Communication Networks

Prof. Laurent Vanbever

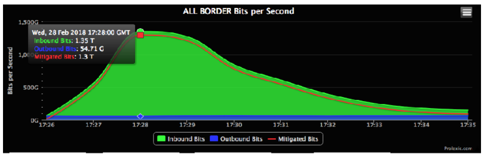
Communication Networks
Spring 2018

Laurent Vanbever
nsg.ee.ethz.ch

ETH Zürich (D-ITET)
March 5 2018

Materials inspired from Scott Shenker & Jennifer Rexford

On Feb 28, Github was the target of the largest Distributed Denial-of-Service (DDoS) attack to date



1.3 Tbps!

https://blogs.akamai.com/2018/03/memcached-fueled-13-tbps-attacks.html
https://githubengineering.com/ddos-incident-report/

Last week on
Communication Networks

Communication Networks
Part 1: General 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?

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?

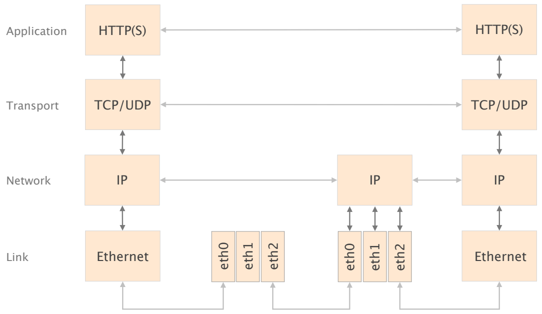
Internet communication can be decomposed in 5 independent layers (or 7 layers for the OSI model)

	layer
L5	Application
L4	Transport
L3	Network
L2	Link
L1	Physical

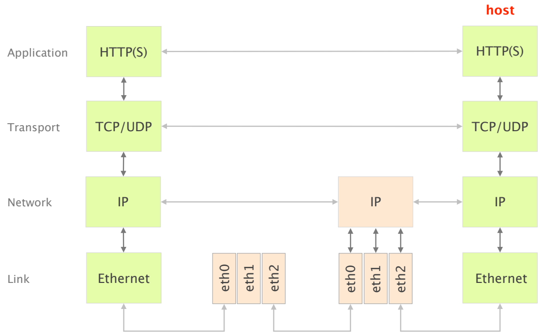
Each layer provides a service to the layer above

	layer	service provided:
L5	Application	network access
L4	Transport	end-to-end delivery (reliable or not)
L3	Network	global best-effort delivery
L2	Link	local best-effort delivery
L1	Physical	physical transfer of bits

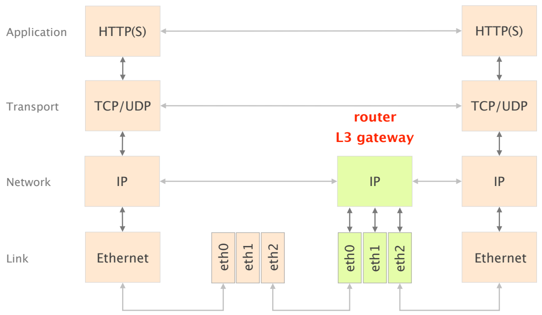
In practice, layers are distributed on every network device



Since when bits arrive they must make it to the application, all the layers exist on a host



Routers act as L3 gateway
as such they implement L2 and L3



Communication Networks

Part 1: General overview

What is a network made of?

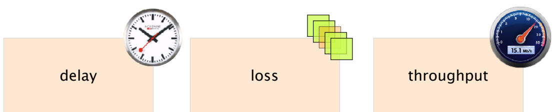
How is it shared?

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



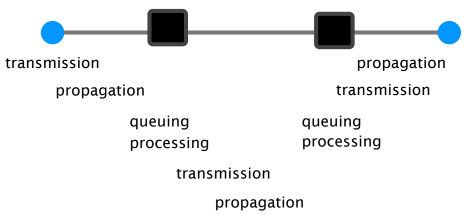
How long does it take for a packet to reach the destination

What fraction of packets sent to a destination are dropped?

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

Each packet suffers from several types of delays at *each node* along the path

transmission	delay	due to link properties
⊕ propagation	delay	
⊕ processing	delay	due to traffic mix & switch internals
⊕ queuing	delay	
= total	delay	

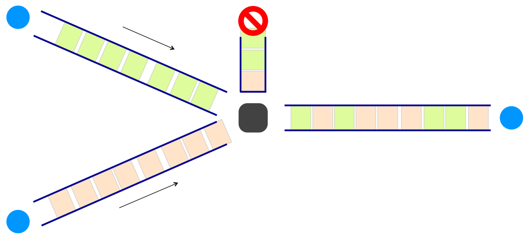


The queuing delay is the amount of time a packet waits (in a buffer) to be transmitted on a link

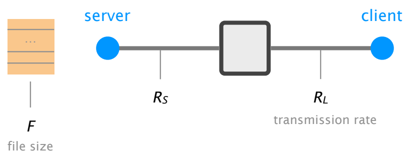
Queuing delay is the hardest to evaluate
as it varies from packet to packet

It is characterized with statistical measures
e.g., average delay & variance, probability of exceeding *x*

If the queue is persistently overloaded,
it will eventually drop packets (loss)



To compute throughput, one has to consider the bottleneck link



Average throughput $\quad \min(R_S, R_L)$

= transmission rate
of the bottleneck link

Communication Networks

Part 1: General overview

What is a network made of?

How is it shared?

How is it organized?

How does communication happen?

How do we characterize it?

This week on
Communication Networks

We will dive in the two 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?

This week

routing

Next week

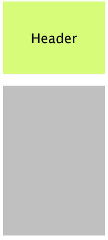
reliable delivery

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

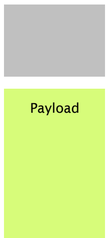
Think of IP packets as envelopes

Packet

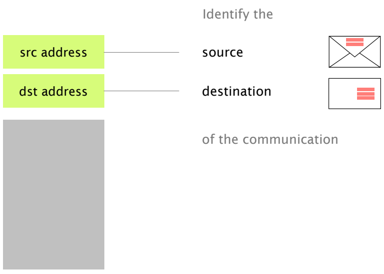
Like an envelope,
packets have a header



Like an envelope,
packets have a payload

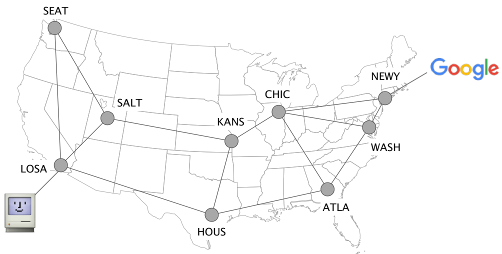


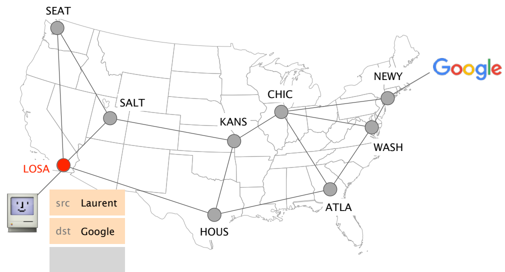
The header contains the metadata
needed to forward the packet

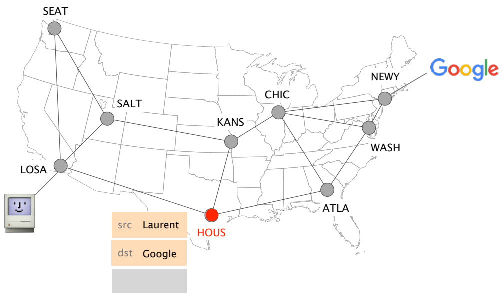


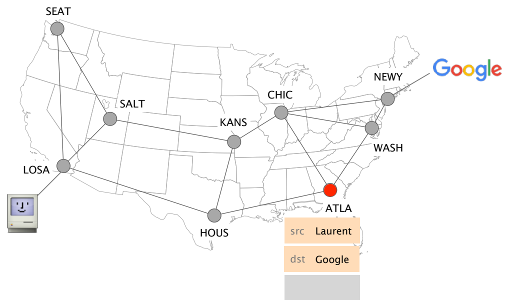
The payload contains
the data to be delivered

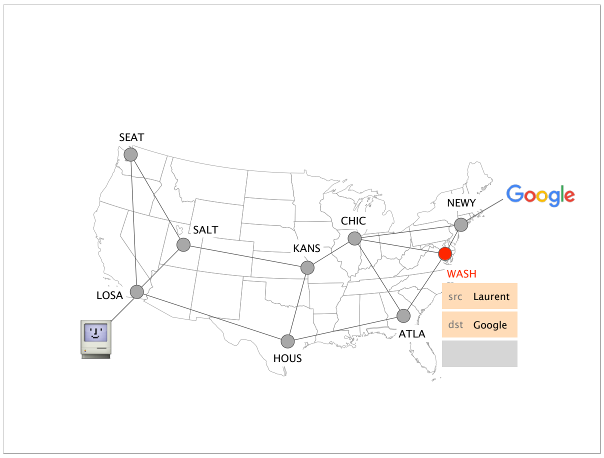
Routers forward IP packets hop-by-hop
towards their destination

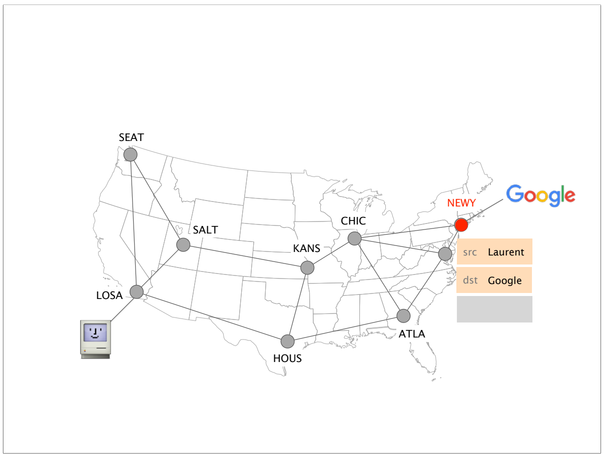


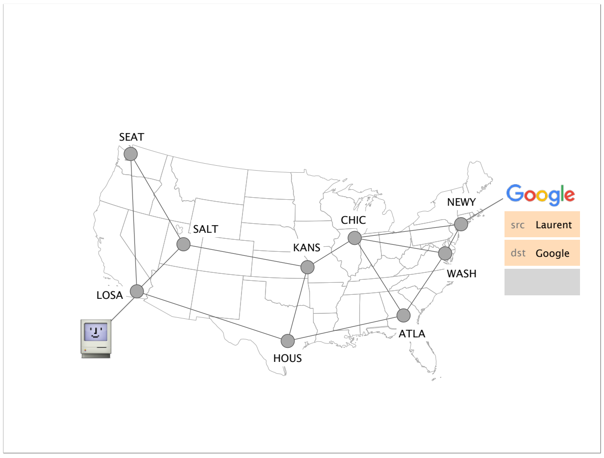


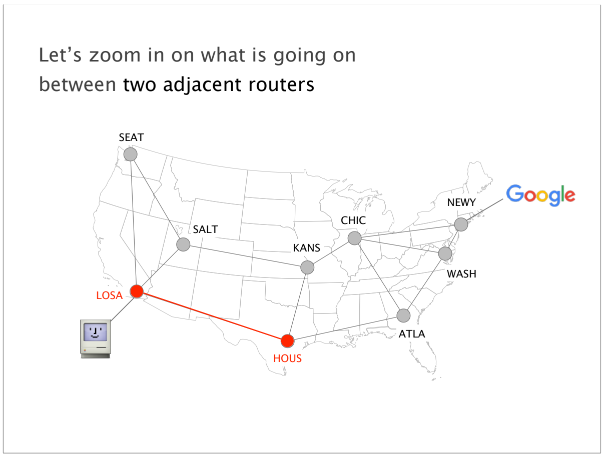


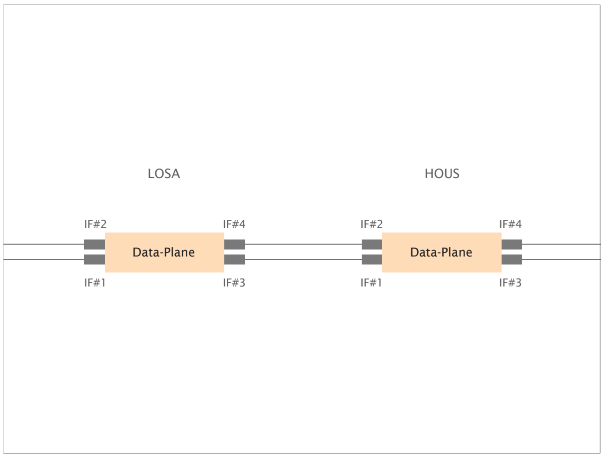


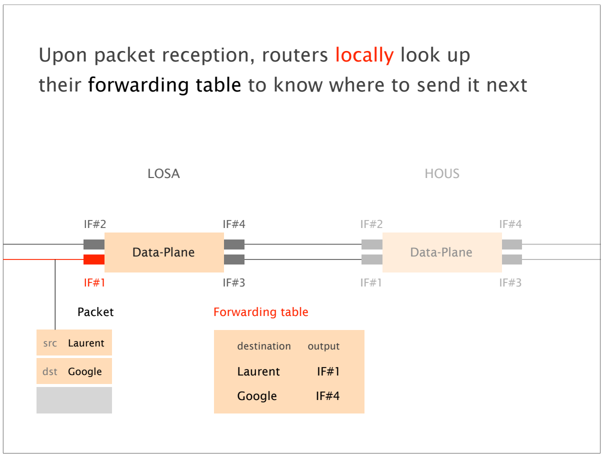

SEAT
SALT
LOSA
KANS
CHIC
NEWY
Google
WASH
HOUS
ATLA
src Laurent
dst Google

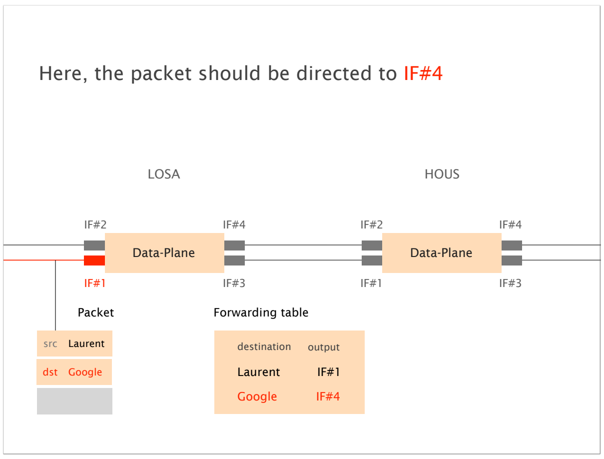

SEAT
SALT
LOSA
KANS
CHIC
NEWY
Google
HOUS
ATLA
src Laurent
dst Google

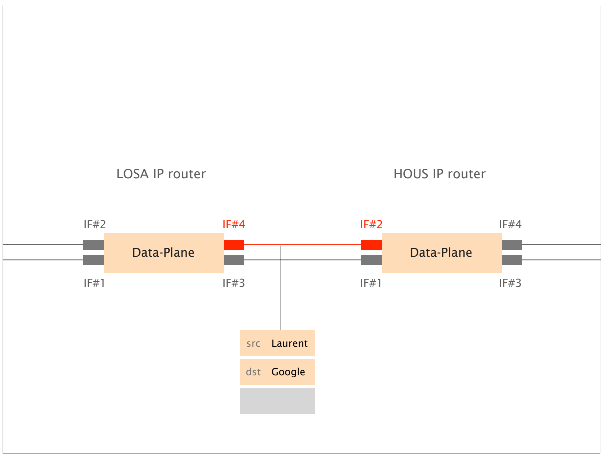

SEAT
SALT
LOSA
KANS
CHIC
NEWY
Google
WASH
HOUS
ATLA
src Laurent
dst Google


Let’s zoom in on what is going on
between two adjacent routers
SEAT
SALT
LOSA
KANS
CHIC
NEWY
Google
WASH
HOUS
ATLA

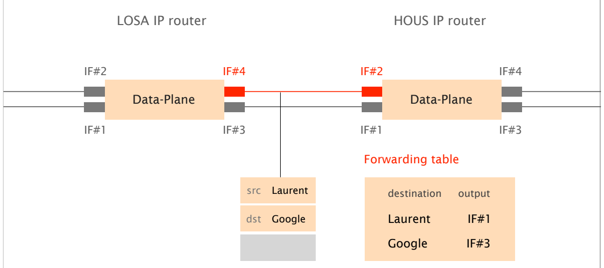

LOSA
HOUS
IF#2
IF#4
Data-Plane
IF#1
IF#3
IF#2
IF#4
Data-Plane
IF#1
IF#3

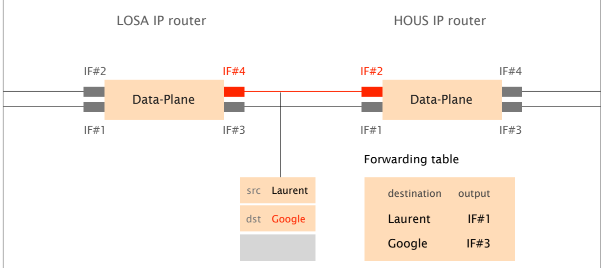

Upon packet reception, routers locally look up
their forwarding table to know where to send it next
LOSA
HOUS
IF#2
IF#4
Data-Plane
IF#1
IF#3
IF#2
IF#4
Data-Plane
IF#1
IF#3
Packet
src Laurent
dst Google
Forwarding table
destination output
Laurent IF#1
Google IF#4

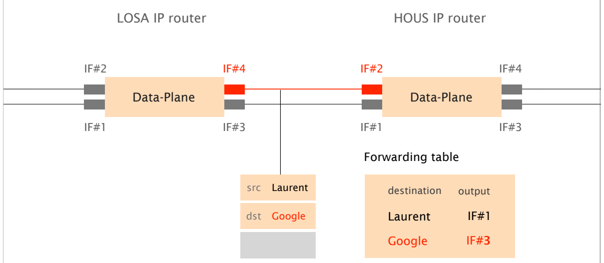

Here, the packet should be directed to IF#4
LOSA
HOUS
IF#2
IF#4
Data-Plane
IF#1
IF#3
IF#2
IF#4
Data-Plane
IF#1
IF#3
Packet
src Laurent
dst Google
Forwarding table
destination output
Laurent IF#1
Google IF#4

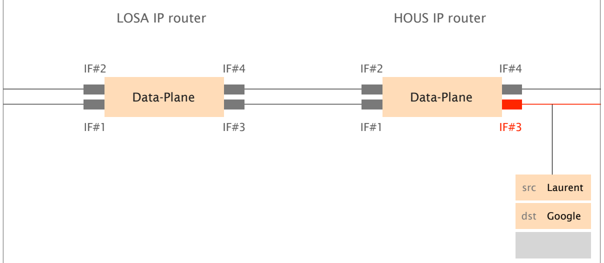

LOSA IP router
HOUS IP router
IF#2
IF#4
Data-Plane
IF#1
IF#3
IF#2
IF#4
Data-Plane
IF#1
IF#3
src Laurent
dst Google

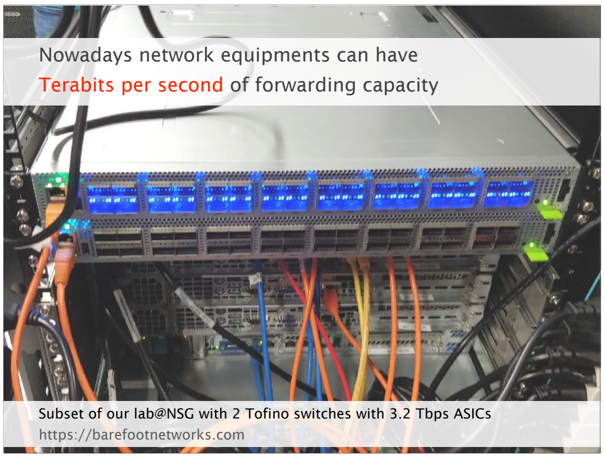
Forwarding is repeated at each router,
until the destination is reached











Subset of our lab@NSG with 2 Tofino switches with 3.2 Tbps ASICs
https://barefootnetworks.com

Forwarding decisions necessarily depend on
the destination, but can also depend on other criteria

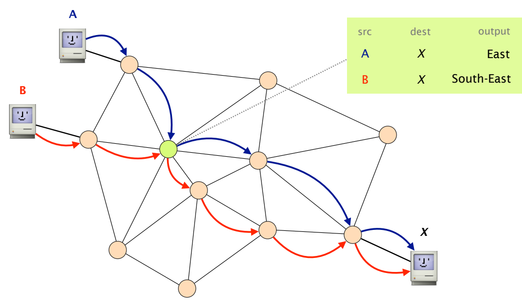
criteria		
	destination	mandatory (why?)
	source	requires n^2 state
	input port	traffic engineering
	+any other header	

destination

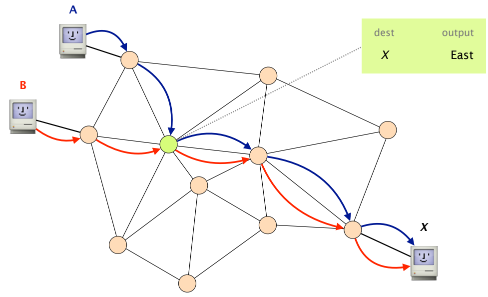
source

Let's compare these two

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



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

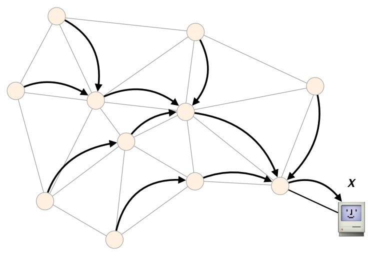


Once path to destination meet,
they will *never* split

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

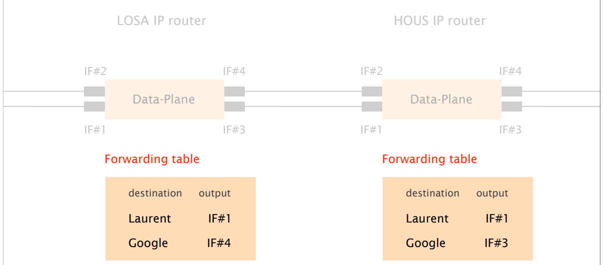
Here is an example of a spanning tree
for destination *X*

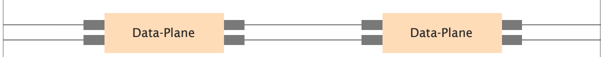


In the rest of the lecture,
we'll consider **destination-based** routing

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**

Think of the **control-plane** as the **router's brain**

Roles

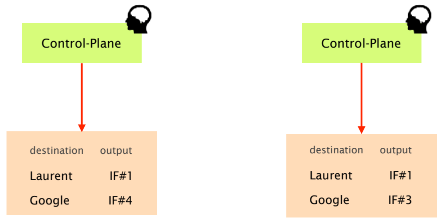
Routing

Configuration

Statistics

…

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

summary

	forwarding	routing
goal	directing packet to an outgoing link	computing the paths packets will follow
scope	local	network-wide
implem.	hardware usually	software always
timescale	nanoseconds	10s of ms hopefully

The goal of routing is to compute valid global forwarding state

Definition — a global forwarding state is valid if

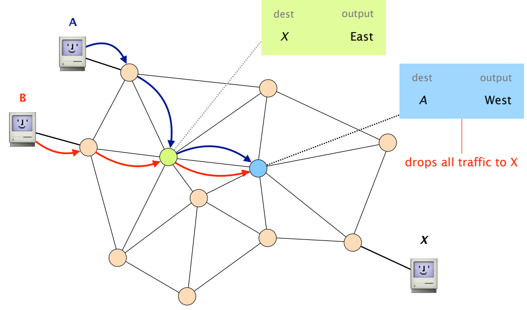
it always delivers packets to the correct destination

sufficient and necessary condition

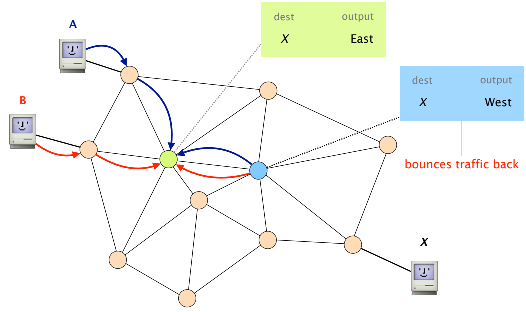
Theorem — a global forwarding state is valid if and only if

- there are no dead ends
  no outgoing port defined in the table
- there are no loops
  packets going around the same set of nodes

A global forwarding state is valid if and only if there are no dead ends



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



sufficient and necessary condition

Theorem — a global forwarding state is valid if and only if

- there are no dead ends
  i.e. no outgoing port defined in the table
- there are no loops
  i.e. packets going around the same set of nodes

Proving the **necessary** condition is **easy**

Theorem — If a routing state is valid then there are no loops or dead-end

Proof — If you run into a dead-end or a loop you'll never reach the destination so the state cannot be correct **(contradiction)**

Proving the **sufficient** condition is **more subtle**

Theorem — If a routing state has no dead end and no loop then it is valid

Proof — There is only a finite number of ports to visit

A packet can never enter a switch via the same port, otherwise it is a loop (which does not exist by assumption)

As such, the packet must eventually reach the destination

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

question 2 — How do we compute valid forwarding state?

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

How do we compute valid forwarding state?

Verifying that a routing state is **valid is easy**

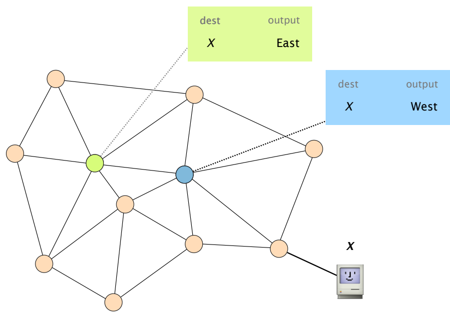
simple algorithm
for one destination

Mark all outgoing ports with an arrow

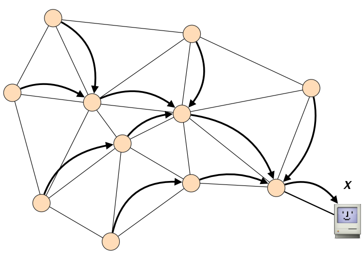
Eliminate all links with no arrow

State is valid *iff* the remaining graph is a spanning-tree

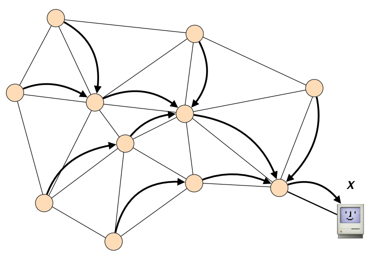
Given a graph with the corresponding forwarding state

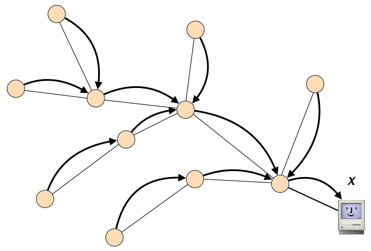


Mark all outgoing ports with an arrow

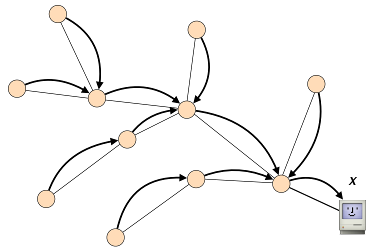


Eliminate all links with no arrow

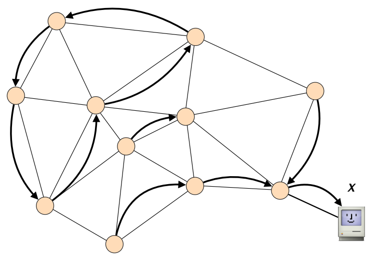




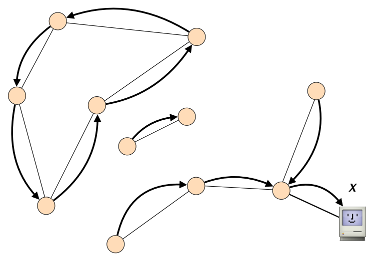
The result is a spanning tree.
This is a valid routing state



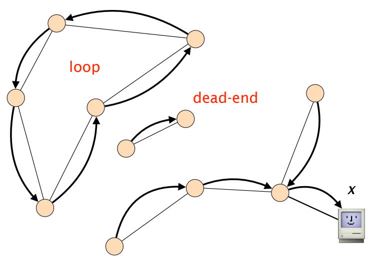
Mark all outgoing ports with an arrow



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?

question 2 How do we compute valid forwarding state?

Producing valid routing state is harder

prevent dead ends
easy

prevent loops
hard

Producing valid routing state is harder
but doable

prevent dead ends
easy

prevent loops
hard

This is the question
you should focus on

Existing routing protocols differ in
how they avoid loops

prevent loops
hard

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

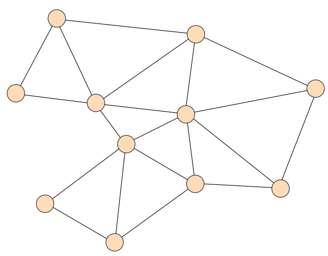
#1	Use tree-like topologies	Spanning-tree
	Rely on a global network view	Link-State SDN
	Rely on distributed computation	Distance-Vector BGP

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

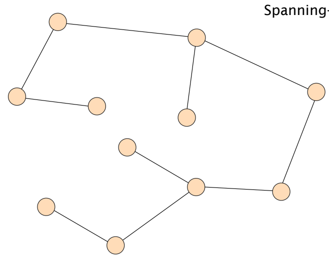
simple algorithm	Take an arbitrary topology Build a spanning tree and ignore all other links Done!
Why does it work?	Spanning-trees have only one path between any two nodes

---

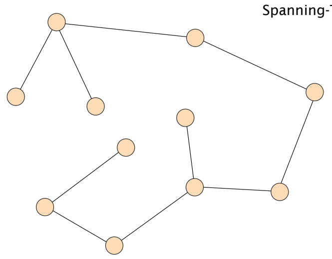
In practice,
there can be *many* spanning-trees for a given topology



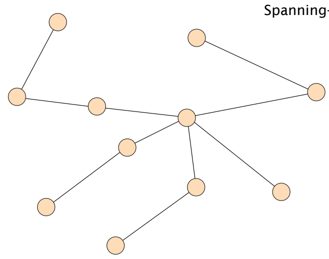
Spanning-Tree #1



Spanning-Tree #2



Spanning-Tree #3

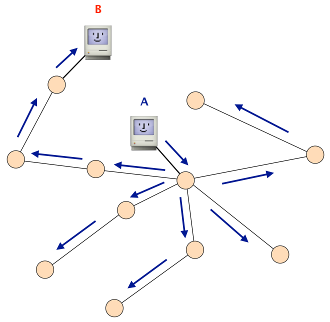


We'll see how to compute spanning-trees in 2 weeks.
For now, assume it is possible

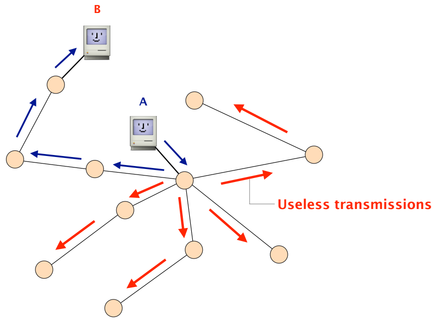
Once we have a spanning tree,
forwarding on it is easy

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

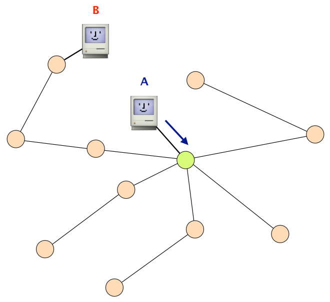
intuition

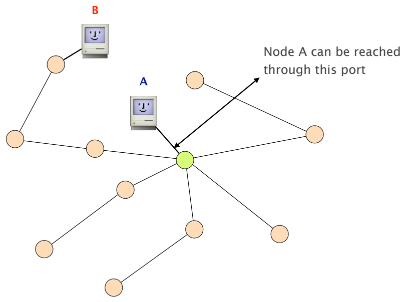
if

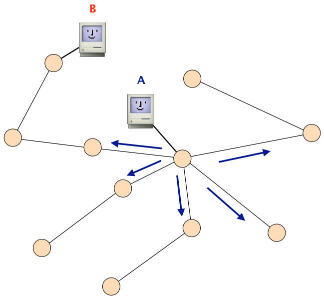
flood packet from node *A*
entered switch *X* on port *4*

then

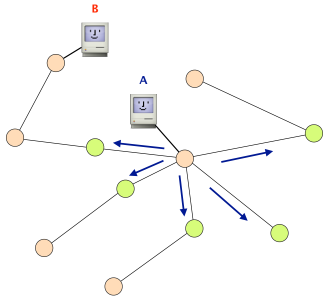
switch *X* can use port *4*
to reach node *A*



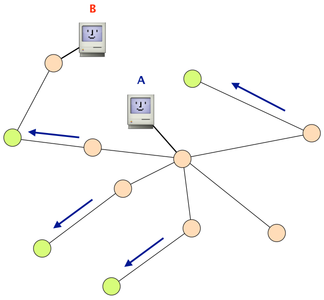




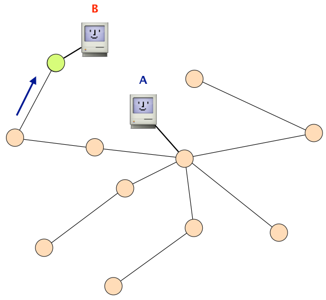
All the green nodes learn how to reach A



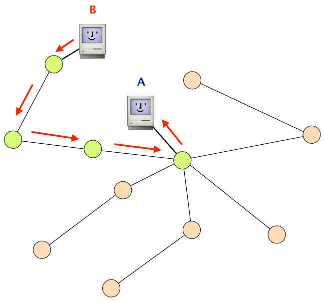
All the green nodes learn how to reach A



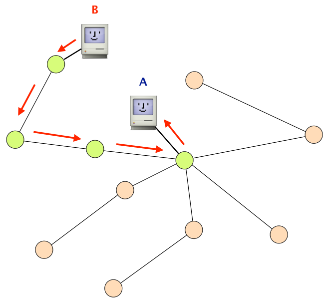
All the nodes know on which port
A can be reached



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

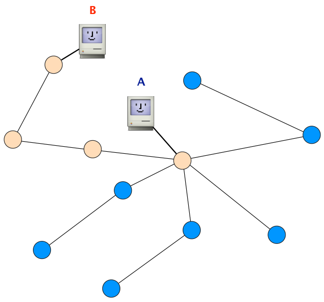


There is no need for flooding here
as the position of A is already known by everybody



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*)}

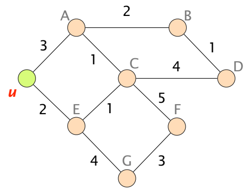
---

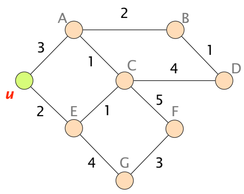
u is the node running the algorithm

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*

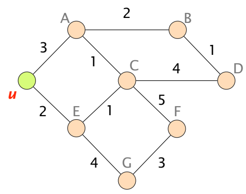




Initialization

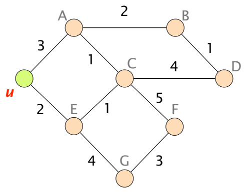
S = {*u*}
for all nodes *v*:
    if (*v* is adjacent to *u*):
        D(*v*) = c(*u*,*v*)
    else:
        D(*v*) = ∞

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



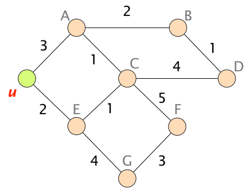
D(.) =   S = {u}

A	3
B	∞
C	∞
D	∞
E	2
F	∞
G	∞



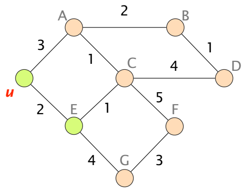
Loop

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*)}



D(.) =   S = {u}

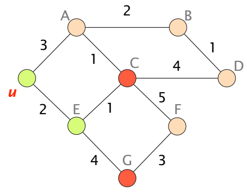
A	3	
B	∞	
C	∞	
D	∞	
E	2	smallest D(*w*)
F	∞	
G	∞	



add E to S

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

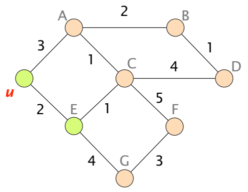
A	3
B	∞
C	∞
D	∞
E	2
F	∞
G	∞



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

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

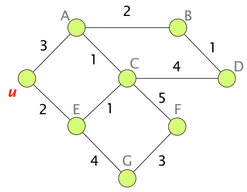
Now, do it by yourself



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

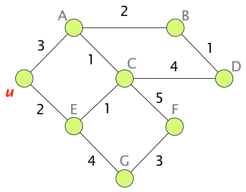
iteration #1	search for minimum through n nodes
iteration #2	search for minimum through n-1 nodes
iteration n	search for minimum through 1 node

$\frac{n(n+1)}{2}$ 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

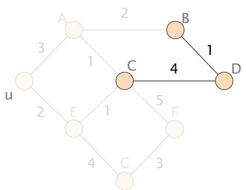


Forwarding table

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

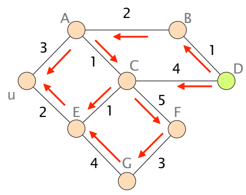
Initially,
routers only know their ID and their neighbors



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

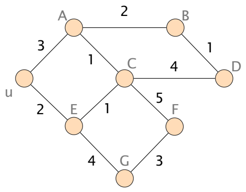


D's Advertisement

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

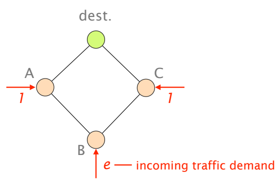
required for correctness
see exercise



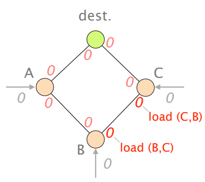
Dijkstra will always converge to a unique stable state
when run on *static* weights

what could go wrong with
changing weights?

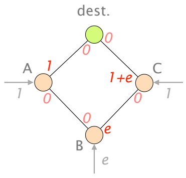
Consider this network where A, B, C send traffic
to the green destination



Unlike before,
weights are bidirectional and represent link load

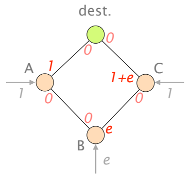


Let's assume the network starts
from this initial state



B & C sends counterclockwise
A sends clockwise

After some time,
B and C detect a better path clockwise



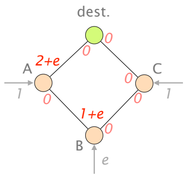
For B

cost(↺): *1+e*
cost(↻): *1*

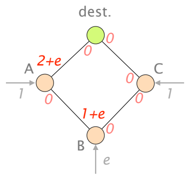
For C

cost(↺): *1+e*
cost(↻): *1*

Now, everybody sends clockwise…



After some time,
A, B, and C switch to the better path counterclockwise



For A

cost(↺): *0*
cost(↻): *2+e*

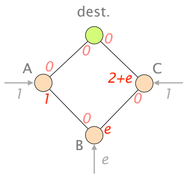
For B

cost(↺): *0*
cost(↻): *2+e*

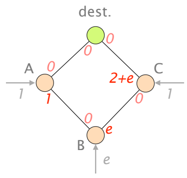
For C

cost(↺): *0*
cost(↻): *2+e*

After some time,
A, B, and C switch to the better path counterclockwise



The network is now trapped in an oscillation,
sending all traffic left, then right



For A

cost(↺): *2+e*
cost(↻): *0*

For B

cost(↺): *2+e*
cost(↻): *0*

For C

cost(↺): *2+e*
cost(↻): *0*

The problem of oscillation is fundamental to
congestion-based routing with local decisions

solution #1 — Use static weights
i.e. don't do congestion-aware routing

solution #2 — Use randomness to break self-synchronization
wait(random(0,50ms)); send(new_link_weight);

solution #3 — Have the routers agree on the paths to use
essentially meaning to rely on circuit-switching

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

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

until convergence

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

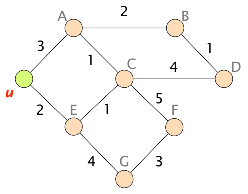
Each node bundles these distances into one message (called a vector) that it repeatedly sends to all its neighbors

until convergence

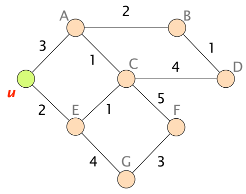
Each node updates its distances based on neighbors' vectors:

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

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



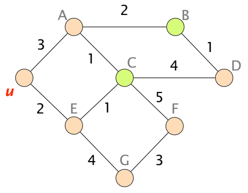
The values computed by a node u depends on what it learns from its neighbors (A and E)



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

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

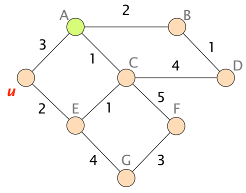
To unfold the recursion, let's start with the direct neighbor of D



$d_B(D) = 1$

$d_C(D) = 4$

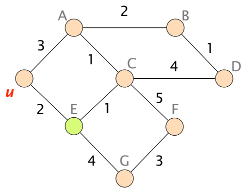
B and C announce their vector to their neighbors, enabling A to compute its shortest-path



$d_A(D) = \min\{ 2 + \overset{1}{d_B(D)}, 1 + \underset{4}{d_C(D)} \}$

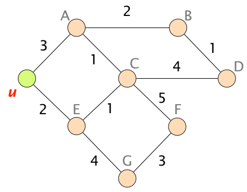
$= 3$

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

Next week on
Communication Networks

Reliable transport!

Communication Networks
Spring 2018

Laurent Vanbever
nsg.ee.ethz.ch

ETH Zürich (D-ITET)
March, 5 2018