Internet Routing Hackathon, Edition 2020
Thursday March 26, 18:00 in ETZ foyer
Register your group (3 students) starting from **Thursday March 5** (see website)
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?

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.

- 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 dive in the two fundamental challenges underlying networking:

- routing
- reliable delivery
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?
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

Mark all outgoing ports with an arrow

Eliminate all links with no arrow

State is valid \textit{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?

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
Before I give you all the answers

it’s your turn

...to figure out a way to route traffic in a network

*instructions given in class*
Essentially, there are three ways to compute valid routing state:

<table>
<thead>
<tr>
<th>Intuition</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>#1 Use tree-like topologies</td>
<td>Spanning-tree</td>
</tr>
<tr>
<td>#2 Rely on a global network view</td>
<td>Link-State</td>
</tr>
<tr>
<td></td>
<td>SDN</td>
</tr>
<tr>
<td>#3 Rely on distributed computation</td>
<td>Distance-Vector</td>
</tr>
<tr>
<td></td>
<td>BGP</td>
</tr>
</tbody>
</table>
Essentially, there are three ways to compute valid routing state:

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

**Simple algorithm**

1. Take an arbitrary topology
2. 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
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 a 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
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

<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 host movement</td>
</tr>
</tbody>
</table>
Essentially, there are three ways to compute valid routing state:

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)
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**

- $S = \{u\}$

**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 $S$:
    - $D(v) = \min\{D(v), D(w) + c(w,v)\}$

for all nodes $v$:

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

$u$ is the node running the algorithm

$S = \{u\}$
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:

\[ D(v) = \infty \]
D is initialized based on u’s weight, and S only contains u itself

\[ D(.) = \]

\[ S = \{u\} \]

\[
\begin{array}{ccccccc}
A & 3 \\
B & \infty \\
C & \infty \\
D & \infty \\
E & 2 \\
F & \infty \\
G & \infty \\
\end{array}
\]
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\}

D(.) =

\begin{align*}
A & : 3 \\
B & : \infty \\
C & : \infty \\
D & : \infty \\
E & : 2 \\
F & : \infty \\
G & : \infty \\
\end{align*}

smallest D(w)
\[
D(.) = \\
A \quad B \quad C \quad D \\
\infty \quad \infty \quad \infty \quad \infty \\
3 \quad 2 \quad 1 \quad 4 \\
\]

add E to S

\[
S = \{u, E\} \\
A \quad 3 \\
B \quad \infty \\
C \quad \infty \\
D \quad \infty \\
E \quad 2 \\
F \quad \infty \\
G \quad \infty \\
\]
$D(.) = \{u, E\}$

$A \quad 3$

$B \quad \infty$

$C \quad \boxed{3} \quad D(v) = \min\{\infty, 2 + 1\}$

$D \quad \infty$

$E \quad 2$

$F \quad \infty$

$G \quad \boxed{6} \quad D(v) = \min\{\infty, 2 + 4\}$
Now, do it by yourself

\[ D(.) = \begin{cases} u & \text{if } u \in S \\ A & 3 \\ B & \infty \\ C & 3 \\ D & \infty \\ E & 2 \\ F & \infty \\ G & 6 \end{cases} \]
Here is the final state

\[ D(.) = \{ u, A, B, C, D, E, F, G \} \]

\[
\begin{array}{c|c|c|c|c|c|c}
& A & E & C & B & D & G \\
\hline
A & 3 & 2 & 1 & 4 & 5 & 3 \\
B & & 5 & 4 & 1 & & \\
C & & & 1 & 2 & 5 & \\
D & & & & 6 & 4 & \\
E & & & & & 3 & \\
F & & & & & & 3 \\
G & & & & & & \\
\end{array}
\]
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

\[ n(n+1) \text{ operations } \Rightarrow O(n^2) \]

\[ \frac{n(n+1)}{2} \]
This algorithm has a complexity \(O(n^2)\) 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, 
\textit{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>
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 router 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. exercise session for the dynamic case.
Essentially, there are three ways to compute valid routing state:

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)
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

Each node updates its distances based on neighbors’ vectors:

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

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

\[ = 3 \]
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
How do you ensure reliable transport on top of best-effort delivery?
In the Internet, reliability is ensured by the end hosts, not by the network.
The Internet puts reliability in L4, just above the Network layer

goals

Keep the network simple, dumb
make it relatively “easy” to build and operate a network

Keep applications as network “unaware” as possible
a developer should focus on its app, not on the network

design

Implement reliability in-between, in the networking stack
relieve the burden from both the app and the network
The Internet puts **reliability in L4**, just above the Network layer.

<table>
<thead>
<tr>
<th>Layer</th>
<th>Service</th>
</tr>
</thead>
<tbody>
<tr>
<td>L4</td>
<td><strong>Transport</strong></td>
</tr>
<tr>
<td>L3</td>
<td><strong>Network</strong></td>
</tr>
</tbody>
</table>

**L4**: Reliable end-to-end delivery

**L3**: Global best-effort delivery
Recall that the Network provides a **best-effort** service, with quite poor guarantees.

<table>
<thead>
<tr>
<th>Layer</th>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>L4</td>
<td>Transport</td>
<td>reliable end-to-end delivery</td>
</tr>
<tr>
<td>L3</td>
<td>Network</td>
<td>global <strong>best-effort</strong> delivery</td>
</tr>
<tr>
<td></td>
<td>Link</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Physical</td>
<td></td>
</tr>
</tbody>
</table>
Let’s consider a simple communication between two end-points, Alice and Bob.

Alice → Internet → Bob

packet 1
packet 2
packet 3

Packet 1
Packet 2
Packet 3
IP packets can get lost or delayed
IP packets can get corrupted
IP packets can get reordered
IP packets can get duplicated
Reliable Transport

1. Correctness condition
   if-and-only if again

2. Design space
   timeliness vs efficiency vs ...

3. Examples
   Go-Back-N & Selective Repeat
Reliable Transport

Correctness condition
if-and-only if again

Design space
timeliness vs efficiency vs …

Examples
Go-Back-N & Selective Repeat
The four goals of reliable transfer

<table>
<thead>
<tr>
<th>goals</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>correctness</td>
<td>ensure data is delivered, in order, and untouched</td>
</tr>
<tr>
<td>timeliness</td>
<td>minimize time until data is transferred</td>
</tr>
<tr>
<td>efficiency</td>
<td>optimal use of bandwidth</td>
</tr>
<tr>
<td>fairness</td>
<td>play well with concurrent communications</td>
</tr>
</tbody>
</table>
goals

**correctness**

ensure data is delivered, in order, and untouched
Routing had a **clean sufficient and necessary correctness 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
We need the same kind of “if and only if” condition for a “correct” reliable transport design
A reliable transport design is correct if…

attempt #1 packets are delivered to the receiver

Wrong Consider that the network is partitioned

We cannot say a transport design is *incorrect* if it doesn’t work in a partitioned network…
A reliable transport design is correct if...

**attempt #2**

packets are delivered to receiver if and only if it was possible to deliver them

**Wrong**

If the network is only available one instant in time, only an oracle would know when to send

We cannot say a transport design is *incorrect* if it doesn’t know the unknowable
A reliable transport design is correct if...

attempt #3 It resends a packet if and only if the previous packet was lost or corrupted

Wrong Consider two cases

- packet made it to the receiver and all packets from receiver were dropped
- packet is dropped on the way and all packets from receiver were dropped
A reliable transport design is correct if…

**attempt #3**

It resends a packet if and only if the previous packet was lost or corrupted.

**Wrong**

In both case, the sender has no feedback at all.

Does it resend or not?
A reliable transport design is correct if...

attempt #3  It resends a packet if and only if the previous packet was lost or corrupted

Wrong

**but better** as it refers to what the design does (which it can control), not whether it always succeeds (which it can’t)
A reliable transport design is correct if...

attempt #4  
A packet is *always resent* if  
the previous packet was lost or corrupted  

A packet *may be resent* at other times

Correct!
A transport mechanism is correct if and only if it resends all dropped or corrupted packets.

Sufficient

“if”

algorithm will always keep trying to deliver undelivered packets

Necessary

“only if”

if it ever let a packet go undelivered without resending it, it isn’t reliable

Note

it is ok to give up after a while but must announce it to the application
Reliable Transport

Correctness condition
if-and-only if again

Design space
timeliness vs efficiency vs …

Examples
Go-Back-N & Selective Repeat
Now, that we have a correctness condition how do we achieve it and with what tradeoffs?

Design a **correct, timely, efficient** and **fair** transport mechanism knowing that packets can get **lost**
corrupted reordered delayed duplicated let’s focus on these aspects first
Alice

for word in list:
    send_packet(word);
    set_timer();

upon timer going off:
    if no ACK received:
        send_packet(word);
        reset_timer();
    upon ACK:
        pass;

Bob

receive_packet(p);
if check(p.payload) == p.checksum:
    send_ack();

    if word not delivered:
        deliver_word(word);
else:
    pass;
There is a clear tradeoff between timeliness and efficiency in the selection of the timeout value.

for word in list:
    send_packet(word);
    set_timer();

upon timer going off:
    if no ACK received:
        send_packet(word);
        reset_timer();
    upon ACK:
        pass

receive_packet(p);
    if check(p.payload) == p.checksum:
        send_ack();
    if word not delivered:
        deliver_word(word);
    else:
        pass;
Timeliness argues for small timers, efficiency for large ones.

- **Timeliness**
  - Small timers
  - Risk
  - Unnecessary retransmissions

- **Efficiency**
  - Large timers
  - Risk
  - Slow transmission
Even with short timers, the timeliness of our protocol is extremely poor: one packet per Round-Trip Time (RTT)
An obvious solution to improve timeliness is to send multiple packets at the same time.

- **Approach**
  - Add sequence number inside each packet
  - Add buffers to the sender and receiver
    - *sender*: Store packets sent & not acknowledged
    - *receiver*: Store out-of-sequence packets received
4 packets sent w/o ACKs

Packet diagram:

- Packet 1
- Packet 2
- Packet 3
- Packet 4
Sending multiple packets improves timeliness, but it can also overwhelm the receiver.

Supercomputer sends 1000 packets per second, which can overwhelm the smartphone that can process only 10 packets per second.
To solve this issue, we need a mechanism for **flow control**.
Using a sliding window is one way to do that

Sender keeps a list of the sequence # it can send
known as the *sending window*

Receiver also keeps a list of the acceptable sequence #
known as the *receiving window*

Sender and receiver negotiate the window size

*sending window* $\leq$ *receiving window*
Example with a window composed of 4 packets

<table>
<thead>
<tr>
<th>ACKed packets</th>
<th>unACK’ed packets</th>
<th>forbidden packets</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3</td>
<td>4 5 6 7</td>
<td>8 9 10 11 ...</td>
</tr>
</tbody>
</table>
Window after sender receives **ACK 4**

- **ACKed packets**
- **unACK’ed packets**
- **available packets**
- **forbidden packets**
Timeliness of the window protocol depends on the size of the sending window
Assuming infinite buffers, how big should the window be to maximize timeliness?

What should be the value of $W$? (in bytes)
Timeliness matters, but what about efficiency?
The efficiency of our protocol essentially depends on two factors:

- **receiver feedback**
- **behavior upon losses**

**How much information does the sender get?**

**How does the sender detect and react to losses?**
The efficiency of our protocol essentially depends on two factors.

receiver feedback

behavior upon losses

How much information does the sender get?
ACKing individual packets provides detailed feedback, but triggers unnecessary retransmission upon losses.

Advantages
- Know fate of each packet
- Simple window algorithm
- W single-packet algorithms
- Not sensitive to reordering

Disadvantages
- Loss of an ACK packet
- Requires a retransmission
- Causes unnecessary retransmission
Cumulative ACKs enables to recover from lost ACKs, but provides coarse-grained information to the sender.

<table>
<thead>
<tr>
<th>approach</th>
<th>ACK the highest sequence number for which all the previous packets have been received</th>
</tr>
</thead>
<tbody>
<tr>
<td>advantages</td>
<td>recover from lost ACKs</td>
</tr>
<tr>
<td>disadvantages</td>
<td>confused by reordering, incomplete information about which packets have arrived, causes unnecessary retransmission</td>
</tr>
</tbody>
</table>
**Full Information Feedback** prevents unnecessary retransmission, but can induce a sizable overhead.

<table>
<thead>
<tr>
<th>Approach</th>
<th>List all packets that have been received highest cumulative ACK, plus any additional packets</th>
</tr>
</thead>
</table>

**Advantages**
- Complete information
- Resilient form of individual ACKs

**Disadvantages**
- Overhead (hence lowering efficiency)
  - E.g., when large gaps between received packets
We see that Internet design is 
all about balancing tradeoffs (again)
The efficiency of our protocol essentially depends on two factors:

- Receiver feedback
- Behavior upon losses

How does the sender detect and react to losses?
As of now, we detect loss by using timers. That’s only one way though.
Losses can also be detected by relying on ACKs
With individual ACKs, missing packets (gaps) are implicit

Assume packet 5 is lost
but no other

ACK stream
1
2
3
4
6
7
...
With full information, missing packets (gaps) are explicit

Assume packet 5 is lost
but no other

ACK stream  
up to 1  
up to 2  
up to 3  
up to 4  
up to 4, plus 6  
up to 4, plus 6—7  
...  
sender learns that 5 is missing  
retransmits after $k$ packets
With cumulative ACKs, missing packets are harder to know

Assume packet 5 is lost

but no other

ACK stream

1
2
3
4
4 sent when 6 arrives
4 sent when 7 arrives
...

Duplicated ACKs are a sign of isolated losses. Dealing with them is trickier though.

Lack of ACK progress means that 5 hasn’t made it

Stream of ACKs means that (some) packets are delivered

Sender could trigger resend upon receiving $k$ duplicates ACKs

but *what do you resend?*

only 5 or 5 and everything after?
What about fairness?

Design a *correct*, *timely*, *efficient* and *fair* transport mechanism knowing that packets can get lost, corrupted, reordered, delayed, duplicated.
When $n$ entities are using our transport mechanism, we want a fair allocation of the available bandwidth.
Consider this simple network in which three hosts are sharing two links.

What is a fair allocation for the 3 flows?
An equal allocation is certainly “fair”, but what about the efficiency of the network?

Total traffic is 1.5 Gbps
Fairness and efficiency don’t always play along, here an unfair allocation ends up *more efficient*.

Total traffic is 2 Gbps!
What is fair anyway?
Equal-per-flow isn’t really fair as (A,C) crosses two links: it uses *more* resources.

Total traffic is 1.5 Gbps.
With equal-per-flow, A ends up with 1 Gbps because it sends 2 flows, while B ends up with 500 Mbps

Is it fair?
Seeking an exact notion of fairness is not productive. What matters is to avoid starvation.

equal-per-flow is good enough for this
Simply dividing the available bandwidth doesn’t work in practice since flows can see different bottleneck links.
Intuitively, we want to give users with "small" demands what they want, and evenly distribute the rest

Max-min fair allocation is such that

the lowest demand is maximized

after the lowest demand has been satisfied,
the second lowest demand is maximized

after the second lowest demand has been satisfied,
the third lowest demand is maximized

and so on...
Max-min fair allocation can easily be computed

step 1  Start with all flows at rate 0
step 2  Increase the flows until there is a new bottleneck in the network
step 3  Hold the fixed rate of the flows that are bottlenecked
step 4  Go to step 2 for the remaining flows

Done!
Max-min fair allocation can be approximated by slowly increasing $W$ until a loss is detected.

**Intuition**

Progressively increase the sending window size. Whenever a loss is detected, decrease the window size. Repeat.

$\text{max}=\text{receiving window}$

$\text{signal of congestion}$
Design a *correct, timely, efficient* and *fair* transport mechanism knowing that packets can get *lost*, *corrupted*, *reordered*, *delayed*, and *duplicated*. 
Dealing with corruption is easy:
Rely on a checksum, treat corrupted packets as lost
The effect of **reordering** depends on the type of ACKing mechanism used:

<table>
<thead>
<tr>
<th>Type</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>individual ACKs</td>
<td>no problem</td>
</tr>
<tr>
<td>full feedback</td>
<td>no problem</td>
</tr>
<tr>
<td>cumm. ACKs</td>
<td>create duplicate ACKs</td>
</tr>
<tr>
<td></td>
<td>why is it a problem?</td>
</tr>
</tbody>
</table>
Long *delays* can create useless timeouts, for all designs
Packets duplicates can lead to duplicate ACKs whose effects will depend on the ACKing mechanism used.

- individual ACKs: no problem
- full feedback: no problem
- cumm. ACKs: problematic
Design a *correct, timely, efficient* and *fair* transport mechanism knowing that packets can get *lost, corrupted, reordered, delayed, duplicated*.
Here is one correct, timely, efficient and fair transport mechanism

ACKing
full information ACK

retransmission
after timeout
after $k$ subsequent ACKs

window management
additive increase upon successful delivery
multiple decrease when timeouts

We'll come back to this when we see TCP
Reliable Transport

Correctness condition
if-and-only if again

Design space
timeliness vs efficiency vs ...

Examples
Go-Back-N & Selective Repeat
Go-Back-N (GBN) is a simple sliding window protocol using cumulative ACKs.

**principle**
receiver should be as simple as possible

**receiver**
delivers packets in-order to the upper layer
for each received segment,
ACK the last in-order packet delivered (cumulative)

**sender**
use a single timer to detect loss, reset at each new ACK
upon timeout, resend all $W$ packets
starting with the lost one
Selective Repeat (SR) avoid unnecessary retransmissions by using per-packet ACKs

principle avoids unnecessary retransmissions

receiver acknowledge each packet, in-order or not
buffer out-of-order packets

sender use per-packet timer to detect loss
upon loss, only resend the lost packet

see Book 3.4.3
Let’s see how it works in practice visually

http://www.ccs-labs.org/teaching/rn/animations/gbn_sr/
Reliable Transport

Correctness condition
if-and-only if again

Design space
timeliness vs efficiency vs …

Examples
Go-Back-N & Selective Repeat
Next week on Communication Networks

Ethernet and Switching