Chapter 3: roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- Principles of congestion control
- TCP congestion control
TCP: overview  RFCs: 793, 1122, 2018, 5681, 7323

- **point-to-point:**
  - one sender, one receiver

- **reliable, in-order byte steam:**
  - no “message boundaries"

- **full duplex data:**
  - bi-directional data flow in same connection
  - MSS: maximum segment size

- **cumulative ACKs**

- **pipelining:**
  - TCP congestion and flow control
  - set window size

- **connection-oriented:**
  - handshaking (exchange of control messages) initializes sender, receiver state before data exchange

- **flow controlled:**
  - sender will not overwhelm receiver
TCP segment structure

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>source port #</td>
<td>Source port number</td>
</tr>
<tr>
<td>dest port #</td>
<td>Destination port number</td>
</tr>
<tr>
<td>sequence number</td>
<td>Sequence number</td>
</tr>
<tr>
<td>acknowledgement number</td>
<td>Acknowledgment number</td>
</tr>
<tr>
<td>receive window</td>
<td>Receive window</td>
</tr>
<tr>
<td>checksum</td>
<td>Internet checksum</td>
</tr>
<tr>
<td>options (variable length)</td>
<td>TCP options (variable length)</td>
</tr>
<tr>
<td>application data</td>
<td>Application data (variable length)</td>
</tr>
<tr>
<td>Urg data pointer</td>
<td>Urgent data pointer</td>
</tr>
<tr>
<td>data sent by application into TCP socket</td>
<td>Data sent by application into TCP socket</td>
</tr>
</tbody>
</table>

**TCP options:**
- C, E: congestion notification
- RST, SYN, FIN: connection management

**Flow control:** # bytes receiver willing to accept

**ACK:** seq # of next expected byte; A bit: this is an ACK

**Segment seq #:** counting bytes of data into bytestream (not segments!)
TCP sequence numbers, ACKs

**Sequence numbers:**
- byte stream “number” of first byte in segment’s data

**Acknowledgements:**
- seq # of next byte expected from other side
- cumulative ACK

**Q:** how receiver handles out-of-order segments
- **A:** TCP spec doesn’t say, - up to implementor
TCP sequence numbers, ACKs

User types ‘C’

host ACKs receipt of echoed ‘C’

Host A

Seq=42, ACK=79, data = ‘C’

Seq=79, ACK=43, data = ‘C’

Seq=43, ACK=80

Host B

host ACKs receipt of ‘C’, echoes back ‘C’

simple telnet scenario
TCP round trip time, timeout

**Q:** how to set TCP timeout value?
- longer than RTT, but RTT varies!
- **too short:** premature timeout, unnecessary retransmissions
- **too long:** slow reaction to segment loss

**Q:** how to estimate RTT?
- **SampleRTT:** measured time from segment transmission until ACK receipt
  - ignore retransmissions
- **SampleRTT** will vary, want estimated RTT “smoother”
  - average several recent measurements, not just current SampleRTT
TCP round trip time, timeout

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

- **exponential weighted moving average (EWMA)**
- influence of past sample decreases exponentially fast
- typical value: \(\alpha = 0.125\)
TCP round trip time, timeout

- **Timeout interval:** $\text{EstimatedRTT}$ plus “safety margin”
  - large variation in $\text{EstimatedRTT}$: want a larger safety margin

  $$\text{TimeoutInterval} = \text{EstimatedRTT} + 4\times\text{DevRTT}$$

- **DevRTT:** EWMA of $\text{SampleRTT}$ deviation from $\text{EstimatedRTT}$:

  $$\text{DevRTT} = (1-\beta)\times\text{DevRTT} + \beta\times|\text{SampleRTT}-\text{EstimatedRTT}|$$

  (typically, $\beta = 0.25$)

* Check out the online interactive exercises for more examples: [http://gaia.cs.umass.edu/kurose_ross/interactive/](http://gaia.cs.umass.edu/kurose_ross/interactive/)
TCP Sender (simplified)

event: data received from application
- 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

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

event: ACK received
- if ACK acknowledges previously unACKed segments
  - update what is known to be ACKed
  - start timer if there are still unACKed segments
## TCP Receiver: ACK generation [RFC 5681]

<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 immediately send single cumulative ACK, ACKing both in-order segments immediately send duplicate ACK, indicating seq. # of next expected byte immediate send ACK, provided that segment starts at lower end of gap</td>
</tr>
</tbody>
</table>
TCP: retransmission scenarios

**Lost ACK scenario**
- Host A: Seq=92, 8 bytes of data
- Host B: ACK=100
- Timeout
- Host A: Seq=92, 8 bytes of data
- Host B: ACK=100

**Premature timeout**
- Host A: Seq=92, 8 bytes of data
- Host B: ACK=100
- Timeout
- Host A: Seq=100, 20 bytes of data
- Host B: SendBase=120
- Host B: SendBase=120
- Host B: SendBase=92
- Host A: SendBase=92
- Host B: send cumulative ACK for 120

Transport Layer: 3-11
TCP: retransmission scenarios

Host A

Seq=92, 8 bytes of data

Seq=100, 20 bytes of data

ACK=100

ACK=120

Host B

Seq=120, 15 bytes of data

cumulative ACK covers for earlier lost ACK

Transport Layer: 3-12
TCP fast retransmit

if sender receives 3 additional ACKs for same data ("triple duplicate ACKs"), resend unACKed segment with smallest seq #

- likely that unACKed segment lost, so don’t wait for timeout

Receipt of three duplicate ACKs indicates 3 segments received after a missing segment – lost segment is likely. So retransmit!
Chapter 3: roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer

**Connection-oriented transport: TCP**
- segment structure
- reliable data transfer
- flow control
- connection management

- Principles of congestion control
- TCP congestion control
Q: What happens if network layer delivers data faster than application layer removes data from socket buffers?
TCP flow control

**Q:** What happens if network layer delivers data faster than application layer removes data from socket buffers?

![Diagram showing TCP flow control](image-url)
Q: What happens if network layer delivers data faster than application layer removes data from socket buffers?
Q: What happens if network layer delivers data faster than application layer removes data from socket buffers?

Flow control
receiver controls sender, so sender won’t overflow receiver’s buffer by transmitting too much, too fast
TCP flow control

- TCP receiver “advertises” free buffer space in `rwnd` field in TCP header
  - `RcvBuffer` size set via socket options (typical default is 4096 bytes)
  - many operating systems autoadjust `RcvBuffer`

- sender limits amount of unACKed (“in-flight”) data to received `rwnd`

- guarantees receive buffer will not overflow
TCP flow control

- TCP receiver “advertises” free buffer space in rwnd field in TCP header
  - RcvBuffer size set via socket options (typical default is 4096 bytes)
  - many operating systems autoadjust RcvBuffer

- sender limits amount of unACKed (“in-flight”) data to received rwnd
- guarantees receive buffer will not overflow
TCP connection management

before exchanging data, sender/receiver “handshake”:

- agree to establish connection (each knowing the other willing to establish connection)
- agree on connection parameters (e.g., starting seq #s)

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 scenarios

choose x

req_conn(x) → ESTAB

acc_conn(x)

data(x+1) → accept
data(x+1) → ACK(x+1)

connection x completes

No problem!
2-way handshake scenarios

Problem: half open connection! (no client)
2-way handshake scenarios

Problem: dup data accepted!
TCP 3-way handshake

Client state

- clientSocket = socket(AF_INET, SOCK_STREAM)
- clientSocket.connect((serverName, serverPort))

SYNSENT

- choose init seq num, x
- send TCP SYN msg

ESTAB

- received SYNACK(x) indicates server is live;
- send ACK for SYNACK;
- this segment may contain client-to-server data

SYNRCVD

- SYNbit=1, Seq=y
- ACKbit=1; ACKnum=x+1

Received ACK(y) indicates client is live

Server state

- serverSocket = socket(AF_INET, SOCK_STREAM)
- serverSocket.bind(('', serverPort))
- serverSocket.listen(1)
- connectionSocket, addr = serverSocket.accept()

LISTEN

- choose init seq num, y
- send TCP SYNACK msg, acking SYN

SYNSENT

- SYNbit=1, Seq=x

ESTAB

- ACKbit=1, ACKnum=y+1

Received ACK(y) indicates client is live

LISTEN

- SYNRCVD

ESTAB

clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect((serverName, serverPort))

serverSocket = socket(AF_INET, SOCK_STREAM)
serverSocket.bind(('', serverPort))
serverSocket.listen(1)
connectionSocket, addr = serverSocket.accept()
A human 3-way handshake protocol

1. On belay?
2. Belay on.
3. Climbing.
Closing a TCP 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