**What Is the Transport Layer?**

**Extends network to the applications**: the transport layer takes messages from the network to applications. In other words, while the network layer (directly below the transport layer) transports messages from one end-system to another, the transport layer delivers the message to and from the relevant application on an end-system.

* The transport layer does not have anything to do with the **core of the network**. Its only responsibility is to take messages from an application on a machine and hand them off to the network layer. The network layer transfers messages from one host to another.
* The transport layer also receives messages from the network layer and transports them to the correct application.

## **Transport Layer Protocols** [**#**](https://www.educative.io/courses/grokking-computer-networking/gxQl90g1279#transport-layer-protocols)

The transport layer has two prominent protocols: the **transmission control protocol** and the **user datagram protocol**. In general, an application developer will have to choose between the two.

**TCP**

1. Delivers messages that we call ‘segments’ reliably and in order.
2. Detects any modifications that may have been introduced in the packets during delivery and corrects them.
3. Handles the volumes of traffic at one time within the network core by sending only an appropriate amount of data at one time.
4. Examples of applications/application protocols that use TCP are: HTTP, E-mail, File Transfers.

**UDP**

1. Does not ensure in-order delivery of messages that we call ‘datagrams.’
2. Detects any modifications that may have been introduced in the packets during delivery but does not correct them by default.
3. Does not ensure reliable delivery.
4. Generally faster than TCP because of the reduced overhead of ensuring uncorrupted delivery of packets in order.

5 Applications that use UDP include: Domain Name System (DNS), live video streaming, and Voice over IP (VoIP).

# Multiplexing and Demultiplexing

What are Multiplexing & Demultiplexing? [#](https://www.educative.io/courses/grokking-computer-networking/RMkqGDqopQO#what-are-multiplexing-demultiplexing)

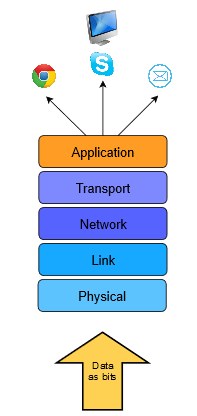
End-systems typically run a variety of applications at the same time. For example, at any given time a browser, a music streaming service, and an email agent could be running.

End systems run many programs at once which leaves us with the question: what process to deliver which packet to?

What is Demultiplexing? [#](https://www.educative.io/courses/grokking-computer-networking/RMkqGDqopQO#what-is-demultiplexing)

Demultiplexing is the process of delivering the correct packets to the correct applications from one stream.

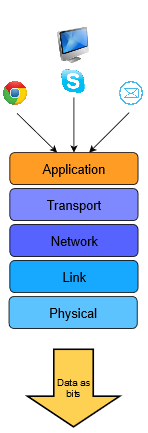
Here’s a useful analogy: deciphering the mail that should be delivered to which houses after a large shipment of packages are received at a post office.



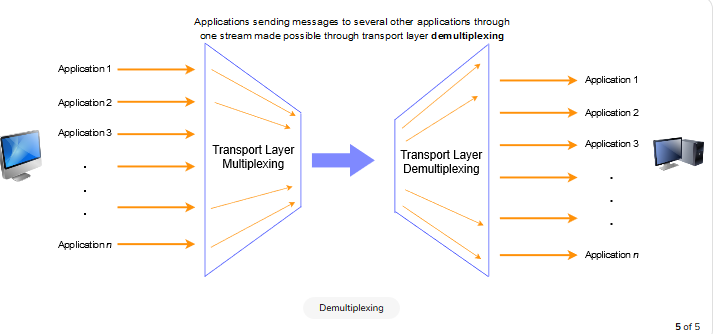
### What is Multiplexing? [#](https://www.educative.io/courses/grokking-computer-networking/RMkqGDqopQO#what-is-multiplexing)

Also, multiplexing allows messages to be sent to more than one destination host via a single medium.

An analogy would be when several packages to several different locations are mailed out from one house.



Multiplexing and demultiplexing are usually a concern when one protocol (TCP for example) is used by many others (HTTP, SMTP, FTP) in an upper layer.



## How Do They Work in the Transport Layer? [#](https://www.educative.io/courses/grokking-computer-networking/RMkqGDqopQO#how-do-they-work-in-the-transport-layer)

Recall that **sockets** are gateways between applications and the network, i.e., if an application wants to send something over to the network, it will write the message to its socket.

Sockets have an associated **port number** with them. We looked at ports briefly in a previous lesson, but here’s a quick overview

* Port numbers are 16-bit long and range from 0 and 65,535.
* The first 1023 ports are reserved for certain applications and are called **well-known** ports. For example, port 80 is reserved for HTTP, port 22 is reserved for SSH

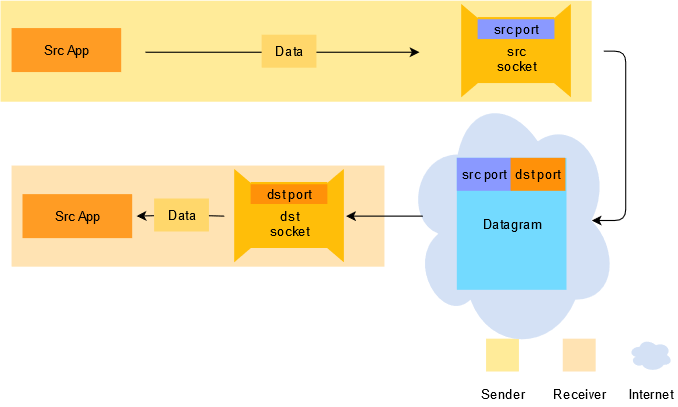
The transport layer **labels** packets with the port number of the application a message is from and the one it is addressed to. This is what allows the layer to multiplex and demultiplex data

# Multiplexing & Demultiplexing in UDP

Connectionless refers to multiplexing and demultiplexing with UDP. Let's dive right in.

**Sockets**, which are gateways to applications, are identified by a combination of an **IP address** and a 16-bit **port number**. That means 2^{16}= 65536 port numbers exist. However, they start from port 0 so they exist in the range of 0−65535.

## Multiplexing & Demultiplexing in UDP [#](https://www.educative.io/courses/grokking-computer-networking/JE4wpnQYl2K#multiplexing-demultiplexing-in-udp)

* When a datagram is sent out from an application, the port number of the associated **source** and **destination** application is appended to it in the UDP protocol header.
* When the datagram is received at the receiving host, it sends the datagram off to the relevant application’s socket based on **destination port number**.
* If the source port and source IP address of two datagrams are different but the destination port and IP address are the same, the datagrams will still get sent to the same application.
* ****

## On Port Assignment in UDP [#](https://www.educative.io/courses/grokking-computer-networking/JE4wpnQYl2K#on-port-assignment-in-udp)

It’s far more common to let the port on the client-side of an application be assigned dynamically instead of choosing a particular port. This is because for communication, both parties must be able to identify each other. Since the client initiates communication to the server, it must know the port number of the application on the server. However, the server doesn’t need to know the client application’s port number in advance. When the first datagram from the client reaches the server, it will carry the client port number, which the server can use to send datagrams back to the client.

However, server-side applications generally do not use dynamically allocated ports! This is because they are running well-known protocols like HTTP and need to be bound to specific ports.

# Introduction to Congestion Control

Congestion physically occurs at the network layer (i.e. in routers), however it’s mainly caused by the transport layer sending too much data at once. That means it will have to be dealt with or ‘controlled’ at the transport layer as well.

📝 **Note** Congestion control also occurs in the network layer, but we’re skipping over that detail for now since the focus of this chapter is the transport layer. So congestion control with TCP is **end-to-end**; it exists on the end-systems and not the network. Also note that in this lesson, the term **delay** means **end-to-end message delay**.

1. Transport layer sends packets at a slower rate in response to congestion,
2. The ‘slower rate’ is still fast enough to make efficient use of the available capacity,
3. Changes in the traffic are also kept track of.

Congestion control algorithms are based on these general ideas and are built into transport layer protocols like TCP.

## Bandwidth Allocation Principles [#](https://www.educative.io/courses/grokking-computer-networking/B1grxg4nooo#bandwidth-allocation-principles)

Well, if it were allocated on a per-host basis, an Internet-enabled doorbell and a busy server would have the same capacity. However, the per-connection allocation can be exploited by hosts opening multiple connections to the same end-system.

Usually, bandwidth is allocated per connection.

### Efficiency & Power [#](https://www.educative.io/courses/grokking-computer-networking/B1grxg4nooo#efficiency-power)

Suppose 4 end-systems are to use a link with a bandwidth of 200 Mbps. It may seem that in order to make the most efficient use of this link, the bandwidth should be divided equally i.e., 200/4 = 50 Mbps should be allocated to each host. However, in a real setting, each end-system would be able to use less than the anticipated 50 while avoiding congestion

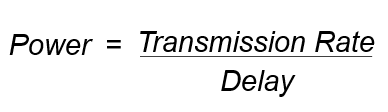
Because real traffic is transmitted in **bursts** and not in one continuous stream. Simultaneous bursts of traffic from all end-systems can cause more than the allocated bandwidth to be used which results in congestion and a consequent drop in performance. Therefore, **bandwidth cannot be divided and allocated equally amongst end-systems!**

#### Transmission Threshold [#](https://www.educative.io/courses/grokking-computer-networking/B1grxg4nooo#transmission-threshold)

The effect of an increase in transmission rate on ‘useful traffic’ (traffic that is actually received by the receiver). The number of packets received by the receiver drastically drop past a certain threshold of the transmission rate despite the fact that the threshold is less than the capacity.

The end-to-end delay in the delivery of the packets increases exponentially when the packet transmission rate increases beyond a certain threshold. Furthermore, the delay can never be infinite, so the packets are simply dropped instead after a certain point.

To sum up, congestion occurs before the maximum capacity of the network is reached and **congestion collapse** occurs as it’s approached.

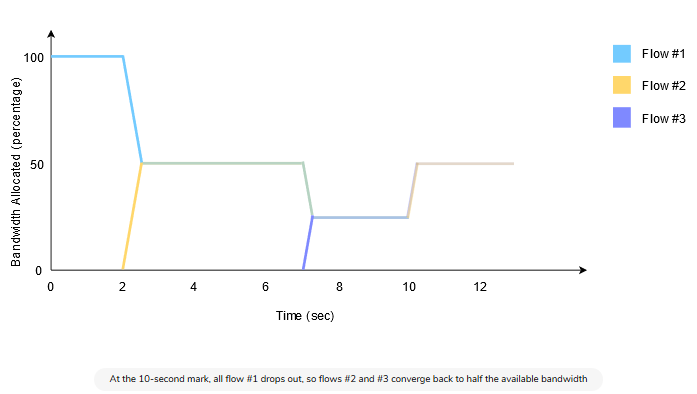


Note that after a certain threshold, increase in transmission rate will cause a very high increase in delay decreasing the overall power.

According to Kelinrock et al., the optimal transmission rate is one for which “power” is MAXIMUM

# More on Principles of Congestion Control

The congestion control scheme should, therefore, ensure that bandwidths are **efficiently used**. Mathematically, the control scheme should ensure that the sum of the transmission rate allocated to all hosts at any given time should be approximately equal to the bottleneck link’s bandwidth.



📝 **Note** These values (for example, ‘half’ the bandwidth) are just for demonstration purposes. In real life, as discussed previously, all of the available bandwidth can never be used. Furthermore, bandwidth allocation works based on a protocol in the network layer which we will discuss in the next chapter. Let’s just assume that this is how it works for now

1. Slow Start Phase: starts slowly increment is exponential to threshold
2. Congestion Avoidance Phase: After reaching the threshold increment is by 1
3. Congestion Detection Phase: Sender goes back to Slow start phase or Congestion avoidance phase.

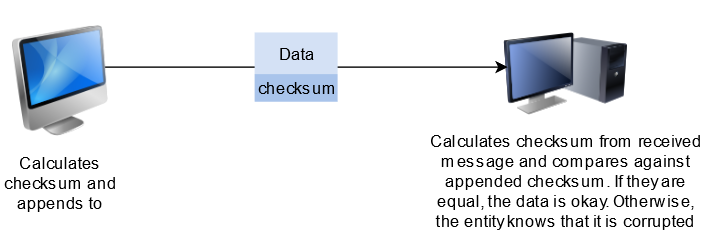
# Principles of Reliable Data Transfer

The transport layer must deal with the imperfections of the network layer service. There are three types of imperfections that must be considered by the transport layer:

1. Segments can be **corrupted** by transmission errors
2. Segments can be **lost**
3. Segments can be **reordered** or **duplicated**

## Checksums [#](https://www.educative.io/courses/grokking-computer-networking/397GLvojwMA#checksums)

The first imperfection of the network layer is that segments **may be corrupted by transmission errors**. The simplest error detection scheme is the **checksum**.



## Retransmission Timers [#](https://www.educative.io/courses/grokking-computer-networking/397GLvojwMA#retransmission-timers)

The second imperfection of the network layer is that **segments may be lost**. Since the receiver sends an acknowledgment segment after having received each data segment, the simplest solution to deal with losses is to use a **retransmission timer**.

A retransmission timer starts when the sender sends a segment. The value of this retransmission timer should be greater than the **round-trip-time**, for example, the delay between the transmission of a data segment and the reception of the corresponding acknowledgment. Note that TCP sends an acknowledgment for almost every segment! When the retransmission timer expires, the sender assumes that the data segment has been lost and retransmits it.

### Limitations of Retransmission Timers [#](https://www.educative.io/courses/grokking-computer-networking/397GLvojwMA#limitations-of-retransmission-timers)

Unfortunately, retransmission timers alone are not sufficient to recover from segment losses. Let us consider the situation depicted below where **an acknowledgment is lost.** In this case, the sender retransmits a data segment that has been received correctly, but not properly acknowledged.

## Sequence Numbers [#](https://www.educative.io/courses/grokking-computer-networking/397GLvojwMA#sequence-numbers)

To identify duplicates, transport protocols associate an identification number with each segment called the **sequence number**. This sequence number is prepended to the segments and sent. This way, the end entity can identify duplicates.

# Reliable Data Transfer: Sliding Window

## Pipelining [#](https://www.educative.io/courses/grokking-computer-networking/B6Yzq7O6xKn#pipelining)

Applications may generate data at a rate much higher than the network can transport it. Processor speed is generally much higher than the speed of writing out and reading data to/from the network (I/O).

Furthermore, reliable message communication is a multi-step process:

1. The network carries the acknowledgment to the sender. So, instead of waiting for an acknowledgment of every packet before transmitting the next one, it’s **more efficient** to **pipeline** the multi-step process. In other words, instead of waiting for the acknowledgment of a message before transmitting the next one, the sender keeps transmitting messages without waiting for an acknowledgment. This makes more efficient use of the processor’s time.

While pipelining allows the **sender** to transmit segments at a **higher rate**, it may cause the **receiver** to become **overloaded** because the receiver may be running on a slower machine than the sender, or the receiving machine may be busy executing other processes.

📝 **Note:** “**stop and wait**” is the term used in literature for when the sending entity waits for the acknowledgment of every transmitted message before sending the next one. That is also a means to recover from lost messages. It retransmits.

### Sliding Window [#](https://www.educative.io/courses/grokking-computer-networking/B6Yzq7O6xKn#sliding-window)

The sliding window is the set of consecutive sequence numbers that the sender can use when transmitting segments without being forced to wait for an acknowledgment. At the beginning of a session, the sender and receiver agree on a sliding window size.

<https://www.youtube.com/watch?v=1DvXxrzq0Wg&list=PLEbnTDJUr_IegfoqO4iPnPYQui46QqT0j&index=12>

Unfortunately, segment losses do not disappear because a transport protocol is using a sliding window. To recover from segment losses, a sliding window protocol must define:

* A heuristic to detect segment losses.
* A retransmission strategy to retransmit the lost segments.

# Reliable Data Transfer: Go-back-n

In the last lesson, we discovered that a sending sliding window alone is not enough to ensure **detection and retransmission of lost packets**. In order to do that, we will look at two protocols:

1. **Go-back-n**
2. **Selective Repeat**

<https://www.youtube.com/watch?v=ZLtkhsgQp8U&list=PLEbnTDJUr_IegfoqO4iPnPYQui46QqT0j&index=13>

Go Back N :

1. Sender window size in GBN = N. i.e GB10 implies sender window size in 10.
2. Always N>1 i.e sender window size in greater that N. If N=1 then it is stop and weight
3. Receiver window size = 1 always.
4. If one packet is lost the sender sends all the N packets to receiver without wasting time for acknowledgements of other remaining packets.
5. ACK are cumulative with ack timer at receiver side.
6. Relationship btw window size and sequence numbers. In any sliding window protocol if sender window size (sws) and receiver window size (rws) and available sequence number (asn) is { sws + rws <= asn }
7. In GBN sender window size is N and receiver window size is 1. So minimum no:of sequence numbers required is N+1

**Selective Repeat**

<https://www.youtube.com/watch?v=Oipm5DdYYAs&list=PLEbnTDJUr_IegfoqO4iPnPYQui46QqT0j&index=14>

1. Sender window size > 1.
2. Sender window size = Receiver Window Size
3. In this if one packet is lost we only send that lost packet. But not the N packet.
4. ACK are independent not cumulative.
5. Negative ACK is sent in Selective Repeat protocol when the packet received but corrupted. In case of GBN the receiver simply discards the packet and doesn’t send any ack.

# The User Datagram Protocol

## How It Works [#](https://www.educative.io/courses/grokking-computer-networking/YVK8ZJPLN3A#how-it-works)

UDP does not involve any initial handshaking like TCP does, and is hence called a **connectionless** protocol. This means that there are no established ‘connections’ between hosts.

UDP prepends the **source and destination ports** to messages from the application layer and hands them off to the network layer. The Internet Protocol of the network layer is a **best-effort** attempt to deliver the message. This means that the message-

1. May or **may not get delivered**.
2. May get **delivered with changes in it**.
3. May get **delivered out of order**.

UDP only adds the **absolute bare minimum** functionality over the network layer. So it…

* Does not ensure that messages get sent.
* It does check, however, if a message got ‘corrupted’ yet does not take any measures to correct the errors by default.

### Header [#](https://www.educative.io/courses/grokking-computer-networking/YVK8ZJPLN3A#header)

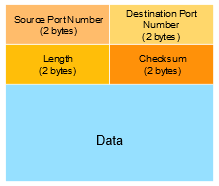
UDP prepends **four** 2-byte header fields to the data it receives from the application layer. So in total, a UDP header is **8 bytes** long. The fields are:

1. **Source** port number
2. **Destination** port number
3. **Length** of the datagram (header and data in bytes)
4. **Checksum** to detect if errors have been introduced into the message. We’ll study this in detail in the [next lesson](https://www.educative.io/collection/page/10370001/6105520698032128/6036281094045696)!

### Data [#](https://www.educative.io/courses/grokking-computer-networking/YVK8ZJPLN3A#data)

Other than the headers, a UDP datagram contains a body of data which can be up to **65,527** bytes long. Since the maximum possible length of a UDP datagram is 65,535 bytes which includes the 888-byte header, we are left with 65,527 bytes available. The nature of the data depends on the overlying application. So if the application is querying a DNS server, it would contain bytes of a zone file.

Here’s what a UDP message looks like:



# UDP Checksum Calculation & Why UDP?

## Why UDP?

You might be wondering why would anyone use UDP when it has so many apparent drawbacks and doesn’t really do anything? Well, there are actually a number of reasons why UDP would be a good choice for certain applications.

1. UDP can be **faster**. Some applications cannot tolerate the load of the retransmission mechanism of TCP, the other [transport layer protocol](https://www.educative.io/collection/page/10370001/6105520698032128/4608034330378240).
2. **Reliability can be built on top** of UDP. TCP ensures that every message is sent by resending it if necessary. However, this reliability can be built in the application itself.
3. UDP gives **finer control** over what message is sent and when it is sent. This can allow the application developer to decide what messages are important and which do not need concrete reliability.
4. With the significantly smaller header gives UDP an edge over TCP in terms of reduced transmission overhead and quicker transmission times.

## Well-Known Applications That Use UDP [#](https://www.educative.io/courses/grokking-computer-networking/qVvBjxOVE33#well-known-applications-that-use-udp)

1. Xbox live is built on UDP.
2. [DNS](https://www.educative.io/collection/page/10370001/6105520698032128/5448697208897536) uses UDP!

Resends the message.

Sends the message to some other server.

Gives a failure message.

Using UDP instead of TCP makes DNS and consequently, web browsing significantly faster.

# Exercise: Capturing UDP Packets

## What is tcpdump? [#](https://www.educative.io/courses/grokking-computer-networking/RMwm0qB0mxE#what-is-tcpdump)

[tcpdump](https://www.tcpdump.org/) is a command-line tool that can be used to view packets being sent and received on a computer. The simplest way to run it is to simply type the following command into a terminal and hit enter. You can try this on the terminal provided at the end of this lesson!

# The Transmission Control Protocol

## Well-Known Applications That Use TCP [#](https://www.educative.io/courses/grokking-computer-networking/7Xw8KoRLYqQ#well-known-applications-that-use-tcp)

**File Transfer** [**#**](https://www.educative.io/courses/grokking-computer-networking/7Xw8KoRLYqQ#file-transfer)

FTP or **File Transfer Protocol** is built on top of TCP. It uses ports **20** and **21**. When transferring files, we wouldn’t want some bytes of the file completely missing, or some chunks in the file re-ordered or some byte values changed during transfer. That’s why TCP is a natural choice for FTP. In other words, it uses TCP for its **reliability**, which is a key part of file transfer.

**Secure Shell SSH** [**#**](https://www.educative.io/courses/grokking-computer-networking/7Xw8KoRLYqQ#secure-shell-ssh)

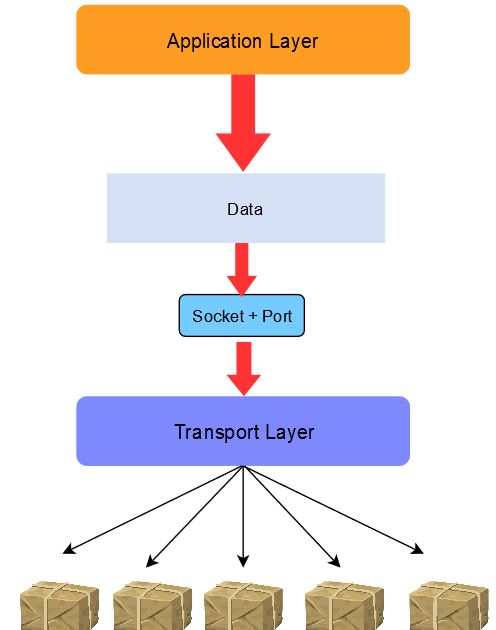
**SSH** or Secure Shell is a protocol to allow a secure connection to a remote host over an unsecured network. It’s widely popular and most programmers use it to date to execute operating system shell commands on remote servers. The reasons that this application uses TCP is similar to FTP, and that’s reliable delivery.

**Email** [**#**](https://www.educative.io/courses/grokking-computer-networking/7Xw8KoRLYqQ#email)

All email protocols, SMTP, IMAP, and POP use TCP to ensure complete and reliable message delivery similar to the reasons that FTP uses TCP.

**Web Browsing** [**#**](https://www.educative.io/courses/grokking-computer-networking/7Xw8KoRLYqQ#web-browsing)

Web browsing on both HTTP and HTTPS is done on TCP as well for the same reasons as FTP.



## What TCP Does [#](https://www.educative.io/courses/grokking-computer-networking/7Xw8KoRLYqQ#what-tcp-does)

Here are some key responsibilities of the protocol.

1. **Send data** at an appropriate transmission rate to make full use of the available capacity but it shouldn’t be so fast as to cause congestion.
2. **Segment data.** The application layer sends the transport layer a continuous and unsegmented stream of data so that there’s no limit to how much data the application layer can give to the transport layer at once. Hence, the transport layer divides it into appropriately sized **segments**. Note that a segment is a collection of bytes. Furthermore, when a TCP segment is too big, the network layer may break it into multiple network layer messages, so the receiving TCP entity would have to re-assemble the network layer messages.
3. **End to end flow control**. Flow control means **not overwhelming the receiver**. It’s not the same as congestion control. Congestion control tries not to choke the network. However, if the receiving machine is slow, it might drown in data even if the network is not choked. Avoiding drowning the receiver in data is end to end flow control. There is also hop by hop flow control, which is done at the data link layer.
4. **Identify and retransmit messages** that do not get delivered. The network layer cannot be relied upon to deliver messages. It is the responsibility of TCP (application layer) to do this job.
5. **Identify when messages are received out of order and reassemble** them. The network layer can also not be relied upon to transmit messages in order. It is the responsibility of TCP (application layer) to do this job too.

On demand movie streaming is a good example of an application that can choose TCP Since this is pre-recorded content, retransmission can be done along with buffering at the receiver. Sure, occasional buffering will be annoying, but lost frames would be ugly.

Live audio conferencing is a prime example of an application that uses UDP.

# Key Features of the Transmission Control Protocol

## Connection Oriented [#](https://www.educative.io/courses/grokking-computer-networking/3927z5jyVq4#connection-oriented)

TCP itself is **connection-oriented** and creates a long term connection between hosts. The connection remains until a certain termination procedure is followed.

## Full Duplex [#](https://www.educative.io/courses/grokking-computer-networking/3927z5jyVq4#full-duplex)

Furthermore, TCP is **full-duplex**, which means that both hosts on a TCP connection can send messages to each other simultaneously.

## Point-to-point Transmission [#](https://www.educative.io/courses/grokking-computer-networking/3927z5jyVq4#point-to-point-transmission)

TCP connections have **exactly two endpoints**! This means that **broadcasting** or **multicasting** is not possible with TCP.

## Error Control

TCP can detect errors in segments and make corrections to them.

## Flow Control

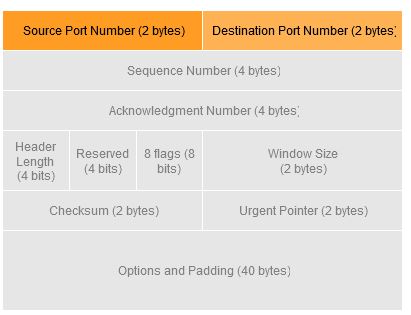
TCP on the sending side controls the amount of data being sent at once based on the receiver’s specified capacity to accept and process it. The sender adjusts the sending rate accordingly.

## Congestion Control

As specified in a previous lesson, TCP has in-built mechanisms to control the amount of congestion on the network.

# TCP Segment Header

The size of the headers range from **20 - 60 bytes**. Let’s discuss the header field by field.



**Sequence Number** [**#**](https://www.educative.io/courses/grokking-computer-networking/RMPXOjLVwnR#sequence-number)

Every byte of the TCP segment’s data is labeled with a number called a **sequence number**. The sequence number field in the header has the sequence number of the first byte of data in the segment.

## Acknowledgement Number [#](https://www.educative.io/courses/grokking-computer-networking/RMPXOjLVwnR#acknowledgement-number)

The **acknowledgment number** is a 4-byte field that represents the sequence number of the next expected segment that the sender will receive.

### Example [#](https://www.educative.io/courses/grokking-computer-networking/RMPXOjLVwnR#example)

So if a segment’s sequence number was 42849 and its data field had 59 bytes of data, the sequence number of the next expected segment or the **acknowledgment number** would be 42849+59+1 = 42909. This helps TCP to identify if a segment was missing or out of order.

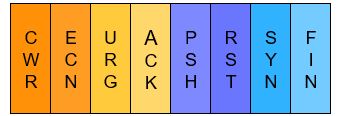
## Header Length [#](https://www.educative.io/courses/grokking-computer-networking/RMPXOjLVwnR#header-length)

The length of the TCP header is specified here. This helps the receiving end to identify where the header ends and the data starts from.

📝 **Note** The header length is represented by **4 bits**, i.e., the numbers 0000→1111 or 0→15 in decimal which is not enough to represent the potential **60 bytes** of the header. Hence, this number is **multiplied by 4** upon receiving. So 1111 would represent 60. In other words, the way the 4-bit header length field is used to represent a maximum header length of 60, is that this field represents the number of 4-byte words in the header.

# TCP Header Flags

TCP headers have eight 1-bit flags that are imperative to signaling in the protocol.



**ACK** [**#**](https://www.educative.io/courses/grokking-computer-networking/gkPznVK1qn9#ack)

This flag is set to 111 in a segment to **acknowledge** a segment that was received previously. when a receiver wants to acknowledge some received data, it sends a TCP segment with the ACK flag and the acknowledgment number field appropriately set. This flag is also used in connection establishment and termination as we will see in more detail later.

**RST** [**#**](https://www.educative.io/courses/grokking-computer-networking/gkPznVK1qn9#rst)

The **reset** flag immediately terminates a connection. This is sent due to the result of some confusion, such as if the host doesn’t recognize the connection, if the host has crashed, or if the host refuses an attempt to open a connection.

**SYN** [**#**](https://www.educative.io/courses/grokking-computer-networking/gkPznVK1qn9#syn)

The **synchronization** flag initiates a connection establishment with a new host. The details will be covered later in the lesson on [connection establishment](https://www.educative.io/collection/page/10370001/6105520698032128/5371969778221056).

**FIN** [**#**](https://www.educative.io/courses/grokking-computer-networking/gkPznVK1qn9#fin)

This flag is used to terminate or **finish** a connection with a host.

**CWR & ECN** [**#**](https://www.educative.io/courses/grokking-computer-networking/gkPznVK1qn9#cwr-ecn)

These flags, **Congestion Window Reduced** and **Explicit Congestion Notification** are used to handle congestion. To put it very simply, the ECN flag is set by the receiver, so that the sender knows that congestion is occurring. The sender sets the CWR flag in response to this so that the receiver knows that the receiver has reduced its congestion window to compensate for congestion and the sender is sending data at a slower rate.

**PSH** [**#**](https://www.educative.io/courses/grokking-computer-networking/gkPznVK1qn9#psh)

The default behavior of TCP is in the interest of efficiency; if multiple small TCP segments were received, the receiving TCP will combine them before handing them over to the application layer. However, when the Push (PSH) flag is set, the receiving end immediately flushes the data from its buffer to the application instead of waiting for the rest of it to arrive.

**URG** [**#**](https://www.educative.io/courses/grokking-computer-networking/gkPznVK1qn9#urg)

The **Urgent** flag marks some data within a message as urgent. Upon receipt of an urgent segment, the receiving host forwards the urgent data to the application with an indication that the data is marked as urgent by the sender.

# TCP Headers: Window Size & More

## Window Size [#](https://www.educative.io/courses/grokking-computer-networking/B13opVG7QRJ#window-size)

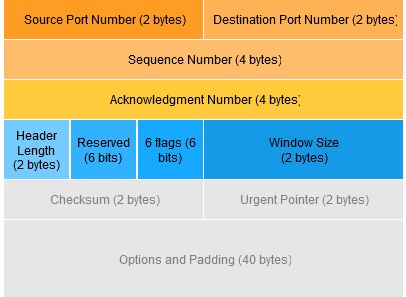
Remember the ‘buffer’ we discussed in the [last lesson](https://www.educative.io/collection/page/10370001/6105520698032128/5250889600204800)? Well, the **window size** is essentially the amount of available space in that buffer. TCP at the receiving end buffers incoming data that has not been processed yet by the overlaying application. The amount of available space in this buffer is specified by the **window size**. The window size can be up to **2 bytes** in size hence, the numerical range of the window size is from 2^0 to 2^16-1 = 65535.

The window size is communicated to the sender by the receiver in every TCP message and gets updated as the buffer fills and empties. If the window size reduces after a bit, the sender will know that it needs to reduce the amount of data being sent, or give the receiver time to clear the buffer.

To put it another way, the window size is at first equal to as much data as the receiving entity is willing and able to receive. As it receives some more data, the window size will decrease and as it hands over some of the received data to the application layer, the window size will increase. This is useful to implement flow control.

The **urgent pointer** defines the byte to the point of which the urgent data exists. This is because a single segment can contain both parts of urgent and regular data. This field is only used in conjunction with the urgent flag.

The **options and padding** field provides up to an extra **40 bytes** to build extra facilities that are not covered by the regular header. The options can vary in length and exist in multiples of 32 bytes using zeros to pad in any extra bits.



# TCP Connection Establishment: Three-way Handshake

When a client host wants to **open a TCP connection** with a server host, it creates and sends a TCP segment with:

Step 1 Initial Connection Request by Client

* The *SYN* flag set
* The sequence number set to a random initial value. So the sequence numbers **do not start with 0**.

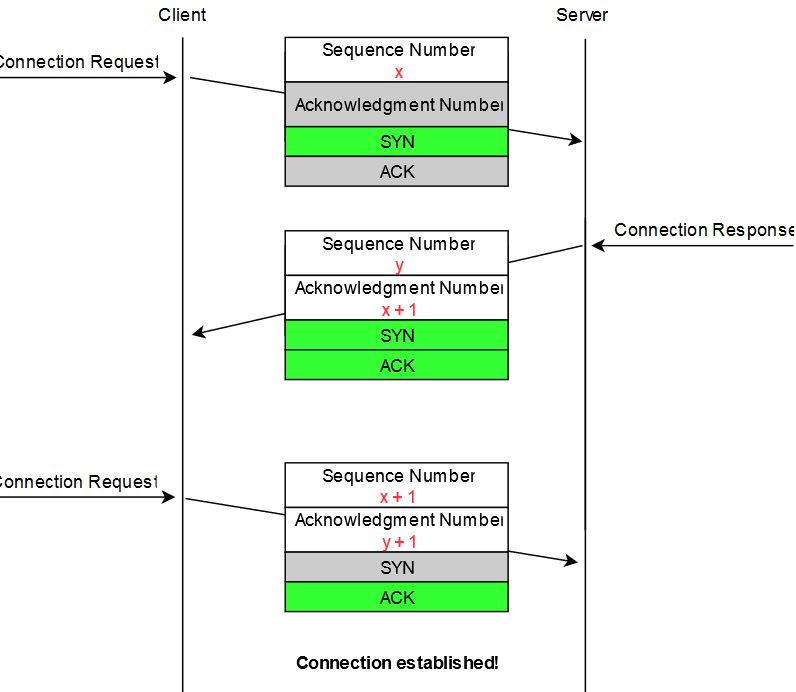
Step 2 Responding to Initial Connection Message with SYN+ACK segment

* the ***SYN* flag** set
* the *sequence number* set to a random number.
* The ***ACK* flag** set
* The **acknowledgment number** set to the sequence number of the received *SYN* segment incremented by 1 mod 2^{32}​​, because the *SYN* segment consumes one byte. This new number may exceed 2^{32}​​, which is the limit of the *ACK* header field, so the modulus by 2^{32}​​ of this number is taken.

This segment is often called a SYN+ACK segment. The acknowledgment confirms to the client that the server has correctly received the SYN segment. The random sequence number of the SYN+ACK segment is used by the server host to verify that the client has received the segment.

Step 3 : Upon reception of the SYN+ACK segment, the client host replies with a segment containing:

* The ***ACK* flag** set
* The **acknowledgment number** set to the sequence number of the received *SYN+ACK* segment incremented by 1. The modulus of the number by 2^{32}​​ is obviously taken. At this point, the TCP connection is open and both the client and the server are allowed to send TCP segments containing data.

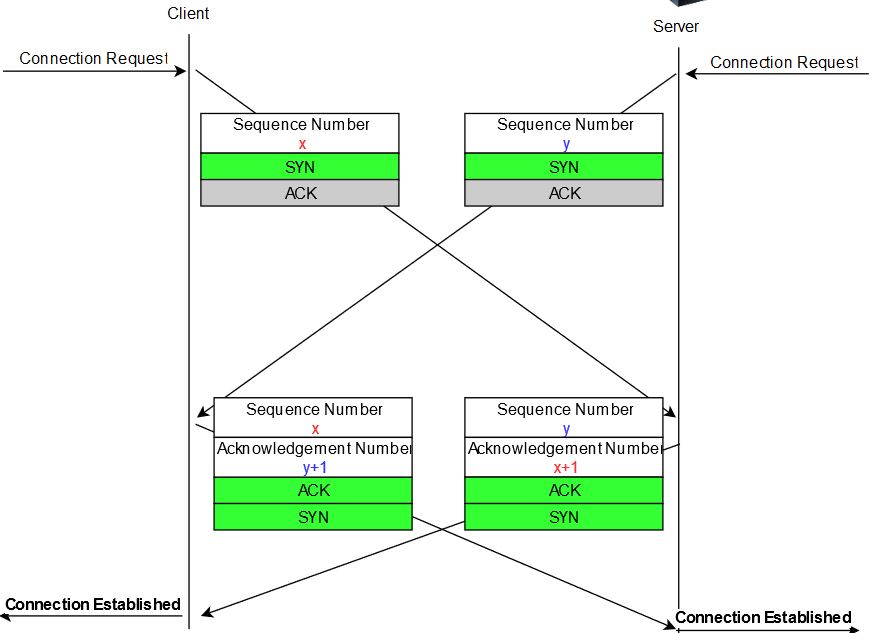


In the figure above, the connection is considered to be established by the client once it has received the SYN+ACK segment, while the server considers the connection to be established upon reception of the ACK segment.

# Other TCP Connection Establishment Methods

**Simultaneous Connection Establishment:**

Both sides must know the port number for each other in this case. It doesn’t have to be a well-known port number or the same on both sides.



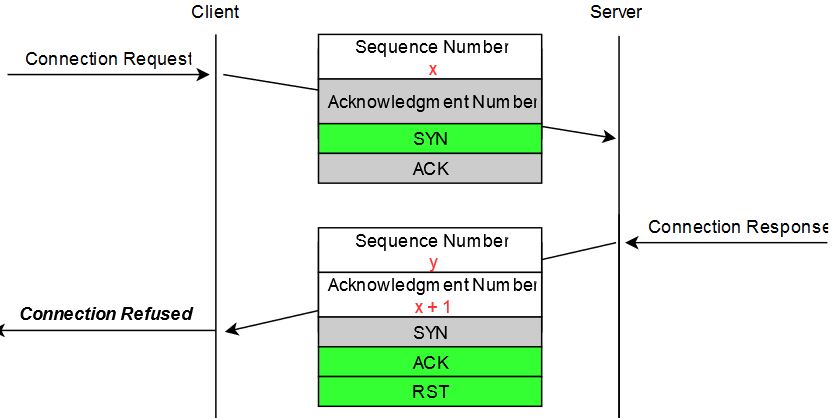
**Hosts Can Refuse Connection Requests** [**#**](https://www.educative.io/courses/grokking-computer-networking/xl7G66JQ8yn#hosts-can-refuse-connection-requests)

A host could refuse to open a TCP connection upon reception of a SYN segment. This refusal may be due to various reasons, for example:

1. There may be **no server process** that’s listening on the destination port of the SYN segment.
2. The server could always refuse connection establishments from a particular client (e.g., due to **security reasons**).
3. The server may **not have enough resources** to accept a new TCP connection at that time.

There are other scenarios in which a connection may be refused but these are the common ones. If a process is listening on a port, but the connection is to be refused, the server sends a SYN segment with the following properties:

1. Has its RST flag set
2. Contains the sequence number of the received SYN segment as its acknowledgment number.



# Efficient data transmission with TCP

There are two simple and extreme implementation choices:

1. Send a TCP segment as soon as the application has requested the transmission of some data.
   * **Advantage**: This allows TCP to provide a **low delay service**.
   * **Disadvantage**: If the application is writing data one byte at a time, TCP would place each byte in a segment containing 20 bytes of the TCP header. This is a **huge overhead** that is not acceptable in wide area networks.
2. Transmit a new TCP segment once the application has produced MSS bytes of data. Recall MSS from this lesson on [TCP Headers](https://www.educative.io/collection/page/10370001/6105520698032128/6673043197788160).
   * **Advantage:** **Reduced overhead**
   * **Disadvantage:** Potentially at the cost of a **very high delay**, which may be unacceptable for interactive applications.

## Nagle’s Algorithm [**#**](https://www.educative.io/courses/grokking-computer-networking/q2Qk8QW43P2#nagles-algorithm)

# 

# Nagle’s algorithm ‘executes’ every time new data comes in from the remote host. Here’s how it works: it sends data if it is at least the size of one MSS and the window size is appropriate. Otherwise, it checks if any unacknowledged segments exist. If so, it ****buffers the data**** and doesn’t send it. There is no timer on this condition, and it will keep buffering data until previous segments are acknowledged. If an ACK segment comes in, it sends the data.

# TCP Window Scaling