Communication Networks
Part 2: Concepts

How do you guide IP packets from a source to destination?

Like an envelope, packets have a header

Packet

Think of IP packets as envelopes

Like an envelope, packets have a payload
The header contains the metadata needed to forward the packet.

Identify the source and destination of the communication.

The payload contains the data to be delivered.

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.

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**: mandatory (why?)
- **source**: requires m^2 state
- **input port**: traffic engineering
- +any other header

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

Let's compare these two:

With **destination-based routing**, paths from different source coincide once they overlap:

Once path to destination meet, they will **never** split

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

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.

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

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

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

Forwarding vs Routing summary:

<table>
<thead>
<tr>
<th></th>
<th>Forwarding</th>
<th>Routing</th>
</tr>
</thead>
<tbody>
<tr>
<td>goal</td>
<td>directing packet to an outgoing link</td>
<td>computing the paths packets will follow</td>
</tr>
<tr>
<td>scope</td>
<td>local</td>
<td>network-wide</td>
</tr>
<tr>
<td>implem.</td>
<td>hardware</td>
<td>software</td>
</tr>
<tr>
<td>timescale</td>
<td>nanoseconds</td>
<td>milliseconds (hopefully)</td>
</tr>
</tbody>
</table>

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

Roles:
- Routing
- Configuration
- Statistics
- …
The goal of routing is to compute valid global forwarding state.

**Definition**
A global forwarding state is valid if it always delivers packets to the correct destination.

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

**Theorem**

**sufficient and necessary condition**

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 routing state is valid is easy**

simple algorithm for one destination

Mark all outgoing ports with an arrow
Eliminate all links with no arrow
State is valid iff 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
- State is valid iff the remaining graph is a spanning tree

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

**question 2**
How do we verify that a forwarding state is valid?
How do we compute valid 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 routing state

Mark all outgoing ports with an arrow

Eliminate all links with no arrow

The result is not a spanning-tree.
The routing state is not valid

How do we verify that a forwarding state is valid?
How do we compute valid forwarding state?
Producing valid routing state is harder, but doable.

Existing routing protocols differ in how they avoid loops.

- Prevent dead ends: easy
- Prevent loops: hard

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

1. Use tree-like topologies
2. Rely on a global network view
3. Rely on distributed computation

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

- Intuition: Spanning tree
- Example: BGP

In practice, there can be many spanning-trees for a given topology.

- Spanning Tree #1

Simple algorithm:

1. Take an arbitrary topology.
2. Build a spanning tree and ignore all other links.
3. Why does it work? Spanning trees have only one path between any two nodes.
Once we have a spanning tree, forwarding on it is easy. Literally just flood the packets everywhere.

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.

intuition

if
flood packet from node A entered switch X on port 4
then
switch X can use port 4 to reach node A.
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).

Routing 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

<table>
<thead>
<tr>
<th>advantages</th>
<th>disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>plug-and-play</td>
<td>mandate a spanning-tree</td>
</tr>
<tr>
<td>configuration-free</td>
<td>eliminate many links from the topology</td>
</tr>
<tr>
<td>automatically adapts to moving host</td>
<td>slow to react to failures</td>
</tr>
<tr>
<td></td>
<td>host movement</td>
</tr>
</tbody>
</table>

If each router knows the entire graph, it can locally compute paths to all other nodes.

Dijkstra maintains two data structures: $S$ and $D$

- $S$: the set of vertices whose shortest path is known
- $D(v)$: the current estimate of the shortest path cost towards vertex $v$

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

Loop

```plaintext
while not all nodes in $S$:
  add $w$ with the smallest $D(w)$ to $S$
  for all adjacent $v$ not in $S$:
    $D(v) = \min(D(v), D(w) + c(w,v))$
```

Let’s compute the shortest-paths from $u$

Essentially, there are three ways to compute valid routing state

<table>
<thead>
<tr>
<th>Use tree-like topologies</th>
<th>Spanning-tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rely on a global network view</td>
<td>Link-State</td>
</tr>
<tr>
<td>Rely on distributed computation</td>
<td>SDN</td>
</tr>
<tr>
<td></td>
<td>Distance-Vector</td>
</tr>
<tr>
<td></td>
<td>BGP</td>
</tr>
</tbody>
</table>

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

Initialization

```plaintext
S = {u}
for all nodes $v$: $D(v) = \infty$
if ($v$ is adjacent to $u$):
  $D(v) = c(u,v)$
else:
  $D(v) = \infty$
```

Loop

```plaintext
while not all nodes in $S$:
  add $w$ with the smallest $D(w)$ to $S$
  for all adjacent $v$ not in $S$:
    $D(v) = \min(D(v), D(w) + c(w,v))$
```

The initialization phase defines the original data structures content

- $u$ is the node running the algorithm
- $c(u,v)$ is the weight of the link connecting $u$ and $v$
- $D(w)$ is the smallest distance currently known by $u$ to reach $v$

Let’s compute the shortest-paths from $u$

```
1  2  1
A B
C D
E F
G
```
**Initialization**

\[ S = \{ u \} \]

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

S only contains \( u \) itself and 
D is initialized based on \( u \)'s weight

**Loop**

\[ \text{while not all nodes in } S: \]
- add \( w \) with the smallest \( D(w) \) to \( S \)
- update \( D(v) \) for all adjacent \( v \) not in \( S \):
  \[ D(v) = \min\{D(v), D(w) + c(w,v)\} \]

Now, do it by yourself

Here is the final state

\[ S = \{ u, A, B, C, D, E, F, G \} \]
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

$\frac{n(n+1)}{2}$ operations $\Rightarrow O(n^2)$

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

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

<table>
<thead>
<tr>
<th>Destination</th>
<th>Next-hop</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>B</td>
<td>A</td>
</tr>
<tr>
<td>C</td>
<td>E</td>
</tr>
<tr>
<td>D</td>
<td>A</td>
</tr>
<tr>
<td>E</td>
<td>E</td>
</tr>
<tr>
<td>F</td>
<td>E</td>
</tr>
<tr>
<td>G</td>
<td>E</td>
</tr>
</tbody>
</table>

Initially, routers only know their ID and their neighbors:

- D only knows, it is connected to B and C along with the weights to reach them (by configuration).

At the end of the flooding process, everybody share the exact same view of the network required for correctness.

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

Dijkstra will always converge to a unique stable state when run on static weights.

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

At the end of the flooding process, everybody share the exact same view of the network required for correctness see exercise.
Essentially, there are three ways to compute valid routing state:

- Use tree-like topologies (Spanning tree)
- Rely on a global network view (Link-State, SDN)
- Rely on distributed computation (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 $y$.

Each node bundles these distances into one message (called a vector) that it repeatedly sends to all its neighbors until convergence.

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):

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

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

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

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

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

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

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

As before, u can directly infer its forwarding table by directing the traffic to the best neighbor, the one which advertised the smallest cost

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

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

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

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