Last week on Communication Networks

Last week on Communication Networks

We explored the concepts behind routing

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

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

Materials inspired from Scott Shenker & Jennifer Rexford

nsg.ee.ethz.ch

Laurent Vanbever

ETH Zürich (D-ITET)

March 12 2018

Communication Networks

Spring 2018

Laurent Vanbever

nsg.ee.ethz.ch
Forwarding is repeated at each router, until the destination is reached.

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

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>usually</td>
<td>always</td>
<td></td>
</tr>
<tr>
<td>timescale</td>
<td>nanoseconds</td>
<td>10s of ms</td>
</tr>
</tbody>
</table>

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

Verifying that a forwarding state is valid is easy.

There are 3 ways to compute valid routing state:

1. **Intuition**
   - **Example**
     - Use tree-like topologies: Spanning tree
     - Rely on a global network view: Link-State
     - Rely on distributed computation: Distance-Vector

This week on Communication Networks
In the Internet, reliability is ensured by the end hosts, not by the network.

The Internet puts reliability in the L4 layer, 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

**Design**
- Implement reliability in-between, in the networking stack
- Relieve the burden from both the app and the network

Recall that the Network provides a best-effort service, with quite poor guarantees.

**Layer**
- Application
- L4 Transport: reliable end-to-end delivery
- L3 Network: global best-effort delivery
- Link
- Physical

IP packets can get lost or delayed.

Example scenarios:
- IP packets can get corrupted.
- IP packets can get reordered.

In the Internet, IP packets can get lost or delayed.
IP packets can get duplicated

Internet

packet 1

packet 2

packet 3

Alice —— Internet —— Bob

Now, it’s your turn
...to design a Internet protocol
instructions given in class

Reliable Transport

1. Correctness condition
   if-and-only-if again

2. Design space
   timeliness vs efficiency vs ...

3. Examples
   Go-Back-N & Selective Repeat

The four goals of reliable transfer

- correctness
  ensure data is delivered, in order, and untouched
- timeliness
  minimize time until data is transferred
- efficiency
  optimal use of bandwidth
- fairness
  play well with concurrent communications

Routing had a clean sufficient and necessary correctness condition

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

We need the same kind of “if and only if” condition for a “correct” reliable transport design
### Communication Networks

**Reliable Transport**

<table>
<thead>
<tr>
<th>Attempt #1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A reliable transport design is correct if...</td>
</tr>
<tr>
<td>packets are delivered to the receiver</td>
</tr>
</tbody>
</table>

**Wrong**

Consider that the network is partitioned
We cannot say a transport design is incorrect if it doesn’t work in a partitioned network...

**Attempt #2**

A reliable transport design is correct if...
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

**Attempt #3**

A reliable transport design is correct if...
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

**Attempt #4**

A reliable transport design is correct if...
A packet is always resent if the previous packet was lost or corrupted
A packet may be resent at other times

**Correct!**

but better as it refers to what the design does (which it can control), not whether it always succeeds (which it can’t)

### Note

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

**Sufficient**
- algorithm will always keep trying to deliver undelivered packets

**Necessary**
- 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
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** or **duplicated**.

Timeliness argues for small timers, efficiency for large ones.

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.

Sending multiple packets improves timeliness, but it can also overwhelm the receiver.

To solve this issue, we need a mechanism for **flow control**.

Using a **sliding window** is one way to do that.
Example with a window composed of 4 packets

<table>
<thead>
<tr>
<th></th>
<th>ACKed packets</th>
<th>unACK'ed packets</th>
<th>forbidden packets</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Window after sender receives ACK 4

<table>
<thead>
<tr>
<th></th>
<th>ACKed packets</th>
<th>unACK'ed packets</th>
<th>forbidden packets</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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?

100 Mbps, 5 ms (one-way)

What should be the value of W? (in bytes)

The efficiency of our protocol essentially depends on two factors

How much information does the sender get?

How does the sender detect and react to losses?

ACKing individual packets provides detailed feedback, but triggers unnecessary retransmission upon losses

<table>
<thead>
<tr>
<th>advantages</th>
<th>disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>know fate of each packet</td>
<td>loss of an ACK packet requires a retransmission causes unnecessary retransmission</td>
</tr>
<tr>
<td>simple window algorithm</td>
<td>W single-packet algorithms</td>
</tr>
<tr>
<td>not sensitive to reordering</td>
<td></td>
</tr>
</tbody>
</table>
Cumulative ACKs enables to recover from lost ACKs, but provides coarse-grained information to the sender

- **Approach**: ACK the highest sequence number for which all the previous packets have been received

- **Advantages**: recover from lost ACKs

- **Disadvantages**: confused by reordering, incomplete information about which packets have arrived, causes unnecessary retransmission

- **Full Information Feedback** prevents unnecessary retransmission, but can induce a sizable overhead

- **Approach**: List all packets that have been received, highest cumulative ACK, plus any additional packets

- **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

<table>
<thead>
<tr>
<th>ACK stream</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>sender can infer that 5 is missing</strong> and resend 5 after $k$ subsequent packets</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

With full information, missing packets (gaps) are explicit

Assume packet 5 is lost but no other

<table>
<thead>
<tr>
<th>ACK stream</th>
<th>up to 1</th>
<th>up to 2</th>
<th>up to 3</th>
<th>up to 4</th>
<th>up to 4, plus 6</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>sender learns that 5 is missing</strong> retransmits after $k$ packets</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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.

situation
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

Consider this simple network
in which three hosts are sharing two links

A
1 Gbps

B
1 Gbps

C

flow 1
flow 2
flow 3

What is a fair allocation for the 3 flows?

An equal allocation is certainly “fair”,
but what about the efficiency of the network?

A
1 Gbps

B
1 Gbps

C

flow 1
500 Mbps
flow 2
500 Mbps
flow 3
500 Mbps

Total traffic is 1.5 Gbps

Fairness and efficiency don’t always play along,
here an unfair allocation ends up more efficient

A
1 Gbps

B
1 Gbps

C

flow 1
1 Gbps
flow 2
1 Gbps
flow 3
0 Mbps

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.

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.

Max-min fair allocation can easily be computed

Max-min fair allocation can be approximated by slowly increasing $W$ until a loss is detected.
Design a correct, timely, efficient and fair transport mechanism knowing that packets can get lost, corrupted, reordered, delayed, 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:

- Individual ACKs: no problem
- Full feedback: no problem
- Cumm. ACKs: create duplicate ACKs, why is it a problem?

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

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

---

Finite State Machine for the receiver
see Book 3.4.3

Finite State Machine for the sender
see Book 3.4.3

Selective Repeat (SR) avoid unnecessary retransmissions by using per-packet ACKs
see Book 3.4.3

**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

---

Let’s see how it works in practice
**visually**

http://www.ccs-labs.org/teaching/ecn/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**

Source: Andrew Hart (Flickr)