Week.4.txt

- February 1st, 2021
    - Transport Layer
        - The application layer and transport layer are connected through the socket layer interface
    - Objectives
        - Understand principles behind transport layer services
            - Multiplexing/Demultiplexing
            - Reliable data transfer
            - Flow control
            - Congestion control
        - Some transport layer services are not unique to the transport layer, and you may encounter some of the services in other layers
            - i.e. (Data) Link layer
        - TCP and UDP protocols
            - Message format
            - Operations
                - Connection setup and tear down of TCP
    - Transport Services & Protocols
        - Provide logical connection between the application processes running on different hosts
        - Transport protocols run in end systems
            - On the sender side, the application message is broken down into units, also called segments, and passed on to the network layer
            - On the receiver side, it reassembles the segments that are sent by the network layer, and puts them together into messages that are accepted by the application layer (and then passes it on)
            - When you send a large file, the transport layer is responsible for breaking down the data into segments that are suitable to be sent over the network
                - This is because particular link layer technologies are only able to support a maximum packet length
                    - If you account for the overhead introduced by the network layer and transport layer, there is a maximum length of the payload of the application layer data
                - Another reason data is broken into small segments is to increase the easiness of reliability
                    - i.e. If the data gets lost, it is easier to recover or retransmit a small segment, than the entire file itself. This helps in minimizing overhead
            - The two most popular transport layer protocols are:
                - TCP
                - UDP
    - Transport Vs. Network Layer

    - The network layer provides logical communication between hosts
    - The transport layer provides logical communication between processes
        - There can be many processes on the same host, that share the same network layer, go through the same network interface, but respond to different processes, and have different sockets associated with their respective process
- Internet Transport Layer Protocols
    - TCP provides reliable, in-order delivery
        - TCP also has other functions such as:
            - Congestion control
                - The transmission rate of the sending process will be regulated so that it will not cause (much) congestion in the network
            - Flow control
                - Helps the receiver buffer in avoiding overflow
                    - The receiver buffer refers to the destination host
            - Connection setup
                - Is able to setup and tear down connections, and maintain state
    - UDP is much simpler than TCP
        - It is a very thin layer on top of IP
        - UDP only supports unreliable, unordered delivery of data between the processes and hosts
            - Think of it as a "no-frills extension of 'best effort' IP"
                - UDP tries its best to deliver the application data from sending process to receiving process, but makes no guarantee of reliability, delivery, doesn't do congestion control, flow control, and it does not maintain connection
            - Has a much lower overhead than TCP
                - Implementing UDP is up to the application developer
    - Neither TCP or UDP will guarantee any kind of:
        - Bandwidth
            - i.e. Fixed rate delivery of application data
        - End-to-end latency
            - Cannot be implemented by the network hosts
        - These services are not available
- Multiplexing & Demultiplexing
    - Are important services performed by the transport layer
    - The network layer provides a logical connection between hosts
        - This connection extends from the network interface card and up to the operating system kernel; it affects the packets that are sent or received to the same host or

transmitted to different hosts

      - But from the application point of view, there might be different application processes that are multiplexing a single network interface
        - This is why there is a need to map multiple processes to the same network layer
          - i.e. This is one of the services provided by TCP or UDP transport layer protocol
    - The network layer does not care which application process, the data comes from
      - It is only concerned with the destination IP address of the data that is sent, and transmitted over the network interface
      - The data is serialized in the network layer
        - But from the application point of view, the data comes from different application processes
          - This process is called multiplexing
  - Multiplexing is the process of gathering data from multiple sockets, and enveloping data with a header
    - The header is later used for demultiplexing
    - In multiplexing, there are multiple incoming data streams from the application, and they are all sent through the network interface
  - Demultiplexing occurs when data is coming from different hosts, and there are different application processes that are destined to the same host
    - i.e. Process #3 on Host #1 sends data TO Process #1 on Host #2
      AND
      Process #4 on Host #3 sends data TO Process #2 on Host #2
      - i.e. Data from Host 1 and 3 are arriving at Host 2, and the packets must be sent to the correct application process
    - Datagrams coming from the network layer are demultiplexed to the different application processes; they are the ultimate destination
      - The port number tells the transport layer which application process the packet is destined for
        - The port number is located in the transport layer header, and is associated with different application processes
    - In summary, the job of demultiplexing is to deliver received segments to correct sockets
  - Multiplexing and demultiplexing are accomplished by both UDP and TCP
    - This is all that UDP implements; in addition to another small service
- How Demultiplexing Works
  - The host receives an IP datagram

    - Each datagram has a source IP and destination IP
    - Each datagram carries 1 transport-layer segment
      - The transport layer segment has a source and destination port number
  - Utilizes the destination port number that is included in the header of the transport layer
    - The host uses IP address and port numbers to direct packets to appropriate sockets
  - In TCP and UDP, the header field of the transport layer contains two very important numbers; source port number and destination port number
    - In a TCP socket, after accepting an incoming connection, a new connection socket is created to exchange data
      - TCP sockets are identified by a 4 tuple: (S-IP, D-IP, SP, DP)
        - Where:
          - S-IP = Source IP
          - D-IP = Destination IP
          - SP   = Source Port
          - DP   = Destination Port
      - If anything in the tuple is different, the packet is sent to the corresponding socket and the associated application process
    - Since there is no notion of connection in UDP sockets, they can accept data from any (other) client
      - Thus, it only needs a destination IP and destination port to identify a particular socket
        - i.e. (D-IP, DP)
          - Where:
            - D-IP = Destination IP
            - DP   = Destination Port
        - If the IP or port is different, then it corresponds to a different socket
        - Datagrams that come from different clients with different source IP and different ports are accepted by the same UDP socket
- Principles Of Reliable Data Transfer Protocols
  - Reliable data transfer is an important service provided by TCP (Transmission Control Protocol)
    - It is used in applications, transport layer, and link layers
    - This service is not unique to the transport layer
      - The (data) link layer may also implement a protocol for reliable data transfer
        - i.e. Reliable data transfer is implemented for lossy links such as wireless connections
        - You don't want to end systems to have a lot of overhead associated with packet recovery
  - There are two mechanisms for reliable data transfer:
    1. Error detection and correction

        - Detect whether the received packet is corrupted
            - i.e. The information sent by the server is not the same as the information received by the client
                - i.e. Bit errors in the received packet
        - In some cases, the error can be corrected after being identified
        - This is one kind of mechanism that can be implemented by different layers
    2. Loss detection and recovery
        - Detect if a packet is missing; even though the server sent it, the client never received it
            - i.e. A packet arriving at a router when its buffer is full may get dropped. In this situation, the receiver will not get the packet at all
- Implementing reliable data transfer mechanisms pose design issues
    - i.e. Should the functionality be implemented in the end hosts or inside the networks
        - Reliable data transfer cannot be fully implemented within the network, and there are applications that do not benefit from functionality inside the network
            - This is why application developers can use TCP or UDP to send data reliably
                - Some apps need reliable data transfer, and some apps can deal with packet loss
                - The IP protocol does not reliably transmit data
    - Efficiency is another issue for error detection and correction, or loss detection and recovery
        - There's a tradeoff between efficiency & reliability
            - i.e. Sending the same message 1000 times is quite reliable, but very inefficient
        - An efficient mechanism that can utilize bandwidth resources properly needs to be developed
            - Utilization is defined by the equation:
              Utilization =
                (Max. app. data rate) / (Available bandwidth)
                    - This is used to judge how good an approach is
            - Given a certain pipe with a fixed bandwidth, you want your mechanism to be able to achieve reliable data transfer, but also achieve it at the maximum possible application data rate it can support
                - The maximum possible value is 100%; all of your network resources are being used to deliver packets from the application
        - All mechanisms utilized for error detection and

correction, or loss detection and recovery incur some kind of overhead
    - i.e. Adding extra bits
    - i.e. Packet retransmission
- Error Detection
    - Is useful during retransmission of data
    - Can also be utilized at end systems
        - i.e. Detect if a file has been corrupted or not
    - The purpose of error detection is to detect whether there are any bit errors in the received packets or frames
        - They key idea in solving this is to add some extra bits to detect whether an error has occured or not
            - This adds overhead
    - Different techniques for error detection:
        - Parity check
            - Often used in other contexts
                - i.e. Filesystems
        - (Internet) Checksum
            - Currently utilized in the network and transport layer for testing whether the packet has been received correctly
        - Sophisticated coding schemes
            - i.e. Reed-Solomon code

- February 3rd, 2021
    - Review Of Previous Lecture
        - Transport layer provides logical communication between processes on end hosts
        - Network layer provides logical communication between network hosts
        - The transport layer should be distinguished from the network layer because there may be multiple processes running on the same host that are sending or receiving packets from some other host
        - One of the most important services provided by the transport layer is its ability to demultiplex and multiplex the data
        - The TCP protocol provides the ability to send data in a reliable manner, and ensure in-order data delivery
        - There are two key mechanisms to ensure reliable data transfer
            1. Error Detection
                - This is the ability to detect if an error has occurred in a packet, and (possibly) correct the error
            2. Loss Detection
                - This mechanism can detect if packet loss has occurred during forwarding of the packet inside the network
                    - It also tries to recover the packet
        - When designing reliable data transfer mechanisms, we need to

be concerned with correctness and efficiency

- Correctness is the ability to achieve reliable data transfer
- Efficiency is utilizing bandwidth resources properly
    - Spending a lot of network resources to achieve high reliability is highly undesirable, because it is too costly to do so
    - Efficiency can be measured by the equation:
      Utilization = (Max. app. data rate) / (Avail. band.)
        - Max   = Maximum
        - App   = Application
        - Avail = Available
        - Band = Bandwidth

- Error Detection
    - The purpose of error detection is to be able to detect bit error in packets
        - Packets may be corrupted during transit
            - This is especially true in the case of wireless communication
                - A typical wireless link will have a bit error rate of 10^-6; 1 bit out of every million will be corrupted
    - Bit corruption in packets is detected by adding redundancy, extra bits, to each packet
        - i.e. Parity check, checksum, and other sophisticated coding schemes like Reed-Solomon code
- Parity Checking (1)
    - Commonly used error detection mechanism
        - Not only is it used in network protocols, but also file systems
    - It is a very simple approach and has relatively low overhead
        - But it also has limited capability in terms of error detection
    - Single Bit Parity
        - Detect single bit errors
        - The idea is to take the information bit you receive, and add an extra bit at the end that is called a parity bit
    - There are two kinds of parity checking
        A. Odd parity check
            - Steps
                1. Sum up all the information bits and mod 2 it. The information bits are the "1's"
                2. If the result from Step 1 is 0, add 1 as the parity bit, otherwise add 0. In other words, if there is an even number of "1's", then the parity bit is 1. And if there is an odd number of "1's", then the parity bit is 0
                - i.e. 0111000110101011 has a parity bit of 0
                  Because there are 9 "1's", and 9 mod 2 is

1, so the parity bit is 0

- Combining the information bits with the parity bits will give you an odd number of bits in the resulting packet

B. Even parity check
- Steps
  1. Sum up all the information bits and mod 2 it. The information bits are the "1's"
  2. If the result from step 1 is 0, then add 0 as the parity bit, otherwise add 1. In other words, if there is an even number of "1's", then the parity bit is 0. And if there is an odd number of "1's", then the parity bit is 1
  - i.e. 011100011010111 will have a parity bit of 1 Because there are 9 "1's", and 9 mod 2 is 1, so the parity bit is 1
- Combining the information bits with the parity bits will give you an even number of bits in the resulting packet

- The receiver knows which parity is being used before the packets are sent; end hosts agree on parity checking in the setup phase
  - The receiver uses the parity bit to check if there are any errors in the received message(s)
- Example (What is the parity):
  - Question:
    A web server sends "0010101" to a client. Assuming that the end-hosts have agreed upon using even parity to check for errors, calculate the parity bit? (Hint: Use even parity-checking)
  - Answer:
    0010101 contains 3 information bits. 3 mod 2 is 1, so the parity bit is also 1
- Example (Checking the parity bit):
  - Question:
    A client host sends "00101010" to a web host. This message contains the parity bit; it is the right most bit. Assuming that both end-hosts have agreed upon using even parity to check for errors, is the message corrupted?
  - Answer:
    00101010 can be divided into 0010101 and 0, where the message is 0010101 and the parity bit is 0. The parity bit of the message is 1 because there are 3 information bits, and 3 mod 2 is 1. Therefore, this message contains a bit error.

- Parity checking cannot tell which bit is the error bit. The error bit can be first one, last one, or any bit in between. Furthermore, parity checking cannot tell how many errors are present, nor can it tell the location of the errors

  - Overhead
    - The overhead introduced by parity checking can be calculated using the following equation:
      R = (# of bits used in total) / (# of bits in message)
      R = (d + p) / (d)
      - Where:
        - d = Stream of data bits (i.e. size of payload)
        - p = Total # of parity bits
    - As the length of the bit stream increases, the overhead decreases
      - Parity checking is a very simple approach, does not add a lot of redundancy, and is very easy to compute
        - However, it cannot detect exactly how many bits are corrupted, and their location in the bit stream
  - Single bit errors can be detected, but multiple bit errors cannot be detected properly or at all
- Parity Checking (2)
  - Stronger error detection and correction is achieved by adding more redundancy to the data
    - i.e. Two dimensional bit parity
  - Two dimensional bit parity
    - It can detect and correct single bit errors, because it includes more parity bits
    - The idea is to organize the bit stream in a (M x N) matrix, and then take the parity for each row and column in the matrix
      - i.e.

        ```
        |  Information Bits   |///////////|
        |---------------------|-----------|
        |  d_1,1  ...  d_1,j  | d_1,j+1   |
        |  d_2,1  ...  d_2,j  | d_2,j+1   |
        |   ...   ...   ...   |  ...      | ROW PARITY
        |   ...   ...   ...   |  ...      |
        |  d_i,1  ...  d_i,j  | d_i,j+1   |
        |---------------------|-----------|
        | d_i+1,1 ... d_i+1,j | d_i+1,j+1 |
        |---------------------|-----------|
             COLUMN PARITY
        ```

      - i.e.

        ```
        1 0 1 1 | 1
        0 1 0 1 | 0
        1 0 1 0 | 0
        0 0 1 1 | 0
        --------|--
        0 1 1 1 | 1
        ```

      - i.e.

        ```
        1 0 1 0 1 | 1
        1 1 1 1 0 | 0
        0 1 1 1 0 | 1
        ```

```
──────────|──
0 0 1 0 1 | 0
```

- - - In the end, you end up with (M + N + 1) parity bits
      - This is the added redundancy
    - In two dimensional bit parity, you can correct a single bit error
      - If the row and column parity bit do not line up, then the information bit at row 'i' and column 'j' is incorrect
    - Redundancy can be calculated using the formula:
      R = (# of bits used in full) / (# of bits in message)
        = [(M + 1)(N + 1)] / [(M)(N)]
    - The added redundancy in two dimensional bit parity is greater than the added redundancy in one dimensional bit parity
      - However, the benefit is that we get to correct errors in addition to detecting them
        - Note: Error correction only works in certain situations. If there are too many errors in the bit stream —— multiple errors in the same row or column —— then the packet cannot be fixed, because it is impossible to tell exactly which bit is corrupted, and guessing is not feasible.
  - When dealing with single bit parity, both odd parity checking and even parity checking will get the job done
    - However, the sender and receiver need to agree on a mechanism beforehand
  - When dealing with two dimensional bit parity, it is recommended to always use even parity; for consistency
    - This is because odd parity may not produce a consistent value for the last parity bit when certain values of 'M' and 'N' are used
  - Two dimensional parity checking is not (really) used in the TCP/IP protocol stack
- Another Example
  - i.e. Calculate the two dimensional (even) parity

```
1 0 1 0 | 0
1 1 1 0 | 1
0 1 0 1 | 0
--------|---
0 0 0 1 | 1
```

- Internet Checksum (1)
  - Predominantly used in TCP/IP
    - This is used in the network layer and the transport layer for error detection of the transport layer segments or network layer packets
  - Works by corresponding to the 16-bit one's complement of the one's complement sum of all 16-bit words in the content to be protected

```
    - The two's complement sum is: summing the numbers (with
      carries)
        - i.e.
            1 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0
            1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
            ---------------------------------
            1 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1
    - One's complement sum is: summing the numbers and adding the
      carry (or carries) back to the sum
        - The idea is to keep the number at 16-bits by adding the
          carries back to the sum
            - The length of the number stays the same
        - i.e.
            The two's complement from above is:
            1 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1
            |
            |--------------------------------> Add back to sum
            After computing the one's complement, the sum is:
            1 0 1 1 1 0 1 1 1 0 1 1 1 1 0 0
            The number remains at 16-bits
    - In an IP packet, only the header is protected
        - The header field is broken into 16-bit numbers
            - Then they are added together, the bits are carried
              over, and the result is one 16-bit number; this is
              the one's complement sum
    - In the transport layer, the checksum is calculated for the
      header information & payload of the transport layer segments
        - Everything is broken down into 16-bit numbers, summed
          up, and carries are added back into the sum to form the
          final sum
        - The last step is to take the one's complement of the
          final sum
            - This is the checksum
            - The one's complement of any bit stream is calculated
              by flipping the bits
                - "1's" become "0's" && "0's" become "1's"
                    - i.e.
                        Sum      = 1 0 1 1 1 0 1 1 1 0 1 1 1 1 0 0
                        Checksum = 0 1 0 0 0 1 0 0 0 1 0 0 0 0 1 1
                - Adding the sum and checksum gives you all "1's"
                    - All "1's" is also known as "FFFF"
                - Subtracting the sum from "FFFF" gives you the
                  checksum
    - The (internet) checksum is exactly 16 bits because their is
      a fixed field in the header of the transport layer or IP
      header that is used to store the checksum information
        - This is why bits are carried and added back to the sum
            - The length of the (check)sum cannot and should not
              exceed 16-bits
- Internet Checksum (2)
```

- Checksum is utilized by both sender and receiver in the following manner:
  - Sender
    1. The contents of the message that will be protected are broken down into sequences of 16-bit integers
    2. Checksum is computed by adding all the 16-bit sequences, and then taking the one's complement of the sum
    3. Checksum value is added to the checksum field in the corresponding packet header of the protocol
  - Receiver
    1. Compute checksum, in the same manner as sender, of the received packet
       - The checksum field is not part of the original message, and is not included when the checksum of the packet is computed
    2. Check if computed checksum equals the checksum in the checksum field in the packet header
       - If the values are not the same, then error is present
       - If the values are the same, then an error was not detected, but it is possible that there is an error, and (internet) checksum cannot detect it
         - Occurs due to multiple bit errors

- Example (Checksum)
  - Question
    - Compute the checksum of the following message:
      01 00 F2 03 F4 F5 F6 F7 00 00
  - Answer
    1. The first step is to break the message into 16 bits
       0100
       F203
       F4F5
       F6F7
       0000
    2. Compute the 2's complement sum by adding up all the values:
       0100 + F203 + F4F5 + F6F7 + 0000 = 2DEEF
    3. Compute the 1's complement sum by adding the carrier bit to the 16-bit sequence:
       2 + DEEF = DEF1
    4. Compute the checksum by flipping the bits; in hex, subtract the 16-bit sequence from "FFFF":
       FFFF - DEF1 = 210E
    5. Verify your answer by adding 1's complement sum with checksum to get "FFFF":
       DEF1 + 210E = FFFF
    6. The checksum is: 210E
  - Notes

    - The sender side adds the checksum, 210E, into the checksum field of the corresponding packet header, and sends it to the destination host
      - The receiver side gets the original message and checksum of the message
        - i.e. 01 00 F2 03 F4 F5 F6 F7 00 00 210E
  - Question
    - If the destination host recieves the following message:
      01 00 F2 03 F4 F5 F6 F7 01 00
      Checksum: 210E
      Is the message corrupted?
  - Answer
    - Yes, the message has been corrupted, the 3rd bit from the right is incorrect; it should be 0
  - Notes
    - It is possible that multiple messages with different information content can have the same checksum
      - This is because checksum is not inversible
        - i.e. The following messages have the same checksum:
          01 00 F2 03 F4 F5 F6 F7 00 00
          01 01 F2 02 F3 F5 F7 F7 00 00
          - Note: All I did here was subtract 1 from a 16-bit sequence and add it to another 16-bit sequence
- Checksum In UDP
  - Even though UDP does not implement reliable data transfer, it does detect bit errors in the UDP datagram
    - It does this by using checksum; which is included in the checksum field in the UDP header
      - i.e.

```
<------------32_bits----------->
|---------------|--------------|
| Source port # | Dest. port # |
|---------------|--------------|
| Length        | Checksum     |
|---------------|--------------|
|                              |
|         Application          |
|            Data              |
|          (Message)           |
|                              |
|------------------------------|
```

      - The UDP header contains information that is used to address the processes that are communicating with each other
        - i.e. Source port number, destination port number, etc.
      - The length field contains the length of the UDP segment including the header

            - Indicates how many bytes are present, in total
                - Length = length(header) + length(payload)
        - The checksum field is used to check if the contents of the message has been changed
            - In the TCP/IP protocol stack, the UDP checksum is computed from the payload, header field, AND from the pseudo header
                - The pseudo header contains the IP header information
                    - i.e. Source IP, destination IP, reserve field, protocol value, and length of IP header
    - The checksum is computed using the payload and a "pseudo header" that contains some of the same information from the real IP header
        - This shows that the layering of the protocol stack is not strictly followed
            - i.e. The UDP checksum takes into account the IP header (referred to as pseudo-header) and all of its corresponding fields, UDP header and all of its corresponding fields, and the payload/ message/application data
        - If there is corruption in the IP header, then it will be detected by the checksum in the UDP header
            - However, the IP header has its own checksum
- Loss Detection
    - There are situations where packets can get completely lost
        - The receiver doesn't even know if the packet was sent or not
    - Causes of packet loss
        - Buffer overflow
            - If the network is heavily congested, a packet arriving at an incoming or outgoing buffer of a router will get dropped because there is no room in the router's buffer
        - Drop after error detection
            - If the router checks the integrity of the packet via checksum, and finds that the packet is corrupted, it will drop the packet
                - The problem is that the router does not notify the sender about this
                    - The sender needs to find a way to detect packet losses inside the network
                    - The router does not notify the sender of dropped packets, because it will add complexity inside the network
                        - Furthermore, if the message from the router gets lost, or corrupted, then the message won't be delivered to the sender
                    - Reliable data transfer is left up to the end

            systems to implement
                - This is the end-to-end design principle
- Detection methods
    - At the crime scene
        - The router handles notifying the sender of dropped packets
            - This is a bad approach because:
                - It adds complexity inside the network
                - Does not fully solve the initial problem entirely
    - At the receiver
        - The destination host can recognize gaps in the received packets, and notify the sender of the situation
        - The receiver needs to be able to detect multiple packet loss
        - The receiver needs help and additional information from the sender in order to detect what packets are missing
- Loss Recovery
    - The sender will insert a sequence number into the packet
        - Sequence numbers increase monotonically
        - If the receiver gets the 1st, 2nd, 3rd and 5th packet, but not the 4th packet, it will be able to detect that something bad has happened
            - This mechanism is also used in the link layer
                - The Wi-Fi receiver can also detect if packet loss has occured
    - Once the receiver detects packet loss, the source needs to be informed
        - It can do this in two (somewhat) different ways:
            - Negative ACK (NACK)
                - Notifies sender that "packet xx is missing"
            - Positive ACK (PACK)
                - Notifies sender that "packet xx has been received"
                    - The sender has to determine which packet has not been received using the PACKS
        - In the negative ACK, the sender knows exactly which packet has been lost and needs to be resent
            - On the other hand, in positive ACK, the sender knows what has been received, but needs to infer what has been lost
    - Once the sender is informed about packet loss, it can do different things like:
        - Wait until the acknowledgement for the previous message, before it transmit the next message,
        - Send a bunch of messages in succession, and wait for feedback from the destination host to see which packets it needs to retransmit

      - For retransmission, the sender needs to determine if it should:
        - Retransmit every un-ACK-ed packet
        - Selectively retransmit the lost packet only
      - The sender can do a number of different things upon detecting packet loss
        - Each technique has its own efficiency and efficacy
          - In terms of utilization of the network end hosts
      - When designing your own reliable data transfer protocol to recover from packet losses, you need to consider the following:
        - Should I use NACK or PACK on the receiver side?
        - When does the sender need to decide to respond to packet loss?
          - What packet needs to be retransmitted?
        - The tradeoffs for each mechanism
        - How efficient each mechanism is
  - Questions
    - Parity checking is faster than internet checksum for error detection
      - However, checksum is more accurate for error detection
    - TCP takes care of packet losses and packet errors
      - Application processes don't have to deal with packet loss/errors
      - If TCP detects packet loss, it will trigger re-transmission, and eventually all the application data will be transmitted reliably and in-order
    - When using UDP, you have to implement an additional mechanism in your application to handle packet losses
    - Corrupted packets are (almost) never delivered to the application; they are dropped somewhere by the network layer or transport layer
      - Even UDP performs checksum
      - If the network layer detects a corrupted packet, it will drop the packet and not further deliver it
    - On the server side, when you do socket programming, you need to create a socket and associate it with a particular port
      - This is done by binding the socket to the port
        - The server can now accept incoming connection requests destined to that port

- February 5th, 2021
  - Recap
    - Last class we talked about:
      - Techniques for error detection
      - Parity checking
        - Two dimensional parity checking
      - Internet checksum
        - Utilized in network layer and transport layer for detecting errors in the packet

- Flow Control
  - May be provided by transport layer
  - In internet terms, flow control aims to control the rate of the sender, so it does not overwhelm the receiver
    - There is a possiblity that the sender and receiver, process packets at different speeds
      - i.e. Sender is on optic fibre internet, and receiver is on dial-up internet. The sender's internet speed is very fast, and the receiver's speed is very slow
      - i.e. Multiple processes are running on the receiver side that need to communicate with multiple hosts. Thus, the receiver may not be able to process incoming packets fast enough
      - In these situations, it is important to control the flow of packets from the sender(s), so it does not overwhelm the receiver
    - The goal is to match the speed of the sender to something that the receiver can process
      - i.e. Rate limiting
        - Sender and receiver negotiate the speed; receiver tells sender that it will only accept data at a rate of 100 kilobits per second
          - This is an explicit way to flow control
      - i.e. Window based control
        - In TCP, the receiver notifies the sender through information in the packet header, and tells the sender that the receiver can only accept a certain number of bytes from this point
          - If the sender tries to send more bytes than what TCP allows, then the transmission speed is throttled
- Congestion Control
  - May be provided by transport layer
  - Differs from flow control
    - The primary goal of flow control is to not overload the receiver
  - The purpose of congestion control is to avoid overwhelming the network
    - In the network, a lot of traffic originates from different hosts
      - It is possible for the incoming traffic to be greater than the maximum outgoing traffic that can be processed by the outgoing interface
        - Congestion may occur
  - Congestion occurs when too many sources send too much data, that is too fast for the network to handle
    - This is the informal definition
  - When congestion occurs, the router's queue will build up either at the incoming interface or optical interface

      - At this point, packets are put in a wait queue before they are processed
        - Packets experience a longer delay
        - When the buffer becomes full, new incoming packets are dropped
    - Congestion inside the network can manifest as:
      - Long delays
        - i.e. Queuing in router buffers
      - Lost packets
        - i.e. Buffer overflow at routers
  - Causes/Costs Of Congestion: Scenario 1
    - Congestion is a bad thing, and should be avoided
      - A packet waiting in a queue to be processed is a waste of time
      - The negative effect of congestion goes beyond packet losses or longer delay
        - In some situations, a lot of senders can be busy stuck sending packets, but none of the packets get through
    - Example Diagram
      - i.e.

```
|--------|                                 |--------|
| Host A |->---|                     |->-->| Host C |
|--------|     |---->|--------|->----|     |--------|
                     | Common |
                     | Router |
|--------|     |---->|--------|->----|     |--------|
| Host B |->---|                     |->-->| Host D |
|--------|                                 |--------|
```

      - The example has two senders, and two receivers:
        - Host A sends packets to Host C
        - Host B sends packets to Host D
      - There is one router, and an infinite buffer
        - Host A and Host B share a common router
          - This is the bottleneck
          - The data rate of the router's interface is limited to R
        - Since the buffer is infinite, a packet can never get dropped
      - 'lambda_in' represents how fast the sending application process can send data
      - 'lambda_out' represents how fast the receiving application process can receive data
      - Host A and Host B generate data at the same rate
        - If they generate a low amount of data, then it will reach the destination host without delay
          - The network is not yet congested
          - There is enough bandwidth at the router to deliver incoming packets
      - If incoming data rate approaches the link bandwidth (R),

then the delay goes up and approaches infinity

        - The link bandwidth is the bottleneck
            - The bottleneck is occurs when the incoming data rate equals the link bandwidth
        - In this case, packets arrive in a stochastic manner, rather than a fixed interval time
        - There is no packet loss because the size of the buffer is infinite, so queuing delay increases
            - Since there is no packet loss, there is no retransmission
    - Regardless of the value of 'lambda_in', the 'lambda_out' value is limited by the smaller value of 'lambda_in' and half of the rate
        - This is because the outgoing interface at the common router cannot deliver at a rate that is bigger than the link bandwidth
    - The maximum per-connection throughput is R/2
        - 'lambda_out' for Host C and Host D is <= R/2
            - This is because they are limited by the outgoing interface of the common router
            - It does not matter how fast Host A and Host B transmit data, the maximum per-connection throughput is R/2
    - As 'lambda_in' approaches R/2, the amount of delays increases
- Causes/Costs Of Congestion: Scenario 2 (1)
    - The following scenario is similar to scenario 1, but now the router has a finite buffer
        - Packets can be dropped/lost
        - This models the real internet
    - Example Diagram
        - i.e.

```
|--------|                                   |--------|
| Host A |->---|                       |->-->| Host C |
|--------|     |---->|--------|->----|       |--------|
                     | Common |
                     | Router |
|--------|     |---->|--------|->----|       |--------|
| Host B |->---|                       |->-->| Host D |
|--------|                                   |--------|
```

    - 'lambda_in' corresponds to how much data the sending applications can send to the router
    - 'lambda_in_prime' is 'lambda_in' plus retransmitted data
        - Since the router has a buffer, packets can be dropped which requires them to be retransmitted
        - If packets are lost/dropped, then the transport layer will try to recover or retransmit them
        - 'lambda_in_prime' is greater than or equals 'lambda_in'
    - If a packet is lost at the router due to a full buffer, the

sender retransmits the lost packet
    - In the application layer: 'lambda_in' = 'lambda_out'
    - In the transport layer: 'lambda_in_prime' >= 'lambda_in'
      - This is because the transport layer handles retransmission, and not the application layer, for the TCP protocol
- Causes/Costs Of Congestion: Scenario 2 (2)
  - In the case that there's a light/moderate amount of traffic, the receiver side gets data at the same rate as the sender side transmits data
    - 'lambda_in' == 'lambda_out'
      - The receiver downloads the data at the same rate the sender uploads it
    - 'lambda_in' == 'lambda_in_prime'
      - This is because there is no packet loss, thus there is no retransmission
      - Application layer data rate is equal to the transport layer data rate
    - This is a perfect rate control
  - If 'lambda_in' approaches R/2 (the link bandwidth), some packets are retransmitted, including packets that have already been delivered
    - Since the router has a finite buffer, packets arrived at a full buffer are dropped/lost, and the transport layer will try to retransmit those packets
      - In this situation, 'lambda_in_prime' is greater than 'lambda_in'
    - Retransmissions cause the effective transmission rate to drop
  - Cost (repercussions) of congestion:
    - Congestion causes:
      - The network to work harder and retransmit more packets to maintain a given "goodput"
      - More unneeded retransmissions
        - Packets that have already been delivered to the destination host are sent again
          - The link carries multiple copies of a packet
        - If the receiving host does not acknowledge packets in a timely manner, then the sending host will retransmit them
  - If applications ignore the congestion signal from the network, then they can shut down the network by making it unusable
    - i.e. Hosts are busy sending packets, flooding routers, and destination hosts are not receiving anything
  - Once network congestion starts it only gets worse
- Causes/Costs Of Congestion Scenario 3 (1)
  - i.e. Diagram

```
      |--------|      ///--------\\\      |--------|
      | Host A |--------| Router |--------| Host C |
      |--------|      \\\--------///      |--------|
           |                                   |
           |                                   |
           |                                   |
           |                                   |
    ///--------\\\                      ///--------\\\
      | Router |                          | Router |
    \\\--------///                      \\\--------///
           |                                   |
           |                                   |
           |                                   |
           |                                   |
      |--------|      ///--------\\\      |--------|
      | Host B |--------| Router |--------| Host D |
      |--------|      \\\--------///      |--------|
```

        - In total there are 4 senders and 4 receivers
            - Host A and Host D send data to each other
                - Two way communication
            - Host B and Host C send data to each other
                - Two way communication
        - Data from hosts can travel in multiple different ways
            - This is multihop paths
        - The routers have a limited buffer; network congestion can occur, causing timeouts and retransmission
            - Link bandwidth for each router is represented by 'R'
            - Lost/dropped packets need to be retransmitted
        - Every single connection has two hops; the data goes through two routers before it reaches the destination
            - Each connection shares the bottleneck bandwith of R
        - Assume that all 4 senders are transmitting data at the same rate of 'lambda_in'
        - 'lambda_in' represents data sent by the appliation layer
        - 'lambda_in_prime' represents data sent by the transport layer
- Causes/Costs Of Congestion: Scenario 3 (2)
    - If Host A starts transmitting more data to Host D, then the corresponding router starts to get congested
        - Since Host B and Host C use the same router, their packets will start to get dropped
            - As 'lambda_in' for Host A & Host B increases, their corresponding routers start to experience congestion
            - Since packets are getting dropped all over the network, the 'lambda_in_prime' value increases for all Hosts; retransmission is increasing
                - The transport layer ramps up the retransmission
    - Cost (repercussions) of congestion:
        - Congestion caused by one part of the network can lead to congesting the entire network; this is similar to the snowball effect
            - i.e. If a shared router is congested, then all the

routers around it that rely on it for transmitting packets will also get congested. This can continue until the entire network is congested

- If packets are not delivered in a timely manner, the sending host will inject more data (send more packets) into the network
  - As 'lambda_in' and 'lambda_in_prime' increase, 'lambda_out' decreases and eventually drops to 0
    - Eventually, every host is busy transmitting and retransmitting data, but no one is receiving anything
- When a packet is dropped, any "upstream transmission" capacity used for that packet was wasted

- The moral of this lesson is that congestion is evil
  - Not only does it lead to longer end-to-end latency, but it also causes packet loss
    - In response to packet loss, the transport layer retransmits packets which causes more congestion
      - Eventually, the entire network collapses, and a lot of bandwidth is wasted
      - More congestion = No messages being received by the end-hosts
  - You cannot let everyone send an arbitrary amount of data through the network whenever they want
    - Restrictions need to be placed on throughput, in order to avoid network congestion
      - One bad apple can ruin the entire stock/batch

- Approaches Towards Congestion Control
  - There are two broad approaches towards congestion control
    1. End-to-end congestion control
       - There is no explicit feedback from the network
         - The router does not send any signal to the end system and notify it of congestion
       - Congestion is inferred, by the end system, through observed loss and prolonged delay
         - i.e. ACKS from destination host are taking much longer to reach the source host, thus there must be congestion
         - Note: Detecting delay is not instantaneous
       - This is the approach taken by TCP
         - This is because getting feedback from the network about congestion is NOT trivial
    2. Network assisted congestion control
       - This is not widely utilized because it adds complexity into the network
         - The router needs to do more work
       - Routers provide feedback to end system about congestion
         - Instead of relying on the end system to observe losses and delay

- Feedback is provided as a single bit which indicates congestion
  - i.e. SNA, DECbi, TCP/IP ECN, ATM, Etc.
    - ECN = Early Congestion Notification
  - When TCP sends an ACK back to the source host, it can have a field that indicates whether there is some congestion in the net-work or not
- Routers do not cut down on congestion, they notify the end-systems that there is congestion, and it's the job of the end-systems to handle it
  - i.e. Cut down on sending rate

- Bag Of Tricks

Functions	TCP	UDP
Multiplexing/ Demultiplexing	YES	YES
Reliable data transfer	- Checksum - Sequence number - ACK from receivers - Retransmission - Bufferring outsanding packets	- Checksum
Flow control	Throttled by the receiver, send reacts	
Congestion control	End-system estimates and adjusts transmission rate based on congestion "signal" from the network	
Connection establishment/ tear down	YES	

  - Both TCP and UDP have to implement multiplexing and demultiplexing
    - Demultiplexing is the ability to dispatch the data coming from the network layer to different application processes
      - This is required because there can be multiple processes that share the same network interface
    - Multiplexing is sending the data from different application processes through the same network interface
  - For reliable data transfer:
    - TCP

                - Utilizes checksum for error detection
                - Uses sequence numbers for every TCP segment
                    - Allows the receiver to tell if packets have been lost
                - Relies on positive ACKS from the receiver to determine if a packet has been lost
                    - Receiver sends ACK for every received packet
                - In response to packet losses, TCP will retransmit lost/dropped packets
                    - Retransmission increases the amount of data that needs to be sent/delivered/transmitted through the network
            - UDP
                - Provides checksum for error detection
                    - This is the only mechanism that UDP provides other than multiplexing and demultiplexing
        - TCP breaks down application data into multiple chunks,
called

          packets or segments, before sending it
        - Both TCP sender and receiver need to maintain some kind of buffer
            - The TCP sender's buffer stores packets that have not been acknowledged by the receiver
                - They get retransmitted later by TCP
            - The TCP receiver's buffer stores incoming packets, and delivers it in one transfer to the application process instead of sending a tiny segment every time
                - This is useful if TCP segments are not received in the same order they are sent, but the packets are sent to the application process in the correct order/manner
        - TCP performs flow control
            - The receiver will indicate to the TCP sender how much buffer space it has left so the TCP sender can send more or less based on the size of the buffer
                - Used for throttling
        - TCP uses congestion control
            - TCP uses a mechanism to estimate the allowable transmission rate based on current congestion
                - Does not rely on explicit congestion signals provided by the router
                - The congestion situation is inferred based on packet losses and prolonged end-to-end delay that is measured from the sent data and received acknowledgement
                    - i.e. (Time of received ACK) -- (time when data was sent)
                        - Subtract the two values
        - TCP provides connection establishment and tear down
            - UDP does not provide this

      - Before any data can be sent between hosts, a connection needs to be established between the senders
        - This feature is unique to TCP
- UDP & TCP
  - The next lectures will focus on:
    - Understanding the protocol details of UDP and TCP
      - How do TCP and UDP actually work
      - The packet formats of TCP and UDP
        - i.e. Header formats
      - TCP state machine
        - i.e. Connection setup and tear-down process
          - There is a mapping between the socket programming interface and the different states in the TCP connection
      - How does TCP perform flow control and congestion control?
        - i.e. Using some mechanism called the sliding window
- UDP: User Datagram Protocol [RFC 768]
  - Stands for user datagram protocol
  - Is a "bare-bones" internet transport protocol
    - Refer to table above in "Bag Of Tricks"
    - UDP only does two things:
      1. Multiplexing/Demultiplexing
      2. Checksum
  - Provides a "best effort" logical connection between processes and hosts
  - UDP does not try to:
    - Maintain connections
    - Recover lost/dropped packets
    - Deliver packets in the order they are transmitted
    - Establish a connection before transmitting data
      - The benefit of this is: less delay
        - Because it skips the initial round trip connection
  - The reason for having UDP is to give application developers maximum control over the behavior of their application
    - i.e. Establishing a connection takes time, and can be wasteful if a few bytes of data needs to be sent
  - UDP is very simple because it does not maintain any kind of connection state information between the sender & receiver, and it does not do any kind of congestion control
    - The overhead in a UDP header is minimal and very small
      - Compared to TCP, TCP headers have a lot of overhead, because they need to maintain a lot of information about the connection
  - The overhead added to application data is very small when using UDP
  - Since UDP has no congestion control, it can blast away as

fast as "desired"

- UDP: More
    - Often used for streaming multimedia apps
        - Is loss tolerant
        - Is rate sensitive
    - Other UDP uses:
        - DNS
        - SNMP
    - To achieve reliable transfer over UDP, reliability needs to be added to the application layer
        - Packet recovery is application specific
            - Developer decides what to do what dropped packets
                - i.e. Ignore dropped packets, recover every 3rd packet, etc.
    - The packet format of a UDP datagram looks like:
        - i.e.

```
<------------32_bits----------->
----------------|---------------
| Source port # | Dest. port # |
|---------------|--------------|
| Length        | Checksum     |
|---------------|--------------|
|                              |
|          Application         |
|             Data             |
|           (Message)          |
|                              |
--------------------------------
```

        - When displaying a packet it is typically organized into rows of 32 bits
            - This makes it easier for us to determine how long the header is and things like that
            - Each row is 32 bits, and there may be multiple rows
        - The packet header consists of:
            - Source port #
                - Is 16-bits
            - Destination port #
                - Is 16-bits
            - Length
                - Is 16-bis
                - Contains the length, in bytes, of the UDP segment including the header
                    - Length = Total bytes in the header as well as the payload
                    - Limits how big the UDP datagram can be
            - Checksum
                - Is a 16-bit one's complement of one's complement sum of the entire datagram (UDP header and payload) and pseudo header
                    - The pseudo header contains information from

            the IP layer, such as source IP, destination IP, reserved field, protocol field, and length of the IP header

        - Only used for detection; is the packet corrupted or not?
            - If the computed checksum is not equal to the checksum in the UDP datagram, then the packet is corrupted
            - The network layer or application layer decide what to do with a corrupted packet
- The source port and destination port, are transport layer addresses that are used to determine which application processes a particular socket is associated with
- The size of the UDP header, source port #, destination port #, length, and checksum, is 128 bits
    - Each field is 32 bits, there are 4 fields, so 128 bits in total
        - Compared to TCP, the overhead of a UDP header is quite small