Part 1: Overview

#1 What is a network made of?
#2 How is it shared?
#3 How is it organized?
#4 How does communication happen?
#5 How do we characterize it?

Part 2: Concepts

How do you guide IP packets from a source to destination?
How do you ensure reliable transport on top of best-effort delivery?
How do you guide IP packets from a source to destination?

Think of IP packets as envelopes

Like an envelope, packets have a header
Like an envelope, packets have a payload.

The payload contains the data to be delivered.

The header contains the metadata needed to forward the packet.

Routers forward IP packets hop-by-hop towards their destination.
Let’s zoom in on what is going on between two adjacent routers

Upon packet reception, routers locally look up their forwarding table to know where to send it next

Packet
Forwarding table

Here, the packet should be directed to IF#4

Forwarding is repeated at each router, until the destination is reached
Forwarding decisions necessarily depend on the destination, but can also depend on other criteria:

- **destination addr**: mandatory (why?)
- **source addr**: (requires n² state)
- **input port**
- + any other header

With source- & destination-based routing, paths from different sources can differ:

- With destination-based routing, paths from different source coincide once they overlap:
  - Set of paths to the destination produce a spanning tree rooted at the destination:
    - cover every router exactly once
    - only one outgoing arrow at each router

Once path to destination meet, they will never split:
Here is an example of a spanning tree for destination X:

In the rest of the lecture, we’ll consider destination-based routing the default in the Internet.

Where are these forwarding tables coming from?

In addition to a data-plane, routers are also equipped with a control-plane:

Think of the control-plane as the router’s brain:

Routing is the control-plane process that computes and populates the forwarding tables:

While forwarding is a local process, routing is inherently a global process:

How can a router know where to direct packets if it does not know what the network looks like?
### Forwarding vs Routing

<table>
<thead>
<tr>
<th></th>
<th>Forwarding</th>
<th>Routing</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Goal</strong></td>
<td>Directing packet to an outgoing link</td>
<td>Computing the paths packets will follow</td>
</tr>
<tr>
<td><strong>Scope</strong></td>
<td>Local</td>
<td>Network-wide</td>
</tr>
<tr>
<td><strong>Implementation</strong></td>
<td>Hardware</td>
<td>Software</td>
</tr>
<tr>
<td><strong>Timescale</strong></td>
<td>Nanoseconds</td>
<td>Milliseconds (hopefully)</td>
</tr>
</tbody>
</table>

### The Goal of Routing

The goal of routing is to compute valid global forwarding state.

**Definition**

A global forwarding state is valid if and only if:
- There are no dead ends
  - No outgoing port defined in the table
- There are no loops
  - Packets going around the same set of nodes

### A Global Forwarding State is Valid

A global forwarding state is valid if and only if there are no forwarding loops.

**Theorem**

A global forwarding state is valid if and only if:
- There are no dead ends
- No outgoing port defined in the table
- There are no loops
- Packets going around the same set of nodes

### Verifying that a Forwarding State is Valid

**Question 1** How do we verify that a forwarding state is valid?

**Question 2** How do we compute valid forwarding state?

**Simple Algorithm for One Destination**

- Mark all outgoing ports with an arrow
- Eliminate all links with no arrow
- State is valid if the remaining graph is a spanning tree
Given a graph with the corresponding forwarding state

Mark all outgoing ports with an arrow

Eliminate all links with no arrow

The result is a spanning tree. This is a valid forwarding state

Mark all outgoing ports with an arrow

Eliminate all links with no arrow

The result is not a spanning-tree. The forwarding state is not valid
How do we verify that a forwarding state is valid?
How do we compute valid forwarding state?

Producing valid forwarding state is harder, but doable

Existing routing protocols differ in how they avoid loops

Essentially, there are three ways to compute valid forwarding state

<table>
<thead>
<tr>
<th>Intuition</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>#1</td>
<td>Use tree-like topologies</td>
</tr>
<tr>
<td>#2</td>
<td>Rely on a global network view</td>
</tr>
<tr>
<td>#3</td>
<td>Rely on distributed computation</td>
</tr>
</tbody>
</table>

The easiest way to avoid loops is to forward traffic on a loop-free topology

- **Simple algorithm**
  - Take an arbitrary topology
  - Build a spanning tree and ignore all other links
  - Done!
- **Why does it work?**
  - Spanning trees have only one path between any two nodes

In practice, there can be **many** spanning-trees for a given topology
Once we have a spanning tree, forwarding on it is easy.

When a packet arrives, simply send it on all ports.

While flooding works, it is quite wasteful.

Useless transmissions.

The issue is that nodes do not know their respective locations.

Nodes can learn how to reach nodes by remembering where packets came from.
Node A can be reached through this port

All the green nodes learn how to reach A

B answers back to A enabling the green nodes to also learn where B is

There is no need for flooding here as the position of A is already known by everybody

Learning is topology-dependent The blue nodes only know how to reach A (not B)
Forwarding by flooding on a spanning-tree in a nutshell

Flood first packet to node you’re trying to reach. All switches learn where you are.

When destination answers, some switches learn where it is some because packet to you is not flooded anymore.

The decision to flood or not is done on each switch depending on who has communicated before.

Spanning-Tree in practice used in Ethernet

**Advantages**
- Plug-and-play
- Configuration-free
- Automatically adapts to moving host

**Disadvantages**
- Slow to react to failures
- Mandate a spanning-tree
- Eliminate many links from the topology

Essentially, there are three ways to compute valid forwarding state

Use tree-like topologies
- Spanning tree

Rely on a global network view
- Link-State
- SDN

Rely on distributed computation
- Distance Vector
- BGP

Once a node \( u \) knows the entire topology, it can compute shortest-paths using Dijkstra’s algorithm

**Initialization**

\[ S = \{u\} \]

For all nodes \( v \):
- If \( v \) is adjacent to \( u \):
  \[ D(v) = c(u,v) \]
- Else:
  \[ D(v) = \infty \]

**Loop**

While not all nodes in \( S \):
- Add \( w \) with the smallest \( D(w) \) to \( S \)
- Update \( D(v) \) for all adjacent \( v \) not in \( S \):
  \[ D(v) = \min(D(v), D(w) + c(w,v)) \]

Dijkstra maintains two data structures:
- \( S \) the set of vertices whose shortest path is known
- \( D(v) \) the current estimate of the shortest path cost towards vertex \( v \)

The initialization phase defines the original data structures content

\[ S = \{u\} \]

For all nodes \( v \):
- If \( v \) is adjacent to \( u \):
  \[ D(v) = c(u,v) \] \( c(u,v) \) is the weight of the link connecting \( u \) and \( v \)
- Else:
  \[ D(v) = \infty \]

\( D(v) \) is the smallest distance currently known by \( u \) to reach \( v \)

Each iteration Dijkstra adds 1 node to \( S \) (the closest one) before updating the distances to reach the others nodes

**Loop**

While not all nodes in \( S \):
- Add \( w \) with the smallest \( D(w) \) to \( S \)
- Update \( D(v) \) for all adjacent \( v \) not in \( S \):
  \[ D(v) = \min(D(v), D(w) + c(w,v)) \]
Let's compute the shortest-paths from \( u \)

**Initialization**

\[ S = \{ u \} \]

for all nodes \( v \):

- if \( v \) is adjacent to \( u \):
  \[ D(v) = c(u,v) \]
- else:
  \[ D(v) = \infty \]

**Loop**

add \( w \) with the smallest \( D(w) \) to \( S \)

update \( D(v) \) for all adjacent \( v \) not in \( S \):

\[ D(v) = \min\{D(v), D(w) + c(w,v)\} \]

while not all nodes in \( S \):

add \( E \) to \( S \)

\[ S = \{ u, E \} \]

Now, do it by yourself

**Initialization**

\[ S = \{ u \} \]

**Loop**

add \( E \) to \( S \)

\[ S = \{ u, E \} \]
Here is the final state

This algorithm has a $O(n^2)$ complexity where $n$ is the number of nodes in the graph

Iteration #1: search for minimum through $n$ nodes
Iteration #2: search for minimum through $n-1$ nodes
Iteration $n$: search for minimum through 1 node
$n(n+1)$ operations => $O(n^2)$

Better implementations rely on a heap to find the next node to expand, bringing down the complexity to $O(n \log n)$

This algorithm has a $O(n^2)$ complexity where $n$ is the number of nodes in the graph

From the shortest-paths, $u$ can directly compute its forwarding table

To build this global view, routers essentially solve a jigsaw puzzle

Initially, routers only know their ID and their neighbors

Each routers builds a message (known as Link-State) and floods it (reliably) in the entire network

At the end of the flooding process, everybody share the exact same view of the network required for correctness see exercise
Dijkstra will always converge to a unique stable state when run on static weights

\[ \text{cf. exercise session for the dynamic case} \]

Essentially, there are three ways to compute valid forwarding state:

- Use tree-like topologies
- Rely on a global network view
- Rely on distributed computation

Spanning-tree
Link-State
SDN
Distance-Vector
BGP

Instead of locally compute paths based on the graph, paths can be computed in a distributed fashion

Let \( d(x(y)) \) be the cost of the least-cost path known by \( x \) to reach \( y \) and each node bundles these distances into one message (called a vector) that it repeatedly sends to all its neighbors until convergence.

Each node updates its distances based on neighbors’ vectors:

\[ d_x(y) = \min\{ c(x,v) + d_v(y) \} \]

over all neighbors \( v \).

Let’s compute the shortest-path from \( u \) to \( D \)

The values computed by a node \( u \) depends on what it learns from its neighbors (A and E)

\[ d_u(D) = \min\{ c(u,A) + d_A(D), c(u,E) + d_E(D) \} \]

over all neighbors \( v \).
To unfold the recursion, let’s start with the direct neighbor of D

\[ d_B(D) = 1 \]
\[ d_C(D) = 4 \]

B and C announce their vector to their neighbors, enabling A to compute its shortest-path

\[ d_A(D) = \min \{ 2 + d_B(D), 1 + d_C(D) \} \]
\[ = 3 \]

As soon as a distance vector changes, each node propagates it to its neighbor

\[ d_E(D) = \min \{ 1 + d_C(D), 4 + d_G(D), 2 + d_u(D) \} \]
\[ = 5 \]

Eventually, the process converges to the shortest-path distance to each destination

\[ d_u(D) = \min \{ 3 + d_A(D), 2 + d_E(D) \} \]
\[ = 6 \]

As before, u can directly infer its forwarding table by directing the traffic to the best neighbor

the one which advertised the smallest cost

Evaluating the complexity of DV is harder, we’ll get back to that in a couple of weeks