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 milestone 1: Schedule on/before Monday, November 7
  – Email later today for signups
HW4: Announcement soon
Warmup: Stop and Wait

**MSS = 5**

conn.write("hello world")

WHAT ARE THE VALUES FOR 0, 2, 3?

ADVERTISED WINDOW, HOW MUCH DATA WE CAN SEND.
Topics for today

• Flow control: Sliding window
• Computing RTO
• Connection termination
TCP and buffering

Recall: TCP stack responsibilities

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

So both sides can send, receive.

- Need to buffer data on each side

Connection is bidirectional.
HTTP

This is not efficient. Need to wait 1 RTT. Pen Solomons 3 once

Send can have a "window" of data it can have in-flight at once.
Sliding window: in abstract terms

- Window of size $w$
- Can send at most $w$ packets before waiting for an ACK
Sliding window: in abstract terms

- Window of size $w$
- Can send at most $w$ packets before waiting for an ACK
- Goal
  - Network "pipe" always filled with data
  - ACKs come back at rate data is delivered => "self-clocking"
Buffering is sequence numbers.
- Sequence numbers
  - Sequence numbers ISN

- Send buffer, Recv buffer
  - Fixed size lives in TCP stack
    - Recommended: \(2^{32-1}\) bytes

Two "domains" for thinking about data:
- "sequence number space" (starts at ISN)
  - Seq ACK Nums sent in packets
- "buffer space": How you data is stored in the TCP stack
  - Implemented using a "circular buffer" or "ring buffer"

- Size of buffer defines bounds of sequence numbers that are valid at any point.
RFC 9293: Sec 3.1, 3.3.1, 3.4

Sliding Window: Sending Side.

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

App loads data (write)

SND, UNA - oldest unacknowledged segment
SND, NXT - next sequence number to be sent
- next byte to be sent

LBW - last byte written

Bytes "in flight" - data that has been sent out, but not acknowledged 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
  \[
  \text{UNA} < S, \text{ACK} \leq \text{NXT} \]
  \[
  \Rightarrow \quad \text{within "Bytes in Flight"}
  \]
  - If not, Ack is invalid/old \(\Rightarrow\) Drop.

**Otherwise**
- UNA + = (How much Data)
  \[\text{(Was Ack'd)}\]
- If Ack fully covered a Segment, remove from Retransmit Queue
**Example**: 10 1-byte segments

- **UNA**: 10
- **W**: 4
- **W**: \([10, 11, 12, 13]\)

- **In Flight**

- **IF YOU GET** **ACK**: 12, **W**: \([12, 13, 14, 15]\)

---

**Ex. 10 BYTE SEGMENTS**

**START**

- **UNA**: 9
- **In Flight**: \([10, 12, 30, 40]\)

- **ACK**: 30

**END**

- **UNA**: 21
- **In Flight**: \([30, 40, 50, 60]\)

**SEQ:**

**FOR EACH SEGMENT**

- **KEEP TIMES** **AND** **OF LAST** **SENT TIME**
- **RETRANSMIT** **IF IT EXPIRES**.
RECEIVING SIDE (RCV)

Data waiting to be read by app

+ MAX BUF SIZE

LBR | NXT

App reads data (comm. read)

"Early arrivals" = packets that arrive 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)

= MAX BUF - (L NXT - 1) - LBR

This is what is sent in window field

Problem: Out of order packets

Solution: Early arrival queue

- Tracks segments arriving after NXT (but within bound)
WHEN RECEIVER GETS A SEGMENT, S
MUST CHECK IF FITS IN WINDOW:
S.seq < RCV.nxt AND S.seq < RCV.nxt + RCV.window

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

- ADD AT POSITION S.seq
  - RCV.nxt += SEGMENT SIZE
  - CHECK EARLY ARRIVAL QUEUE - MOVE UP TO NEXT CONTINUOUS PART
Flow Control: Sender

Invariants

- $\text{LastByteSent} - \text{LastByteAcked} \leq \text{AdvertisedWindow}$
- $\text{EffectiveWindow} = \text{AdvertisedWindow} - (\text{BytesInFlight})$
- $\text{LastByteWritten} - \text{LastByteAcked} \leq \text{MaxSendBuffer}$
Flow Control: Sender

Invariants

- $\text{LastByteSent} - \text{LastByteAcked} \leq \text{AdvertisedWindow}$
- $\text{EffectiveWindow} = \text{AdvertisedWindow} - (\text{BytesInFlight})$
- $\text{LastByteWritten} - \text{LastByteAcked} \leq \text{MaxSendBuffer}$

Useful Sliding Window Terminology:
RFC 9293, Sec 3.3.1
Flow control: receiver

- Can accept data if space in window
- Available window = BufferSize - ((NextByteExpected-1) - LastByteRead)

- On receiving segment for byte S
  - if s is outside window, ignore packet
  - if s == NextByteExpected:
    - Deliver to application (Update LastByteReceived)
    - If next segment was early arrival, deliver it too
  - If s > NextByteExpected, but within window
    - Queue as early arrival

- Send ACK for highest contiguous byte received, available window
Flow control: receiver

- Can accept data if space in window
- Available window = BufferSize - ((NextByteExpected-1) - LastByteRead)

- On receiving segment for byte S
  - if s is outside window, ignore packet
  - if s == NextByteExpected:
    - Deliver to application (Update LastByteReceived)
    - If next segment was early arrival, deliver it too
  - If s > NextByteExpected, but within window
    - Queue as early arrival

- Send ACK for highest contiguous byte received, available window
Flow Control

• Advertised window can fall to 0
  – How?
  – Sender eventually stops sending, blocks application

• Resolution: zero window probing: sender sends 1-byte segments until window comes back > 0
Sequence numbers
(Circumference = 0 to $2^{32}$ slots)

- Initial sequence number
- Data received, acknowledged and delivered to application
- Data received, acknowledged but not yet delivered to application
- Data received, but not acknowledged
- Unfilled buffer

Window shifts

Receiver's window (Allocation buffer)
Up to $2^{16} - 1$ slots

In all cases
How do ACKs work?

- ACK contains next expected sequence number
- If one segment is missed but new ones received, send duplicate ACK
- Retransmit when:
  - Receive timeout (RTO) expires
  - Possibly other conditions, for certain TCP variants (e.g., 3 dup ACKs)
- How to set RTO?
Some Visualizations

• Normal conditions: [https://www.youtube.com/watch?v=zY3Sxvj8kZA](https://www.youtube.com/watch?v=zY3Sxvj8kZA)

• With packet loss: [https://www.youtube.com/watch?v=lk27yiITOvU](https://www.youtube.com/watch?v=lk27yiITOvU)
MORE CONTENT
WE WILL COVER NEXT CLASS

FEEL FREE TO READ AHEAD!
When to time out?

Should expect an ACK within one Round Trip Time (RTT)

- Problem: RTT can be highly variable

- Strategy: expected RTT based on ACKs received
  - Use exponentially weighted moving average (EWMA)
  - RFC793 version (“smoothed RTT”):
When to time out?

Should expect an ACK within one Round Trip Time (RTT)
- Problem: RTT can be highly variable

- Strategy: expected RTT based on ACKs received
  - Use exponentially weighted moving average (EWMA)
  - RFC793 version ("smoothed RTT"):
    \[
    SRTT = (\alpha \times SRTT) + (1 - \alpha) \times RTT_{\text{Measured}}
    \]
    \[
    RTO = \max(RTO_{\text{Min}}, \min(\beta \times SRTT, RTO_{\text{Max}}))
    \]

\(\alpha = \) "Smoothing factor": .8-.9
\(\beta = \) "Delay variance factor": 1.3—2.0

RFC793, Sec 3.7
This is only the beginning...

- Problem 1: what if segment is a retransmission?
This is only the beginning...

- Problem 1: what if segment is a retransmission?
  - Solution: don’t update RTT if segment was retransmitted
This is only the beginning...

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

- Problem 2: RTT can have high variance
  - Initial implementation doesn’t account for this
This is only the beginning...

• Problem 1: what if segment is a retransmission?
  – Solution: don’t update RTT if segment was retransmitted

• Problem 2: RTT can have high variance
  – Initial implementation doesn’t account for this
  – Congestion control: modeling network load
When to Transmit?

Nagle’s algorithm

• Goal: reduce the overhead of small packets
  if (there is data to send) and (window >= MSS)
    Send a MSS segment
  else
    if there is unAcked data in flight
      buffer the new data until ACK arrives
    else
      send all the new data now

• Receiver should avoid advertising a window <= MSS after advertising a window of 0
Delayed Acknowledgments

• Goal: Piggy-back ACKs on data
  – Delay ACK for 200ms in case application sends data
  – If more data received, immediately ACK second segment
  – Note: never delay duplicate ACKs (if missing a segment)
Delayed Acknowledgments

• Goal: Piggy-back ACKs on data
  – Delay ACK for 200ms in case application sends data
  – If more data received, immediately ACK second segment
  – Note: never delay duplicate ACKs (if missing a segment)

• Warning: can interact badly with Nagle for some applications
  – Nagle waits for ACK until send => Temporary deadlock
  – App can disable Nagle with TCP_NODELAY
  – App should also avoid many small writes
Summary: flow control

• Flow control provides correctness: reliable, in order delivery
• Need more for performance
  – What if the network is the bottleneck?
• Sending too fast will cause queue overflows, heavy packet loss
• Need more for performance: congestion control
Connection Termination

• When you have no more data to send, send a FIN
  – Sent by close() or shutdown()
• Both sides close connection separately!
Connection Termination

• When you have no more data to send, send a FIN
  – Sent by close() or shutdown()
• Both sides close connection separately!
• TIME_WAIT: initiating side should wait for 2*MSL before deleting TCB
  – MSL = Longest time a segment might be delayed (configurable, ~1min)
TCP State Diagram

1. **CLOSED**
   - Transition to:**LISTEN**
     - **CONNECT/SYN** (Step 1 of the 3-way-handshake)
   - Transition from:**CLOSE/A**

2. **LISTEN**
   - Transition to:**SYN/SYN+ACK**
     - **SYN/SYN+ACK** (Step 2 of the 3-way-handshake)
   - Transition from:**CLOSE/A**

3. **SYN/SYN+ACK**
   - Transition to:**SYN SENT**
     - **SEND/SYN**
   - Transition from:**CLOSE/A**

4. **SYN SENT**
   - Transition to:**SYN+ACK/ACK**
     - **SYN+ACK/ACK** (Step 3 of the 3-way-handshake)
   - Transition from:**CLOSE/A**

5. **SYN RECEIVED**
   - Transition to:**ESTABLISHED**
     - **ACK**
   - Transition from:**CLOSE/FIN**

6. **ESTABLISHED**
   - Transition to:**FIN WAIT 1**
     - **FIN**
   - Transition from:**CLOSE/FIN**

7. **FIN WAIT 1**
   - Transition to:**FIN WAIT 2**
     - **FIN**
   - Transition from:**TIME WAIT**

8. **FIN WAIT 2**
   - Transition to:**TIME WAIT**
     - **TIME OUT**
   - Transition from:**CLOSED**

9. **TIME WAIT**
   - Transition to:**CLOSED**
     - **TIME OUT**
   - Transition from:**CLOSE/FIN**

10. **CLOSED**
    - Transition to:**CLOSED**
      - **TIME OUT**
    - Transition from:**CLOSE/FIN**

11. **CLOSE/FIN**
    - Transition to:**CLOSED**
      - **TIME OUT**
    - Transition from:**CLOSE/FIN**