CSCI-1680
Transport Layer II

Data over TCP: Flow Control

Nick DeMarinis

Based partly on lecture notes by Rodrigo Fonseca, David Mazières, Phil Levis, John Jannotti
TCP Gearup I TONIGHT (10/26) 5-7pm, CIT368
  – How the project works, how to think about sockets
  – Stuff you need for milestone 1

TCP milestone 1: Schedule on/before Thursday, November 2
  – Email later today for signups

HW2: Due Mon, Oct 30
  – Last problem helpful for milestone 1
Topics for today

• Flow control: Sliding window
• Computing RTO
• Connection termination
The story so far

Stop and Wait: Simplest TCP sender/receiver

Key features
- SEQ/ACK numbers denote where sender/receiver are in data stream
- Only one segment is “in flight” at a time
Warmup: Stop and Wait

What are the values for the SEQ and ACK fields?

```
conn.Write("hello_world")
```

SEQ: where segment is in data stream
ACK: next byte the sender expects to get

eg. ACK x “I have up to (x - 1), send me x next”

(TCP Handshake)
Warmup:  Stop and Wait

What are the values for the SEQ and ACK fields?

```python
conn.Write("hello_world")
```

Key features
- **SEQ**: Position of this segment in the data stream
- **ACK**: Next sequence number the receiver expects to receive (ACK N == “I have up to (N - 1)"")

**Advertised window**: how much space the receiver has left in its receive buffer

=> Window (WIN) field in TCP header

(TCP Handshake)
LOTS OF UNUSED BW

WANT: MORE DATA "IN FLIGHT" AT A TIME.

⇒ SLIDING WINDOW

STOP WAIT:
TCP and buffering

Recall: TCP stack responsibilities

- **Sender**: breaking application data into segments
- **Receiver**: receiving segments, reassembling them in order

TCP stack needs to buffer data for both parts

- **Sender**: data waiting to be sent, not yet ACK’d
- **Receiver**: data not yet read by app, out-of-order segments

**Remember**: in reality, both sides can send and receive!

=> All sockets have both a send and receive buffer
RFC 9293: Sec 3.1, 3.3.1, 3.4

**Sliding Window**:

- **W**: App loads data into buffer (comm. write)
- **R**: TCP stack sends data to be sent
- **K**: App loads data (write)

- **SND**: Window size
- **UNA**: Oldest unacked segment
- **NXT**: Next sequence number to be sent
- **LBW**: Last byte written

- **Bytes in Flight**: Data that has been sent out, but not ack'd yet

*Note: If buffer becomes full, write from app should block until data available.*
**SENDER**
- **OPERATION**
  - Send up to window (advance **NXT**)
  - Bytes in flight < advertised window
  - Keep track of "in flight" segments, retransmit on timeout ("retransmit queue")
- On **ACK** for some segment **S**,
  - **ACK** must fall within window

\[ \text{UNA} < S, \text{ACK} \leq \text{NXT} \]

- **IF NOT**, **ACK** is invalid/old \( \Rightarrow \) drop.
- **OTHERWISE**
  - **UNA** += \( (\text{How much data was ACK'd}) \)
  - **IF ACK** fully covered a segment, remove from retransmit queue.
Example: 10 1-byte segments

UNA = 10
W = [10, 11, 12, 13]

IN FLIGHT

IF YOU GET ACK = 12
W = [12, 13, 14, 15]

CAN SEND.

Ex. 10 Byte Segments

START

1. UNA = 9
IN FLIGHT = [10, 20, 30, 40]

@ACK: 30

2. UNA = 31
IN FLIGHT = [30, 40, 50, 60]

@SEQ

@ACK: "I have received up to 30" (expect SEG 30 next)

FOR EACH SEGMENT
- KEEP TIMESTAMP OF LAST SENT TIME
- RETRANSMIT IF IT EXPIRES.
**Receiving Side (RCV)**

Data waiting to be read by app

- Max Buf Size

**Early Arrivals**

- Pocket that arrives out of order

**RCV.NXT** - Next Byte Expect to Receive
- Next Seq Num Expect to RCV

**LBR** - Last Byte Read By APP

**Advertised Window** - Amount of Space Remaining in Buffer (Can be 0)

\[ \text{Advertised Window} = \text{Max Buf} - (\text{NXT} - 1) - \text{LBR} \]

\[ \text{This is what is sent in Window Field} \]

**Problem:** Out of Order Packets

**Solution:** Early Arrival Queue
- Tracks segments arriving after NXT (But Win Bound)
WHEN RECEIVER GETS A SEGMENT, S
MUST CHECK IF IT FITS IN WINDOW:
S:SEQ < RCV:NXT AND S:SEQ < RCV, NXT + RCV:WND

(SIMILAR CHECK FOR END OF WINDOW)  
(RFC 9293, Sec 3.4)

- ADD AT POSITION S:SEQ
- NXT += SEGMENT SIZE
- CHECK EARLY ARRIVAL
- QUEUES - MOVE UP TO NEXT CONTINUOUS PART
Sender example from class (cleaner version on next page)

ACK number: last segment the receiver has in order
UPDATE CONSIDER EARLY ARRIVALS

ACKS ALWAYS REFLECT LAST BYTE RECEIVED IN ORDER

SEND: 1 2 3 4 5 6
recv: 1 2 4 5

ACK (2)

ACK (3)

ACK (3)

ACK (6)

SENDER

RECEIVER

ACK (3)

ACK (3)

ACK (3)

ACK (6)

SEND: 1 2 3 4 5 6
recv: 1 2 4 5

ACK (3)

ACK (3)

ACK (3)

ACK (6)

SENDER

RECEIVER

ACK (3)

ACK (3)

ACK (3)

ACK (6)
Some Visualizations

• Normal conditions:  https://www.youtube.com/watch?v=zY3Sxvj8kZA

• With packet loss:  https://www.youtube.com/watch?v=lk27yilTOvU
What happens if the receiving app never reads from its buffer?

- Receive buffer fills up
- Advertised window goes to zero
- When WIN=0, sender must stop sending
- Send buffer will fill up (if app keeps sending)
- If send buffer is full, sender’s Write() will block
What happens if the receiving app never reads from its buffer?

Problem: need a way for sender to know when space is available again!

Resolution: zero window probing
- Sender periodically sends 1-byte segments
- Receiver sends back ACK with advertised window (even if it has no room for segment)
Problem: need a way for sender to know when space is available again!

Resolution: **zero window probing**

- Sender periodically sends 1-byte segments
- Receiver sends back ACK with advertised window (even if it has no room for segment)
- Sender can resume sending when \( \text{win} \neq 0 \) (preferably when \( \text{win} \geq \text{MSS} \))
Complete send/recv example with zero-window probing (ZWP part on next page)

DATA "in flight"
(WAITING FOR ACK)
(BOUNDED BY)

Send must stop sending.

Enter zero window mode.
What to do when window is full?

Send window probe.

- Sender: Sends window probe periodically.
- Receiver: Will indicate if window has changed.

Sender sends 1 byte, receiver sends ACK=1, window=1.

Next byte.

No space available, resume!

Space now available, continue.
TCP State Diagram
How do ACKs work?

- ACK contains next expected sequence number
- Sender: if one segment is missed but new ones received, send duplicate ACK
How do ACKs work?

• ACK contains next expected sequence number
• Sender: if one segment is missed but new ones received, send duplicate ACK
• Receiver retransmits when:
  – Receive timeout (RTO) expires
  – Possibly other conditions, for certain TCP variants (eg. 3 dup ACKs)

• How to set RTO?
What’s a good timeout value?
- 0.5s? 1s? 0.01s?

Thoughts?
- If timeout is too small, packet might have not arrived (latency)
- If timeout is too long, will affect throughput

=> Can’t just pick a fixed timeout value

Strategy: measure RTT based on ACKs received, use this to set a timeout value
=> Timeout time is called RTO
Computing RTO

Strategy: measure expected RTT based on ACKs received

Use exponentially weighted moving average (EWMA)

- RFC793 version ("smoothed RTT"): 
  \[
  SRTT = (\alpha \times SRTT_{\text{Last}}) + (1 - \alpha) \times RTT_{\text{Measured}} \\
  RTO = \max(RTO_{\text{Min}}, \min(\beta \times SRTT, RTO_{\text{Max}}))
  \]

  \(\alpha = \) "Smoothing factor": .8-.9  
  \(\beta = \) "Delay variance factor": 1.3—2.0  
  \(RTO_{\text{Min}} = 1\) second

RFC793, Sec 3.7
RFC6298 (slightly more complicated, also measures variance)
Using the RTO timer

Recommended by RFC6298

• Maintain ONE timer per connection
• When segment is sent => set timer to expire after $t_{RTO}$
• When ACK is received with new data, reset the timer

When the timer expires:

• Retransmit earliest unacknowledged segment
• $RTO = 2 \times RTO$ (up to some max)
• If no data after N retransmissions => give up, terminate connection
This is only the beginning...

- Problem 1: what if ACK is for a retransmitted segment?
  - Solution: don’t update RTT if segment was retransmitted

- Problem 2: RTT can have high variance
  - Initial implementation doesn’t account for this (modern version, RFC6298)
  - Congestion control: modeling network load