TCP (Contd.)

CPSC 433/533, Spring 2021

Anurag Khandelwal

TCP Header

Source Port
Destination Port
Sequence Number
Acknowledgement
Header Length
0
Flags
Advertised Window
Checksum
Urgent Pointer
Options (variable)
Payload

TCP Connection Establishment and Initial Sequence Numbers

Timing Diagram: 3-way Handshaking

Timing Diagram: 3-way Handshaking

Timing Diagram: 3-way Handshaking

Timing Diagram: 3-way Handshaking

Tearing Down the Connection

Normal Termination, One Side At a Time

Normal Termination, One Side At a Time

- Finish (**FIN**) to close and receive remaining bytes
  - **FIN** occupies one byte in the sequence space

Normal Termination, One Side At a Time

- Finish (**FIN**) to close and receive remaining bytes
  - **FIN** occupies one byte in the sequence space
- Other host ACKs the byte to confirm

Normal Termination, One Side At a Time

Connection now **half-closed**

- Finish (**FIN**) to close and receive remaining bytes
  - **FIN** occupies one byte in the sequence space
- Other host ACKs the byte to confirm
- Closes A's side of the connection but *not* B's

Normal Termination, One Side At a Time

- Finish (**FIN**) to close and receive remaining bytes
  - **FIN** occupies one byte in the sequence space
- Other host ACKs the byte to confirm
- Closes A's side of the connection but *not* B's
  - Unti B likewise sends a **FIN**

Normal Termination, One Side At a Time

- Finish (**FIN**) to close and receive remaining bytes
  - **FIN** occupies one byte in the sequence space
- Other host ACKs the byte to confirm
- Closes A's side of the connection but *not* B's
  - Unti B likewise sends a **FIN**
  - Which A then ACKs

Normal Termination, One Side At a Time

TIME_WAIT:
Avoid reincarnation;
B will retransmit FIN if
ACK is lost

- Finish (**FIN**) to close and receive remaining bytes
  - **FIN** occupies one byte in the sequence space
- Other host ACKs the byte to confirm
- Closes A's side of the connection but *not* B's
  - Unti B likewise sends a **FIN**
  - Which A then ACKs

Normal Termination, One Side At a Time

Connection now **closed**

- Finish (**FIN**) to close and receive remaining bytes
  - **FIN** occupies one byte in the sequence space
- Other host ACKs the byte to confirm
- Closes A's side of the connection but *not* B's
  - Unti B likewise sends a **FIN**
  - Which A then ACKs

Normal Termination, Both Together

Normal Termination, Both Together

- Same as before, but B sets FIN with their ACK of A's FIN

Normal Termination, Both Together

- Same as before, but B sets FIN with their ACK of A's FIN

Normal Termination, Both Together

- Same as before, but B sets FIN with their ACK of A's FIN

Abrupt Termination

Abrupt Termination

- A sends RESET (**RST**) to B

Abrupt Termination

- A sends RESET (**RST**) to B
  - E.g., because application on process A crashed

Abrupt Termination

- A sends RESET (**RST**) to B
  - E.g., because application on process A crashed
- **That's it**
  - B does *not* ACK the RST, i.e., RST is not delivered reliably
  - And: any data in flight is lost

Abrupt Termination

- A sends RESET (**RST**) to B
  - E.g., because application on process A crashed
- **That's it**
  - B does *not* ACK the RST, i.e., RST is not delivered reliably
  - And: any data in flight is lost
  - But: if B sends anything more, will elicit another RST **(So rude!)**

Abrupt Termination

- A sends RESET (**RST**) to B
  - E.g., because application on process A crashed
- **That's it**
  - B does *not* ACK the RST, i.e., RST is not delivered reliably
  - And: any data in flight is lost
  - But: if B sends anything more, will elicit another RST **(So rude!)**

Abrupt Termination

- A sends RESET (**RST**) to B
  - E.g., because application on process A crashed
- **That's it**
  - B does *not* ACK the RST, i.e., RST is not delivered reliably
  - And: any data in flight is lost
  - But: if B sends anything more, will elicit another RST **(So rude!)**

TCP Header

SYN
ACK
FIN
RST
PSH
URG

TCP State Transitions

TCP State Transitions

Data, ACK exchanges are in here

A Simpler View on the Client Side

A Simpler View on the Client Side

A Simpler View on the Client Side

A Simpler View on the Client Side

A Simpler View on the Client Side

A Simpler View on the Client Side

A Simpler View on the Client Side

TCP Header

Used to negotiate use of additional features

TCP Header

Source Port
Destination Port
Sequence Number
Acknowledgement
Header Length
0
Flags
Advertised Window
Checksum
Urgent Pointer
Options (variable)
Payload

TCP: Sliding Window (so far)

TCP: Sliding Window (so far)

- **Both sender & receiver maintain a <u>window</u>**

TCP: Sliding Window (so far)

- **Both sender & receiver maintain a <u>window</u>**
- **Left edge of window:**
  - Sender: beginning of ***unacknowledged*** data
  - Receiver: beginning of ***expected*** data
    - First "hole" in received data
    - When sender gets ACK, knows that the received's window has moved

TCP: Sliding Window (so far)

- **Both sender & receiver maintain a <u>window</u>**
- **Left edge of window:**
  - Sender: beginning of ***unacknowledged*** data
  - Receiver: beginning of ***expected*** data
    - First "hole" in received data
    - When sender gets ACK, knows that the received's window has moved
- **Right edge: left edge + constant**
    - Constant only limited by buffer size in the transport layer

TCP: Sliding Window (so far)

- **Both sender & receiver maintain a <u>window</u>**
- **Left edge of window:**
  - Sender: beginning of ***unacknowledged*** data
  - Receiver: beginning of ***expected*** data
    - First "hole" in received data
    - When sender gets ACK, knows that the received's window has moved
- **Right edge: left edge + constant**
    - Constant only limited by buffer size in the transport layer

TCP: Sliding Window (so far)

- **Both sender & receiver maintain a <u>window</u>**
- **Left edge of window:**
  - Sender: beginning of ***unacknowledged*** data
  - Receiver: beginning of ***expected*** data
    - First "hole" in received data
    - When sender gets ACK, knows that the received's window has moved
- **Right edge: left edge + constant**
    - Constant only limited by buffer size in the transport layer

Where in the transport layer?

TCP: Sliding Window (so far)

- **Both sender & receiver maintain a <u>window</u>**
- **Left edge of window:**
  - Sender: beginning of ***unacknowledged*** data
  - Receiver: beginning of ***expected*** data
    - First "hole" in received data
    - When sender gets ACK, knows that the received's window has moved
- **Right edge: left edge + constant**
    - Constant only limited by buffer size in the transport layer

Where in the transport layer?

Only the transport layer?

Where is the buffer?

Where is the buffer?

Where is the buffer?

Where is the buffer?

Where is the buffer?

Where is the buffer?

How does TCP deal with buffer limits?

How does TCP deal with buffer limits?

How does TCP deal with buffer limits?

Sliding Window at Sender

Sliding Window at Sender

Sliding Window at Sender

Sliding Window at Sender

Sliding Window at Sender

Sliding Window at Sender

Sliding Window at Receiver

Sliding Window at Receiver

Sliding Window at Receiver

Sliding Window at Receiver

Sliding Window at Receiver

Sliding Window at Receiver

Sliding Window at Receiver

Solution: Advertised Window (Flow Control)

Solution: Advertised Window (Flow Control)

- **Receiver uses an "Advertised Window" (W) to prevent sender from overflowing its window**
  - Receiver indicates value of W in ACKs
  - Sender limits number of bytes it can have in flight <= W

Sliding Window at Receiver

Sliding Window at Receiver

TCP Header

Source Port
Destination Port
Sequence Number
Acknowledgement
Header Length
0
Flags
Advertised Window
Checksum
Urgent Pointer
Options (variable)
Payload

Sliding Window at Sender

Sliding Window at Sender

Sliding Window at Sender

Sending Window with Flow Control

Sending Window with Flow Control

- **Sender:** window advances *when new data ACKed*

Sending Window with Flow Control

- **Sender:** window advances *when new data ACKed*
- **Receiver:** window advances as *receiving process consumes data*

Sending Window with Flow Control

- **Sender:** window advances *when new data ACKed*
- **Receiver:** window advances as *receiving process consumes data*
- Receiver *advertises* to sender where receiver window currently ends ("right hand edge")
  - Sender agrees not to exceed this amount

Advertised Window Limits Rate

Advertised Window Limits Rate

- Sender can send no faster than W/RTT bytes/sec

Advertised Window Limits Rate

- Sender can send no faster than W/RTT bytes/sec
- Receiver only advertises more space when it has consumed old arriving data

Advertised Window Limits Rate

- Sender can send no faster than W/RTT bytes/sec
- Receiver only advertises more space when it has consumed old arriving data
- In original TCP design, that was the **sole** protocol mechanism controlling the sender's rate

Advertised Window Limits Rate

- Sender can send no faster than W/RTT bytes/sec
- Receiver only advertises more space when it has consumed old arriving data
- In original TCP design, that was the **sole** protocol mechanism controlling the sender's rate
- What was missing?

How does TCP deal with buffer limits?

How does TCP deal with buffer limits?

How does TCP deal with buffer limits?

What's the challenge of doing this in TCP?

Statistical Multiplexing → Congestion

Statistical Multiplexing → Congestion

- **If two packets arrive at a router the same time for the same port**
  - Router will transmit only one, and buffer/drop the other

Statistical Multiplexing → Congestion

- **If two packets arrive at a router the same time for the same port**
  - Router will transmit only one, and buffer/drop the other
- **Internet traffic is *<u>bursty</u>***

Statistical Multiplexing → Congestion

- **If two packets arrive at a router the same time for the same port**
  - Router will transmit only one, and buffer/drop the other
- **Internet traffic is *<u>bursty</u>***
- **If many packets arrive in close time**
  - The router cannot keep up → gets congested
  - Causes packet delays and drops

A few design considerations

…if you were starting with TCP

A few design considerations

…if you were starting with TCP

- **How do we know when the network is congested?**

A few design considerations

…if you were starting with TCP

- **How do we know when the network is congested?**
- **Who takes care of congestion?**
  - Network, end hosts, both?

A few design considerations

…if you were starting with TCP

- **How do we know when the network is congested?**
- **Who takes care of congestion?**
  - Network, end hosts, both?
- **How do we handle congestion?**

TCP's Approach

TCP's Approach

- **Based on implicit feedback from the network**
  - Why not explicit?

TCP's Approach

- **Based on implicit feedback from the network**
  - Why not explicit?
- **End-hosts adjust sending rate**

TCP's Approach

- **Based on implicit feedback from the network**
  - Why not explicit?
- **End-hosts adjust sending rate**
- **How?**
  - Rest of this lecture…

TCP's Approach

- **Based on implicit feedback from the network**
  - Why not explicit?
- **End-hosts adjust sending rate**
- **How?**
  - Rest of this lecture…
- **Why?**
  - A consequence of history rather than planning
  - *Not* the only approach

Some History: TCP in the 1980s

Some History: TCP in the 1980s

- **Sending rate only limited by flow control**
  - Dropped packets → senders (repeatedly!) retransmit

Some History: TCP in the 1980s

- **Sending rate only limited by flow control**
  - Dropped packets → senders (repeatedly!) retransmit
- **Led to "congestion collapse" in October 1986**
  - Throughput on the NSF network dropped from 32 Kbits/s to 40 bits/s

Some History: TCP in the 1980s

- **Sending rate only limited by flow control**
  - Dropped packets → senders (repeatedly!) retransmit
- **Led to "congestion collapse" in October 1986**
  - Throughput on the NSF network dropped from 32 Kbits/s to 40 bits/s
- **"Fixed" by Van Jacobson's development of TCP's congestion control (CC) algorithms**

Jacobson's Approach

Jacobson's Approach

- **Extend TCP's existing window-based protocol, but adapt the window size in response to congestion**
  - In addition to adjusting it in response to receiver window advertisements

Jacobson's Approach

- **Extend TCP's existing window-based protocol, but adapt the window size in response to congestion**
  - In addition to adjusting it in response to receiver window advertisements
- **A pragmatic and effective solution**
  - Required no upgrades to routers or applications
  - Patch of a few lines of code to TCP implementations

Jacobson's Approach

- **Extend TCP's existing window-based protocol, but adapt the window size in response to congestion**
  - In addition to adjusting it in response to receiver window advertisements
- **A pragmatic and effective solution**
  - Required no upgrades to routers or applications
  - Patch of a few lines of code to TCP implementations
- **Extensively researched and improved upon**
  - Especially now with datacenters and cloud services

Three Issues to Consider

Three Issues to Consider

- **Discovering** the available bottleneck

Three Issues to Consider

- **Discovering** the available bottleneck
- **Adjusting** to variations in bandwidth

Three Issues to Consider

- **Discovering** the available bottleneck
- **Adjusting** to variations in bandwidth
- **Sharing** bandwidth between flows

True Picture

True Picture

True Picture

Congestion control is a resource allocation problem involving many flows, many links, and complicated global dynamics!

Abstract View

Abstract View

- **Ignore internal structure of router and model it as a single queue for an input-output pair**

Discovering Available Bandwidth

- **Pick sending rate to match bottleneck bandwidth**
  - Without any a priori knowledge
  - Could be a gigabit link, could be a modem

Discovering Available Bandwidth

- **Pick sending rate to match bottleneck bandwidth**
  - Without any a priori knowledge
  - Could be a gigabit link, could be a modem

Discovering Available Bandwidth

- **Pick sending rate to match bottleneck bandwidth**
  - Without any a priori knowledge
  - Could be a gigabit link, could be a modem

Discovering Available Bandwidth

- **Pick sending rate to match bottleneck bandwidth**
  - Without any a priori knowledge
  - Could be a gigabit link, could be a modem

Discovering Available Bandwidth

- **Pick sending rate to match bottleneck bandwidth**
  - Without any a priori knowledge
  - Could be a gigabit link, could be a modem

Adjusting to Variations in Bandwidth

- **Adjust rate to match instantaneous bandwidth**
  - Assuming you have a rough idea of bandwidth

Adjusting to Variations in Bandwidth

- **Adjust rate to match instantaneous bandwidth**
  - Assuming you have a rough idea of bandwidth

Adjusting to Variations in Bandwidth

- **Adjust rate to match instantaneous bandwidth**
  - Assuming you have a rough idea of bandwidth

Adjusting to Variations in Bandwidth

- **Adjust rate to match instantaneous bandwidth**
  - Assuming you have a rough idea of bandwidth

Adjusting to Variations in Bandwidth

- **Adjust rate to match instantaneous bandwidth**
  - Assuming you have a rough idea of bandwidth

Multiple Flows & Sharing Bandwidth

- **Two issues:**
  - Collaboratively adjust total sending rate (across all flows) to match bandwidth
  - Allocation/division of bandwidth between flows

Multiple Flows & Sharing Bandwidth

- **Two issues:**
  - Collaboratively adjust total sending rate (across all flows) to match bandwidth
  - Allocation/division of bandwidth between flows

Multiple Flows & Sharing Bandwidth

- **Two issues:**
  - Collaboratively adjust total sending rate (across all flows) to match bandwidth
  - Allocation/division of bandwidth between flows

Multiple Flows & Sharing Bandwidth

- **Two issues:**
  - Collaboratively adjust total sending rate (across all flows) to match bandwidth
  - Allocation/division of bandwidth between flows

Multiple Flows & Sharing Bandwidth

- **Two issues:**
  - Collaboratively adjust total sending rate (across all flows) to match bandwidth
  - Allocation/division of bandwidth between flows

TCP's Approach in a Nutshell

TCP's Approach in a Nutshell

- **TCP connection has window**
  - Controls number of packets in flight

TCP's Approach in a Nutshell

- **TCP connection has window**
  - Controls number of packets in flight
- **Sending rate: ~WindowSize/RTT**

TCP's Approach in a Nutshell

- **TCP connection has window**
  - Controls number of packets in flight
- **Sending rate: ~WindowSize/RTT**
- **Vary window size to control sending rate**

So. Many. Windows.

So. Many. Windows.

- **Flow control window: Advertised Window (RWND)**
  - How many bytes can be sent without overflowing receiver's buffer
  - Determined by the receiver and reported to the sender

So. Many. Windows.

- **Flow control window: Advertised Window (RWND)**
  - How many bytes can be sent without overflowing receiver's buffer
  - Determined by the receiver and reported to the sender
- **Congestion Window: CWND**
  - How many bytes can be sent without overflowing router buffers
  - Computed by the sender using congestion control algorithm

So. Many. Windows.

- **Flow control window: Advertised Window (RWND)**
  - How many bytes can be sent without overflowing receiver's buffer
  - Determined by the receiver and reported to the sender
- **Congestion Window: CWND**
  - How many bytes can be sent without overflowing router buffers
  - Computed by the sender using congestion control algorithm
- **Sender-side Window = minimum{CWND, RWND}**
  - Assume for the rest of this lecture that RWND >> CWND

Note

Note

- **This lecture, we will talk about CWND in units of MSS**
  - (Recall MSS: Maximum Segment Size, the amount of payload in a TCP packet)
  - This is only for pedagogical purposes

Note

- **This lecture, we will talk about CWND in units of MSS**
  - (Recall MSS: Maximum Segment Size, the amount of payload in a TCP packet)
  - This is only for pedagogical purposes
- **Keep in mind that real implementations maintain CWND in bytes**

Two Basic Questions

Two Basic Questions

- **How does the sender detect congestion?**

Two Basic Questions

- **How does the sender detect congestion?**
- **How does the sender adjust its sending rate?**
  - To address three issues:
    - Finding available bottleneck bandwidth
    - Adjusting to bandwidth variations
    - Sharing bandwidth

Detecting Congestion

Detecting Congestion

- **Packet Delays**
  - Tricky: Noisy signal (delay often varies considerably)

Detecting Congestion

- **Packet Delays**
  - Tricky: Noisy signal (delay often varies considerably)
- **Routers tell end-hosts when they're congested**
  - Requires modifying the routers

Detecting Congestion

- **Packet Delays**
  - Tricky: Noisy signal (delay often varies considerably)
- **Routers tell end-hosts when they're congested**
  - Requires modifying the routers
- **Packet loss**
  - Fail-safe signal that TCP already must detect
  - Complication: non-congestive loss (e.g., checksum errors)

Not all Losses are Created Equal

Not all Losses are Created Equal

- **Duplicate ACKs: isolated loss**
  - Still getting ACKs

Not all Losses are Created Equal

- **Duplicate ACKs: isolated loss**
  - Still getting ACKs
- **Timeout: much more serious**
  - Not enough dupACKs
  - Must have suffered several losses

Not all Losses are Created Equal

- **Duplicate ACKs: isolated loss**
  - Still getting ACKs
- **Timeout: much more serious**
  - Not enough dupACKs
  - Must have suffered several losses
- **Will adjust rate differently for each case**

Rate Adjustment

Rate Adjustment

- **Basic structure:**
  - Upon receipt of ACK (of new data): increase rate
  - Upon detection of loss: decrease rate

Rate Adjustment

- **Basic structure:**
  - Upon receipt of ACK (of new data): increase rate
  - Upon detection of loss: decrease rate
- **How we increase/decrease the rate depends on the phase of congestion control we're in**
  - Discovering available bottleneck bandwidth, vs.,
  - Adjusting to bandwidth variations

Bandwidth Discovery with Slow Start

Bandwidth Discovery with Slow Start

- **Goal: estimate available bandwidth**
  - Start slow (for safety)
  - But ramp up quickly (for efficiency)

Bandwidth Discovery with Slow Start

- **Goal: estimate available bandwidth**
  - Start slow (for safety)
  - But ramp up quickly (for efficiency)
- **Consider**
  - RTT = 100ms, MSS = 1000B
  - Window size to fill 1Mbps of B/W = 12.5 packets
  - Window size to fill 1Gbps = 12,500 packets
  - Either is possible!

“Slow Start” Phase

“Slow Start” Phase

- **Sender starts at a slow rate, but increases *exponentially* until first loss**

“Slow Start” Phase

- **Sender starts at a slow rate, but increases *exponentially* until first loss**
- **Start with a small congestion window**
  - Initially, CWND=1
  - So initial sending rate is MSS/RTT

“Slow Start” Phase

- **Sender starts at a slow rate, but increases *exponentially* until first loss**
- **Start with a small congestion window**
  - Initially, CWND=1
  - So initial sending rate is MSS/RTT
- **Double the CWND for each RTT with no loss**

“Slow Start” in Action

“Slow Start” in Action

- **For each RTT:** double CWND

“Slow Start” in Action

- **For each RTT:** double CWND
- **Simpler implementation:** for each ACK, CWND += 1

“Slow Start” in Action

- **For each RTT:** double CWND
- **Simpler implementation:** for each ACK, CWND += 1

Linear increase per <u>ACK</u> (CWND+1) → exponential increase per <u>RTT</u> (2xCWND)

"Slow Start" in Action

- **For each RTT:** double CWND
- **Simpler implementation:** for each ACK, CWND += 1

"Slow Start" in Action

- **For each RTT:** double CWND
- **Simpler implementation:** for each ACK, CWND += 1

"Slow Start" in Action

- **For each RTT:** double CWND
- **Simpler implementation:** for each ACK, CWND += 1

"Slow Start" in Action

- **For each RTT:** double CWND
- **Simpler implementation:** for each ACK, CWND += 1

"Slow Start" in Action

- **For each RTT:** double CWND
- **Simpler implementation:** for each ACK, CWND += 1

"Slow Start" in Action

- **For each RTT:** double CWND
- **Simpler implementation:** for each ACK, CWND += 1

"Slow Start" in Action

- **For each RTT:** double CWND
- **Simpler implementation:** for each ACK, CWND += 1

"Slow Start" in Action

- **For each RTT:** double CWND
- **Simpler implementation:** for each ACK, CWND += 1

"Slow Start" in Action

- **For each RTT:** double CWND
- **Simpler implementation:** for each ACK, CWND += 1

"Slow Start" in Action

- **For each RTT:** double CWND
- **Simpler implementation:** for each ACK, CWND += 1

"Slow Start" in Action

- **For each RTT:** double CWND
- **Simpler implementation:** for each ACK, CWND += 1

"Slow Start" in Action

- **For each RTT:** double CWND
- **Simpler implementation:** for each ACK, CWND += 1

"Slow Start" in Action

- **For each RTT:** double CWND
- **Simpler implementation:** for each ACK, CWND += 1

"Slow Start" in Action

- **For each RTT:** double CWND
- **Simpler implementation:** for each ACK, CWND += 1

"Slow Start" in Action

- **For each RTT:** double CWND
- **Simpler implementation:** for each ACK, CWND += 1

"Slow Start" in Action

- **For each RTT:** double CWND
- **Simpler implementation:** for each ACK, CWND += 1

Adjusting to Varying Bandwidth

Adjusting to Varying Bandwidth

- **Slow start gave an estimate of available bandwidth**

Adjusting to Varying Bandwidth

- **Slow start gave an estimate of available bandwidth**
- **Now, want to track variations in this available bandwidth, oscillating around its current value**
  - Repeated probing (rate increase) and back-off (decrease)

Adjusting to Varying Bandwidth

- **Slow start gave an estimate of available bandwidth**
- **Now, want to track variations in this available bandwidth, oscillating around its current value**
  - Repeated probing (rate increase) and back-off (decrease)
- **TCP uses "Additive Increase Multiplicative Decrease" (AIMD)**
  - We'll see why shortly

AIMD

AIMD

- **Additive Increase**
  - Window grows by one MSS for every RTT with no loss
  - For each successful RTT, CWND = CWND + 1
  - Simpler implementation: for each ACK, CWND = CWND + 1/CWND

AIMD

- **Additive Increase**
  - Window grows by one MSS for every RTT with no loss
  - For each successful RTT, CWND = CWND + 1
  - Simpler implementation: for each ACK, CWND = CWND + 1/CWND
- **Multiplicative Decrease**
  - On loss of packet, divide congestion window in **<u>half</u>**
  - On loss, CWND = CWND / 2

Leads to the TCP "Sawtooth"

Leads to the TCP "Sawtooth"

Leads to the TCP "Sawtooth"

Leads to the TCP "Sawtooth"

Slow Start vs. AIMD

Slow Start vs. AIMD

- **When does a sender stop Slow Start and start Additive Increase?**

Slow Start vs. AIMD

- **When does a sender stop Slow Start and start Additive Increase?**
- **Introduce a "Slow Start Threshold" (ssthresh)**
  - Initialized to a large value
  - On timeout, ssthresh = CWND / 2

Slow Start vs. AIMD

- **When does a sender stop Slow Start and start Additive Increase?**
- **Introduce a "Slow Start Threshold" (ssthresh)**
  - Initialized to a large value
  - On timeout, ssthresh = CWND / 2
- **When CWND = ssthresh, sender switches from Slow Start to AIMD-style increase**

Why AIMD?

Recall: Three Issues

Recall: Three Issues

- **Discovering the available (bottleneck) bandwidth**
  - Slow Start

Recall: Three Issues

- **Discovering the available (bottleneck) bandwidth**
  - Slow Start
- **Adjusting to variations in bandwidth**
  - AIMD

Recall: Three Issues

- **Discovering the available (bottleneck) bandwidth**
  - Slow Start
- **Adjusting to variations in bandwidth**
  - AIMD
- **Sharing bandwidth between flows**

Goals for Bandwidth Sharing

Goals for Bandwidth Sharing

- **Efficiency:** High utilization of link bandwidth

Goals for Bandwidth Sharing

- **Efficiency:** High utilization of link bandwidth
- **Fairness:** Each flow gets equal share

Why AIMD?

Why AIMD?

- **Some rate adjustment options: Every RTT, we can:**
  - Multiplicative increase or decrease: **CWND ← a x CWND**
  - Additive increase or decrease: **CWND ← CWND + b**

Why AIMD?

- **Some rate adjustment options: Every RTT, we can:**
  - Multiplicative increase or decrease: **CWND ← a x CWND**
  - Additive increase or decrease: **CWND ← CWND + b**
- **Four alternatives:**
  - AIAD: gentle increase, gentle decrease
  - AIMD: gentle increase, drastic decrease
  - MIAD: drastic increase, gentle decrease
  - MIMD: drastic increase, drastic decrease

Simple Model of Congestion Control

Simple Model of Congestion Control

- **Two users**
  - Rates x1 and x2

Simple Model of Congestion Control

- **Two users**
  - Rates x1 and x2

Simple Model of Congestion Control

- **Two users**
  - Rates x1 and x2
- **Congestion when x1+x2 >1**

Simple Model of Congestion Control

- **Two users**
  - Rates x1 and x2
- **Congestion when x1+x2 >1**
- **Inefficient when x1+x2<1**

Simple Model of Congestion Control

- **Two users**
  - Rates x1 and x2
- **Congestion when x1+x2 >1**
- **Inefficient when x1+x2<1**
- **Fair when x1=x2**

Example

- **Two users**
  - Rates x1 and x2
- **Congestion when x1+x2 >1**
- **Inefficient when x1+x2<1**
- **Fair when x1=x2**

Example

- **Two users**
  - Rates x1 and x2
- **Congestion when x1+x2 >1**
- **Inefficient when x1+x2<1**
- **Fair when x1=x2**

Example

- **Two users**
  - Rates x1 and x2
- **Congestion when x1+x2 >1**
- **Inefficient when x1+x2<1**
- **Fair when x1=x2**

Example

- **Two users**
  - Rates x1 and x2
- **Congestion when x1+x2 >1**
- **Inefficient when x1+x2<1**
- **Fair when x1=x2**

Example

- **Two users**
  - Rates x1 and x2
- **Congestion when x1+x2 >1**
- **Inefficient when x1+x2<1**
- **Fair when x1=x2**

AIAD

- **Increase:** $x + a_I$
- **Decrease:** $x - a_D$

AIAD

- **Increase:** $x + a_I$
- **Decrease:** $x - a_D$

AIAD

- **Increase:** $x + a_I$
- **Decrease:** $x - a_D$

AIAD

- **Increase:** $x + a_I$
- **Decrease:** $x - a_D$

AIAD

- **Increase:** $x + a_I$
- **Decrease:** $x - a_D$
- **Does not converge to fairness!**

AIAD Sharing Dynamics

AIAD Sharing Dynamics

AIAD Sharing Dynamics

AIAD Sharing Dynamics

MIMD

- **Increase:** $x * b_I$
- **Decrease:** $x * b_D$

MIMD

- **Increase:** $x * b_I$
- **Decrease:** $x * b_D$

MIMD

- **Increase:** $x * b_I$
- **Decrease:** $x * b_D$

MIMD

- **Increase:** $x * b_I$
- **Decrease:** $x * b_D$

MIMD

- **Increase:** $x * b_I$
- **Decrease:** $x * b_D$
- **Does not converge to fairness!**

AIMD

- **Increase:** $x + a_I$
- **Decrease:** $x * b_D$

AIMD

- **Increase:** $x + a_I$
- **Decrease:** $x * b_D$

AIMD

- **Increase:** $x + a_I$
- **Decrease:** $x * b_D$

AIMD

- **Increase:** $x + a_I$
- **Decrease:** $x * b_D$

AIMD

- **Increase:** $x + a_I$
- **Decrease:** $x * b_D$

AIMD

- **Increase:** $x + a_I$
- **Decrease:** $x * b_D$

AIMD

- **Increase:** $x + a_I$
- **Decrease:** $x * b_D$
- **Converges to fairness!**

AIMD Sharing Dynamics

AIMD Sharing Dynamics

AIMD Sharing Dynamics

AIMD Sharing Dynamics

AIMD Sharing Dynamics

Rates equalize → Fair share

TCP Congestion Control Details

Implementation

Implementation

- **State at sender**
  - CWND (initialized to a small constant)
  - ssthresh (initialized to a large constant)

Implementation

- **State at sender**
  - CWND (initialized to a small constant)
  - ssthresh (initialized to a large constant)
- **Events:**
  - ACK (new data)
  - dupACK (duplicate ACK for old data)
  - Timeout

Event: ACK (new data)

- **If CWND < ssthresh**
  - CWND += 1

Event: ACK (new data)

- **If CWND < ssthresh**
  - CWND += 1

CWND packets per RTT
Hence after each RTT with no packet drops:
CWND = 2 x CWND

Event: ACK (new data)

- **If CWND < ssthresh**
  - CWND += 1

- **Else:**
  - CWND += 1/CWND

Event: ACK (new data)

- **If CWND < ssthresh**
  - CWND += 1

- **Else:**
  - CWND += 1/CWND

CWND packets per RTT
Hence after each RTT with no packet drops:
CWND = CWND + 1

Event: ACK (new data)

- **If CWND < ssthresh**
  - CWND += 1

Slow Start Phase

- **Else:**
  - CWND += 1/CWND

Congestion Avoidance Phase
(Additive Increase)

Event: Timeout

- **On timeout**
  - ssthresh ← CWND/2
  - CWND ← 1

Event: dupACK

- **dupACKcount++**
- **If dupACKcount = 3 /* Fast retransmit */**
  - ssthresh ← CWND/2
  - CWND ← CWND/2

Example

Example

Example

Example

Example

Timeout
Fast Retransmission
ssthresh set to here
Window Size
Time

Example

Slow Start restart: Go back to CWND=1, but take advantage of knowing the previous value of CWND

Example

Slow Start restart: Go back to CWND=1, but take advantage of knowing the previous value of CWND

Example

Slow Start restart: Go back to CWND=1, but take advantage of knowing the previous value of CWND

One Final Phase: Fast Recovery

- **The problem: congestion avoidance too slow in recovering from an isolated loss**

Example

- **Consider a TCP connection with:**
  - CWND = 10 packets
  - Last ACK was for packet # 101
    - i.e., receiver expecting next packet to have sequence number 101
- **10 packets [101, 102, 103, ..., 110] are in flight**
  - Packet 101 is dropped
  - What ACKs do they generate?
  - And how does the sender respond?

Timeline

Timeline

- ACK 101 (due to 102) cwnd=10 dupACK#1 (no xmit)

Timeline

- ACK 101 (due to 102) cwnd=10 dupACK#1 (no xmit)
- ACK 101 (due to 103) cwnd=10 dupACK#2 (no xmit)

Timeline

- ACK 101 (due to 102) cwnd=10 dupACK#1 (no xmit)
- ACK 101 (due to 103) cwnd=10 dupACK#2 (no xmit)
- ACK 101 (due to 104) cwnd=10 dupACK#3 (no xmit)

Timeline

- ACK 101 (due to 102) cwnd=10 dupACK#1 (no xmit)
- ACK 101 (due to 103) cwnd=10 dupACK#2 (no xmit)
- ACK 101 (due to 104) cwnd=10 dupACK#3 (no xmit)
- RETRANSMIT 101 ssthresh=5 cwnd= 5

Timeline

- ACK 101 (due to 102) cwnd=10 dupACK#1 (no xmit)
- ACK 101 (due to 103) cwnd=10 dupACK#2 (no xmit)
- ACK 101 (due to 104) cwnd=10 dupACK#3 (no xmit)
- RETRANSMIT 101 ssthresh=5 cwnd= 5
- ACK 101 (due to 105) cwnd=5 + 1/5 (no xmit)

Timeline

- ACK 101 (due to 102) cwnd=10 dupACK#1 (no xmit)
- ACK 101 (due to 103) cwnd=10 dupACK#2 (no xmit)
- ACK 101 (due to 104) cwnd=10 dupACK#3 (no xmit)
- RETRANSMIT 101 ssthresh=5 cwnd= 5
- ACK 101 (due to 105) cwnd=5 + 1/5 (no xmit)
- ACK 101 (due to 106) cwnd=5 + 2/5 (no xmit)

Timeline

- ACK 101 (due to 102) cwnd=10 dupACK#1 (no xmit)
- ACK 101 (due to 103) cwnd=10 dupACK#2 (no xmit)
- ACK 101 (due to 104) cwnd=10 dupACK#3 (no xmit)
- RETRANSMIT 101 ssthresh=5 cwnd= 5
- ACK 101 (due to 105) cwnd=5 + 1/5 (no xmit)
- ACK 101 (due to 106) cwnd=5 + 2/5 (no xmit)
- ACK 101 (due to 107) cwnd=5 + 3/5 (no xmit)

Timeline

- ACK 101 (due to 102) cwnd=10 dupACK#1 (no xmit)
- ACK 101 (due to 103) cwnd=10 dupACK#2 (no xmit)
- ACK 101 (due to 104) cwnd=10 dupACK#3 (no xmit)
- RETRANSMIT 101 ssthresh=5 cwnd= 5
- ACK 101 (due to 105) cwnd=5 + 1/5 (no xmit)
- ACK 101 (due to 106) cwnd=5 + 2/5 (no xmit)
- ACK 101 (due to 107) cwnd=5 + 3/5 (no xmit)
- ACK 101 (due to 108) cwnd=5 + 4/5 (no xmit)

Timeline

- ACK 101 (due to 102) cwnd=10 dupACK#1 (no xmit)
- ACK 101 (due to 103) cwnd=10 dupACK#2 (no xmit)
- ACK 101 (due to 104) cwnd=10 dupACK#3 (no xmit)
- RETRANSMIT 101 ssthresh=5 cwnd= 5
- ACK 101 (due to 105) cwnd=5 + 1/5 (no xmit)
- ACK 101 (due to 106) cwnd=5 + 2/5 (no xmit)
- ACK 101 (due to 107) cwnd=5 + 3/5 (no xmit)
- ACK 101 (due to 108) cwnd=5 + 4/5 (no xmit)
- ACK 101 (due to 109) cwnd=5 + 5/5 (no xmit)

Timeline

- ACK 101 (due to 102) cwnd=10 dupACK#1 (no xmit)
- ACK 101 (due to 103) cwnd=10 dupACK#2 (no xmit)
- ACK 101 (due to 104) cwnd=10 dupACK#3 (no xmit)
- RETRANSMIT 101 ssthresh=5 cwnd= 5
- ACK 101 (due to 105) cwnd=5 + 1/5 (no xmit)
- ACK 101 (due to 106) cwnd=5 + 2/5 (no xmit)
- ACK 101 (due to 107) cwnd=5 + 3/5 (no xmit)
- ACK 101 (due to 108) cwnd=5 + 4/5 (no xmit)
- ACK 101 (due to 109) cwnd=5 + 5/5 (no xmit)
- ACK 101 (due to 110) cwnd=6 + 1/5 (no xmit)

Timeline

- ACK 101 (due to 102) cwnd=10 dupACK#1 (no xmit)
- ACK 101 (due to 103) cwnd=10 dupACK#2 (no xmit)
- ACK 101 (due to 104) cwnd=10 dupACK#3 (no xmit)
- RETRANSMIT 101 ssthresh=5 cwnd= 5
- ACK 101 (due to 105) cwnd=5 + 1/5 (no xmit)
- ACK 101 (due to 106) cwnd=5 + 2/5 (no xmit)
- ACK 101 (due to 107) cwnd=5 + 3/5 (no xmit)
- ACK 101 (due to 108) cwnd=5 + 4/5 (no xmit)
- ACK 101 (due to 109) cwnd=5 + 5/5 (no xmit)
- ACK 101 (due to 110) cwnd=6 + 1/5 (no xmit)
- ACK 111 (due to 101) ← only now can we transmit new packets

Timeline

- ACK 101 (due to 102) cwnd=10 dupACK#1 (no xmit)
- ACK 101 (due to 103) cwnd=10 dupACK#2 (no xmit)
- ACK 101 (due to 104) cwnd=10 dupACK#3 (no xmit)
- RETRANSMIT 101 ssthresh=5 cwnd= 5
- ACK 101 (due to 105) cwnd=5 + 1/5 (no xmit)
- ACK 101 (due to 106) cwnd=5 + 2/5 (no xmit)
- ACK 101 (due to 107) cwnd=5 + 3/5 (no xmit)
- ACK 101 (due to 108) cwnd=5 + 4/5 (no xmit)
- ACK 101 (due to 109) cwnd=5 + 5/5 (no xmit)
- ACK 101 (due to 110) cwnd=6 + 1/5 (no xmit)
- ACK 111 (due to 101) ← only now can we transmit new packets
- Plus no packets in flight so ACK "clocking" (to increase CWND) stalls for another RTT

Solution: Fast Recovery

- **Idea: Grant the sender temporary "credit" for each dupACK so as to keep packets in flight**
- **If dupACKcount = 3**
  - ssthresh = CWND / 2
  - CWND = ssthresh + 3
- **While in fast recovery**
  - CWND = CWND + 1 for each additional duplicate ACK
- **Exit fast recovery after receiving new ACK**
  - Set CWND = ssthresh

Example

- **Consider a TCP connection with:**
  - CWND = 10 packets
  - Last ACK was for packet #101
    - i.e., receiver expecting next packet to have seqno 101
- **10 packets [101, 102, 103, ..., 110] are in flight**
  - Packet 101 is dropped

Timeline

Timeline

- ACK 101 (due to 102) cwnd=10 dupACK#1 (no xmit)

Timeline

- ACK 101 (due to 102) cwnd=10 dupACK#1 (no xmit)
- ACK 101 (due to 103) cwnd=10 dupACK#2 (no xmit)

Timeline

- ACK 101 (due to 102) cwnd=10 dupACK#1 (no xmit)
- ACK 101 (due to 103) cwnd=10 dupACK#2 (no xmit)
- ACK 101 (due to 104) cwnd=10 dupACK#3 (no xmit)

Timeline

- ACK 101 (due to 102) cwnd=10 dupACK#1 (no xmit)
- ACK 101 (due to 103) cwnd=10 dupACK#2 (no xmit)
- ACK 101 (due to 104) cwnd=10 dupACK#3 (no xmit)
- RETRANSMIT 101 ssthresh=5 cwnd= 5

Timeline

- ACK 101 (due to 102) cwnd=10 dupACK#1 (no xmit)
- ACK 101 (due to 103) cwnd=10 dupACK#2 (no xmit)
- ACK 101 (due to 104) cwnd=10 dupACK#3 (no xmit)
- RETRANSMIT 101 ssthresh=5 cwnd= 5
- ACK 101 (due to 105) cwnd=9 (no xmit)

Timeline

- ACK 101 (due to 102) cwnd=10 dupACK#1 (no xmit)
- ACK 101 (due to 103) cwnd=10 dupACK#2 (no xmit)
- ACK 101 (due to 104) cwnd=10 dupACK#3 (no xmit)
- RETRANSMIT 101 ssthresh=5 cwnd= 5
- ACK 101 (due to 105) cwnd=9 (no xmit)
- ACK 101 (due to 106) cwnd=10 (no xmit)

Timeline

- ACK 101 (due to 102) cwnd=10 dupACK#1 (no xmit)
- ACK 101 (due to 103) cwnd=10 dupACK#2 (no xmit)
- ACK 101 (due to 104) cwnd=10 dupACK#3 (no xmit)
- RETRANSMIT 101 ssthresh=5 cwnd= 5
- ACK 101 (due to 105) cwnd=9 (no xmit)
- ACK 101 (due to 106) cwnd=10 (no xmit)
- ACK 101 (due to 107) cwnd=11 (xmit 111)

Timeline

- ACK 101 (due to 102) cwnd=10 dupACK#1 (no xmit)
- ACK 101 (due to 103) cwnd=10 dupACK#2 (no xmit)
- ACK 101 (due to 104) cwnd=10 dupACK#3 (no xmit)
- RETRANSMIT 101 ssthresh=5 cwnd= 5
- ACK 101 (due to 105) cwnd=9 (no xmit)
- ACK 101 (due to 106) cwnd=10 (no xmit)
- ACK 101 (due to 107) cwnd=11 (xmit 111)
- ACK 101 (due to 108) cwnd=12 (xmit 112)

Timeline

- ACK 101 (due to 102) cwnd=10 dupACK#1 (no xmit)
- ACK 101 (due to 103) cwnd=10 dupACK#2 (no xmit)
- ACK 101 (due to 104) cwnd=10 dupACK#3 (no xmit)
- RETRANSMIT 101 ssthresh=5 cwnd= 5
- ACK 101 (due to 105) cwnd=9 (no xmit)
- ACK 101 (due to 106) cwnd=10 (no xmit)
- ACK 101 (due to 107) cwnd=11 (xmit 111)
- ACK 101 (due to 108) cwnd=12 (xmit 112)
- ACK 101 (due to 109) cwnd=13 (xmit 113)

Timeline

- ACK 101 (due to 102) cwnd=10 dupACK#1 (no xmit)
- ACK 101 (due to 103) cwnd=10 dupACK#2 (no xmit)
- ACK 101 (due to 104) cwnd=10 dupACK#3 (no xmit)
- RETRANSMIT 101 ssthresh=5 cwnd= 5
- ACK 101 (due to 105) cwnd=9 (no xmit)
- ACK 101 (due to 106) cwnd=10 (no xmit)
- ACK 101 (due to 107) cwnd=11 (xmit 111)
- ACK 101 (due to 108) cwnd=12 (xmit 112)
- ACK 101 (due to 109) cwnd=13 (xmit 113)
- ACK 101 (due to 110) cwnd=14 (xmit 114)

Timeline

- ACK 101 (due to 102) cwnd=10 dupACK#1 (no xmit)
- ACK 101 (due to 103) cwnd=10 dupACK#2 (no xmit)
- ACK 101 (due to 104) cwnd=10 dupACK#3 (no xmit)
- RETRANSMIT 101 ssthresh=5 cwnd= 5
- ACK 101 (due to 105) cwnd=9 (no xmit)
- ACK 101 (due to 106) cwnd=10 (no xmit)
- ACK 101 (due to 107) cwnd=11 (xmit 111)
- ACK 101 (due to 108) cwnd=12 (xmit 112)
- ACK 101 (due to 109) cwnd=13 (xmit 113)
- ACK 101 (due to 110) cwnd=14 (xmit 114)
- ACK 111 (due to 101) cwnd=5 (xmit 115) ← exiting fast recovery

Timeline

- ACK 101 (due to 102) cwnd=10 dupACK#1 (no xmit)
- ACK 101 (due to 103) cwnd=10 dupACK#2 (no xmit)
- ACK 101 (due to 104) cwnd=10 dupACK#3 (no xmit)
- RETRANSMIT 101 ssthresh=5 cwnd= 5
- ACK 101 (due to 105) cwnd=9 (no xmit)
- ACK 101 (due to 106) cwnd=10 (no xmit)
- ACK 101 (due to 107) cwnd=11 (xmit 111)
- ACK 101 (due to 108) cwnd=12 (xmit 112)
- ACK 101 (due to 109) cwnd=13 (xmit 113)
- ACK 101 (due to 110) cwnd=14 (xmit 114)
- ACK 111 (due to 101) cwnd=5 (xmit 115) ← exiting fast recovery
- Packets 111-114 already in flight

Timeline

- ACK 101 (due to 102) cwnd=10 dupACK#1 (no xmit)
- ACK 101 (due to 103) cwnd=10 dupACK#2 (no xmit)
- ACK 101 (due to 104) cwnd=10 dupACK#3 (no xmit)
- RETRANSMIT 101 ssthresh=5 cwnd= 5
- ACK 101 (due to 105) cwnd=9 (no xmit)
- ACK 101 (due to 106) cwnd=10 (no xmit)
- ACK 101 (due to 107) cwnd=11 (xmit 111)
- ACK 101 (due to 108) cwnd=12 (xmit 112)
- ACK 101 (due to 109) cwnd=13 (xmit 113)
- ACK 101 (due to 110) cwnd=14 (xmit 114)
- ACK 111 (due to 101) cwnd=5 (xmit 115) ← exiting fast recovery
- Packets 111-114 already in flight
- ACK 112 (due to 111) cwnd=5 + 1/5 ← back in congestion avoidance

TCP State Machine

TCP State Machine

TCP State Machine

TCP State Machine

TCP Flavors

TCP Flavors

- **TCP Tahoe**
  - CWND=1 on triple dupACK

TCP Flavors

- **TCP Tahoe**
  - CWND=1 on triple dupACK
- **TCP Reno**
  - CWND = 1 on timeout
  - CWND = CWND / 2 on triple dupACK

TCP Flavors

- **TCP Tahoe**
  - CWND=1 on triple dupACK
- **TCP Reno**
  - CWND = 1 on timeout
  - CWND = CWND / 2 on triple dupACK
- **TCP newReno**
  - TCP Reno + improved fast recovery

TCP Flavors

- **TCP Tahoe**
  - CWND=1 on triple dupACK
- **TCP Reno**
  - CWND = 1 on timeout
  - CWND = CWND / 2 on triple dupACK
- **TCP newReno**
  - TCP Reno + improved fast recovery
- **TCP SACK**
  - Incorporates selective acknowledgements

Taking Stock

- The concepts underlying TCP are simple
  - Acknowledgements
  - Timers
  - Sliding Windows
  - Buffer Management
  - Sequence Numbers

Taking Stock

- The concepts underlying TCP are simple
- But tricky in the details
  - How do we set timers
  - What is the seqno for an ACK only packet
  - What happens if the advertised window = 0
  - What if the advertised window is 1/2 an MSS
  - Should receiver acknowledge packets right away
  - What if the application generates data in units of 0.1 MSS
  - What happens if I get a duplicate SYN? Or an RST while I'm in FIN_WAIT?
  - *etc., etc., etc.*

Taking Stock

- The concepts underlying TCP are simple
- But tricky in the details
- Do the details matter?