Transport Protocols: TCP and UDP

→ end-to-end protocol

→ runs on top of network layer protocols

→ treat network layer & below as black box

Three-level encapsulation:

<table>
<thead>
<tr>
<th>MAC</th>
<th>IP</th>
<th>TCP/UDP</th>
<th>Payload (TCP/UDP)</th>
</tr>
</thead>
</table>

→ meaning of protocol “stack”: push/pop headers

→ common TCP payload: HTTP
Network layer (IP) assumptions:

- unreliable
- out-of-order delivery
- absence of QoS guarantees (delay, throughput, etc.)
- insecure (IPv4)
  → IPsec

Additional performance properties:

- Works “ok”
- Can break down under high load conditions
  → Atlanta Olympics
  → DoS and worm attack
- Wide behavioral range
  → sometimes good, so so, or bad
Goal of UDP (User Datagram Protocol):

→ process identification

→ port number as demux key

→ minimal support beyond IP
UDP packet format:

<table>
<thead>
<tr>
<th>Source Port</th>
<th>Destination Port</th>
</tr>
</thead>
<tbody>
<tr>
<td>Length</td>
<td>Checksum</td>
</tr>
</tbody>
</table>

Payload

Checksum calculation (pseudo header):

<table>
<thead>
<tr>
<th>Source Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Destination Address</td>
</tr>
<tr>
<td>00 ··· 0</td>
</tr>
</tbody>
</table>

→ pseudo header, UDP header and payload
UDP usage:

- multimedia streaming
  - lean and nimble
  - at minimum requires process identification
  - congestion control carried out above UDP

- stateless client/server applications
  - persistent state a hinderance
  - lightweight
Goals of TCP (Transmission Control Protocol):

• process identification

• reliable communication: ARQ

• speedy communication: congestion control

• segmentation

→ connection-oriented, i.e., stateful

→ complex mixture of functionalities
Segmentation task: provide “stream” interface to higher level protocols

→ exported semantics: contiguous byte stream

→ recall ARQ

• segment stream of bytes into blocks of fixed size

• segment size determined by TCP MTU (Maximum Transmission Unit)

• actual unit of transmission in ARQ
TCP packet format:

<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>Acknowledgement Number</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Header Length</th>
<th>U R G K A C S H P S Y T N F</th>
<th>Window Size</th>
</tr>
</thead>
<tbody>
<tr>
<td></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 (if any)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DATA (if any)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>
• Sequence Number: position of first byte of payload
• Acknowledgement: next byte of data expected (receiver)
• Header Length (4 bits): 4 B units
• URG: urgent pointer flag
• ACK: ACK packet flag
• PSH: override TCP buffering
• RST: reset connection
• SYN: establish connection
• FIN: close connection
• Window Size: receiver’s advertised window size
• Checksum: prepend pseudo-header
• Urgent Pointer: byte offset in current payload where urgent data begins
• Options: MTU; take min of sender & receiver (default 556 B)
Checksum calculation (pseudo header):

<table>
<thead>
<tr>
<th>Source Address</th>
<th>Destination Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 · · · 0</td>
<td>Protocol</td>
</tr>
<tr>
<td></td>
<td>TCP Segment Length</td>
</tr>
</tbody>
</table>

→ pseudo header, TCP header and payload
TCP connection establishment (3-way handshake):

- \( X, Y \) are chosen randomly
  \( \rightarrow \) sequence number prediction
- piggybacking
2-person consensus problem: are $A$ and $B$ in agreement about the state of affairs after 3-way handshake?

$\rightarrow$ in general: impossible

$\rightarrow$ can be proven

$\rightarrow$ “acknowledging the ACK problem”

$\rightarrow$ also TCP session ending

$\rightarrow$ lunch date problem
TCP connection termination:

- full duplex
- half duplex
More generally, finite state machine representation of TCP’s control mechanism:

→ state transition diagram
Features to notice:

- **Connection set-up:**
  - client’s transition to **ESTABLISHED** state without ACK
  - how is server to reach **ESTABLISHED** if client ACK is lost?
  - **ESTABLISHED** is macrostate (partial diagram)

- **Connection tear-down:**
  - three normal cases
  - special issue with **TIME WAIT** state
  - employs hack
Basic TCP data transfer:

A

Seq = 0

Ack = 1024, Win = 1024

Seq = 1024

Ack = 2048, Win = 0

Timer Expires; Retransmit

Seq = 1024

Ack = 2048, Win = 0

Ack = 2048, Win = 1024

Seq = 2048

B

2K

1K

0K

0K

1K

0K
TCP's sliding window protocol

- sender, receiver maintain buffers MaxSendBuffer, MaxRcvBuffer
Note asynchrony between TCP module and application.

Sender side: maintain invariants

- \( \text{LastByteAcked} \leq \text{LastByteSent} \leq \text{LastByteWritten} \)
- \( \text{LastByteWritten} - \text{LastByteAcked} < \text{MaxSendBuffer} \)

\[ \rightarrow \text{ buffer flushing (advance window)} \]
\[ \rightarrow \text{ application blocking} \]

- \( \text{LastByteSent} - \text{LastByteAcked} \leq \text{AdvertisedWindow} \)

Thus,

\[ \text{EffectiveWindow} = \text{AdvertisedWindow} - (\text{LastByteSent} - \text{LastByteAcked}) \]

\[ \rightarrow \text{ upper bound on new send volume} \]
Actually, one additional refinement:

\[ \rightarrow \text{CongestionWindow} \]

**EffectiveWindow** update procedure:

\[
\text{EffectiveWindow} = \text{MaxWindow} - (\text{LastByteSent} - \text{LastByteAcked})
\]

where

\[
\text{MaxWindow} = \min\{ \text{AdvertisedWindow}, \text{CongestionWindow} \}
\]

How to set **CongestionWindow**.

\[ \rightarrow \text{domain of TCP congestion control} \]
Receiver side: maintain invariants

- $\text{LastByteRead} < \text{NextByteExpected} \leq \text{LastByteRcvd} + 1$
- $\text{LastByteRcvd} - \text{NextByteRead} < \text{MaxRcvBuffer}$

  $\rightarrow$ buffer flushing (advance window)
  $\rightarrow$ application blocking

Thus,

$$\text{AdvertisedWindow} = \text{MaxRcvBuffer} - (\text{LastByteRcvd} - \text{LastByteRead})$$
Issues:

How to let sender know of change in receiver window size after $\text{AdvertisedWindow}$ becomes 0?

- trigger ACK event on receiver side when $\text{AdvertisedWindow}$ becomes positive
- sender periodically sends 1-byte probing packet

$\rightarrow$ design choice: smart sender/dumb receiver

$\rightarrow$ same situation for congestion control
Silly window syndrome: Assuming receiver buffer is full, what if application reads one byte at a time with long pauses?

- can cause excessive 1-byte traffic

- if $\text{AdvertisedWindow} < \text{MSS}$ then set

  $\text{AdvertisedWindow} \leftarrow 0$
Do not want to send too many 1 B payload packets.

Nagle’s algorithm:

• rule: connection can have only one such unacknowledged packet outstanding

• while waiting for ACK, incoming bytes are accumulated (i.e., buffered)

... compromise between real-time constraints and efficiency.

→ useful for telnet-type applications
Sequence number wrap-around problem: recall sufficient condition

\[ \text{SenderWindowSize} < \frac{\text{MaxSeqNum} + 1}{2} \]

\[ \rightarrow \quad \text{32-bit sequence space/16-bit window space} \]

However, more importantly, time until wrap-around important due to possibility of roaming packets.

<table>
<thead>
<tr>
<th>bandwidth</th>
<th>time until wrap-around (\uparrow)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1 (1.5 Mbps)</td>
<td>6.4 hrs</td>
</tr>
<tr>
<td>Ethernet (10 Mbps)</td>
<td>57 min</td>
</tr>
<tr>
<td>T3 (45 Mbps)</td>
<td>13 min</td>
</tr>
<tr>
<td>F/E (100 Mbps)</td>
<td>6 min</td>
</tr>
<tr>
<td>OC-3 (155 Mbps)</td>
<td>4 min</td>
</tr>
<tr>
<td>OC-12 (622 Mbps)</td>
<td>55 sec</td>
</tr>
<tr>
<td>OC-24 (1.2 Gbps)</td>
<td>28 sec</td>
</tr>
</tbody>
</table>
**RTT estimation**

... important to not underestimate nor overestimate.

Karn/Partridge: Maintain running average with precautions

\[
\text{EstimateRTT} \leftarrow \alpha \cdot \text{EstimateRTT} + \beta \cdot \text{SampleRTT}
\]

- SampleRTT computed by sender using timer
- \( \alpha + \beta = 1; \ 0.8 \leq \alpha \leq 0.9, \ 0.1 \leq \beta \leq 0.2 \)
- TimeOut \(\leftarrow 2 \cdot \text{EstimateRTT} \) or
  \[
  \text{TimeOut} \leftarrow 2 \cdot \text{TimeOut} \quad \text{(if retransmit)}
  \]

\[\rightarrow\] need to be careful when taking SampleRTT
\[\rightarrow\] infusion of complexity
\[\rightarrow\] still remaining problems
Hypothetical RTT distribution:

$\rightarrow$ need to account for variance

$\rightarrow$ not nearly as nice
Jacobson/Karels:

- Difference = SampleRTT − EstimatedRTT
- EstimatedRTT = EstimatedRTT + δ · Difference
- Deviation = Deviation + δ(|Difference| − Deviation)

Here 0 < δ < 1.

Finally,

- TimeOut = μ · EstimatedRTT + φ · Deviation

where μ = 1, φ = 4.

→ persistence timer

→ how to keep multiple timers in UNIX