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>

- Sequence Number
- Acknowledgement Number
- Window Size
- Checksum
- Urgent Pointer
- Options (if any)
- DATA (if any)
• 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>
<th>Protocol</th>
<th>TCP Segment Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></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?

\[ \rightarrow \text{ in general: impossible} \]

\[ \rightarrow \text{ can be proven} \]

\[ \rightarrow \text{ “acknowledging the ACK problem”} \]

\[ \rightarrow \text{ also TCP session ending} \]

\[ \rightarrow \text{ lunch date problem} \]
Call Collision:

\[
\begin{align*}
\text{A} & \quad \text{B} \\
\text{SYN} = 1, \quad \text{Seq. No.} = X \\
\text{SYN} = 1, \quad \text{Seq. No.} = Y \\
\text{SYN} = 1, \quad \text{Seq. No.} = Y \\
\text{Ack. No.} = X + 1 \\
\text{SYN} = 1, \quad \text{Seq. No.} = X \\
\text{Ack. No.} = Y + 1 \\
\end{align*}
\]

\[\rightarrow\text{ only single TCB gets allocated}\]

\[\rightarrow\text{ unique full association}\]
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 \texttt{ESTABLISHED} state without ACK
  - how is server to reach \texttt{ESTABLISHED} if client ACK is lost?
  - \texttt{ESTABLISHED} is macrostate (partial diagram)

- Connection tear-down:
  - three normal cases
  - special issue with \texttt{TIME WAIT} state
  - employs hack
Basic TCP data transfer:
TCP’s sliding window protocol

- sender, receiver maintain buffers $\text{MaxSendBuffer}$, $\text{MaxRcvBuffer}$
Note asynchrony between TCP module and application.

Sender side: maintain invariants

- $\text{LastByteAcked} \leq \text{LastByteSent} \leq \text{LastByteWritten}$
- $\text{LastByteWritten} - \text{LastByteAcked} < \text{MaxSendBuffer}$

$\rightarrow$ buffer flushing (advance window)

$\rightarrow$ application blocking

- $\text{LastByteSent} - \text{LastByteAcked} \leq \text{AdvertisedWindow}$

Thus,

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

$\rightarrow$ 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

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

How to set **CongestionWindow**.

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

- \( \text{LastByteRead} < \text{NextByteExpected} \leq \text{LastByteRcvd} + 1 \)
- \( \text{LastByteRcvd} - \text{NextByteRead} < \text{MaxRcvBuffer} \)
  \[ \rightarrow \text{ buffer flushing (advance window)} \]
  \[ \rightarrow \text{ application blocking} \]

Thus,

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

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

- trigger ACK event on receiver side when $AdvertisedWindow$ 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

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

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

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 ¹</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>
Even more importantly, “keeping-the-pipe-full” consideration.

<table>
<thead>
<tr>
<th>Bandwidth</th>
<th>Delay-bandwidth product</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1 (1.5 Mbps)</td>
<td>18 kB</td>
</tr>
<tr>
<td>Ethernet (10 Mbps)</td>
<td>122 kB</td>
</tr>
<tr>
<td>T3 (45 Mbps)</td>
<td>549 kB</td>
</tr>
<tr>
<td>FDDI (100 Mbps)</td>
<td>1.2 MB</td>
</tr>
<tr>
<td>OC-3 (155 Mbps)</td>
<td>1.8 MB</td>
</tr>
<tr>
<td>OC-12 (622 Mbps)</td>
<td>7.4 MB</td>
</tr>
<tr>
<td>OC-24 (1.2 Gbps)</td>
<td>14.8 MB</td>
</tr>
</tbody>
</table>

→ 100 ms latency

Also, throughput limitation imposed by TCP receiver window size.

→ e.g., high-performance grid apps
RTT estimation

... important to not underestimate nor overestimate.

Karn/Partridge: Maintain running average with precautions

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

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

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

→ need to account for variance

→ not nearly as nice
Jacobson/Karels:

- $\text{Difference} = \text{SampleRTT} - \text{EstimatedRTT}$
- $\text{EstimatedRTT} = \text{EstimatedRTT} + \delta \cdot \text{Difference}$
- $\text{Deviation} = \text{Deviation} + \delta (|\text{Difference}| - \text{Deviation})$

Here $0 < \delta < 1$.

Finally,

- $\text{TimeOut} = \mu \cdot \text{EstimatedRTT} + \phi \cdot \text{Deviation}$

where $\mu = 1$, $\phi = 4$.

$\rightarrow$ persistence timer

$\rightarrow$ how to keep multiple timers in UNIX