TCP: The Transmission Control Protocol

Prof. Matthias Grossglauser

School of Computer and Communication Sciences
EPFL

http://lcawww.epfl.ch
Objectives

- Connection-oriented, reliable transport: Transmission Control Protocol (TCP)
  - Segment structure
  - Reliable data transfer, based on principles seen last time
  - Connection management
- Congestion control
  - Principles
  - TCP
TCP: Overview

- **RFCs:**
  - 793, 1122, 1323, 2018, 2581
- **Point-to-point:**
  - One sender, one receiver
- **Reliable, in-order byte steam:**
  - No “message boundaries”
- **Full duplex:**
  - Bi-directional data flow in same connection
- **Connection-oriented:**
  - Handshaking (exchange of control msgs) init’s sender, receiver state before data exchange
- **Flow controlled:**
  - Sender will not overwhelm receiver
- **Pipelined:**
  - TCP congestion and flow control set window size
- **Send & receive buffers**
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 of bytes sent or acknowledged</td>
</tr>
<tr>
<td>acknowledgement number</td>
<td>Acknowledgement number of bytes acknowledged</td>
</tr>
<tr>
<td>head len</td>
<td>Length of header in bytes</td>
</tr>
<tr>
<td>not used</td>
<td>Not used</td>
</tr>
<tr>
<td>UAPR</td>
<td>Flags: URG, ACK, PSH, RST, SYN, FIN</td>
</tr>
<tr>
<td>RSF</td>
<td>Reserved for future use</td>
</tr>
<tr>
<td>Receive window</td>
<td>Size of window to receive more data</td>
</tr>
<tr>
<td>Urg data pnter</td>
<td>Pointer to urgent data</td>
</tr>
<tr>
<td>Options (variable length)</td>
<td>Options for variable length</td>
</tr>
<tr>
<td>application data</td>
<td>Application data (variable length)</td>
</tr>
</tbody>
</table>

**URG**: urgent data (generally not used)

**ACK**: ACK # valid

**PSH**: push data now (generally not used)

**RST, SYN, FIN**: connection estab (setup, teardown commands)

**Internet checksum** (as in UDP)

**Receive window** counting by bytes of data (not segments!)

**Options** # bytes rcvr willing to accept
TCP Sequence and Acknowledgment Numbers

**Seq. #’s:**
- byte stream “number” of first byte in segment’s data

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

**Q:** how does receiver handle out-of-order segments
- A: TCP spec doesn’t say, - up to implementer

---

Host A

User types ‘C’

Seq=42, ACK=79, data = ‘C’

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

Seq=79, ACK=43, data = ‘C’

host ACKs receipt of echoed ‘C’

Seq=43, ACK=80

simple telnet scenario
TCP Round Trip Time (RTT) and 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 and Timeout

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

- Exponential weighted moving average
- Influence of past sample decreases exponentially fast
- Typical value: \( \alpha = 0.125 \)
Example RTT Estimation

![Graph showing RTT measurements over time. The y-axis represents RTT in milliseconds (RTT (milliseconds)), and the x-axis represents time in seconds (Time (seconds)). The graph shows a series of measurements with peaks and valleys, indicating variability in RTT. There are annotations for Sample RTT and Estimated RTT.]
TCP Round Trip Time and Timeout

**Setting the timeout**

- **EstimatedRTT** plus “safety margin”
  - large variation in EstimatedRTT \( \rightarrow \) larger safety margin
- first estimate of how much SampleRTT deviates from EstimatedRTT:

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

  (typically, \( \beta = 0.25 \))

Then set timeout interval:

\[
\text{TimeoutInterval} = \text{EstimatedRTT} + 4 \times \text{DevRTT}
\]
TCP Reliable Data Transfer

- TCP creates reliable service on top of IP’s unreliable service

- Pipelined segments
  - One segment encapsulated into one IP packet
  - MSS: Maximum segment size, approx. 1500 bytes

- Cumulative acks

- TCP uses single retransmission timer

- Retransmissions are triggered by:
  - timeout events
  - duplicate acks

- Initially consider simplified TCP sender:
  - ignore duplicate acks
  - ignore flow control, congestion control
TCP Sender Events:

- **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

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

- **Ack received:**
  - If acknowledges previously unacked segments
    - update what is known to be acked
    - start timer if there are outstanding segments
TCP Sender (Simplified)

NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

loop (forever) {
    switch(event)

    event: data received from application above
        create TCP segment with sequence number NextSeqNum
        if (timer currently not running)
            start timer
        pass segment to IP
        NextSeqNum = NextSeqNum + length(data)

    event: timer timeout
        retransmit not-yet-acknowledged segment with
            smallest sequence number
        start timer

    event: ACK received, with ACK field value of y
        if (y > SendBase) {
            SendBase = y
            if (there are currently not-yet-acknowledged segments)
                start timer
        }
}

/* end of loop forever */

Comment:
• SendBase-1: last cumulatively ack’ed byte
Example:
• SendBase-1 = 71; y= 73, so the rcvr wants 73+ ;
y > SendBase, so that new data is acked
TCP: Retransmission Scenarios

Host A

Seq=100, 20 bytes data
ACK=100

Host B

Seq=92, 8 bytes data
ACK=120

Seq=92, 8 bytes data
ACK=120

Seq=92 timeout

lost ACK scenario

SendBase = 100

SendBase = 120

SendBase = 100

SendBase = 120

SendBase = 100

SendBase = 120

premature timeout
TCP Retransmission Scenarios (more)

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

Host B
X
Seq=100, 20 bytes data
ACK=100

SendBase = 120

timeout

loss

Cumulative ACK scenario

SendBase = 120

Cumulative ACK scenario

time

loss

timeout
### TCP ACK generation [RFC 1122, RFC 2581]

<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>
Fast Retransmit

- Time-out period often relatively long:
  - important to be conservative -> RTT estimation overestimates
  - 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

- If sender receives 3 ACKs for the same data, it supposes that segment after ACKed data was lost:
  - “ACK(i),ACK(i),ACK(i)”: probably segment (i+1) was lost, afterwards at least two further segments (i+2,i+3) received and ackd by receiver
  - Fast retransmit: resend segment before timer expires
Fast Retransmit Algorithm

event: ACK received, with ACK field value of y:
  if (y > SendBase) {
    SendBase = y
    if (there are currently not-yet-ackd segments)
      start timer
  }
else {
  increment count of dup ACKs received for y
  if (count of dup ACKs received for y = 3) {
    resend segment with sequence number y
  }
}

a duplicate ACK for already ACKed segment

fast retransmit
TCP Connection Management

**Recall:** TCP sender, receiver establish “connection” before exchanging data segments
- initialize TCP variables:
  - sequence #s
  - buffers, flow control info
- **client:** connection initiator
  ```java
  Socket clientSocket = new Socket("hostname","port number");
  ```
- **server:** contacted by client
  ```java
  Socket connectionSocket = welcomeSocket.accept();
  ```

**Three way handshake:**

**Step 1:** client host sends TCP SYN segment to server
- specifies initial seq #
- no data

**Step 2:** server host receives SYN, replies with SYNACK segment
- server allocates buffers
- specifies server initial seq. #

**Step 3:** client receives SYNACK, replies with ACK segment, which may contain data
Three-Way Handshake for Connection Setup

Client host

Connection request

SYN=1, seq=client_isn

ACK

Server host

SYN=1, seq=server_isn, ack=client_isn+1

SYN-ACK

 SYN=0, seq=client_isn+1, ack=server_isn+1

ACK

Connection granted

Time

Time
Connection Teardown

Closing a connection:

- client closes socket: clientSocket.close();
- Step 1: client -> FIN -> server
- Step 2: server receives FIN, replies with ACK. Closes connection, sends FIN.
- Step 3: client receives FIN, replies with ACK.
  - Enters “timed wait” - will respond with ACK to received FINs
- Step 4: server, receives ACK. Connection closed.
- Note: can handle simultaneous FINs.
TCP Connection FSM

Not shown: various timers to get around losses, etc.
TCP Connection Management

- **Why so complex? Many challenges:**
  - Handle simultaneous connection establishment and teardown requests from both ends
  - Packet reordering can lead to complex error conditions:
    - Connection (IP1, port1, IP2, port2) established, torn down, new connection (IP1, port1, IP2, port2) established: how to avoid leakage from first to second connection through old segments?
  - Teardown: connections can be half-open, i.e., able to receive, but not send
  - etc.
Principles of Congestion Control

- **Congestion:**
  - informally: “too many sources sending too much data too fast for network to handle”
  - different from flow control!
  - manifestations:
    - lost packets (buffer overflow at routers)
    - long delays (queueing in router buffers)
  - a top-10 problem!
Causes/Costs of Congestion: Scenario 1

- two senders, two receivers
- one router, infinite buffers
- no retransmission

- large delays when congested
- maximum achievable throughput
Causes/Costs of Congestion: Scenario 2

- one router, \textit{finite} buffers
- sender retransmission of lost packet
Causes/Costs of Congestion: Scenario 2

- always: \( \lambda_{\text{in}} = \lambda_{\text{out}} \) (goodput)
- “perfect” retransmission only when loss: \( \lambda_{\text{in}}' > \lambda_{\text{out}} \)
- retransmission of delayed (not lost) packet makes \( \lambda_{\text{in}}' \) larger (than perfect case) for same \( \lambda_{\text{out}} \)

“costs” of congestion:
- more work (retransmission) for given “goodput”
- unneeded retransmissions: link carries multiple copies of pkt
Causes/Costs of Congestion: Scenario 3

- four senders
- multihop paths
- timeout/retransmit

Q: what happens as $\lambda_{in}$ and $\lambda'_{in}$ increase?
Causes/Costs of Congestion: Scenario 3

Another “cost” of congestion:
- when packet dropped, any “upstream transmission capacity used for that packet was wasted!
Two broad approaches towards congestion control:

**End-end congestion control:**
- no explicit feedback from network
- congestion inferred from end-system observed loss, delay
- approach taken by TCP

**Network-assisted congestion control:**
- routers provide feedback to end systems
  - single bit indicating congestion (SNA, DECbit, TCP/IP ECN, ATM)
  - explicit rate sender should send at
Case Study: ATM ABR Congestion Control

- **ABR: available bit rate:**
  - “Elastic service”
  - If sender’s path “underloaded”:
    - sender should use available bandwidth
  - If sender’s path congested:
    - Sender throttled to minimum guaranteed rate

- **RM (resource management) cells:**
  - sent by sender, interspersed with data cells
  - bits in RM cell set by switches (“network-assisted”)
    - NI bit: no increase in rate (mild congestion)
    - CI bit: congestion indication
  - RM cells returned to sender by receiver, with bits intact
Case Study: ATM ABR Congestion Control

- **Two-byte ER (explicit rate) field in RM cell**
  - congested switch may lower ER value in cell
  - sender’ send rate thus minimum supportable rate on path

- **EFCI bit in data cells: set to 1 in congested switch**
  - if data cell preceding RM cell has EFCI set, sender sets CI bit in returned RM cell
TCP Congestion Control

- End-end control (no network assistance)
- Sender limits transmission:
  \[ \text{LastByteSent} - \text{LastByteAcked} \leq \text{CongWin} \]
- Roughly, \( \text{CongWin} \) is dynamic, function of perceived network congestion

\[ \text{rate } R = \frac{\text{CongWin}}{\text{RTT}} \text{ Bytes/sec} \]

- How does sender perceive congestion?
  - Loss event = timeout or 3 duplicate acks
- TCP sender reduces rate (\( \text{CongWin} \)) after loss event
- Two mechanisms:
  - AIMD: Additive-Increase-Multiplicative-Decrease
  - Slow start: at beginning, after “really bad” congestion (timeout)
TCP AIMD in Congestion Avoidance (CA) State

- **Approach:**
  - Probe for available capacity by slowly increasing rate, back down when loss occurs

- **Additive increase:**
  - Increase CongWin by 1 MSS every RTT in the absence of loss events: probing

- **Multiplicative decrease:**
  - Cut CongWin in half after loss even
TCP Exponential Rampup in Slow Start State

- When connection begins: $\text{CongWin} = 1 \text{ MSS}$
  - Example: MSS = 500 bytes & RTT = 200 msec
  - Initial rate $R = 20 \text{ kbps}$
- Available bandwidth may be $\gg \text{MSS/RTT}$
  - Desirable to quickly ramp up to respectable rate
- When connection begins, increase rate exponentially fast until first loss event
TCP Slow Start (more)

- When connection begins, increase rate exponentially until first loss event:
  - Double CongWin every RTT, by incrementing CongWin for every ACK received

- Summary:
  - Initial rate is slow
  - But ramps up exponentially fast!
  - (maybe it should be called FastStart...)

![Diagram showing segment transmission over time between Host A and Host B with RTT intervals and segment counts labeled.]
Refinement: SlowStart after Timeout

- After 3 dup ACKs, i.e., “ACK(i), ACK(i), ACK(i)”
  - CongWin is cut in half
  - Window then grows linearly

- But after timeout event: SlowStart again
  - CongWin instead set to 1 MSS;
  - Window then grows exponentially to a threshold,
  - Then grows linearly

- Philosophy:
  - 3 dup ACKs indicates network capable of delivering some segments -> keep going at lower rate
  - timeout before 3 dup ACKs is “more alarming” -> be more careful and slow down drastically
Q: When should the exponential increase switch to linear?
- When CongWin gets to 1/2 of its value before timeout

Implementation:
- Variable Threshold
- At loss event, Threshold is set to 1/2 of CongWin just before loss event
# TCP Sender Congestion Control

<table>
<thead>
<tr>
<th>Event</th>
<th>State</th>
<th>TCP Sender Action</th>
<th>Commentary</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACK receipt for previously unacked data</td>
<td>Slow Start (SS)</td>
<td>CongWin = CongWin + MSS, If (CongWin &gt; Threshold) set state to “Congestion Avoidance”</td>
<td>Resulting in a doubling of CongWin every RTT</td>
</tr>
<tr>
<td>ACK receipt for previously unacked data</td>
<td>Congestion Avoidance (CA)</td>
<td>CongWin = CongWin+MSS * (MSS/CongWin)</td>
<td>Additive increase, resulting in increase of CongWin by 1 MSS every RTT</td>
</tr>
<tr>
<td>Loss event detected by triple duplicate ACK</td>
<td>SS or CA</td>
<td>Threshold = CongWin/2, CongWin = Threshold, Set state to “Congestion Avoidance”</td>
<td>Fast recovery, implementing multiplicative decrease. CongWin will not drop below 1 MSS.</td>
</tr>
<tr>
<td>Timeout</td>
<td>SS or CA</td>
<td>Threshold = CongWin/2, CongWin = 1 MSS, Set state to “Slow Start”</td>
<td>Enter slow start</td>
</tr>
<tr>
<td>Duplicate ACK</td>
<td>SS or CA</td>
<td>Increment duplicate ACK count for segment being acked</td>
<td>CongWin and Threshold not changed</td>
</tr>
</tbody>
</table>
Summary: TCP Congestion Control

- When $\text{CongWin}$ is below $\text{Threshold}$, sender is in slow-start phase, window grows exponentially.

- When $\text{CongWin}$ is above $\text{Threshold}$, sender is in congestion-avoidance phase, window grows linearly.

- When a triple duplicate ACK occurs, $\text{Threshold}$ set to $\text{CongWin}/2$ and $\text{CongWin}$ set to $\text{Threshold}$.

- When timeout occurs, $\text{Threshold}$ set to $\text{CongWin}/2$ and $\text{CongWin}$ is set to 1 MSS.
**TCP Fairness**

**Fairness goal:** if $K$ TCP sessions share same bottleneck link of bandwidth $R$, each should have average rate of $R/K$.

![TCP Fairness Diagram](image-url)
Why is TCP Fair?

- Two competing sessions:
  - Additive increase gives slope of 1, as throughput increases
  - Multiplicative decrease decreases throughput proportionally
Fairness (more)

- Fairness and UDP
  - Multimedia apps often do not use TCP
    - do not want rate throttled by congestion control
  - Instead use UDP:
    - pump audio/video at constant rate, tolerate packet loss
  - Research area: TCP friendly

- Fairness and parallel TCP connections
  - nothing prevents app from opening parallel connections between 2 hosts.
  - Web browsers do this
  - Example: link of rate R supporting 9 existing connections
    - new app asks for 1 TCP, gets rate R/10
    - new app asks for 11 TCPs, gets R/2!
TCP Throughput

- **Goal:** compute average throughput $R(L, RTT)$
  - Ignore slow start
- **Model:** perfect sawtooth
  - Loss exactly periodic
  - No RTT fluctuation
TCP Throughput (more)

- Let $W$ be the window size when loss occurs
  - When window is $W$, throughput is $W/RTT$
  - Just after loss, throughput drops to $W/(2 \times RTT)$
  - Average throughput: $R = 0.75 \times W/RTT$

- In one round:
  - One packet lost
  - $K = (W/2) + (W/2+1) + \ldots + W$ packets sent
  - $K = \frac{3}{8} W^2 + \frac{3}{4} W \approx \frac{3}{8} W^2$ for $W \gg 1$

- Loss rate:
  - $L = 1/K = 8/3 \times W^{-2}$

- Average rate:
  
  $$R = \frac{1.22 \times MSS}{RTT \sqrt{L}}$$
TCP Futures

- Example: MSS=1500 byte segments, 100ms RTT, want 10 Gbps throughput
  - Requires window size $W = 83333$ in-flight segments
  - Throughput in terms of loss rate:
    - Solve for $L \Rightarrow L = 2 \cdot 10^{-10}$
    - one segment in 5'000'000'000 lost: impossibly low!
    - bit error rate in optical fibers in same ballpark, wireless bit error rate many orders of magnitude higher
- Traditional TCP cannot operate in this regime
  - New versions of TCP for high-speed needed!
Summary

▪ Reliable transport in TCP:
  ▪ cumulative ACKs
  ▪ window, sequence numbers in bytes
  ▪ bi-directional
  ▪ optimizations: adaptive timer through RTT estimation, fast retransmit

▪ Window size is dynamic, control parameter for:
  ▪ flow control
  ▪ congestion control

▪ TCP congestion control:
  ▪ goal: utilize network fully without overloading, fairness
  ▪ fully distributed, end-to-end; no explicit network signal for congestion
  ▪ “discovery” of fair share through slow start/congestion avoidance

▪ Next:
  ▪ leaving the network “edge”
  ▪ into the “core” -> routers