Transport Layer

- **Function:**
  - Demultiplexing of data streams
- **Optional functions:**
  - Creating long lived connections
  - Reliable, in-order packet delivery
  - Error detection
  - Flow and congestion control
- **Key challenges:**
  - Detecting and responding to congestion
  - Balancing fairness against high utilization
Outline

- Congestion Control
- Evolution of TCP
- Problems with TCP
What is Congestion?

- Load on the network is higher than capacity
What is Congestion?

- Load on the network is higher than capacity
- Capacity is not uniform across networks
  - Modem vs. Cellular vs. Cable vs. Fiber Optics
- There are multiple flows competing for bandwidth
  - Residential cable modem vs. corporate datacenter
- Load is not uniform over time
  - 10pm, Sunday night = Bittorrent Game of Thrones
Why is Congestion Bad?

- Results in packet **loss**
  - Routers have finite buffers
  - Internet traffic is self similar, no buffer can prevent all drops
  - When routers get overloaded, packets will be dropped

- Practical consequences
  - Router queues build up, **delay** increases
  - Wasted bandwidth from **retransmissions**
  - Low network goodput
The Danger of Increasing Load

- Knee – point after which
  - Throughput increases very slow
  - Delay increases fast
- In an M/M/1 queue
  - Delay = \( \frac{1}{1 - \text{utilization}} \)
- Cliff – point after which
  - Throughput \( \to 0 \)
  - Delay \( \to \infty \)
The Danger of Increasing Load

- Knee – point after which
  - Throughput increases very slow
  - Delay increases fast
- In an $M/M/1$ queue
  - Delay $= \frac{1}{1 - \text{utilization}}$
- Cliff – point after which
  - Throughput $\to 0$
  - Delay $\to \infty$
The Danger of Increasing Load

- **Knee** – point after which
  - Throughput increases very slow
  - Delay increases fast
- **Cliff** – point after which
  - Throughput $\to 0$
  - Delay $\to \infty$

In an M/M/1 queue
- Delay $= \frac{1}{1 - \text{utilization}}$
Cong. Control vs. Cong. Avoidance

- Congestion
- Collapse
- Goodput
- Knee
- Cliff
- Load

Graph showing Goodput vs. Load with three stages: Knee, Cliff, and Congestion Collapse.
Congestion Control vs. Congestion Avoidance

Congestion Avoidance: Stay left of the knee

Load vs. Goodput graph with 'Knee' and 'Cliff' regions.
Congestion Avoidance: Stay left of the knee

Congestion Control: Stay left of the cliff

Congestion Collapse
Advertised Window, Revisited

- Does TCP’s advertised window solve congestion?
Advertised Window, Revisited

- Does TCP’s advertised window solve congestion? **NO**
- The advertised window only protects the receiver
- A sufficiently fast receiver can max the window
  - What if the network is slower than the receiver?
  - What if there are other concurrent flows?
Advertised Window, Revisited

- Does TCP’s advertised window solve congestion? **NO**
- The advertised window only protects the receiver
- A sufficiently fast receiver can max the window
  - What if the network is slower than the receiver?
  - What if there are other concurrent flows?
- Key points
  - Window size determines send rate
  - Window must be adjusted to prevent congestion collapse
Goals of Congestion Control
Goals of Congestion Control

1. Adjusting to the bottleneck bandwidth
2. Adjusting to variations in bandwidth
3. Sharing bandwidth between flows
4. Maximizing throughput
General Approaches

- Do nothing, send packets indiscriminately
  - Many packets will drop, totally unpredictable performance
  - May lead to congestion collapse
General Approaches

- Do nothing, send packets indiscriminately
  - Many packets will drop, totally unpredictable performance
  - May lead to congestion collapse

- Reservations
  - Pre-arrange bandwidth allocations for flows
  - Requires negotiation before sending packets
  - Must be supported by the network
General Approaches

- Do nothing, send packets indiscriminately
  - Many packets will drop, totally unpredictable performance
  - May lead to congestion collapse

- Reservations
  - Pre-arrange bandwidth allocations for flows
  - Requires negotiation before sending packets
  - Must be supported by the network

- Dynamic adjustment
  - Use probes to estimate level of congestion
  - Speed up when congestion is low
  - Slow down when congestion increases
  - Messy dynamics, requires distributed coordination
General Approaches

- Do nothing, send packets indiscriminately
  - Many packets will drop, totally unpredictable performance
  - May lead to congestion collapse
- Reservations
  - Pre-arrange bandwidth allocations for flows
  - Requires negotiation before sending packets
  - Must be supported by the network
- Dynamic adjustment
  - Use probes to estimate level of congestion
  - Speed up when congestion is low
  - Slow down when congestion increases
  - Messy dynamics, requires distributed coordination
TCP Congestion Control

- Each TCP connection has a window
  - Controls the number of unACKEd packets
- Sending rate is $\sim \text{window}/\text{RTT}$
- Idea: vary the window size to control the send rate
TCP Congestion Control

- Each TCP connection has a window
  - Controls the number of unACKed packets
- Sending rate is \( \sim \) window/RTT
- Idea: vary the window size to control the send rate
- Introduce a *congestion window* at the sender
  - Congestion control is sender-side problem
Congestion Window (cwnd)

- Limits how much data is in transit
- Denominated in bytes

1. \( \text{wnd} = \min(\text{cwnd}, \text{adv\_wnd}) \);
2. \( \text{effective\_wnd} = \text{wnd} - (\text{last\_byte\_sent} - \text{last\_byte\_acked}) \);
Congestion Window (cwnd)

- Limits how much data is in transit
- Denominated in bytes

1. \( \text{wnd} = \min(\text{cwnd}, \text{adv}_\text{wnd}); \)
2. \( \text{effective}_\text{wnd} = \text{wnd} - (\text{last}_\text{byte}_\text{sent} - \text{last}_\text{byte}_\text{acked}); \)
Congestion Window (cwnd)

- Limits how much data is in transit
- Denominated in bytes

1. \( \text{wnd} = \min(\text{cwnd}, \text{adv}_\text{wnd}) \);
2. \( \text{effective}_\text{wnd} = \text{wnd} - (\text{last}_\text{byte}_\text{sent} - \text{last}_\text{byte}_\text{acked}) \);
Two Basic Components

1. Detect congestion
Two Basic Components

1. Detect congestion
   - Packet dropping is most reliably signal
     - Delay-based methods are hard and risky
   - How do you detect packet drops? ACKs
     - Timeout after not receiving an ACK
     - Several duplicate ACKs in a row (ignore for now)
Two Basic Components

1. Detect congestion
   - Packet dropping is most reliably signal
     - Delay-based methods are hard and risky
   - How do you detect packet drops? ACKs
     - Timeout after not receiving an ACK
     - Several duplicate ACKs in a row (ignore for now)

Except on wireless networks
Two Basic Components

1. Detect congestion
   - Packet dropping is most reliably signal
     - Delay-based methods are hard and risky
   - How do you detect packet drops? ACKs
     - Timeout after not receiving an ACK
     - Several duplicate ACKs in a row (ignore for now)

2. Rate adjustment algorithm
   - Modify cwnd
   - Probe for bandwidth
   - Responding to congestion

Except on wireless networks
Rate Adjustment

- Recall: TCP is ACK clocked
  - Congestion = delay = long wait between ACKs
  - No congestion = low delay = ACKs arrive quickly
Rate Adjustment

- Recall: TCP is ACK clocked
  - Congestion = delay = long wait between ACKs
  - No congestion = low delay = ACKs arrive quickly

- Basic algorithm
  - Upon receipt of ACK: increase cwnd
    - Data was delivered, perhaps we can send faster
    - cwnd growth is proportional to RTT
  - On loss: decrease cwnd
    - Data is being lost, there must be congestion
Rate Adjustment

- Recall: TCP is ACK clocked
  - Congestion = delay = long wait between ACKs
  - No congestion = low delay = ACKs arrive quickly

- Basic algorithm
  - Upon receipt of ACK: increase cwnd
    - Data was delivered, perhaps we can send faster
    - cwnd growth is proportional to RTT
  - On loss: decrease cwnd
    - Data is being lost, there must be congestion

- Question: increase/decrease functions to use?
Utilization and Fairness

Flow 2 Throughput

Flow 1 Throughput
Utilization and Fairness

Flow 1 Throughput

Flow 2 Throughput
Utilization and Fairness

Flow 1 Throughput

Flow 2 Throughput
Utilization and Fairness

Flow 1 Throughput

Flow 2 Throughput

Max throughput for flow 2

Zero throughput for flow 1
Utilization and Fairness

Flow 1 Throughput

Max throughput for flow 1

Flow 2 Throughput

Zero throughput for flow 2
Utilization and Fairness

- Flow 1 Throughput
- Flow 2 Throughput

Less than full utilization
Utilization and Fairness

Less than full utilization

More than full utilization (congestion)
Utilization and Fairness

Equal throughput (fairness)
Utilization and Fairness

Flow 1 Throughput

Flow 2 Throughput

Ideal point
- Max efficiency
- Perfect fairness
Multiplicative Increase, Additive Decrease
Multiplicative Increase, Additive Decrease
Multiplicative Increase, Additive Decrease
Multiplicative Increase, Additive Decrease

Flow 1 Throughput

Flow 2 Throughput
Multiplicative Increase, Additive Decrease

Flow 2 Throughput

Flow 1 Throughput
Multiplicative Increase, Additive Decrease

- Not stable!
Multiplicative Increase, Additive Decrease

- Not stable!
- Veers away from fairness
Additive Increase, Additive Decrease
Additive Increase, Additive Decrease
Additive Increase, Additive Decrease

Flow 1 Throughput vs. Flow 2 Throughput diagram with points indicating the relationship between the two flows.
Additive Increase, Additive Decrease

- Stable

Diagram:
- Two axes: Flow 1 Throughput and Flow 2 Throughput.
- Two points on the graph indicating stable conditions.
Additive Increase, Additive Decrease

- Stable
- But does not converge to fairness
Multiplicative Increase, Multiplicative Decrease

Flow 1 Throughput

Flow 2 Throughput
Multiplicative Increase, Multiplicative Decrease

Flow 1 Throughput

Flow 2 Throughput
Multiplicative Increase, Multiplicative Decrease

Flow 2 Throughput

Flow 1 Throughput
Multiplicative Increase, Multiplicative Decrease

- Stable
Multiplicative Increase, Multiplicative Decrease

- Stable
- But does not converge to fairness
Additive Increase, Multiplicative Decrease
Additive Increase, Multiplicative Decrease
Additive Increase, Multiplicative Decrease
Additive Increase, Multiplicative Decrease
Additive Increase, Multiplicative Decrease

- Converges to stable and fair cycle
Additive Increase, Multiplicative Decrease

- Converges to stable and fair cycle
Additive Increase, Multiplicative Decrease

- Converges to stable and fair cycle
- Symmetric around $y = x$
Implementing Congestion Control

- Maintains three variables:
  - cwnd: congestion window
  - adv_wnd: receiver advertised window
  - ssthresh: threshold size (used to update cwnd)

- For sending, use: $\text{wnd} = \min(cwnd, adv\_wnd)$
Implementing Congestion Control

- Maintains three variables:
  - cwnd: congestion window
  - adv wnd: receiver advertised window
  - ssthresh: threshold size (used to update cwnd)
- For sending, use: \( \text{wnd} = \min(\text{cwnd}, \text{adv}_\text{wnd}) \)
- Two phases of congestion control
  1. Slow start (cwnd < ssthresh)
     - Probe for bottleneck bandwidth
  2. Congestion avoidance (cwnd \( \geq \) ssthresh)
     - AIMD
Slow Start

- Goal: reach knee quickly
- Upon starting (or restarting) a connection:
  - \( cwnd = 1 \)
  - \( sssthresh = adv\_wnd \)
  - Each time a segment is ACKed, \( cwnd++ \)
**Slow Start**

- **Goal:** reach knee quickly
- **Upon starting (or restarting) a connection**
  - $cwnd = 1$
  - $ssthresh = adv_wnd$
  - Each time a segment is ACKed, $cwnd++$
- **Continues until...**
  - $ssthresh$ is reached
  - Or a packet is lost
Slow Start

- Goal: reach knee quickly
- Upon starting (or restarting) a connection:
  - $cwnd = 1$
  - $ssthresh = adv_wnd$
  - Each time a segment is ACKed, $cwnd++$
- Continues until...
  - $ssthresh$ is reached
  - Or a packet is lost
- Slow Start is not actually slow
  - $cwnd$ increases exponentially
Slow Start Example

cwnd = 1
Slow Start Example

cwnd = 1

cwnd = 2
Slow Start Example

cwnd = 1

1

cwnd = 2

2
3

cwnd = 4
Slow Start Example

cwnd = 1
cwnd = 2
cwnd = 4
cwnd = 8
- cwnd grows rapidly
- Slows down when...
  - cwnd $\geq$ ssthresh
  - Or a packet drops
Congestion Avoidance

- AIMD mode
- `ssthresh` is lower-bound guess about location of the knee

**If** \( cwnd \geq ssthresh \) **then**
  
  each time a segment is ACKed
  
  increment \( cwnd \) by \( 1/cwnd \) (\( cwnd += 1/cwnd \)).

- So \( cwnd \) is increased by one only if all segments have been acknowledged
Congestion Avoidance Example

**Round Trip Times**

- $t=0$
- $t=1$
- $t=2$
- $t=3$
- $t=4$
- $t=5$
- $t=6$
- $t=7$

**cwnd** (in segments):
- $cwnd = 1$
- $cwnd = 2$
- $cwnd = 4$
- $cwnd = 8$
- $cwnd = 9$

**ssthresh** = 8
Congestion Avoidance Example

**Round Trip Times**

- $cwnd = 1$
- $cwnd = 2$
- $cwnd = 4$
- $cwnd = 8$
- $cwnd = 9$

**Slow Start**

$ssthresh = 8$
Congestion Avoidance Example

$cwnd \geq ssthresh$

$ssthresh = 8$

$t=0$ $t=1$ $t=2$ $t=3$ $t=4$ $t=5$ $t=6$ $t=7$

$cwnd = 1$
$cwnd = 2$
$cwnd = 4$
$cwnd = 8$
$cwnd = 9$

Round Trip Times
TCP Pseudocode

Initially:
    cwnd = 1;
    ssthresh = adv_wnd;

New ack received:
    if (cwnd < ssthresh)
        /* Slow Start*/
        cwnd = cwnd + 1;
    else
        /* Congestion Avoidance */
        cwnd = cwnd + 1/cwnd;

Timeout:
    /* Multiplicative decrease */
    ssthresh = cwnd / 2;
    cwnd = 1;
The Big Picture

\[ \text{cwnd} \]

\[ \text{ssthresh} \]

Time
The Big Picture

slow start

$cwnd$

$ssthresh$

Time
The Big Picture

- Timeout
- Slow Start
- ssthresh

Diagram shows the relationship between congestion window (cwnd) and time.
The Big Picture

- **cwnd**: Horizontal axis representing time.
- **ssthresh**: Vertical axis representing the congestion window (cwnd).
- **Slow Start**: Initial phase of TCP where the sender starts transmitting with a small window.
- **Timeout**: Point where the sender timeouts and reduces the window size.

Graph illustrates the TCP communication process with a focus on the relationship between cwnd and time, highlighting slow start and timeout events.
The Big Picture

- Slow Start
- Timeout
- Congestion Avoidance

$ssthresh$
The Big Picture

- Slow Start
- Timeout
- Congestion Avoidance
- ssthresh
The Big Picture

- **Slow Start**
  - cwnd: In this phase, the window of the congestion window (cwnd) grows slowly.

- **Timeout**
  - ssthresh: Indicates the point at which the congestion window drops to a lower threshold.

- **Congestion Avoidance**
  - The phase where the window size is adjusted to avoid further congestion.

- **Time**
  - The x-axis represents time, which increases as the graph progresses.
Outline

- Congestion Control
- Evolution of TCP
- Problems with TCP
The Evolution of TCP

- Thus far, we have discussed TCP Tahoe
  - Original version of TCP
- However, TCP was invented in 1974!
  - Today, there are many variants of TCP
The Evolution of TCP

- Thus far, we have discussed TCP Tahoe
  - Original version of TCP
- However, TCP was invented in 1974!
  - Today, there are many variants of TCP
- Early, popular variant: TCP Reno
  - Tahoe features, plus...
  - Fast retransmit
  - Fast recovery
TCP Reno: Fast Retransmit

- Problem: in Tahoe, if segment is lost, there is a long wait until the RTO
- Reno: retransmit after 3 duplicate ACKs

![Diagram showing TCP Reno with cwnd values: cwnd = 1, cwnd = 2, cwnd = 4]
TCP Reno: Fast Retransmit

- Problem: in Tahoe, if segment is lost, there is a long wait until the RTO
- Reno: retransmit after 3 duplicate ACKs

\[ \text{cwnd} = 1 \]
\[ \text{cwnd} = 2 \]
\[ \text{cwnd} = 4 \]
TCP Reno: Fast Recovery

- After a fast-retransmit set cwnd to ssthresh/2
  - i.e. don’t reset cwnd to 1
  - Avoid unnecessary return to slow start
  - Prevents expensive timeouts
- But when RTO expires still do cwnd = 1
  - Return to slow start, same as Tahoe
  - Indicates packets aren’t being delivered at all
  - i.e. congestion must be really bad
Fast Retransmit and Fast Recovery

\[ \text{cwnd} \]

\[ \text{ssthresh} \]

\[ \text{Time} \]
Fast Retransmit and Fast Recovery

Slow Start

Time

cwnd

$ssthresh$
Fast Retransmit and Fast Recovery

![Graph showing Fast Retransmit and Fast Recovery](image)

- **cwnd**: Congestion Window
- **ssthresh**: Slow Start Threshold
- **Slow Start**: Initial phase of congestion control
- **Timeout**: Event causing fast retransmit

The graph illustrates the behavior of the congestion window (cwnd) over time (x-axis) as it grows exponentially during the slow start phase and remains steady until a timeout occurs, triggering fast retransmit and fast recovery protocols.
Fast Retransmit and Fast Recovery

- **cwnd**: Slow Start
- **ssthresh**: Timeout
- **Time**: Time
Fast Retransmit and Fast Recovery

- Slow Start
- Timeout
- Congestion Avoidance
- Fast Retransmit/Recovery

$ssthresh$
Fast Retransmit and Fast Recovery
At steady state, $cwnd$ oscillates around the optimal window size.
At steady state, $cwnd$ oscillates around the optimal window size.

TCP always forces packet drops.
Many TCP Variants...

- Tahoe: the original
  - Slow start with AIMD
  - Dynamic RTO based on RTT estimate
- Reno: fast retransmit and fast recovery
Many TCP Variants...

- Tahoe: the original
  - Slow start with AIMD
  - Dynamic RTO based on RTT estimate
- Reno: fast retransmit and fast recovery
- NewReno: improved fast retransmit
  - Each duplicate ACK triggers a retransmission
  - Problem: >3 out-of-order packets causes pathological retransmissions
Many TCP Variants...

- Tahoe: the original
  - Slow start with AIMD
  - Dynamic RTO based on RTT estimate
- Reno: fast retransmit and fast recovery
- NewReno: improved fast retransmit
  - Each duplicate ACK triggers a retransmission
  - Problem: >3 out-of-order packets causes pathological retransmissions
- Vegas: delay-based congestion avoidance
Many TCP Variants...

- Tahoe: the original
  - Slow start with AIMD
  - Dynamic RTO based on RTT estimate
- Reno: fast retransmit and fast recovery
- NewReno: improved fast retransmit
  - Each duplicate ACK triggers a retransmission
  - Problem: >3 out-of-order packets causes pathological retransmissions
- Vegas: delay-based congestion avoidance
- And many, many, many more…
TCP in the Real World

- What are the most popular variants today?
  - Key problem: TCP performs poorly on high bandwidth-delay product networks (like the modern Internet)
  - Compound TCP (Windows)
    - Based on Reno
    - Uses two congestion windows: delay based and loss based
    - Thus, it uses a compound congestion controller
  - TCP CUBIC (Linux)
    - Enhancement of BIC (Binary Increase Congestion Control)
    - Window size controlled by cubic function
    - Parameterized by the time $T$ since the last dropped packet
High Bandwidth-Delay Product

- Key Problem: TCP performs poorly when
  - The capacity of the network (bandwidth) is large
  - The delay (RTT) of the network is large
  - Or, when bandwidth * delay is large
    - \( b \times d = \) maximum amount of in-flight data in the network
    - a.k.a. the bandwidth-delay product
High Bandwidth-Delay Product

- Key Problem: TCP performs poorly when
  - The capacity of the network (bandwidth) is large
  - The delay (RTT) of the network is large
  - Or, when bandwidth * delay is large
    - \( b \times d = \text{maximum amount of in-flight data in the network} \)
    - a.k.a. the bandwidth-delay product

- Why does TCP perform poorly?
  - Slow start and additive increase are slow to converge
  - TCP is ACK clocked
    - i.e. TCP can only react as quickly as ACKs are received
    - Large RTT \( \rightarrow \) ACKs are delayed \( \rightarrow \) TCP is slow to react
Poor Performance of TCP Reno CC

Buffer = BW x Delay
RTT = 80 ms

50 flows in both directions

Buffer = BW x Delay
BW = 155 Mb/s
Goals

- Fast window growth
  - Slow start and additive increase are too slow when bandwidth is large
  - Want to converge more quickly
Goals

- Fast window growth
  - Slow start and additive increase are too slow when bandwidth is large
  - Want to converge more quickly
- Maintain fairness with other TCP variants
  - Window growth cannot be too aggressive
Goals

- Fast window growth
  - Slow start and additive increase are too slow when bandwidth is large
  - Want to converge more quickly
- Maintain fairness with other TCP variants
  - Window growth cannot be too aggressive
- Improve RTT fairness
  - TCP Tahoe/Reno flows are not fair when RTTs vary widely
Goals

- Fast window growth
  - Slow start and additive increase are too slow when bandwidth is large
  - Want to converge more quickly
- Maintain fairness with other TCP variants
  - Window growth cannot be too aggressive
- Improve RTT fairness
  - TCP Tahoe/Reno flows are not fair when RTTs vary widely
- Simple implementation
Compound TCP Implementation

- Default TCP implementation in Windows
- Key idea: split cwnd into two separate windows
  - Traditional, loss-based window
  - New, delay-based window
Default TCP implementation in Windows

Key idea: split $cwnd$ into two separate windows
- Traditional, loss-based window
- New, delay-based window

$wnd = \min(cwnd + dwnd, adv_wnd)$
- $cwnd$ is controlled by AIMD
- $dwnd$ is the delay window
Default TCP implementation in Windows

Key idea: split cwnd into two separate windows

- Traditional, loss-based window
- New, delay-based window

\[ \text{wnd} = \min(\text{cwnd} + \text{dwnd}, \text{adv}_\text{wnd}) \]

- cwnd is controlled by AIMD
- dwnd is the delay window

Rules for adjusting d wnd:

- If RTT is increasing, decrease d wnd (dwnd \( \geq 0 \))
- If RTT is decreasing, increase d wnd
- Increase/decrease are proportional to the rate of change
Compound TCP Example
Compound TCP Example
Compound TCP Example

Diagram showing the relationship between cwnd and time, with stages labeled "Slow Start" and "Timeout."
Compound TCP Example

- Slow Start
- Timeout
- High RTT
Compound TCP Example

- Slow Start
- Timeout
- Slower cwnd growth
- High RTT
Compound TCP Example

- **Slow Start**
- **Timeout**
- **Slower cwnd growth**
- **High RTT**
Compound TCP Example

- Slow Start
- Timeout
- Slower cwnd growth
- High RTT
- Low RTT
Compound TCP Example

- Slow Start
- Timeout
- Slower cwnd growth
- High RTT
- Faster cwnd growth
- Low RTT

Graph shows the relationship between cwnd and time, highlighting different stages of TCP operation.
Compound TCP Example

- **Slow Start**
- **Timeout**
- **Slower cwnd growth**
- **High RTT**
- **Faster cwnd growth**
- **Low RTT**
- **Timeout**
Aggressiveness corresponds to changes in RTT
 Compound TCP Example

- Aggressiveness corresponds to changes in RTT
- Advantages: fast ramp up, more fair to flows with different RTTs
Aggressiveness corresponds to changes in RTT

Advantages: fast ramp up, more fair to flows with different RTTs

Disadvantage: must estimate RTT, which is very challenging
TCP CUBIC Implementation

- Default TCP implementation in Linux
- Replace AIMD with cubic function

$$W_{cubic} = C(T - K)^3 + W_{max}$$  \hspace{1cm} (1)

- $C$ is a scaling constant, and $K = \sqrt[3]{\frac{W_{max}}{C} \beta}$
- $B \rightarrow$ a constant fraction for multiplicative increase
- $T \rightarrow$ time since last packet drop
- $W_{max} \rightarrow$ cwnd when last packet dropped
TCP CUBIC Example

Time

cwnd
TCP CUBIC Example

![Diagram showing TCP CUBIC example with slow start, timeout, and cwnd_max]
TCP CUBIC Example

- Slow Start
- Timeout
- $cwnd_{max}$
TCP CUBIC Example

Time

cwnd

Slow Start

Timeout

$\text{cwnd}_{\text{max}}$

Fast ramp up
TCP CUBIC Example

- **Timeout**
- **cwnd_{max}**
- **Slow Start**
- **Fast ramp up**
- **Stable Region**
TCP CUBIC Example

- **Slow Start**: Initially, the congestion window (cwnd) grows slowly.
- **Fast ramp up**: The cwnd increases rapidly to find the maximum sustainable rate.
- **Stable Region**: The cwnd reaches its maximum (cwnd_max) and stabilizes, slowly accelerating to probe for bandwidth.
- **Timeout**: If no acknowledgments are received, the connection is timed out.

Slowly accelerate to probe for bandwidth.
TCP CUBIC Example

CUBIC Function

Timeout

Slow Start

cwnd

Time

\( cwnd_{\text{max}} \)
TCP CUBIC Example

CUBIC Function

\[ cwnd_{\text{max}} \]

Time

Slow Start

Timeout

\( cwnd \)
TCP CUBIC Example

Time

Slow Start

Timeout

CUBIC Function

$\text{cwnd}_{\text{max}}$

\text{cwnd}
TCP CUBIC Example

- **Slow Start**
- **Timeout**
- **CUBIC Function**
- \( cwnd_{max} \)
TCP CUBIC Example

CUBIC Function

Timeout

Slow Start

\( cwnd_{\text{max}} \)
TCP CUBIC Example

- Less wasted bandwidth due to fast ramp up
TCP CUBIC Example

- Less wasted bandwidth due to fast ramp up
- Stable region and slow acceleration help maintain fairness
  - Fast ramp up is more aggressive than additive increase
  - To be fair to Tahoe/Reno, CUBIC needs to be less aggressive
Simulations of CUBIC Flows
Simulations of CUBIC Flows
Deploying TCP Variants

- TCP assumes all flows employ TCP-like congestion control
  - TCP-friendly or TCP-compatible
  - Violated by UDP :(
Deploying TCP Variants

- TCP assumes all flows employ TCP-like congestion control
  - TCP-friendly or TCP-compatible
  - Violated by UDP 😞
- If new congestion control algorithms are developed, they must be TCP-friendly
Deploying TCP Variants

- TCP assumes all flows employ TCP-like congestion control
  - TCP-friendly or TCP-compatible
  - Violated by UDP :( 
- If new congestion control algorithms are developed, they must be TCP-friendly
- Be wary of unforeseen interactions
  - Variants work well with others like themselves
  - Different variants competing for resources may trigger unfair, pathological behavior
TCP Perspectives

- Cerf/Kahn
  - Provide flow control
  - Congestion handled by retransmission
TCP Perspectives

- Cerf/Kahn
  - Provide flow control
  - Congestion handled by retransmission

- Jacobson / Karels
  - Need to avoid congestion
  - RTT estimates critical
  - Queuing theory can help
TCP Perspectives

- **Cerf/Kahn**
  - Provide flow control
  - Congestion handled by retransmission

- **Jacobson / Karels**
  - Need to avoid congestion
  - RTT estimates critical
  - Queuing theory can help

- **Winstein/Balakrishnan**
  - TCP is maximizing an objective function
    - Fairness/efficiency
    - Throughput/delay
  - Let a machine pick the best fit for your environment
Outline

- Congestion Control
- Evolution of TCP
- Problems with TCP
## Common TCP Options

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source Port</td>
<td>0-4</td>
</tr>
<tr>
<td>Destination Port</td>
<td>4-16</td>
</tr>
<tr>
<td>Sequence Number</td>
<td>16-23</td>
</tr>
<tr>
<td>Acknowledgement Number</td>
<td>23-29</td>
</tr>
<tr>
<td>HLen</td>
<td>29-31</td>
</tr>
<tr>
<td>Flags</td>
<td>31-31</td>
</tr>
<tr>
<td>Advertised Window</td>
<td>0-4</td>
</tr>
<tr>
<td>Urgent Pointer</td>
<td>4-16</td>
</tr>
<tr>
<td>Checksum</td>
<td>16-23</td>
</tr>
<tr>
<td>Options</td>
<td>23-31</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Options</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Options</td>
<td>0-31</td>
</tr>
</tbody>
</table>
Common TCP Options

- Source Port
- Destination Port
- Sequence Number
- Acknowledgement Number
- Advertised Window
- Urgent Pointer
- Flags
- Checksum
- Options

Port Values:
- 0
- 16
- 31

Options Position:
0 to 15
Common TCP Options

- Window scaling
Common TCP Options

- Window scaling
- SACK: selective acknowledgement
Common TCP Options

- Window scaling
- SACK: selective acknowledgement
- Maximum segment size (MSS)
Common TCP Options

- Window scaling
- SACK: selective acknowledgement
- Maximum segment size (MSS)
- Timestamp
Window Scaling

- Problem: the advertised window is only 16-bits
  - Effectively caps the window at 65536B, 64KB
  - Example: 1.5Mbps link, 513ms RTT
Window Scaling

- Problem: the advertised window is only 16-bits
- Effectively caps the window at 65536B, 64KB
- Example: 1.5Mbps link, 513ms RTT

\[
(1.5\text{Mbps} \times 0.513\text{s}) = 94\text{KB}
\]

\[
64\text{KB} / 94\text{KB} = 68\% \text{ of maximum possible speed}
\]
Window Scaling

- Problem: the advertised window is only 16-bits
  - Effectively caps the window at 65536B, 64KB
  - Example: 1.5Mbps link, 513ms RTT
    \[(1.5\text{Mbps} \times 0.513\text{s}) = 94\text{KB}\]
    \[
    \frac{64\text{KB}}{94\text{KB}} = 68\% \text{ of maximum possible speed}
    \]
- Solution: introduce a window scaling value
  - \(\text{wnd} = \text{adv}_\text{wnd} \ll \text{wnd}_\text{scale};\)
  - Maximum shift is 14 bits, 1GB maximum window
SACK: Selective Acknowledgment
SACK: Selective Acknowledgment

- Problem: duplicate ACKs only tell us about 1 missing packet
- Multiple rounds of dup ACKs needed to fill all holes
SACK: Selective Acknowledgment

- **Problem:** duplicate ACKs only tell us about 1 missing packet
  - Multiple rounds of dup ACKs needed to fill all holes
- **Solution:** selective ACK
  - Include received, out-of-order sequence numbers in TCP header
  - Explicitly tells the sender about holes in the sequence
Other Common Options

- Maximum segment size (MSS)
  - Essentially, what is the hosts MTU
  - Saves on path discovery overhead
Other Common Options

- **Maximum segment size (MSS)**
  - Essentially, what is the hosts MTU
  - Saves on path discovery overhead

- **Timestamp**
  - When was the packet sent (approximately)?
  - Used to prevent sequence number wraparound
  - PAWS algorithm
Issues with TCP

- The vast majority of Internet traffic is TCP
- However, many issues with the protocol
  - Lack of fairness
  - Synchronization of flows
  - Poor performance with small flows
  - Really poor performance on wireless networks
  - Susceptibility to denial of service
Problem: TCP throughput depends on RTT
Problem: TCP throughput depends on RTT
Fairness

Problem: TCP throughput depends on RTT
Problem: TCP throughput depends on RTT
Problem: TCP throughput depends on RTT
Problem: TCP throughput depends on RTT

- 100 ms
- 1000 ms
Problem: TCP throughput depends on RTT

100 ms

1 Mbps

1000 ms

1 Mbps
Fairness

- Problem: TCP throughput depends on RTT
  - 100 ms
  - 1 Mbps
  - 1 Mbps
  - 1 Mbps
  - 1 Mbps
  - 1 Mbps

- ACK clocking makes TCP inherently unfair
- Possible solution: maintain a separate delay window
- Implemented by Microsoft’s Compound TCP
Synchronization of Flows

- Ideal bandwidth sharing

Diagram:

- cwnd

---
Synchronization of Flows

- Ideal bandwidth sharing
- Oscillating, but high overall utilization
Synchronization of Flows

- **Ideal bandwidth sharing**
  - Oscillating, but high overall utilization

- In reality, flows synchronize
Synchronization of Flows

- Ideal bandwidth sharing
- Oscillating, but high overall utilization
- In reality, flows synchronize

One flow causes all flows to drop packets
Synchronization of Flows

- Ideal bandwidth sharing
- Oscillating, but high overall utilization

In reality, flows synchronize

One flow causes all flows to drop packets

Periodic lulls of low utilization
Small Flows

- Problem: TCP is biased against short flows
  - 1 RTT wasted for connection setup (SYN, SYN/ACK)
  - $cwnd$ always starts at 1
Small Flows

- Problem: TCP is biased against short flows
  - 1 RTT wasted for connection setup (SYN, SYN/ACK)
  - cwnd always starts at 1
- Vast majority of Internet traffic is short flows
  - Mostly HTTP transfers, <100KB
  - Most TCP flows never leave slow start!
Small Flows

- **Problem**: TCP is biased against short flows
  - 1 RTT wasted for connection setup (SYN, SYN/ACK)
  - $cwnd$ always starts at 1

- Vast majority of Internet traffic is short flows
  - Mostly HTTP transfers, <100KB
  - Most TCP flows never leave slow start!

- **Proposed solutions** (driven by Google):
  - Increase initial $cwnd$ to 10
  - TCP Fast Open: use cryptographic hashes to identify receivers, eliminate the need for three-way handshake
Problem: Tahoe and Reno assume loss = congestion
- True on the WAN, bit errors are very rare
- False on wireless, interference is very common
Wireless Networks

- Problem: Tahoe and Reno assume loss = congestion
  - True on the WAN, bit errors are very rare
  - False on wireless, interference is very common

- TCP throughput $\sim 1/\sqrt{\text{drop rate}}$
  - Even a few interference drops can kill performance
Wireless Networks

- Problem: Tahoe and Reno assume loss = congestion
  - True on the WAN, bit errors are very rare
  - False on wireless, interference is very common

- TCP throughput $\sim 1/\sqrt{\text{drop rate}}$
  - Even a few interference drops can kill performance

- Possible solutions:
  - Break layering, push data link info up to TCP
  - Use delay-based congestion detection (TCP Vegas)
  - Explicit congestion notification (ECN)
Denial of Service

- Problem: TCP connections require state
  - Initial SYN allocates resources on the server
  - State must persist for several minutes (RTO)
Denial of Service

- Problem: TCP connections require state
  - Initial SYN allocates resources on the server
  - State must persist for several minutes (RTO)
- SYN flood: send enough SYN's to a server to allocate all memory/meltdown the kernel
Denial of Service

- **Problem:** TCP connections require state
  - Initial SYN allocates resources on the server
  - State must persist for several minutes (RTO)

- **SYN flood:** send enough SYNs to a server to allocate all memory/meltdown the kernel

- **Solution:** SYN cookies
  - Idea: don’t store initial state on the server
  - Securely insert state into the SYN/ACK packet
  - Client will reflect the state back to the server
SYN Cookies

Sequence Number: 0
SYN Cookies

- Timestamp
- MSS
- Crypto Hash of Client IP & Port
Did the client really send me a SYN recently?
- Timestamp: freshness check
- Cryptographic hash: prevents spoofed packets
SYN Cookies

- Did the client really send me a SYN recently?
  - Timestamp: freshness check
  - Cryptographic hash: prevents spoofed packets

- Maximum segment size (MSS)
  - Usually stated by the client during initial SYN
  - Server should store this value…
  - Reflect the clients value back through them
SYN Cookies in Practice

- Advantages
  - Effective at mitigating SYN floods
  - Compatible with all TCP versions
  - Only need to modify the server
  - No need for client support
SYN Cookies in Practice

- **Advantages**
  - Effective at mitigating SYN floods
  - Compatible with all TCP versions
  - Only need to modify the server
  - No need for client support

- **Disadvantages**
  - MSS limited to 3 bits, may be smaller than clients actual MSS
  - Server forgets all other TCP options included with the client’s SYN
    - SACK support, window scaling, etc.