Transmission Control Protocol

Instructor: Carey Williamson
Office: ICT 740
Email: carey@cpsc.ucalgary.ca

Class Location: ICT 122
Lectures: MWF 12:00 - 12:50


Slides are adapted from the book’s companion Web site, with changes by Anirban Mahanti and Carey Williamson.
TCP segment structure

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)

application data (variable length)

source port # | dest port #
--------------|--------------

sequence number

acknowledgement number

head len not used UAPR SF Receive window

checksum Urg data pnter

Options (variable length)

counting by bytes of data (not segments!)

# bytes rcvr willing to accept

CPSC 441:TCP 2
Sequence and Acknowledgement Number

TCP views data as unstructured, but ordered stream of bytes. Sequence numbers are over bytes, not segments.

Initial sequence number is chosen randomly.

TCP is full duplex – numbering of data is independent in each direction.

Acknowledgement number - sequence number of the next byte expected from the sender.

ACKs are cumulative.
TCP seq. #’s and ACKs

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 receiver handles out-of-order segments
A: TCP spec doesn’t say, - up to implementor

Host A
1000 byte data
Seq=42, ACK=79, data

Host B
host ACKs receipt of data
Seq=79, ACK=1043, no data

Host sends another 500 bytes
Seq=1043, ACK=79, data

Seq=79, ACK=1544, no data

Seq=79, ACK=1544, no data
TCP reliable data transfer

TCP creates rdt service on top of IP's unreliable service
Pipelined segments
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 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 acknowledges previously unacked segments
update what is known to be acked
start timer if there are outstanding segments
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 */
TCP Flow Control

The receive side of TCP connection has a receive buffer:

- **RevWindow**
- **RevBuffer**

- App process may be slow at reading from buffer
- Send rate is not too much, too fast
- Speed-matching service: matching the send rate to the receiving app’s drain rate

Sender won’t overflow receiver’s buffer by transmitting too much, too fast
TCP Flow control: how it works

(Suppose TCP receiver discards out-of-order segments)

spare room in buffer

= RcvWindow
= RcvBuffer - [LastByteRcvd - LastByteRead]

Rcvr advertises spare room by including value of RcvWindow in segments
Sender limits unACKed data to RcvWindow guarantees receive buffer doesn’t overflow
Silly Window Syndrome

Recall: TCP uses sliding window

“Silly Window” occurs when small-sized segments are transmitted, resulting in inefficient use of the network pipe

For e.g., suppose that TCP sender generates data slowly, 1-byte at a time

Solution: wait until sender has enough data to transmit – “Nagle’s Algorithm”
Nagle’s Algorithm

1. TCP sender sends the first piece of data obtained from the application (even if data is only a few bytes).

2. Wait until enough bytes have accumulated in the TCP send buffer or until an ACK is received.

3. Repeat step 2 for the remainder of the transmission.
Silly Window Continued ...

Suppose that the receiver consumes data slowly

Receive Window opens slowly, and thus sender is forced to send small-sized segments

Solutions

Delayed ACK
Advertise Receive Window = 0, until reasonable amount of space available in receiver’s buffer
TCP Connection Management

Recall: TCP sender, receiver establish “connection” before exchanging data segments
initialize TCP variables:
  - seq. #s
  - buffers, flow control info (e.g. RcvWindow)

client: connection initiator

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

server: contacted by client

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
TCP Connection Establishment

- Passive open: SYN/SYN+ACK
- Active open: SYN
- SYN_RCVD
- SYN_SENT
- Established
- SYN+ACK/ACK

Solid line for client
Dashed line for server
TCP Connection Termination

- client
  - FIN
  - FIN_WAIT1
  - FIN_WAIT2
  - TIME_WAIT
  - CLOSED

- server
  - ACK
  - FIN
  - LAST_ACK
  - CLOSE_WAIT
  - CLOSED

- States:
  - FIN_WAIT1
  - FIN_WAIT2
  - CLOSE_WAIT
  - LAST_ACK
  - CLOSED
  - TIME_WAIT
  - closing
  - ACK
  - timed wait
Principles of Congestion Control

**Congestion**: informally: “too many sources sending too much data too fast for network to handle”

Different from flow control!

**Manifestations:**
- Packet loss (buffer overflow at routers)
- Increased end-to-end delays (queuing in router buffers)

Results in unfairness and poor utilization of network resources
- Resources used by dropped packets (before they were lost)
- Retransmissions
- Poor resource allocation at high load
Historical Perspective

October 1986, Internet had its first congestion collapse
Link LBL to UC Berkeley
  400 yards, 3 hops, 32 Kbps
  throughput dropped to 40 bps
  factor of ~1000 drop!

Van Jacobson proposes TCP Congestion Control:
  Achieve high utilization
  Avoid congestion
  Share bandwidth
Congestion Control: Approaches

**Goal:** Throttle senders as needed to ensure load on the network is “reasonable”

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 (e.g., ECN)
- explicit rate sender should send at
TCP Congestion Control: Overview

end-end control (no network assistance)
Limit the number of packets in the network to window $W$
Roughly,

$$\text{rate} = \frac{W}{\text{RTT}} \text{ Bytes/sec}$$

$W$ is dynamic, function of perceived network congestion
TCP Congestion Controls

Tahoe (Jacobson 1988)
  Slow Start
  Congestion Avoidance
  Fast Retransmit
Reno (Jacobson 1990)
  Fast Recovery
SACK
Vegas (Brakmo & Peterson 1994)
  Delay and loss as indicators of congestion
“Slow Start” is used to reach the equilibrium state
Initially: \( W = 1 \) (slow start)
On each successful ACK:
\[ W \leftarrow W + 1 \]
Exponential growth of \( W \)
each RTT:
\[ W \leftarrow 2 \times W \]
Enter CA when
\[ W \geq \text{ssthresh} \]
\text{ssthresh: window size after which TCP cautiously probes for bandwidth}
Congestion Avoidance

Starts when

\[ W \geq ssthresh \]

On each successful ACK:

\[ W \leftarrow W + \frac{1}{W} \]

Linear growth of \( W \) each RTT:

\[ W \leftarrow W + 1 \]
CA: Additive Increase, Multiplicative Decrease

We have “additive increase” in the absence of loss events.

After loss event, decrease congestion window by half - “multiplicative decrease”

\[ \text{ssthresh} = \frac{W}{2} \]

Enter Slow Start
Detecting Packet Loss

Assumption: loss indicates congestion

Option 1: time-out
Waiting for a time-out can be long!

Option 2: duplicate ACKs
How many? At least 3.
Fast Retransmit

Wait for a timeout is quite long
Immediately retransmits after 3 dupACKs without waiting for timeout
Adjusts ssthresh

\[ \text{ssthresh} \leftarrow \frac{W}{2} \]

Enter Slow Start

\[ W = 1 \]
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
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

EstimatedRTT = (1 - $\alpha$)*EstimatedRTT + $\alpha$*SampleRTT

EWMA
influence of past sample decreases exponentially fast
typical value: $\alpha = 0.125$
TCP Round Trip Time and Timeout

[Jacobson/Karels Algorithm]

**Setting the timeout**

EstimatedRTT plus “safety margin”
- large variation in EstimatedRTT → larger safety margin
- first estimate how much SampleRTT deviates from EstimatedRTT:

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

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

Then set timeout interval:

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

Typically, \( \mu = 1 \) and \( \phi = 4 \).
TCP Tahoe: Summary

Basic ideas
- Gently probe network for spare capacity
- Drastically reduce rate on congestion
- Windowing: self-clocking
- Other functions: round trip time estimation, error recovery

```plaintext
for every ACK {
    if (W < ssthresh) then W++  (SS)
    else W += 1/W  (CA)
}
for every loss {
    ssthresh = W/2
    W = 1
}
```
TCP Tahoe

Reached initial ssthresh value; switch to CA mode

Slow Start

Window

$W_1$

$ssthresh = W_1/2$

$W_2$

$ssthresh = W_2/2$

$W_1/2$

$W_2/2$

Time
Questions?

Q. 1. To what value is ssthresh initialized to at the start of the algorithm?

Q. 2. Why is “Fast Retransmit” triggered on receiving 3 duplicate ACKs (i.e., why isn’t it triggered on receiving a single duplicate ACK)?

Q. 3. Can we do better than TCP Tahoe?
TCP Reno

Note how there is “Fast Recovery” after cutting Window in half
TCP Reno: Fast Recovery

Objective: prevent `pipe’ from emptying after fast retransmit

- each dup ACK represents a packet having left the pipe (successfully received)
- Let’s enter the “FR/FR” mode on 3 dup ACKs

ssthresh ← W/2
retransmit lost packet
W ← ssthresh + ndup (window inflation)
Wait till W is large enough; transmit new packet(s)
On non-dup ACK (1 RTT later)
  W ← ssthresh (window deflation)
  enter CA mode
TCP Reno: Summary

Fast Recovery along with Fast Retransmit used to avoid slow start

On 3 duplicate ACKs
   Fast retransmit and fast recovery

On timeout
   Fast retransmit and slow start
TCP Throughput

What’s the average throughout of TCP as a function of window size and RTT?

Ignore slow start

Let $W$ be the window size when loss occurs.

When window is $W$, throughput is $W/RTT$

Just after loss, window drops to $W/2$, throughput to $W/2RTT$.

Average throughout: .75 $W/RTT$
TCP Futures

Example: 1500 byte segments, 100ms RTT, want 10 Gbps throughput
Requires window size $W = 83,333$ in-flight segments
Throughput in terms of loss rate:

$$\frac{1.22 \cdot MSS}{RTT \sqrt{L}}$$

$\Rightarrow L = 2 \cdot 10^{-10}$ Wow

New versions of TCP for high-speed needed!
TCP Fairness

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

TCP fairness: dependency on RTT
- Connections with long RTT get less throughput

Parallel TCP connections

TCP friendliness for UDP streams
Chapter 3: Summary

principles behind transport layer services:
  multiplexing, demultiplexing
  reliable data transfer
  flow control
  congestion control
instantiation and implementation in the Internet
  UDP
  TCP

Next:
leaving the network “edge” (application, transport layers) into the network “core”