- February 22nd, 2021
    - TCP: Congestion Control
        - TCP uses various techniques to achieve reliable, in-order delivery of messages to and from hosts/processes
            - Sequence number
                - This number is included in the TCP header of the segment
                - It indicates the byte offset in the byte stream the application sends
                    - The receiver side can tell if some messages have been lost, by looking at the gaps in the sequence numbers
            - Acknowledgement number
                - Used by the receiver to indicate which segments it has received in order up to a certain point
            - Cumulative acknowledgement
                - This is used to deal with the potential loss of acknowledgements
                - As long as subsequent segments are being received, cumulative acknowledgements can be utilized to indicate what has been received so far
                    - Loss of previous acknowledgements is tolerable
            - Retransmission via timeout
                - Upon transmission of a segment, the TCP sender will start a timer
                    - If no acknowledgement has been received within that period of time, TCP will retransmit the packet
                    - Timeout values are based on the measurements of instantaneous SampleRTT and estimation of the variance of RTT
                - Retransmission can be triggered by 3 duplicate ACKs
                    - This method is quicker than retransmission via timeout
        - Congestion control is a very important function of TCP
            - Network congestion can have devastating or damaging effects
                - In the worst case scenario, everyone is busy transmitting and retransmitting, and no useful data can ever get across the network
                - Thus, it is important for senders to regulate the amount of data that is injected into the network
                    - TCP does not explicitly say how much bandwidth is available
                        - Instead, TCP utilizes a window based congestion control mechanism
        - The Internet does not explicitly indicate congestion
            - Routers are not tasked with relaying this kind of

information
      - This is done to simplify their implementation
    - TCP senders need infer the occurrence of congestion
  - Congestion detection
    - TCP infers congestion through a loss event
      - This can take place via:
        - Timeout
          OR
        - 3 Duplicate ACKs
      - In a wired network, it is assumed that packet loss is the result of a congested network, rather than a poor link quality; which is the case for wireless
    - After a loss event, the TCP sender reduces data transfer rate (congestion window)
      - The TCP sender needs to appropriately respond to congestion
        - i.e. Not be too aggressive and overflow the network, and not be to pessimistic so that it doesn't utilize the available bandwidth
        - In some settings, you would want TCP to be able to ramp up its transmission rate as fast as possible
          - i.e. In data centers, where intra-rack communication is very frequent
  - Rate adjustment (probing)
    - Slow start
      - When the transmission begins, TCP has very little information about the congestion level
        - Thus, TCP tries to ramp up its transmission rate as quick as possible, until it experiences some losses. At this point, TCP will fine tune the transmission rate
        - The congestion level of the network is always changing due to new connections being formed and old connections being terminated
      - In the beginning, TCP aggressively probes the network, and then does fine tuning once it has a reasonable guess of the available bandwidth in the network
    - Additive increase and multiplicative decrease (AIMD)
    - Conservative after timeout events
- Congestion Window (CWND)
  - TCP does not utilize rate-based control
    - Instead, it utilizes window based mechanism which allows it to control its rate of injecting data in the network
  - The congestion window limits how much data can be in transit
    - It is implemented in number of bytes
  - The sender maintains a window
    - The leftmost side of the window corresponds to data that has already been sent by the sender, acknowledged by the

    receiver, and delivered to the destination application
        - This data can be removed from the sender's buffer, because this data has been received by the corresponding application
    - Immediately on the right side of the last byte ACK'd is data that has been sent by the sender, but not yet acknowledged by the receiver
    - The rightmost side corresponds to the effective window
        - This is data that the sender is allowed to send, but has not yet sent it
- The maximum window is determined by 2 values:
    - Receiver window (rwnd)
        - Included in the TCP header, from the receiver, for the purpose of flow control
        - The receiver tells the sender how much buffer space it has
            - i.e. How much data the receiver can take
            - The sender should not send more data than 'rwnd'
    - Congestion window (cwnd)
        - Reflects how much the network can take
- The maximum window is used by the TCP window to determine how much more data it can send without any acknowledgement
    - TCP takes the minimum of the receiver window and congestion window
        - Thus, it does not overwhelm the receiver window or congestion window
- The effective window corresponds to how much data TCP is allowed to transmit at a given moment
    - If the effective window is greater than 0, then TCP is allowed to send more data
    - If the effective window is 0, then TCP is NOT allowed to send data
        - When an acknowledgement arrives, the window shifts to the right by 1, which increases the size of the effective window by 1
            - This allows TCP to send 1 more data packet
- When the receiver acknowledges data, the windows are shifted to the right, allowing the sender to transmit more data
- i.e. Diagram of The Window

```
                                        |--- Effective
                                        |    Window
                                        |
                                |---------------|
                                |               |
                                V               V
|---------------|---------------|---------------|
|               |///////////////|               |
|---------------|---------------|---------------|
                |               |
                V               V
```

```
            Last Byte ACK'd   Last Byte Sent

        MaxWindow = min(cwnd, rwnd)
        EffectiveWindow = MaxWindow - (LastByteSent - LastByteACK)
      - Roughly, the rate at which TCP can send data is the maximum
        window divided by RTT:
        - i.e.
                     (min(rwnd, cwnd))
            rate ~ ----------------- (bytes/sec)
                            RTT
        - The top part of the equation is the maximum window
          - This corresponds to the amount of data we can inject
            in one batch into the network, before an
            acknowledgement is received
        - The maximum window is divided by 1 RTT, since it takes 1
          RTT for an acknowledgement to be received from the time
          the first segment is sent in the window
        - The size of the maximum window is in bytes
          - i.e. ((Number of segments) * (Segment length))
        - By controlling the size of the congestion window and
          receiver window, we can regulate the rate at which TCP
          injects data into the network
        - The equation tells TCP how much data to send, but not
          when to send it
      - If TCP detects packet loss, it will lower the rate at which
        it sends data into the network
    - Self Clocking
      - The window (discussed in the previous slide) is moved based
        on the reception of the acknowledgements
        - At the receiver side, the acknowledgements are generated
          in response to the in-order delivery of the packets
          - Assume each packet generates 1 acknowledgement, and
            assume that there is no packet loss
        - The spacing time between the acknowledgements roughly
          corresponds to the transmission time of the packets that
          were sent by the sender
          - This is a rough approximation because ACKs are
            easier to transmit than application data
            - i.e. It does not take time to put ACKs on the
                   'wire'
          - The spacing between acknowledgements is also
            referred to as inter-arrival time
          - The transmission time of the packet(s) sent by the
            sender is determined by:
            ((Length of the packet) / (Bottleneck bandwidth))
            - The bigger the bandwidth, the shorter the
              transmission time
        - The sender observes that:
          (Received ACK spacing) ~ (L / Bottleneck bandwidth)
          - L = Length of the packet
```

    - When the sender receives an acknowledgement, it will pump up the moving window by 1 segment
        - This is the self-clocking mechanism
            - The arrival of acknowledgements, and the spacing between them can regulate the timing between the transmission of the packets from the sender side
            - The reception of acknowledgements tells the sender how much data it can send in one batch, and how frequently packets can be sent
            - i.e. If we have a large window, ACKs "self-clock" the data to the rate of the bottleneck link
    - To summarize, the transmission time of the packets that are sent by the sender is roughly equal to the ACK spacing of the packets sent by the receiver
        - The space between the ACKs gives the sender a rough idea of how much congestion is in the network
        - Put simply, the sender sends packets to the receiver based on how many ACKs it gets from the receiver
            - More ACKs = More packets are sent
            - Less ACKs = Less packets are sent
                - There must be congestion in the network
                - TCP throttles packet transmission
- Phases of TCP Congestion Control
    - Initially, TCP does not know how much bandwidth is available in the network
        - In addition, the network's condition is constantly changing due to background traffic
            - TCP needs to be able to adapt to these changes
    - Congestion window (cwnd)
        - Every TCP sender starts off with a 'cwnd' size of 1 MSS
            - The unit of MSS is bytes
            - Note: MSS = Maximum Segment Size
    - Slow start threshold value (ss_thresh)
        - Initial value is the receiver's window size
            - It is possible for the initial 'ss_thresh' value to be to aggressive
        - Artifically sets a threshold to indicate when to go from slow start phase to congestion avoidance phase
        - 'ss_thresh' is sort of used as the upper bound
    - Slow start (getting to equilibrium)
        - During slow start, TCP will grow 'cwnd' exponentially, by doubling the transfer rate each time
            - 'cwnd' will either reach the 'ss_thresh' OR TCP experiences packet loss
                - It's possible that the initial 'ss_thresh' is too aggressive
                - If packet loss occurs then congestion is present in the network, and TCP needs to cut down its transmission rate
        - While 'cwnd' < 'ss_thresh', TCP is in the slow start

phase

    - We want to find the upper limit very-very fast, and not waste any time
  - This is (paradoxically) called slow start because the overall transmission rate of TCP is relatively slow
- Congestion avoidance
  - If 'cwnd' >= 'ss_thresh', then TCP enters the congestion avoidance phase
    - Now, TCP has a rough idea of the available bandwidth on the network
  - Once TCP enters the congestion avoidance phase, it uses AIMD to gently probe the network for available bandwidth
    - AIMD = Additive Increase Multiplicative Decrease
    - TCP responds to packet loss or ACKs in a gentle manner
  - TCP reacts to network conditions upon entering the congestion avoidance phase
    - It adjusts its transmission rate in a less aggressive manner

- TCP: Slow Start
  - The goal of the slow start phase is to quickly discover proper sending rate (roughly)
  - Whenever a new TCP connection is started, or whenever traffic is increased after congestion (timeout) is experienced:
    - The initial 'cwnd' = 1 MSS
    - Each time a segment is acknowledged, the 'cwnd' is increased by 1 MSS
      - This continues until:
        - 'ss_thresh' is reached
          OR
        - TCP experiences packet loss
- Slow Start Illustration
  - The congestion window size grows very rapidly
    - It doubles every RTT
  - TCP slows down the increase of 'cwnd' when: 'cwnd' >= 'ss_thresh'
  - Observe:
    - Each ACK generates two additional packets that are sent
    - Slow start increases rate exponentially fast
      - Double every RTT
  - i.e. Diagram of Slow Start

```
                  |                          |
                  |    Segment 1             |
    'cwnd' = 1 MSS | ========================> |---|
                  |                          |   |--- 1 RTT
                  |    ACK For Segment 1     |   |     (#1)
    'cwnd' = 2 MSS | <------------------------ |---|
                  |                          |
                  |    Segment 2             |
               | =========================> |---|
               |                           |   |
               |    Segment 3              |   |
               | =========================> |   |--- 1 RTT
               |                           |   |     (#2)
               |    ACK For Segment 2, 3   |   |
'cwnd' = 4 MSS | <------------------------- |---|
               |                           |
               |    Segment 4              |
               | =========================> |---|
               |                           |   |
               |    Segment 5              |   |
               | =========================> |   |
               |                           |   |
               |    Segment 6              |   |
               | =========================> |   |--- 1 RTT
               |                           |   |     (#3)
               |    Segment 7              |   |
               | =========================> |   |
               |                           |   |
               |    ACK For Segment 4,5,6,7 |   |
'cwnd' = 8 MSS | <------------------------- |---|
               |                           |
```

    - Assume 1 MSS = 1 Byte
    - During slow start, the congestion window doubles every RTT
        - In the 1st RTT, 1 segment is sent
            - 'cwnd' = 1 MSS, in 1st RTT
        - In the 2nd RTT, 2 segments are sent
            - 'cwnd' = 2 MSS, in 2nd RTT
        - In the 3rd RTT, 4 segments are sent
            - 'cwnd' = 4 MSS, in 3rd RTT
    - During slow start, the sender doubles the amount of information it transmits every RTT
- After the 3-way handshake, the initial congestion window size is 1 MSS, and 1 segment is sent from the sender to the receiver
- The rule of the slow start phase is: upon reception of every acknowledgement, the size of the congestion window increases by 1 MSS
    - Before the reception of the ACK for segment #1, the maximum window size is 1
        - The effective window is 0 because the sender already finished transmitting the one segment that it is allowed to send, and TCP cannot send any more data
    - The moment the ACK for segment #1 arrives, TCP will increase 'cwnd' by 1
        - Now, 'cwnd' = 2 MSS, and MaximumWindow = 2
            - The EffectiveWindow is also 2

          - As a result, TCP is able to send 2 segments, back-to-back, on the next transmission
        - Upon reception of the ACK(s) for segment #2 and segment #3, the 'cwnd' increases by 2
          - Now, 'cwnd' = 4 MSS
          - As a result, TCP can send 4 segments, back-to-back, on the next transmission
          - Note: The sender can send individual ACKs for each segment, or a single ACK for all segments
        - Once the TCP sender receives the ACKs for segment 4, 5, 6, 7, the 'cwnd' increases by 4
          - Now, 'cwnd' = 8 MSS
        - The size of the congestion window, 'cwnd', increases until the 'ss_thresh' is reached, or TCP experiences some type of packet loss
      - The growth of 'cwnd' is exponential, because it doubles after the last ACK, of a round, is received
      - RTT does not depend on the congestion window, it depends on the level of congestion in the network
  - Additional Information (AQA)
    - The default start behaviour for TCP is to execute slow start every single time
      - This is (partly) because TCP connections are not necessarily destined to the same host
        - There can be multiple TCP connections on the same host, but to different peers
          - Thus, the corresponding available bandwidth tends to change
    - Despite TCP's exponential growth, enterprise networks (i.e. Data centers) find TCP's slow start phase to be too slow
      - There are some algorithms that try to shorten the slow start phase, so traffic can ramp up quicker

- February 24th, 2021
  - Phases Of TCP Congestion Control
    - The purposes of TCP congestion control are:
      1. Probe the available bandwidth in the network, and try to ramp up the transmission rate of the sender as quickly as possible
         - This corresponds to the slow start phase
      2. Adapt to fluctuations in available bandwidth
         - This corresponds to the congestion avoidance phase
    - Congestion window ('cwnd')
      - Initial value is 1 MSS
        - Counted in bytes
        - MSS = Maximum Segment Size
    - Slow start threshold value ('ss_thresh')
      - Initial value is the receiver's window size
    - Slow start (getting to equilibrium)

- While ('cwnd' < 'ss_thresh'), TCP is in slow start phase
- Want to find the upper bound of transfer very-very fast, and not waste time calculating it
  - TCP doubles the 'cwnd' size every 'RTT', until:
    - 'ss_thresh' is reached
      OR
    - A packet loss event occurs
- The term 'slow start' is sort of misleading, because TCP finds the upper bound for transmitting data very quickly
- Slow start enables the TCP sender to be able to quickly ramp up its transfer rate, and reach equilibrium

- Congestion avoidance
  - Congestion avoidance phase starts when: 'cwnd' >= 'ss_thresh'
    - The congestion avoidance phase takes place after the slow start phase
  - The key variable is: 'cwnd' (congestion window)
    - It determines, on average, how much data the TCP sender can send to the network without any acknowledgement
    - The transfer rate of the sender is roughly equal to: ('cwnd' / RTT)
      - However, this assumes that TCP is not limited by the receiver's window size
  - During congestion avoidance, TCP probes the available bandwidth, in the network, more gently
    - TCP uses AIMD to increase/decrease transmission
      - AIMD = Additive Increase,
        Multiplicative Decrease
      - Transfer rate is increased linearly
  - The congestion avoidance phase reacts to network conditions

- TCP: Slow Start
  - The goal of TCP's slow start is to quickly discover the proper sending rate (roughly)
  - The slow start phase happens:
    - At the beginning of a new connection
      OR
    - If traffic is increased after a congestion (timeout)
  - At the beginning of the slow start phase, the initial 'cwnd' = 1 MSS
    - Each time a segment is acknowledged, increment 'cwnd' by 1 MSS
      - This continues until:
        - 'ss_thresh' is reached
          OR
        - Packet loss occurs
- Slow Start Illustration
  - The congestion window size grows very rapidly
    - Slow start is a phase that allows TCP to ramp up its

sending rate quickly, because for every 'RTT' the congestion window size is doubled (exponential growth)
            - Equivalently, the sending rate is doubled for every 'RTT' (exponential growth)
            - This assumes that the packets are successfully delivered
    - TCP slows down the increase of 'cwnd' when: 'cwnd >= 'ss_thresh'
    - Observe:
        - Each ACK generates 2 more packets
        - Slow start increaes rate exponentially fast
            - It is doubled every 'RTT'
    - The slow start phase is run for every new TCP connection
- Congestion Avoidance (After Slow Start)
    - Slow start figures out roughly the rate at which the network starts getting congested
        - Due to varying number of connections and background traffic TCP still has to make adjustments
    - During the congestion avoidance phase, TCP continues to react to network condition
        - TCP probes for more bandwidth, and increases 'cwnd' if more bandwidth is available, but in a less aggressive manner
        - If congestion is detected (i.e. Timeout occurs OR Duplicate ACKs detected), aggressively cut back 'cwnd'
        - Uses AIMD
- TCP: Additive Increase & Multiplicative Decrease (AIMD)
    - Additive increase
        - Increase 'cwnd' by 1 MSS every 'RTT' in the absence of congestion/loss events
            - This is a linear increase; it is a gentle probe
            - If bandwidth is available in the network, TCP will increase its transmission rate
        - TCP gently (linearly) increases its congestion window
            - Note: This is different from 'slow start', because 'slow start' doubles 'cwnd'
    - Multiplicative decrease
        - Cut 'cwnd' in half after loss/congestion event
        - The decrease is done aggressively
            - It is exponential reduction of 'cwnd'
    - AIMD gives TCP a 'sawtooth' behaviour when 'cwnd' with respect to time is plotted on a graph
    - AIMD is a congestion avoidance algorithm that takes place after slow start
        - As long as application data needs to be sent, AIMD is used to control the transmission rate
- Why AIMD?
    - There are many congestion avoidance algorithms that can be implemented in TCP
        - i.e.

        - AILD = Additive increase, Linear decrease
            - Linear decrease is also referred to as subtractive decrease
        - MIMD = Multiplicative increase, Multiplicative decrease
        - EIMD = Exponential increase, Multiplicative decrease
        - MILD = Multiplicative increase, Linear decrease
            - Linear decrease is also referred to as subtractive decrease
    - However, none of these congestion avoidance algorithms are ideal for implementing in the Internet
- Benefits of AIMD:
    - Makes sure that the network bandwidth is being efficiently utilized
        - On the flip side, if everyone sends data at a very slow rate, congestion can be avoided. However, this is bad because there is a lot of unutilized bandwidth, which leads to lots of inefficiencies
    - Maintains fairness
        - If two connections share the same bottleneck bandwidth, the ideal case is for them to share the remaining bandwidth in an equal manner
            - We do not want one connection to have a lot of bandwidth, and the other connection is penalized
- i.e. Graph of Two Connections Sharing Bandwidth

```
                Y
                |
                |
            R_2 |
                |\
                | \             /
                |  \           / <---- Equal bandwidth share
                |   \         /
    Connection  |    \       /
        2       |     \     /
    Throughput  |      \   /
                |       \ /
                |        \ <---------- R_F
                |       / \
                |      /   \
                |     /     \
                |    /       \
                |   /         \
                |  /           \
                | /             \
                |/_______________\_________________________
                   Connection     R_1                      X
                       1
                   Throughput
```

    - This is a simple situation where only 2 connections share a bottleneck bandwidth

- i.e. Two competing sessions

- Additive increase (AI) gives a slope of 1, as throughput increases
- Multiplicative decrease (MD) decreases throughput proportionally
- The 'bottleneck bandwidth' = R
- On the X-axis is the throughput achieved by connection 1
- On the y-axis is the throughput achieved by connection 2
- The throughput can be calculated as the: ((Congestion window size) / (Round trip time))
    - It is assumed that the 'RTT' is equal for both connections
- Point 'R_1' means that connection 1 has throughput, and connection 2 has no throughput
- Point 'R_2' means that connection 2 has throughput, and connection 1 has no throughput
- Any point on line 'R' means that the bandwidth is being fully utilized
    - However, not all points are fairly distributing bandwidth among the two connections
        - i.e. Point 'R_1' and 'R_2'
- The point at which the two lines intersect, 'R_F', means that the bandwidth is being equally shared by both connections
    - 'R_F' is the ideal case; it is the sweet spot
        - This is what we desire/want
    - However, the other points on the 'equal bandwidth share' line are not fully utilizing the available bandwidth on the network
- AIMD helps the connections reach the sweet spot, 'R_F'

- Regardless of where the slow start phase ends up, AIMD will reach the equilibrium point, where all connections equally share bandwidth
    - The equilibrium point allows applications to transmit data as fast as possible while being fair to others
- In short, AIMD solves issues such as:
    - The connections are underutilizing the network
        - i.e. The bandwidth is not fully utilized
    - One connection is transmitting data at a higher rate than the others
        - i.e. The bandwidth is not equally shared

- AIMD Example
    - i.e. Diagram of Senders & Receivers

```
/---\    x                                /---\
| A |---------->//=============\\---------->| B |
\---/           ||             ||          \---/
                ||             ||
/---\    y      ||             ||          /---\
| B |---------->\\=============//---------->| E |
    \---/                                       \---/
```
- The senders are: 'A' and 'D'
  - The sending rate (throughput) of 'A' is represented via 'x'
- The receivers are: 'B' and 'E'
  - The sending rate (throughput) of 'B' is represented via 'y'

- i.e. Graph of AIMD On Active Connections

```
    Y
    |
    |
    |\ C
    | \
    |  \
    |   \
    |    \    *
    |     \  (2)        /
    |      \           /
    |   *   \         /
    |  (1)   \   *   /
    |         \ (4) /
    |          \   /
    |     *     \ /
    |    (3)   * \
    |       (5)/ \
    |          /   \
    |         /     \
    |        /       \
    |       /         \
    |      /           \
    |     /             \
    |    /               \
    |   /                 \
    |  /                   \
    | /                     \
    |/_________________________\____________________________
                                                           X
```

- The throughput of the connections follows the path of the asterisk
  - i.e. (1) -> (2) -> (3) -> (4) -> (5)
- Points under the dotted line, 'C' are underutilizing the network's resources
- In AIMD, when the network is underutilized, the connections increase their congestion window by 1 'MSS' for every 'RTT'
- In AIMD, the congestion window is decreased in a multiplicative manner for all connections
- After a few rounds of AIMD, the connections will reach

        the equilibrium pointer where they equally share the
        network's bandwidth
          - i.e. The transfer rate of 'x' is equal to the
            transfer rate of 'y'
          - The connections approach the bottleneck bandwidth
      - The connections never stabilize at the sweet spot, they
        will continue to fluctuate around the fair sharing line
    - Despite fluctuations, the advantage of AIMD is that it can
      reach a state of equilibrium that allows fair sharing, and
      fully utilizes the network's resources
- AIMD Sharing Dynamics
    - Through the AIMD algorithm, two connections that start off
      at different throughputs will eventually fluctuate around
      the same average throughput
        - The equalization of transfer rates indicates fair share
    - In AIMD, when there is:
        - No congestion -> Transfer rate increases by one
          (packet/RTT) every RTT
        - Congestion -> Transfer rate decreases by a factor of 2
- Example of Slow Start + Congestion Avoidance
    - i.e. Diagram of TCP Connection Phases

```
                    A                                   B
                    |                                   |
                    |    Segment 1                      |
        'cwnd' = 1  | ================================> |
                    |                                   |
                    |    ACK For Segment  1             |
        'cwnd' = 2  | <-------------------------------- |
                    |                                   |
                    |    Segment 2, 3                   |
                    | ================================> |
                    |                                   |
                    |    ACKs For Segment 2, 3          |
        'cwnd' = 4  | <-------------------------------- |
                    |                                   |
                    |    Segment 4, 5, 6, 7             |
                    | ================================> |
                    |                                   |
                    |    ACKs For Segment 4, 5, 6, 7    |
        'cwnd' = 8  | <-------------------------------- |
                    |                                   |
                    |    Segment 8, 9, 10, 11, 12,      |
                    |            13, 14, 15             |
                    | ================================> |
                    |                                   |
                    |    ACKs For Segment 8, 9, 10, 11  |
                    |                   12, 13, 14, 15  |
        'cwnd' = 9  | <-------------------------------- |
                    |                                   |
                    |    Segment 8, 9, 10, 11, 12,      |
                    |           13, 14, 15, 16          |
                    | ================================> |
                    |                                   |
                    |    ACKs For Segment 8, 9, 10, 11  |
                    |               12, 13, 14, 15, 16  |
        'cwnd' = 10 | <-------------------------------- |
                    |                                   |
                    |                                   |
                    V                                   V
```

- Assume that 'ss_thresh' = 8 MSS
    - Initially, 'cwnd' = 1 MSS
- Host 'A' is sending packets/segments to Host 'B'
- The size of the congestion window ('cwnd') changes for every 'RTT'
- In the slow start phase, for every 'RTT' the size of the congestion window doubles
    - i.e. Table Summarizing Congestion Window With Respect To Round Trip Times

'RTTs'	'cwnd'	Phase
0	1	Slow Start
1	2	Slow Start
2	4	Slow Start
3	8	'ss_thresh' is reached TCP has entered congestion avoidance phase. AIMD takes over
4	9	Congestion Avoidance
5	10	Congestion Avoidance
6	11	Congestion Avoidance

- While 'cwnd < ss_thresh' is true, TCP is in the slow start phase
    - The 'cwnd' doubles at the end of each round trip
- Once 'cwnd' = 8, the condition 'cwnd >= ss_thresh' is satisfied, and TCP enters the congestion avoidance phase
    - AIMD takes over and the 'cwnd' increases by 1 MSS at the end of each round trip

- Response To Congestion (Loss)
    - There are many versions of TCP
        - Traditionally, TCP implementations are named after cities
    - There are algorithms developed for TCP to respond to

congestion
        - TCP Tahoe
            - Timeout only, start from slow start
            - TCP Tahoe only uses timeout to detect packet loss
                - Does not use triple-duplicate ACKs to detect loss events
                - When a timeout occurs it will retransmit the packet, and start from the slow start phase
                    - This happens every time there is a loss event
            - Very aggressive in cutting back transmission during a timeout
        - TCP Reno
            - Tahoe + fast retransmit & fast recovery
                - Fast retransmit utilizes 3-duplicate ACKs to detect packet loss
                - Fast recovery means that TCP Reno does not use slow start when it detects congestion
                    - Instead, it goes through the congestion avoidance phase via AIMD
            - Most end hosts today implement TCP Reno
                - Currently, Reno dominates the Internet
        - TCP Vegas
            - Mostly used in research
            - Uses timings of ACKs to avoid loss
        - TCP SACK
            - Planned for future deployment
            - Only sends selective ACKs
                - The receiver intelligently informs the transmitter what segments have been lost, instead of cumulative acknowledgement
- Sawtooth Behaviour: TCP Tahoe
    - TCP Tahoe switches between the slow start phase and congestion avoidance phase
        - Every time a packet is lost:
            - The 'ss_thresh' is set to half the current value of the congestion window ('cwnd')
            - The congestion window ('cwnd') is set to 1 MSS
                - The TCP connection goes through the slow start phase, again
        - Throughout the connection, 'cwnd' fluctuates between 1 and the initial 'ss_thresh' value
    - After the 3-way handshake, TCP Tahoe enters the slow start phase where the congestion window doubles every time
        - The transfer rate grows exponentially during slow start
    - If a packet is lost, TCP Tahoe waits for a timeout
        - i.e. If the timer finishes before an acknowledgement arrives, then the lost segment is retransmitted
    - TCP Tahoe treats timeouts as an aggressive version of congestion, the 'ss_thresh' is set to half of the current

value of (the) 'cwnd', and 'cwnd' is reset to 1 MSS
    - Setting 'ss_thresh' to half of the current value of 'cwnd' allows TCP Tahoe to better estimate what the congestion window should be
  - Since (the) 'cwnd' is now 1 MSS, the connection has to go through the slow start phase, once again
    - The congestion window ('cwnd') exponentially increases until it reaches the new (smaller) 'ss_thresh' value
      - Assuming that no timeout occurs, the connection enters the congestion avoidance phase and linearly increases 'cwnd'
        - The congestion window ('cwnd') is increased until a loss event occurs, and steps 2-4 are repeated
  - To summarize:
    - TCP Tahoe illustrates a sawtooth behaviour when plotted
    - Upon timeout, the 'ss_thresh'is cut by 1/2, and the 'cwnd' is set to 1 MSS
      - Then, go to slow start phase and increase transfer rate; this process is repeated until the connection is terminated
- TCP: Reno
  - The main difference between TCP Tahoe and TCP Reno is that Reno avoids timing out every single time there is a loss
    - Since timeout is based on various 'RTTs', the wait tends to be a large value
      - During timeout, there is no transmission of data
  - TCP Reno avoids timing out, every single time there is a loss event, by utilizing a fast retransmit mechanism
    - Fast retransmit is triggered by the reception of 3 duplicate acknowledgements
      - Instead of timing out, it will retransmit the last packet
      - This makes retransmission much quicker; hence the name "fast retransmit"
  - Note: It is possible for TCP Reno to experience a timeout
    - This is because sending 3 duplicate acknowledgements is not always feasible
      - i.e. There is not enough data that has been sent by the sender, because the congestion window is too small
        - If the congestion window is 2, it's not possible to receive 3 duplicate acknowledgements
      - i.e. If most of the acknowledgements get lost then the sender may not receive a duplicate ACK
  - TCP Reno interprets 3 duplicate ACKS as mild congestion
    - The fact that the sender is able to receive 3 duplicate ACKs implies that the network is not too congested
      - Therefore, TCP Reno does not enter slow start because setting 'cwnd' to 1 MSS is unnecessary and

very aggressive
            - Since the network is not too congested, it is better to enter the congestion avoidance phase, rather than slow start
                - Thus, TCP Reno does not have huge fluctuations as TCP Tahoe does
    - To summarize:
        - The problem with TCP Tahoe is: If a segment is lost, there is a long wait until timeout
        - TCP Reno adds a fast retransmit and fast recovery mechanism
            - Fast retransmit:
                - Upon receiving 3 duplicate acknowledgements, retransmit the (presumed) lost segment
            - Fast recovery:
                - Instead of entering the slow start phase, enter congestion avoidance instead, because the network is not too congested
- Fast Retransmit & Fast Recovery
    - After the 3-way handshake, TCP Reno enters the slow start phase, where it probes the network's resources
        - Initially, (the) 'cwnd' is set to 1 MSS, and it grows exponentially
    - If a packet gets lost, before reaching the 'ss_thresh', and if the sender receives 3 duplicate ACKs, then TCP Reno enters the congestion avoidance phase
        - Firstly, (the) 'cwnd' is cut in half
            - This is multiplicative decrease
        - Secondly, (the) 'cwnd' is linearly increased until congestion is encountered in the network
            - This is additive increase
    - TCP Reno exhibits a 'wave' behaviour if it does not experience a timeout
        - It will never go to the slow start phase if it only receives 3 duplicate ACKs
            - 'cwnd' is cut in half, but not dropped to 1 MSS
    - When 'cwnd' with respect to time is plotted, the area under the graph represents total data sent
        - Dividing this value by time yields speed/throughput
        - The larger the area under the curve, up to a fixed time, the higher the throughput (or speed)
        - Since TCP Reno does not drop to slow start, its area tends to be bigger than Tahoe's
            - Thus, TCP Reno is more efficient than TCP Tahoe, by utilizing fast retransmit and fast recovery
    - To summarize:
        - Slow start only once per session
            - Only if there are no timeouts
        - In a steady state, (the) 'cwnd' oscillates around the ideal window size

- Fast Transmit Algorithm
    - i.e. Pseudocode For Retransmit Algorithm

```
event: ACK received, with ACK field value of y
    if (y > LastByteACKed + 1) {
        LastByteACKed = y - 1
        if (there are currently not-yet ACK'd segments)
            start timer
    } else {
        ### Duplicate ACK for an already ACK'd segment ###
        increment count of dup ACKs received for y
        if (count of dup ACKs received for y = 3) {
            ### Fast Retransmit ###
            resend segment with sequence number y
        }
    }
```

        - When the sender receives an acknowledgement, the 'LastByteACKed' is increased by 1
        - If the sender receives a duplicate acknowledgement, then a counter for duplicate ACKs is incremented
            - When this duplicate ACK counter reaches 3, then retransmission is triggered
    - i.e. Diagram of Sender's Window

```
                                        MaxWindow
                                            |
                                            |
                                            |
                        |---------------------------------------|
                        |                                       |
                        |                                       |
                        V                                       V
        |---------------|-------------------|-------------------|
        |***************|###################|                   |
        |---------------|-------------------|-------------------|
                        |                  /|                   |
                        |                 / |                   |
                 LastByteACK'd           /  |                   |
                                        /   |                   |
                                       /    |-------------------|
                              LastByteSent            |
                                                      |
                                                      |
                                              EffectiveWindow
```

        - The '*' corresponds to all packets that have been ACK'd by the receiver and successfully sent to the sender
            - The rightmost packet/segment is the last byte that was successfully ACK'd by the receiver
        - The '#' corresponds to packets that have been sent to the receiver, but an ACK has not reached the sender
            - The leftmost packet/segment is the last byte that the sender transmitted to the receiver

- TCP Reno: Fast Recovery
    - After a fast retransmit:
        1. 'cwnd' = ('cwnd' / 2)
            - TCP Reno cuts the congestion window by half
                - VS. 1 in TCP Tahoe
        2. 'ss_thresh' = 'cwnd'
            - The 'ss_thresh' value is equal to the new value of the 'cwnd'
        3. Start congestion avoidance phase with new 'cwnd'
            - TCP Reno does not do slow start with 'cwnd' = 1 MSS
        - Note: A fast retransmit is triggered when the receiver sends 3 duplicate ACKs to the sender
            - The network is mildly congested
    - After a timeout:
        1. 'ss_thresh' = ('cwnd' / 2)
            - The 'ss_thresh' value is set to half of the current congestion window size
            - Similar to TCP Tahoe
        2. 'cwnd' = 1 MSS
            - Congestion window is set to 1 MSS
        3. Perform slow start
            - This is identical to TCP Tahoe
        - Note: If a timeout occurs in TCP Reno, then the network is (probably) severely congested; thus, the sender cannot receive 3 duplicate ACKs
            - The network is severely congested
    - The benefit of TCP Reno is to differentiate between the different levels of congestion in the network
        - i.e. Mild congestion VS. Severe congestion
        - i.e. Fast retransmit VS. Timeout
            - A fast retransmit indicates mild congestion in the network
            - A timeout indicates severe congestion in the network
- Additional Information (AQA)
    - TCP Vegas
        - Research for TCP Vegas was finished in the early 2000s
        - Adoption rate for TCP Vegas is unclear
            - This is because there is a fairness issue between TCP Vegas and TCP Reno
                - There may be a situation where a host that runs TCP Reno is penalized
    - An operating system's scheduler can prioritize some applications over others
        - i.e. Downloading a movie has higher priority/speed than uploading an email attachment
    - Inside the network, there are certain ways that priorities can be set for different flows, which can be treated differentially by the routers
        - This is not easy to do, because there needs to be a way to tell routers that certain packets has a higher

priority than others

- February 26th, 2021
    - Recap From Last Class
        - The two main versions of TCP are: Tahoe and Reno
        - The biggest difference between Tahoe and Reno is how they behave in response to packet losses in the network
            - TCP Tahoe detects packet losses via timeout
                - Upon timeout, it will reduce its congestion window to 1 MSS, and enter the slow start phase
                - During the slow start phase, the congestion window size is doubled every round trip time
                - Once the condition 'cwnd' >= 'ss_thresh' is satisfied, TCP Tahoe will enter the congestion avoidance phase
            - TCP Reno, which is the most dominant version of TCP adopted today, utilizes two mechanisms called 'fast retransmit' and 'fast recovery'
                - Fast retransmit is triggered by the reception of 3 duplicate acknowledgements
                    - TCP Reno will retransmit the last packet
                    - Instead of timing out and entering the slow start phase, TCP Reno will enter the congestion avoidance phase
                        - The congestion window is cut in half
                            - This is multiplicative decrease
                    - When TCP Reno enters the congestion avoidance phase, it will increase the congestion window linearly, as long as there is no packet loss
        - TCP Reno's response to a loss event is not as dramatic as TCP Tahoe's
        - TCP Reno and Tahoe can commnunicate with each other
            - Operating systems can implement their own version of TCP
                - Thus, different TCP versions can talk to each other because the TCP packet format/options are the same
            - Note: If two different versions of TCP are co-existing, then it is possible that they won't fairly share the available bandwidth
    - Tahoe VS. Reno Example
        - Assume that:
            - Initial value for 'ss_thresh' is 8 MSS
            - There is no packet loss in the slow start phase
            - The first two packets transmitted in round 8 are lost, and there are no subsequent losses
        - What is the 'ss_thresh' and 'cwnd' in round 9?
            - In round 8, (the) 'cwnd' is 12 MSS
                - Thus, the TCP sender can send a total of 12 segments
            - The losses in round 8 will generate multiple duplicate acknowledgements, because subsequent segments are still correctly received

- TCP Reno will half the congestion window from 12 to 6, and then set 'ss_thresh' to the new 'cwnd' value, 6
  - Both 'cwnd' and 'ss_thresh' are 6
  - Next, TCP Reno will enter the congestion avoidance phase, where it will gently increase (the) 'cwnd' via AIMD
- TCP Tahoe will set the 'ss_thresh' to half the current 'cwnd', and then (the) 'cwnd' is set to 1 MSS
  - 'ss_thresh' is 6, and 'cwnd' is 1
  - Then, TCP Tahoe enters the slow start phase, where it doubles 'cwnd' every round, until the condition ('cwnd' >= 'ss_thresh') is satisfied
- i.e. Table of Congestion Window With Respect To Round, For TCP Tahoe & TCP Reno

Beginning Of Round Number	Value of 'ss_thresh'	Congestion Window For TCP Tahoe	Congestion Window For TCP Reno
1	8	1	1
2	8	2	2
3	8	4	4
4	8	8	8
5	8	9	9
6	8	10	10
7	8	11	11
8	8	12	12
9	6	1	6
10	6	2	7
11	6	4	8
12	6	6	9
13	6	7	10
14	6	8	11
15	6	9	12

  - Throughout the connection, the 'ss_thresh' is the

      same for TCP Tahoe and Reno
        - When a loss event occurs, 'ss_thresh' is set to half of 'cwnd'
        - 'ss_thresh' dictates when TCP leaves the slow start phase and enters congestion avoidance
      - Upon acknowledgement of every segment that has been sent during the slow start phase, (the) 'cwnd' will increase by 1
        - The congestion window ('cwnd') does not wait until the end of the round to increase/double
          - Rather, the reception of an ACK immediately increments (the) 'cwnd'
      - When 'cwnd' >= 'ss_thresh', TCP enters congestion avoidance phase
        - TCP does not wait until the congestion window size is doubled before it enters the congestion avoidance phase
          - Instead, the congestion avoidance phase is immediately entered when the above condition becomes true, rather than the end of a round
      - The area underneath the curve of a plotted TCP graph corresponds to the total number of bytes that have been transmitted up to a certain time
        - Dividing this number by time yields the average throughput/speed of the connection
          - Typically, TCP Tahoe is not as efficient as TCP Reno
          - Under the same network conditions, TCP Reno's throughput/speed tends to be greater than TCP Tahoe's throughput
      - When TCP Tahoe encounters congestion, it reduces the congestion window ('cwnd') down to 1 MSS
      - When TCP Reno encounters congestion, it cuts 'cwnd' by half
        - This is multiplicative decrease
      - If the network is heavily congested, timeouts can still occur in TCP Reno
        - In this case, TCP Reno behaves the same way as TCP Tahoe
        - When a time occurs, TCP sets:
          - 'ss_thresh' = 'cwnd' / 2
          - 'cwnd' = 1 MSS

- Comparison
  - i.e. Table Summarizing TCP Tahoe

State	Event	Sender Action	Comment
SS	ACK receipt for previously unacked data	cwnd = cwnd + MSS if (cwnd > ss_thresh) {	Resulting in a doubling

		set state to CA }	of 'cwnd' every RTT
CA	ACK receipt for previously unacked data	cwnd = cwnd + MSS * (MSS/cwnd)	Additive increase, resulting in in-crease of cwnd by 1 MSS every RTT
SS or CA	Loss event detected by triple duplicate ACK	N/A; TCP Tahoe does not have a mechanism for fast retransmit.	
SS or CA	Timeout	ss_thresh = (cwnd / 2) cwnd = 1 MSS Set state to SS	Enter SS
SS or CA	Duplicate ACK	N/A; TCP Tahoe does not keep track of duplicate ACKs	

- 'SS' = Slow start phase
- 'CA' = Congestion avoidance phase
- 'cwnd' = Congestion window
- 'ss_thresh' = Slow start threshold

- i.e. Table Summarizing TCP Reno

State	Event	Sender Action	Comment
SS	ACK receipt for previously unacked data	cwnd = cwnd + MSS if (cwnd > ss_thresh) { set state to CA }	Resulting in a doubling of 'cwnd' every RTT
CA	ACK receipt for previously unacked data	cwnd = cwnd + MSS * (MSS/cwnd)	Additive increase, resulting in in-crease of cwnd by 1 MSS every RTT
SS or CA	Loss event detected by	ss_thresh = (cwnd / 2)	Fast recovery,

	triple duplicate ACK	cwnd = ss_thresh Set state to CA	implementing multiplicative decrease. cwnd will not drop below 1 MSS
SS or CA	Timeout	ss_thresh = (cwnd / 2) cwnd = 1 MSS Set state to SS	Enter SS
SS or CA	Duplicate ACK	Increase duplicate ACK count for segment being ACKed	cwnd and ss_thresh are not effected

    - 'SS' = Slow start phase
    - 'CA' = Congestion avoidance phase
    - 'cwnd' = Congestion window
    - 'ss_thresh' = Slow start threshold

- Summary
  - Multiplexing and demultiplexing
    - Port number
  - Error detection/recovery
    - Internet checksum
      - Inclusion of pseudo header
    - Stop-n-Wait, Go-back-N, Selective repeat
  - Connection establishment/termination
  - Flow control
  - Congestion control
    - Sliding window
    - AIMD
    - Slow start
    - Fast retransmit & recovery
  - TIP: Be able to distinguish algorithms and protocols
    - Basic principles and mechanisms to achieve reliable data transfer
    - Impact of congestion (control) in the network
    - Between TCP & UDP, the common services provided are multiplexing and demultiplexing
      - TCP performs error detection, and error recovery through retransmission of the packet
      - UDP detects packet error, but does not recover it
    - TCP maintains connections, because it provides a bi-directional byte stream between the end-processes

      - TCP performs flow control to avoid overwhelming the sender
      - TCP performs congestion control to avoid overwhelming the network
- IP Layer
  - This is the next chapter
  - IP = Internet Protocol
  - The IP layer is, arguably, one of the most important layer in the TCP/IP protocol suite
    - Answers the question: "How does a packet figure out the end-to-end path it needs to follow from source to destination host?"
      - Rough answer: "There are a collection of protocols in the network layer that facilitate end-to-end routing of packets in the network"
- Outline
  - Introduction
  - What's inside a router
  - IP: Internet Protocol
    - Datagram format
    - IPv4 addressing
    - IPv6
    - ICMP
  - Routing algorithms
- Network Layer
  - The purpose of the network layer is to transport packets/ segments from sending to receiving hosts
    - In contrast, the transport layer provides an abstraction between processes on end-hosts
  - On the sending side the network layer will take the segment, coming from the transport layer, and encapsulate the segments into datagrams
    - In addition to the payload, that essentially is the transport layer segment, the datagram includes header information that is used for the purpose of addressing, controlling the behaviour of packet delivery, and packet forwarding in the network
  - On receiving side, the network layer delivers segments to the transport layer
  - Network layer protocols are found in every (end) host, router, etc.
    - The network layer is also found in routers inside the network
      - In contrast, the transport layer only resides on end-systems/hosts
  - Network routers examine/utilize information in the header fields of IP datagrams, and use it to make decisions as to which outgoing interface a particular packet needs to be forwarded to
- Internet Structure: Network of Networks

- The network is organized in a hierarchical manner
    - There are different tiers of ISPs
        - ISPs within the same tier may have a peering relationship, (i.e. customer-provider), between:
            - Customer and service provider
            - Different tiers of ISPs
- A message passes through many networks
    - In order to determine the end-to-end pass, or route, a packet needs to consider the following:
        - Which post is connected to which local ISP
        - How are ISPs of the same tier, or different tiers, connected to each other?
        - Policy issues across different tiers and ISPs
            - Individual ISPs may have a complete picture of their own network structure/topology, but such information is often considered a trade secret and is not available to other ISPs
                - Despite the lack of transparency, the internet still works
            - There needs to be a way for different ISPs to exchange a limited amount of information, but still be able to figure out the end-to-end pass between hosts

- IPv4 Address
    - IP addresses are the mechanism utilized in the network layer
        - Currently, the dominant address format in use is IPv4
    - IPv4 is a unique 32-bit number associated with a host, and a router interface
        - If a router has multiple network interfaces, then each interface should be associated with a different IP address
    - IPv4 addresses are represented in a dotted-quad notation
        - The 32-bit address is broken down into 4 parts
            - 1 part is 1 byte
            - Decimal representation is used to represent each part, and dots are used to separate the parts
                - This makes the address human-readable and memorable
        - i.e. '200.23.16.5'
            - i.e. Diagram of IPv4 Address In Binary

```
|----------|----------|----------|----------|
| 11001000 | 00010111 | 00010000 | 00000101 |
|----------|----------|----------|----------|
    200        23         16          5
```

            - This is the conventional way to represent IPv4 addresses
    - Interface: Connection between host/router and physical link
        - Router's typically have multiple interfaces
        - Host typically has one interface
- Subnets

```
- Relevant for local area networks, as well as address lookups
  in the routers
- 32 bits in the number are partitioned into 2 parts:
    - Prefix part
        - Also called the network address
        - Corresponds to:
            - The network that an IP address belongs to
               OR
            - A block of IP addresses who may have shared the
              same destination port inside the router
                - i.e. Collection of prefix of the top most
                       significant bits in the IP address
    - Host part
        - Also called host addresses
        - Unique to different network interfaces within the
          same network
    - There is no strict definition of how many bits should be
      in the prefix part or host
        - For different local area networks, you can configure
          the network mask
            - The network mask tells you how many bits in the
              prefix correspond to the network address portion
              of IP address, and then the remaining bits
              correpsond to the host address
        - The size of the prefix depends on the routers,
          because it's utilized to represent a block of
          addresses
            - The prefix field can be anywhere from 1 to 31
    - i.e. Diagram of Partitioned IPv4 Address
             200        23         16          5
        |----------|----------|----------|----------|
        | 11001000 | 00010111 | 00010000 | 00000101 |
        |----------|----------|----------|----------|
        |<-------------------------------->|<-------->|
                          |                    |
                          |                    |
                          V                    V
                 Prefix (23 bits)        Host (9 bits)
- Interdomain routing operates on the prefix
    - This is very important to know
- Notation: 200.23.16.5/23 represents a subnet with a 23 bit
            prefix and 2^9 host addresses
    - The number following the slash indicates the number of
      bits in the prefix field
        - i.e. In a 32-bit address, if 23 bits are allocated
               for the prefix field, then 9 bits are allocated
               for the host field because 32 - 23 = 9 bits
    - The local area network that corresponds to the IPv4
      address 200.23.16.5/23, will have a (total) maximum of
      2^9, or 512, host addresses
```

            - The hosts share the common prefix, but the last 9 bits are different
        - In a local area network when you send data to all hosts in the same network (i.e. Broadcast), essentially, you send data to all hosts that share the same prefix
    - The Internet Network Layer
        - The ultimate job of a router is to forward packets along the network
            - For incoming packets, a router needs to lookup its destination IP address, and decide which outgoing interface the particular packet needs to be forwarded to
            - This is done via information retrieved from a forwarding table
                - Forwarding tables store information about blocks of IP addresses, and their corresponding outgoing interfaces
        - Protocols help routers populate forwarding tables, and implement router functions
            - The IP protocol specifies:
                - Addressing convention
                    - i.e. IPv4
                - Datagram format in the IP header
                - Some packet handling conventions
            - Routing protocols
                - The purpose of routing protocols is to populate the forwarding tables
                - Routers need to exchange information to be able to compute the end-to-end path
                    - Plus, routers are responsible for handling issues where ISPs only exchange a limited amount of information
                - There are different protocols to handle routing within the same ISPs, and routing across multiple ISPs
                    - As a result, multiple routing protocols are implemented in the internet
                - Ultimately, routing protocols exchange information to be able to populate forwarding tables
                    - Example of information exchanged:
                        - Topology information
                        - Connectivity information
                        - Available bandwidth
                        - Latency across routers
            - ICMP protocol
                - The main purpose of ICMP is primarily for error reporting, and in some cases for signaling
                    - i.e. `traceroute` utilizes the ICMP protocol in the network layer to determine whether a packet has reached a certain number of hops, and whether it has reached the

destination

- i.e. Diagram of Host & Router Network Layers

```
|---------------------------------|
| Transport Layer                 |
|   - i.e. TCP, UDP, Etc.         |
|---------------------------------|-----|
| Routing protocols               |     |
|   - Path selection              |     |
|   - RIP, OSPF, BGP              |     |
| IP Protocol                     |     |
|   - Addressing conventions      |     |--- Network Layer
|   - Datagram format             |     |
|   - Packet handling conventions |     |
| ICMP Protocol                   |     |
|   - Error reporting             |     |
|   - Router "signaling"          |     |
|---------------------------------|-----|
| Link Layer                      |
|---------------------------------|
| Physical Layer                  |
|---------------------------------|
```

- Router Architecture Overview
  - Routers have 3 main functions:
    - At the top, there is a routing processor
      - Essentially, the routing processor is a piece of software, and in some cases it can be a general purpose processor to run software/algorithms
      - The job of a routing processor is to:
        - Determine routes
        - Populate routing table
        - Manage the router
    - Below the network processor, there are multiple network interfaces
      - Each interface is associated with a different port
      - From a packet's point of view, it goes into an input port, and will be delivered to an output port
        - In reality, all input/output ports are network interfaces
        - The ports are connected through some kind of switching fabric, that allows internal routing of packets from an input port to an output port
  - In recent years, the notion of a software defined network (SDN) has become very popular
    - SDNs breakdown router functionalities into:
      - Control plane operations
        - Decides what to do about the packet
          - i.e. Compute the path
          - i.e. Should a particular packet receive preferential treatment
      - Data plane operations

        - Does the actual work of forwarding the packet from input port to output port
    - You can think of the control plane like the manager, and the data plane is the employee that does the hard work of relaying the packet from input to output at a very high speed
    - Innovations in SDNs are primarily done in the control plane
        - i.e. How to implement different functions and support different applications
- i.e. Diagram of Router Architecture

```
                    |-------------------------|
                    |                         |
                    |         routing         |
                    |        processor        |
                    |            |            |
                    |------------|------------|
                                 |
                                 |
    -x-x-x-x-x-x-x-x-x-x-x-x-x-x | x-x-x-x-x-x-x-x-x-x-x-x-x-
                                 |
                                 V
    |---------------|      |----------|      |---------------|
    |  |-|  |----|  |      |          |      |  |----|  |-|  |
    |  | |  |    |  |      |          |      |  |    |  | |  |
    ---------------------> |          | --------------------->
    |  | |  |    |  |      |          |      |  |    |  | |  |
    |  |-|  |----|  |      |          |      |  |----|  |-|  |
    |---------------|      |          |      |---------------|
                           | high-    |
            *              | speed    |             *
                           |          |
            *              | switch-  |             *
                           | ing      |
            *              | fabric   |             *
                           |          |
    |---------------|      |          |      |---------------|
    |  |-|  |----|  |      |          |      |  |----|  |-|  |
    |  | |  |    |  |      |          |      |  |    |  | |  |
    ---------------------> |          | --------------------->
    |  | |  |    |  |      |          |      |  |    |  | |  |
    |  |-|  |----|  |      |          |      |  |----|  |-|  |
    |---------------|      |----------|      |---------------|
```

    - The routing processor is contains the routing management software
        - It is the control plane; software
    - The high-speed switching fabric is responsible for forwarding packets
        - It is the data plane; hardware
- To summarize:

    - Two key router functions:
      - Run routing algorithms/protocol
        - i.e. RIP, OSPF, BGP, Etc.
      - Forwarding datagrams from incoming to outgoing link
- Input Port Functions
  - i.e. Diagram of Switching Fabric From Input Port Side

```
|---------------------------------------------|  //
|                               |----------|  |  \\
|                 |--------|    | lookup,  |  |  //
|  |---------|    | link   |    | forward- |  |  \\
|  | line    |    | layer  |    |  ing,    |  |  // switch
-->| termin- |--->| proto- |--->| [] [] [] |---> \\ fabric
|  | ation   |    |  col   |    | [] [] [] |  |  //
|  |---------|    | (rece) |    | queue-   |  |  \\
|                 |--------|    |  ing,    |  |  //
|                               |----------|  |  \\
|---------------------------------------------|  //
```

      - Note: 'rece' = receive
    - Physical layer
      - Bit level reception
    - Data link layer
      - i.e. Ethernet
        - Refer to chapter 5
    - Decentralized switching
      - Given datagram destination, lookup output port using forwarding table in input port memory
        - "Match plus action"
      - The goal is to complete input processing at "line speed"
      - Queueing occurs if datagrams arrive faster than forwarding rate into switch fabric
  - Both input and output ports are essential, and necessary, network interfaces
  - The physical layer and (data) link layer receive packets
  - The input port has a forwarding table
    - It is used to quickly lookup which output port a packet needs to be forwarded to
  - The input port has a queue that temporarily stores incoming packets during lookup operations
- Forward Table
  - Given a destination IP address, the forwarding table needs to determine what outgoing interface the packet needs to be sent to
  - It is not feasible for a forwarding table to store information for all 2^32 (~4 billion) internet hosts
    - Storing this information will take up a lot of space
    - Processing this lookup is computationally expensive
  - Forwarding tables are organized in blocks of IP addresses
    - Similar to how mail distribution centers operate on physical mail

        - i.e. Canada Post
        - The mail center groups letters by destination, and then groups them together by region
            - i.e. All mail going to Ontario in bin #420
- Forwarding tables do not provide a lookup for every single IP address
    - Instead, it will try as much as it can to aggregate a range of addresses together to map them to a particular outgoing interface
        - This reduces computation time and storage complexity
- i.e. Diagram of Network Interfaces

Destination Address Range	Link Interface
11001000 00010111 00010000 00000000 through 11001000 00010111 00010111 11111111	0
11001000 00010111 00011000 00000000 through 11001000 00010111 00011000 11111111	1
11001000 00010111 00011001 00000000 through 11001000 00010111 00011111 11111111	2
otherwise	3

    - In this example, there are 4 network interfaces, each interface corresponds to a block of IP addresses
        - i.e. The 1st block accounts for IP addresses within the range:

```
          11001000 00010111 00010000 00000000
                                ---
          11001000 00010111 00010111 11111111
```

            - If an IP address falls within the range of the first block, then it needs to be delivered to the '0' link interface
    - For the purposes of IP forwarding we only care about destination IP address, not source address
        - It does not matter where the packet originated from, the forwarding decision only depends on where the packet is headed to
    - If an IP address does not fall into 1 of the 3 blocks, then it is forwarded to link interface '3'
    - A 32-bit address yields more than 4 billion possible entries
- i.e. Diagram of Router With Different Interfaces

```
    |-----------------------|
    |                       |
|   ------\   /------   |
|         \ /           |
|          \            |
|         / \           |
|   ------/   \------   |
|                       |
|-----------------------|
     /   /   \   \
    /   /     \   \
   /   /       \   \
  /   /         \   \
 /   /           \   \
/   /             \   \
1   2               3   4
```

- Forward Table: Longest Prefix Matching (1)
    - The destination address ranges are stored in the forwarding table, and are utilized during packet forwarding
        - Incoming IP packets are compared against the destination address ranges, to determine which block of IP address the corresponding destination IP address falls into
            - The mechanism used to do this is called the longest prefix match
    - i.e. Diagram of Forward Table With Longest Prefix Matching

Destination Address Range	Link Interface
11001000 00010111 00010000 00000000	
^^^^^^^^ ^^^^^^^^ ^^^^^	
through	0
11001000 00010111 00010111 11111111	
^^^^^^^^ ^^^^^^^^ ^^^^^	
11001000 00010111 00011000 00000000	
^^^^^^^^ ^^^^^^^^ ^^^^^^^^	
through	1
11001000 00010111 00011000 11111111	
^^^^^^^^ ^^^^^^^^ ^^^^^^^^	
11001000 00010111 00011001 00000000	
^^^^^^^^ ^^^^^^^^ ^^^^^	
through	2
11001000 00010111 00011111 11111111	
^^^^^^^^ ^^^^^^^^ ^^^^^	
otherwise	3

        - For an incoming packet, the destination IP address in the packet is compared against the 3 blocks
            - If there is no match, then the packet is forwarded to link interface #3

    - You examine each block, and determine what is the longest prefix in common among addresses in the block
      - For a given packet, you want to find which block has the longest prefix match, and set its corresponding link interface as the outgoing interface
    - In the example above, the 1st block's address range has (the first) 21 bits in common
      - For the 2nd block, the first 24 bits are common in the address range
        - For the 3rd block, the first 21 bits are common
      - These are the longest prefix that correspond to all the addresses with each range
  - Listing out all possible IP addresses within a range is very ineffective
    - Instead, it is better to keep track of the smallest IP address and biggest IP address within the range, and note down the longest prefix that corresponds to each of the blocks
      - This gives a more compact representation, and enables the data plane to perform an IP address lookup more effectively
  - Longest prefix matching: When looking for forwarding table entry for given destination, use longest address prefix that matches destination address
- Forward Table: Longest Prefix Matching (2)
  - i.e. Longest Prefix Match Example

Prefix Match	Link Interface
11001000 00010111 00010	0
11001000 00010111 00011000	1
11001000 00010111 00011	2
otherwise	3

    - When performing a lookup, you need to find which IP address in the range has the most number of consecutive bits in common, starting from the most significant bit
      - Sometimes there may be more than 1 match; you need to filter through and find the longest prefix match
    - Which interface are the following destination addresses (DA) forwarded to?
      - DA: 11001000 00010111 00010110 10100101

        ```
        DA: 11001000 00010111 00010110 10100101
            ^^^^^^^^ ^^^^^^^^ ^^^^^
        ```
        - This packet is forwarded to: link interface #0
          - This is because the longest prefix match is with the first block, for the first 21 bits
      - DA: 11001000 00010111 00011000 11100000

```
^^^^^^^^ ^^^^^^^^ ^^^^^^^^
```

        - This packet is forwarded to: link interface #1
            - This is because the longest prefix match is with the second block, for the first 24 bits
        - Even though the third block is a match, we are looking for the longest prefix match
            - Thus, the second block is the answer
    - To summarize:
        - The longest prefix for each address block and its corresponding interface is stored in the forwarding table
        - During the lookup, we try to find the longest prefix match with the destination IP address with the prefixes in the lookup table
            - The corresponding interface is where the packet is forwarded to
- Switching Fabrics
    - Once the lookup is completed, and a link interface is decided, it is the job of the switching fabric, to transfer the packet from the input port to the output port
        - Note: The switching fabric connects the input port, and the output port
        - It is imperative that the switching fabric/mechanism that performs this is very fast and effective
    - Ideally, you would want to be able to send packets in parallel to different, or the same, outgoing interfaces that the packets need to be sent to
        - Speed and parallelization are important factors in the design of the switching fabric
    - There are 3 different types of switching fabrics:
        1. Memory
            - After finishing the lookup, the packets that need to be delivered are stored in memory
                - Then, when the outgoing interface becomes available, the packet is retrieved from the corresponding memory location
            - This is not a good solution, because first you need to write to memory, and then read from memory
                - This is a two step process
        2. Bus
            - The interfaces are connected through a bus
                - When you need to deliver a packet from an input port to an output port, you put the packet on the bus which will see the interface address that the packet needs to be delivered to
            - The problem with buses is that only 1 packet can be delivered at a time, from 1 interface to another
                - Buses don't support simultaneous packet delivery
                    - Thus buses are not an efficient mechanism

        3. Crossbar
            - Currently, the internet/routers implement crossbar
                - This is the predominant implementation
            - Provide a grid of interconnectivities between the input ports and output ports
            - In the best case scenario, you can have a throughput of `N` packets
                - You can concurrently deliver `N` packets from the incoming interface to the outgoing interface
                    - This accelerates the forwarding plane operation
    - To summarize:
        - The purpose of a switching fabric is to transfer packet from input buffer to appropriate output buffer
        - Switching rate: Rate at which packets can be transferred from inputs to outputs
            - Often measured as multiple of input/output line rate
            - `N` inputs: Switching rate `N` times line rate desirable
        - Three types of switching fabrics
            - Memory
            - Bus
            - Crossbar
- Output Ports
    - When the data is sent through the switching fabric to the output interface/port, it may temporarily buffer the packet
        - This occurs if the arrival rate of the packets is faster than the transmission rate of the outgoing link
        - Typically, packet losses occur during buffering
            - If a packet arrives at a full buffer, it is dropped
    - In the output port, at the queue, a priority scheduling mechanism can be implemented
        - This allows prioritization of certain packets
        - Different scheduling policies can be implemented
            - i.e. First in first out (FIFO), Round Robin (RR), priority scheduling based on certain bits that are marked in the IP packet, etc.
        - ISPs can implement different policies for forwarding decisions
            - i.e. For different flows from different customers
    - i.e. Diagram of Switching Fabric From Output Port Side

```
       //  |---------------------------------------------|
       \\  |                                             |
       //  |   |----------|    |---------|               |
       \\  |   | datagram |    | link    |    |---------|  |
switch //  |   | buffer   |    | layer   |    | line    |  |
fabric \\ ---->| [] [] [] |---> proto-   |---> termin- |-->
       //  |   | [] [] [] |    |  col    |    | ation   |  |
       \\  |   | queueing |    | (send)  |    |---------|  |
       //  |   |----------|    |---------|               |
               \\  |                                             |
               //  |---------------------------------------------|
```

- To summarize:
    - Buffering is required when datagrams arrive from fabric faster than the transmission rate
        - Datagram/packets can be lost due to congestion, lack of buffers, etc.
    - Scheduling discipline chooses among queued datagrams for transmission
        - Priority scheduling: Who gets best performance, network neutrality, etc.

- Outline
    - See previous slide for outline
    - The most important thing in today's lecture is "Longest prefix match"
        - Relates to how packet lookup works, and the notion of a subnet