Network Layer

Pavlos Grigoriadis - pgrigor@csd.uoc.gr
Christina Papachristoudi - chrisp@csd.uoc.gr

Network Layer

Data Plane

How a router forwards the packets that arrive in its **incoming interfaces** to **outgoing interfaces**

Consists of:

- Header inspection
- Implemented mainly in hardware
- Follows the instructions given by the Control Plane
- Forwarding Table

Control Plane

How a packet is routed among the routers (**end-to-end** routing).

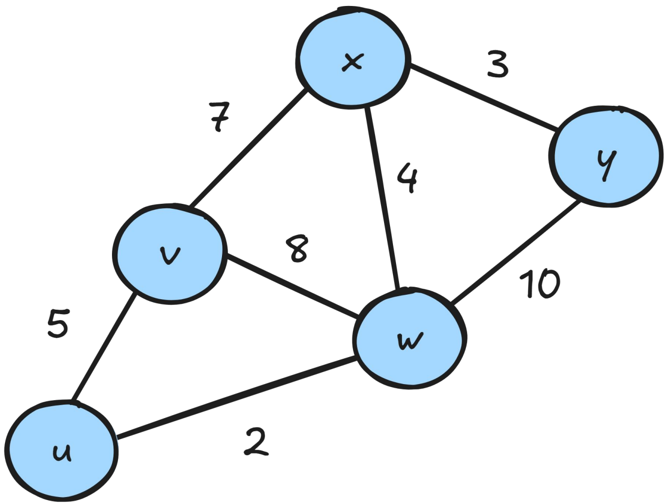
Consists of:

- Implemented mainly in software
- Routing Algorithms and Protocols
- Routing Table

Control Plane

Routing Algorithms

- Goal: to find "good" paths from a source to a destination
- What is a good path?
  - **Least-cost** Path
  - If all link costs are the same then the least-cost path is the **Shortest Path**
- Mainly two types of routing algorithms
  - **Centralized (Link-State)**
  - **Decentralized**

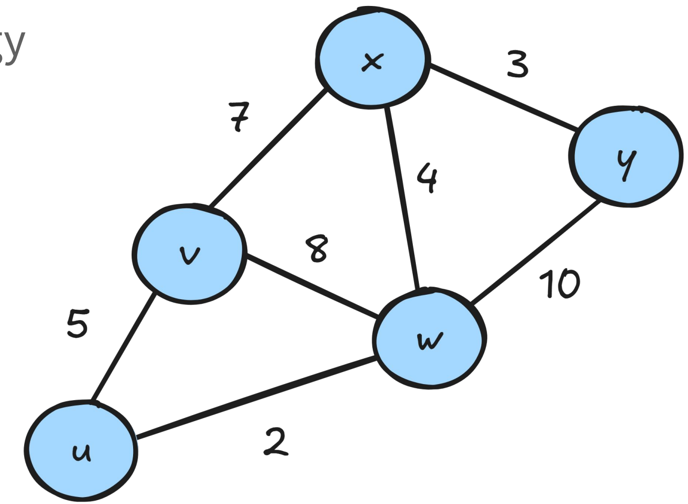


Link-State Routing Algorithms

- **Centralized** routing algorithms
- Requires knowledge of the **entire topology**
- That means it takes as input:
  - All the links **between nodes**
  - All the link **costs**
- How does each node learn all this?
  - **Link-state broadcast** algorithm
  - Each node broadcasts its **link-state information (links to neighbors, costs)**
  - At the end every node has the **same complete view of the topology**
  - Now each node can run the LS algorithm and find the same least-cost paths

Dijkstra Algorithm

- A **link-state** algorithm
- Computes all the **least-cost paths** from a **source node** to **all other nodes** of the topology
- For **k nodes** it needs **k iterations** to complete

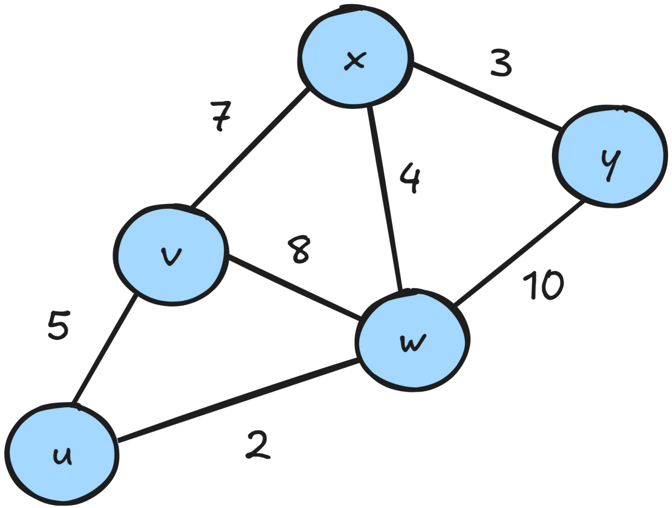


Dijkstra Algorithm

Find the least-cost paths from u to every other router

- **D(v)**: Distance from u to v
- **p(v)**: previous node of v on the current path

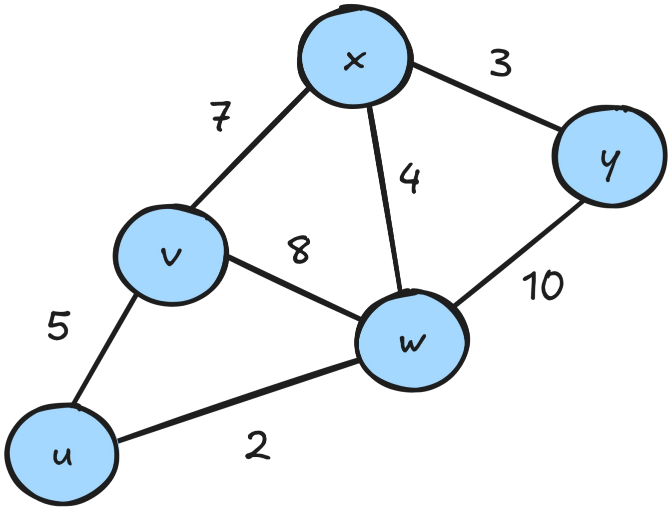
Nodes	D(v), p(v)	D(w), p(w)	D(x), p(x)	D(y), p(y)
u	5, u	2, u	∞, -	∞, -



Dijkstra Algorithm

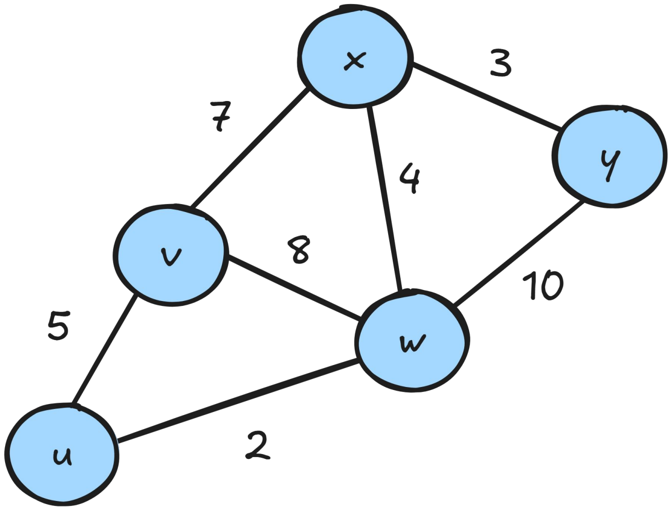
- Choose w because it has the smallest distance
- Add it to the nodes

Nodes	D(v), p(v)	D(w), p(w)	D(x), p(x)	D(y), p(y)
u	5, u	**2, u**	∞, -	∞, -
uw	5, u *(10, w)*	-	6, w	12, w



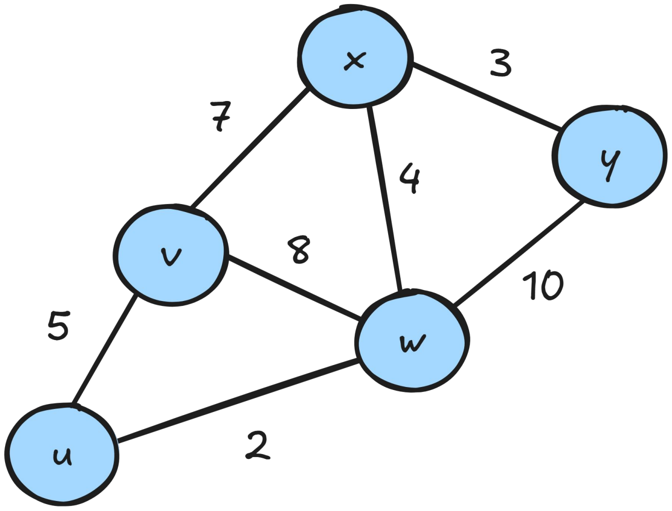
Dijkstra Algorithm

Nodes	D(v), p(v)	D(w), p(w)	D(x), p(x)	D(y), p(y)
u	5, u	2, u	∞, -	∞, -
uw	**5, u**	-	6, w	12, w
uwv	-	-	6, w *(12, v)*	12, w



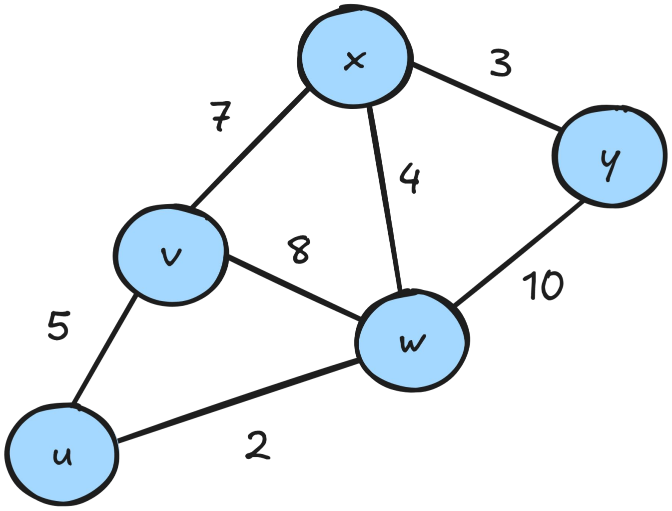
Dijkstra Algorithm

Nodes	D(v), p(v)	D(w), p(w)	D(x), p(x)	D(y), p(y)
u	5, u	2, u	∞, -	∞, -
uw	5, u	-	6, w	12, w
uwv	-	-	**6, w**	12, w
uwvx	-	-	-	12, w *(9, x)*



Dijkstra Algorithm

Nodes	D(v), p(v)	D(w), p(w)	D(x), p(x)	D(y), p(y)
u	5, u	2, u	∞, -	∞, -
uw	5, u	-	6, w	12, w
uwv	-	-	6, w	12, w
uwvx	-	-	-	**9, x**
uwvxy	-	-	-	-

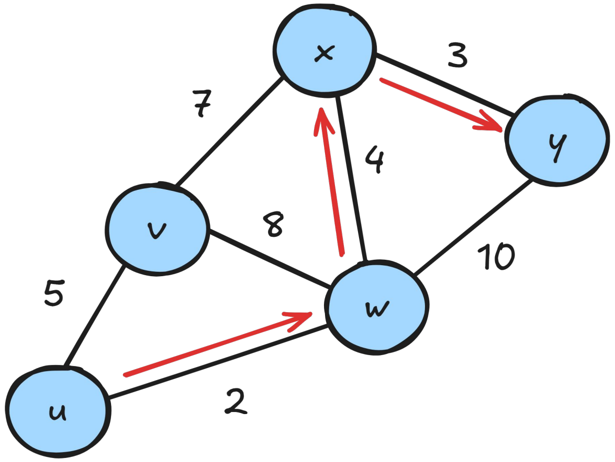


Dijkstra Algorithm

So to find the path from, let's say, **u → y**, we choose y and go backwards:

- Previous of y? p(y) = x
- p(x) = w
- p(w) = u

Path u→y: **u → w → x → y**

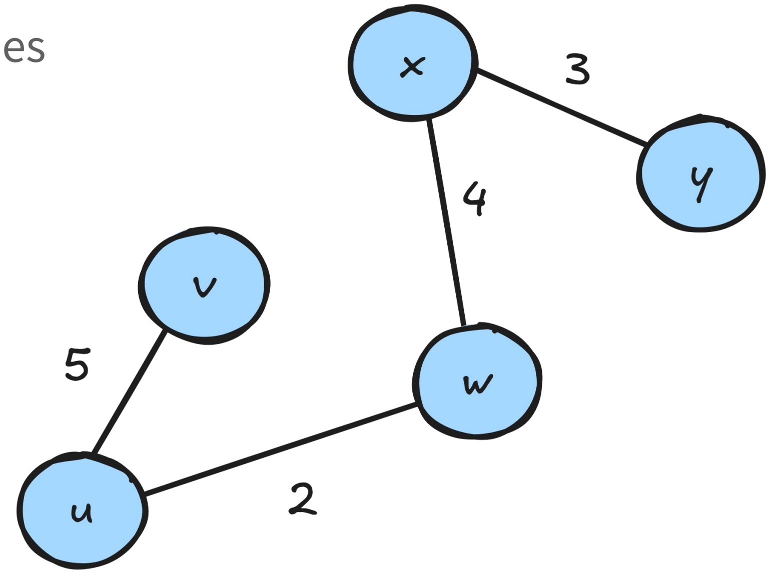


Nodes	D(v), p(v)	D(w), p(w)	D(x), p(x)	D(y), p(y)
u	5, u	2, **u**	∞, -	∞, -
uw	5, u	-	6, w	12, w
uwv	-	-	6, **w**	12, w
uwvx	-	-	-	9, **x**
uwvxy	-	-	-	-

Dijkstra Algorithm - Least-Cost Graph

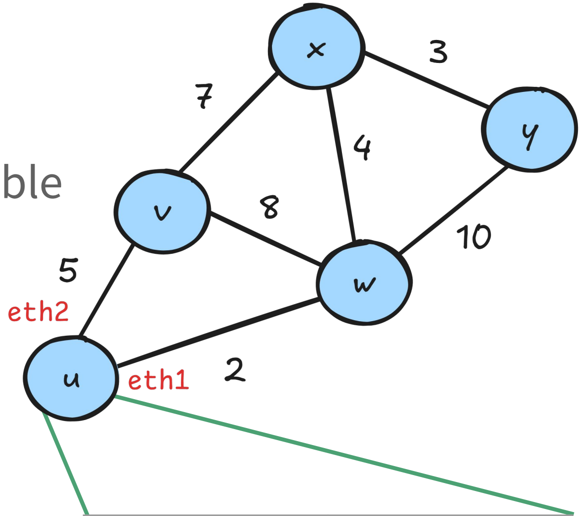
To find the least-cost graph:

- Find least-cost paths from **u** to all other nodes
- Keep only the **nodes and links** that exist in these paths



Populating Forwarding Table

- Routing algorithms are used to fill up the forwarding table
- Based on the least-cost paths:
  - Path to y: **u → w → x → y**
  - So in order to reach **y** → send to **w**
  - So for destination **y** the **outgoing interface: eth1**
  - And so on…



Destination	Interface
w	eth1
v	eth2
x	eth1
y	eth1

Decentralized Algorithms

- Each node only knows about the link costs that are **directly connected to it**
- Each node gets/sends updates from/to its **neighbors (only!)**
- **Iterative**
    - Based on these updates each node calculates the new least-cost path
    - These updates continue until there aren’t any more changes
    - The algorithm has **converged**

Distance Vector

- **Decentralized** routing algorithm
- Keeps a vector of all the costs (distances) to the other nodes
  - hence the name distance vector
- Uses the **Bellman-Ford equation**

Distance Vector

Initialization state

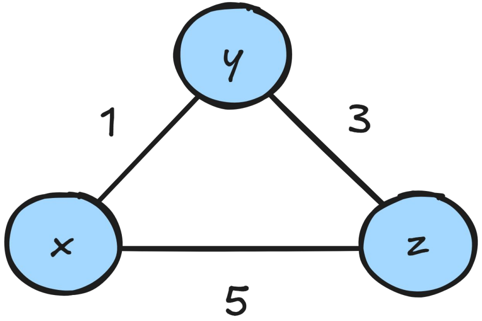


Table of node x

	x	y	z
x	0	1	5
y	∞	∞	∞
z	∞	∞	∞

Table of node y

	x	y	z
x	∞	∞	∞
y	1	0	3
z	∞	∞	∞

Table of node z

	x	y	z
x	∞	∞	∞
y	∞	∞	∞
z	5	3	0

Distance Vector

Each node advertises its table to the neighbors

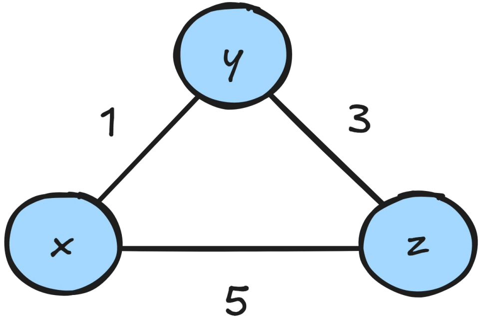


Table of node x

	x	y	z
x	0	1	5
y	**1**	**0**	**3**
z	**5**	**3**	**0**

Table of node y

	x	y	z
x	**0**	**1**	**5**
y	1	0	3
z	**5**	**3**	**0**

Table of node z

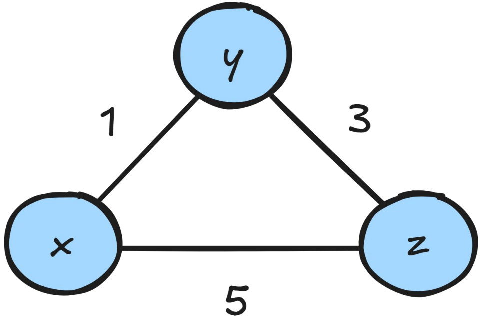
	x	y	z
x	**0**	**1**	**5**
y	**1**	**0**	**3**
z	5	3	0

x will advertise table to **y, z**

y will advertise table to **x, z**

z will advertise table to **x, y**

Distance Vector



Bellman-Ford Equation:

$$D_x(y) = \min_v \{ c(x, v) + D_v(y) \}$$

- **D_x(y)**: the least cost from x to y
- **c(x, v)**: the current cost from x to v (based on the current value of node x's vector)
- **D_v(y)**: the least cost from a node v to y

Algorithm runs for every **node v** that is a **neighbor of x**

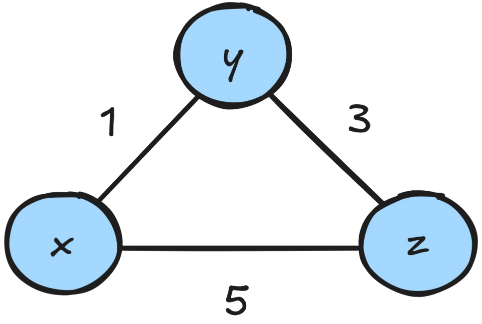
For neighbor z:
- **c(x, z) = 5**
- **D_z(y) = 3**

For neighbor y:
- **c(x, y) = 1**
- **D_y(y) = 0**

Table of node x

	x	y	z
x	0	1	5
y	1	0	3
z	5	3	0

Distance Vector



Bellman-Ford Equation:

$$D_x(y) = \min_v \{ c(x, v) + D_v(y) \}$$

- **D_x(y)**: the least cost from x to y
- **c(x, v)**: the current cost from x to v (based on the current value of node x's vector)
- **D_v(y)**: the least cost from a node v to y

Algorithm runs for every **node v** that is a **neighbor of x**

For neighbor z:

- **c(x, z) = 5**
- **D_z(y) = 3**

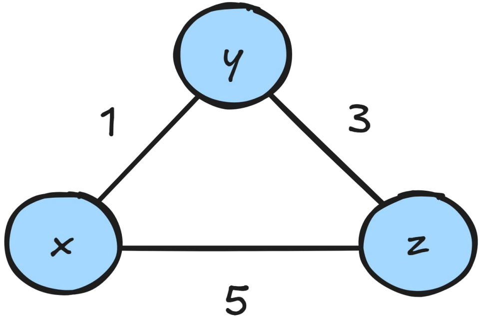
For neighbor y:

- **c(x, y) = 1**
- **D_y(y) = 0**

Table of node x

	x	y	z
x	0	1	**5**
y	1	0	3
z	5	3	0

Distance Vector



Bellman-Ford Equation:

$$D_x(y) = \min_v \{ c(x, v) + D_v(y) \}$$

- **D_x(y)**: the least cost from x to y
- **c(x, v)**: the current cost from x to v (based on the current value of node x's vector)
- **D_v(y)**: the least cost from a node v to y

Algorithm runs for every **node v** that is a **neighbor of x**

For neighbor z:

- **c(x, z) = 5**
- **D_z(y) = 3**

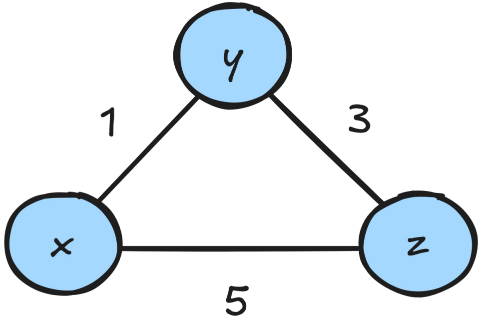
For neighbor y:

- **c(x, y) = 1**
- **D_y(y) = 0**

Table of node x

	x	y	z
x	0	1	5
y	1	0	3
z	5	**3**	0

Distance Vector



Bellman-Ford Equation:

$$D_x(y) = \min_v \{ c(x, v) + D_v(y) \}$$

- **$D_x(y)$**: the least cost from x to y
- **c(x, v)**: the current cost from x to v (based on the current value of node x's vector)
- **$D_v(y)$**: the least cost from a node v to y

Algorithm runs for every **node v** that is a **neighbor of x**

For neighbor z:

- **c(x, z) = 5**
- **$D_z(y)$ = 3**

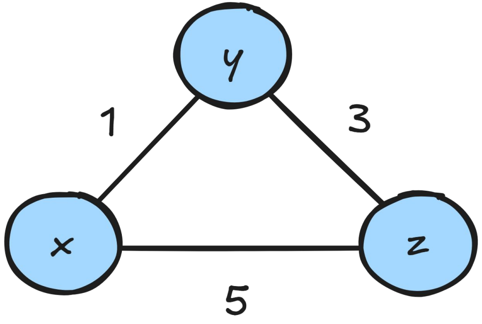
For neighbor y:

- **c(x, y) = 1**
- **$D_y(y)$ = 0**

Table of node x

	x	y	z
x	0	**1**	5
y	1	0	3
z	5	3	0

Distance Vector



Bellman-Ford Equation:

$$D_x(y) = \min_v \{ c(x, v) + D_v(y) \}$$

- **$D_x(y)$**: the least cost from x to y
- **c(x, v)**: the current cost from x to v (based on the current value of node x's vector)
- **$D_v(y)$**: the least cost from a node v to y

Algorithm runs for every **node v** that is a **neighbor of x**

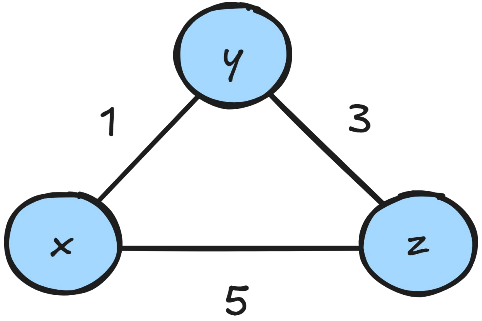
For neighbor z:
- **c(x, z) = 5**
- **$D_z(y) = 3$**

For neighbor y:
- **c(x, y) = 1**
- **$D_y(y) = 0$**

Table of node x

	x	y	z
x	0	1	5
y	1	0	3
z	5	3	0

Distance Vector



Bellman-Ford Equation:

$$D_x(y) = \min_v \{ c(x, v) + D_v(y) \}$$

- For z: $D_x(y) = c(x, z) + D_z(y) = 5 + 3 = 8$
- For y: $D_x(y) = c(x, y) + D_y(y) = 1 + 0 = 1$

So finally: $D_x(y) = \min_v \{ c(x, v) + D_v(y) \} = \min(8, 1) =$ **1**

- We re-calculated the distance from to x to y based on the neighbors' updates.
- We update x's vector with the new value.
(In this case it stays the same = 1)

Table of node x

	x	y	z
x	0	**1**	5
y	1	0	3
z	5	3	0

Distance Vector

Each node re-calculates its vector taking into account the vectors of its neighbor

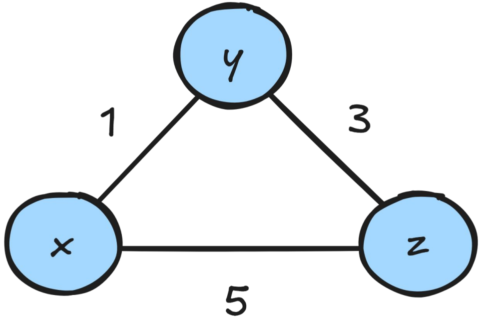


Table of node x

	x	y	z
x	0	1	5
y	1	0	3
z	5	3	0

Table of node y

	x	y	z
x	0	1	5
y	1	0	3
z	5	3	0

Table of node z

	x	y	z
x	0	1	5
y	1	0	3
z	5	3	0

y: $D_x(y) = c(x,y) + D_y(y) = 1 + 0 = 1$
$D_x(z) = c(x,y) + D_y(z) = 1 + 3 = \mathbf{4}$

z: $D_x(y) = c(x,z) + D_z(y) = 5 + 3 = 8$
$D_x(z) = c(x,z) + D_z(z) = 5 + 0 = 5$

x: $D_y(x) = c(y,x) + D_x(x) = 1 + 0 = 1$
$D_y(z) = c(y,x) + D_x(z) = 1 + 5 = 6$

z: $D_y(x) = c(y,z) + D_z(x) = 3 + 5 = 8$
$D_y(z) = c(y,z) + D_z(z) = 3 + 0 = 3$

x: $D_z(x) = c(z,x) + D_x(x) = 5 + 0 = 5$
$D_z(y) = c(z,x) + D_x(y) = 5 + 1 = 6$

y: $D_z(x) = c(z,y) + D_y(x) = 3 + 1 = \mathbf{4}$
$D_z(y) = c(z,y) + D_y(y) = 3 + 0 = 3$

Distance Vector

Each node updates its vector with the smallest distance

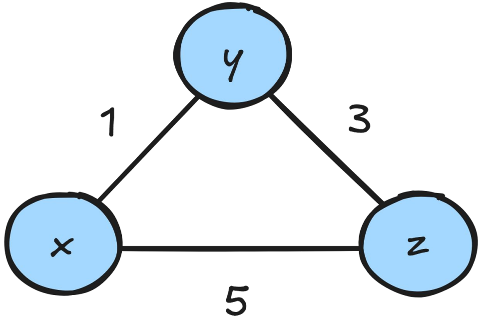


Table of node x

	x	y	z
x	0	1	**4**
y	1	0	3
z	5	3	0

Table of node y

	x	y	z
x	0	1	5
y	1	0	3
z	5	3	0

Table of node z

	x	y	z
x	0	1	5
y	1	0	3
z	**4**	3	0

y: $D_x(y) = c(x,y) + D_y(y) = 1 + 0 = 1$
$D_x(z) = c(x,y) + D_y(z) = 1 + 3 = \mathbf{4}$

z: $D_x(y) = c(x,z) + D_z(y) = 5 + 3 = 8$
$D_x(z) = c(x,z) + D_z(z) = 5 + 0 = 5$

x: $D_y(x) = c(y,x) + D_x(x) = 1 + 0 = 1$
$D_y(z) = c(y,x) + D_x(z) = 1 + 5 = 6$

z: $D_y(x) = c(y,z) + D_z(x) = 3 + 5 = 8$
$D_y(z) = c(y,z) + D_z(z) = 3 + 0 = 3$

x: $D_z(x) = c(z,x) + D_x(x) = 5 + 0 = 5$
$D_z(y) = c(z,x) + D_x(y) = 5 + 1 = 6$

y: $D_z(x) = c(z,y) + D_y(x) = 3 + 1 = \mathbf{4}$
$D_z(y) = c(z,y) + D_y(y) = 3 + 0 = 3$

Distance Vector

Each node advertises its updated table to the neighbors

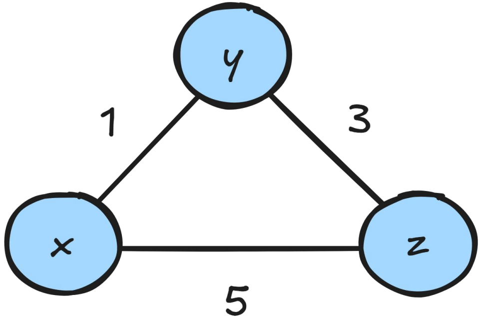


Table of node x

	x	y	z
x	0	1	4
y	1	0	3
z	**4**	**3**	**0**

Table of node y

	x	y	z
x	**0**	**1**	**4**
y	1	0	3
z	**4**	**3**	**0**

Table of node z

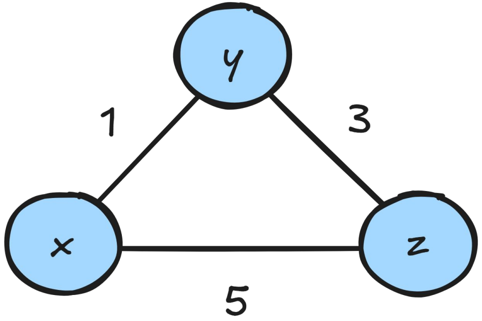
	x	y	z
x	**0**	**1**	**4**
y	1	0	3
z	4	3	0

x will advertise table to **y**, **z**

y's vector did not change so it does not advertise

z will advertise table to **x, y**

Distance Vector



Each node re-calculates its vector
There are no changes → the algorithm has **converged!**

Table of node x

	x	y	z
x	0	1	4
y	1	0	3
z	4	3	0

Table of node y

	x	y	z
x	0	1	4
y	1	0	3
z	4	3	0

Table of node z

	x	y	z
x	0	1	4
y	1	0	3
z	4	3	0

z: $D_x(y) = c(x,z) + D_z(y) = 4 + 3 = 7$
$D_x(z) = c(x,z) + D_z(z) = 4 + 0 = 4$

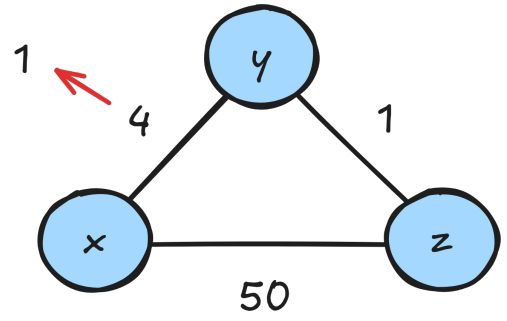
x: $D_y(x) = c(y,x) + D_x(x) = 1 + 0 = 1$
$D_y(z) = c(y,x) + D_x(z) = 1 + 4 = 5$

z: $D_y(x) = c(y,z) + D_z(x) = 3 + 4 = 7$
$D_y(z) = c(y,z) + D_z(z) = 3 + 0 = 3$

x: $D_z(x) = c(z,x) + D_x(x) = 4 + 0 = 4$
$D_z(y) = c(z,x) + D_x(y) = 4 + 1 = 5$

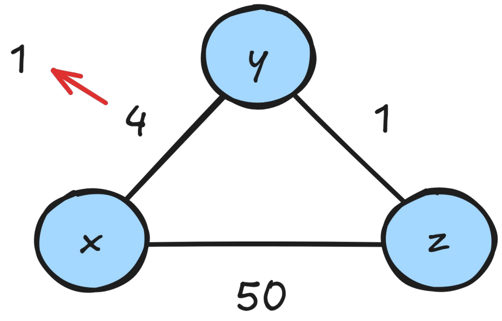
Distance Vector - Good News Travels Fast

- Link **(x-y)** changes cost: **4 → 1**
- $\mathbf{t_0}$: **y** detects the change
    - updates its vector
    - notifies its neighbors
- $\mathbf{t_1}$: **z** receives update from **y**
    - updates its vector for **z → x**: 5 → 2
    - notifies its neighbors
- $\mathbf{t_2}$: **y** receives update from **z**
    - updates its table
    - least costs have not changed → does not advertise
    - algorithm has converged



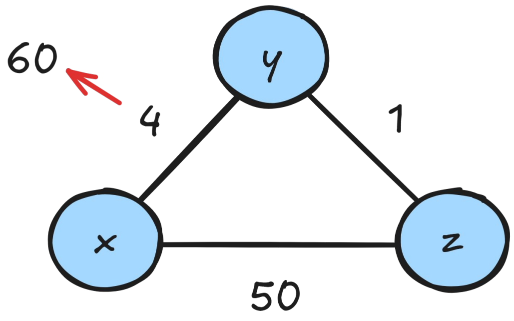
Distance Vector - Good News Travels Fast

- Link **(x-y)** changes cost: **4 → 1**
- **t_0**: **y** detects the change
  - updates its vector
  - notifies its neighbors
- **t_1**: **z** receives update from **y**
  - updates its vector for **z → x**: 5 → 2
  - notifies its neighbors
- **t_2**: **y** receives update from **z**
  - updates its table
  - least costs have not changed → does not advertise
  - algorithm has converged
- Only 2 iterations for the algorithm to converge → **Good news travels fast!** 🐇



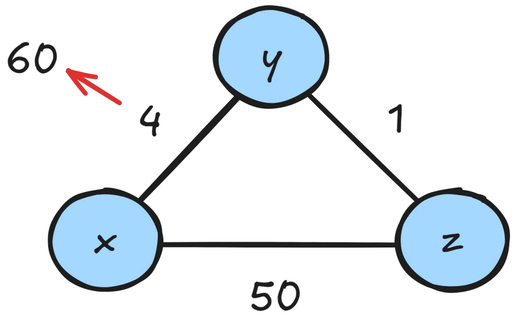
Distance Vector - Bad News Travels Slow

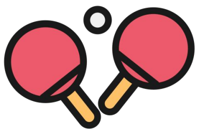
- Before change: $D_y(x) = 4$, $D_y(z) = 1$, $D_z(y) = 1$, $D_z(x) = 5$
- Link **(x-y)** changes cost: **4 → 60**
- **t_0: y** detects the change
  - Calculates new cost to **x**
  - $D_y(x) = \min\{ c(x, y) + D_x(x), c(y, z) + D_z(x) \} = \min\{ 60 + 0, 1 + 5 \} = 6$
  - By looking at the graph we can tell that this new cost is obviously **wrong**
  - But node y only knows what is in its table
  - So node **y**, in order to reach **x**, will route through **z**
    - expecting **z** to be able to reach **x** with only cost 5
  - **Routing loop**
    - node **y**, in order to reach **x**, will **route through z**
    - node **z**, in order to reach **x**, will **route through y**



Distance Vector - Bad News Travels Slow

- Before change: $D_y(x) = 4$, $D_y(z) = 1$, $D_z(y) = 1$, $D_z(x) = 5$
- Link **(x-y)** changes cost: **4 → 60**
- **t_0: y** detects the change
    - Calculates new cost to **x**
    - $D_y(x) = \min\{ c(x, y) + D_x(x), c(y, z) + D_z(x) \} = \min\{ 60 + 0, 1 + 5 \} = 6$
    - By looking at the graph we can tell that this new cost is obviously **wrong**
    - But node y only knows what is in its table
    - So node **y**, in order to reach **x**, will route through **z**
        - expecting **z** to be able to reach **x** with only cost 5
    - **Routing loop**
        - node **y**, in order to reach **x**, will **route through z**
        - node **z**, in order to reach **x**, will **route through y**
        - ping-pong situation

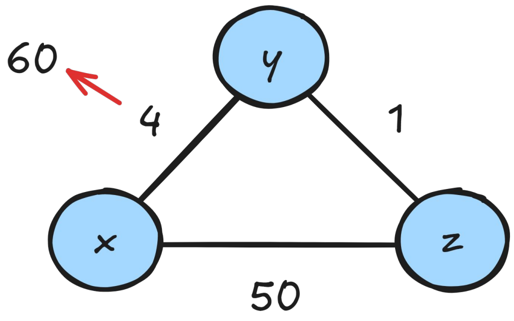




Distance Vector - Bad News Travels Slow

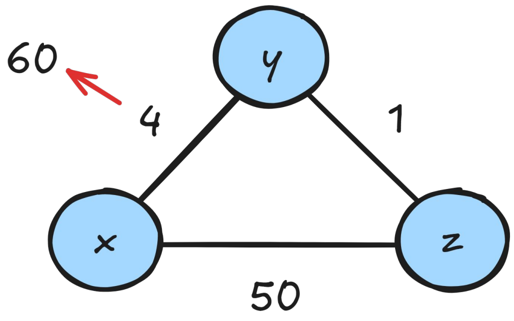
- t_1: **y** calculates new cost to **x**
  - Notifies **z**

  z receives update: $D_y(x) = 6$
  - Calculates new cost to **x**
  - $D_z(x) = \min\{ 50 + 0, 1 + 6 \} = 7$
  - Updates its vector
  - Notifies **y**
- **y** receives update, calculates new $D_y(x) = 8$
- **z** receives update, calculates new $D_y(x) = 9$
- and so on…



Distance Vector - Bad News Travels Slow

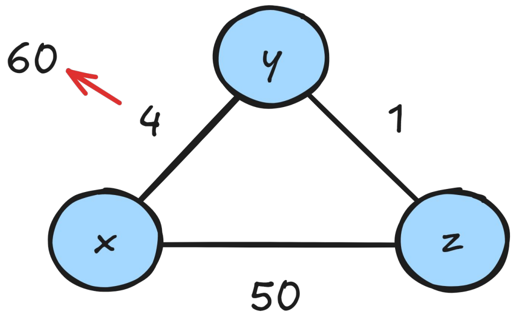
- How long will this go on?
  - 44 iterations
  - Until **z** calculates that the cost through **y** is higher than 50
  - **z** will set the path to **x** through the direct link with **x**
  - **y** will set the path to **x** through **z**
- **Count-to-Infinity Problem**
  - Because of the many iterations



- 44 iterations for the algorithm to converge → **Bad news travels slow!** 🐢

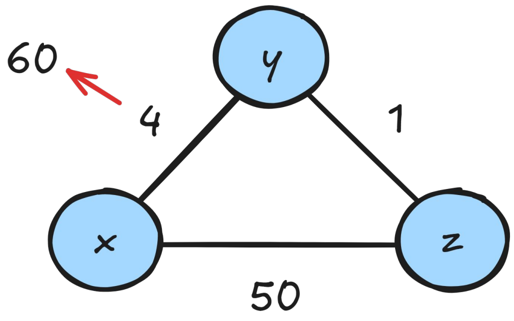
Distance Vector - Poisoned Reverse

- The previous scenario can be avoided
- Using a technique called **poisoned reverse**
  - If **z** routes through **y** to reach **z**
  - Then **z** will tell **y** that its distance from **x** is infinite: $\mathbf{D_z(x) = \infty}$
  - **y** now believes that **z** does not have a route to **x**
  - So **y** will not try to route **though z** to reach **x**
- When the change from 4 to 60 happens:
  - **y** updates its table
  - Keeps routing to **x** via the **direct link**
  - Informs **z** about its new cost to x: $\mathbf{D_y(x) = 60}$



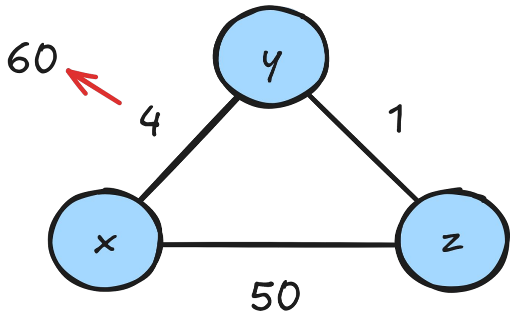
Distance Vector - Poisoned Reverse

- **z** receives update from **y**
  - changes its vector the cost of the direct link: $\mathbf{D_z(x) = 50}$
  - notifies **y**
- **y** receives update from **z**
  - changes its vector: $\mathbf{D_y(x) = 51}$
  - now **y** is the one doing the poisoning
  - tells **z** that $\mathbf{D_y(x) = \infty}$



Distance Vector - Poisoned Reverse

- **z** receives update from **y**
  - changes its vector the cost of the direct link: $\mathbf{D_z(x) = 50}$
  - notifies **y**
- **y** receives update from **z**
  - changes its vector: $\mathbf{D_y(x) = 51}$
  - now **y** is the one doing the poisoning
  - tells **z** that $\mathbf{D_y(x) = \infty}$



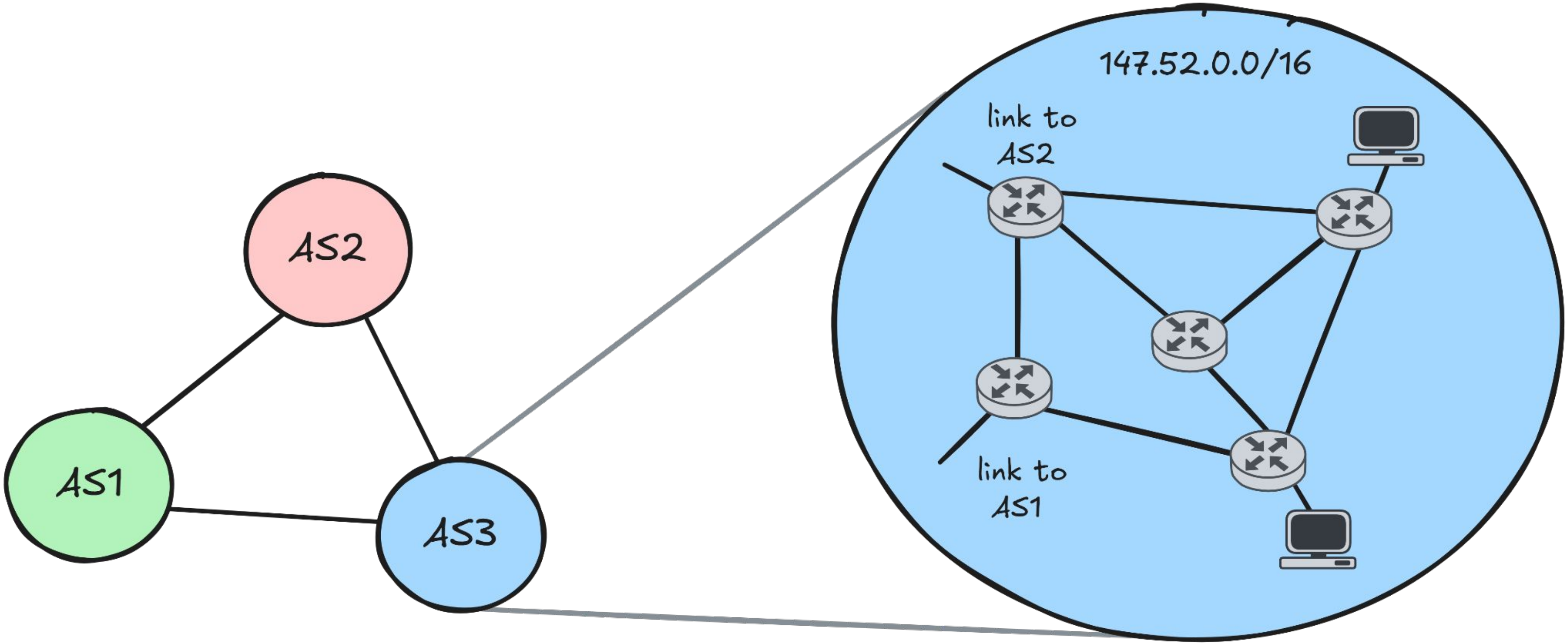
Does poisoned reverse solve the count-to-infinity problem?

No, if the loops included 3 or more nodes, they would not be detected.

Autonomous System

- A **group of routers** which operate under the **same management**
- Each AS has a unique number identifier called **Autonomous System Number (ASN)** (ex. 6867)
- The routers of each AS share a **common prefix** (ex. 147.52.0.0/16)
- Each organization (Facebook, Google, Amazon etc.) has one or more ASes in different locations

Autonomous System



AS Routing Protocols

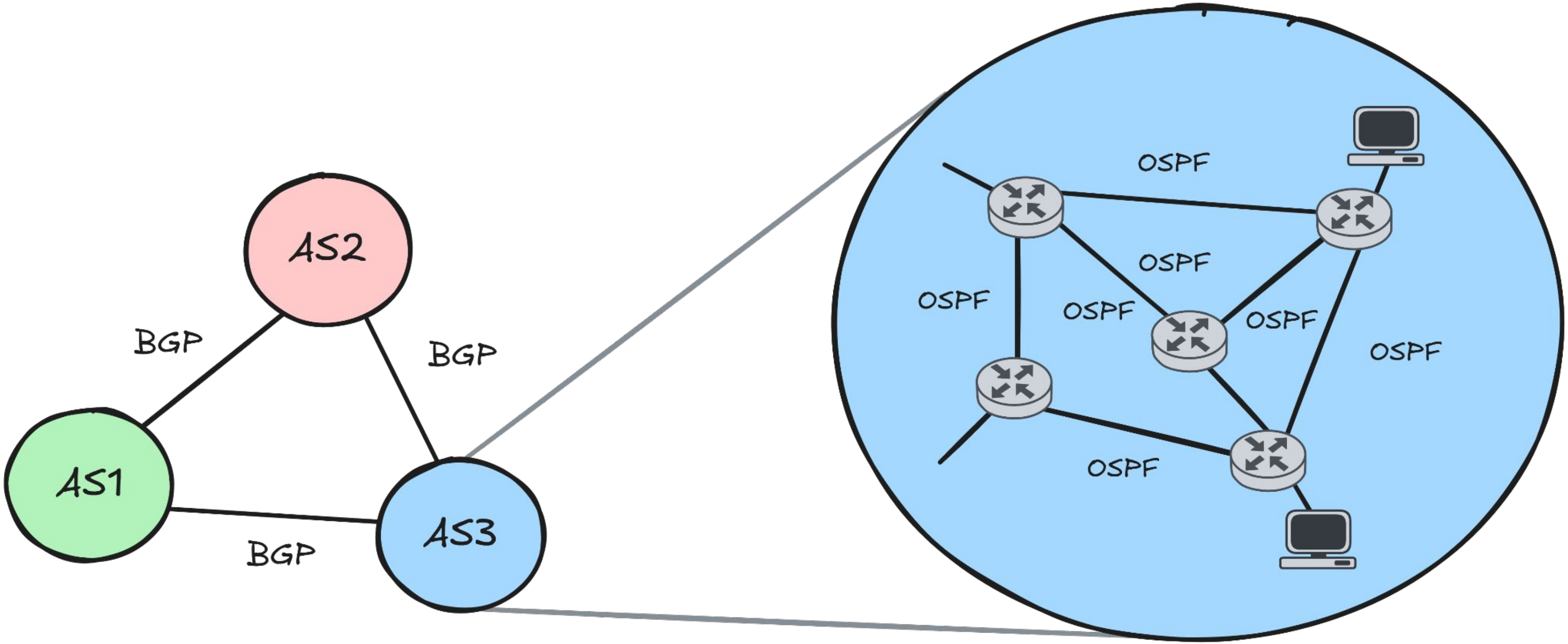
Intra-AS routing

- Determines how the traffic flows **inside the AS**
- Example: OSPF, IGRP, RIP, IS-IS

Inter-AS routing

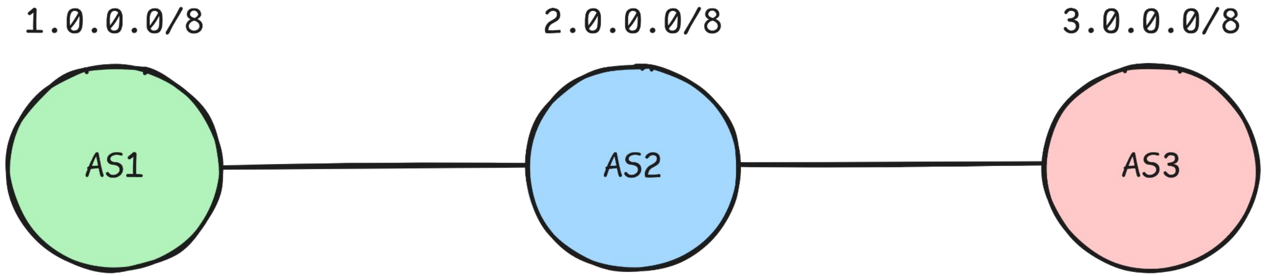
- Determines how the traffic flows **among different ASes**
- Example: BGP
  - **This is what the actual Internet uses**

AS Routing Protocols



Border Gateway Protocol (BGP)

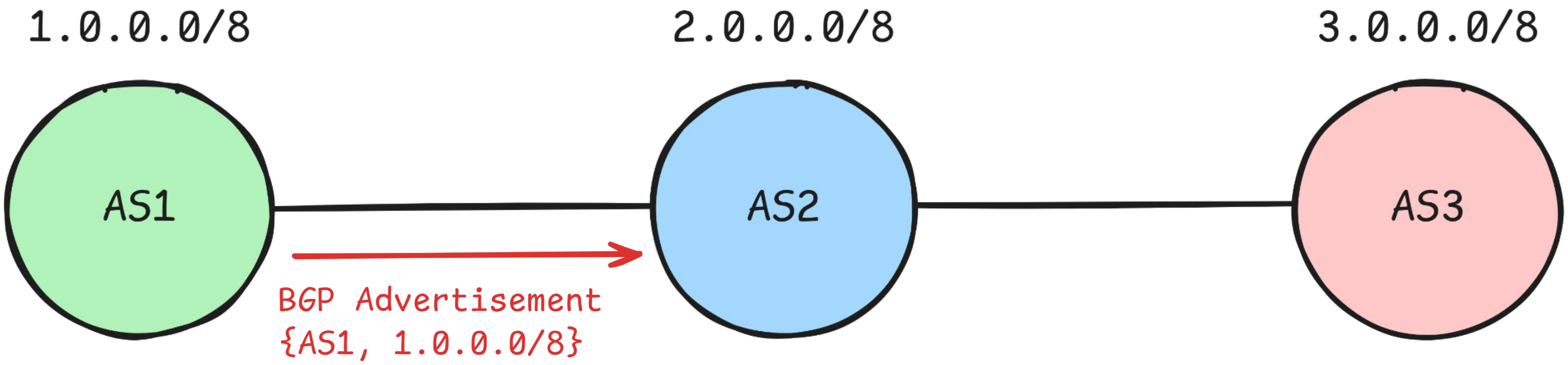
- Purpose: routing between ASes
- An AS will advertise its prefix so that it is known to the rest of the Internet
- It does not calculate routes towards a specific IP
- But to a **prefix** that belongs to an AS
- Uses **Path Vector algorithm**
  - A modified version of **distance vector**
  - Uses **paths (sequence of ASes)** instead of distances



Border Gateway Protocol (BGP)

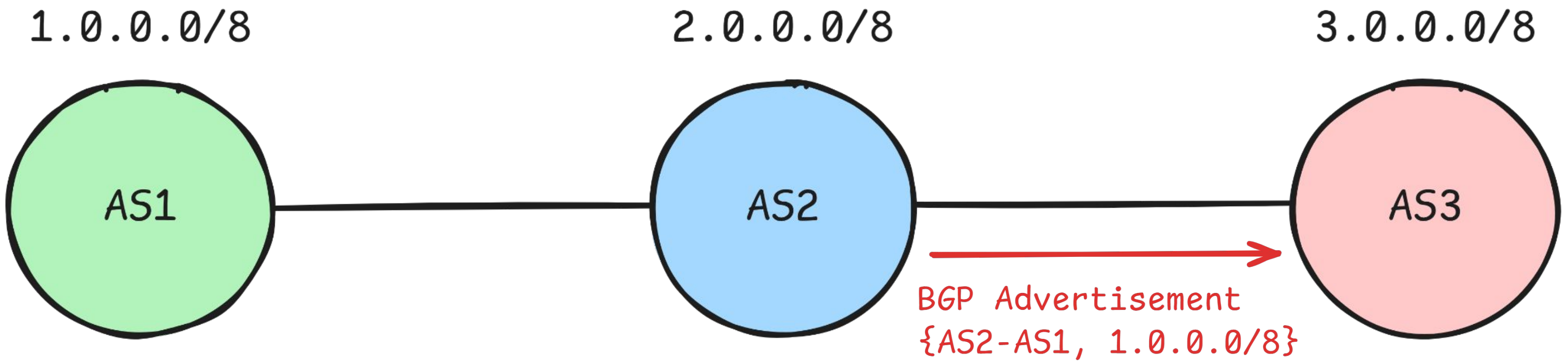
Let's see a simplified example:

- **AS1** will send a **BGP advertisement** to its neighbor AS2
- Includes: **ASN** and its **prefix**



Border Gateway Protocol (BGP)

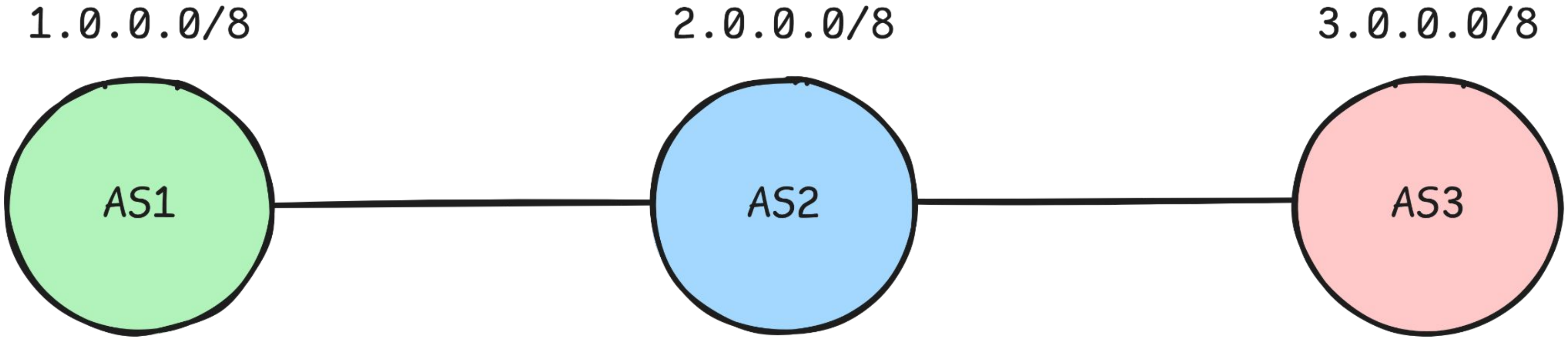
- **AS2** will add its **ASN** to the AS-PATH
- And send a **BGP advertisement** to its neighbor AS3



Border Gateway Protocol (BGP)

Now AS2 and AS3 know that:

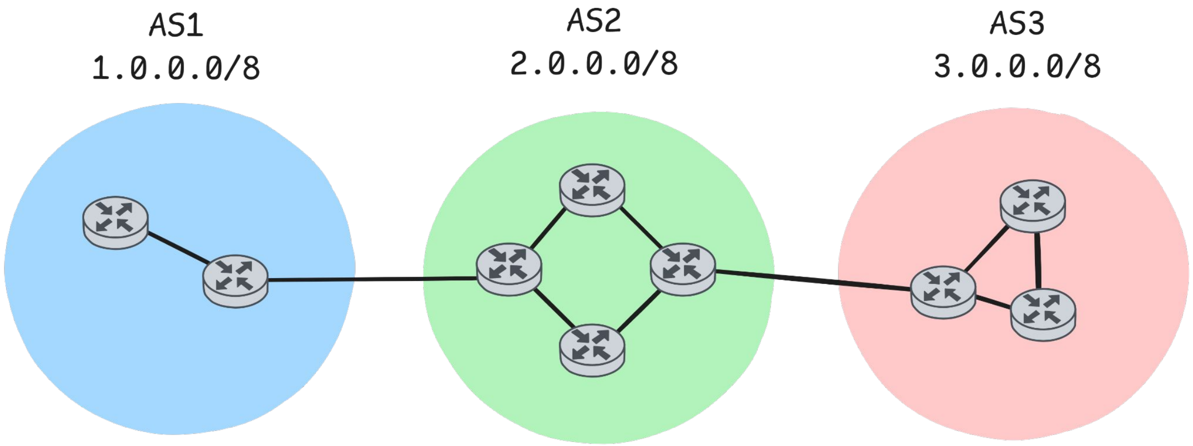
- to **reach any destination IP** that belongs to **prefix 1.0.0.0/8**
- must **send to AS1**



Border Gateway Protocol (BGP)

Let's take a closer look at BGP:

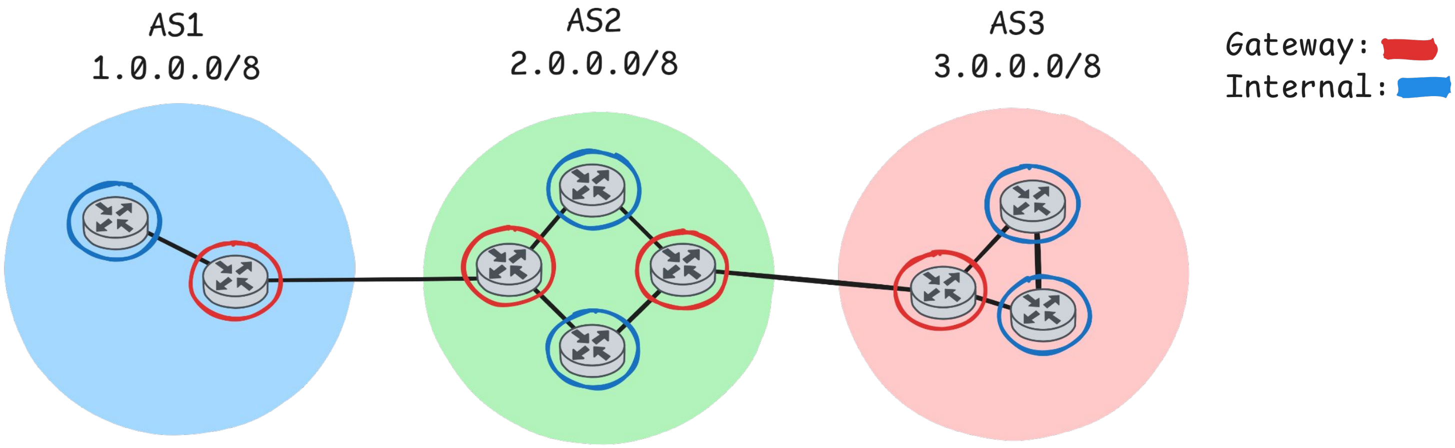
- **Gateway router:** router that connects to *another AS*
- **Internal router:** router *inside an AS*, connects only to routers/hosts inside that AS



BGP - Routers

Let's take a closer look at BGP:

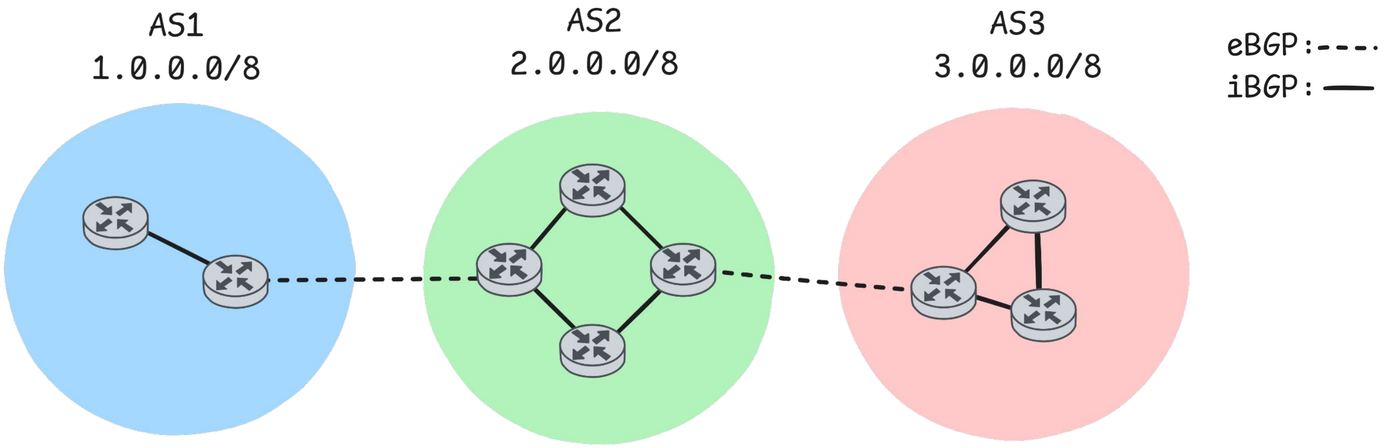
- **Gateway router:** router that connects to *another AS*
- **Internal router:** router *inside an AS*, connects only to routers/hosts inside that AS



BGP - eBGP & iBGP

BGP is separated to two protocols:

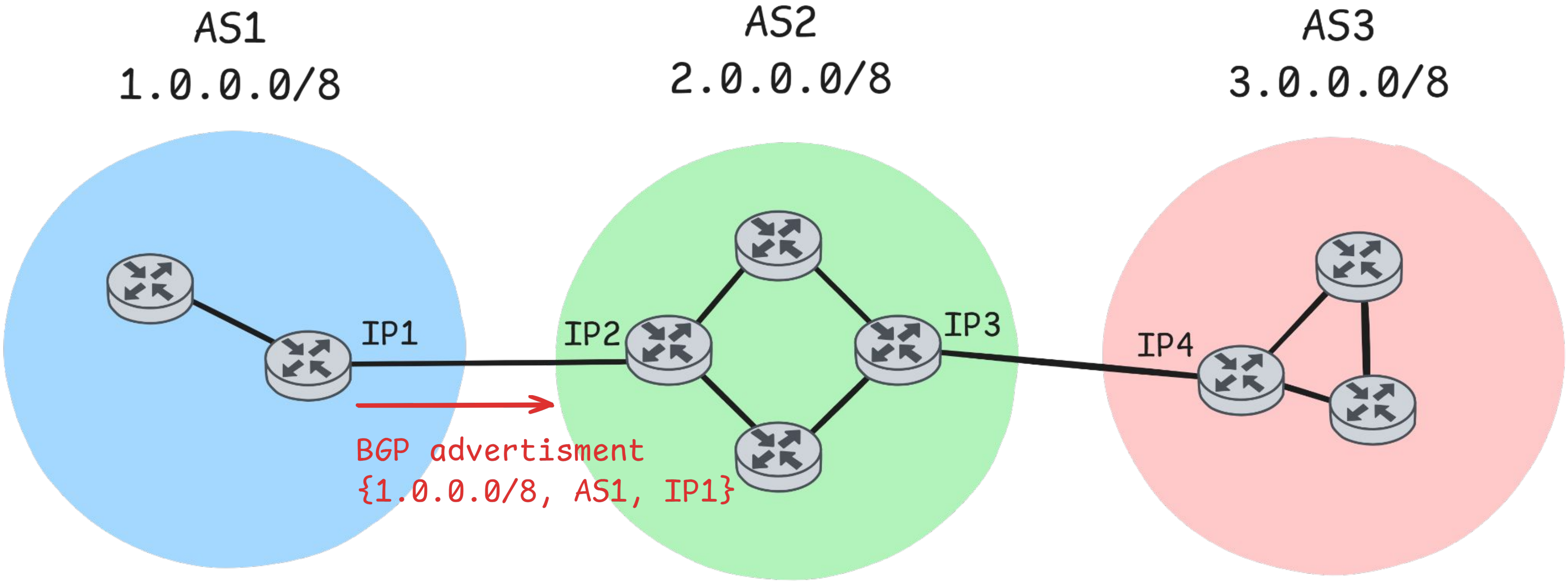
- **eBGP**: exchange reachability information from neighboring ASes
- **iBGP**: propagate this information to all AS-internal routers
  - Internal routers run iBGP,
  - Gateway routers run **both** eBGP and iBGP



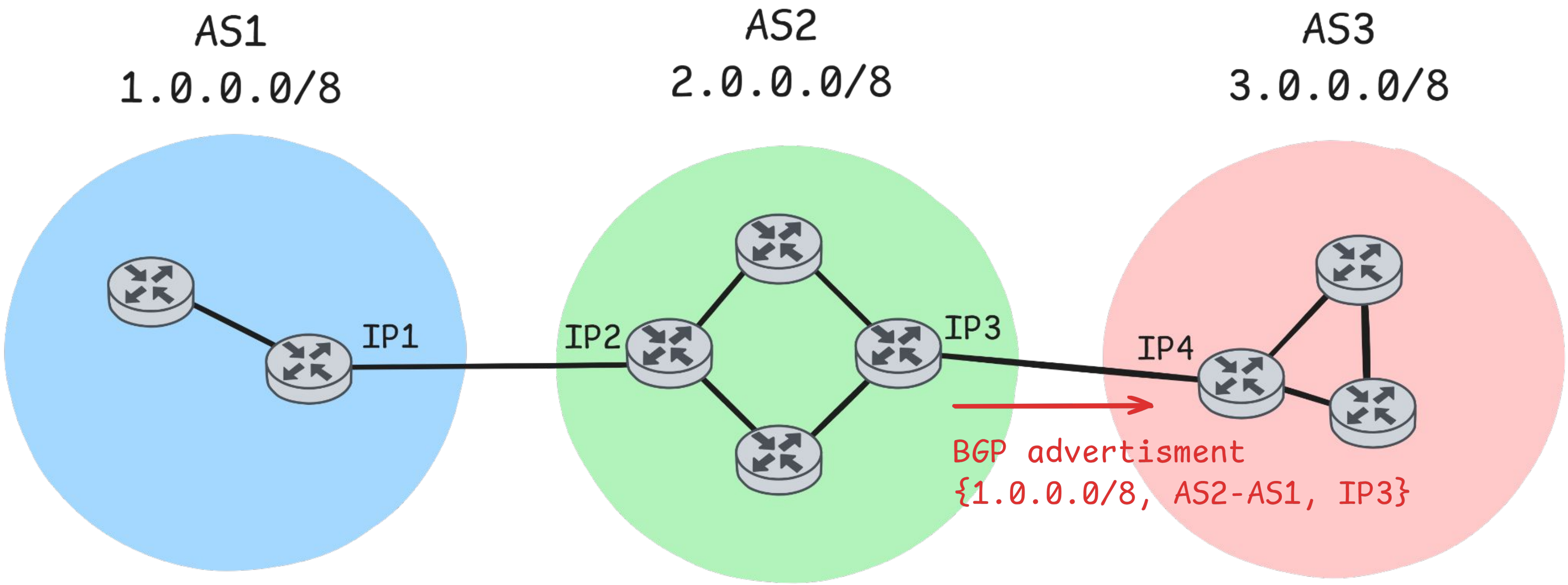
BGP - Attributes

- Beside prefix, BGP advertisements include **attributes**
- *Prefix + Attributes = BGP Route*
- Some important attributes:
  - **AS-PATH**: Shows the ASes from which the advertisement has passed through (the sequence of ASNs we saw in previous examples)
  - **NEXT-HOP**: Shows the IP of the next hop router interface, that the AS-PATH starts from

Border Gateway Protocol (BGP)

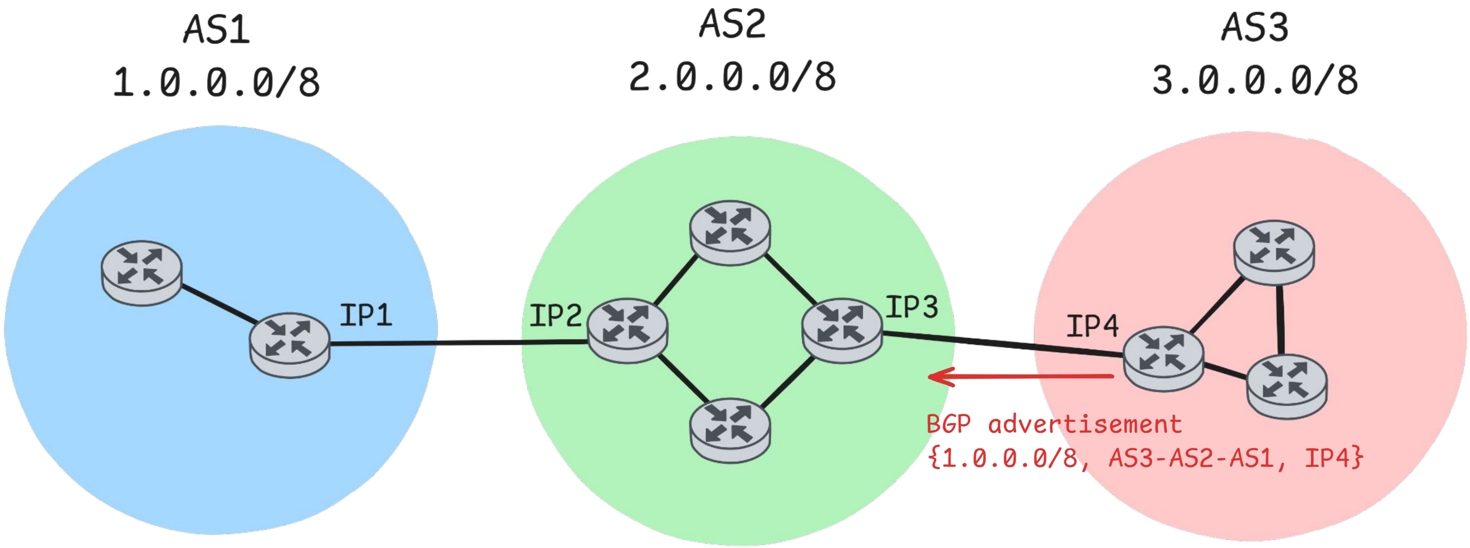


Border Gateway Protocol (BGP)



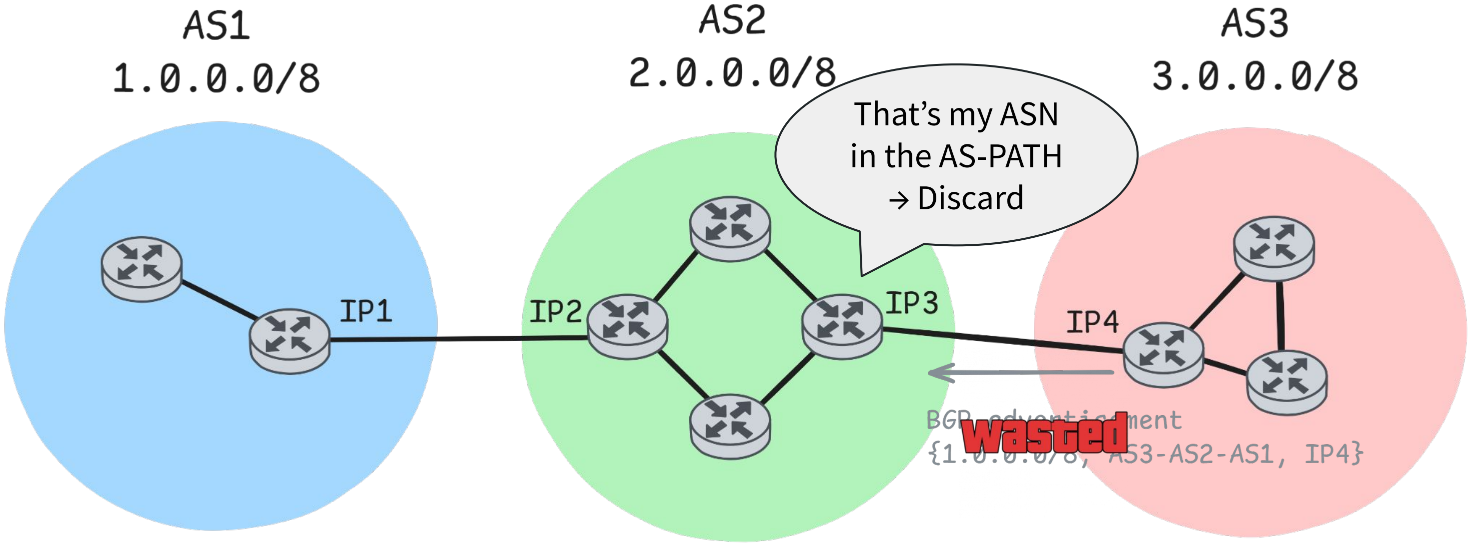
BGP - Loop Prevention

- AS3 will advertise the route back to AS2, since AS2 is a neighbor
- That could lead to **advertisement loops**
    - AS3 sends to AS2, AS2 sends to AS1 and AS2 again and so on…



BGP - Loop Prevention

- To prevent loops, if an AS sees **its own ASN** in the **AS-PATH**
- It **discards** the advertisement



BGP - Path Selection

How are paths chosen?

- By default BGP chooses the **shortest path**
  - Path that passes through the least ASes
- **Local Preference:** a BGP attribute
  - An integer value
  - Used to apply **policies** (i.e. prefer to send traffic through this AS, avoid this AS etc.)
  - Paths with **higher** local preference **are prefered**

Sum Up

Routing Algorithms

Link-State Routing Algorithms (Centralized)

Dijkstra's Algorithm

A link-state algorithm

OSPF

Protocol that uses **Dijkstra's** Algorithm
Intra-AS routing

Decentralized Routing Algorithms

Distance Vector Algorithm

A decentralized algorithm
Uses the **Bellman-Ford** equation

BGP

Protocol that uses **Path Vector** algorithm
(a modification of Distance Vector)
Inter-AS routing

Questions???