Communication Networks

Prof. Laurent Vanbever

Communication Networks
Spring 2021

Laurent Vanbever
nsg.ee.ethz.ch

ETH Zürich (D-ITET)
8 March 2021

Materials inspired from Scott Shenker & Jennifer Rexford

Last week on
Communication Networks

Communication Networks
Part 1: General overview

What is a network made of?

How is it shared?

How is it organized?

#4 How does communication happen?

How do we characterize it?

The Internet should allow

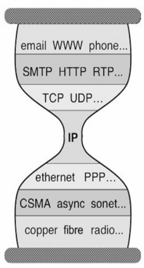
processes on different hosts
to exchange data

everything else is just commentary…

In practice, there exists **a lot** of network protocols.
How does the Internet organize **this?**

Each layer provides a service to the layer above
by using the services of the layer directly below it

Applications
…built on…
Reliable (or unreliable) transport
…built on…
Best-effort global packet delivery
…built on…
Best-effort local packet delivery
…built on…
Physical transfer of bits



Communication Networks
Part 1: General overview

What is a network made of?

How is it shared?

How is it organized?

How does communication happen?

#5 How do we characterize it?

A network *connection* is characterized by its delay, loss rate and throughput

delay

loss

throughput

How long does it take for a packet to reach the destination

What fraction of packets sent to a destination are dropped?

At what rate is the destination receiving data from the source?

This week on
Communication Networks

We will start diving in the fundamental challenges underlying networking

routing

reliable delivery

---

routing

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

reliable delivery

How do you ensure reliable transport on top of best-effort delivery?

---

routing

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

reliable delivery

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

	Intuition	Example
#1	Use tree-like topologies	Spanning-tree
#2	Rely on a global network view	Link-State SDN
#3	Rely on distributed computation	Distance-Vector BGP

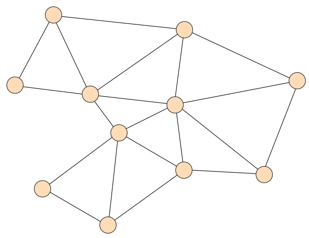
Essentially,
there are three ways to compute valid routing state

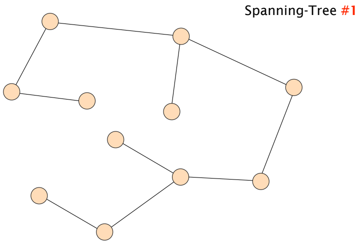
#1	Use tree-like topologies	Spanning-tree
	Rely on a global network view	Link-State SDN
	Rely on distributed computation	Distance-Vector BGP

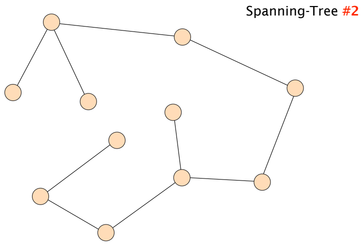
The easiest way to **avoid loops** is to route traffic on a loop-free topology

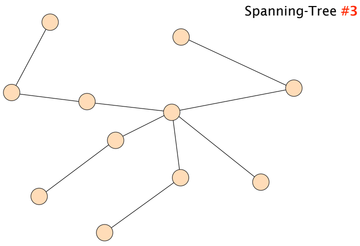
simple algorithm	Take an arbitrary topology Build a spanning tree and ignore all other links Done!
Why does it work?	Spanning-trees have only one path between any two nodes

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







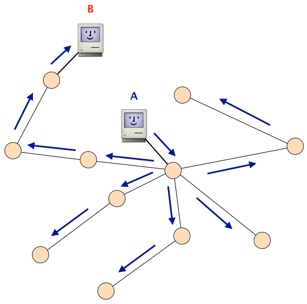


We'll see how to compute spanning-trees in 2 weeks.
For now, assume it is possible

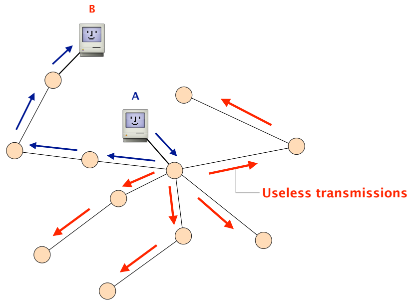
Once we have a spanning tree,
forwarding on it is easy

literally just flood
the packets everywhere

When a packet arrives,
simply send it on all ports



While flooding works,
it is quite wasteful



The issue is that nodes do not know their respective locations

Nodes can learn how to reach nodes by remembering where packets came from

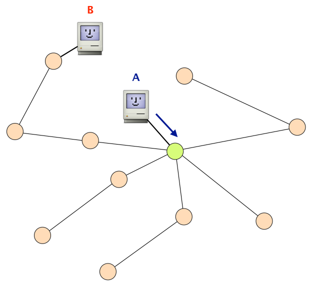
intuition

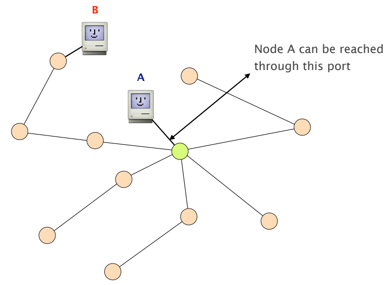
if

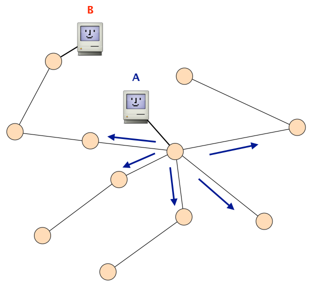
flood packet from node *A* entered switch *X* on port *4*

then

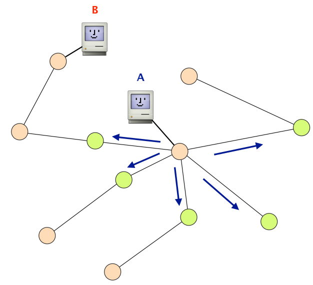
switch *X* can use port *4* to reach node *A*



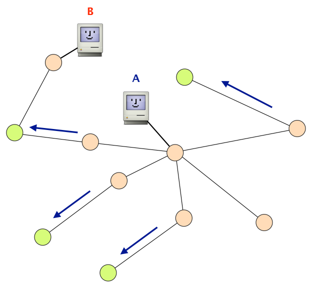




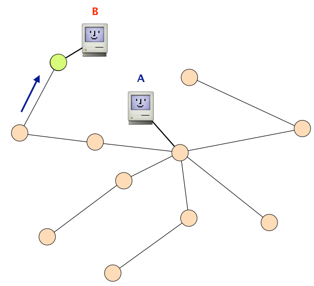
All the green nodes learn how to reach A



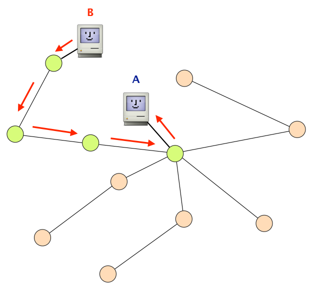
All the green nodes learn how to reach A



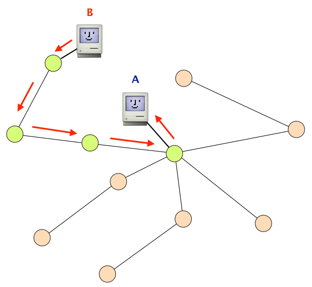
All the nodes know on which port A can be reached



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

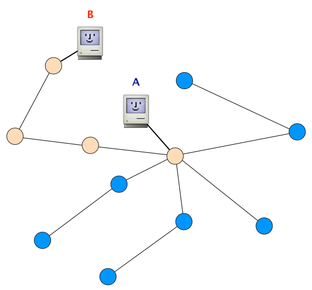


There is no need for flooding here
as the position of A is already known by everybody



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

used in Ethernet

advantages	disadvantages
plug-and-play configuration-free	mandate a spanning-tree eliminate many links from the topology
automatically adapts to moving host	slow to react to failures host movement

Essentially,
there are three ways to compute valid routing state

	Use tree-like topologies	Spanning-tree
#2	Rely on a global network view	Link-State SDN
	Rely on distributed computation	Distance-Vector BGP

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

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) = ∞
```

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)}
```

u is the node running the algorithm

S = {*u*}

for all nodes *v*:

    if (*v* is adjacent to *u*):

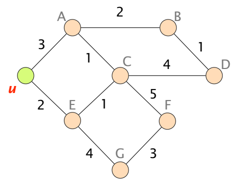
        D(*v*) = c(*u*,*v*) — *c(u,v)* is the weight of the link connecting *u* and *v*

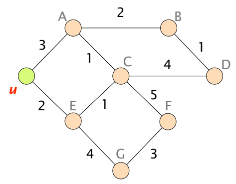
    else:

        D(*v*) = ∞

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

Let's compute the shortest-paths from *u*





Initialization

S = {*u*}

for all nodes *v*:

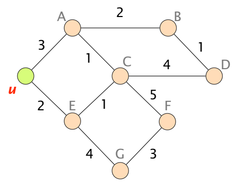
    if (*v* is adjacent to *u*):

        D(*v*) = c(*u*,*v*)

    else:

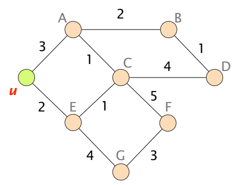
        D(*v*) = ∞

D is initialized based on u's weight, and S only contains u itself



S = {u}

D(.) =	
A	3
B	∞
C	∞
D	∞
E	2
F	∞
G	∞



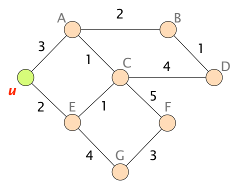
Loop

while *not* all nodes in S:

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

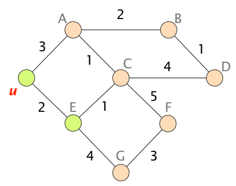
    update D(*v*) for all adjacent *v* not in S:

        D(*v*) = min{D(*v*), D(*w*) + c(*w*,*v*)}



S = {u}

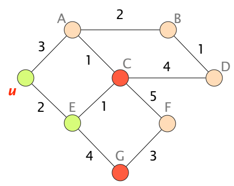
D(.) =		
A	3	
B	∞	
C	∞	
D	∞	
E	2	smallest D(*w*)
F	∞	
G	∞	



add E to S

S = {u, E}

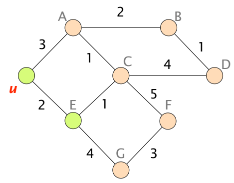
D(.) =	
A	3
B	∞
C	∞
D	∞
E	2
F	∞
G	∞



S = {u, E}

D(.) =		
A	3	
B	∞	
C	3	D(*v*) = min{∞, 2 + 1}
D	∞	
E	2	
F	∞	
G	6	D(*v*) = min{∞, 2 + 4}

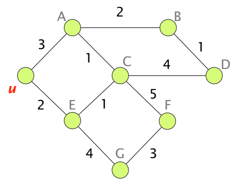
Now, do it by yourself



D(.) =   S = {u, E}

A	3
B	∞
C	3
D	∞
E	2
F	∞
G	6

Here is the final state



D(.) =   S = {u, A, B, C, D, E, F,G}

A	3
B	5
C	3
D	6
E	2
F	8
G	6

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

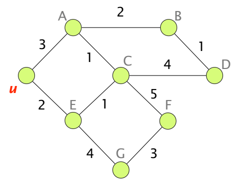
$\frac{n(n+1)}{2}$ operations => O(n^2)

This algorithm has a O(n^2) complexity

where n is the number of nodes in the graph

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

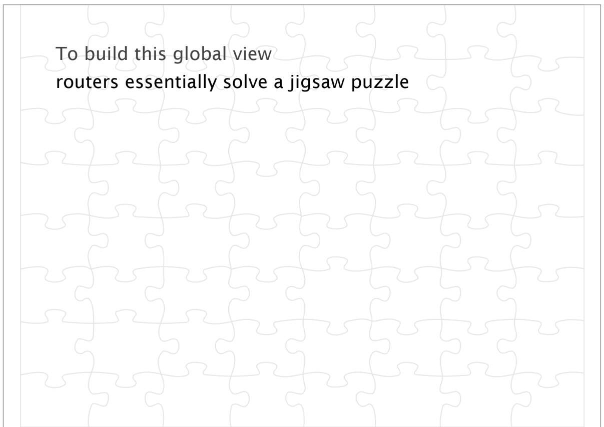
From the shortest-paths, u can directly compute its forwarding table



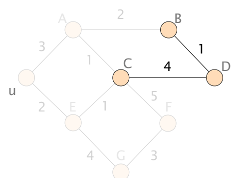
Forwarding table

destination	*next-hop*
A	A
B	A
C	E
D	A
E	E
F	E
G	E

To build this global view routers essentially solve a jigsaw puzzle



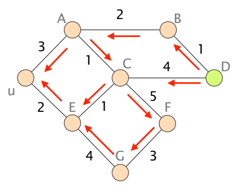
Initially, routers only know their ID and their neighbors



D only knows,
it is connected to B and C

along with the weights to reach them
(by configuration)

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

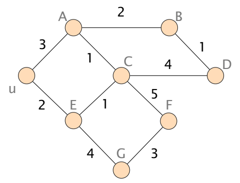


D's Advertisement

edge (D,B); cost: 1
edge (D,C); cost: 4

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

cf. exercice session
for the dynamic case

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
#3	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

Let $d_x(y)$ be the cost of the least-cost path
known by x to reach y

until convergence

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

Let $d_x(y)$ be the cost of the least-cost path
known by x to reach y

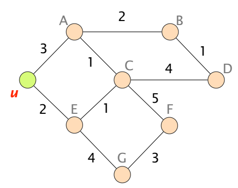
until convergence

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

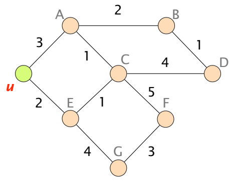
Each node updates its distances
based on neighbors' vectors:

$d_x(y) = \min\{ c(x,v) + d_v(y) \}$ over all neighbors v

Let's compute the shortest-path
from *u* to D



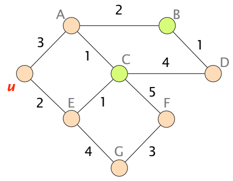
The values computed by a node *u*
depends on what it learns from its neighbors (A and E)



$d_x(y) = \min\{ c(x,v) + d_v(y) \}$
over all neighbors *v*

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

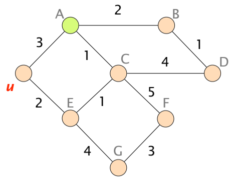
To unfold the recursion,
let's start with the direct neighbor of D



$d_B(D) = 1$

$d_C(D) = 4$

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

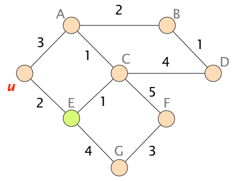


$d_A(D) = \min\{ 2 + d_B(D),$
$1 + d_C(D) \}$

1
4

$= 3$

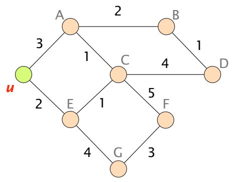
As soon as a distance vector changes,
each node propagates it to its neighbor



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

$= 5$

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

Next week on
Communication Networks

Reliable transport!