Computer Networks: UDP and TCP

Ankit Singla

Several slides adapted from Jennifer Rexford, Scott Shenker, Laurent Vanbever

Photo: ETH Zürich / Gian Marco Castelberg
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 ↔ transport interface
What do we need in the transport layer?

- Data delivery to the correct application
  - IP just points towards next protocol

- 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
How does Internet transport provide these?

Data delivery to the correct application
  Demultiplexing using application identifiers (ports)

Files or byte-streams abstractions for applications
  Do segmentation and reassembly

Reliable transfer (if needed): ACKs, checksums

Not overloading the receiver: “Flow control” via receiver window

Not overloading the network: “Congestion control” via sender window
UDP: "User Datagram Protocol"

Data delivery to the correct application

Demultiplexing using application identifiers (ports)

Checksums (optionally)

No-frills, simple extension of IP
This is all UDP is

<table>
<thead>
<tr>
<th>SRC port</th>
<th>DST port</th>
</tr>
</thead>
<tbody>
<tr>
<td>checksum</td>
<td>length</td>
</tr>
</tbody>
</table>

DATA
Why would anyone want UDP?

Avoids overhead and delays of ordered, reliable delivery

No delay for connection establishment

Finer control over what data is sent and when

No connection state, buffers, timers, leading to greater scalability

Small per-packet overhead, as headers are minimal

DNS  Gaming  VoIP
TCP: “Transmission Control Protocol”

Data delivery to the correct application

Demultiplexing using application identifiers (ports)

Files or byte-streams abstractions for applications

Do segmentation and reassembly

Reliable transfer (if needed): ACKs, checksums

Not overloading the receiver: “Flow control” via receiver window

Not overloading the network: “Congestion control” via sender window
In the Internet, reliability is ensured by the end hosts, not by the network.
Reliability is a L4 function, implemented 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
Reliability is a **L4 function**, implemented just above the network layer.

<table>
<thead>
<tr>
<th>Layer</th>
<th>Application</th>
<th>L4 Transport</th>
<th>L3 Network</th>
<th>Link</th>
<th>Physical</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td><strong>reliable</strong> end-to-end delivery (if needed)</td>
<td>global best-effort delivery</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Recall that L3 provides best-effort service, with no guarantees.
TCP: Reliable, in-order delivery

TCP is a connection-oriented, reliable, bytestream transport service.

What UDP provides, plus:

- Retransmission of lost and corrupted packets
- Flow control (to not overflow receiver)
- Congestion control (to not overload network)
- “Connection” set-up & tear-down
Connections (or sessions)

Reliability requires keeping state
  Sender: packets sent but not ACKed, and related timers
  Receiver: non-contiguous packets

Each bytestream is called a connection or session
  Each with their own connection state
  State is in hosts, not network!
What transport protocols do not provide

Delay and/or bandwidth guarantees
   This cannot be offered by transport
   Would require support at IP level

Sessions that survive change-of-IP-address
   This is an artifact of current implementations
   As we shall see….
Transmission Control Protocol (TCP)

Reliable, in-order delivery *(previously, but quick review)*
- Ensures byte stream (eventually) arrives intact
- In the presence of corruption and loss

Connection oriented *(today)*
- Explicit set-up and tear-down of TCP session

Full duplex stream-of-bytes service *(today)*
- Sends and receives a stream of bytes, not messages

Flow control *(previously, but quick review)*
- Ensures that sender doesn’t overwhelm receiver

Congestion control *(next lecture)*
- Dynamic adaptation to network path’s capacity
Basic Components of Reliability

ACKs
• Can’t be reliable without knowing whether data has arrived
• TCP uses byte sequence numbers to identify payloads

Checksums
• Can’t be reliable without knowing whether data is corrupted
• TCP does checksum over TCP and pseudoheader

Timeouts and retransmissions
• Can’t be reliable without retransmitting lost/corrupted data
• TCP retransmits based on timeouts and duplicate ACKs
• Timeout based on estimate of RTT
Other TCP Design Decisions

- Sliding window flow control
  - Allow $W$ contiguous bytes to be in flight

Cumulative acknowledgements
- Selective ACKs (full information) also supported (ignore)

Single timer set after each payload is ACKed
- Timer is effectively for the “next expected payload”
- When timer goes off, resend that payload and wait
  - And double timeout period

Various tricks related to “fast retransmit”
- Using duplicate ACKs to trigger retransmission
# TCP Header

<table>
<thead>
<tr>
<th>Source port</th>
<th>Destination port</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sequence number</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Acknowledgment</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Advertised window</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>HdrLen</th>
<th>Flags</th>
<th>Advertised window</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Checksum</th>
<th>Urgent pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Options (variable)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>
TCP Header

<table>
<thead>
<tr>
<th>Source port</th>
<th>Destination port</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence number</td>
<td></td>
</tr>
<tr>
<td>Acknowledgment</td>
<td></td>
</tr>
<tr>
<td>HdrLen</td>
<td>Flags</td>
</tr>
<tr>
<td>Checksum</td>
<td>Urgent pointer</td>
</tr>
<tr>
<td>Options (variable)</td>
<td></td>
</tr>
</tbody>
</table>

These should be familiar
Segments and Sequence Numbers
TCP “Stream of Bytes” Service...

Application @ Host A

Application @ Host B
... Provided Using TCP “Segments”

Transmitted when:
- Segment full (Max Segment Size)
- Not full, but times out
TCP Segment

IP packet
- No bigger than Maximum Transmission Unit (MTU)
- E.g., up to 1500 bytes with Ethernet

TCP packet
- IP packet with a TCP header and data inside
- TCP header ≥ 20 bytes long

TCP segment
- No more than Maximum Segment Size (MSS) bytes
- E.g., up to 1460 consecutive bytes from the stream
- MSS = MTU – (IP header) – (TCP header)
Sequence Numbers

ISN (initial sequence number)

\[ \text{ISN} + n \]
Sequence Numbers

ISN (initial sequence number)

ACK sequence number = next expected byte
ACKing and Sequence Numbers

Sender sends packet

- Data starts with sequence number X
- Packet contains B bytes
  - X, X+1, X+2, …, X+B-1

Upon receipt of packet, receiver sends an ACK

- If all data prior to X already received:
  - ACK acknowledges X+B (because that is next expected byte)
- If highest contiguous byte received is smaller value Y
  - ACK acknowledges Y+1
  - Even if this has been ACKed before
Normal Pattern

Sender: seqno=X, length=B
Receiver: ACK=X+B

Sender: seqno=X+B, length=B
Receiver: ACK=X+2B

Sender: seqno=X+2B, length=B
...

Seqno of next packet is same as last ACK field
### TCP Header

<table>
<thead>
<tr>
<th>Source port</th>
<th>Destination port</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Sequence number**
- **Acknowledgment**
- **HdrLen**: 0
- **Flags**
- **Advertised window**
- **Checksum**
- **Urgent pointer**
- **Options (variable)**
- **Data**

Starting **byte offset of data carried in this segment**
### TCP Header

<table>
<thead>
<tr>
<th>Source port</th>
<th>Destination port</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sequence number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acknowledgment</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>HdrLen</th>
<th>Flags</th>
<th>Advertised window</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Checksum</th>
<th>Urgent pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Options (variable)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data</th>
</tr>
</thead>
</table>
# TCP Header

<table>
<thead>
<tr>
<th>Source port</th>
<th>Destination port</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sequence number</th>
<th>Acknowledgment</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>HdrLen</th>
<th>Flags</th>
<th>Advertised window</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Checksum</th>
<th>Urgent pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Options (variable)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

Data

How much the sender can send
Sliding Window Flow Control

Advertised Window, $W$

Can send $W$ bytes beyond the next expected byte

Receiver uses $W$ to prevent sender from overflowing buffer

Limits number of bytes sender can have in flight
Filling the Pipe

Simple example:
- $W$ (in bytes), which we assume is constant
- $RTT$ (in sec), which we assume is constant
- $B$ (in $\text{bytes/sec}$)

How fast will data be transferred?

If $W/RTT < B$, the transfer has speed $W/RTT$
If $W/RTT > B$, the transfer has speed $B$
Advertised Window Limits Rate

Sender can send no faster than W/RTT bytes/sec

Receiver only advertises more space when it has consumed old arriving data

In original design, sole protocol mechanism controlling sender’s rate!

What’s missing?
Implementing Sliding Window

Both sender & receiver maintain a window
   Sender: not yet ACKed
   Receiver: not yet delivered to application

Left edge of window:
   Sender: beginning of unacknowledged data
   Receiver: beginning of undelivered data

For the sender:
   Window size = maximum amount of data in flight

For the receiver:
   Window size = maximum amount of undelivered data
Sliding Window

Allow a larger amount of data “in flight”

- Allow sender to get ahead of the receiver
- … though not too far ahead
Sliding Window

When sender receives a new ACK, send window advances (slides forward)
When sender receives a new ACK, send window advances (slides forward)
Sliding Window

When the receiving process consumes data, receive window advances.

Receiving process

- Last byte read
- Next byte needed
- Last byte received
When the receiving process consumes data, receive window advances.
Sender: window advances when new data ACKed

Receiver: window advances as receiving process consumes data

Receiver advertises to the sender where the receiver window currently ends (“righthand edge”)

- Sender agrees not to exceed this amount
- It makes sure by setting its own window size to a value that can’t send beyond the receiver’s righthand edge