Lecture 13: The transport layer

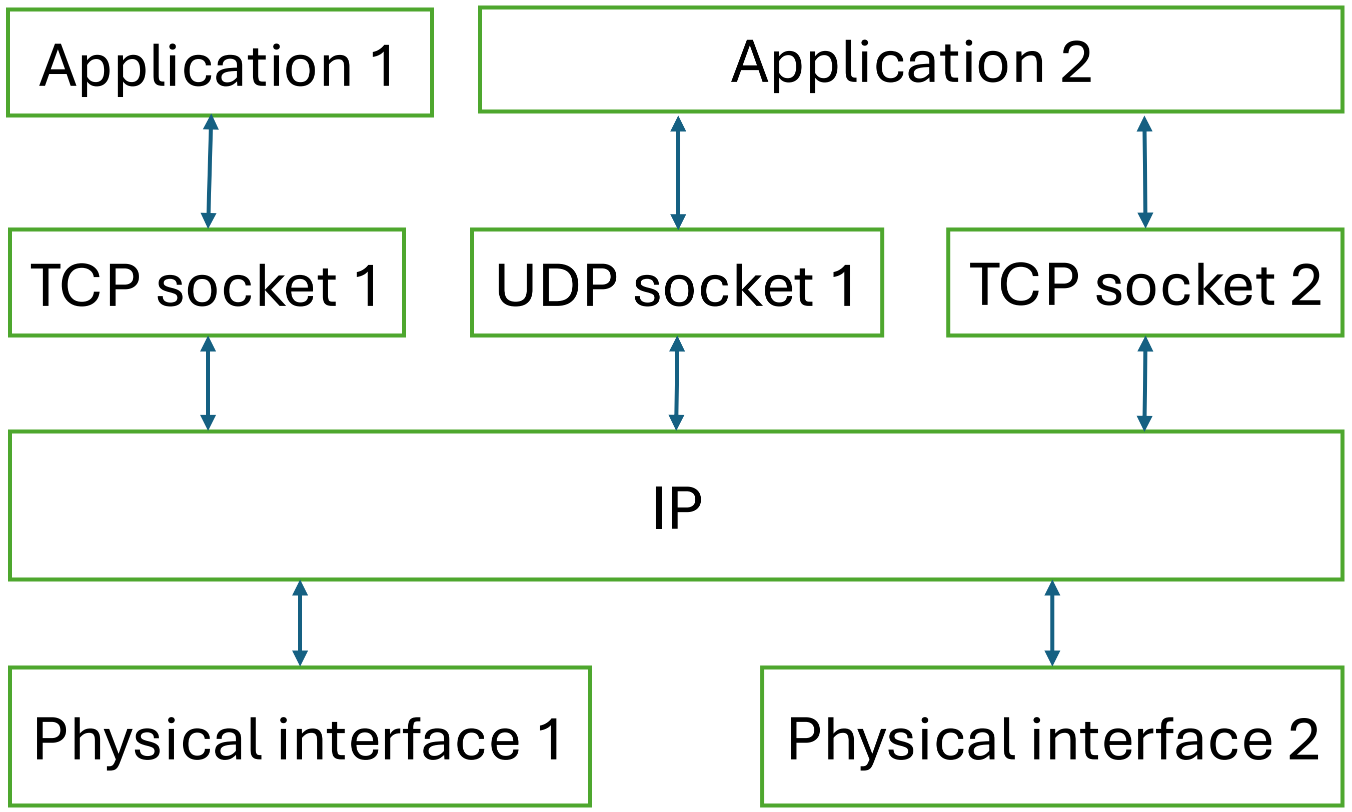
Lecturer: Venkat Arun

Chapters 5.1 and 5.2 from the book

Logistics

- Assignment 3 is out
- We will vote on a proposed change to the grading structure
- Old system:
  - Programming assignments (35%)
  - Two quizzes (30%)
  - Final exam (30%)
  - Class participation (5%)
- Proposed system
  - Programming assignments (40%)
  - Two quizzes (30%)
  - Final exam (30%)
  - Class participation (5% bonus)
- Note: Cheating, for example on the class participation, will NOT be tolerated

Story so far: layers inside a single host



Sockets are associated with applications

What socket to send a received packet to is determined by the src/dst IP and port
All sent packets are sent via IP

Which interface to send a packet via is determined by the routing table.
All received packets go to IP

Story so far

- Physical layer
    - Communication between two “adjacent” nodes.
    - What constitutes as “adjacent” varies.
        - Could mean they are physically close, required for most wireless networks.
        - Could be connected via a wire, or be on the same local ethernet network
        - You could call the communication link between Voyager 1 and the earth a single physical link!
- Network layer (IP)
    - Offers *global, best-effort* packet delivery between any two nodes on the internet*
    - The only layer of the internet which constitutes a *single* protocol. In all the other layers, there are alternatives. Thus, IP is called the “narrow waist” of the internet
- Transport layer
    - Offers process-to-process communication
    - There are multiple protocols specified by an 8-bit field in the IP header
    - TCP (#6) and UDP (#17) are the most common ones used by applications.
    - Routing protocols often have their own protocol numbers.
    - Most assigned numbers are no longer widely used: https://en.wikipedia.org/wiki/List_of_IP_protocol_numbers

*Like most things in this course, there are exceptions. Sometimes we use IP inside local networks. Firewalls also prevent global connectivity. But let us not miss the forest for the trees.

UDP – A simple encapsulation over IP

Header format

Offset	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	*Source Port*																*Destination Port*															
4	32	*Length*																*Checksum*															
8...	64...																	**Data**															

- Super simple header format:
  - Source and destination port – used to determine which socket gets the packet
  - Checksum – Error detecting code for the UDP header and data
  - Length and data – Self explanatory
- Only function is to tell which process to forward the packet to
- Inherits IP's best-effort nature

TCP – A more feature-rich transport layer

TCP has three features:

- **Flow control**: Ensures the receiver has enough memory to receive packets sent by the sender
- **Reliable delivery**: Guarantees that bytes will appear at the receiver in the same order that they were sent. If packets are dropped, it retransmits them

This week

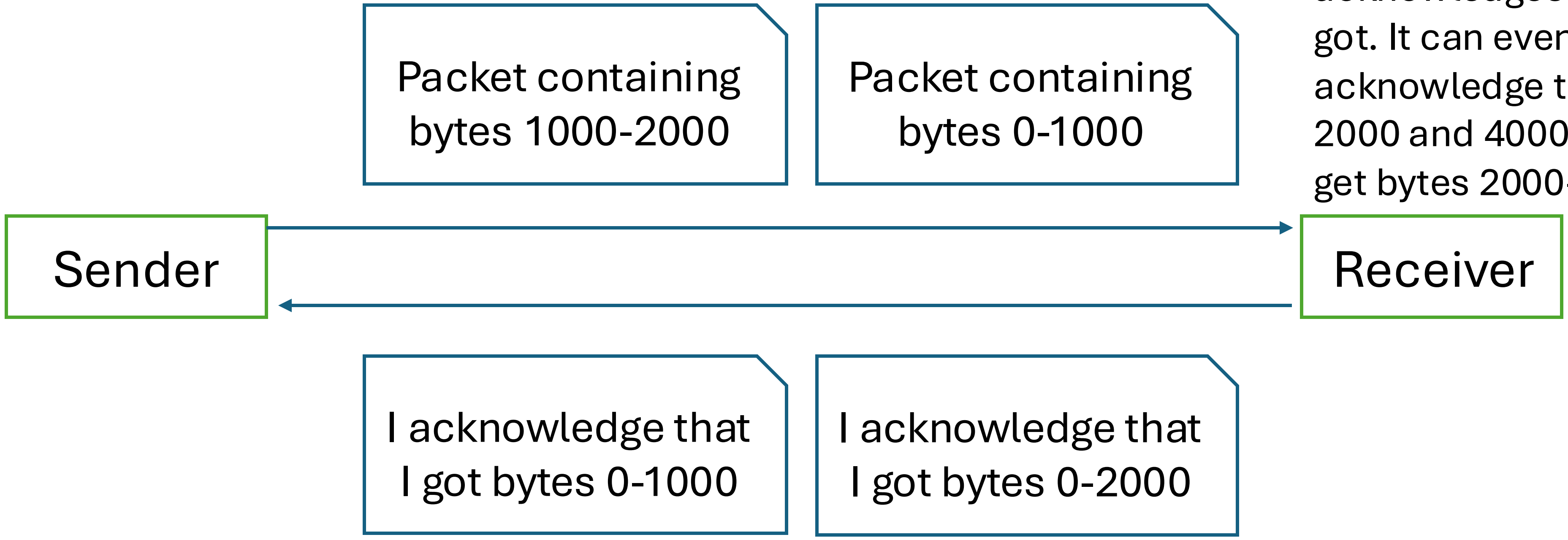
- **Congestion control**: Controls the rate at which the sender sends packets.
  - If it is too fast, the network may get overloaded and drop packets.
  - If it is too slow, performance suffers
  - If multiple TCP flows are bottlenecked at the same link, it tries to ensure that both are sending at roughly the same rate

Next week

TCP communication structure

- Application gives the sender a sequence of bytes. Each byte is assigned a sequence in order. That is, the first byte gets sequence number 0, the second byte gets number 1 and so on…

Note: receiver always acknowledges *all* the bytes that it got. It can even say “I acknowledge that I got bytes 0-2000 and 4000-6000” if it did not get bytes 2000-400


Packet containing bytes 1000-2000
Packet containing bytes 0-1000
Sender
Receiver
I acknowledge that I got bytes 0-1000
I acknowledge that I got bytes 0-2000

TCP is bidirectional

- Thus far, I’ve used the language of “sender” and “receiver” when discussing TCP
- In reality, TCP is bidirectional. That is, each packet can contain data to be sent and an acknowledgment of data sent by the other side
- It is nevertheless easier to think of one end as the sender and the other as a receiver

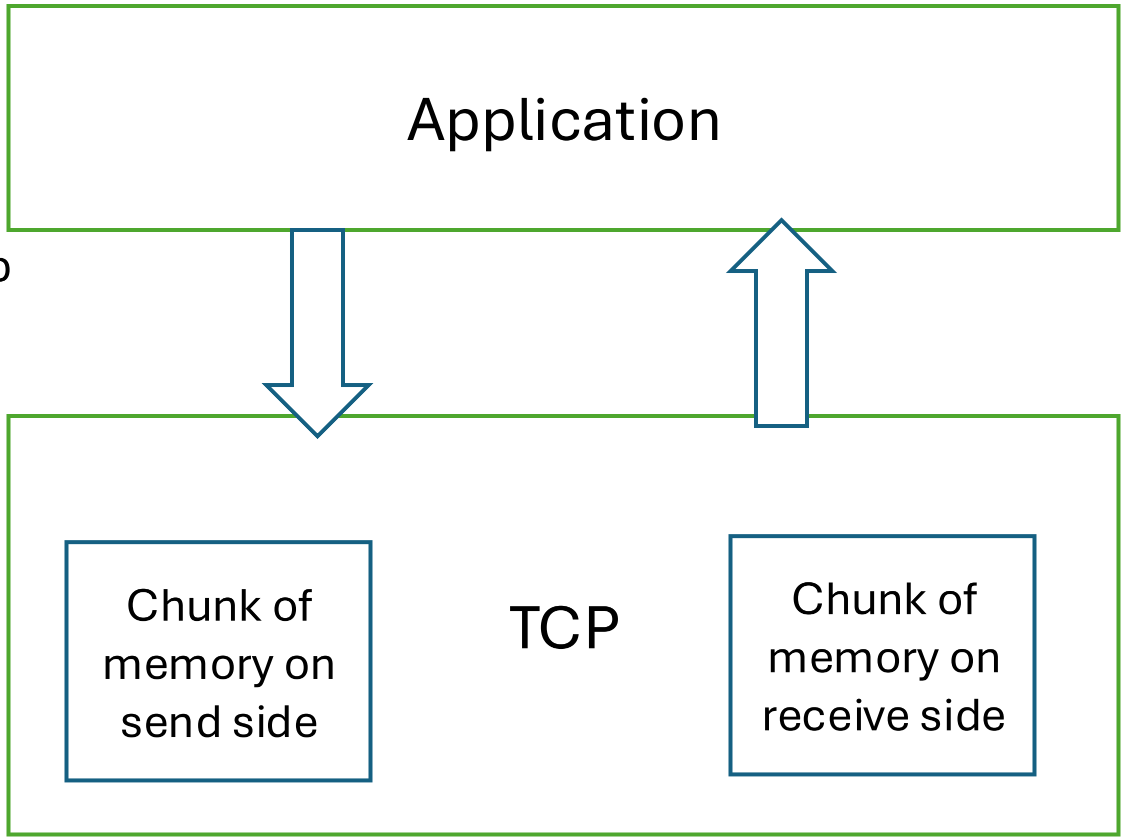
Flow control

Disclaimer:

This lecture presents one possible way to design things. Alternate designs are possible. In this process, sometimes I will say incorrect things about TCP. We will fix all inaccuracies by the next class.

TCP's interface to the application in detail

Send buffer holds bytes that were sent by the app to TCP, but have not yet been acknowledged by receiver



Receive buffer holds bytes that have been received *in the order they were sent by the other side* but have not yet been consumed by the application

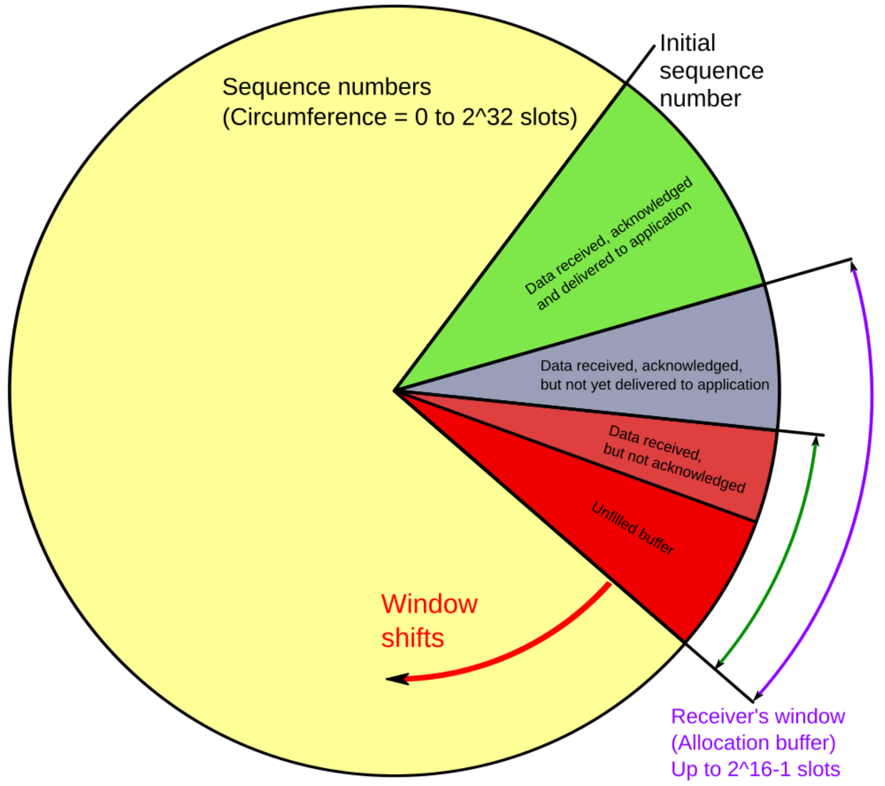
TCP reserves a **chunk of memory on the receive side** that holds bytes that have been received but cannot yet be delivered to the receive buffer because there is a gap
E.g. if sequence numbers [0-1000] and [2000-3000] have been received, then 0-1000 can be delivered, but we are still waiting to receive 2000-3000

TCP reserves a **chunk of memory on the send side** that contains bytes that have been sent but not acknowledged. Hence, they may have to be transmitted again

Receive window

For now, assume initial sequence number is 0 and that there is no upper limit on how large sequence numbers can be

- The previous slide showed that TCP allocates a chunk of memory for storing bytes that have been received but not yet consume by the application.
- The application is free to consume bytes at any rate that it likes. Thus, if it stops consuming bytes, the sender needs to stop sending them otherwise the receiver will run out of memory
- To ensure this, the receiver sends a "receive window" with each acknowledgment. This puts a cap on the maximum number of bytes that the sender can have "in-flight"
- Definition: "in-flight bytes" is the number of bytes that the sender has sent, but have not yet been acknowledged or determined to be lost
- If the receiver has more spare memory than the bytes in flight, it can never overflow


Sequence numbers
(Circumference = 0 to 2^32 slots)
Initial sequence number
Data received, acknowledged and delivered to application
Data received, acknowledged, but not yet delivered to application
Data received, but not acknowledged
Unfilled buffer
Window shifts
Receiver's window (Allocation buffer) Up to 2^16-1 slots

Algorithm for calculating receive window

- When a connection is initiated, allocate some total memory T
    - The OS decides T based on configuration parameters and based on how much spare memory is available. Servers that handle a lot of flows can get bottlenecked on available memory for the receive buffer
- When X new bytes a received, reduce the receive window by X
- When the application consumes Y bytes, increase the receive window by Y
- The TCP header has a receive window field. Every packet sent by either side is filled with the value calculated above https://en.wikipedia.org/wiki/Transmission_Control_Protocol#TCP_segment_structure
- If the OS decides to change T, update receive window accordingly.
    - Side note: depending on how this is done, memory may overflow, and packets may have to be dropped. For instance, if the receiver suddenly reduces T to 0, then it will take some time for the sender to be notified of this update. Until then, it will continue to send. This is ok since TCP can handle losses via retransmission.

Reliability

TL;DR If a packet is lost, retransmit it

Recall how acknowledgments work

- The receiver acknowledges *exactly* which packets it received by using a list of ranges
- In the common case, this list will have only one range: (0, last byte received)
- If packets were lost or reordered, there may be gaps
  - E.g. [(0, 10000), (11000, 13000), (14000, 20000)]
- This tells the sender that bytes (10000, 11000) and (13000, 14000) were not used
- This gives it clues on which packets may be lost and require retransmission

How the sender decides whether a packet has been lost: **Timeout**

There are two loss detection mechanisms. The first is a timeout:

- When it sends a packet, it starts a timer that will ΔT seconds later. If the bytes have not been acknowledged by then, it will assume that they have been lost
- ΔT is set conservatively so that it is unlikely to fire if a packet has not been lost. Standards have proposed using
  Smoothed RTT + max(G, 4 * RTT Variation)
- Here, G is a constant (e.g. 100 ms) and RTT is the "round trip time", the time between when *other* bytes were set and when they were acknowledged
  - Smoothed RTT is the average of the RTTs measured for the last few packets
  - RTT variation is the variation experienced over the last few packts

How the sender decides whether a packet has been lost: **DupACKs**

There are two loss detection mechanisms. The second is “duplicate acknowledgments (DupACK)”

- Suppose the sender receives an ACK for sequence numbers [(0,1000), (2000, 3000)]
- Can it conclude that bytes (1000, 2000) were lost?
- No, the network may reorder packets. So, the packet containing bytes (1000, 2000) may be received after the one containing (2000, 3000). In this case, the next ACK will be [(0, 3000)]
- The convention is to wait for 3 acknowledgments where a given range of bytes has not been ACKed before declaring loss. For example, it may declare loss after it receives the following three ACKs:
  - [(0, 1000), (2000, 3000)]
  - [(0, 1000), (2000, 4000)]
  - [(0, 1000), (2000, 5000)]

How the sender decides whether a packet has been lost: **DupACKs**

We note the following about DupACKs

- The reason it is called a “duplicate” acknowledgment is historical
- In the previous example, each packet contains 1000 bytes. This need not always be the case.
- The receiver sends an acknowledgment for every packet it receives*. Thus, as long as networks do not reorder packets by more than 3, we will not incorrectly declare loss
- Why 3? It was measured to be a good number a long time ago. Now, it is a self-fulfilling prophecy since all links will try to limit reordering to 3. Otherwise, TCP will go beserk and assume everything is lost

*well… this is not strictly true, but we will not bother with that here

Sequence numbers are finite

- TCP uses a 32-bit integer for the sequence number. This means that if we send more than ~4 GB (= 2^32 bytes), the sequence number will return to 0.
- Thus, TCP needs to handle this carefully. However this is boring, so we will ignore this detail
- Also, for safety reasons that we will not discuss, the initial sequence number is picked randomly. It is not 0

TCP limits the list of ranges in an ACK to 3

- This means that if the set of acknowledged bytes has more than 3 contiguous segments, the receiver can only send 3
- For example, suppose it got the ranges:
  - (0, 4000), (5000, 6000), (8000, 10000), (11000, 12000)
- Which 3 would you pick? Why?
- This is what the IETF RFC 2018 has to say on the matter (I have included this just to give you a sense of how things are standardized. There is no need to sweat the details):
  - The first SACK block (i.e., the one immediately following the kind and length fields in the option) MUST specify the contiguous block of data containing the segment which triggered this ACK, unless that segment advanced the Acknowledgment Number field in the header. This assures that the ACK with the SACK option reflects the most recent change in the data receiver's buffer queue.
  - The data receiver SHOULD include as many distinct SACK blocks as possible in the SACK option. Note that the maximum available option space may not be sufficient to report all blocks present in the receiver's queue.
  - The SACK option SHOULD be filled out by repeating the most recently reported SACK blocks (based on first SACK blocks in previous SACK options) that are not subsets of a SACK block already included in the SACK option being constructed. This assures that in normal operation, any segment remaining part of a non-contiguous block of data held by the data receiver is reported in at least three successive SACK options, even for large-window TCP implementations [RFC1323]). After the first SACK block, the following SACK blocks in the SACK option may be listed in arbitrary order.

When should you use TCP vs UDP?