Computer Networks: Transport

Ankit Singla

Several slides adapted from Jennifer Rexford, Scott Shenker, Laurent Vanbever
Where we are in the course ...

Part 1: Overview & Principles

Part 2: Applications

Part 3: Transport

Part 4: Algorithms
Computer networks

Part 3: Transport

#1 What do we need in the transport layer?

#2 How do we build reliable transport?

#3 How does the Internet’s transport work?

#4 Sockets: the application $\leftrightarrow$ transport interface
Computer networks

Part 3: Transport

#1 What do we need in the transport layer?

#2 How do we build reliable transport?

#3 How does the Internet’s transport work?

#4 Sockets: the application ↔ transport interface
What do we need in the transport layer?

Network
- Keep minimal (easy to build, broadly applicable)
- Global best-effort packet delivery

Applications
- Keep minimal (easy to write)
- Restricted to app-specific functionality

“Host networking stack”
- The shared networking code on the host
- This relieves burden from both app and network
- The transport layer is a key component here
What do we need in the transport layer?

Data delivery to the correct application
  IP just points towards next protocol
  Transport needs to demultiplex incoming data (ports)

Files or byte-streams abstractions for applications
  Network deals with packets
  Transport layer needs to translate between them

Reliable transfer (if needed)

Not overloading the receiver

Not overloading the network
Computer networks

Part 3: Transport

#1 What do we need in the transport layer?

#2 How do we build reliable transport?

#3 How does the Internet’s transport work?

#4 Sockets: the application ↔ transport interface
How do we build reliable transport?

YOU
Alice wants to transmit a text word-by-word to Bob via the Internet.
The Internet is an unreliable environment though
Data packets can get lost
Data packets can get corrupted
Data packets can get reordered
Data packets can get duplicated
Your job is to build a reliable way
to communicate with Bob

correctness  Bob should read exactly what you’ve typed
             in the same order, and without any gap

timeliness  Bob should receive the complete text as fast as possible
             minimize time until data is transferred

efficiency  Minimize the use of bandwidth
             don’t send too many packets
Your task in 15 minutes

Design a protocol that can deal with packet loss, corruption, reordering, and duplication
Design a protocol that can deal with packet loss, corruption, reordering, and duplication.

What fields do you add to the packets?
What code do you run on the end-points?

Treat “words” and “packets” interchangeably.
Assume L3 provides functions send_word() and receive_word()
Even “Donaudampfschifffahrtsgesellschaftskapitänswitwe” is ok :)
Sub-tasks and what I expect

**first**  Pseudo-code for protocol to send one word at a time
while handling loss, corruption, and duplication

**next**  Think about how to send multiple words at a time
and address reordering

**output**  Sender and receiver procedures; packet headers
Idea for >1 outstanding packets

Groups of at most 4. I’ll ask some of you to explain your solution!
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;

Alice

receive_packet(p);
if check(p.payload) == p.checksum:
    send_ack();
else:
    if word not delivered:
        deliver_word(word);
    else:
        pass;

Bob
1 Correctness condition
   if-and-only if …

2 Design space
   timeliness vs efficiency vs …

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

1. **Correctness condition**
   if-and-only if …

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
correctness  ensure data is delivered, in order, and untouched
We want a crisp, formal translation of this correctness goal
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 cases, 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 reliable 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 resenting it, it isn’t reliable

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

1. Correctness condition
   if-and-only if …

2. Design space
   timeliness vs efficiency vs …

3. 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
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, our protocol is extremely slow:
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
Sending multiple packets improves timeliness, but it can also overwhelm the receiver.

---

**Diagram:**

- Supercomputer sends 1000 packets per second.
- Smartphone can process 10 packets per second.
- Overwhelmed when the number of packets exceeds the processing capacity.
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 <= receiving window
Example with a window composed of 4 packets

0 1 2 3 4 5 6 7

unACK'ed packets

forbidden packets

ACKed packets

available packets

8 9 10 11 ...

Window after sender receives ACK 4
Window after sender receives **ACK 4**

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

What should be the value of \( W \)?

(in bytes)

“Bandwidth-delay product (BDP)”
Timeliness matters,
but what about efficiency?
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?