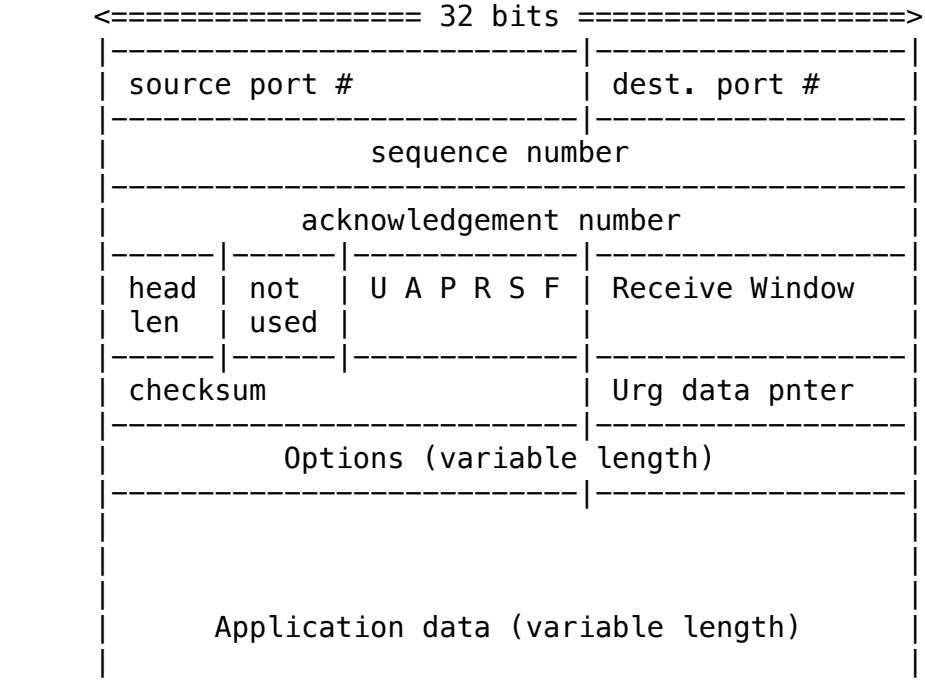
Week.5.txt

- February 8th, 2021
    - TCP Outline
        - Overview of TCP
        - TCP header format
        - TCP connection establishment & tear down
            - What are the error scenarios?
        - Reliable data transfer
        - Congestion control
    - TCP: Overview
        - RFCs: 793, 1122, 1323, 2018, 2581
        - Main services that TCP provides are:
            - Point-to-point logical communication between sender and receiver application processes
                - One sender, one receiver
                    - You cannot use one TCP connection to send to multiple application processes, or have multiple processes send to the same receiver
                    - There is a one-to-one mapping between sender and receiver
            - Reliable, in-order byte stream
                - No "message boundaries"
                - TCP may breakdown application layer data into segments, and send them in a TCP packet
                    - Regardless of how the segmentation is done, the application data is viewed as a byte stream
                - It is very important for TCP to maintain the order
                    - From an application point of view, the data that has been sent first, arrives first at the application process on the other side
            - Pipelined
                - Multiple segments can be sent before an acknowledgement for the previous segment is back
                    - In contrast, a naive way of reliable data transfer is to send a packet, and wait for an acknowledgement before sending the next packet
                - Allows end-points to effectively utilize the network's resources
                - TCP congestion and flow control set window size
            - Send & receive buffers
                - The buffer on the sender side is utilized to buffer packets that have been sent, but not yet acknowledged
                - On the receiver side, the buffer is used to handle packets that may arrive out-of-order
            - Full duplex data
                - Bi-directional data flow in same connection
                    - Both sides can send and receive data from/to each other

        - i.e. HTTP connection between server and client
    - MSS: Maximum segment size
        - Large application data is split into segments, by TCP, before it is sent
        - The size is determined by the TCP peers
            - Factors include:
                - Maximum transfer unit that can be supported along the links on the end-to-end path
                    - Some paths do not support large segment sizes
                - IP header overhead
                - Transport header overhead
- Connection-oriented
    - Handshaking initiates sender-receiver state before data exchange
        - Handshaking is the exchange of control messages
        - A 3-way handshake is used establish a connection
            - The connection is requested by the client to send their application data
        - Connection tear down closes the socket between sender and receiver
- Flow controlled
    - Sender will not overwhelm receiver
    - TCP performs congestion control to prevent the sender from congesting the network
        - A congested network prevents end-hosts from receiving data from clients
- TCP Segment Structure
    - i.e. Diagram of TCP Packet



```
    |                                                 |
    |                                                 |
    |-------------------------------------------------|
```

  - 'head len' = 'header length'
  - 'R' = RST
  - 'S' = SYN
  - 'F' = FIN
  - 'Urg data pnter' = 'Urgent data pointer'
- The diagram above shows TCP header information and payload
  - The TCP header starts at 'source port #' and ends just before the start of 'Application data'
    - The minimum number of rows in a TCP header is 5
      - Each row corresponds to 4 bytes
      - The minimum length of a TCP header is 20 bytes
        - The actual length depends on whether or not there are any options
  - The payload is the 'Application data'
    - It can vary in length
- Compared to UDP, TCP has a lot more information included in its header
  - Thus, TCP has more overhead than UDP
- The 'source port #' and 'dest. port #' are utilized by the transport layer for the purpose of multiplexing and de-multiplexing
  - These fields are also found in UDP packets
  - The port numbers are connected to something well known
    - i.e. HTTP is on port 80
  - The port numbers can be filled in by the operating system
  - Each port number is 16-bits
- The 'sequence number' counts the number of bytes that corresponds to the position of the payload in the application string
  - Indicates the application data that is sent to the sender
  - The unit of the 'sequence number' is bytes, and NOT the number/count of packets/segments
- The 'acknowledgement number' is only valid when the TCP segment is also used as an acknowledgement
  - Since TCP connections are bi-directional, the end-hosts can send acknowledgements to each other in addition to application data
  - Indicates what is the next byte the receiver is expecting
    - The unit is bytes; similar to 'sequence number'
- The 'header length' field counts the total header length in units of 32-bit words
  - Indicates if there is additional information, in the form of optional fields, beyond the mandatory 20-bytes
- 'UAPRSF' are single-bit fields and correspond to flags

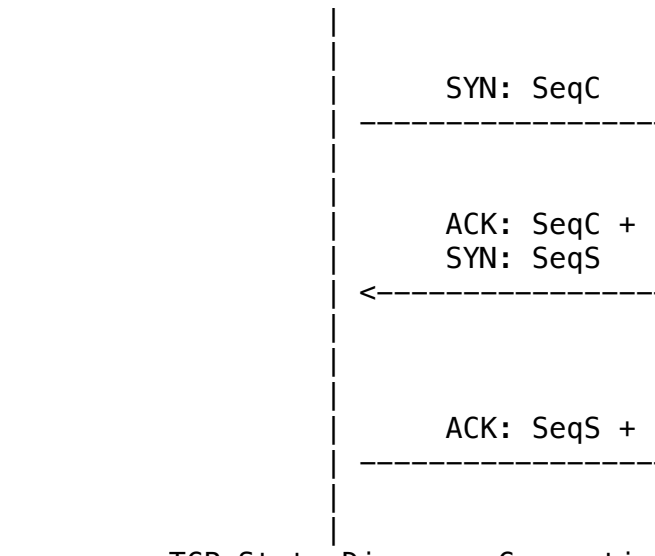
        - 'U' = Urgent data
            - Generally not used
        - 'A' = Acknowledgement
            - Indicates whether the TCP segment includes the acknowledgement number or not
        - 'P' = Push data now
            - Immediately sends the data, instead of waiting for the TCP segment to fill up
                - Useful when sending commands to a remote computer over the network
            - Typically, TCP waits until there is a sufficient amount of data to put in from the application side before putting it into a segment. TCP tries to fill-up the segment before sending it off
                - The 'P' flag vetos this
        - 'R', 'S', 'F'
            - Related to connection setup and tear down
    - The 'receiver window' is utilized for TCP flow control
        - TCP flow control is necessary to prevent the sender from overwhelming the receiver
        - The receiver maintains a buffer, and fills this field to indicate to the TCP sender how much buffer space it has left
            - If the 'receiver window' is set to 0, the TCP sender will wait for the buffer to free up
    - The 'checksum' field is used by TCP to validate if there is any bit error in the received segments
        - Checksum is computed via the TCP header, TCP payload, and pseudo-header that contains the IP header
            - Similar to UDP checksum
    - The 'urgent data pointer' is used in combination with the 'U' flag
        - It points to the payload, indicating that there might be urgent data that needs to be processesed immediately
        - Note: This is not commonly used
- TCP: Segments
    - TCP "Stream of Bytes" Service
        - TCP provides an end-to-end bitstream between application processes
        - This is why the sequence numbers are counted in units of bytes
    - TCP segment
        - No more than Maximum Segment Size (MSS) bytes
        - Segment sent when segment full (MSS) or "Pushed" by application
            - TCP sends data when the size of the segment reaches MSS, OR if the application indicates that the data needs to be sent immediately regardless of the segment size
            - MSS mostly refers to the payload; it does not

account for the TCP header
    - Example:
        - Assume that the MSS is 100 bytes, and the application needs to send 1000 bytes. What is the maximum size of each TCP segment, and how many segments are sent?
            - The maximum size of each TCP segment is 100 bytes
            - In total, 11 segments are sent
                - i.e.
                Segment 01 = Byte 0000 -- Byte 099
                Segment 02 = Byte 0100 -- Byte 199
                Segment 03 = Byte 0200 -- Byte 299
                Segment 04 = Byte 0300 -- Byte 399
                Segment 05 = Byte 0400 -- Byte 499
                Segment 06 = Byte 0500 -- Byte 599
                Segment 07 = Byte 0600 -- Byte 699
                Segment 08 = Byte 0700 -- Byte 799
                Segment 09 = Byte 0800 -- Byte 899
                Segment 10 = Byte 0900 -- Byte 999
                Segment 11 = Byte 1000
- TCP: Sequence Numbers, ACKs
    - TCP "Stream of Bytes" Service
    - Sequence numbers
        - Indicate starting byte offset of data carried in this segment
            - The offset is calculated with respect to the application layer data
        - Sequence numbers increase as more segments are sent
        - The numbers correspond to the position in the byte stream

    - ACKs
        - Gives sequence number just beyond highest sequence number received in order
        - "What byte is next"
            - Tells the sender what segments have been sent up to the acknowledgement number
        - Used by the receiver to indicate whether a certain amount of bytes have been received
        - TCP sends cumulative acknowledgements
- TCP: Sequence Numbers, ACKs Example
    - Assume that the sequence number is 1001. The sender sends 500 bytes. What is the ACK number that the receiver acknowledges with? (Assume that there is no packet loss).
        - Options:
            A) 501
            B) 1002
            C) 1500
            D) 1501
            E) 1502
        - The correct answer is 1501, because 1001 + 500 = 1501

    - The next sequence number that is sent by the sender is:
        - Options:
            A) 1500
            B) 1501
            C) 1502
        - The correct answer is 1501, because the next sequence number sent by the sender is the same as the previous acknowledgement number sent by the receiver
- Establishing Connection
    - Connection setup and tear down are very important steps in TCP
        - Before TCP sends data, it needs to establish a connection
        - Once a connection is setup, all data flows through the point-to-point pipe
            - At this point there is no need to include IP addresses in the TCP header, because the IP layer already has them
    - Three way handshake
        - It is called 3-way handshake, because there are in total 3 messages involved
            - The first message is sent by the client to the server
                - In this message, the 'SYN' flag is set to 1
                - The client picks a random initial sequence number
                    - Randomizing the initial sequence number helps thwarting replay attacks
                    - All subsequent sequence numbers are shifted by this value
            - The second message is sent by the server to the client, upon receiving the first message
                - The server acknowledges the initial 'SYN' sent by the client
                - The server sends its own 'SYN' number, which indicates that it is also willing to initiate a connection
                    - This number is randomly generated
                - Once this message is received, the client can start sending application data to the server
            - The third message is sent by the client to the server
                - The client acknowledges the message sent by the server, indicating that the client has received the message and the sequence number
                - Sometimes this message may contain application level data
        - To summarize:
            - Each side notifies other of starting sequence number it will use for sending

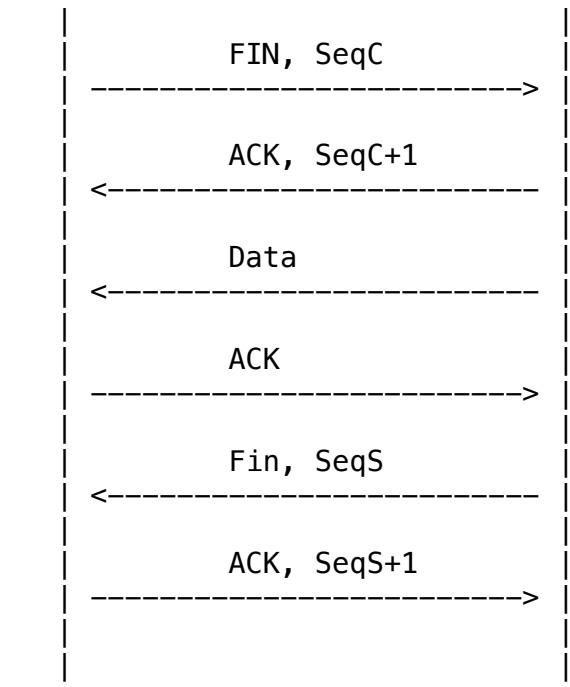
        - Each side acknowledges other's sequence number
            - SYN-ACK: Acknowledge sequence number + 1
                - In this segment, both 'SYN' and 'ACK' flags are set to 1
        - The third segment may piggyback some data
    - Why 3-way handshake?
        - The 3rd message is required because:
            - The TCP client needs to confirm the initial sequence number used by the TCP server
            - It is needed to acknowledge the passive open of the connection by the server
    - i.e. Diagram of 3-Way Handshake



- TCP State Diagram: Connection Setup
    - The action that TCP takes depends on what state it is in
        - i.e. Upon receiving the same message it may take different actions, because TCP is in different states
    - There are a total of 5 states in TCP:
        - CLOSED
        - LISTEN
        - SYN RECEIVED
        - SYN SENT
        - ESTABLISH
    - From the client's perspective:
        - After the socket is created, the client must connect to the server; this is the first step
            - The client provides the information of the IP address, and the port number of the server you want to create an active open connection with
        - Going from a CLOSED state to a 'SYN' SENT state is triggered by the client actively opening its connection
            - Under the hood, TCP sends the first message in the 3-way handshake

            - It sends a 'SYN', and its initial sequence number
        - Once the server receives the 'SYN' message from the client, it sends an acknowledgement and its own 'SYN' message to the client
        - The client sends its third message, an acknowledgement, to the server
            - Now, the client is in the 'ESTABLISH' state
                - At this point, the client can send application layer data to the server
    - From the server's perspective:
        - Clients are the entities that initiate connections, and servers are passively waiting for incoming connection requests
            - On the server side this is called 'passive open'
        - Once the server is binded to a port, it moves from a 'CLOSED' state to a 'LISTEN' state
        - When the server gets a 'SYN' message from a client, it sends an acknowledgement to the 'SYN', and its own 'SYN' message, which contains the server's initial sequence number, to the client
            - The (TCP) server is now in a 'SYN RECEIVED' state
        - Once the server gets an acknowledgement to its own 'SYN' message, from the client, it enters the 'ESTABLISH' state
            - Now, all subsequent data will correspond to the application layer
    - Messages sent over the internet can get lost
        - i.e. The initial 'SYN' message, the subsequent acknowledgement, etc. can get lost
        - TCP will retransmit lost messages
- Tearing Down Connection
    - Closing sockets after using them is a good practice, because it releases resources
    - Since TCP connections are bi-directional, you need to close both pipes to fully terminate the connection
        - i.e.
            - The client-to-server pipe needs to be terminated
            - The server-to-client pipe needs to be terminated
        - It is possible for the other side to continue sending data, as long as its pipe is still open/active
        - The connection is only completely disconnected after both sides tear down their pipes
    - Either side can initiate tear down
        - Tear down is initiated by sending a 'FIN' signal to the other side
            - This is a field in the TCP header; it is set to 1
            - The 'FIN' signal must be acknowledged by the other side
            - Tells the other side that "I'm not going to send any

more data"
- Other side can continue sending data
  - This is a half open connection
  - Must continue to acknowledge
- Acknowledging 'FIN'
  - Acknowledge (last sequence number + 1)
- i.e. Diagram of Tearing Down Connection

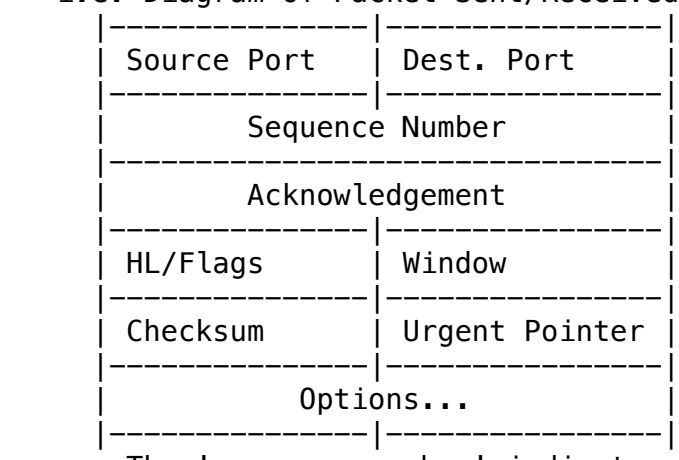


- February 10th, 2021
  - Tearing Down Connection
    - TCP is a connection oriented transport layer protocol
      - It needs to maintain a connection between the two communicating application processes
        - When one side has finished sending data, it needs to tear down the connection
    - The side that initiates tear down sends a 'FIN' segment, and a sequence number to the other side
      - The 'FIN' flag is sent to 1
    - The other side sends an acknowledgement to the 'FIN' signal
      - If the other side has more data to send, it will keep doing so
        - The side receiving data is responsible for replying with ACKs
    - A TCP connection is fully torn down when both sides have closed their pipes; typically done via 'FIN' signals
  - State Diagram: Connection Tear Down
    - The states are separated based on which side initiates the tear down first
    - In connection setup, there is a clear distinction between

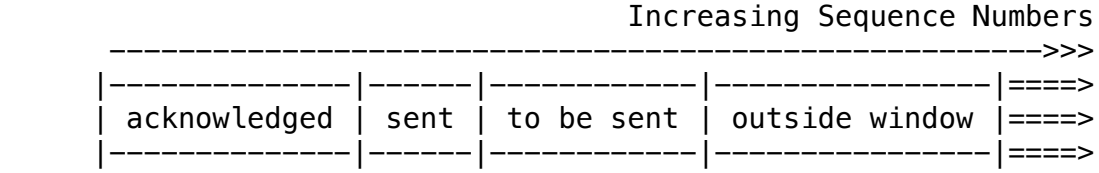
client and server
  - Typically, the client initiates the connection while the server passively waits for incoming connections
- Any side can initiate connection tear down
  - Depends on which side has no more data to send
- The side that initiates tear down is known as 'active close'
  - This can be either a server or a client
    - A 'FIN' segment is sent to the other side
  - To close a connection in socket programming, simply call the 'close()' method
  - The state of the side that initiates connection tear down moves from 'ESTABLISH' to 'FIN WAIT-1'
    - The 'FIN WAIT-1' state means that a 'FIN' segment has been sent, but it is currently waiting for an acknowledgement from the other side
- Once the other side acknowledges the 'FIN', the side that initiated the connection moves from 'FIN WAIT-1' to 'FIN WAIT-2'
  - At this point, one side of the connection has been terminated
    - However, it still needs to acknowledge data sent from the other side
  - It stays in 'FIN WAIT-2' until the other side sends a 'FIN'
- If a 'FIN' is sent from the other side, the side that initiated the connection enters a 'TIME WAIT' state
  - The TCP connection has to wait for a certain amount of time, typically 30 seconds, before the connection is completely closed
  - The 'TIME WAIT' state ensures that each side gets the messages
    - If messages are lost in the network, they are retransmitted
    - The timeout period allows the other side to request retransmission
- The side that does not initiate connection tear down is called passive close, and it can continue transmitting its data
  - This side moves from an 'ESTABLISH' state to a 'CLOSE WAIT' state
  - If this side wants to close its connection, it sends a 'FIN' to the other side and enters the 'LAST ACK' state
    - Once an acknowledgement for the 'FIN' is received, the TCP connection has been completely terminated
      - The state is 'CLOSED'
- A half closed TCP connection/socket does not release resources
  - The resource is released only after the connection has been completely closed
- The side that initiates tear down first is called 'active

close'
  - The side that initiates tear down later is called 'passive close'
- If both sides simultaneously close their connections, then the sides follow the states: 'ESTABLISH' --> 'FIN WAIT-1' --> 'CLOSING' --> 'TIME WAIT' --> 'CLOSED'

- TCP Window Control
  - i.e. Diagram of Packet Sent/Received

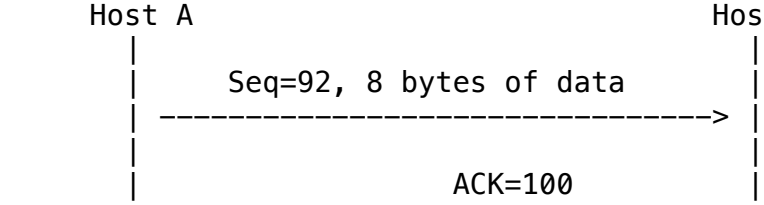


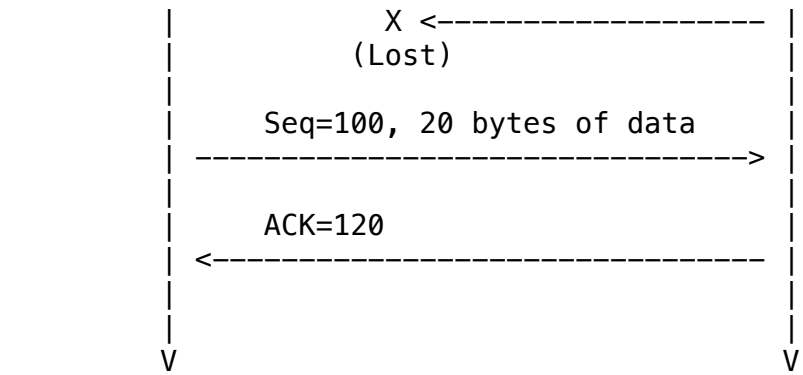
    - The 'sequence number' indicates the offset, in bytes, of the particular segment in the byte stream of application layer data
    - The 'acknowledgement number' corresponds to what packet has been received in terms of the bytes in the byte stream of application data
      - Used by the receiver to indicate that packets have been received
    - The 'Window' is used by the receiver to indicate how much buffer space is available
      - This tells the sender how much more data can be sent
  - i.e. Diagram of Byte Stream



    - Everything up to the 'acknowledgement number' has been sent and acknowledged by the receiver
    - The bytes within the 'sent' interval have been sent by the TCP sender, but the TCP sender has yet to receive an acknowledgement from the receiver side
      - This data has to be buffered at the TCP sender, because there is a chance that parts of it may get lost; retransmission may be required
    - The TCP sender is only allowed to send bytes contained in 'to be sent'
      - As long as 'to be sent' is not 0, the TCP sender

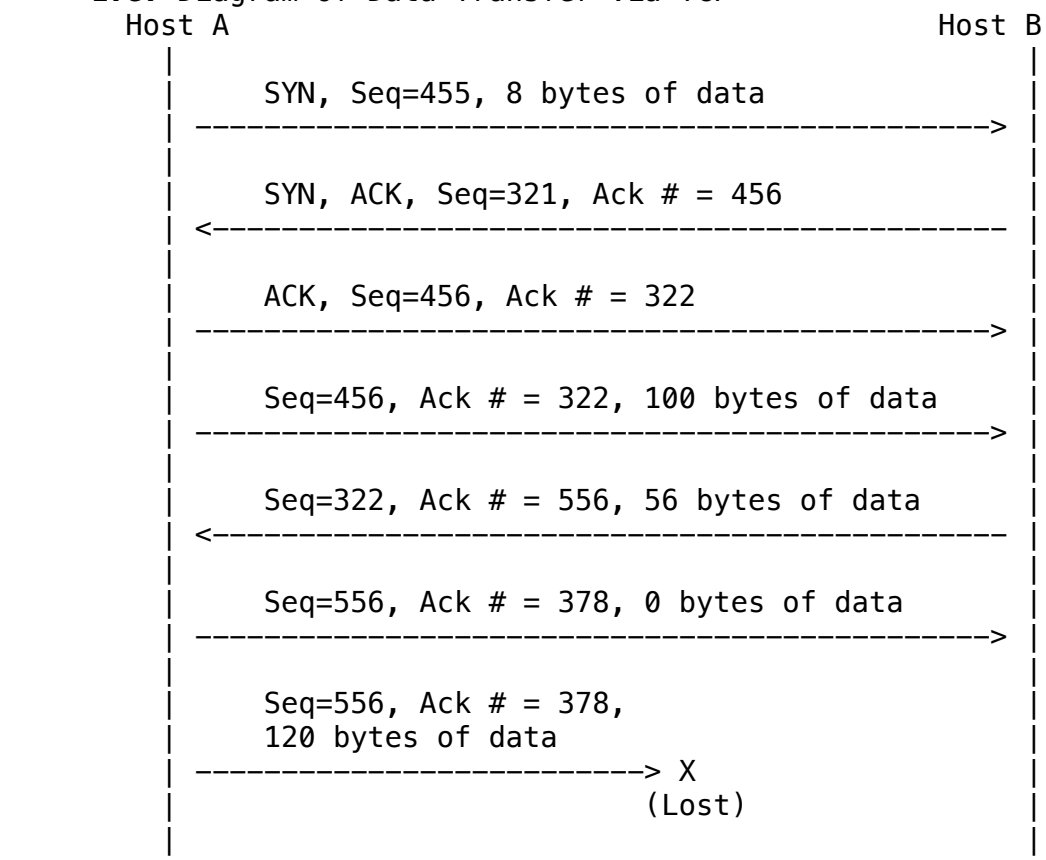
        will be able to send additional segments
    - The receiver may buffer data that has been sent, but not yet acknowledged
    - The sender's buffer is ('sent' + 'to be sent' + 'outside window')
    - The application is not allowed to write outside the 'outside window'
    - The window moves when the leftmost 'sent' data is acknowledged by the receiver
      - As the leftmost 'sent' data is acknowledged, the 'acknowledged' window increases in size
      - As a result, more data can be sent and written to the receiver's buffer
- TCP Reliable Data Transfer (1)
  - In addition to reliable data transfer, TCP allows efficiency by implementing a pipeline
    - TCP does not send one segment at a time, and waits for an acknowledgement
    - TCP sends a buffer of data, multiple segments, to the receiver
      - The receiver sends multiple acknowledgements back to the sender
  - TCP creates reliable data transport service on top of IP's unreliable service
    - Pipelined segments
  - The time it takes to send a segment, and receive an acknowledgement for the segment is called round trip time, RTT
  - In one round trip time (RTT), a maximum of `BW * RTT` bits can be sent to fill up the pipe
    - 'BW' is bottleneck bandwidth
      - Due to the dynamic nature of the network, we don't know the 'BW'
- TCP Reliable Data Transfer (2)
  - TCP utilizes cumulative acknowledgements
    - The point of cumulative acknowledgement is to acknowledge the current segment and all previous segments
    - Allows for redundancy
      - If some acknowledgements get lost, the subsequent acknowledgements will make up for them
    - Fault tolerant
      - Can tolerate losses in acknowledgements
  - i.e. Diagram of Data Transfer Via TCP

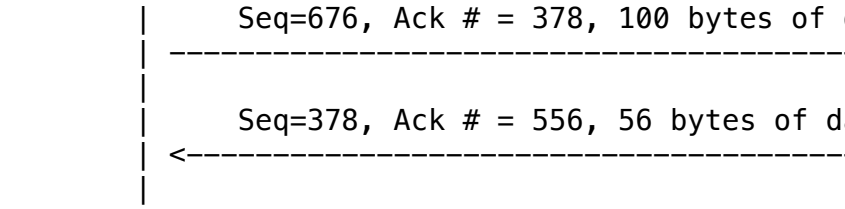




- Host A is sending data to Host B
- The first segment Host A sends to Host B has a sequence number of 92, and contains 8 bytes of data
  - The acknowledgement from Host B is lost
- The second segment Host A sends to Host B has a sequence number of 100, and contains 100 bytes of data
  - This acknowledgement is not lost, and tells Host A that Host B has received the latest segment and all previous segments

- TCP Reliable Data Transfer (3)
  - Retransmission can be triggered by timeout
    - Example is below
  - i.e. Diagram of Data Transfer Via TCP

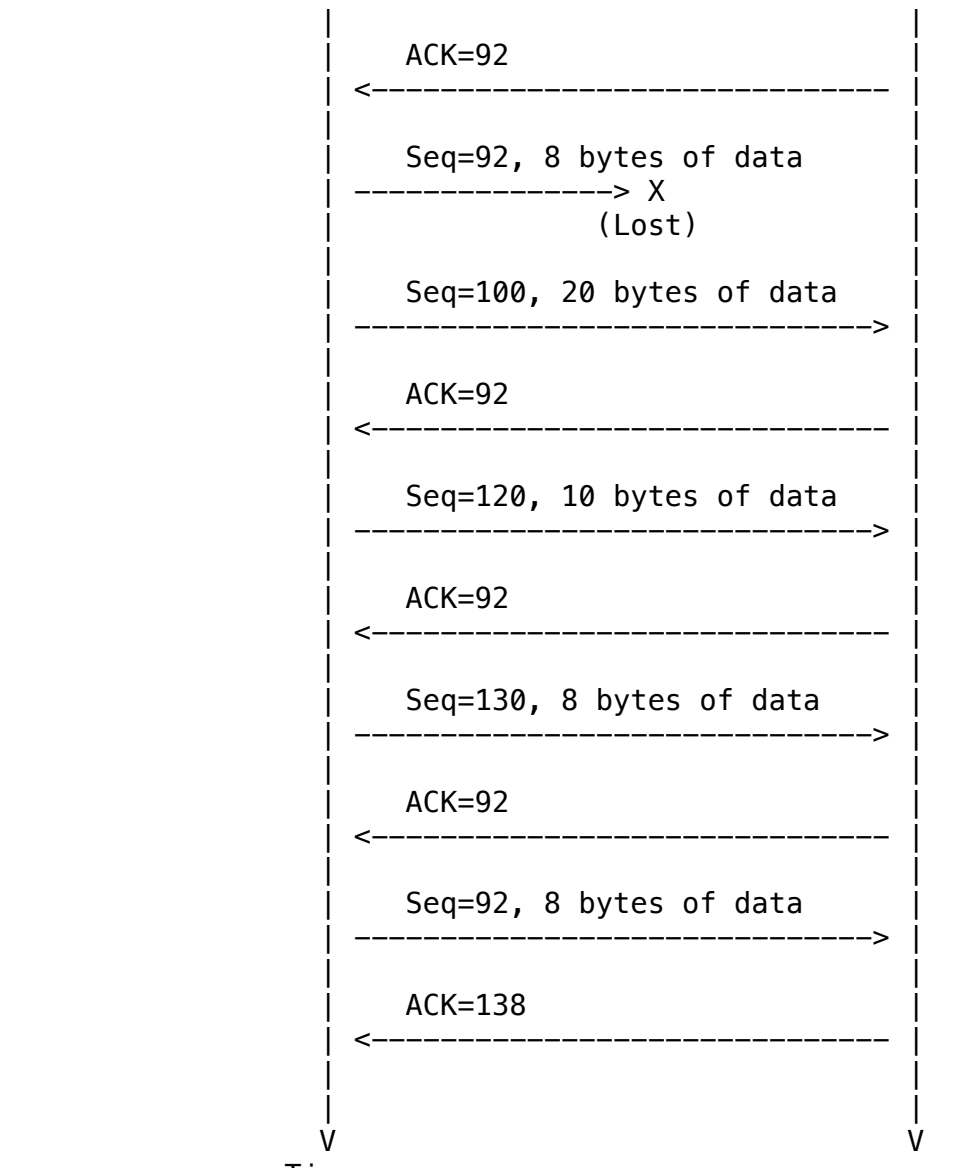




- The first 3 packets correspond to the 3-way handshake
- When segments are lost, the receiver sends an acknowledgement number that is equal to the sequence number of the lost segment
  - The sender starts a timer and waits for the receiver to transmit further acknowledgements
    - The sender is waiting for the receiver to send an acknowledgement for the lost segment
    - If the timer expires, the sender will retransmit the lost segment
      - During the timeout period, the sender does not transmit any new data
    - This is called 'retransmission triggered by timeout'
  - When segments are not acknowledged in time, the sender retransmits the lost packets
- Since TCP is bi-directional, the client can send acknowledgements to the server, and the server can send acknowledgements to the client
  - When one side sends 0 bytes of data to the other, it is sending an acknowledgement

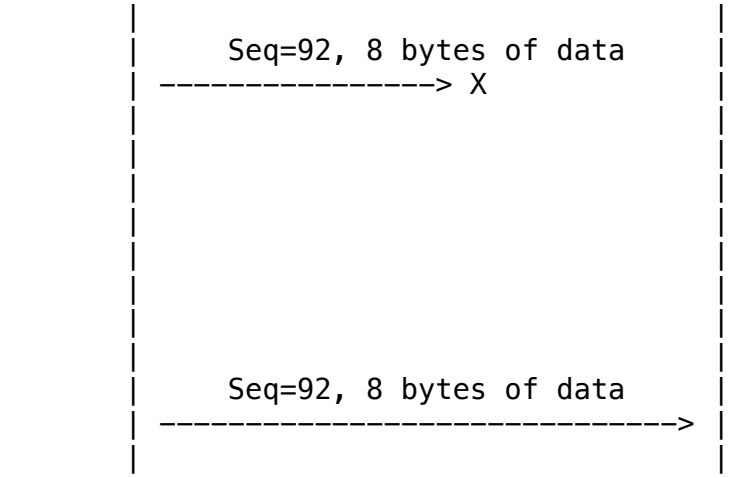
- February 12th, 2021
  - TCP Reliable Data Transfer (1)
    - Cumulative ACKs
      - This mechanism is utilized in TCP to bring more tolerance to lost acknowledgements
      - Every single acknowledgement will indicate in the number field the bytes that the receiver has received so far
        - Even if some acknowledgements get lost, subsequent acknowledgements can still be utilized to acknowledge previously received segments
  - TCP Reliable Data Transfer (2)
    - Retransmission can be triggered by timeout
      - This occurs if an acknowledgement does not arrive within a certain amount of time for a transmitted segment
        - The TCP sender will retransmit the corresponding segment
      - This tends to be a slow process because the timeout value is set to multiple RTTs
        - If possible, we try to avoid this mechanism
  - TCP Reliable Data Transfer
    - Retransmission can be triggered by duplicate ACKs
      - More efficient than the timeout method

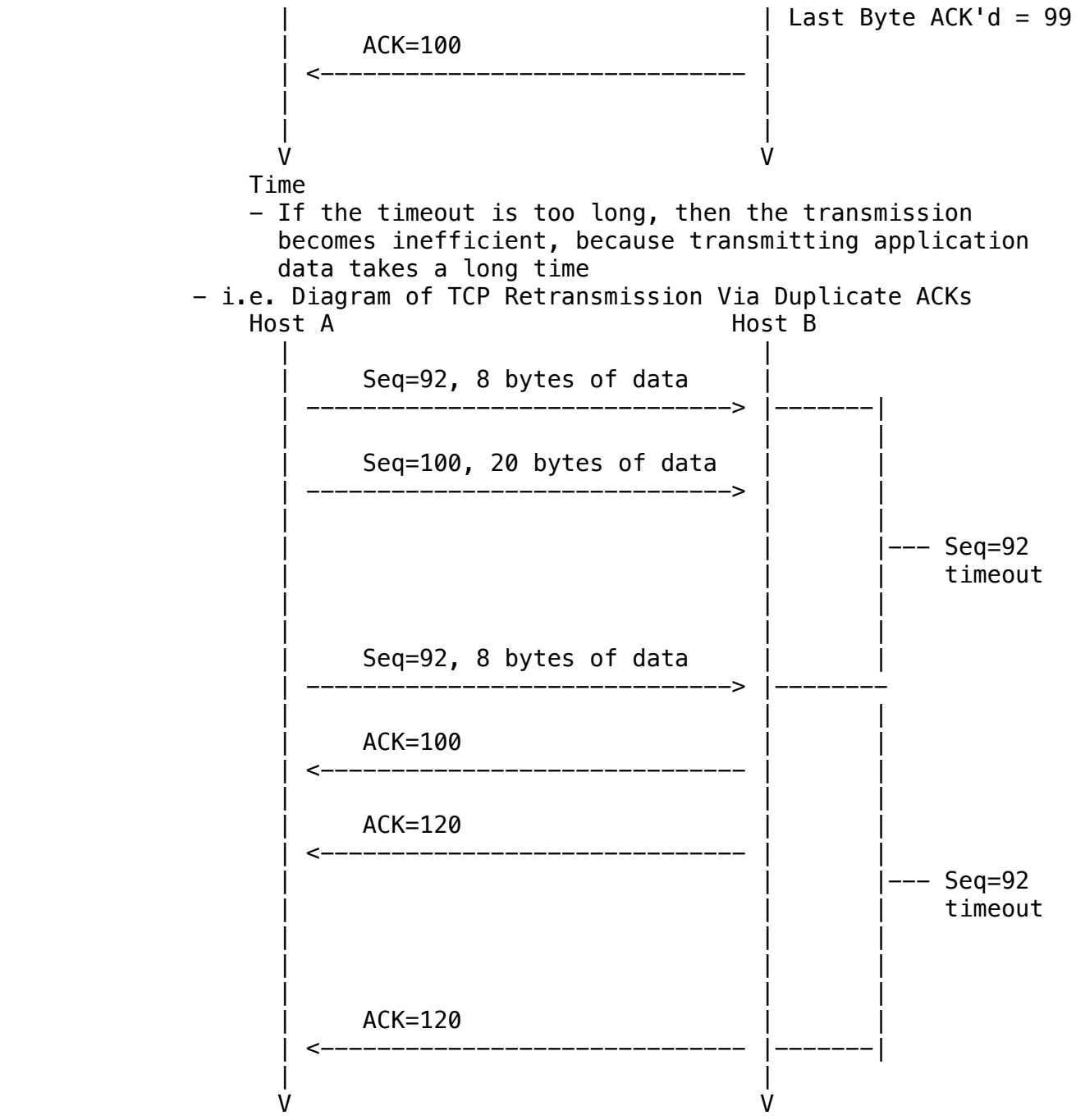
    - Example is below
- i.e. Diagram of Data Transfer Via TCP



    - Acknowledgement number 92 is sent to the sender multiple times
      - These duplicate ACKs trigger a retransmission of the segment that corresponds to sequence number 92
      - Every time the sender transmits a segment that is not #92, the receiver sends an `ACK=92`
        - This tells the sender that the segment with sequence #92 has been lost
      - Upon receiving the third duplicate acknowledgement, the sender retransmits the lost segment
    - It takes a little over one RTT for the sender to determine that a segment has been lost

        - This mechanism is faster than retransmission by timeout
    - It takes 3 duplicate acknowledgements for retransmission to be triggered
        - Setting this number to be very small can cause the sender to retransmit packets that have already been sent
        - Setting this number to be very large won't work because there might not be enough buffer space
- TCP: Retransmission Timeout (1)
  - TCP responds differently to timeouts and duplicate ACKs
    - A timeout is an indication of severe congestion in the network
    - Duplicate ACKs indicate:
        - Lost packet due to random variation
          OR
        - Minor congestion in the network
  - Based off the timeout, TCP will change the rate at which data is transmitted
  - If the sender has not received an ACK by timeout, retransmit the first packet in the window, and restart timer
  - How do we pick a timeout value?
    - If the timer utilized to trigger retransmission timeout is too short, then we may prematurely timeout, and waste network resources by retransmitting something that has already been delivered
    - If the timeout is too large, then the TCP sender has to wait a long period of time before it can recover from packet loss
        - This will slowdown the transmission of the application data
    - It is very important to have a suitable value to determine the timeout interval
- TCP: Retransmission Timeout (2)
  - i.e. Diagram of TCP Retransmission Via Timeout





- If the timeout is too long, then the transmission becomes inefficient, because transmitting application data takes a long time

- i.e. Diagram of TCP Retransmission Via Duplicate ACKs

- If the timeout is too short, then duplicate packets are transmitted
  - This occurs because the timer runs out before the acknowledgement for the first segment arrives
- Ideally, the timeout should be set to `c * RTT`, where `c` is a constant that varies based on the network

- TCP: Round Trip Time
  - Question: How to estimate RTT?
    - SampleRTT: Difference in time between the ACK receipt

        and when the segment was initially transmitted
          - Retransmissions are ignored for accuracy reasons; we cannot differentiate if received ACKs are for the initial segment or retransmitted segment(s)
          - SampleRTT is only measured for segments that are not retransmitted
      - The SampleRTT will vary for each segment; thus taking the current SampleRTT is a bad approach
          - To obtain a "smoother" RTT, we average out several recent measurements
- Example RTT Estimation
  - SampleRTT tends to be very dynamic
      - It fluctuates a lot
  - The instantaneous round trip time varies a lot because of the queueing delay along the routers from the sender to the receiver
  - To smooth out SampleRTT we need to do some kind of low pass filtering to remove the high frequency components of the SampleRTT
      - There are many sophisticated filters that can do this
          - In TCP, this is done using exponential weighted moving average
- TCP: Round Trip Time & Timeout (1)
  - EstimatedRTT = ((1 - 'alpha') * EstimatedRTT) + ('alpha' * SampleRTT)
          - The first part of the equation is the history
          - The second part of the equation is the observation
          - This equation is quick and easy to compute
              - It is also storage efficient because previous values are not saved anywhere
      - Exponential weighted moving average
          - Over time the older history weighs less-and-less, and the recent observations will have a greater effect on the EstimatedRTT
      - Influence of past sample decreases exponentially fast
          - If 'alpha' is very large, then the equation puts more weight on the SampleRTT
          - If 'alpha' is equal to 1, then EstimatedRTT is equal to SampleRTT
              - This is bad because SampleRTT fluctuates a lot
          - If 'alpha' is relatively small, then EstimatedRTT will change very slowly, and won't account for the latest observations
      - Typical value of 'alpha' = 1/8 = 0.125
          - There needs to be a good tradeoff between the weight of EstimatedRTT and SampleRTT
          - Most TCP implementations give history (EstimatedRTT) a weight of 7/8, and latest observation (SampleRTT) a weight of 1/8
              - In other words, historical data weighs more than

observation data
- TCP: Round Trip Time & Timeout (2)
  - Since the instantaneous RTT fluctuates around the EstimatedRTT, we need to add a margin on top of the EstimatedRTT to achieve a better timeout value
    - This is like adding a "safety margin" to the EstimatedRTT
      - Large variation in EstimatedRTT results in larger safety margin
  - Calculating the timeout value:
    - First estimate of how much SampleRTT deviates from EstimatedRTT:
      - DevRTT = ((1 − 'beta') ∗ DevRTT) + ('beta' ∗ | SampleRTT − EstimatedRTT |)
        - The first part is the history
        - The second part is the newly sampled standard deviation
      - Typically, 'beta' = 0.25
        - 'beta' has the same effect as 'alpha'
        - More weight is put on history, and less on the newly estimated standard deviation
      - The DevRTT is kind of like a standard deviation of 'RTT'
        - DevRTT is calculated via the exponential weighted moving average
    - Then, set timeout interval:
      - TimeoutInterval = (EstimatedRtt + (4 ∗ DevRTT))
    - Note: The timeout interval is updated for every single SampleRTT that you estimate for a 'fresh' segment that has been acknowledged
  - The timeout value cannot be too conservative or too aggressive
- TCP Receiver Event/ACK Generation
  - The RFCs are 1122 and 2581, respectively
  - The following table summarizes the receiver's behaviour:

Event At Receiver	TCP Receiver Action
Arrival of in-order segment with expected sequence number. All data up to expected seq. number already ACKed.	Delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK.
Arrival of in-order segment with expected sequence number. One other segment has ACK pending.	Immediately send single cumulative ACK, ACKing both in-order segments.

Arrival of out-of-order segment higher than expected sequence number. Gap detected.	Immediately send duplicate ACK, indicating sequence number of next expected byte.
Arrival of segment that partially or completely fills gap.	Immediate send ACK, provided that segment starts lower end of gap.

- Note: All scenarios assume that there is space in the receiver's buffer

- Upon reception of the last segment, that has been delivered in-order, the receiver should generate an acknowledgement in response to the arrival of the last segment
  - The acknowledgement number should equal the segment number of the last segment plus the number of bytes in the payload
  - If a bunch of segments arrive in-order, then TCP will not immediately send an acknowledgement
    - This is known as a delayed ACK
    - TCP does this to cut down on ACK traffic
      - Instead of ACKing every segment, it will ACK every other segment
  - If a segment is received, that arrives in-order, and there is no delayed acknowledgement, then the current acknowledgement is delayed
- If a segment arrives, in-order, and another in-order segment has a pending ACK, TCP sends a single cumulative ACK for the two segments that were received in-order
  - This cuts down the ACK traffic by half, assuming everything arrives in-order and nothing gets lost
- If a few segments are lost in the network, this causes the arrival of out-of-order segments
  - A gap is detected by TCP, due to the sequence numbers not lining up
  - TCP will send a duplicate ACK, indicating the sequence number of the next expected segment
    - Note: The TCP receiver will always acknowledge the last bite received, in-order, plus one
- If the next arrived segment partially/fully fills the gap, on the leftmost side, then TCP will immediately send an ACK
  - The acknowledgement number gets number
- If the segment does not completely fill in the gap (i.e. The segment fills up the right side), then TCP will generate another duplicate acknowledgement to indicate that there may be some missing segments
- If an incoming segment fills in the gap of a buffer, TCP will always send an acknowledgement
  - This acknowledgement varies based on where the gap is filled

        - If the segment fills the leftmost side, then the acknowledgement number is updated
        - If the segment fills the right side, then a duplicate ACK is generated
- TCP: Congestion Control
  - TCP is effective at reliably delivering packets in-order
    - This is accomplished via cumulative acknowledgement, retransmission via timeout or duplicate acks, buffers at sender/receiver, and through the TCP header
  - The purpose of congestion control is to make sure that the sender does not overwhelm the network
    - The sender should not send data at a rate that the network cannot handle
  - In practice, TCP does not know how fast the network can deliver data
    - The available bandwidth in the network is constantly changing
      - i.e. New connections coming in, old connections tearing down, etc.
      - TCP needs to guess how much bandwidth is available in the network, without the help of the router
  - Congestion detection
    - Implemented at the end systems, and not on the routers
      - TCP does not take any explicit signal from the routers
        - However, some routers have an option to tag packets with an early congestion signal. However, this is a binary signal; the router does not explicitly say how much bandwidth it has available
    - When packets are lost, it is often times the result of congestion
      - Loss event = Timeout OR 3 duplicate ACKs
    - Packet loss can also occur due to the type of the network
      - i.e. Wired networks are more reliable than wireless networks. Wireless networks are known to be very lossy
    - When congestion is detected, the TCP sender reduces rate (congestion window) after loss event
      - TCP utilizes a window based mechanism to determine the rate at which it can send data over the network
      - TCP probes the local bandwidth to see how much data it can send
        - It does this at the start of the connection, and after network congestion clears up
  - Rate adjustment (probing)
    - Slow start
    - Additive increase and multiplicative decrease (AIMD)
      - TCP uses AIMD to probe the bandwidth on the network

    - During congestion, AIMD responds aggressively
        - When congestion clears up, AIMD ramps up the transmission rate timidly/slowly
    - TCP is constantly adapting its transmission rate, based on its estimation of available network bandwidth
    - AIMD has a 'sawtooth' behaviour, because the transmission rate is always fluctuating
        - TCP does not flatline
- Conservative after timeout events