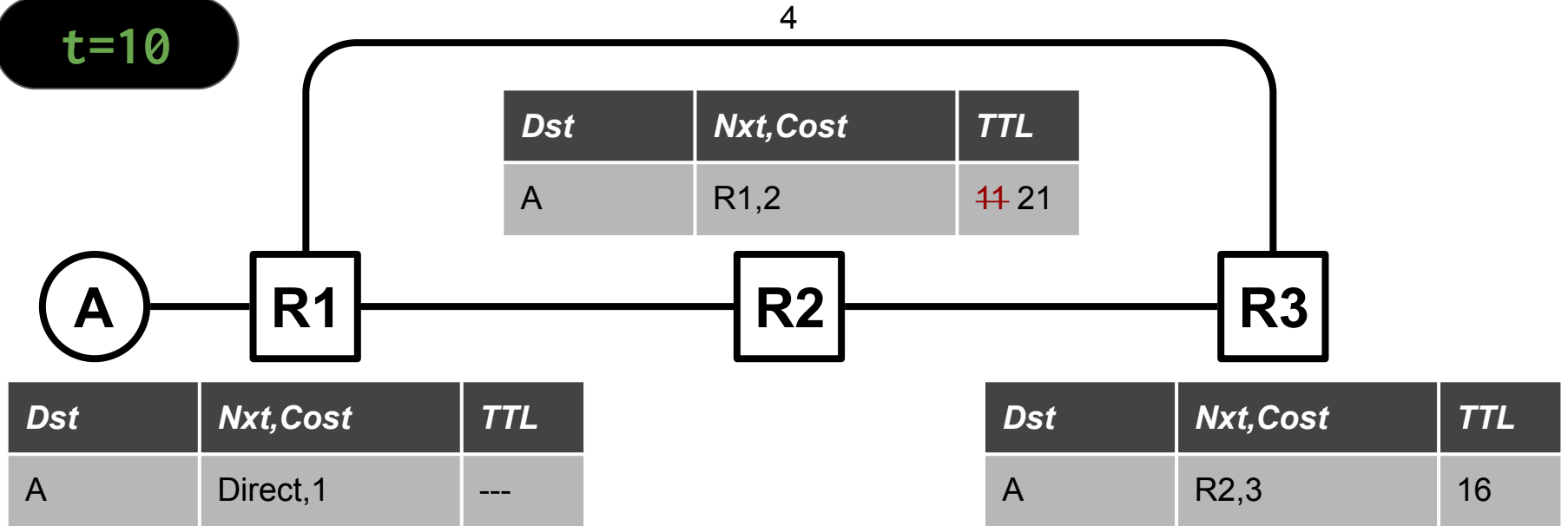


<i>Dst</i>	<i>Nxt,Cost</i>	<i>TTL</i>
A	Direct,1	---

<i>Dst</i>	<i>Nxt,Cost</i>	<i>TTL</i>
A	R2,3	16

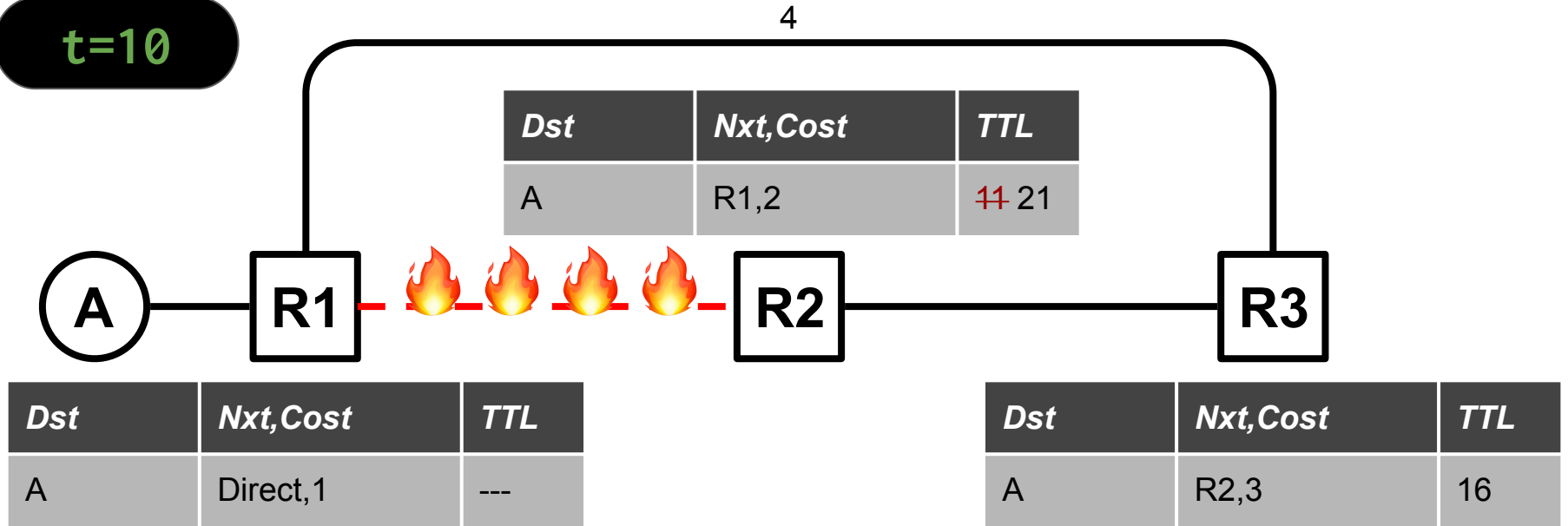
# Distance-Vector: Failures

**t=10**



# Distance-Vector: Failures

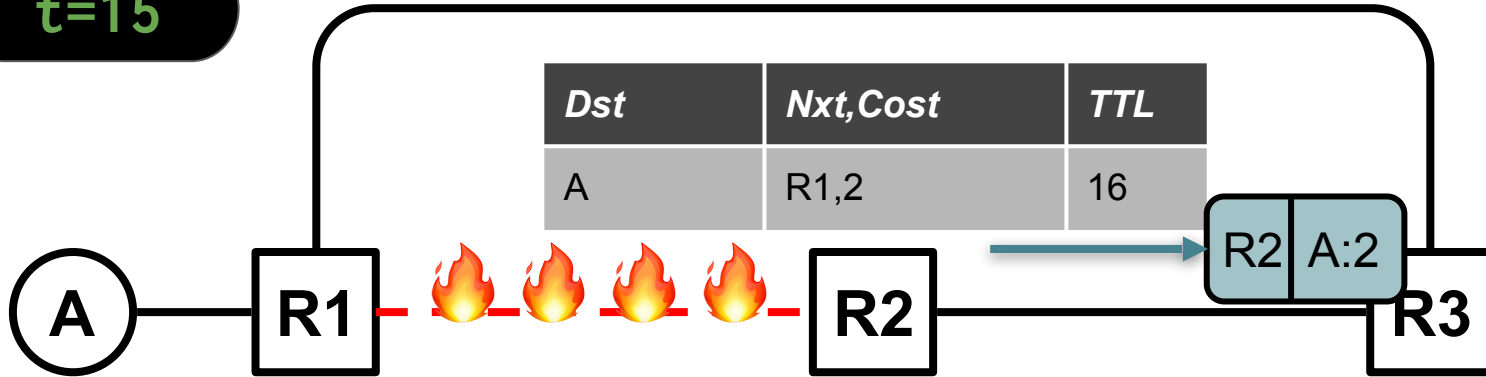
**t=10**



# Distance-Vector: Failures

t=15

4



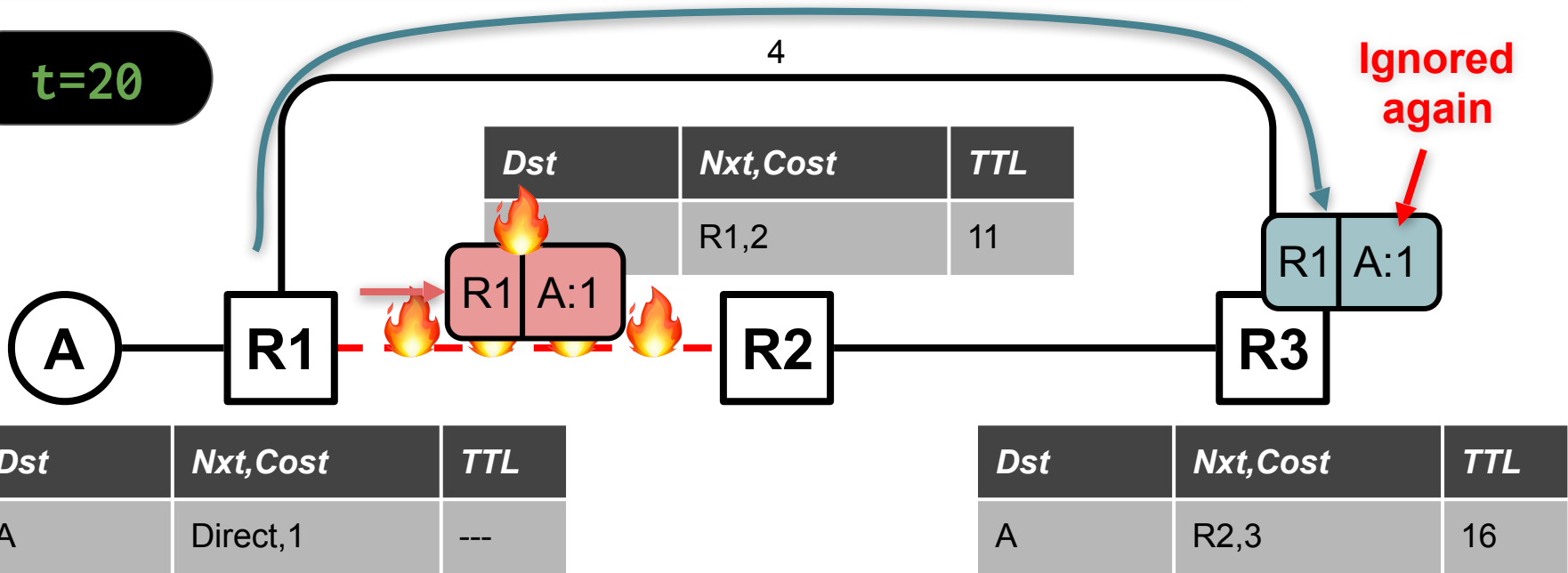
<i>Dst</i>	<i>Nxt,Cost</i>	<i>TTL</i>
A	Direct, 1	---

<i>Dst</i>	<i>Nxt,Cost</i>	<i>TTL</i>
A	R2, 3	<del>11</del> 21



# Distance-Vector: Failures

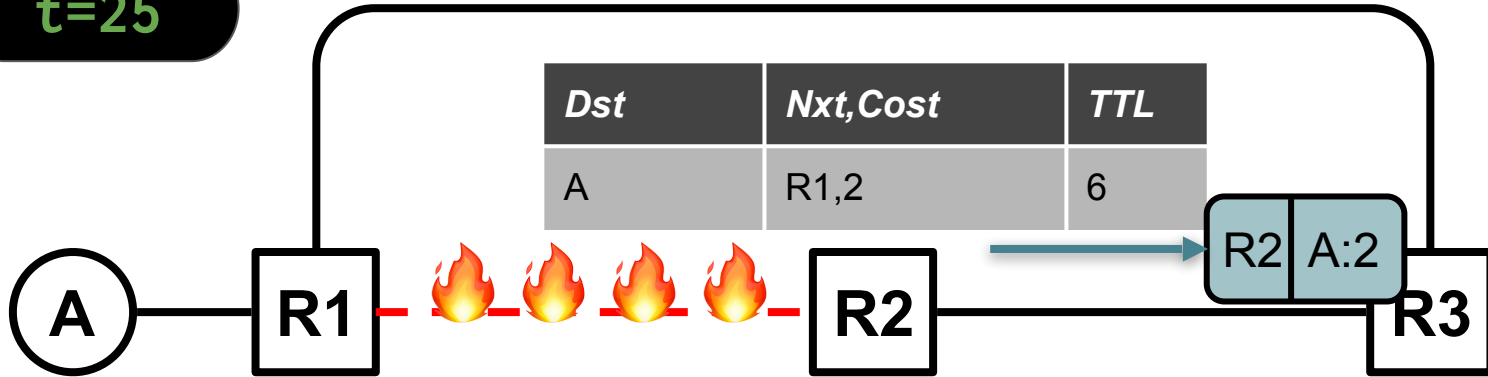
**t=20**



# Distance-Vector: Failures

t=25

4

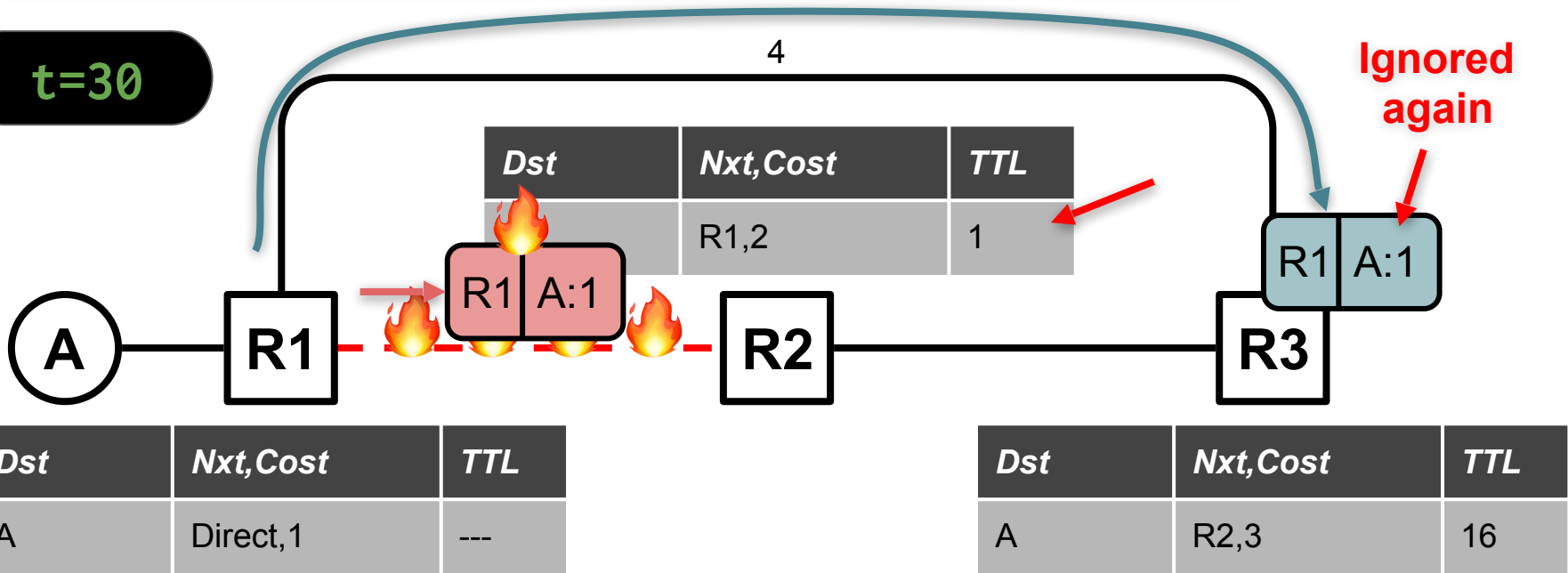


<i>Dst</i>	<i>Nxt,Cost</i>	<i>TTL</i>
A	Direct,1	---

<i>Dst</i>	<i>Nxt,Cost</i>	<i>TTL</i>
A	R2,3	<del>16</del> 21

# Distance-Vector: Failures

**t=30**

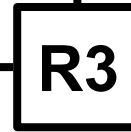
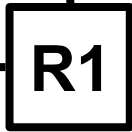


# Distance-Vector: Failures

t=31

4

<i>Dst</i>	<i>Nxt, Cost</i>	<i>TTL</i>
A	R1,2	0



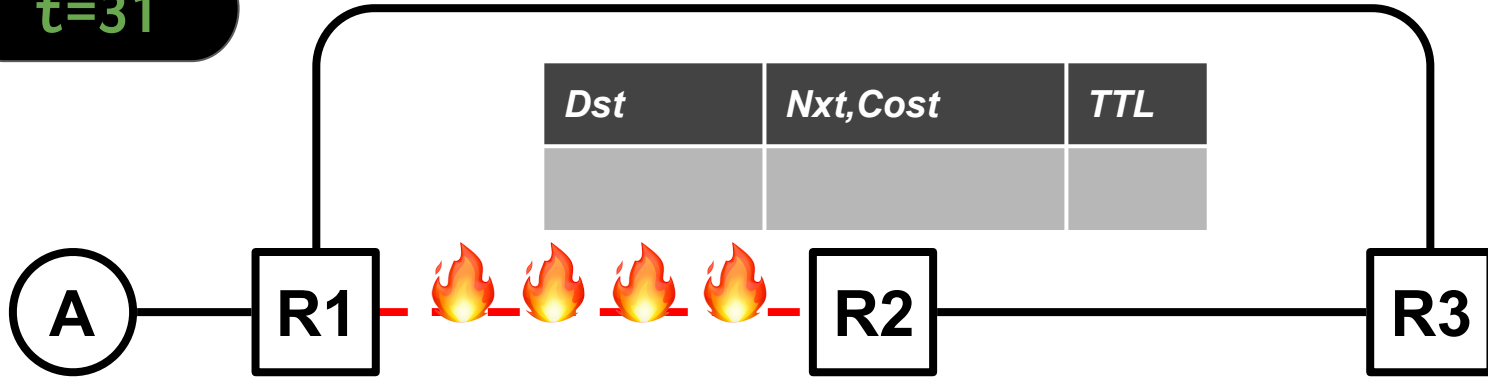
<i>Dst</i>	<i>Nxt, Cost</i>	<i>TTL</i>
A	Direct, 1	---

<i>Dst</i>	<i>Nxt, Cost</i>	<i>TTL</i>
A	R2, 3	15

# Distance-Vector: Failures

t=31

4



<i>Dst</i>	<i>Nxt, Cost</i>	<i>TTL</i>
A	Direct, 1	---

<i>Dst</i>	<i>Nxt, Cost</i>	<i>TTL</i>
A	R2, 3	15



# Distance-Vector: Failures

t=46

4



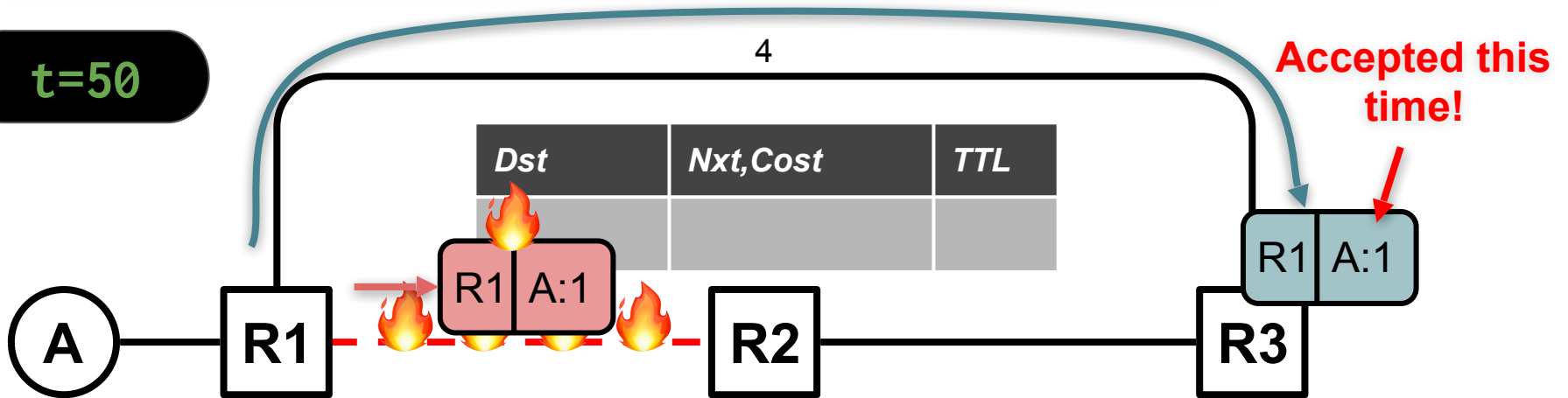
<i>Dst</i>	<i>Nxt, Cost</i>	<i>TTL</i>

<i>Dst</i>	<i>Nxt, Cost</i>	<i>TTL</i>
A	Direct, 1	---

<i>Dst</i>	<i>Nxt, Cost</i>	<i>TTL</i>

# Distance-Vector: Failures

**t=50**



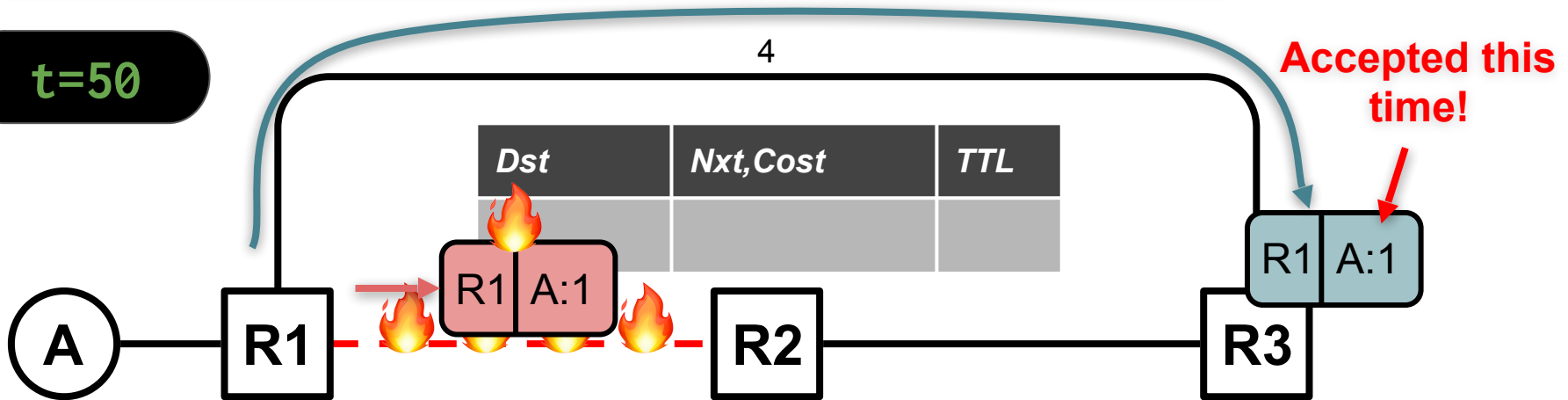
<i>Dst</i>	<i>Nxt,Cost</i>	<i>TTL</i>

<i>Dst</i>	<i>Nxt,Cost</i>	<i>TTL</i>
A	Direct, 1	---

<i>Dst</i>	<i>Nxt,Cost</i>	<i>TTL</i>

# Distance-Vector: Failures

**t=50**



<i>Dst</i>	<i>Nxt,Cost</i>	<i>TTL</i>
A	Direct, 1	---

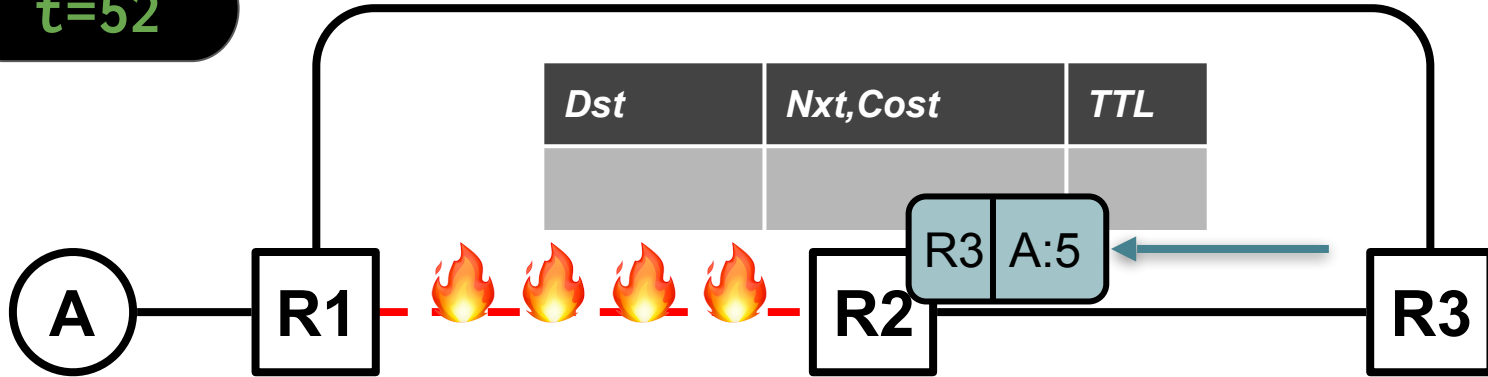
<i>Dst</i>	<i>Nxt,Cost</i>	<i>TTL</i>
A	R1, 5	21



# Distance-Vector: Failures

t=52

4

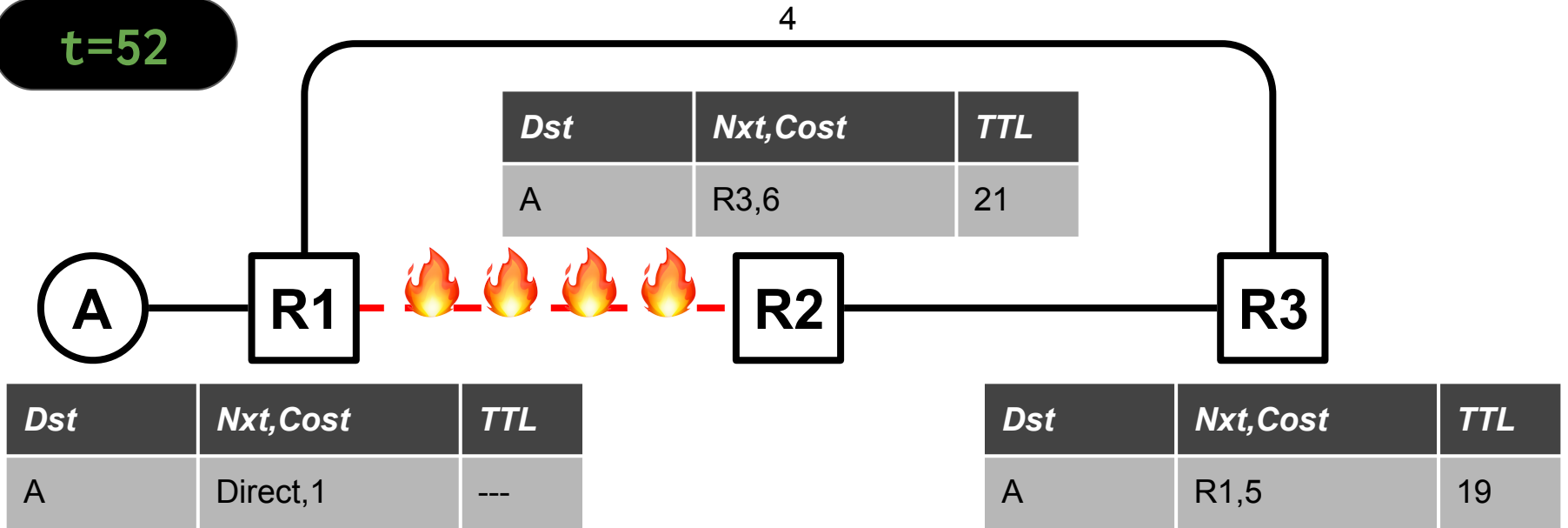


<i>Dst</i>	<i>Nxt, Cost</i>	<i>TTL</i>
A	Direct, 1	---

<i>Dst</i>	<i>Nxt, Cost</i>	<i>TTL</i>
A	R1, 5	19

# Distance-Vector: Failures

**t=52**



Questions?

# Distance-Vector

---

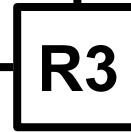
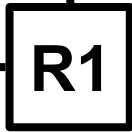
Evidence of Absence (of Routes)  
(Poisoning)

# Distance-Vector: Poison

t=31

4

<i>Dst</i>	<i>Nxt,Cost</i>	<i>TTL</i>
A	R1,2	0



<i>Dst</i>	<i>Nxt,Cost</i>	<i>TTL</i>
A	Direct,1	---

<i>Dst</i>	<i>Nxt,Cost</i>	<i>TTL</i>
A	R2,3	15

# Distance-Vector: Poison

t=31

4



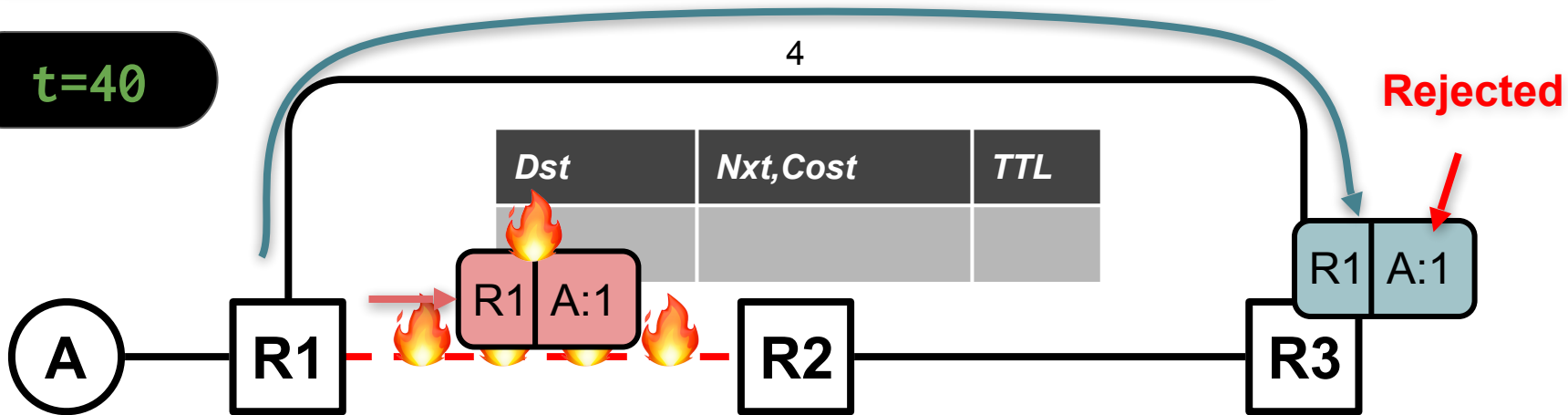
<i>Dst</i>	<i>Nxt,Cost</i>	<i>TTL</i>
A	Direct,1	---

<i>Dst</i>	<i>Nxt,Cost</i>	<i>TTL</i>
A	R2,3	15



# Distance-Vector: Poison

**t=40**



<i>Dst</i>	<i>Nxt,Cost</i>	<i>TTL</i>

<i>Dst</i>	<i>Nxt,Cost</i>	<i>TTL</i>
A	Direct,1	---

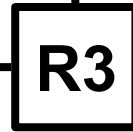
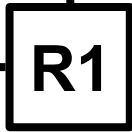
<i>Dst</i>	<i>Nxt,Cost</i>	<i>TTL</i>
A	R2,3	6

# Distance-Vector: Poison

t=31

4

<i>Dst</i>	<i>Nxt,Cost</i>	<i>TTL</i>
A	R1,2	0



<i>Dst</i>	<i>Nxt,Cost</i>	<i>TTL</i>
A	Direct,1	---

<i>Dst</i>	<i>Nxt,Cost</i>	<i>TTL</i>
A	R2,3	15

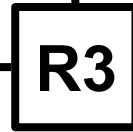
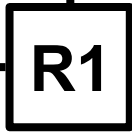


# Distance-Vector: Poison

t=31

4

<i>Dst</i>	<i>Nxt, Cost</i>	<i>TTL</i>
A	<del>R1,2</del> None, $\infty$	21



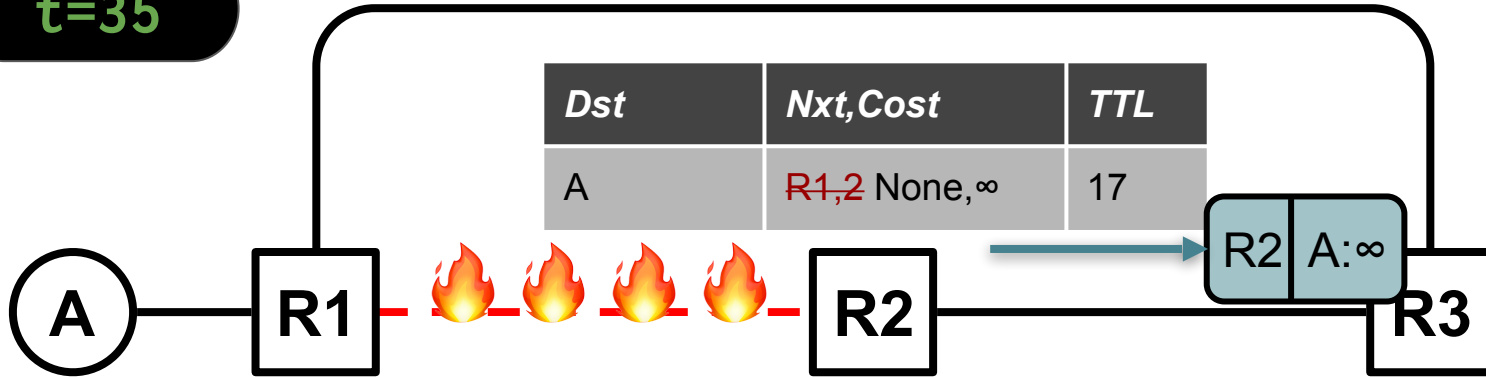
<i>Dst</i>	<i>Nxt, Cost</i>	<i>TTL</i>
A	Direct, 1	---

<i>Dst</i>	<i>Nxt, Cost</i>	<i>TTL</i>
A	R2, 3	15

# Distance-Vector: Poison

t=35

4



<i>Dst</i>	<i>Nxt,Cost</i>	<i>TTL</i>
A	R1,2 None,∞	17

R2 A:∞

<i>Dst</i>	<i>Nxt,Cost</i>	<i>TTL</i>
A	Direct,1	---

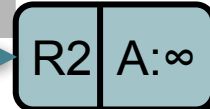
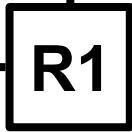
<i>Dst</i>	<i>Nxt,Cost</i>	<i>TTL</i>
A	R2,3	11

# Distance-Vector: Poison

t=35

4

<i>Dst</i>	<i>Nxt, Cost</i>	<i>TTL</i>
A	R1,2 None, $\infty$	17

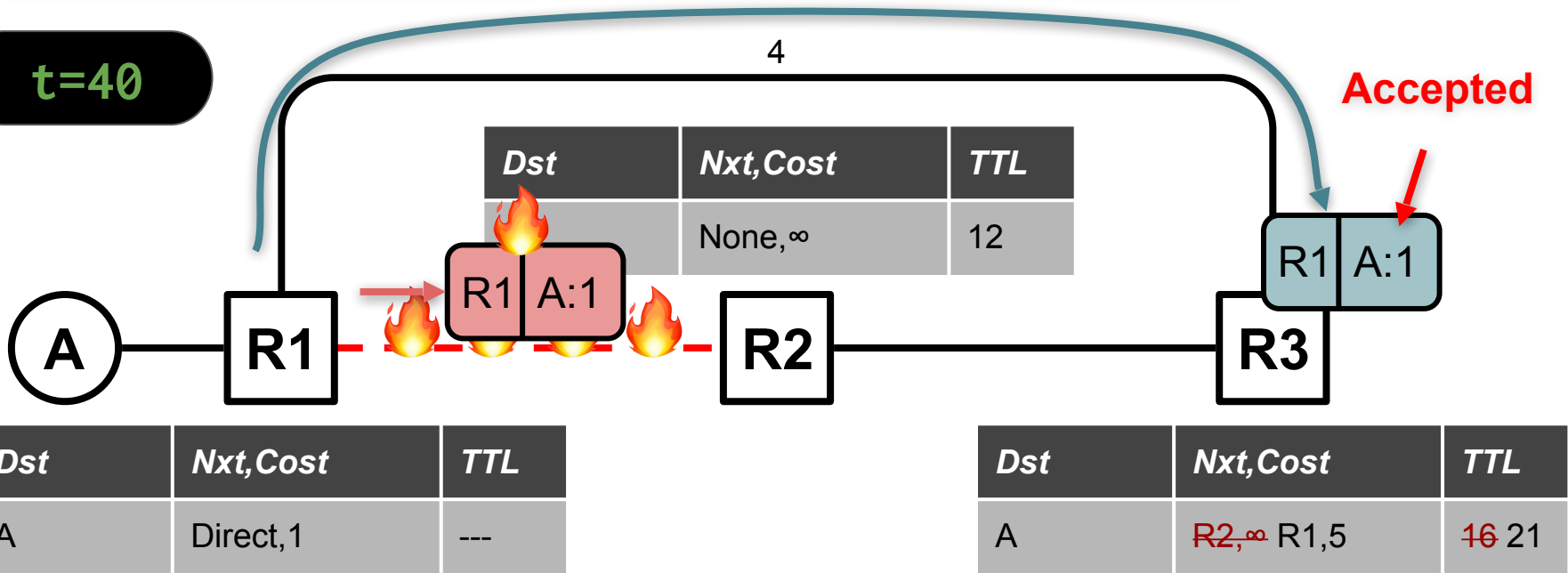


<i>Dst</i>	<i>Nxt, Cost</i>	<i>TTL</i>
A	Direct, 1	---

<i>Dst</i>	<i>Nxt, Cost</i>	<i>TTL</i>
A	R2, <del>3</del> $\infty$	21

# Distance-Vector: Poison

**t=40**



# Distance-Vector: Poison

---

- Key idea:
  - Instead of just *not* advertising a route
  - .. actively advertise that you *don't* have a route
- Do this by advertising an impossibly high cost
  - A “poison” route
- This route should propagate like other routes, poisoning the entry on any other router that was using it
- Can be much faster than waiting for timeouts!

# Distance-Vector: Poison

---

- And this doesn't just work for timed advertisements...
- If you get a poison advertisement and it changes your table...
  - Will trigger you to send poison
  - Propagates dead routes as fast as they can reach and be processed by neighbor!
- .. can be much, *much* faster than waiting for timeouts!

# Distance-Vector: Poison

---

- Besides expired routes, where else did we *not* advertise something?

# Distance-Vector: Poison

---

- Besides expired routes, where else did we *not* advertise something?
  - Split horizon!
- In split horizon, we had a route but chose not to advertise
  - Don't want to advertise a route back to router that advertised it to us!
  - Can lead to sending things backwards (or even looping)



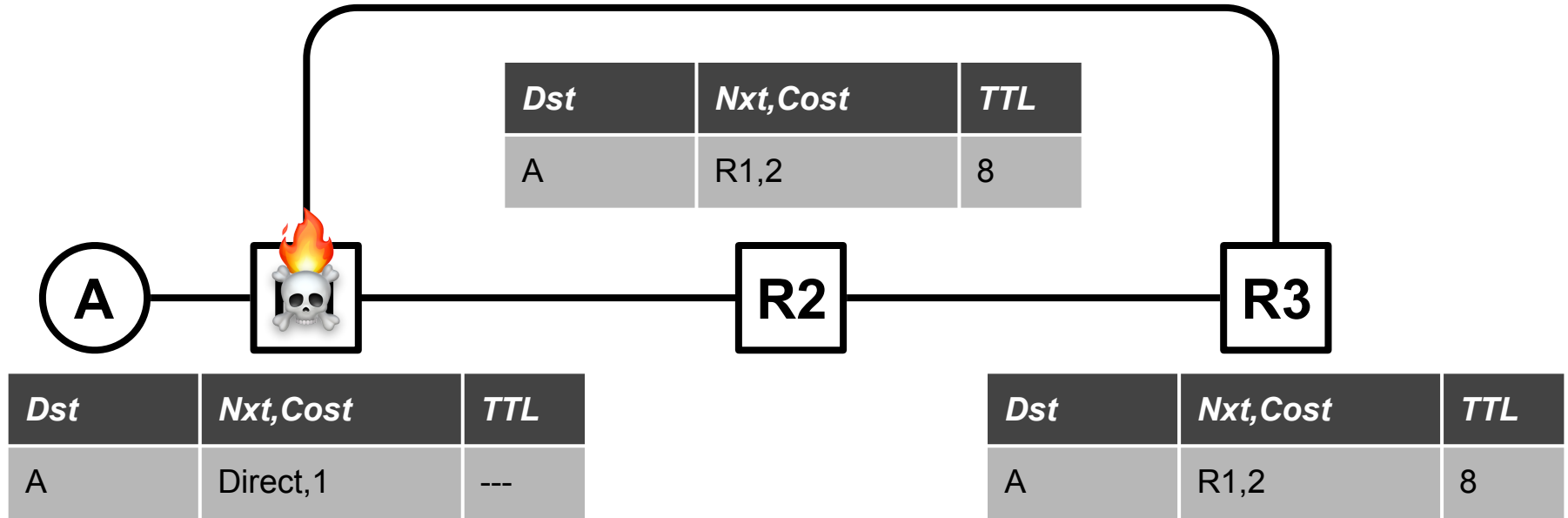
# Distance-Vector: Poison

---

- Besides expired routes, where else did we *not* advertise something?
  - Split horizon!
- In split horizon, we had a route but chose not to advertise
  - Don't want to advertise a route back to router that advertised it to us!
  - Can lead to sending things backwards (or even looping)
- Instead of *not* advertising in this case... *advertise infinite cost*
  - We call this *poison reverse*
  - Same exact idea as split horizon, but more aggressive

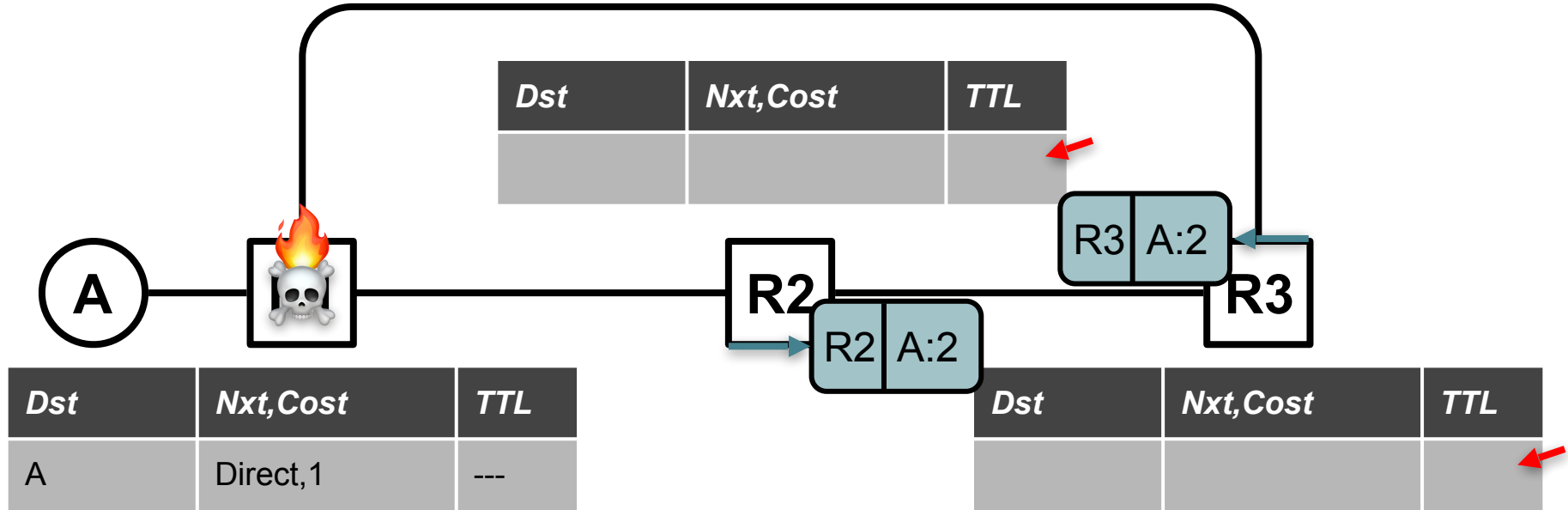
# Distance-Vector: Poison Reverse

---



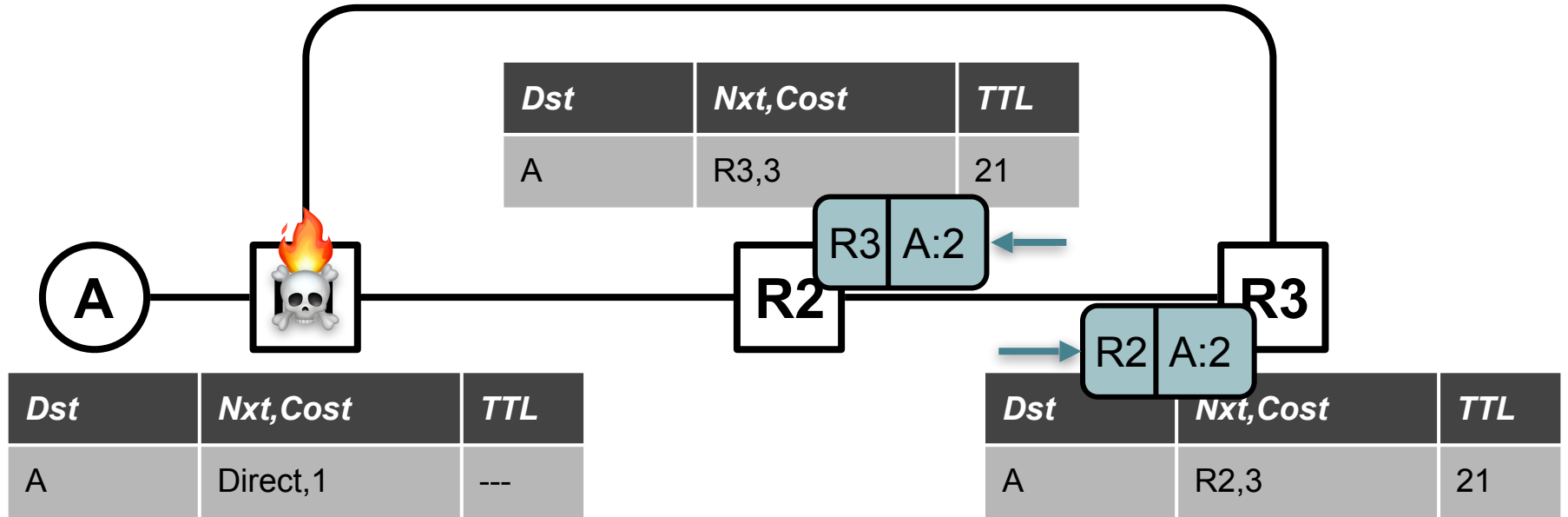
# Distance-Vector: Poison Reverse

---



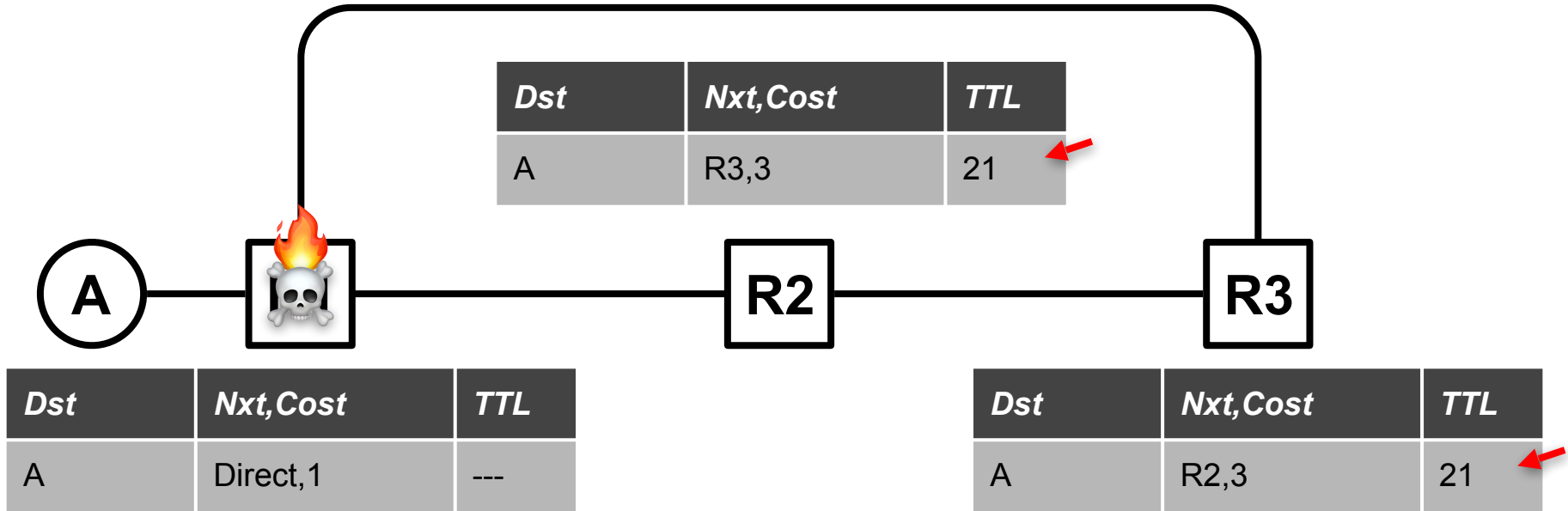
# Distance-Vector: Poison Reverse

---



# Distance-Vector: Poison Reverse

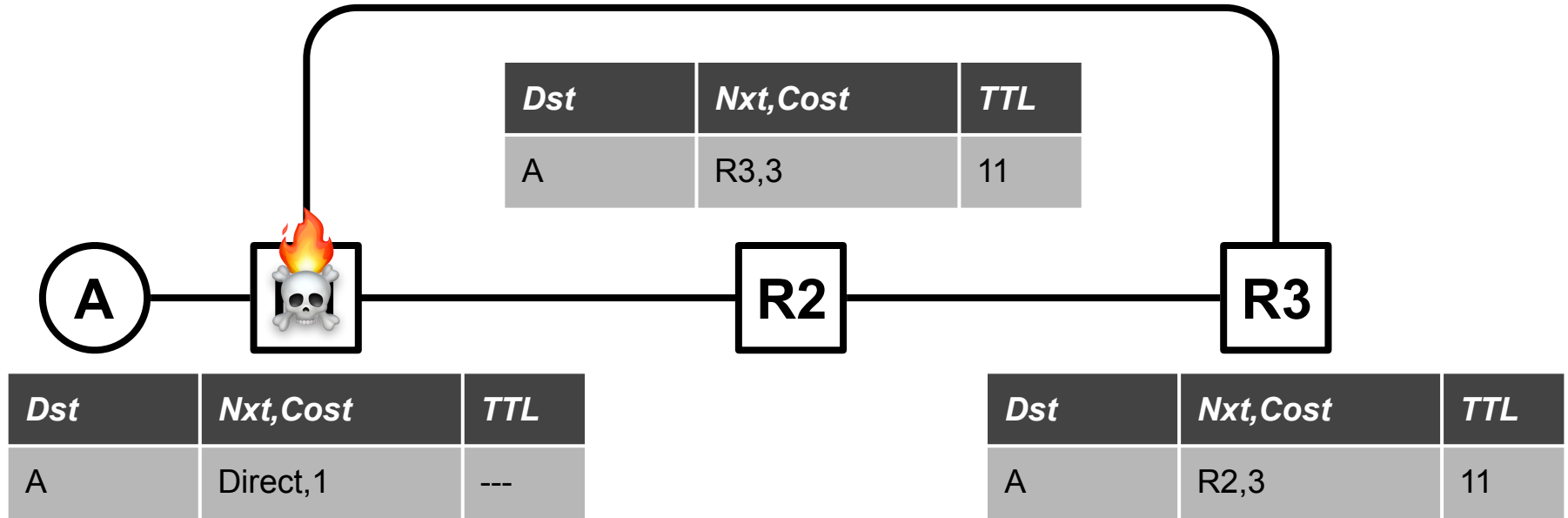
---



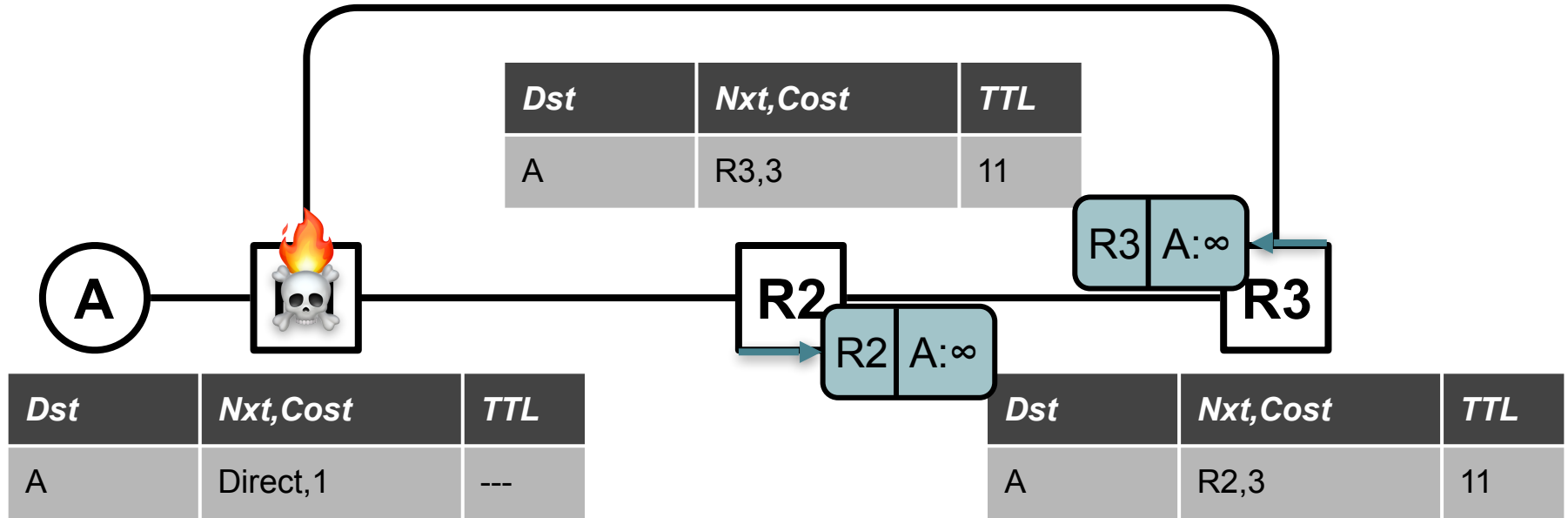
**With split horizon, loopy state exists until expiration**

# Distance-Vector: Poison Reverse

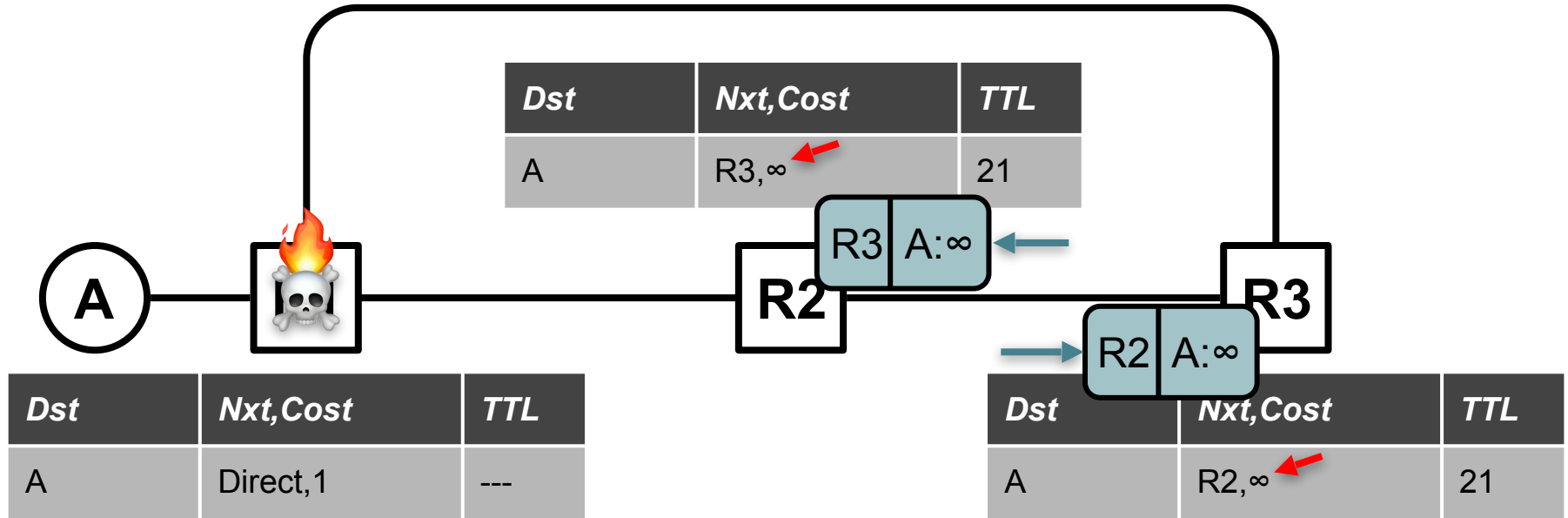
---



# Distance-Vector: Poison Reverse



# Distance-Vector: Poison Reverse



**With poison reverse, loopy state exists until next advertisement**



Questions?

# Distance-Vector: Poison

---

- Poisoning and poison reverse...
- In both cases, without poisoning, you would have *not* sent a route
- Instead, *send an explicitly terrible route* (any other route will be better)
  - (And never forward using these terrible infinite-length routes.)

# Distance-Vector

---

More events to trigger on

# Distance-Vector: More triggers

---

- We know that our table changing should trigger us to send an update
- Can be useful to handle other events too...

# Distance-Vector: More triggers

---

- We know that our table changing should trigger us to send an update
- Can be useful to handle other events too...
- Sometimes we can detect when a link becomes available
  - Immediately send new neighbor advertisements
  - No need to wait for timer

# Distance-Vector: More triggers

---

- We know that our table changing should trigger us to send an update
- Can be useful to handle other events too...
- Sometimes we can detect when a link becomes available
  - Immediately send new neighbor advertisements
  - No need to wait for timer
- Sometimes we can detect when a link fails
  - Immediately poison all table entries using that link
  - .. if there are any, advertise the newly poisoned ones!

# Distance-Vector

---

Summing up...

# From B-F to D-V

---

- We refined our update rule
- We resolved some wacky problems with split horizon / poison reverse
- We ensured that we eventually converge instead of counting to infinity
- We made it robust to packet drops/ordering by advertising periodically
- We saw that we can adapt to new links easily
- We can identify failed links and dead routes by missing advertisements
- We can converge faster by explicitly signaling the absence of a route
- We can adapt more quickly by advertising when “triggered” by events
  
- This is now a pretty good routing protocol!



Questions?

# Link-State Routing

# Link-State Routing

---

- Another major class of routing protocols: Link-State routing
- Newer than Distance-Vector
- Very common as an Interior Gateway Protocol!
  
- Two major examples:
  - IS-IS (Intermediate System to Intermediate System)
  - OSPF (Open Shortest Path First)
    - Used for Berkeley's network
  
- Works very differently than Distance-Vector!
  
- Let's explore Link-State and sketch out a design...

# Distance-Vector vs. Link-State

---

- Distance-Vector
  - Global computation (it's distributed across all nodes)
  - .. using local data (from just itself and its neighbors)

# Distance-Vector vs. Link-State

---

- Distance-Vector
  - Global computation (it's distributed across all nodes)
  - .. using local data (from just itself and its neighbors)
- Link-State
  - Local computation
  - .. using global data (from all parts of the network)

# Distance-Vector vs. Link-State

---

- Distance-Vector
  - Global computation (it's distributed across all nodes)
  - .. using local data (from just itself and its neighbors)
- Link-State
  - Local computation
  - .. using global data (from all parts of the network)
- What does this mean?
  - Hopefully the D-V part makes sense to you!
  - Let's look at the L-S part...

# Link-State

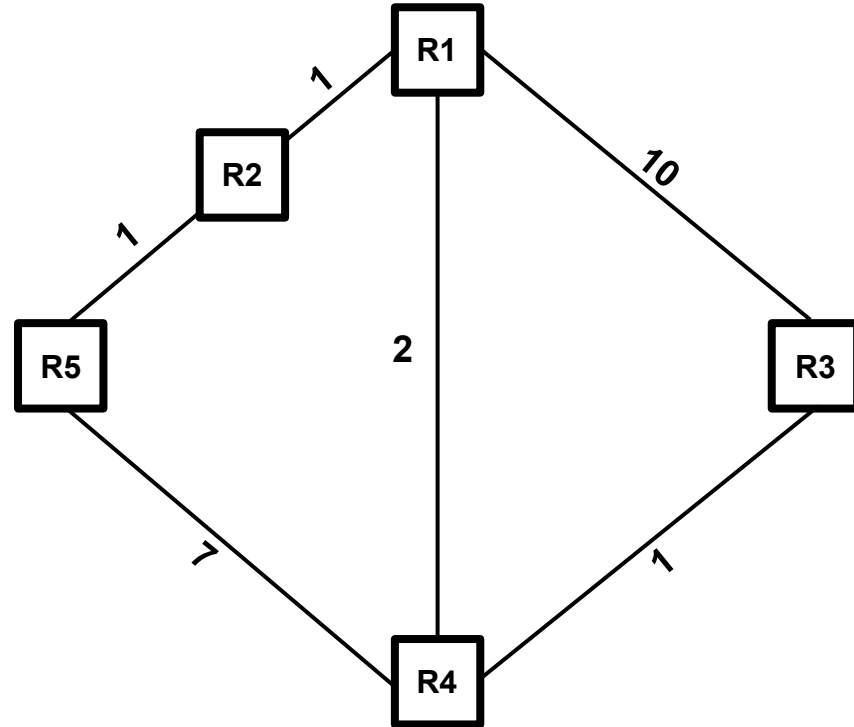
---

- A router locally computes routing state...
- .. using “global data (from all parts of the network)”
- What’s this “global data”?
  - The *state of every link* (hence: link-state)
    - Is it up?
    - What is its cost?

# Link-State: Global Data

---

- Information about the *state of links*:
  - Link R1-R2 exists and has cost 1
  - Link R1-R3 exists and has cost 10
  - Link R4-R5 exists exists and has cost 7
  - Link R1-R4 exists and has cost 2
  - Link R2-R5 exists and has cost 1
  - Link R3-R4 exists and has cost 1
- What are we missing info about?
  - Destinations!

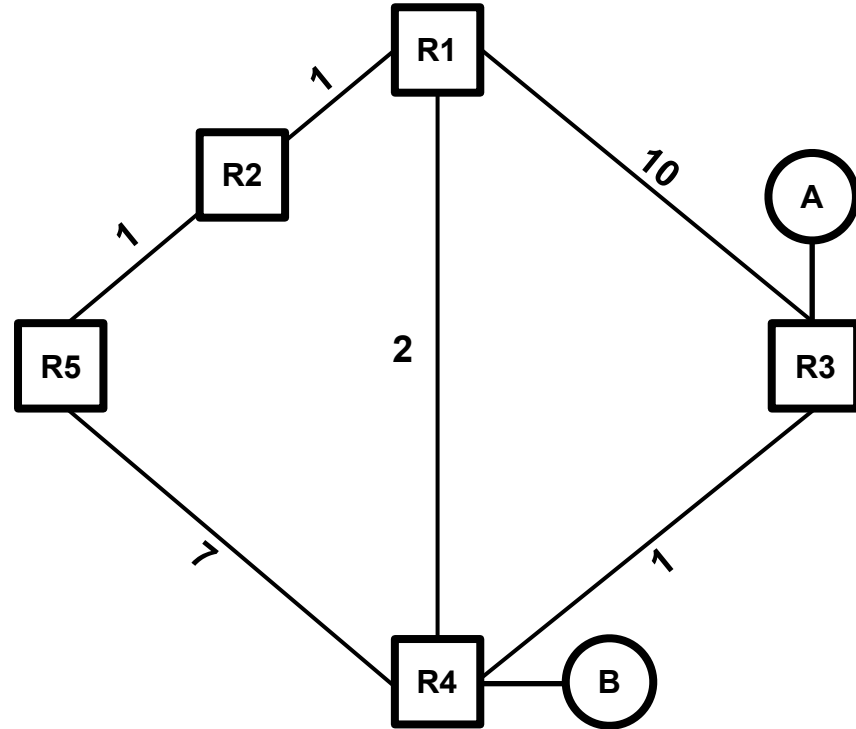




# Link-State: Global Data

---

- Information about the *state of links*:
  - Link R1-R2 exists and has cost 1
  - Link R1-R3 exists and has cost 10
  - Link R4-R5 exists exists and has cost 7
  - Link R1-R4 exists and has cost 2
  - Link R2-R5 exists and has cost 1
  - Link R3-R4 exists and has cost 1
- Information about destinations:
  - R3 has destination A
  - R4 has destination B
- .. we can use this info to build complete map (*global view*) of topology

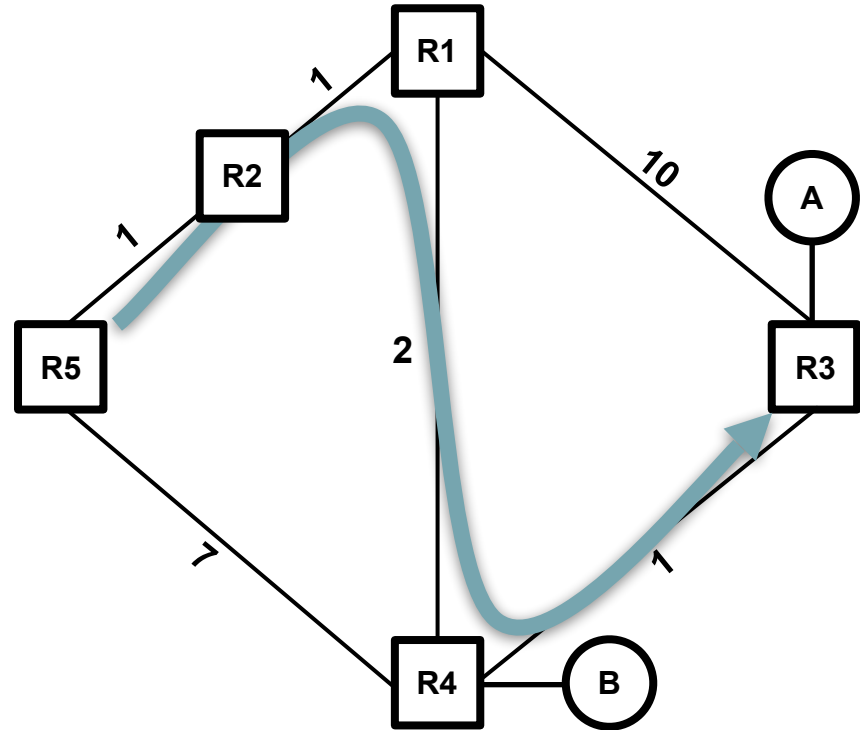


# Link-State: Global Data

---

- *If* router had global view, could easily compute paths
- Imagine you're R5
- What's the best path to A?
  - R5,R2,R1,R4,R3,A
- Which of that is useful to R5?
  - Only the R2 part!

<i>R5's Table</i>	
<i>Dst</i>	<i>Nxt</i>
A	R2



# Link-State: Overview

---

- Every router:
  - Gets the state of all links and location of all destinations
  - Uses that global information to build full graph
  - Finds paths from itself to every destination on graph
  - Uses the second hop in those paths to populate its forwarding table

# Link-State: Overview

---

- Every router:
  - Gets the state of all links and location of all destinations
    - How?! We'll come back to this in a second...
  - Uses that global information to build full graph
  - Finds paths from itself to every destination on graph
  - Uses the second hop in those paths to populate its forwarding table

# Link-State: Overview

---

- Every router:
  - Gets the state of all links and location of all destinations
    - How?! We'll come back to this in a second...
  - Uses that global information to build full graph
  - Finds paths from itself to every destination on graph
  - Uses the second hop in those paths to populate its forwarding table

# Link-State: Overview

---

- Every router:
  - Gets the state of all links and location of all destinations
    - How?! We'll come back to this in a second...
  - Uses that global information to build full graph
    - Just pastes all link/destination info together into a graph
  - Finds paths from itself to every destination on graph
  - Uses the second hop in those paths to populate its forwarding table

# Link-State: Overview

---

- Every router:
  - Gets the state of all links and location of all destinations
    - How?! We'll come back to this in a second...
  - Uses that global information to build full graph
    - Just pastes all link/destination info together into a graph
  - Finds paths from itself to every destination on graph
- Uses the second hop in those paths to populate its forwarding table

# Link-State Routing: How to Find Paths?

---

- Each router has the complete topology; can basically do it however it wants!
  - For least-cost routes, this is called Single Source Shortest Path (SSSP)
- Some obvious algorithms
  - Bellman-Ford algorithm — ( $O(|E| \cdot |V|)$ )
  - Dijkstra's algorithm — ( $O(|E| \log |V|)$ )
- Can you do better?
  - Breadth First Search — ( $O(|E| + |V|)$ )
  - Dynamic shortest path algorithms — (various)
  - Approximate shortest path algorithms — (various)
  - Parallel SSSP algorithms — (various)
  - ?

But there's nothing that says  
you *need* to do least-cost routing!

(But beware the next point...)



# Link-State: Overview

---

- Every router:
  - Gets the state of all links and location of all destinations
    - How?! We'll come back to this in a second...
  - Uses that global information to build full graph
    - Just pastes all link/destination info together into a graph
  - Finds paths from itself to every destination on graph
    - Using any pathfinding algorithm (e.g., Dijkstra's)
  - Uses the second hop in those paths to populate its forwarding table

# Link-State: Overview

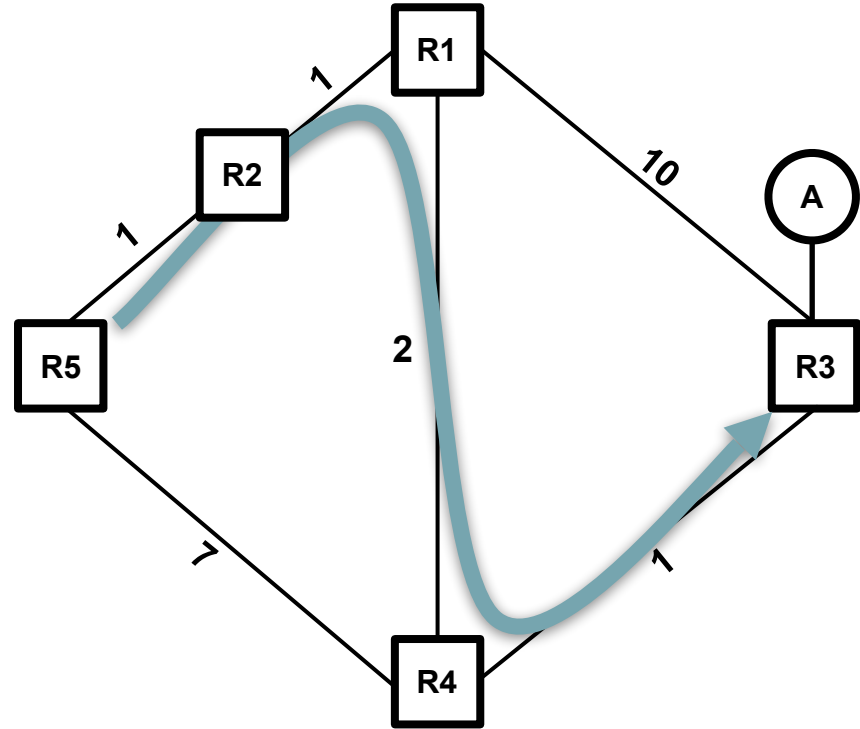
---

- Every router:
  - Gets the state of all links and location of all destinations
    - How?! We'll come back to this in a second...
  - Uses that global information to build full graph
    - Just pastes all link/destination info together into a graph
  - Finds paths from itself to every destination on graph
    - Using any pathfinding algorithm (e.g., Dijkstra's)
  - Uses the second hop in those paths to populate its forwarding table

# Link-State: Populating Tables

- Important: Remember, each router can only influence its own next hop!
- Other routers must be coming up with paths which are “compatible”

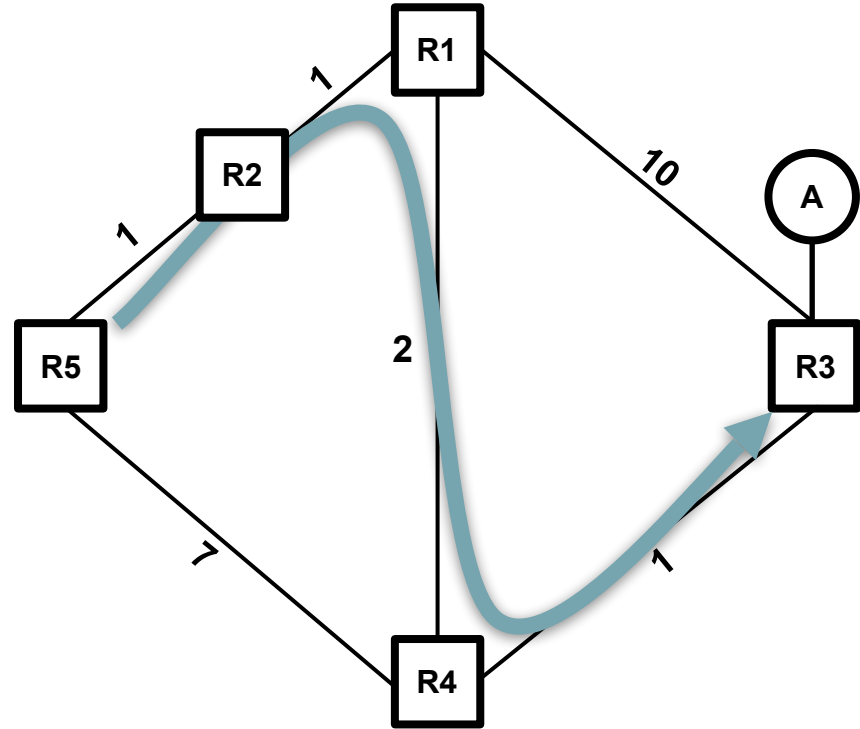
<i>R5's Table</i>	
<i>Dst</i>	<i>Nxt</i>
A	R2



# Link-State: Populating Tables

- Important: Remember, each router can only influence its own next hop!
- Other routers must be coming up with paths which are “compatible”
- Pretty easy for least-cost routing if:
  - Minimizing the same cost
  - All costs are  $> 0$

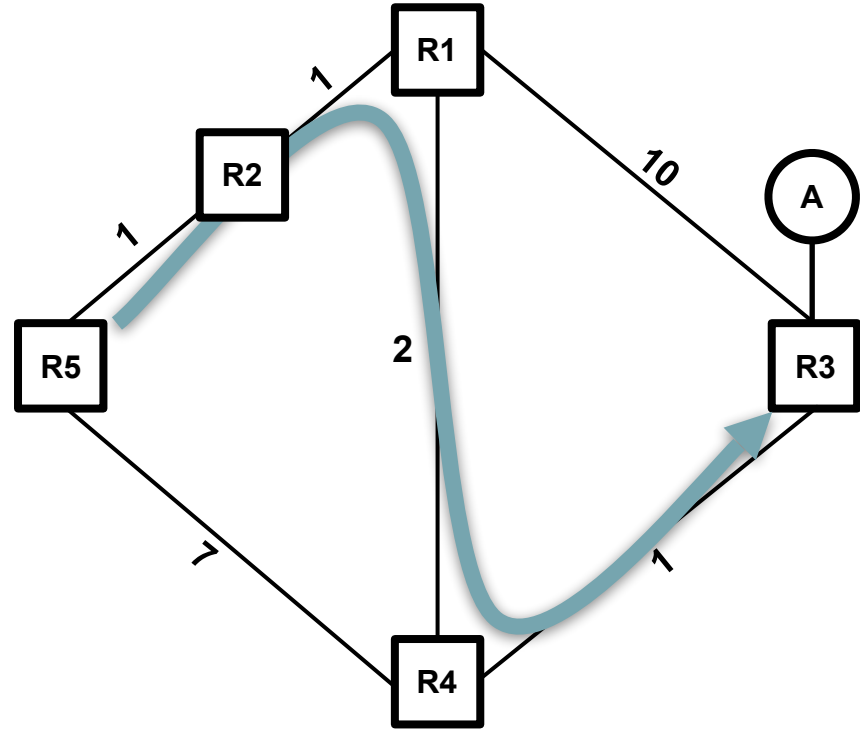
<i>R5's Table</i>	
<i>Dst</i>	<i>Nxt</i>
A	R2



# Link-State: Populating Tables

- Important: Remember, each router can only influence its own next hop!
- Other routers must be coming up with paths which are “compatible”
- Pretty easy for least-cost routing if:
  - Minimizing the same cost
  - All costs are  $> 0$
  - All routers agree on topology!

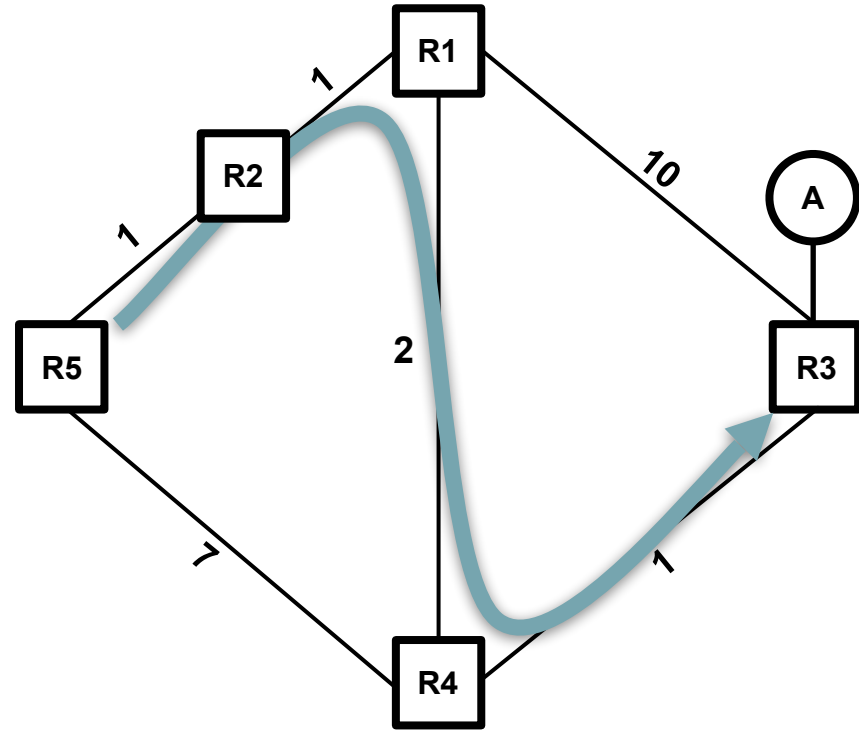
<i>R5's Table</i>	
<i>Dst</i>	<i>Nxt</i>
A	R2



# Link-State: Populating Tables

---

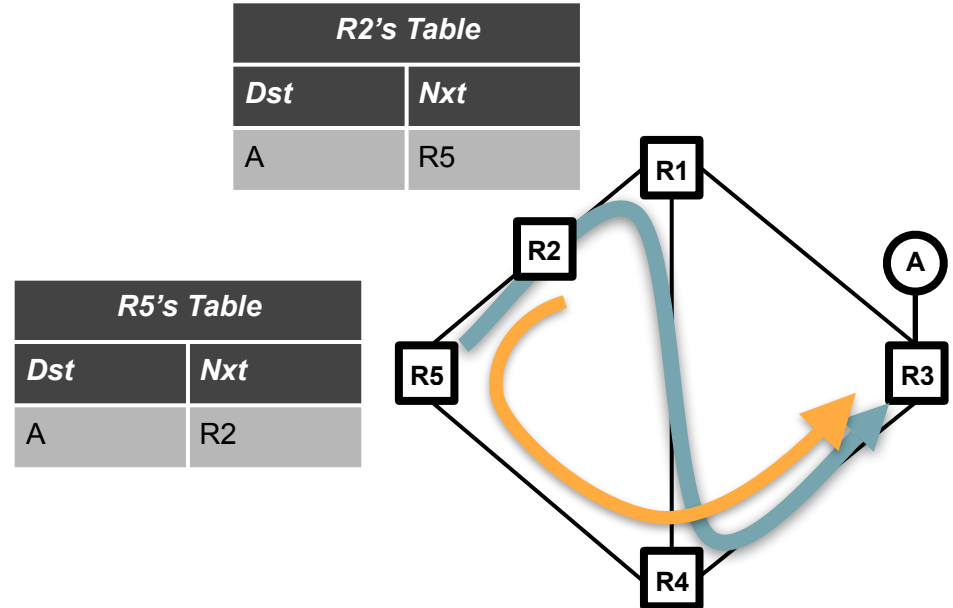
- Important: Remember, each router can only influence its own next hop!
- Other routers must be coming up with paths which are “compatible”
- Pretty easy for least-cost routing if:
  - Minimizing the same cost
  - All costs are  $> 0$
  - All routers agree on topology!
- Given all those, don't even need to implement same exact algorithm (e.g., break ties exactly the same)



# Link-State: Populating Tables

---

- Counterexample: R2 and R5 both compute apparently-feasible paths...
- .. but they don't work together!
- (Packets loop between R2 and R5)



# Link-State: Overview

---

- Every router:
  - Gets the state of all links and location of all destinations
    - How?! We'll come back to this in a second...
  - Uses that global information to build full graph
    - Just pastes all link/destination info together into a graph
  - Finds paths from itself to every destination on graph
    - Using any pathfinding algorithm (e.g., Dijkstra's)
  - Uses the second hop in those paths to populate its forwarding table
    - If every router chooses compatible paths, we're done!



# Link-State: Overview

---

- Every router:
  - Gets the state of all links and location of all destinations
    - ...
  - Uses that global information to build full graph
    - Just pastes all link/destination info together into a graph
  - Finds paths from itself to every destination on graph
    - Using any pathfinding algorithm (e.g., Dijkstra's)
  - Uses the second hop in those paths to populate its forwarding table
    - If every router chooses compatible paths, we're done!

# Link-State: Sharing Info Globally

---

- All routers need info about:
  - All links between all routers
  - All destinations
- Every router must:
  - Find out who its neighbors are
  - Tell everyone about its neighbors (i.e., its links to them)
  - Tell everyone else about any adjacent destinations

# Link-State: Sharing Info Globally

---

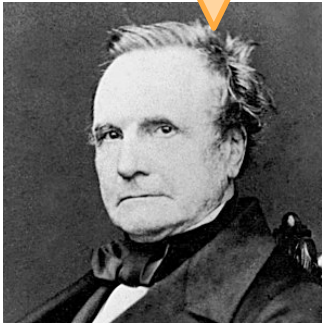
- All routers need info about:
  - All links between all routers
  - All destinations
- Every router must:
  - Find out who its neighbors are
  - Tell everyone about its neighbors (i.e., its links to them)
  - Tell everyone else about any adjacent destinations

# Link-State: Finding Your Neighbors

---

- How does anyone ever know who their neighbors are?
  - Introduce yourselves!
- Routers periodically send *hello messages* to neighbors
  - If a neighbor goes quiet, eventually assume they're gone

Hi, I'm Charles.



My neighbors: Ada



Hi, I'm Ada.



My neighbors: Charles & Margaret



Hi, I'm Margaret.



My neighbors: Ada



# Link-State: Sharing Info Globally

---

- All routers need info about:
  - All links between all routers
  - All destinations
- Every router must:
  - Find out who its neighbors are
    - By exchanging *hellos*
  - Tell everyone about its neighbors (i.e., its links to them)
  - Tell everyone else about any adjacent destinations

# Link-State: Sharing Info Globally

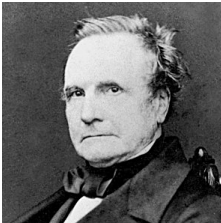
---

- All routers need info about:
  - All links between all routers
  - All destinations
- Every router must:
  - Find out who its neighbors are
    - By exchanging *hellos*
  - Tell everyone about its neighbors (i.e., its links to them)
  - Tell everyone else about any adjacent destinations

# Link-State: Flooding

---

- Exchanging hellos tells you who your neighbors are (local info)
- But we need to know who everyone's neighbors are!
- Solution is called *flooding*
- Strawman solution:
  - When local information (e.g., neighbors) changes, send to all neighbors
  - When you receive info packet from neighbor, send to all *other* neighbors



Margaret is neighbors with Ada; pass it on!



# Link-State: Flooding

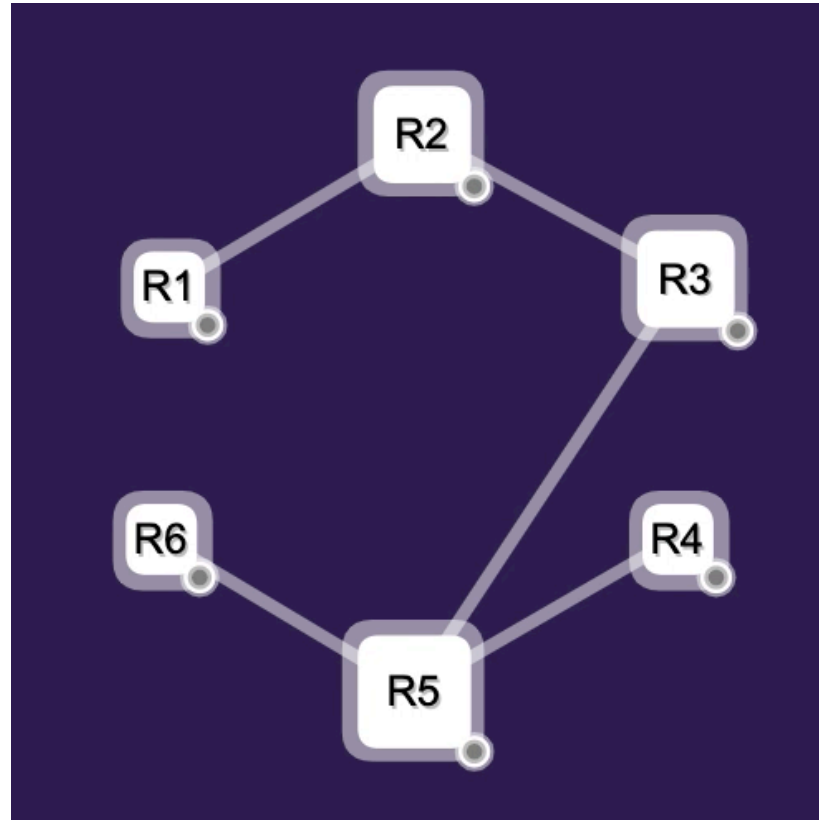
---

- Exchanging hellos tells you who your neighbors are (local info)
- But we need to know who everyone's neighbors are!
  
- Solution is called *flooding*
  
- Strawman solution:
  - When local information (e.g., neighbors) changes, send to all neighbors
  - When you receive info packet from neighbor, send to all *other* neighbors
  
  - Does this always work?



# Link-State: Flooding

---



**Note that you can easily recreate this demo in the Project 1 simulator using the example “hub” switch.**

**Just set up the shown topology and send a ping.**

**Then add a link between R2 and R6.**

**Then add a link between R3 and R4.**

**(Or use any other topology with one or two loops!)**

# Link-State: Flooding

---

- Naive solution doesn't work when topology has loops...
  - One loop: Packets loop around cycle forever (bad)
  - Multiple loops: Packets multiply exponentially (**very** bad)
- Solution?
- When local information (e.g., neighbors) changes, send to all neighbors
- When you receive info packet from neighbor, send to all other neighbors
  - .. *unless you've already seen this info packet (in which case, drop it)*
- How do you know if it's the first time you've seen it?
  - Easy solution: routers put a *sequence number* in their updates

# Link-State: Flooding

---

- Every router has its own sequence number
  - When it sends a routing message, it puts it in the packet...
  - .. and then increments it
- Every router *tracks* largest sequence number seen from every other router  
**Would this be a problem if we used this protocol as an EGP?**
- .. if it sees an update with a smaller/equal sequence number...
  - the update is old — drop it
- .. if it sees an update with a larger sequence number...
  - the update is new — remember sequence number and flood update to all other neighbors

# Link-State: Flooding

---

- How to make flooding reliable?
- Can use our same old trick: periodically resend it
- IS-IS and OSPF both do this
- .. but do more clever stuff too
  - .. delivers reliability faster without resending more often
  - We won't explore this in detail in context of Link-State
  - .. but we'll talk about reliability in more detail in week 6

# Link-State: Sharing Info Globally

---

- All routers need info about:
  - All links between all routers
  - All destinations
- Every router must:
  - Find out who its neighbors are
    - By exchanging *hellos*
  - Tell everyone about its neighbors (i.e., its links to them)
  - Tell everyone else about any adjacent destinations

# Link-State: Sharing Info Globally

---

- All routers need info about:
  - All links between all routers
  - All destinations
- Every router must:
  - Find out who its neighbors are
    - By exchanging *hellos*
  - Tell everyone about its neighbors (i.e., its links to them)
    - By *flooding* this information
  - Tell everyone else about any adjacent destinations

# Link-State: Sharing Info Globally

---

- All routers need info about:
  - All links between all routers
  - All destinations
- Every router must:
  - Find out who its neighbors are
    - By exchanging *hellos*
  - Tell everyone about its neighbors (i.e., its links to them)
    - By *flooding* this information
  - Tell everyone else about any adjacent destinations

# Link-State: Sharing Info Globally

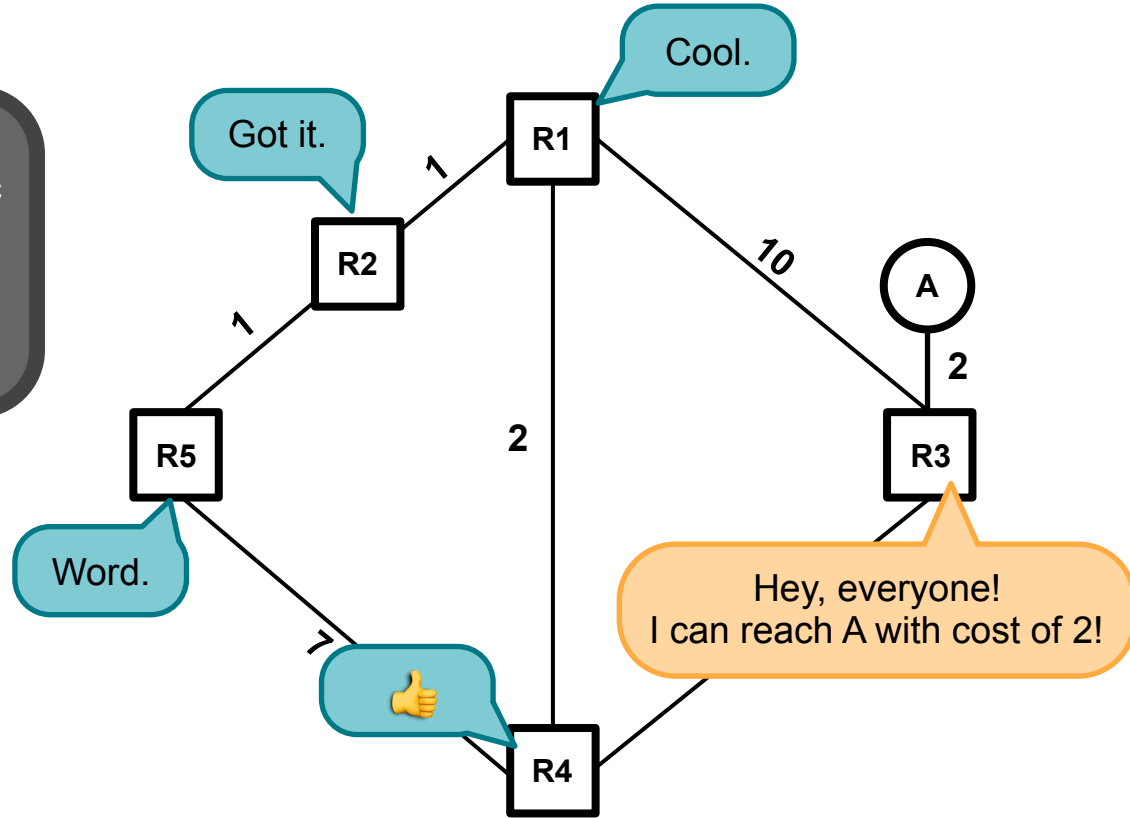
---

- All routers need info about:
  - All links between all routers
  - All destinations
- Every router must:
  - Find out who its neighbors are
    - By exchanging *hellos*
  - Tell everyone about its neighbors (i.e., its links to them)
    - By *flooding* this information
  - Tell everyone else about any adjacent destinations
    - By flooding info about destinations too (e.g., static routes)



# Link-State: Sharing Info Globally

Flood info about *destinations* (e.g., static routes to hosts) using same flooding mechanism you use to share info about *router topology* (neighboring routers).



# Link-State: Sharing Info Globally

---

- All routers need info about:
  - All links between all routers
  - All destinations
- Every router must:
  - Find out who its neighbors are
    - By exchanging *hellos*
  - Tell everyone about its neighbors (i.e., its links to them)
    - By *flooding* this information
  - Tell everyone else about any adjacent destinations
    - By flooding info about destinations too (e.g., static routes)

# Link-State: Sharing Info Globally

---

- All routers need info about:
  - All links between all routers
  - All destinations
- Every router must:
  - Find out who its neighbors are
    - By exchanging *hellos*
  - Tell everyone about its neighbors (i.e., its links to them)
    - By *flooding* this information
  - Tell everyone else about any adjacent destinations
    - By flooding info about destinations too (e.g., static routes)

# Link-State: Overview

---

- Every router:
  - Gets the state of all links and location of all destinations
    - ...
  - Uses that global information to build full graph
    - Just pastes all link/destination info together into a graph
  - Finds paths from itself to every destination on graph
    - Using any pathfinding algorithm (e.g., Dijkstra's)
  - Uses the second hop in those paths to populate its forwarding table
    - If every router chooses compatible paths, we're done!

# Link-State: Overview

---

- Every router:
  - Gets the state of all links and location of all destinations
    - Via hellos and flooding
  - Uses that global information to build full graph
    - Just pastes all link/destination info together into a graph
  - Finds paths from itself to every destination on graph
    - Using any pathfinding algorithm (e.g., Dijkstra's)
  - Uses the second hop in those paths to populate its forwarding table
    - If every router chooses compatible paths, we're done!

# Link-State: Overview

---

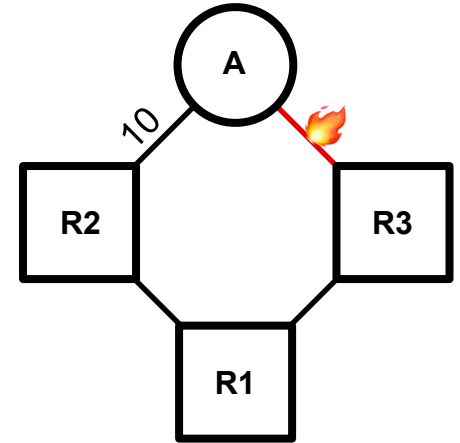
- Every router:
  - Gets the state of all links and location of all destinations
    - Via hellos and flooding
  - Uses that global information to build full graph
    - Just pastes all link/destination info together into a graph
  - Finds paths from itself to every destination on graph
    - Using any pathfinding algorithm (e.g., Dijkstra's)
  - Uses the second hop in those paths to populate its forwarding table
    - If every router chooses compatible paths, we're done!

Questions?

# Link-State: Convergence

---

- Using plain non-parallel Dijkstra's algorithm (or whatever)
  - Dijkstra's will never find a looping path
  - So we never have loops in Link-State protocols
    - Is this true?
  - It's false!
  - We only have control of our own next hop!
    - If routers don't have same global view of topology, all bets are off!
    - For example:
      - R1 doesn't know about failure yet, sends packet to R3
      - R3 gets packet, sends to to R1
      - (Loop)





# Link-State: Convergence

---

- Sources of convergence delay:
  - Time to detect failure
  - Time to flood link-state information (proportional to network diameter)
  - Time to re-compute paths/tables
- Problems during convergence period:
  - Deadends
  - Looping packets
  - Out-of-order packets reaching the destination
    - Should not cause semantic problems
    - But can create performance problems!
    - (We'll see why later in semester)

# Link-State: Timeline for Local Failure

---

- Failure not detected
  - Packets sent into dead link (dropped)
- Detected, not recomputed
  - Deadends
- Detected/computed, not globally notified/computed
  - Could be loops
- As nodes become aware, routes may change
  - Continued looping, and possible reorderings
  - **Why reordering?**

Questions?

# Link-State in a single slide


---

- Link state is super simple conceptually:
  - Everyone floods link/destination information
  - Everyone then has global map of network
  - Everyone independently computes next hops
- .. *all the complexity is in the details*

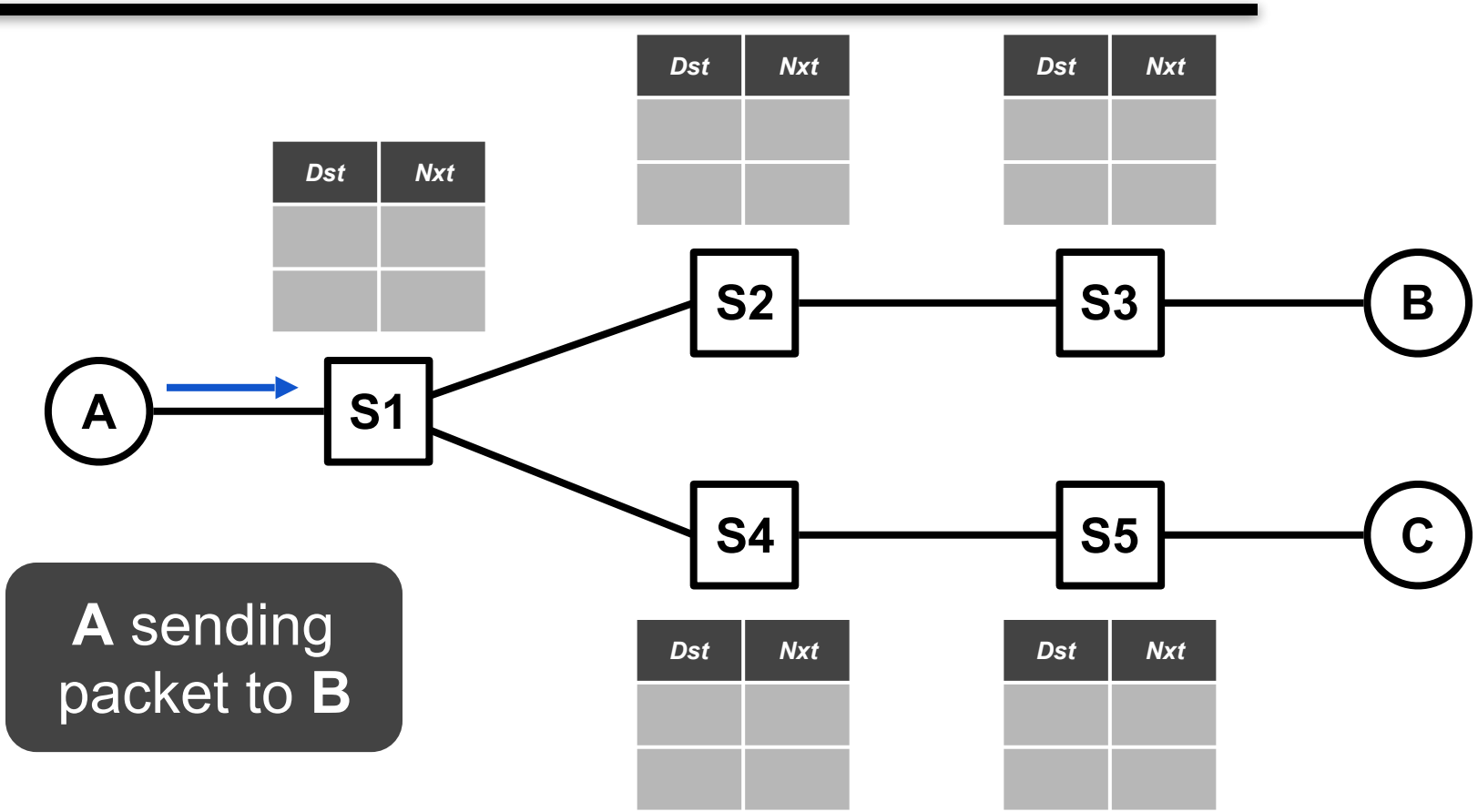
# Learning Switches & The Spanning Tree Protocol

# Learning Switches

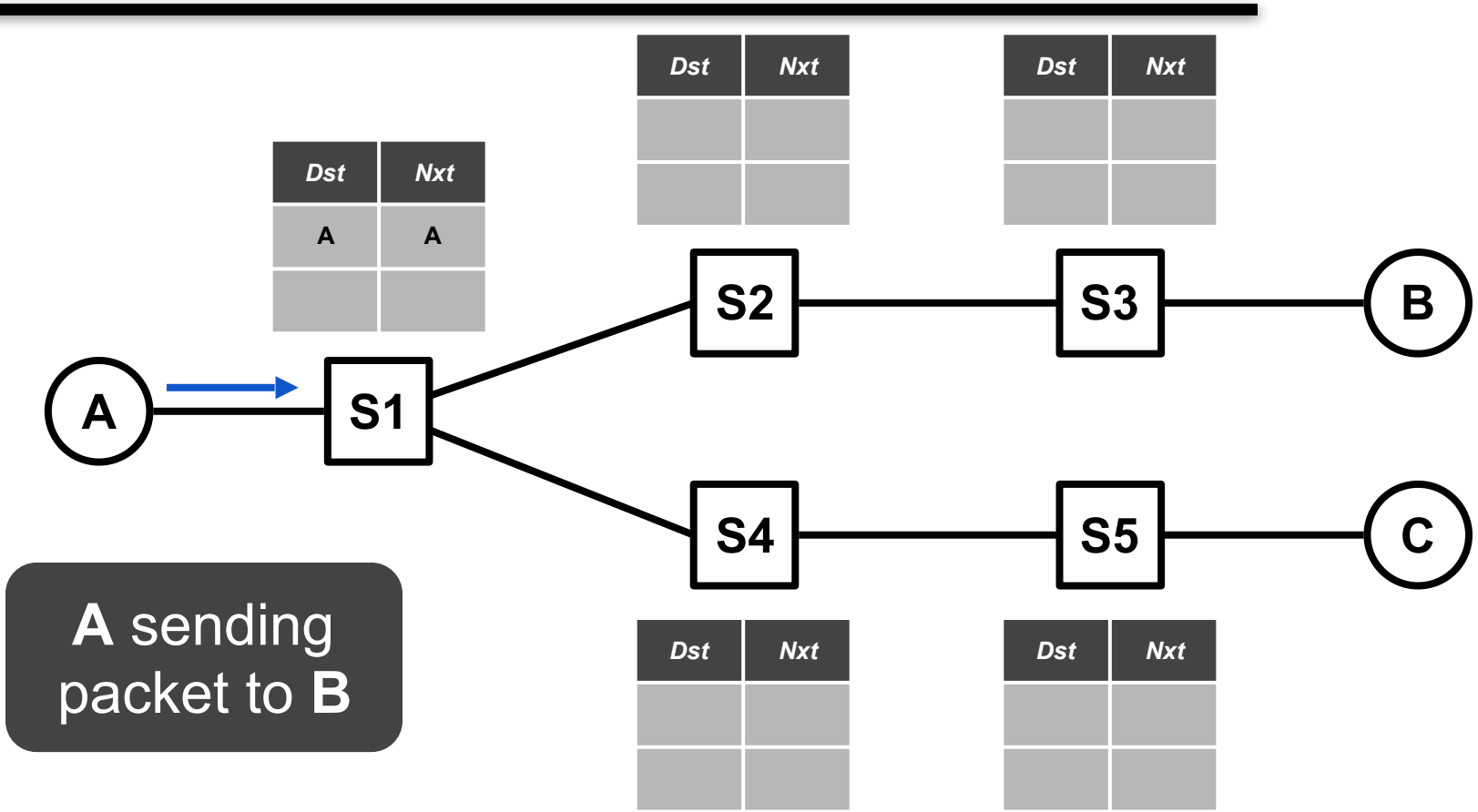
---

- We've 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
- Let's look at 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 it is not routing, but if it looks like a duck, quacks like a duck, and fills in forwarding tables like a duck... 
    - (I may be messing up this metaphor.)

# Learning Switches

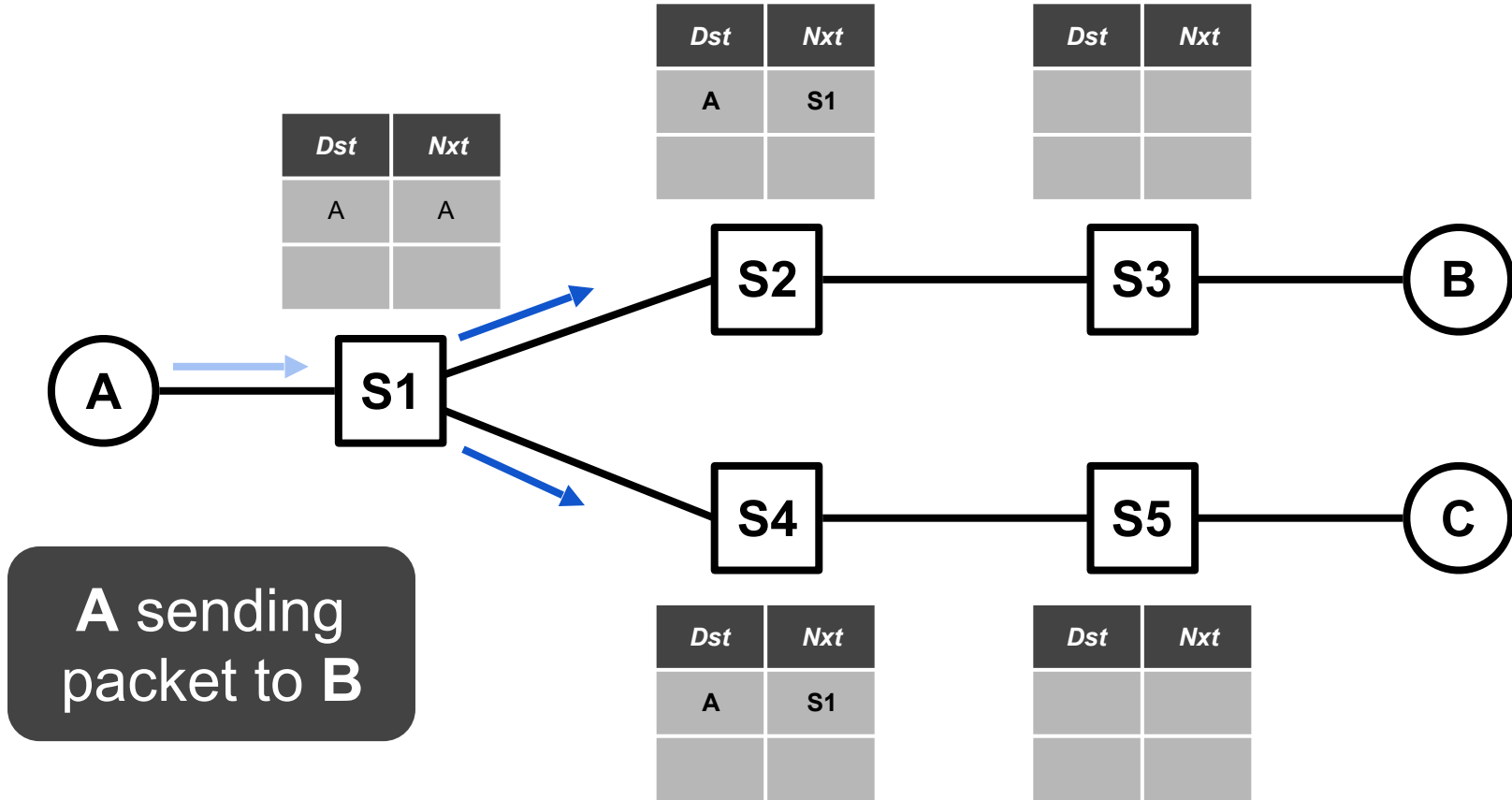


# Learning Switches

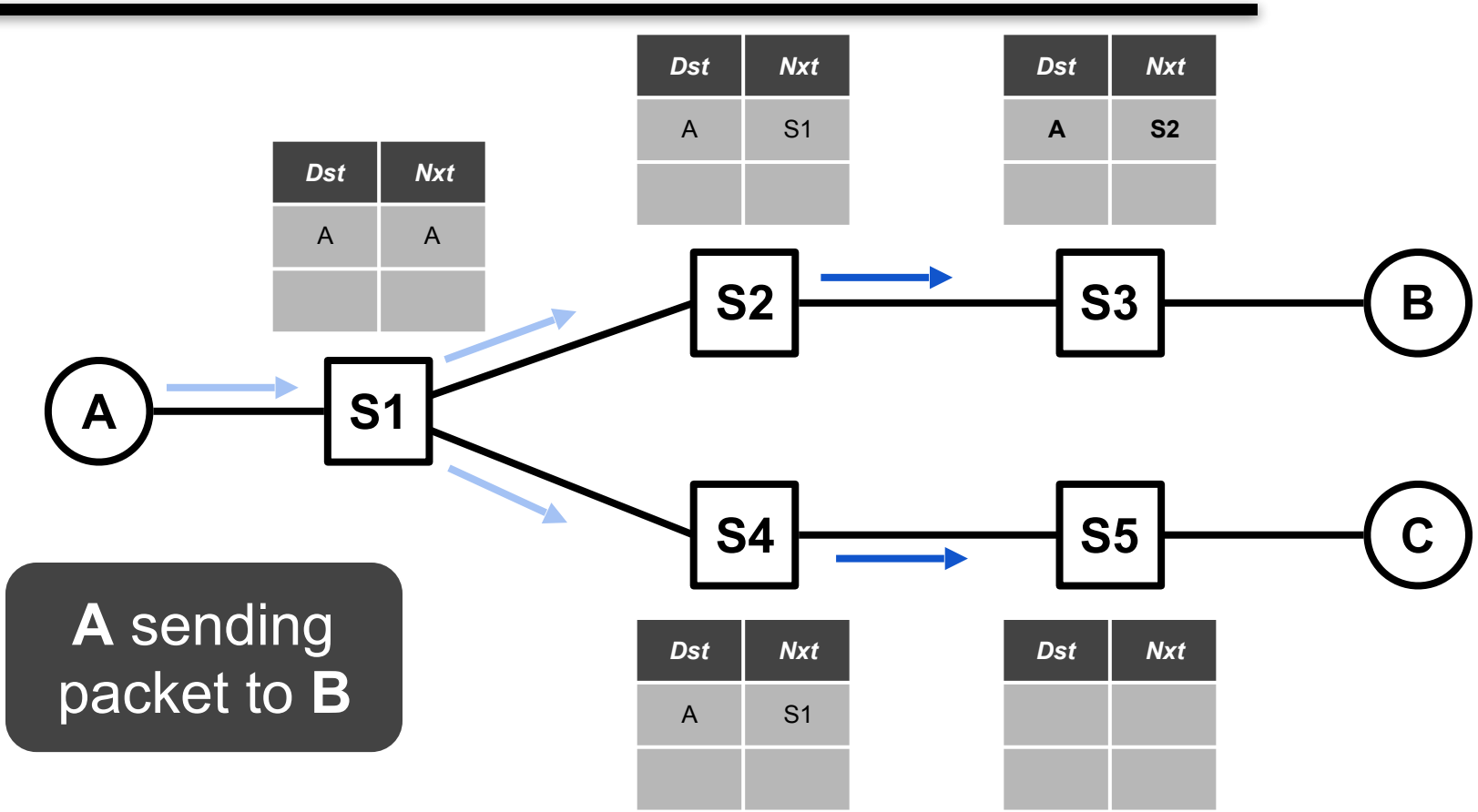




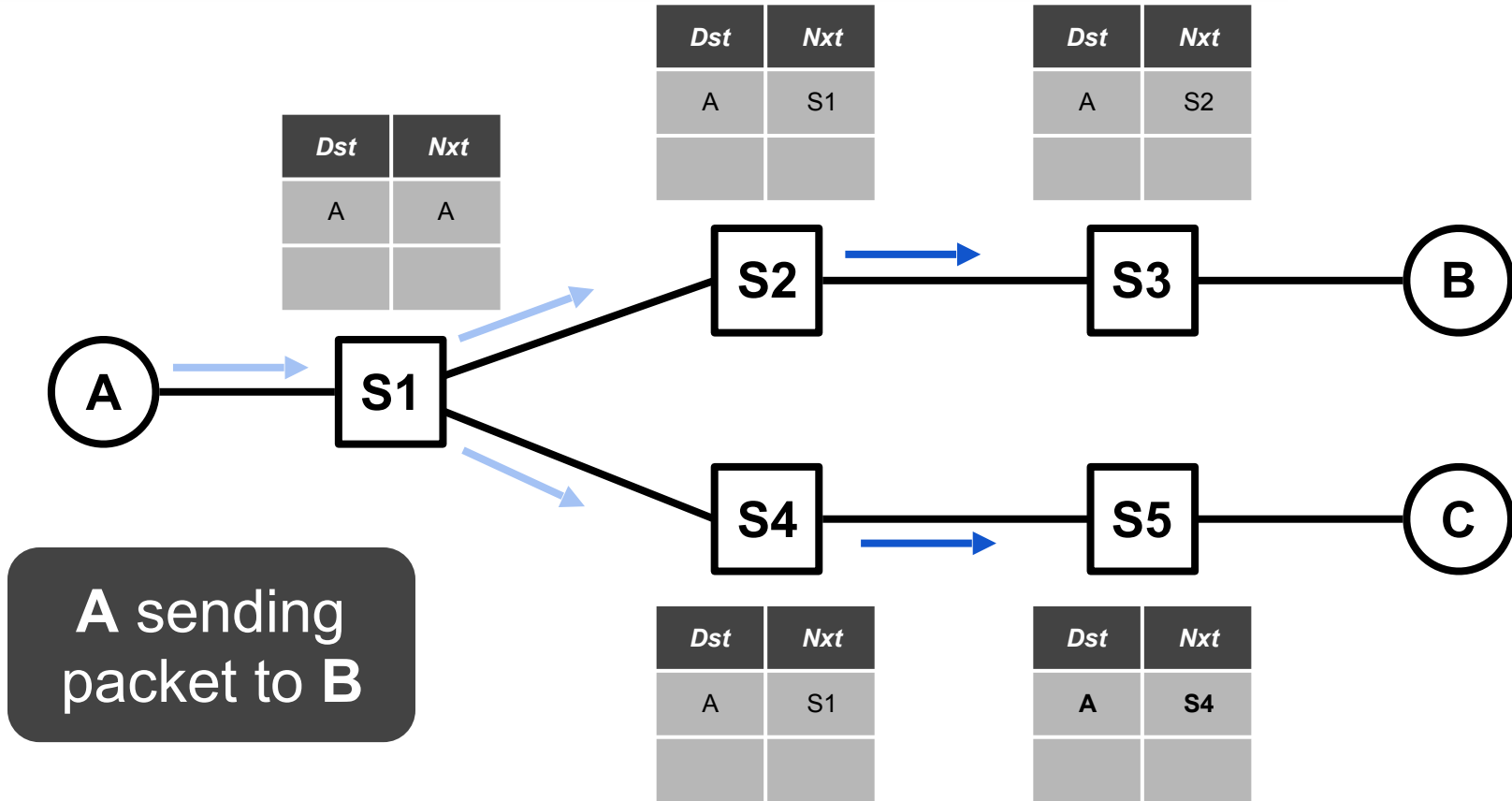
# Learning Switches



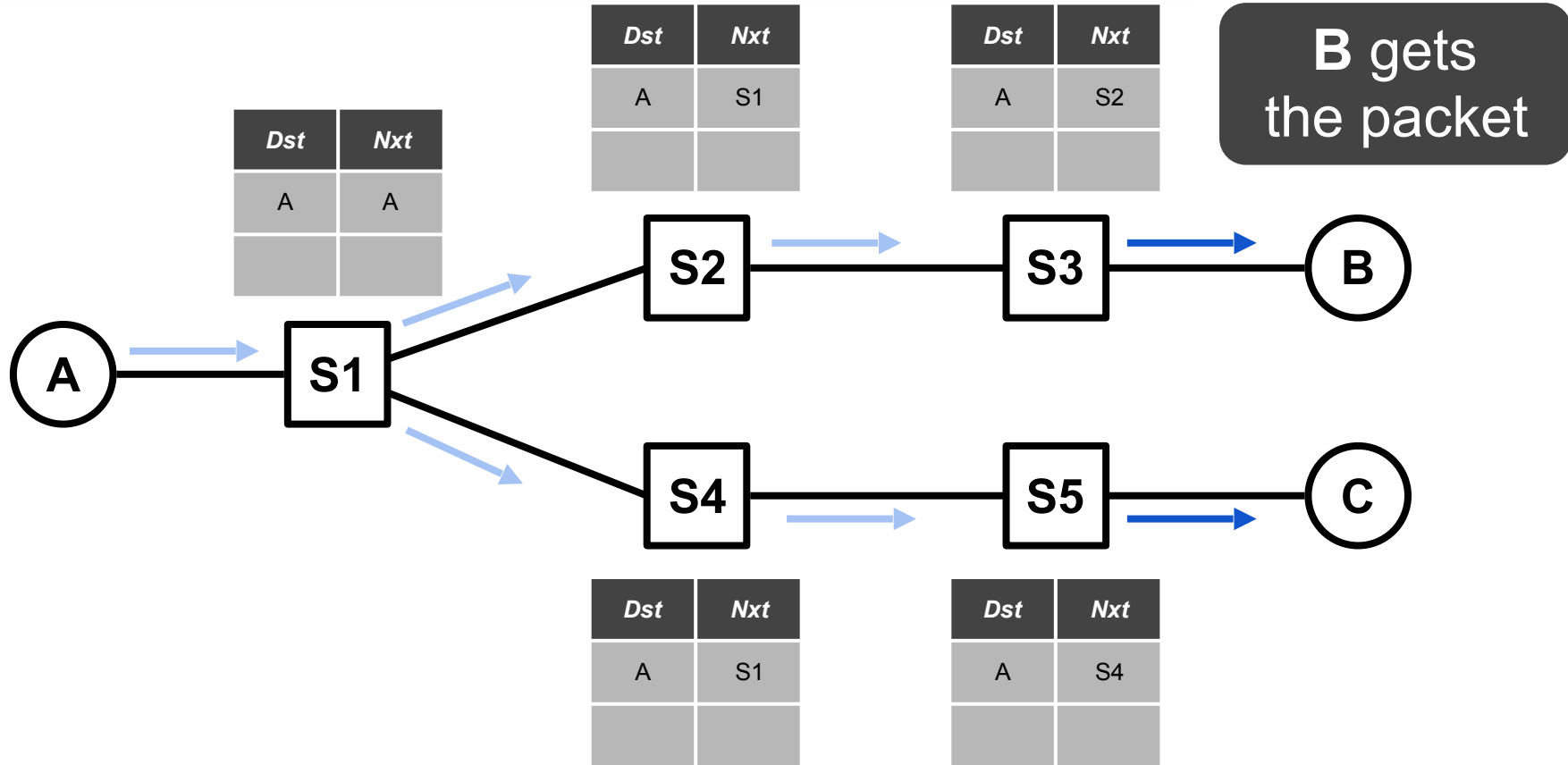
# Learning Switches



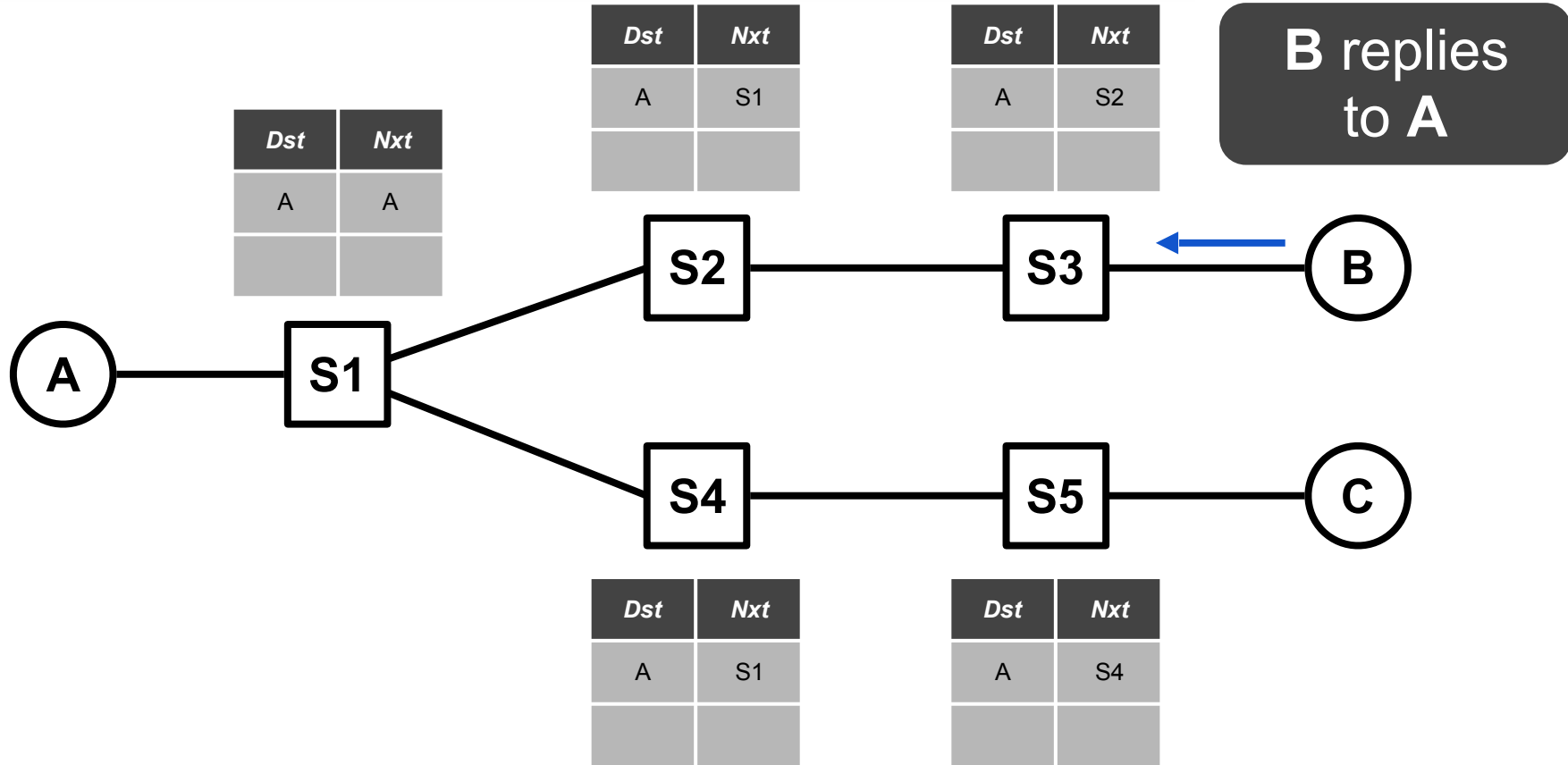
# Learning Switches



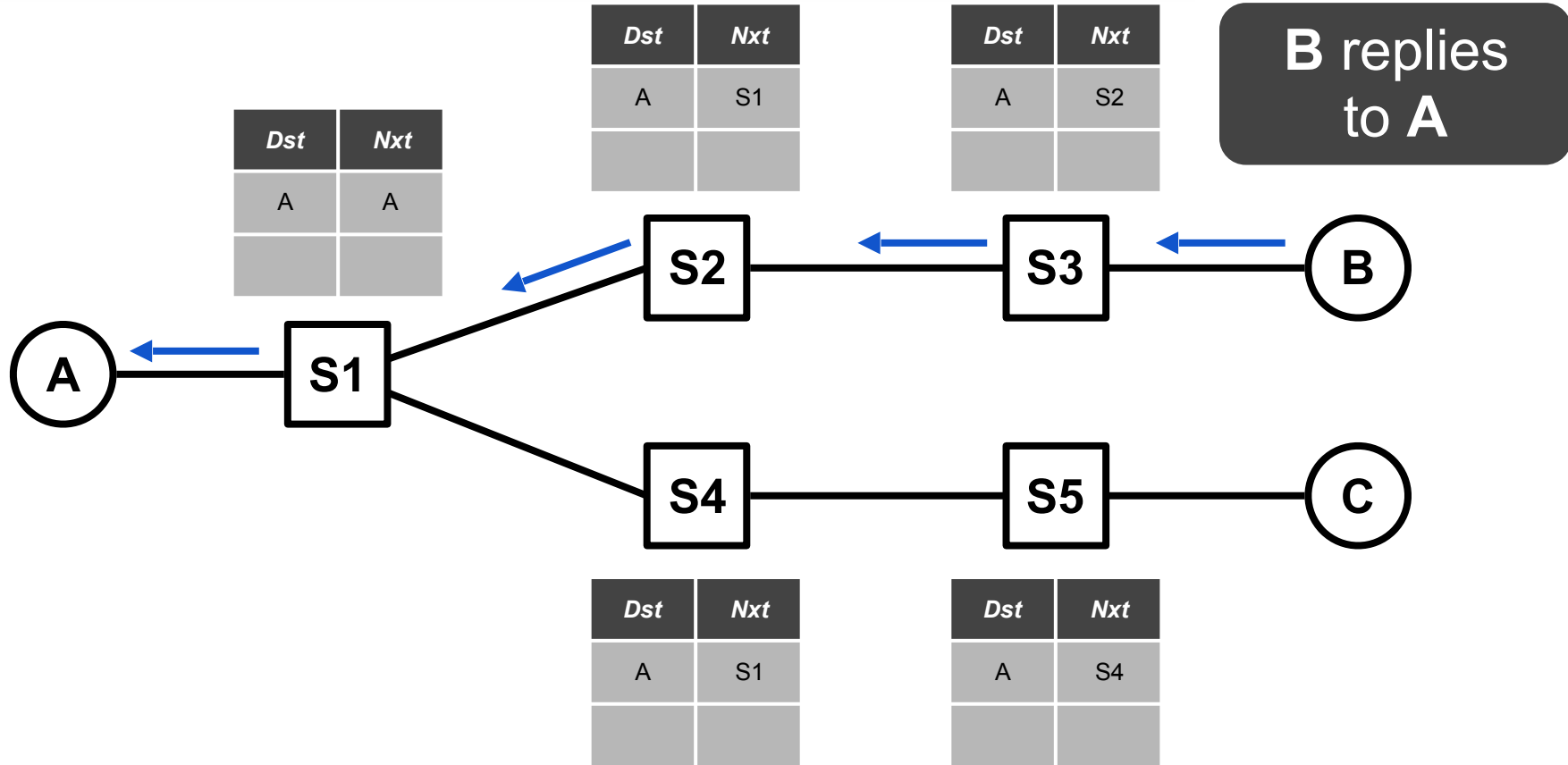
# Learning Switches



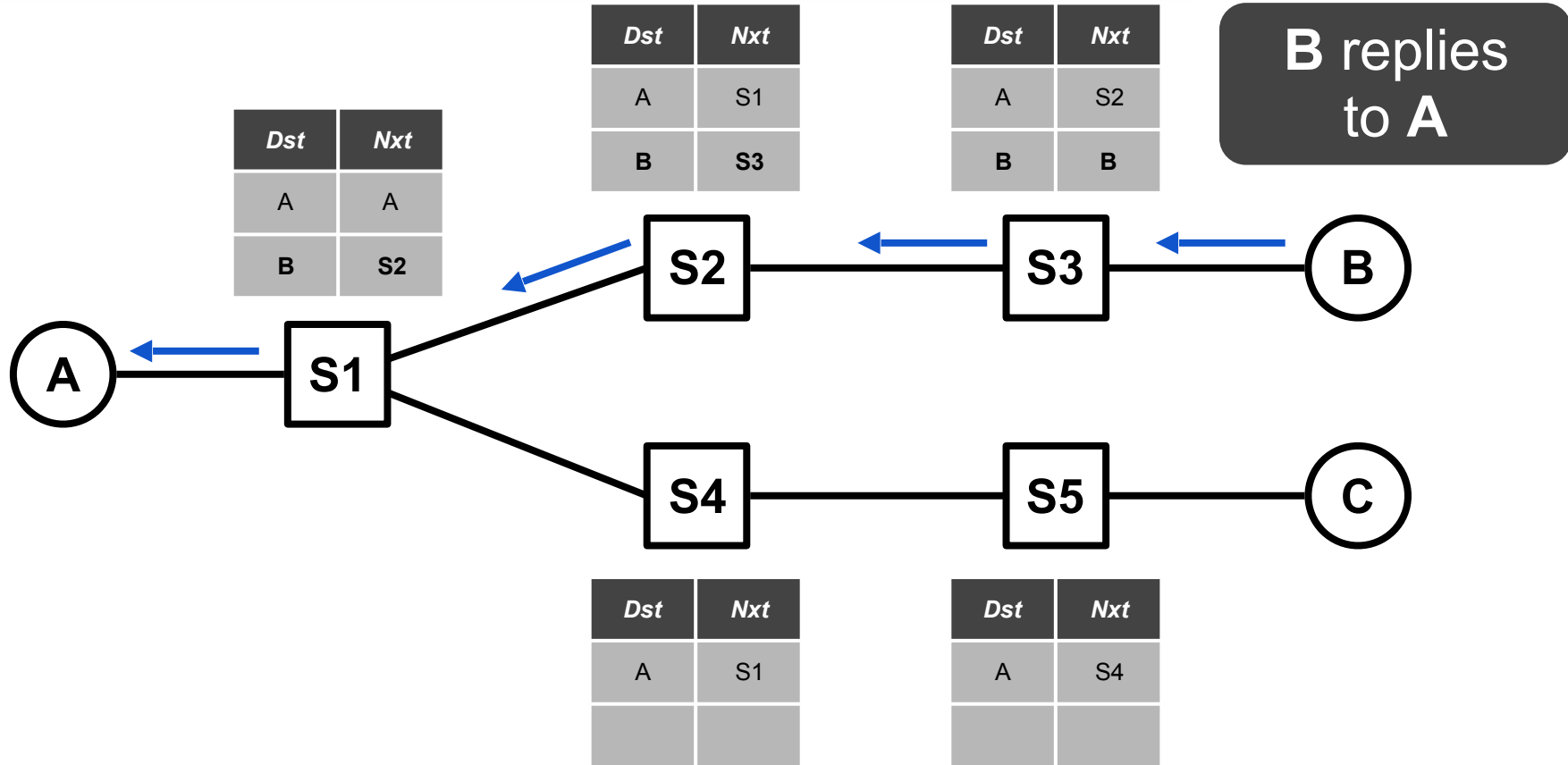
# Learning Switches



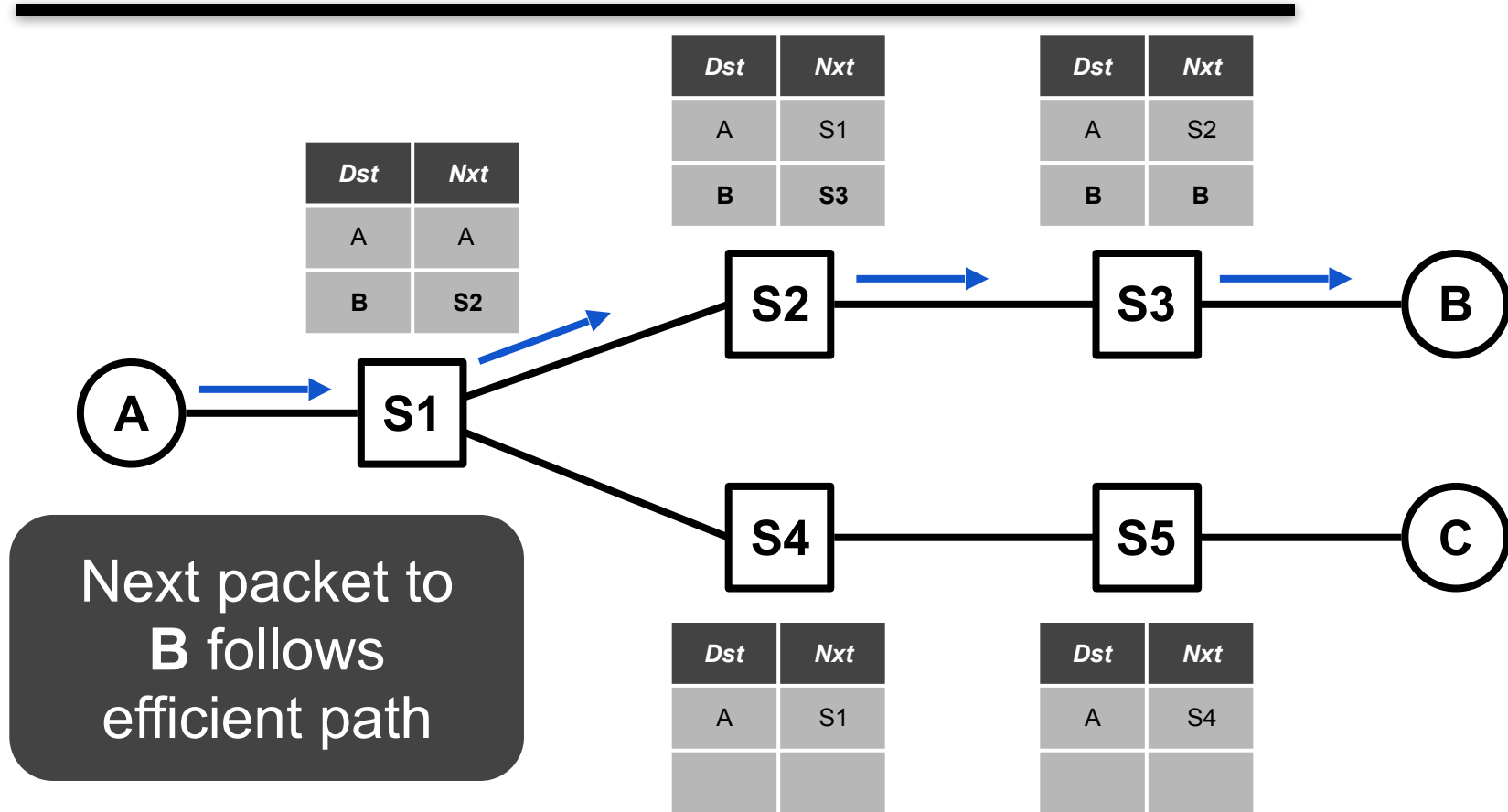
# Learning Switches



# Learning Switches

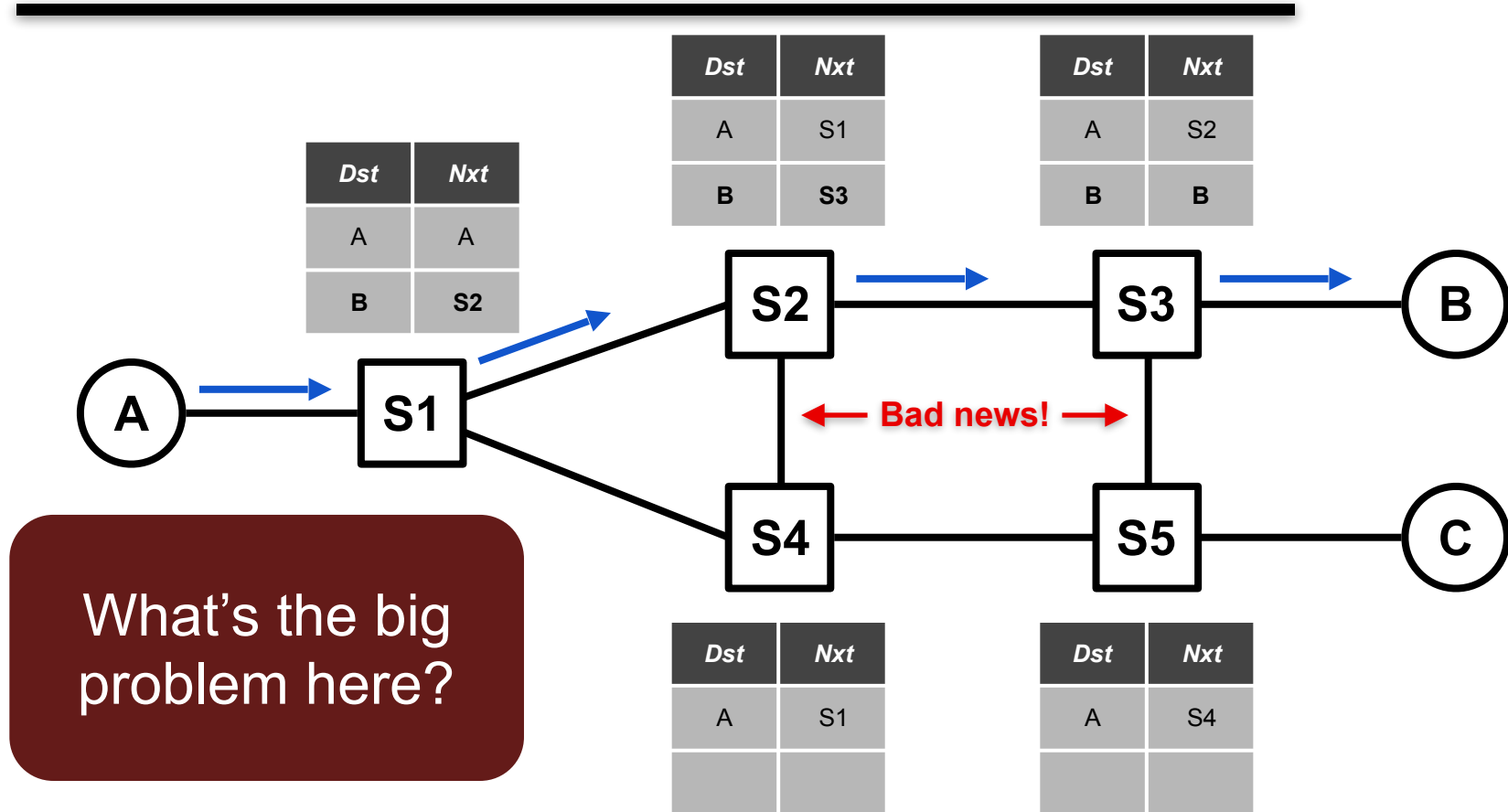


# Learning Switches





# Learning Switches



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

**To Be Continued...**

# Poison Reverse vs. Route Poisoning

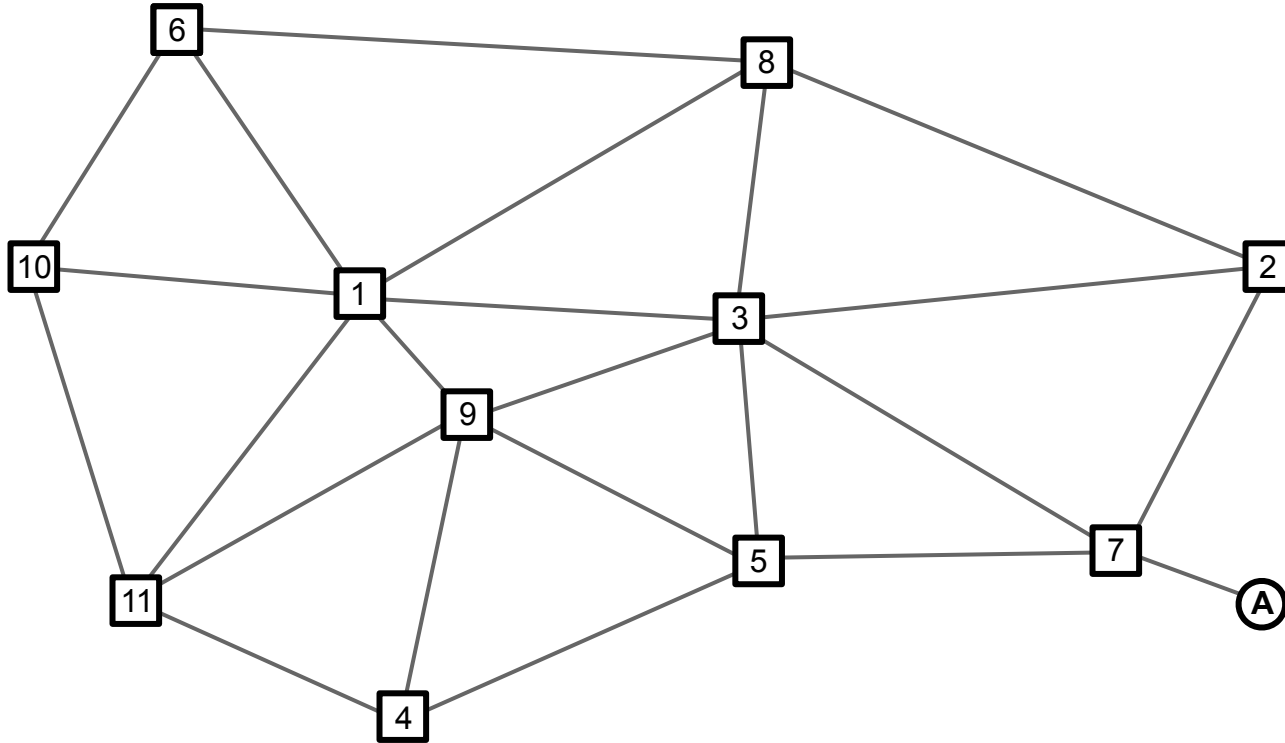
# Poison Reverse vs. Route Poisoning

---

- Poison reverse is just split horizon taken up a notch
  - Try to prevent your next hop from using you as a next hop
  - Done using a special case in your route advertisement code

# Split Horizon and Poison Reverse

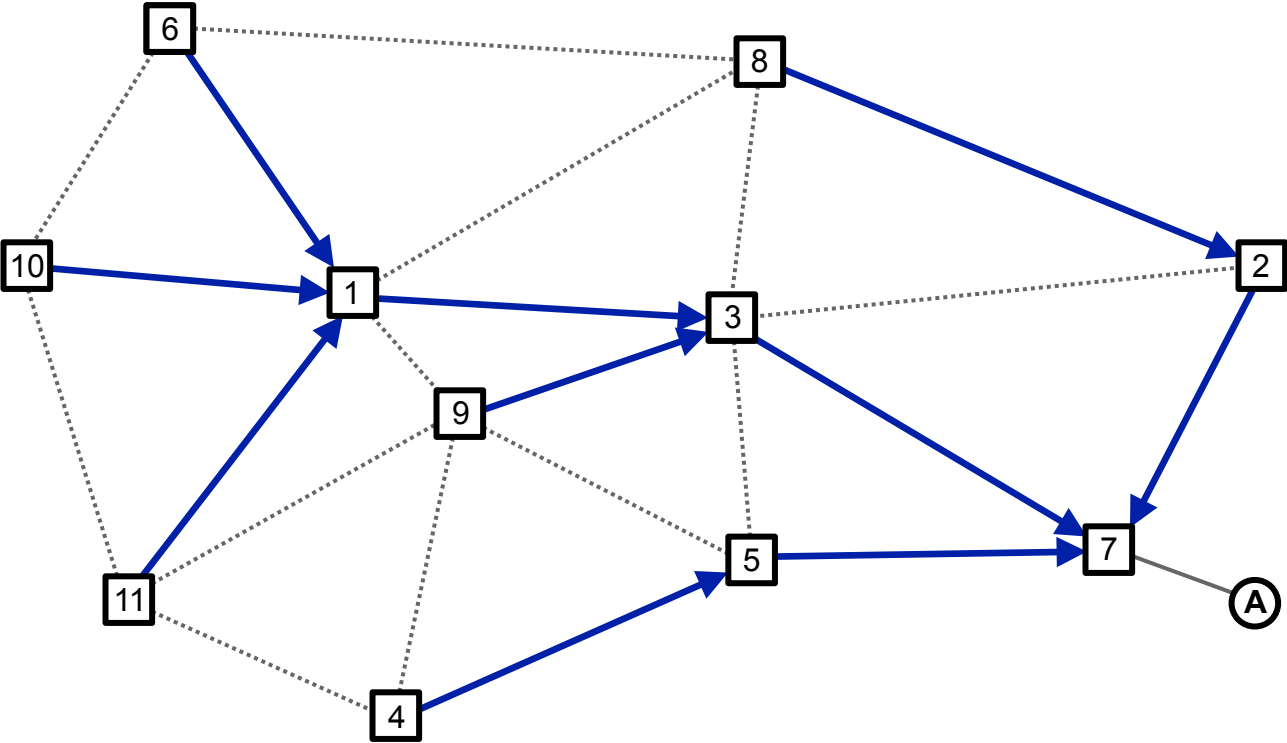
---





# Split Horizon and Poison Reverse

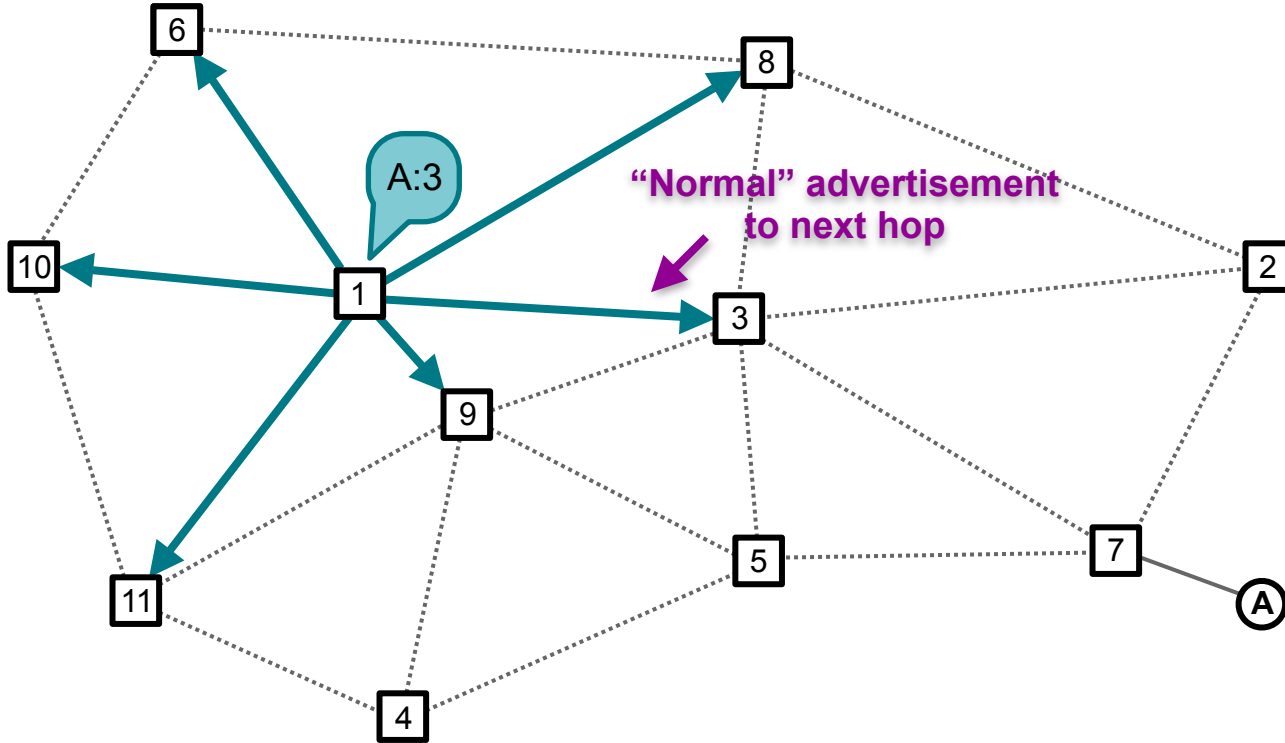
---



Best paths to A

# Split Horizon and Poison Reverse

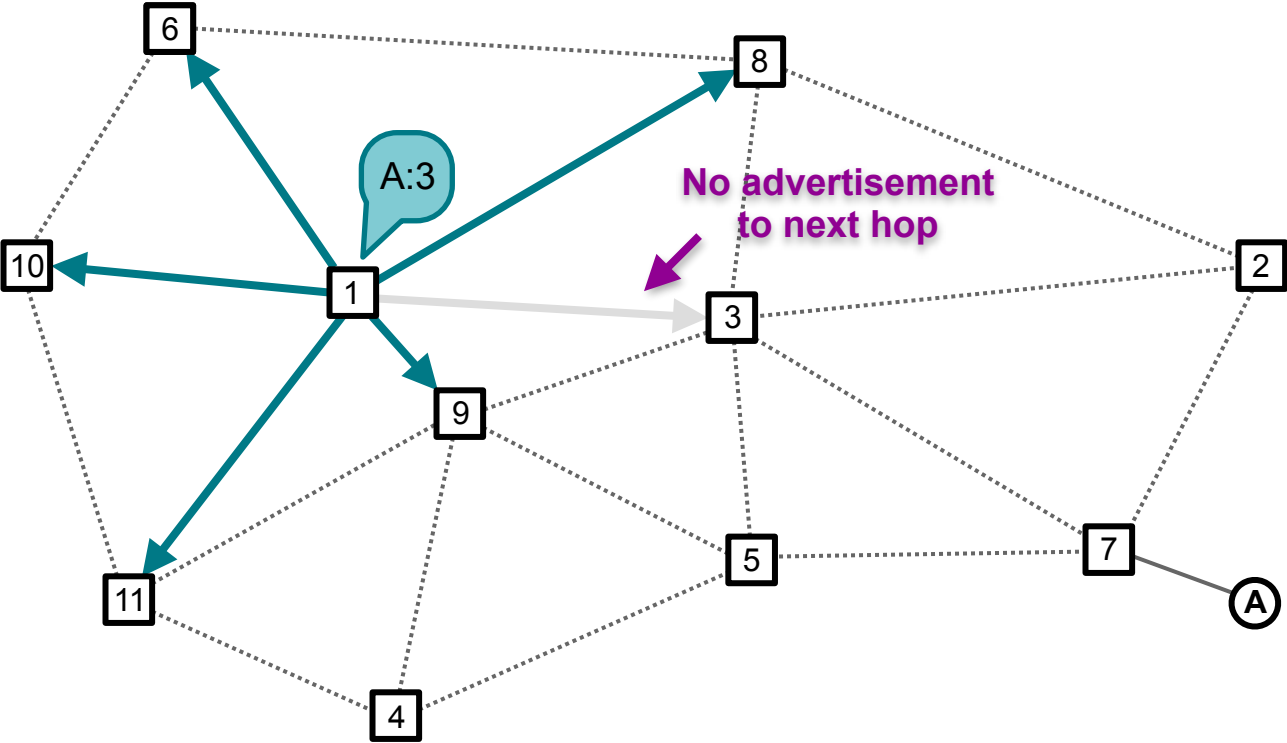
---



Advertisements from switch 1 with no split horizon or poison reverse

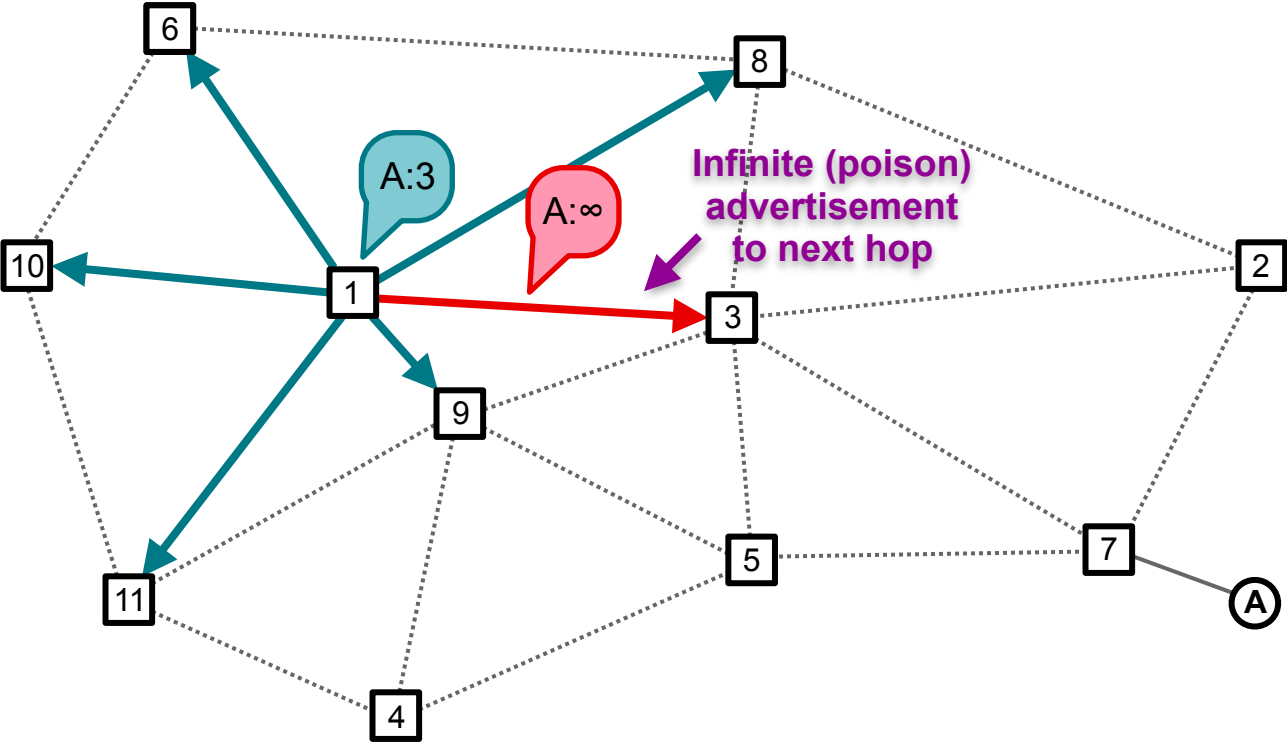
# Split Horizon and Poison Reverse

---



Advertisements from switch 1 with split horizon

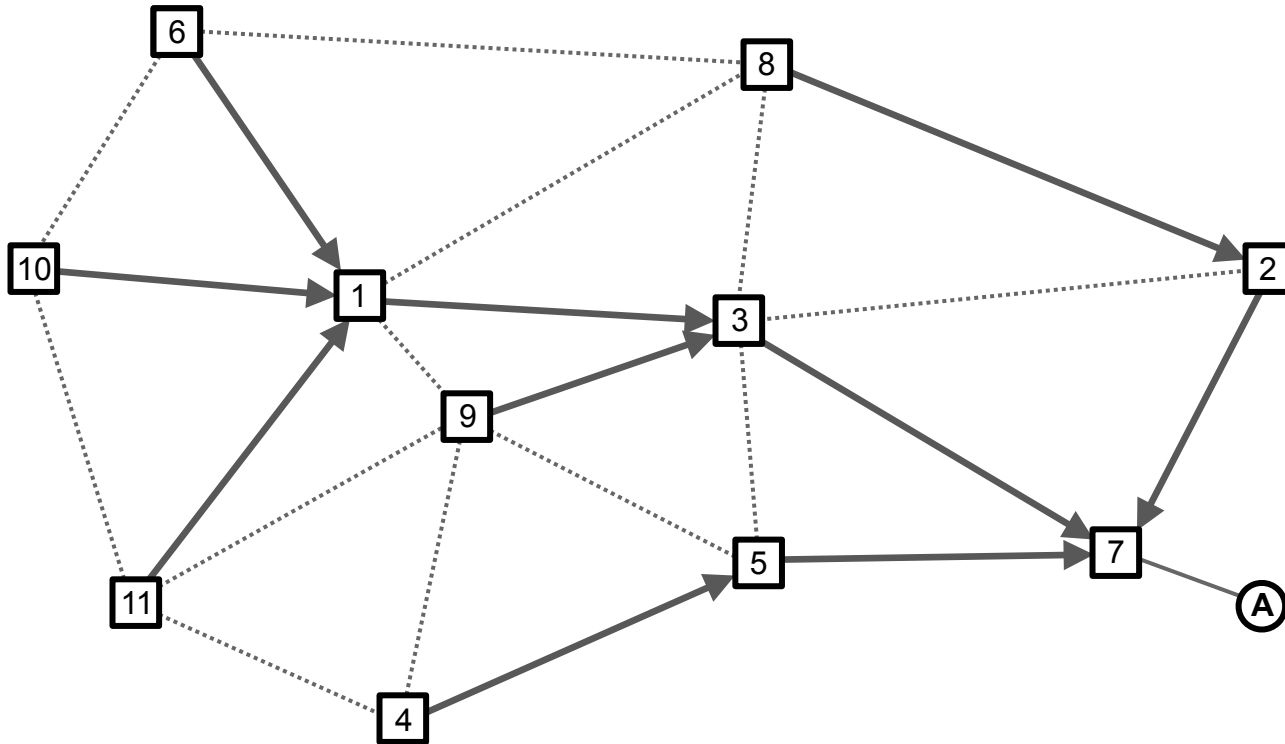
# Split Horizon and Poison Reverse



Advertisements from switch 1 with poison reverse

# Split Horizon and Poison Reverse

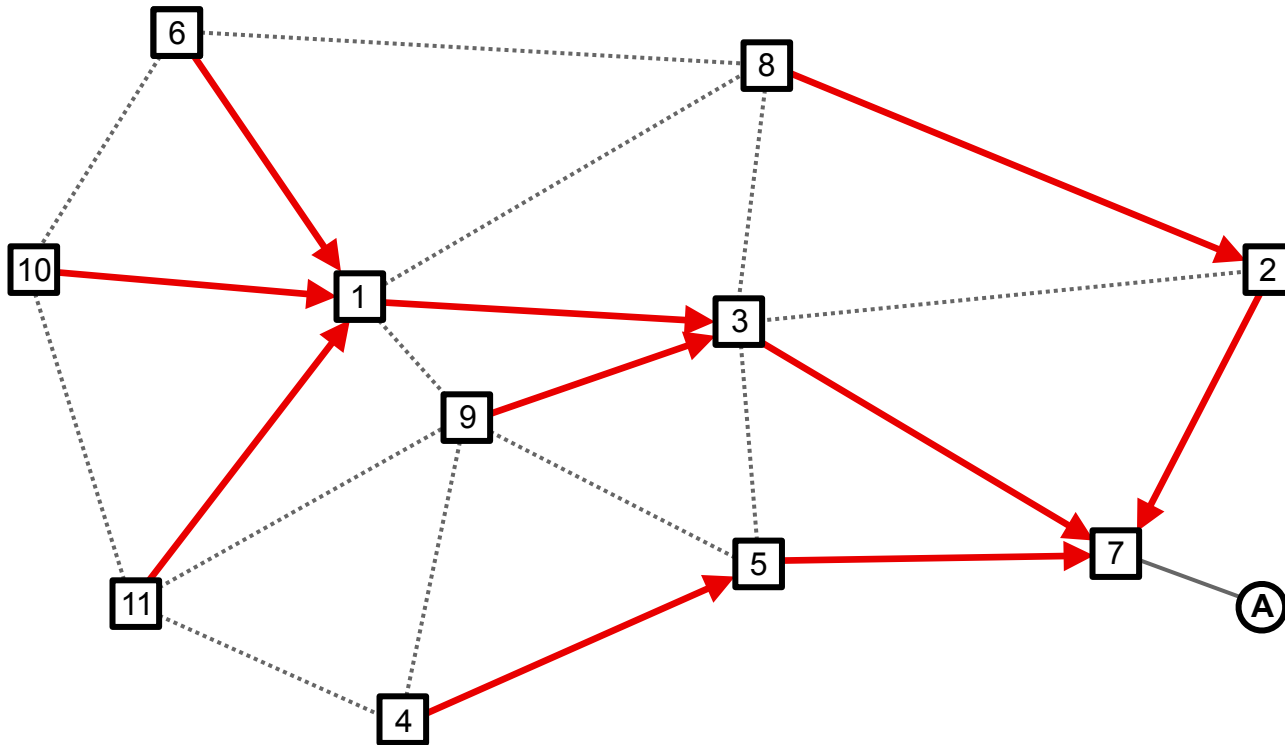
---



All advertisements from all switches not sent due to split horizon

# Split Horizon and Poison Reverse

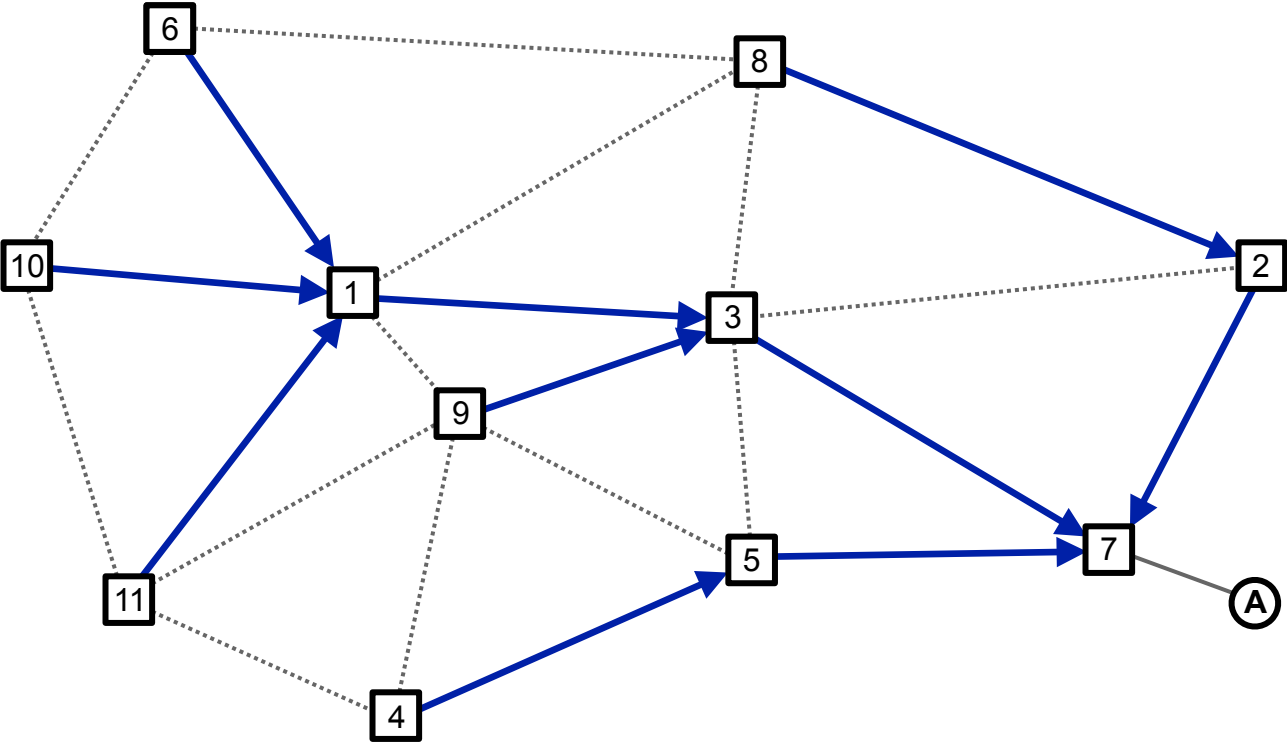
---



All poison reverse advertisements from all switches

# Split Horizon and Poison Reverse

---



Best paths again -- note the relationship to SH / PR advertisements!

# Split Horizon and Poison Reverse

---

- Split horizon:
  - Don't send advertisement to your next hop
- Poison reverse:
  - Send infinite advertisement to your next hop
  
- Both intended to prevent your next hop from using *you* as *their* next hop
  - It wouldn't make any sense!
  - Why would this ever happen?
    - See examples in lecture, discussion, project...



# Poison Reverse vs. Route Poisoning

---

- Poison reverse is just split horizon taken up a notch
  - Try to prevent your next hop from using you as a next hop
  - Done using a special case in your route advertisement code

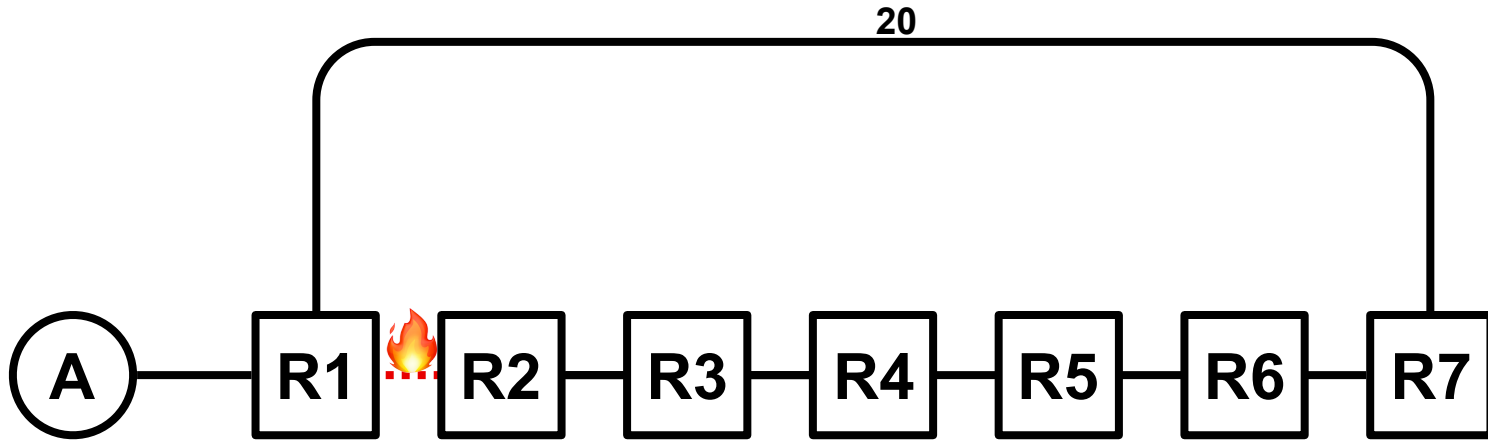
# Poison Reverse vs. Route Poisoning

---

- Poison reverse is just split horizon taken up a notch
  - Try to prevent your next hop from using you as a next hop
  - Done using a special case in your route advertisement code
- Poisoning is about *actively telling neighbors about non-routes*
  - Don't just wait for them to time out...
  - Tell them you are infinitely far away from destination
  - Uses normal advertisement code
    - Actually put an infinite route in your table and it mostly “just works”

# Route Poisoning

---



- With no poisoning, route through R1 will have to expire on each switch consecutively before R7 will accept the alternate route.
- This can take as long as like six expiration intervals, and triggered updates don't help!
- With poisoning, when R2 notices the link go down (either directly or via timeout), it changes distance of route using R1 to infinity (poison).
- This takes more like six *advertisements* to reach R7! Even with only periodic advertisements, this is likely a big savings. With triggered updates, it can be huge!

# Poison Reverse vs. Route Poisoning

---

- Poison reverse is just split horizon taken up a notch
  - Try to prevent your next hop from using you as a next hop
  - Done using a special case in your route advertisement code
- Poisoning is about *actively telling neighbors about non-routes*
  - Don't just wait for them to time out...
  - Tell them you are infinitely far away from destination
  - Uses normal advertisement code
    - Actually put an infinite route in your table and it mostly “just works”

# Poison Reverse vs. Route Poisoning

---

- Poison reverse is just split horizon taken up a notch
  - Try to prevent your next hop from using you as a next hop
  - Done using a special case in your route advertisement code
- Poisoning is about *actively telling neighbors about non-routes*
  - Don't just wait for them to time out...
  - Tell them you are infinitely far away from destination
  - Uses normal advertisement code
    - Actually put an infinite route in your table and it mostly “just works”
- Note: Receiver of  $\infty$  advertisement can't tell the difference between them!
  - Just knows not to use sender as part of its path!