Connection-oriented transport
TCP in details

TCP: Overview  RFCs:
793,1122,1323, 2018, 2581

• Point-to-point:
  – one sender, one receiver
• Reliable, in-order byte steam:
  – no “message boundaries”
• Pipelined:
  – TCP congestion and flow control set window size
• Pull duplex data:
  – bi-directional data flow in same connection
  – MSS: maximum segment size
• Connection-oriented:
  – handshaking (exchange of control msgs) inits
    sender, receiver state before data exchange
• Flow controlled:
  – sender will not overwhelm receiver
Once through the door ...

TCP directs data to the connection's send buffer.

Periodically TCP "grabs" chunks of data and ships it through the network layer portal*

*TCP specifications are very laid back about when this should happen stating:
"Send that data in segments at its own convenience."

The man behind the curtain

- TCP pairs each chunk of client data* with a TCP header, forming a TCP segment.

- Segments are passed down the network layer, where they are separately encapsulated within network-layer IP datagrams and sent out over the network.

*Maximum Segment Size (MSS) is determined by OS. Common values are 1,500, 536, and 512 bytes.
Old friends

We have already discussed source and destination port numbers ...

... as well as the Internet checksum

Chunk of data being sent

More TCP fields

Specifies the length of TCP header in 32-bit words (4 bits)

TCP header length is variable due to option field. Typically, options field is empty, so length is usually 20 bytes.
Two rarely used bit fields

*Sequence, acknowledgement, receive window, and the remaining bit fields are coming attractions.

Sequence and acknowledgment numbers

- The sequence number for a segment is a byte-stream number of the first byte in the segment.
- The acknowledgement number that host A puts in its segment is the sequence number of the next byte host A is expecting from host B.
- TCP only acknowledges bytes up to the first missing byte in the stream (cumulative acknowledgement).

<table>
<thead>
<tr>
<th>File</th>
<th>Data for 1st segment</th>
<th>Data for 2nd segment</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1,000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1,999</td>
</tr>
<tr>
<td></td>
<td></td>
<td>499,999</td>
</tr>
</tbody>
</table>
Host B sends A a packet*

We assume the initial sequence number was zero. In truth, both sides of a TCP connection randomly choose an initial sequence number.

*Host A having already sent bytes 0 through 4999 of its data to host B.

Host A sends B next 1000 bytes + ACK

*Host A having already sent bytes 0 through 4999 of its data to host B.
Host B sends A next 1000 bytes & ACK

<table>
<thead>
<tr>
<th>Seq #</th>
<th>Ack #</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>6000</td>
</tr>
</tbody>
</table>

Bytes 1000 -- 1999 of Host B’s file

B’s packet 1000 -- 1999 is lost

<table>
<thead>
<tr>
<th>Seq #</th>
<th>Ack #</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000</td>
<td>6000</td>
</tr>
</tbody>
</table>

Bytes 2000 -- 2999 of Host B’s file

B’s packet 1000 -- 1999 is lost

<table>
<thead>
<tr>
<th>Seq #</th>
<th>Ack #</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>6000</td>
</tr>
</tbody>
</table>

Bytes 1000 -- 1999 of Host B’s file

<table>
<thead>
<tr>
<th>Seq #</th>
<th>Ack #</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000</td>
<td>6000</td>
</tr>
</tbody>
</table>

Bytes 2000 -- 2999 of Host B’s file
Host A responds ACK 1000

Cumulative Ack: TCP only acknowledges bytes up to the first missing byte

Host B will resend missing packet
What does A do with out-of-order packets?

TCP RFCs are mum on this

Two Choices:

1. The receiver immediately discards out-of-order segments;

2. The receiver keeps the out-of-order bytes and waits for the missing bytes to fill in the gaps.
Telnet application over TCP

A segment with a sequence number, but no data; hum ...

Acknowledgement for client-to-server is "piggybacked" in a segment carrying server-to-client data

Timeout

- Like our rdt protocol, TCP uses a timeout/retransmit mechanism to recover from lost segments.

- Timeout should be larger than the connection’s RTT

- But,
  - how much larger?
  - how does TCP know the RRT in the first place.
**EstimatedRTT: An exponential weighted moving average**

- TCP takes one `SampleRTT` measurement at a time

\[
\text{EstimatedRTT} = (1 - \alpha) \cdot \text{EstimatedRTT} + \alpha \cdot \text{SampleRTT}
\]

Recommended alpha is 1/8

SampleRTT only measured for segments that have been transmitted once

The average puts more weight on recent samples than on older samples

Weight given to past SampleRTT decay exponentially over time
Should we use EstimatedRTT as our timeout value?

DevRTT

- DevRTT estimates of how much SampleRTT bounces around

\[
\text{DevRTT} = (1-\beta)\cdot\text{DevRTT} + \beta\cdot|\text{SampleRTT} - \text{EstimatedRTT}|
\]

Recommended beta is 1/4
Setting and managing timeout interval

• TCP’s method for determining the retransmission timeout interval is

\[
\text{TimeoutInterval} = \text{EstimatedRTT} + 4\times\text{DevRTT}
\]

• When a timeout event occurs, TCP retransmits the not yet ACKed segment with the smallest sequence number and sets the timeout interval to twice the previous value.
TCP sender events

**data rcvd from app:**
- create segment with seq #
- seq # is byte-stream number of first data byte in segment
- start timer if not already running
  - think of timer as for oldest unacked segment
  - expiration interval: TimeOutInterval

**timeout:**
- retransmit segment that caused timeout
- restart timer

**ack rcvd:**
- if ack acknowledges previously unacked segments
  - update what is known to be ACKed
  - start timer if there are still unacked segments

TCP sender (simplified)

Data received from application above
create segment, seq. #: NextSeqNum
pass segment to IP (i.e., “send”)
NextSeqNum = NextSeqNum + length(data)
if (timer currently not running)
start timer

ACK received, with ACK field value y
if (y > SendBase) {
  SendBase = y
  /* SendBase-1: last cumulatively ACKed byte */
  if (there are currently not-yet-acked segments)
    start timer
  else stop timer
}
TCP: retransmission scenarios

lost ACK scenario

Host A
Seq=92, 8 bytes of data
ACK=100

Host B
Seq=92, 8 bytes of data

timeout

ACK=100

TCP: retransmission scenarios

premature timeout

Host A
SendBase=92

Host B
Seq=92, 8 bytes of data
ACK=100

SendBase=100

SendBase=120

timeout

ACK=120

SendBase=120

Seq=100, 20 bytes of data

SendBase=120

ACK=120
## TCP: retransmission scenarios

### TCP Receiver action

<table>
<thead>
<tr>
<th>Event at Receiver</th>
<th>TCP Receiver action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed</td>
<td>Delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK</td>
</tr>
<tr>
<td>Arrival of in-order segment with expected seq #. One other segment has ACK pending</td>
<td>Immediately send single cumulative ACK, ACKing both in-order segments</td>
</tr>
<tr>
<td>Arrival of out-of-order segment higher-than-expect seq #. Gap detected</td>
<td>Immediately send duplicate ACK, indicating seq. # of next expected byte</td>
</tr>
<tr>
<td>Arrival of segment that partially or completely fills gap</td>
<td>Immediate send ACK, provided that segment starts at lower end of gap</td>
</tr>
</tbody>
</table>

---

### Event at Receiver

- Arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed
- Arrival of in-order segment with expected seq #. One other segment has ACK pending
- Arrival of out-of-order segment higher-than-expect seq #. Gap detected
- Arrival of segment that partially or completely fills gap
TCP fast retransmit

- Time-out period often relatively long:
  - long delay before resending lost packet
- Detect lost segments via duplicate ACKs.
  - sender often sends many segments back-to-back
  - if segment is lost, there will likely be many duplicate ACKs.

*TCP fast retransmit*

If sender receives 3 ACKs for same data ("triple duplicate ACKs"), resend unacked segment with smallest seq #
- likely that unacked segment lost, so don’t wait for timeout

---

Host A

<table>
<thead>
<tr>
<th>Seq=92, 8 bytes of data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq=100, 20 bytes of data</td>
</tr>
</tbody>
</table>

**Host B**

<table>
<thead>
<tr>
<th>ACK=100</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACK=100</td>
</tr>
<tr>
<td>ACK=100</td>
</tr>
<tr>
<td>ACK=100</td>
</tr>
<tr>
<td>Seq=100, 20 bytes of data</td>
</tr>
</tbody>
</table>

fast retransmit after sender receipt of triple duplicate ACK
Fast retransmit algorithm

**event:** ACK received, with ACK field value of \( y \)

if \( y > \text{SendBase} \) {
    \text{SendBase} = y
    if (there are currently not-yet-acknowledged segments)
        start timer
} else {
    increment count of dup ACKs received for \( y \)
    if (count of dup ACKs received for \( y \) is 3) {
        resend segment with sequence number \( y \)
    }
}

TCP
Flow Control
TCP flow control

- Receiver informs sender of spare room
  - Recall TCP is full-duplex, so while “sender” transmits data to “receiver”, the latter is posting its vacancy rate, RcvWindow, to “sender” in every segment.
  - The sending host makes sure LastByteSent - LastByteAcked ≤ RcvWindow.

(receiver controls sender, so sender won’t overflow receiver’s buffer by transmitting too much, too fast)
Avoiding gridlock

- There is one minor technical problem with this scheme: Suppose the the B’s receive buffer fills, and $RcvWindow = 0$ is sent to “sender” A.

- Suppose further, that the B has no additional data to the “sender”.*

- So what’s the problem?

Problem …

- TCP sends a segment to the sender only if
  - it has data to send, or
  - it has an acknowledgement to send
- The sender is never informed when space becomes available in the receive buffer.

Solution

- The solution is TCP specs require host A to continue to send segments with one data byte when B’s receive window is zero.

- These segments will be acknowledge by the receiver.

- Eventually the buffer will begin to empty and the acknowledgements will contain a nonzero RcvWindow value.
TCP

Connection management

Connection Management

Before exchanging data, sender/receiver “handshake”:
- agree to establish connection (each knowing the other willing to establish connection)
- agree on connection parameters

Socket clientSocket = newSocket("hostname","port number");

Socket connectionSocket = welcomeSocket.accept();
Agreeing to establish a connection

2-way handshake:

Q: Will 2-way handshake always work in network?

- variable delays
- retransmitted messages (e.g. req_conn(x)) due to message loss
- message reordering
- can’t “see” other side

2-way handshake failure scenarios:

- half open connection! (no client!)
- client terminates
- server forgets x

- retransmit req_conn(x)
- accept data(x+1)
- connection x completes
- server forgets x

- choose x
- req_conn(x)
- ESTAB
- acc_conn(x)
- ESTAB

- choose x
- req_conn(x)
- ESTAB
- acc_conn(x)
- ESTAB
TCP 3-way handshake

1. Client requests TCP connection: Hi

Client host sends TCP SYN segment to server specifying initial sequence number ...

...and setting the SYN bit, but sends no application-layer data.
2. The server replies: Hello!

Server host receives SYN, allocates the TCP buffers and variables to the client TCP, and replies with SYNACK by choosing its initial sequence number, setting its Acknowledgment field to the client_isn+1, and setting the ACK and SYN bits (still no data).

3. Client responds: Can we talk?

Client host receives ACKSYN, allocates its own TCP buffers and variables to the server TCP, replies with SYN bit set to zero since connection is established, and may include data.
TCP 3-way handshake: FSM

TCP: closing a connection

- Client, server each close their side of connection
  - send TCP segment with FIN bit = 1

- Respond to received FIN with ACK
  - on receiving FIN, ACK can be combined with own FIN

- Simultaneous FIN exchanges can be handled
TCP: closing a connection

**client state**

- **ESTAB**: clientSocket.close() can no longer send but can receive data
- **FIN_WAIT_1**: wait for server close
- **FIN_WAIT_2**: timed wait for 2*max segment lifetime
- **TIMED_WAIT**: can no longer send but can receive data
- **CLOSED**: timed wait for 2*max segment lifetime

**server state**

- **ESTAB**: can still send data
- **CLOSE_WAIT**: can no longer send data
- **LAST_ACK**: can no longer send data
- **CLOSED**: