

# Course Learning Outcomes

An overview of what you'll learn and the structure of this course.

## WE'LL COVER THE FOLLOWING ^

- Welcome to the Course
- What Will I Be Able to Do by the End of the Course?
- The Components
  - An Introduction to the Basics
  - The Layered Architecture
  - Socket Programming Basics

## Welcome to the Course #

Welcome to Network Fundamentals! In this course, you'll get an in-depth overview of the key concepts of computer networks. Here are a few key components that will help you get the most out of this course!

## What Will I Be Able to Do by the End of the Course? #

- **Develop** socket programs in Python.
- **Trace** networks metadata through the command line.
- Learn about tools that help **debug** network problems.
- **Understand** how applications such as browsers and mobile apps work from the perspective of the network.
- **View and understand** browser interactions with websites.

## The Components #

## An Introduction to the Basics #

The course starts off by introducing some key concepts and networks lingo that will be used throughout the rest of the course.

## The Layered Architecture #

Computer networks are organized into conceptual layers. This makes learning concepts easier and is a standard teaching method in most books and courses.

You will learn about each layer in a separate chapter. Each chapter will start off by laying down the theoretical foundations and then will give some hands-on experience with `bash shell` or `python` playgrounds that you can run commands in.

## Socket Programming Basics #

A chapter of this course teaches python socket programming through basic server and client programs in python. It builds up to the programs line-by-line; each line is explained in detail before the next one is added on.

---

Let's dive right in!

# Learning Instruments

Here's a short summary of some tools used throughout this course to maximize your learning.

## WE'LL COVER THE FOLLOWING ^

- Coding Widgets
- Fun Facts
- Notes
- Security Warnings
- Quotes
- Graphics
- Quizzes
- Links

## Coding Widgets #

The course demonstrates the usage of some key **command-line network tools**. These will help you immensely in your career regardless of what area of computer science you work in. Additionally, you'll learn the essentials of **socket programming in Python**.

## Fun Facts #

 **Did You Know?** This course is scattered with tidbits of information that will make you feel more involved with the field! These aren't absolutely *essential* to learning the fundamentals of computer networks so you can skip them if you want. But they definitely keep things interesting.

## Notes #



**Note** This course is scattered with memos and ‘FYIs’ as well. These notes are important to read and generally cannot be skipped.

## Security Warnings #



**Security Warning:** we’ll broach the topic of some potential network security issues with the topics we’re discussing.

## Quotes #

*“Cool quotes from important people will be included in prompts such as this one.”*

— The Author of This Course

## Graphics #

We have several slides and drawings throughout this course that make learning easier.

## Quizzes #

Last but not least, you’ll be taking a fair amount of quizzes to help solidify your knowledge!

## Links #

We’ve given links to several external resources throughout this course if you are interested in further reading. Most of them can be skipped.

---

Let’s begin!



# What is the Internet?

Let's start with the big question.

## WE'LL COVER THE FOLLOWING

- But First, What's a Network?
- The Internet: A Network of Networks

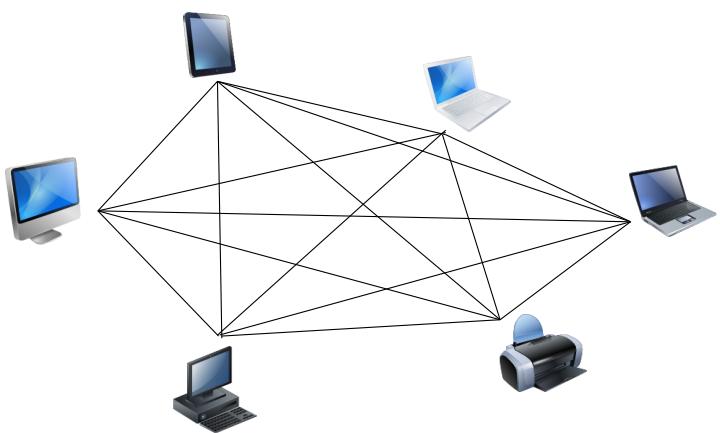
The Internet permeates our very existence. Most of us cannot imagine life without it. We often depend on it for livelihoods, for routine commutes, and for entertainment. It has become almost like a utility. You're accessing this course through the Internet.

But how does it actually work? What goes on behind the scenes? Well, you've come to the right place to learn that! The Internet is a global network of computer networks.

## But First, What's a Network? #

A **network** is officially defined as  
a group or system of  
interconnected people or items.

So, by this definition, train stations connected to each other with rail tracks make up a *railway* network. People who follow each other on Twitter make up an online *social* network.



A basic computer network

Similarly, computers connected to each other with cable or wireless

radio make up a *computer network*.

## Why Computer Networks?

There are **two main purposes** of computer networks:

**Communication** using computers and **sharing of resources**. An “internet” allows doing these two things across different computer networks.



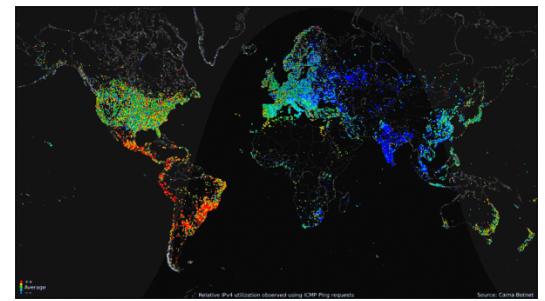
**Note: internet vs. Internet** An internet with a lowercase “*i*” is *any* interconnection of computer networks. Whereas the global Internet is always spelled with a capital I.

## The Internet

The Internet is essentially a network of computer networks.

So your personal computer is connected to other computers at your house or workplace to create a small computer network, which is in turn connected to other computer networks. And so the global Internet encompasses a complex web of interconnected computer networks.

This concept can be better visualized in the illustration below:



A heat map of the usage of the Internet over a 24-hour period

## The Internet: A Network of Networks #

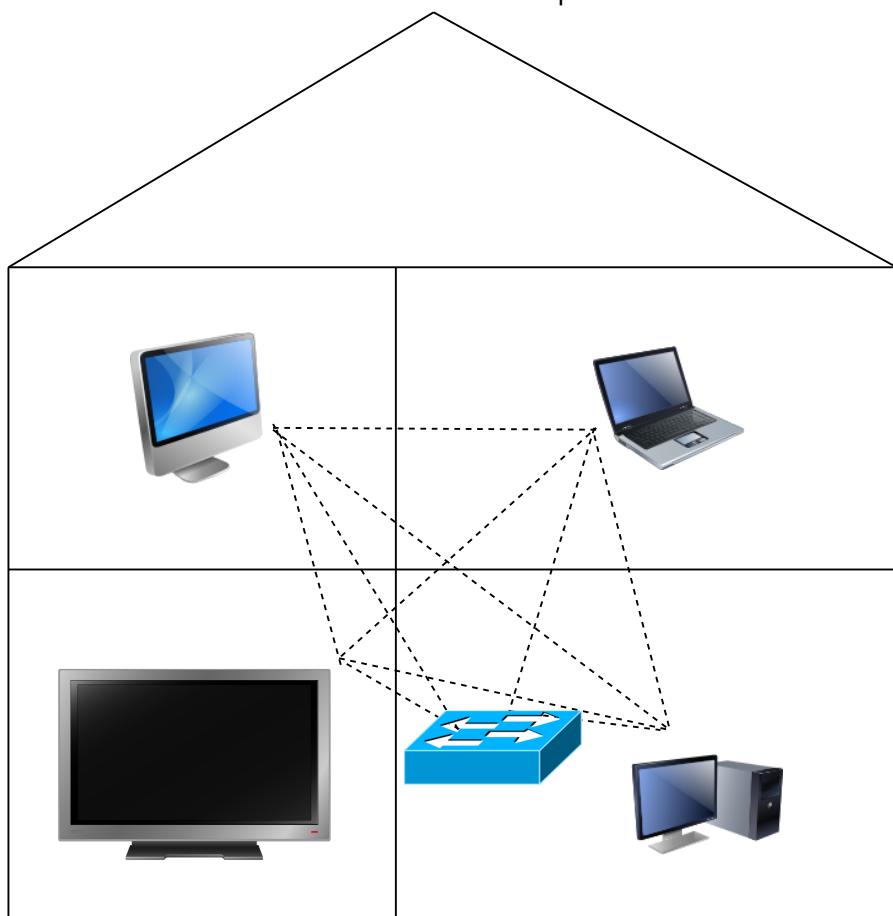
Your Computer



The Internet: A Network of Networks

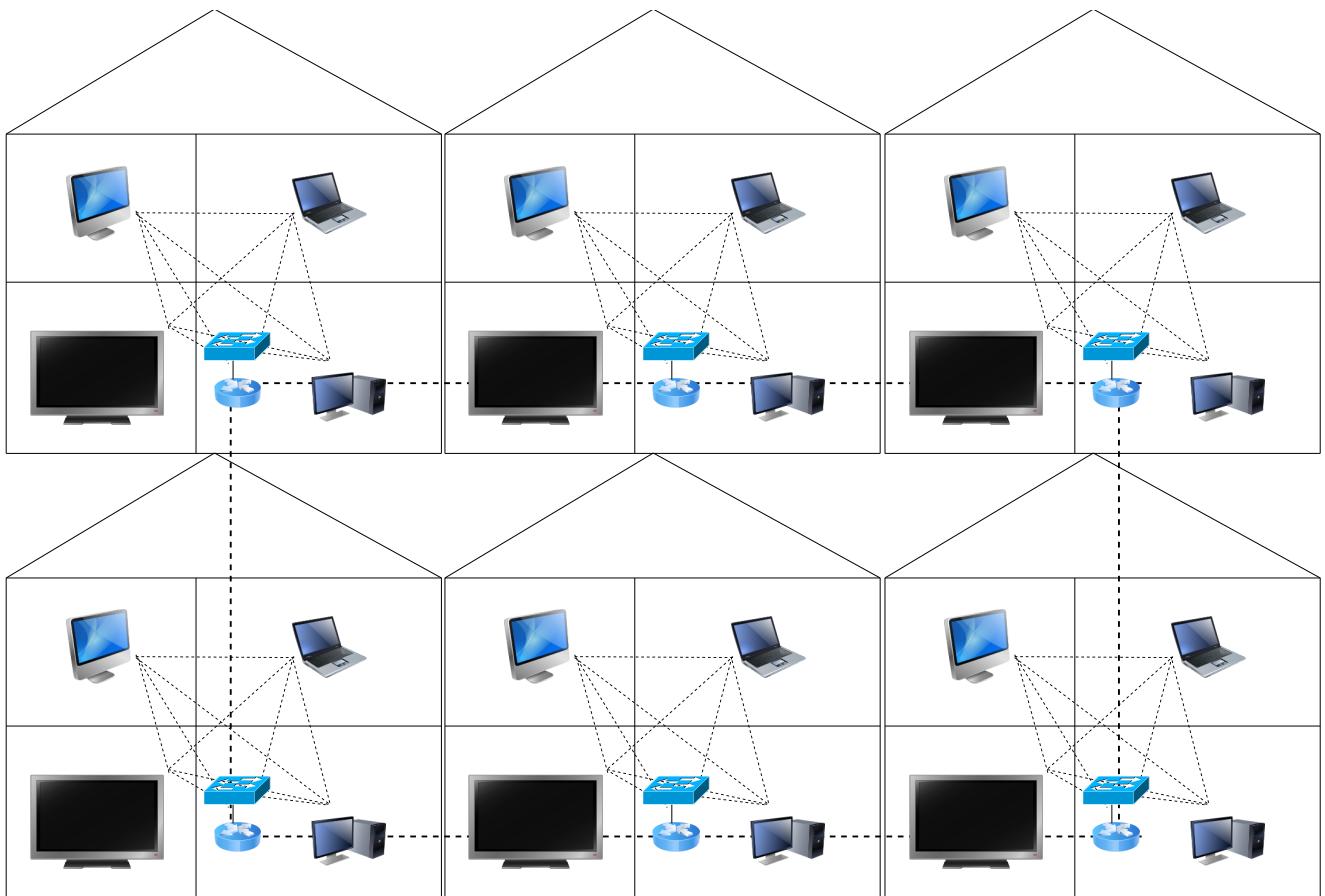
1 of 5

Then the devices at your workplace or home are connected to each other to make up a network



The Internet: A Network of Networks

2 of 5



Then, all the houses in a neighborhood are connected usually via devices called 'routers'.  
Routers forward messages from one end to another.

The Internet: A Network of Networks

3 of 5

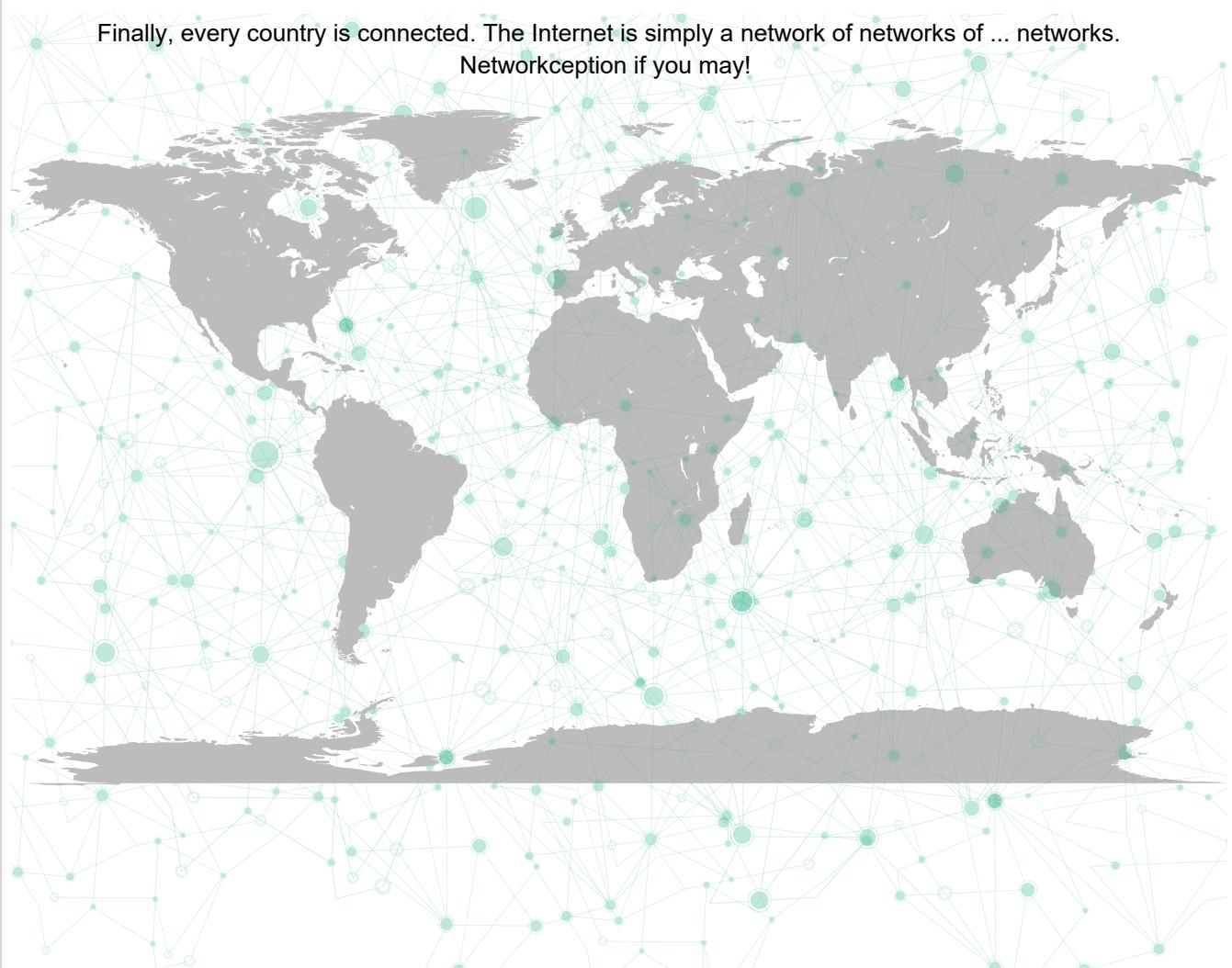


Then, all of the neighborhoods and cities in your country are interconnected to make up a network of networks..

#### The Internet: A Network of Networks

4 of 5

Finally, every country is connected. The Internet is simply a network of networks of ... networks.  
Networkception if you may!



The Internet: A Network of Networks

5 of 5



In the next lesson, we'll take a look at a short history of the Internet and how it all began.

# A Quick History of The Internet #

Here's a short overview of how the Internet came to be.

## WE'LL COVER THE FOLLOWING ^

- Why Learn History?
- The '50s: The Cold War
- The '60s-'70s: ARPANET
- The '80s: Protocols
  - Search Engines
- The 21st Century: The Age of the Internet
- A Video
- Quick Quiz!

## Why Learn History? #

- Learn about the mistakes made during the development of the Internet and avoid repeating them.
- Understand why some things are designed and work the way they do today.
- Lastly, it's interesting!

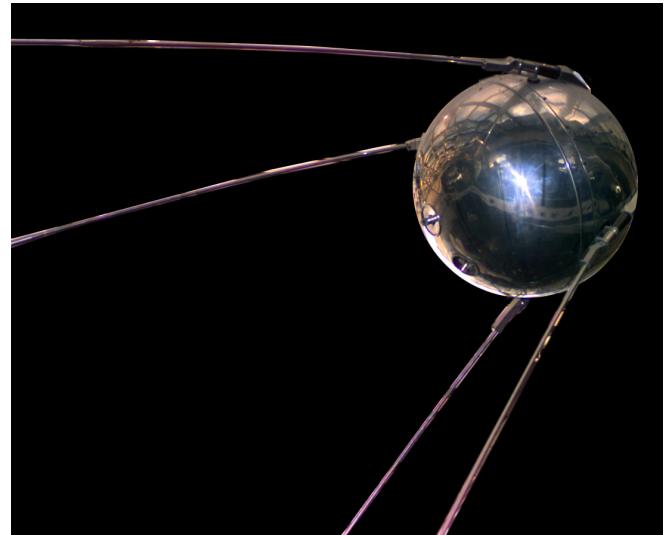
*“Those who do not read history are doomed to repeat it.”*

*– George Santayana*

Let's get right into it.

## The '50s: The Cold War #

- The birth of the Internet can be traced back to the middle of the last century.
- The U.S. was in the midst of a cold war with The Soviet Union and both nations were trying to gain an advantage over the other.



Replica of Sputnik I

- In 1957, the Soviets [launched Sputnik](#), the world's first satellite, propelling us into the space age.
- This caught the US off-guard, and in response the U.S. government created [ARPA](#) (Advanced Research Projects Agency) which was mandated with the responsibility of the technological and scientific advancement of the country.

## The '60s-'70s: ARPANET #

- ARPA was meant to facilitate research. But **their computers could not talk to each other.**
- ARPA sent out a request for the **design of a network that would allow computers across the entire country to talk to each other.**
- A network called the **ARPANET** was developed over the course of a year. In September 1969, the ARPANET was turned on. **The first four nodes** were at UCLA, Stanford, MIT, and the University of Utah. Over the '70s, other computer networks just like ARPANET sprang up.

While the computers on one network could communicate via a default gateway, they could not communicate with computers on other networks.

- While the computers on one network could communicate via a default way to communicate set by the network administrator of each network, the computers on different networks could not communicate since every network had its own language or - more formally – **protocol**, which we will introduce in a [later lesson](#).
- Then, a standardized protocol called the **Transmission Control Protocol** was invented ([RFC 675](#)).
- It was also in [RFC 675](#) that the term “Internet” was first used. Later RFCs continued the use.



**Did You Know?** The Advanced Research Projects Agency (ARPA) is now known as Defense Advanced Research Projects Agency (DARPA) to emphasize the its focus on defense!

## The '80s: Protocols #

- ARPANET was fully migrated to TCP/IP.
- As we moved into the 1980s, computers were added to the Internet at an increasing rate. These computers were primarily from government, academic, and research organizations. Much to the surprise of the engineers, the early popularity of the Internet was driven by the use of electronic mail.

## The '90s: The World Wide Web

- During the 90's, the researchers at the **European Council for Nuclear Research (CERN)** felt the need for automated sharing of their findings between their computers. CERN had documents that cross-referenced other documents. So, there were these (hyper) links between documents.



Tim Berners Lee at CERN. Taken from:  
<https://www.flickr.com/photos/itupictures/16662336315>  
under CC BY 2.0

- In 1990 Tim Berners-Lee introduced his World Wide Web project to store and retrieve these inter-connected documents.
- Check out a restored version of [the first website ever!](#)
- Later, educational, commercial and so many other applications were realized.
- The World Wide Web got even more popular with the advent of browsers such as **Mosaic** and **Netscape** which allowed combining graphics with web navigation!

*... Creating the web was really an act of desperation because the situation without it was very difficult when I was working at CERN later. Most of the technology involved in the web, like the hypertext, like the Internet, multi font text objects, had all been designed already. I just had to put them together. It was a step of generalizing, going to a higher level of abstraction, thinking about all the documentation systems out there as being possibly part of a larger imaginary documentation system."*

– Tim Berners Lee

## Search Engines #

Another fun fact that initially, **there were no search engines**.

So how would you find a website? Well, you couldn't. Either you knew it or you didn't. Of course, you could land on a website by accident or by following a link from another website that you knew.

Then, people started creating [static indices of the web](#) - a categorized listing of websites. People would sit and randomly click on links to find web pages and add links to their index. But this couldn't scale. So, eventually, search engines

add links to their index. But this couldn't scale. So, eventually, search engines were “invented.”

Altavista and Yahoo! were among the earlier search engines. According to [this Wikipedia article](#), the first automated web index was World Wide Web Wanderer. Soon afterward, the first web search engine, the W3Catalog was invented.

## The 21st Century: The Age of the Internet #



And now here we are. In an age in which all our devices run on the Internet, even toasters for some – the era of the **Internet of Things** is here! Life without it seems almost inconceivable.

## A Video #

Here's the history of the Internet in a short video.



## Quick Quiz! #

1

Tim Berners Lee invented the Internet

COMPLETED 0%

1 of 7



Now that we have an overview of where the Internet comes from, let's discuss what it's actually made of in the next lesson!



# Internet Standards Documents

Let's look at what can be called the "official documentation of the Internet" now! We'll refer to these standards documents throughout this course.

## WE'LL COVER THE FOLLOWING ^

- Why Care About Internet Standards?
- What Is an RFC?
  - Who Writes RFCs?
  - History
- Contents of an RFC
- Types of RFCs
  - Standards Track
  - Historic
  - Unknown
- Quick Quiz!

## Why Care About Internet Standards? #

Standardization has allowed us to achieve **interoperability**. Different organizations and vendors can develop hardware and software to be connected to the Internet. Unless they agree on a protocol, their hardware and software wouldn't be able to talk to each other. Standardization is all interested stakeholders sitting together, debating and agreeing on a protocol or design.

Also, it's important to know **what's out there, who designed what and why**.

Also, you'd know where to submit your ideas in case you come up with a better design for any of the protocols we're going to study.

## What Is an RFC? #

- An RFC or [Request For Comments](#) is a document that contains proposals for new protocols or systems.
- Today, RFCs are submitted to and handled by the [Internet Society](#) which has a sub-body called the **Internet Engineering Task Force (IETF)**. This sub-body works on the standardization of Internet protocols and systems.
- An RFC is then deliberated on by experts, revised and then hopefully, eventually adopted as a standard.

## Who Writes RFCs? #

- RFCs are generally written by those who work at IETF, Internet researchers, and specialists. However, an RFC *can* be written by **anyone!** Yes, even you can write one. Just write up your findings and submit them to the Internet society's [Independent Submissions](#) page!
- All Internet protocols, like the world wide web, are described by one or more RFCs.

## History #

- RFCs were started by [Steve Crocker](#) to document details of ARPANET while it was being created. These documents were called **Requests For Comments** to encourage discussion and not seem too assertive. They used to be written on a typewriter and distributed around ARPA's office as physical copies with requests for comments.

## Contents of an RFC #

- RFC's generally start off with a header that contains the category of the document, its identification number, the name(s) of the author(s), and the date.
- Then the document contains its title, a status, and an abstract.
- Then a table of contents after which the document starts.
- The document usually starts with an introduction.

Here is an example of the first page RFC standards document, [RFC 2046](#).

Network Working Group  
Request for Comments: 2046  
Obsoletes: 1521, 1522, 1590  
Category: Standards Track

N. Freed  
Innosoft  
N. Borenstein  
First Virtual  
November 1996

Multipurpose Internet Mail Extensions  
(MIME) Part Two:  
Media Types

Status of this Memo

This document specifies an Internet standards track protocol for the Internet community, and requests discussion and suggestions for improvements. Please refer to the current edition of the "Internet Official Protocol Standards" (STD 1) for the standardization state and status of this protocol. Distribution of this memo is unlimited.

Abstract

STD 11, [RFC 822](#) defines a message representation protocol specifying considerable detail about US-ASCII message headers, but which leaves the message content, or message body, as flat US-ASCII text. This set of documents, collectively called the Multipurpose Internet Mail Extensions, or MIME, redefines the format of messages to allow for

- (1) textual message bodies in character sets other than US-ASCII,
- (2) an extensible set of different formats for non-textual message bodies,
- (3) multi-part message bodies, and
- (4) textual header information in character sets other than US-ASCII.

These documents are based on earlier work documented in [RFC 934](#), STD 11, and [RFC 1049](#), but extends and revises them. Because [RFC 822](#) said so little about message bodies, these documents are largely orthogonal to (rather than a revision of) [RFC 822](#).

An example of the first page of an RFC

## Types of RFCs #

### Standards Track #

There are two kinds of standards documents: **Proposed Standard**, and **Internet Standard**.

Proposed Standard documents are well reviewed and stable but not as mature as an Internet Standard document. Internet Standard documents are technically competent, practically applicable, and publicly recognized. Perhaps one of the most important standard documents that we know about from the [Internet history lesson](#) is the one on the Internet protocol, [RFC 791](#).

IETF has documented its internet standards process in [RFC 2026](#). Have a look if you want to learn more about it.

## Historic #

These RFCs are usually obsolete and contain details about technologies that are not in use anymore.

## Unknown #

Some RFCs cannot be categorized or often do not specify any new standards or findings. These are categorized as unknown.

You can browse all of these categories of RFCs on the [RFC retrieve page](#).

## Quick Quiz! #

1

Standardization enables \_\_\_\_\_

COMPLETED 0%

1 of 2



Now that we have a solid foundation to start learning all about computer networks, let's get right into it from the next chapter!

# Communication Over the Internet

Before we dive deeper into the course, let's study some key concepts to understand how communication over the Internet works

## WE'LL COVER THE FOLLOWING ^

- What Is a Protocol?
  - An Analogy
  - TCP
  - UDP
  - HTTP
- Packets
- Addressing
  - IP Addresses
  - Ports
- Quick Quiz!

## What Is a Protocol? #

### An Analogy #

Let's start with an analogy. Think of your routine conversations. They usually **follow a general pattern** dictated by predefined rules. For example, most conversations start with greetings and end with goodbyes. They probably go something like this:

You: Hello

Friend: Hey!

**...conversation ensues...**

You: Bye!

Friend: Goodbye :)

Turns out that end systems also follow such **protocols to communicate with each other effectively** on the network.

Formally, according to the Oxford Dictionary, a **protocol** is “a set of rules governing the exchange or transmission of data between devices.” In the next few chapters, we’ll study several network protocols in detail.

## TCP #

The **Transmission Control Protocol (TCP)** is one such protocol. It was created to allow **end systems to communicate effectively**. The distinguishing feature of TCP is that it ensures that data reaches the intended destination and is not corrupted along the way.

## UDP #

The **User Datagram Protocol (UDP)** is also one such key protocol. However, it **does not ensure** that data reaches the destination and that it remains incorrupt.

## HTTP #

**HyperText Transfer Protocol (HTTP)** is a web protocol that defines the format of messages to be exchanged between web clients, e.g., web browsers and web servers and what action is to be taken in response to the message. The World Wide Web uses this as its underlying protocol.

## Packets #

Now that we've established that end systems communicate with each other based on set protocols, let's discuss *how* they actually communicate.

Computers send messages to each other that are made up of ones and zeros (bits).

However, instead of sending messages of possibly *trillions* of bits all in one go, they're broken down into smaller units called **packets** to make transmission more manageable. These smaller sizes make transmission more manageable because most links are shared by a few end-systems. Sending smaller units in succession instead of one big file all in one go makes usage of the network fairer amongst end-systems.

We'll talk about the exact technical definition of a packet in a future chapter.

## Addressing #

So, applications communicate with each other by sending messages based on protocols. However, packets have to be addressed to a certain application on a certain end system. How do you do that out of potentially millions of end systems and hundreds of applications on each of them? The answer lies in addressing.

An address identifies a sending entity and a receiving entity.

## IP Addresses #

Every device that is connected to the Internet has an address called an 'IP Address' which is much like a mailing address.

- IP addresses are 32 bit numbers (in IP version 4).
- The human readable way for looking at these numbers is the **dotted decimal notation**, whereby the number is considered one octet of bits (8 bits) at a time. Those octets are read out in decimals, then separated by dots.
  - Hence, each number can be from 0 to 255. For example, **1.2.3.4**.

- Some IP addresses are reserved for specific functions. We'll discuss them in more depth in later lessons.

Check yours by running the following command on a shell on your local setup.

```
curl ifconfig.me -s
```



All of the code on our platform is run on one of our servers, and the output is returned and printed on your screen. Hence, **the IP address here belongs to an Educative server!**

## Ports #

Any host connected to the Internet could be running many network applications. In order to distinguish these applications, all bound to the same IP address, from one another, another form of addressing, known as **port numbers**, is used. Each endpoint in a communication session is identified with a unique IP address and port combination. This combination is also known as a **socket**. So in essence, ports help to address the packet to **specific applications** on hosts.

- IP addresses identify end systems but ports identify an application on the end system.
- Every application has a 16-bit port number. So the port number could range from 0 to  $2^{16} = 65535$ .
- The ports 0 – 1023 are reserved for specific applications and are called **well-known ports**.
  - For instance, port 80 is reserved for HTTP traffic.
- The ports 1024 – 49152 are known as **registered** ports and they are used by specific, potentially proprietary, applications that are known but not system defined.
  - SQL server for example, uses port 1433

- It is generally considered best practice not to use these ports for any user defined applications although there is no technical restriction on using them.
- The ports 49152–65535 can be used by user applications or for other purposes (dynamic port allocation for instance, but more on that later).

Here is a visual of these ports.



## Quick Quiz! #

1

Which of the following is a valid IP version 4 address?

COMPLETED 0%

1 of 4



In the next lesson, we'll study some physical and hardware aspects of

computer networks.

# The Edge of The Internet: End Systems

Here's an introduction to the important physical components of the Internet that make it tick!

## WE'LL COVER THE FOLLOWING ^

- End Systems
- The Network Edge
- Quick Quiz!

So far, we know that the Internet is a network of networks. Each network is made up of devices, called **end systems**, connected to each other with **communication links**. Let's look at each in more detail.

## End Systems #

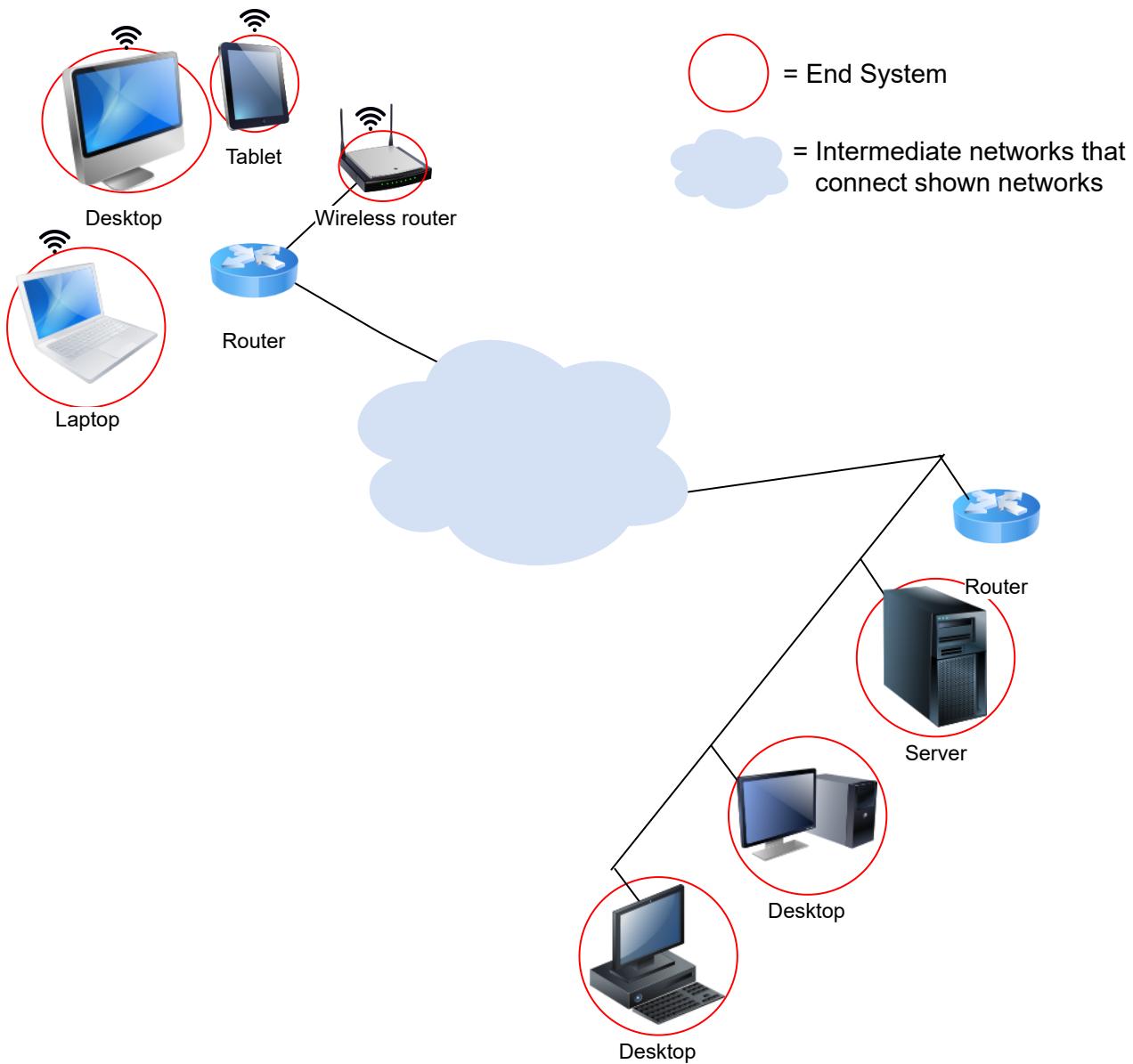
**End systems** are devices that are connected to the Internet. They include:

- Desktop computers
- Servers
- Mobile devices
- IoT devices.

So, an end system can be anything from a rack server to an Internet-enabled toaster. These devices are **often also called edge systems** in networking jargon because they are technically situated at the 'edge' of the Internet since **they don't relay data from one device to another**.

## The Network Edge #

The **network edge** is simply the collection of end-systems that we use every day: smartphones, laptops, tablets, etc. However, note that **devices that relay messages (such as routers) are not part of the edge of the Internet**.



The end systems out of them -- they reside on the "edge of the network"

Note that the two networks shown **could be connected through any number of intermediate networks** such as those for their Internet Service Providers. Since the actual path doesn't matter, we **obfuscate** the interconnectivity by using the **cloud symbol**.

## Quick Quiz! #

1

Which of the following is NOT an end system?

COMPLETED 0%

1 of 2



---

Now that we understand the basic structure of the Internet, let's look into how end systems access the Internet in the next lesson!

# What Are Access Networks?

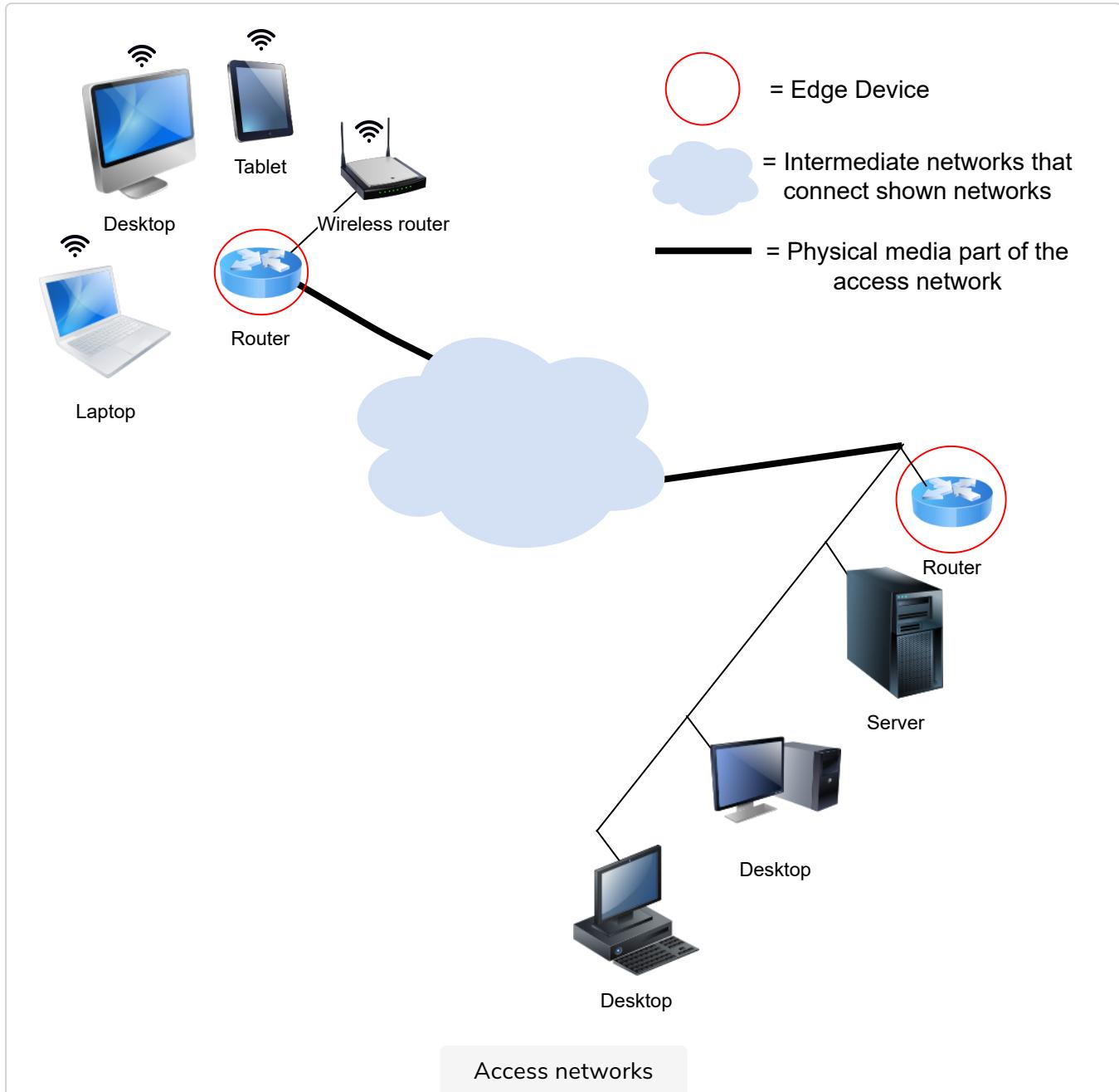
Let's study how end systems access the Internet.

## WE'LL COVER THE FOLLOWING ^

- Access Networks
- Transmission Rates
- Quick Quiz!

## Access Networks #

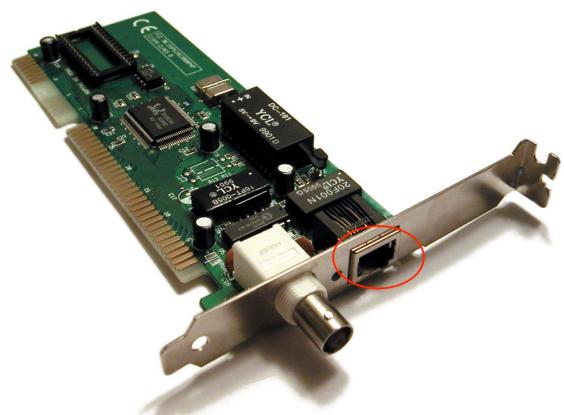
Access networks, also known as **the last mile**, consist of the media through which end systems connect to the Internet. In other words, **access networks** are networks that physically **connect end systems to the first router** on a path which connects them to some other end systems.



## Network Interface Adapter

The network interface adapter enables a computer to attach to a network. Since there are so many different types of networks, network adapters are used so that the user can install one to suit the network to which they want to attach.

Network interfaces also usually have an address associated with them. One



A network card. Taken from:

[https://commons.wikimedia.org/wiki/File:Network\\_card.jpg#filelinks](https://commons.wikimedia.org/wiki/File:Network_card.jpg#filelinks)

Notice the port that the cable would go into.

machine may have multiple such interfaces.

These interfaces are essentially the **physical gateways** that connect devices to the Internet.

Most machines then have external ports which network cables can be plugged into. The type of access network depends on the physical media involved. Here are some common access networks:

1. **Digital Subscriber Line (DSL)**
2. **Cable Internet**
3. **Fiber To The Home (FTTH)**
4. **Dial-Up**
5. **Satellite**
6. **WiFi**

We'll go through the what and how of most of these access network in the next few lessons.

But first, how is the speed of a network measured? What exactly is the unit of speed of a network? Let's have a look!

## Transmission Rates #

The rate at which data is transmitted from one point to another in a network is called the **transmission rate**. In other words, the speed of the network is its transmission rate.

The smallest unit that digital data can be divided into is a *bit*: a 1 or a 0. Transmission rates are measured in units of **bits/sec**. However, since bits/sec is a really small unit, multiples/prefixes are commonly used. Common prefixes and their interpretation is given below:

$$1 \text{ kilobit/s} = 1 \text{ kbps} = 1000 \text{ bits/s} = 10^3 \text{ bits/s}$$

$$1 \text{ megabit/s} = 1 \text{ mbps} = 1000000 \text{ bits/s} = 10^6 \text{ bits/s}$$

$$1 \text{ gigabit/s} = 1 \text{ gbps} = 1000000000 \text{ bits/s} = 10^9 \text{ bits/s}$$

For example, a speed of 240 Mbit/second means that 240, 000, 000 or 240 million bits get transmitted every second!

There are essentially **two** ways that data flows in a network: *from* an end system or *to* an end system. The outgoing transmission rate is called the **upload rate**, and the incoming transmission rate is called the **download rate**.

Some networks have varying upload and download transmission rates, called **asymmetric transmission rates**. This is useful because traffic going out from end hosts generally consists of small requests which solicit comparatively much larger responses.



**Note** You can check the upload and download transmission rate of *your* Internet connection from <https://www.speedtest.net>!

## Quick Quiz! #

1

What does 'asymmetric transmission rate' mean?

COMPLETED 0%

1 of 2



Now that we've covered the basics, let's move on to some detail about the physical media that actually make up these networks.

# Types of Access Networks: DSL

There are a number of ways that your end system can access the Internet, let's look at each in detail!

## WE'LL COVER THE FOLLOWING ^

- Digital Subscriber Line: DSL
  - Internet Service Providers
  - How DSL Works
  - Quick Quiz!

Now that we know *what* access networks are, let's look at some common types.

## Digital Subscriber Line: DSL #

A Digital Subscriber Line or **DSL** uses the existing groundwork of telephone lines for an Internet connection. DSL connections are generally provided by the same company that provides local wired phone access.

## Internet Service Providers #

An ISP is just the company that provides end users with an Internet connection. For instance, AT&T and Verizon are ISPs. So the telephone company or **telco** is the Internet Service Provider or **ISP** in the case of DSL!

## How DSL Works #

- A device on the home user's end called a **DSL modem** *modulates* the digital signals that a computer outputs into high-frequency analog audio signals that are out of the human voice and hearing range.
- The telephone wire's frequency spectrum is divided into 3 parts:
  1. A **downstream channel** (which is used to *receive* data), in the 50 kHz to 1 MHz frequency range, on 'band'

to 1 MHz frequency range or ‘band’

2. An **upstream channel** (used to *send* data) which takes up the 4 kHz to 50 kHz band
3. A **regular channel** used for telephone conversations taking up the 0 to 4kHz range

 **Did You Know?** Modulation - demodulation is where the name **MoDem** comes from.

For reference, the human hearing range goes from 20 Hz to 20 kHz and the average human voice range goes from 85 Hz to 255 Hz.



Let's see how many frequencies are allotted to what channel

1 of 4



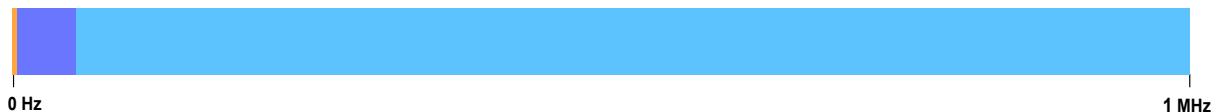
The regular channel takes up the 1-4kHz band

2 of 4



The upstream channel takes up the 4-50KHz band

3 of 4



The downstream channel takes up the 4-50KHz band

4 of 4



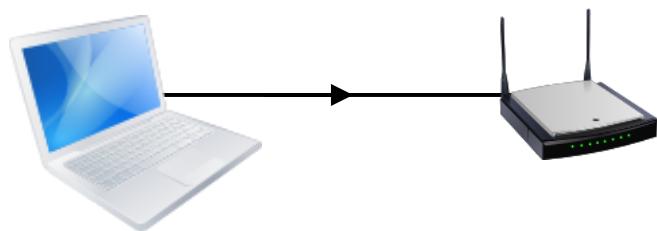
- These signals are then carried by telephone wires over to the ISP
- Then, these high-frequency analog signals are converted back to digital signals using a device at the ISP's end called a **Digital Subscriber Line Access Multiplexer (DSLAM)**.
- The signals are then forwarded to the end system that it was meant to reach

Here are slides that depict this process:



An end system

1 of 5

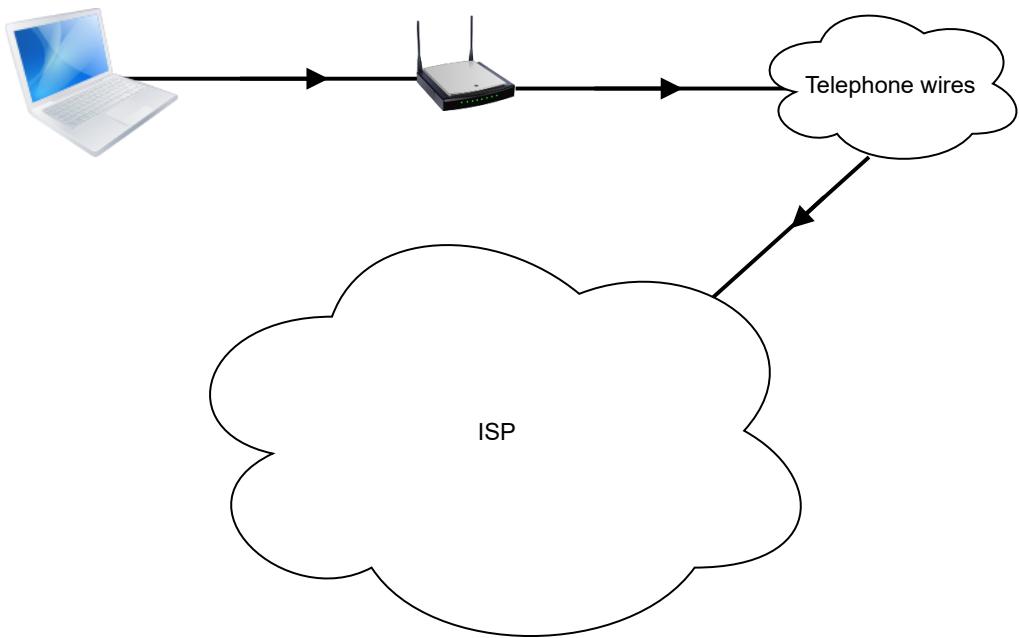


The end system outputs data to a 'DSL router' (a DSL modem and router combined into one)

2 of 5

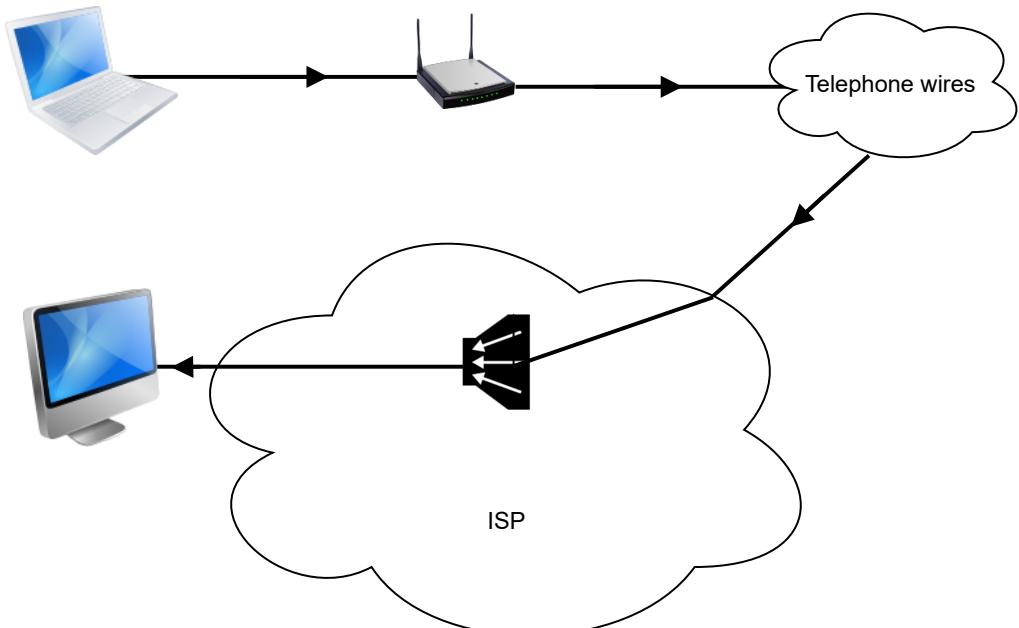


The DSL router encodes the digital data into audio frequencies and sends it over telephone wires towards the ISP



The ISP receives the signals

4 of 5



The DSLAM decodes the audio frequencies into digital signals and sends them over to the intended end system

5 of 5

Due to the asymmetry between the width of the three channels, this type of DSL is termed as **Asymmetric DSL (ADSL)**. **Symmetric DSL**, on the other hand, offers equal upstream and downstream bandwidth.

Did You Know? Steve Jobs and Steve Wozniak, founders of Apple Inc., built a **Blue Box** in the 1970s that allowed them to make free international telephone calls. It worked by generating the same audio frequencies into telephone receivers that were generated by operators to make long distance calls essentially bypassing the telephone company's toll collection system. With a little bit of knowledge, they ended up rigging an international infrastructure!



Steve Wozniak (left) and Steve Jobs (right):  
[https://www.flickr.com/photos/mac\\_filko/4309049355](https://www.flickr.com/photos/mac_filko/4309049355)

## Quick Quiz! #



What's a DSLAM?

---

Now that we have an overview of DSL, let's look at a few other common access networks in the next lesson!

# Types of Access Networks: Cable, FTTH, Dial-Up, and Satellite

Let's discuss a few other common access networks.

## WE'LL COVER THE FOLLOWING ^

- Cable Internet
  - How It Works
  - Slower During Peak Hours
  - Hybrid Fiber Coax
  - Transmission Rate
- Fiber To The Home: FTTH
  - Transmission Rate
- Dial-Up
  - Transmission Rate
- Satellite
  - Transmission Rate
- Quick Quiz!

## Cable Internet #

- In the case of cable Internet, the TV cable company is the ISP and it relies on the preexisting infrastructure of cable TV to grant Internet access.
- It runs on *coaxial cable*. Coaxial cable has enough of a frequency range to carry TV channels and a stream of upstream and downstream Internet.

## How It Works #

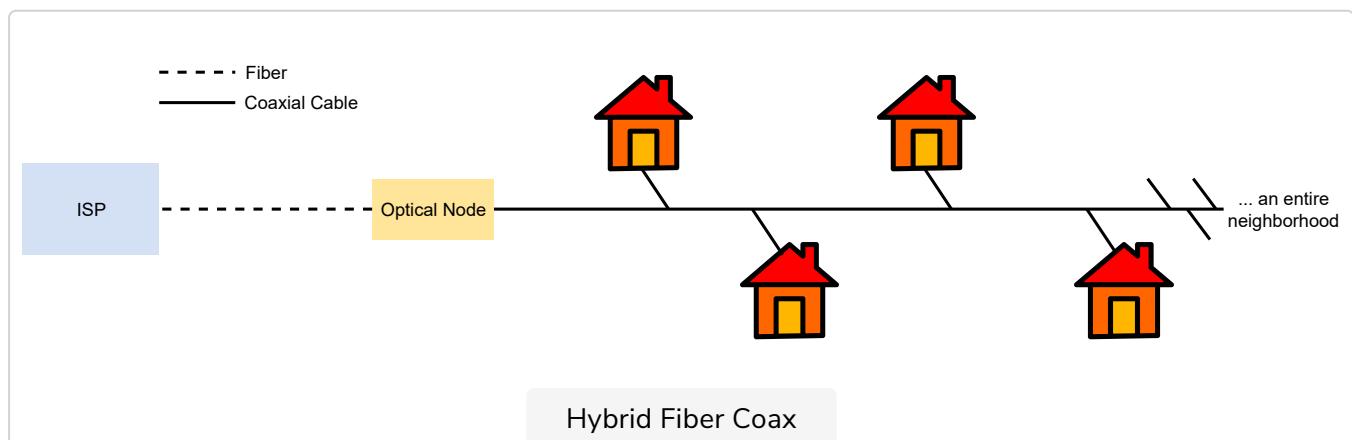
- In essence, cable Internet works very similarly to DSL.
- A device on both the user's end and the ISP's end modulates the analog signals to digital and vice versa.

## Slower During Peak Hours #

However, cable Internet can be slower during *peak hours*, when a majority of users are online at once. This is because cable Internet is a *shared broadcast medium*: every signal that comes from the ISP is sent to every single home regardless of which one it was meant for.

## Hybrid Fiber Coax #

Usually, cable Internet works with a combination of coaxial cable and optic fiber (which we'll discuss in the next chapter), where the fiber connects optical nodes that exist in every neighborhood to the ISP and coaxial cable further connects the nodes to the houses. This is sometimes referred to as a hybrid fiber coax (HFC). Have a look at the figure below to see how hybrid fiber coax works.



## Transmission Rate #

According to [DOCSIS 4.0](#), cable Internet can now operate in symmetric speeds (where both upstream and downstream channels have the same speed) of up to 10 gbps.

## Fiber To The Home: FTTH #

Although DSL and cable Internet are incredibly popular, Fiber To The Home or **FTTH** is another access method. Optic fiber cables are claimed to be the cleanest method to transmit data. We'll discuss them in more depth in the next lesson.

## Transmission Rate #

FTTH can be very fast – up to 2.5 gbps.

FTTH can be very fast – up to 2.5 gbps.

## Dial-Up #

Dial-Up uses a modem over the telephone line, but does not fully utilize the spectrum of the transmission medium. It only uses the traditional voice channel frequencies. Hence, it is slower than DSL.

## Transmission Rate #

Dial-up is non-broadband and very uncommon now. The speed is at most 56 kbps.

## Satellite #

The Internet can also be accessed via satellites. This can be beneficial in remote areas where other physical access networks are not available.

## Transmission Rate #

This would depend on a number of factors including the kind of satellite. Some setups can provide incredibly fast downlink and uplink connections, however, on average, the download rate is at around 1 mbps and the average upload rate 256 kbps.

## Quick Quiz! #

1

Which access network utilizes telephony infrastructure on the last mile?

COMPLETED 0%

1 of 4



---

In the next chapter, we'll start to look at types of computer networks.

# Types by Physical Medium: Guided Physical Media

Now that we've discussed the infrastructure of the edge of the Internet let's discuss some actual hardware components.

## WE'LL COVER THE FOLLOWING

- Communication Media
  - The Internet Is Under the Sea, Not in the Clouds!
  - Quick Quiz!



## Communication Media #

Data needs to be transmitted from one end system to another over a medium. There are two kinds of media: **guided** and **unguided**. Each has its own advantages and disadvantages. Let's discuss the common *guided* ones in more detail now.

## Guided Media

A medium in which the signal is transported on a **confined pathway** is called *guided*. Some commonly used examples are given below.



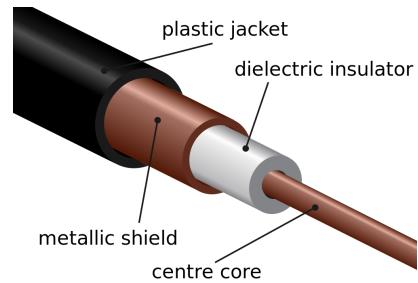
### Twisted Pair Copper Wire

This kind of wire is used in DSL and Dial-Up consists of pairs of copper wires that are twisted together to reduce electrical interference. Each wire is about a millimeter thick and is one communication link. Generally, several pairs are bundled together in a protective plastic or rubber wrapping.

Twisted pair copper wire taken from  
[https://commons.wikimedia.org/wiki/File:TwistedPair\\_FTP.jpg](https://commons.wikimedia.org/wiki/File:TwistedPair_FTP.jpg) under GNU free documentation license

## Coaxial Cable

Cable Internet runs on coaxial cables. A coaxial cable consists of one copper wire surrounded by an insulating material, followed by a mesh-like cylindrical metallic shield, followed by another insulating cover.



Coaxial Cable taken from

[https://en.wikipedia.org/wiki/File:Coaxial\\_cable\\_cutaway](https://en.wikipedia.org/wiki/File:Coaxial_cable_cutaway)  
under CC-BY-3.0

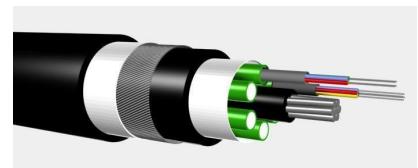
## Fiber Optics

Fiber optic cables carry light instead of electrical signals.

Metallic media suffer from electrical noise and interference from nearby electrical sources such as mains wiring. Since optical fiber carries signals in the form of light, it is **not susceptible to the abundant electrical noise and interference.**

Interference from other light sources is easily mitigated by opaque covering around the optic fiber. Hence, these cables can have incredibly fast transmission rates and can be stretched out over long distances, unlike the rest.

Optical fibers are frequently used in public and enterprise networks when the distance between the communication devices is larger than one kilometer.



fiber optic cable taken from

[https://commons.wikimedia.org/wiki/File:Optical\\_fiber\\_cable.jpg](https://commons.wikimedia.org/wiki/File:Optical_fiber_cable.jpg)  
under CC-BY-SA 3.0

There are two main types of optical fibers: **multimode and single mode.**

### Multimode

- Multimode uses LED send signals.
- Therefore it's, significantly cheaper than counterpart.

- It can work over several tens of kilometers.

- It can work over several tens of kilometers.
- Multiple light signals travel through the same optic fiber while reflecting off the edges of the fiber at different angles.

### Monomode

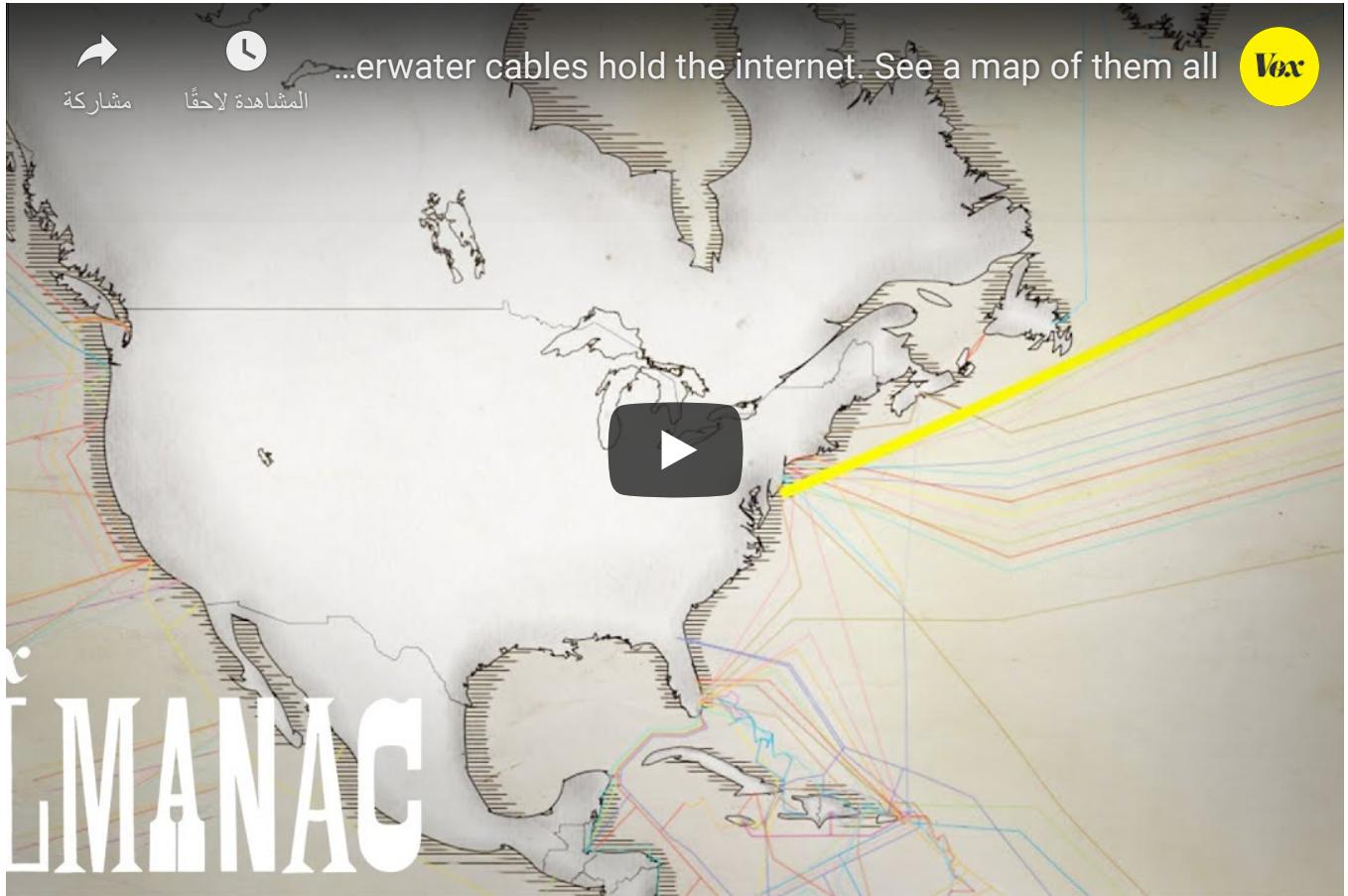
- Monomode uses **laser** for transmission.
- It's more expensive than multimode.
- Monomode fibers can only work over a few kilometers.

However, fiber optic has not dominated over the rest, because of the high cost of optical devices. However, **fiber to the home** is becoming increasingly common.

Also, check out the [Optical Carrier transmission rates](#). They specify the transmission rates of fiber optic cable. At the time of writing, they range from 51.84 Mbit/sec to 200 Gbit/sec!

## The Internet Is Under the Sea, Not in the Clouds! #

Most cross country connections, in fact, are made over fiber optic cable under the sea. If you're more interested in how underwater cables work, here is an interesting YouTube video by Vox called "*Thin underwater cables hold the Internet. See a map of them all.*"



## Quick Quiz! #

1

Why is fiber optic cable not as popular for commercial use as one would expect considering the advantages of high speed minimal loss transmission?

In the next lesson, we'll look at **unguided physical media!**

# Types by Physical Medium: Unguided Physical Media

In this lesson, we'll discuss unguided physical media.

## WE'LL COVER THE FOLLOWING



- Unguided Media
- Terrestrial Radio Channels
  - Long-Term Evolution (LTE)
- Free-space optical communication
- Quick Quiz!

## Unguided Media #

Means of transmission that are **not bound by a confined pathway** are called *unguided media*, such as radio waves. Let's discuss them in more detail.

## Terrestrial Radio Channels #

Radio waves encompass a certain band of the electromagnetic spectrum. They provide many advantages, such as not needing to be physically wired through a building, the ability to cross physical barriers like walls and other objects, and allowing for end systems to be mobile. However, they also have some disadvantages: they're considered to be less secure since interception is relatively easy.

There are 3 kinds of terrestrial radio channels:

1. The kind that operate over very short distances  
1-2 meters (**Bluetooth**)
2. The kind that operate over a few 10s to a few hundreds of meters (**WiFi**)

### 3. Those that operate over a range of kilometers **(3G, 4G, and LTE)**

The same infrastructure that provides access to cellphones provides access to the Internet. Users typically only need to be within a few kilometers of a base station to connect. Let's look at a couple of examples.

#### Third-Generation Wireless (3G)

**Third-generation wireless**, or **3G**, allows wide area Internet access that utilizes existing telephone networks. It can provide speeds of the order of 1 Mbps.



the top of a cell tower taken from  
[https://en.wikipedia.org/wiki/Cell\\_site#/media/File:Cell\\_Tower.jpg](https://en.wikipedia.org/wiki/Cell_site#/media/File:Cell_Tower.jpg) under CC-BY-SA 2.5

#### Long-Term Evolution (LTE) #

**Long-Term Evolution** or **LTE** is rooted in 3G technology, but is faster and can achieve transmission rates of 10s of Mbps. Don't let the name confuse you, it is not technologically very different from 3G.

#### Free-space optical communication #

Free space optical communication is a medium that **employs light to transmit data**. Among the many uses are: communication in space and in remote controls.

In free space optics, lasers can be used to achieve high data rates. However, free space optics suffers from interference by factors like fog, dust particles and smog. Recently, researchers have demonstrated the utility of free space optics for high speed communication in data centers.

#### Quick Quiz! #

Quick quiz on physical media!

1

Which of the following is not guided media?

COMPLETED 0%

1 of 2



In the next lesson, we'll study different types of networks.

# Types by Geographical Distance

In this lesson, we'll discuss the types of networks based on geographical distance.

## WE'LL COVER THE FOLLOWING



- Introduction
- Local Area Networks (LANs)
  - Ethernet
  - WiFi
- Metropolitan Area Network (MAN)
- Wide Area Networks (WAN)
  - SONET/SDH
  - Frame Relay
- Quick Quiz!

## Introduction #

Computers or end systems are generally connected together to share resources and information such as an Internet connection and devices such as printers. These networks can be classified by the geographical distance that they span. Have a look.

## Local Area Networks (LANs) #

A Local Area Network, or a **LAN**, is a computer network in a small area like a home, office, or school.



**Note** ‘small area’ does not imply anything about the number of end systems connected together – just the geographical area. A LAN can consist of hundreds or even thousands of systems.

Let's discuss some examples of LANs which also are access networks that we skipped previously.

## Ethernet #

Most LANs consist of end hosts connected using Ethernet network adapters to Ethernet switches. Every Ethernet switch has a limited number of ports, and therefore can interconnect a limited number of end hosts. Larger networks within a building are built using multiple Ethernet switches interconnecting different sets of end hosts. These switches may then be connected to each other and the Internet

## WiFi #

Increasingly, however, wireless Internet access has become very common. In Wireless LANs or WLANs, a wireless router interconnects different “subnets” and/or may have connectivity to the Internet, which it can extend to the hosts connected to it.

## Metropolitan Area Network (MAN) #

A **metropolitan area network (MAN)** is a computer network that spans the geographical distance of a metropolitan area, such as a city. A MAN may also refer to a set of interconnected LANs via point-to-point links, for example, on a university network. Recently, wireless MANs have become increasingly common.

## Wide Area Networks (WAN) #

**Wide Area Networks** or **WANs** refer to networks that allow interconnection across large distances. They may span over cities or even countries.

WANs are typically optical fiber-based. **Frame relay, ATM, and SONET/SDH** are examples of technologies that may be in use.



**Note** The Internet itself is a whole lot of LANs, interconnected by means of MANs and WANs.

## SONET/SDH #

Synchronous optical networking (SONET) and the international equivalent, Synchronous Digital Hierarchy (SDH) carries data as optical signals over optical fiber, which means that they can cover large distances. These technologies are incredibly prevalent today.

## Frame Relay #

A frame relay was a popular way to connect your LAN to the Internet or to provide an interconnection between LANs at two or more company sites.

## Quick Quiz! #

1

A computer network that spans a large geographical area connecting several sites of an organization, such as a university or company, possibly across many countries is called a \_\_\_\_\_

COMPLETED 0%

1 of 2



In the next lesson, we'll study some common network topologies!



# Types by Topology

We'll study the types of computer networks based on topologies in this lesson.

## WE'LL COVER THE FOLLOWING ^

- Bus
  - Limitations
- Ring
  - Limitations
- Star
  - Limitations
  - Tree
- Mesh
  - Limitations
- Quick Quiz!

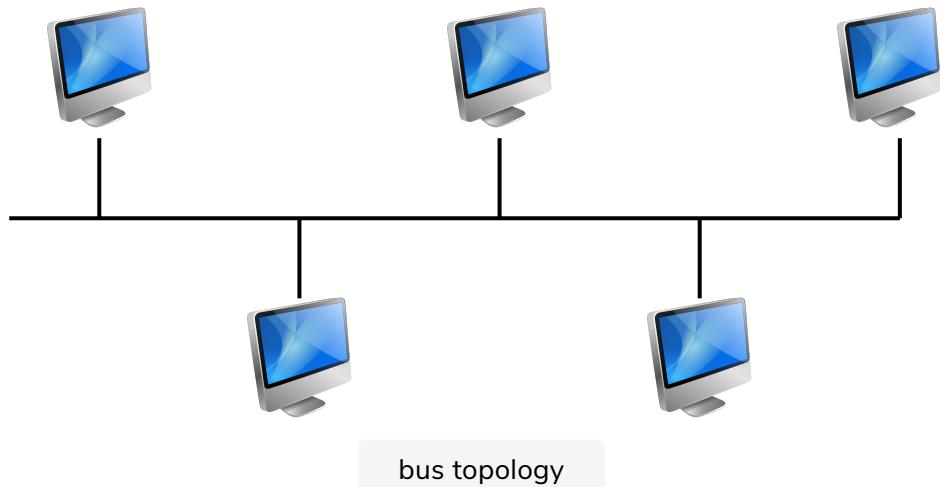
Computer networks can also be categorized in terms of **network topologies** that you might have studied about in a high school computer science class. These topologies include: **bus, ring, star, tree, and mesh**. Note that these topologies are strictly *logical*, i.e., they do not dictate how the wires would be connected physically, but they do dictate how the data flows in the network.

## Bus #

Every end system will receive any signal that is sent on the main or **backbone** medium. The medium can be guided or unguided.

## Limitations #

- A break in the cable will disrupt the entire network.
- Only one system can transmit at a time.

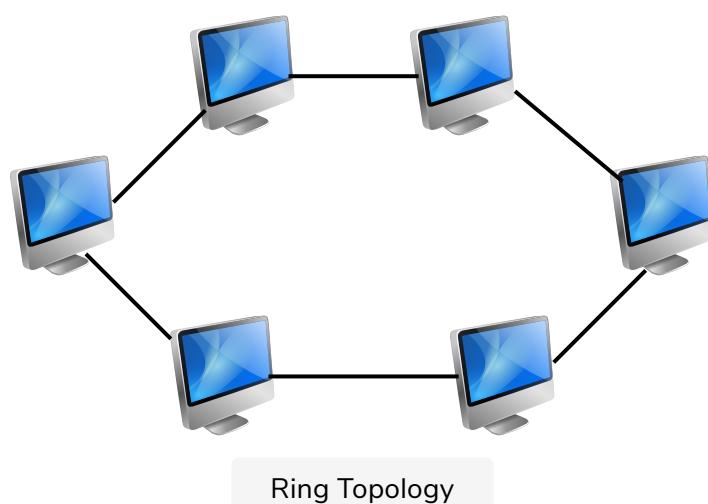


## Ring #

In this topology, end systems communicate with each other **through each other**. So the message travels along the ring passing each system until the target system itself is reached. Theoretically,  $n/2$  systems can be transmitting to their adjacent neighbor at the same time.

## Limitations #

- The basic ring topology is unidirectional so  $n - 1$  end systems would have to transfer messages for end system #1 to talk to end system #n
- A break in the cable will disrupt the entire network.

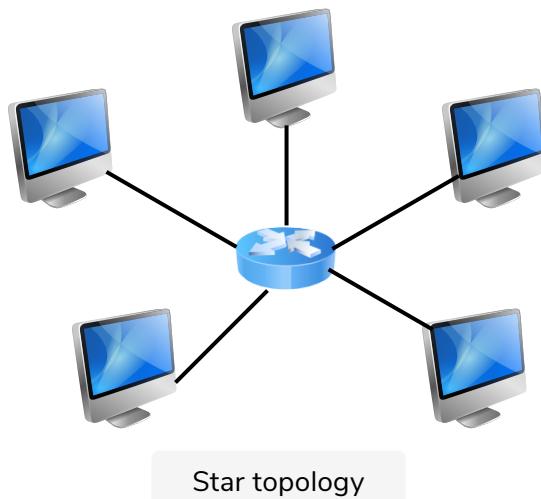


## Star #

All end systems talk to each other through one central device such as a router or switch. Routers and switches are discussed in-depth in the data link layer chapter!

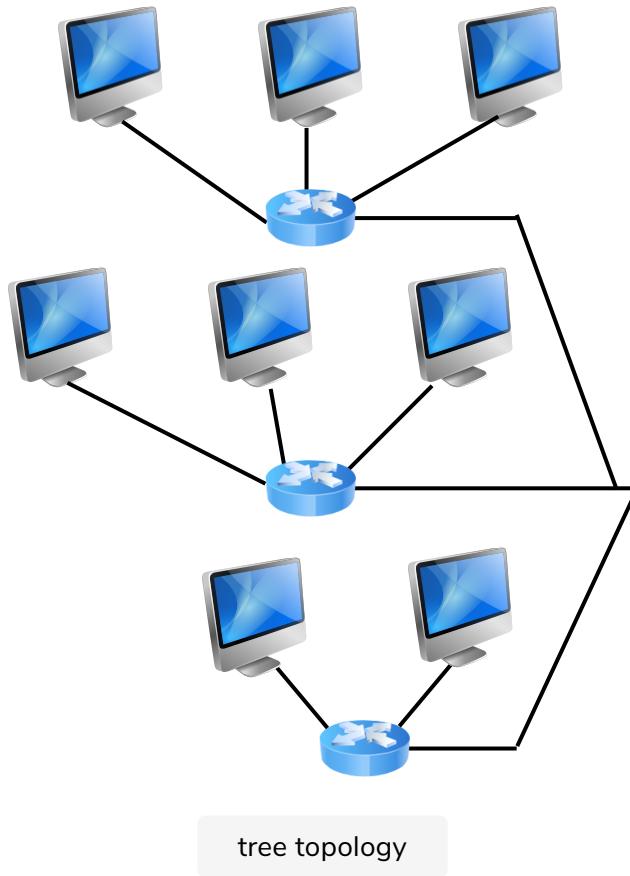
## Limitations #

- Hosts can all be transmitting at the same time. However, if the central device fails, the network is completely down.



## Tree #

This topology is also known as the **star-bus** topology. It essentially consists of a bunch of star networks connected together with a large bus.



## Mesh #

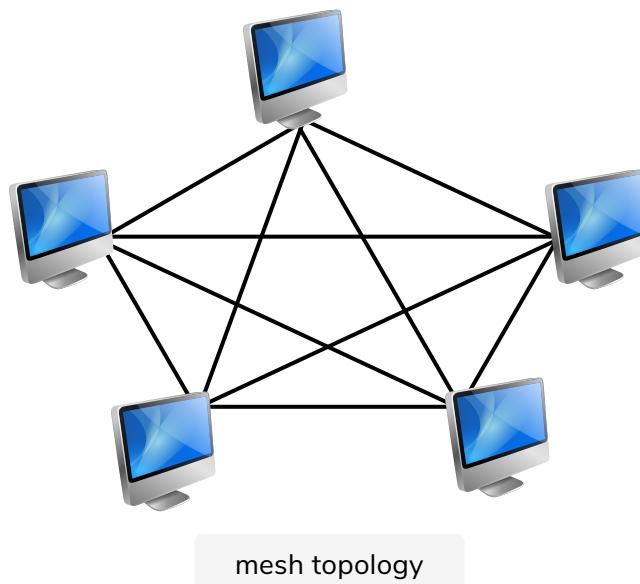
In this topology, every end system is **directly connected** to every other end

system.

## Limitations #

The mesh topology (if physically realized as a mesh):

- Is expensive
- Hard to scale
- Used in specialized applications only



mesh topology

## Quick Quiz! #

1

A disadvantage of the bus topology is that if the backbone wire is broken, the network may get negatively impacted

COMPLETED 0%

1 of 2



Starting in the next lesson, we'll learn about how the working of the Internet is organized into conceptual layers.

is organized into conceptual layers.

# Layered Architectures & Protocol Stacks

Layered architectures are a way to organize computer networks. Let's dive right in!

## WE'LL COVER THE FOLLOWING ^

- Introduction to Layered Architectures
  - Why Layers?
- An Analogy: Post
- Layers As Services To Each Other: Layers Are Vertical
  - Vertical Layers in Post
  - Vertical Layers in Networks
- Layers Communicate with Their Parallels: Layers Are Horizontal
  - Horizontal Layers in Post
  - Horizontal Layers in Networks
- Layers Evolve Independently
  - Independent Evolution in Post
  - Independent Evolution in Networks
- Encapsulation & Decapsulation

## Introduction to Layered Architectures #

When building a large complex system, it helps to approach the problem at gradually increasing levels of abstraction. Thus, systems can be composed of **layers**, each performing a specific set of tasks.

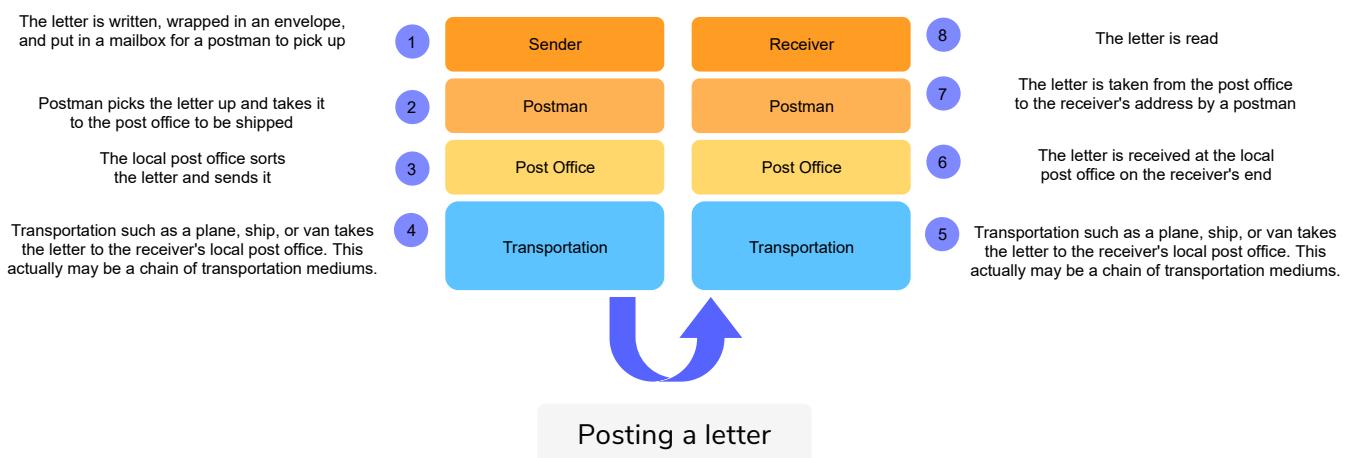
## Why Layers? #

Layered architectures give us modularity by allowing us to discuss **specific, well-defined parts of larger systems**. This makes **changing implementation-level details** and **identifying bugs easier**.

## An Analogy: Post #

Before we dive deep into different models of the network layer stack, let's look at an interesting analogy.

Think about posting a letter or a package. The general steps to doing so are as follows,



Notice that a few things are in **parallel with computer networking** here.

Here are some examples of how that is the case:

## Layers As Services To Each Other: Layers Are Vertical #

**Each layer provides some services to the layer above it.** Furthermore, the layer above is **not concerned with the details of how the layer below performs its services.** This is called **abstraction**. So in this way, the layers communicate with each other in a *vertical* fashion.

### Vertical Layers in Post #

In our letter analogy, each layer is servicing the layer above it. For instance, the postman provides services to the senders and receivers. They collect dropped letters from mailboxes and deliver mail to the houses.

Furthermore, all a sender knows and cares about is that once they write a letter, put it in an envelope, stick a stamp on it and drop it in a letterbox, it will eventually be delivered at the destination. Whether it's transported on pickup trucks, on railway trains or by air is irrelevant and immaterial to senders. So, how layer 4 does its job is irrelevant to the layers above, and that's called abstraction.

that's called **abstraction**.

## Vertical Layers in Networks #

Similarly, computer networks are conceptually divided into layers that each serves the layer above and below it.

- For example, the top layer in most layered models is called the **application layer**. End-user applications live in the application layer, which includes the web and email and are almost always implemented in software. The application layer is also where an outgoing message starts its journey.
- The application needs an underlying service that can get application messages delivered from source to destination and bring back replies which is what the layer(s) after do(es).

Since the underlying layer collects messages from the upper layer for delivery to the destination and hands over messages destined for the upper layer, it **serves the application layer**. Furthermore, the application layer **abstracts**, and hence is not concerned with any implementation details of the layers below.

## Layers Communicate with Their Parallels: Layers Are Horizontal #

Note that each layer at the sending end has a parallel in the receiving end.

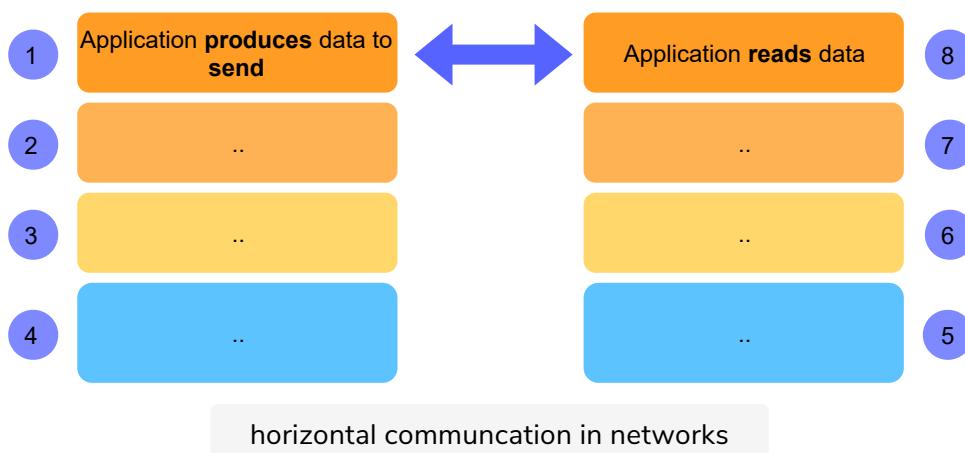
## Horizontal Layers in Post #

In the post analogy, the letter writer and receiver appear to be in direct communication with each other. The writer writes, the reader reads, oblivious to the man-hours spent in the lower layers. Similarly, the post office at the sender's side is in communication with some other post office. They cooperate in getting the letter delivered. At the lowest layer, there could be multiple hops. For instance, there is a bicycle pickup of letters from a box, delivered to the post office. Then, the letters are bundled and sent by pickup truck to an airport. The airport flies the postage to a different airport. The airport sends the postage to a post office by a pickup truck and the delivery ensues. Sometimes there are multiple entities horizontally, but we only see the sender

and the receiver.

## Horizontal Layers in Networks #

This makes more sense in the case of computer networks. For example, applications in the **application layer** send and receive data from the network. The application layer on one end system has a parallel on another end system, i.e., a chat app on one end system could be communicating with a chat app on another. **These applications in the application layer are seemingly communicating with each other directly or horizontally.** They are not aware of the layer below.



## Layers Evolve Independently #

Any lower layer in this model provides certain services that the upper layer can build other services upon. The upper layer can evolve to build different kinds of services, like going from text-based email to attachments, to the world wide web, to dynamic websites, interactive gaming, interactive video conferencing and so on, all happening in the top layer over the same infrastructure.

## Independent Evolution in Post #

1. For instance, the item being sent does not necessarily have to be a **letter** – It can also be a **package**.
2. It can be put in an **envelope or a box**.
3. It can be taken to the **post office, dropped off into a post box, or picked up**.
4. The receiver's end can be a **P.O. box, a home or an office**.

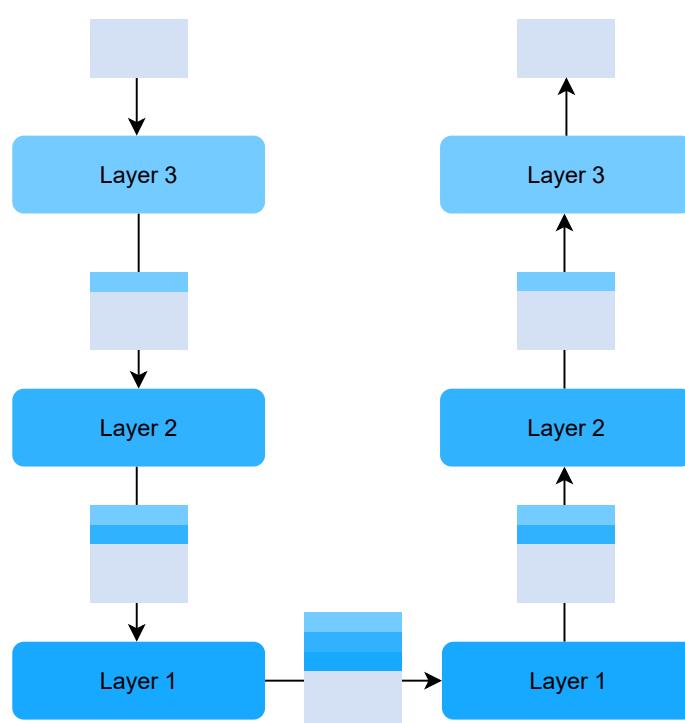
## Independent Evolution in Networks #

The applications in the application layer can send and receive almost any form of data, be it an **mp3 file or a word document**.

## Encapsulation & Decapsulation #

Each layer adds its own header to the message coming from above and the receiving entity on the other end removes it. The information in each header is useful for transmitting the message to the layer above. Adding the header is called encapsulation and removing it is called decapsulation. Have a look at the following slides to see how this works.

Take a look at the following drawing. We have not given names to these layers because we have not introduced them yet, but the general idea is depicted.



Encapsulation and decapsulation

Now that we have an introduction to layered architectures, let's discuss a popular network reference model in the next lesson.



# The Open Systems Interconnection (OSI) Model

The OSI layer model will help us to understand the overall picture of how computer networks work without getting into too many low-level details.

## WE'LL COVER THE FOLLOWING ^

- Common Models
- The OSI Model
- The Layers of the OSI Model
  - Mnemonic
  - Application Layer
  - Presentation Layer
  - Session Layer
  - Transport Layer
  - Network Layer
  - Data Link Layer
  - Physical Layer
- Quick Quiz!

## Common Models #

There are several models along which computer networks are organized. The two most common ones are the **Open Systems Interconnection (OSI)** model and the **Transmission Control Protocol/Internet Protocol (TCP/IP)** model.

We will discuss each model and the differences between the two in detail starting with the OSI model.

## The OSI Model #

The OSI Model was developed in the '70s by the Organization for Standardization (ISO). At this time, the Internet was in its infancy, but it

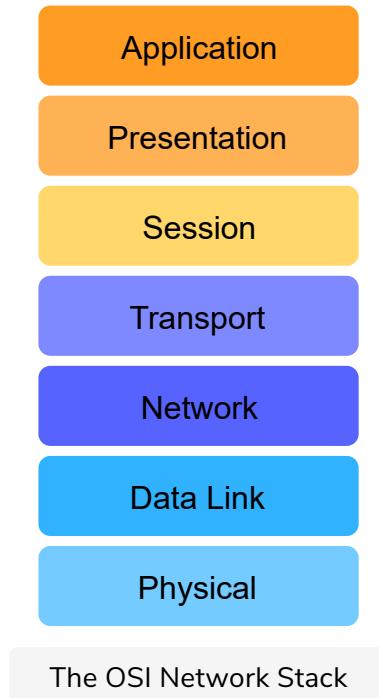
Standardization (ISO). At this time, the Internet was in its infancy and its

protocols had not fully matured. The OSI model **provides a standard** for different computer systems to be able to communicate with each other.

## The Layers of the OSI Model #

The model splits up a communication system into **7 abstract layers**, stacked upon each other.

Here are the seven layers of the OSI Model.



## Mnemonic #

A good mnemonic device to help remember these layers is:

**Please Do Not Throw Sausage Pizza Away**



Network protocols are implemented in software, hardware or a combination of both, and their hardware and software components are organized into these layers. So the **main purpose** of this ‘network stack’ is to **understand how the components of these protocols fit into the stack and work with each other.**

Here are some key responsibilities of each layer. Note that we are listing only *some* of the responsibilities of each layer. The exhaustive discussion is deferred to later chapters.

## Application Layer #

- These applications or protocols are almost **always implemented in software.**
- **End-users interact** with the application layer.
- The application layer is where most **end-user applications** such as web browsing and email **live**.
- The application layer is where an outgoing message starts its journey so it **provides data for the layer below.**

 **Did You Know?** Ascribing to the OSI model, **Layer 8** is a pseudo-layer that consists of the **end-users!** A lot of IT support humor has involved telling unsuspecting non-tech-savvy users that the issue is a “Layer 8 issue.” So the application layer can technically be said to be sitting in between layers 8 and 6!

## Presentation Layer #

- **Presents data** in a way that can be easily understood and displayed by the application layer.
  - **Encoding** is an example of such presentation. The underlying layers might use a different character encoding compared to the one used by the application layer. The presentation layer is responsible for the

translation.

- **Encryption** (changing the data so that it is only readable by the parties it was intended for) is also usually done at this layer.
- **Abstracts:** the presentation layer assumes that a user session is being maintained by the lower layers and transforms content presentation to suit the application.
- **End-to-end Compression:** The presentation layer might also implement end to end compression to reduce the traffic in the network.

## Session Layer #

- The session layer's responsibility is to take the services of the transport layer and build a service on top of it that **manages user sessions**.
  - As we will see shortly, the transport layer is responsible for transporting session layer messages across the network to the destination. The session layer must manage the mapping of messages delivered by the transport layer to the sessions.
- A session is an exchange of information between local applications and remote services on other end systems.
  - For example, one session spans a customer's interaction with an e-commerce site whereby they search, browse and select products, then make the payment and logout.
- **Abstracts:** the session layer assumes that connections establishment and packet transportation is handled by the layers below it.

## Transport Layer #

- The **transport layer** also has protocols implemented largely in software.
- Since the application, presentation and session layers may be handing off large chunks of data, the transport layer segments it into smaller chunks.
  - These chunks are called **datagrams or segments** depending on the protocol used.
- Furthermore, sometimes some **additional information** is required to transmit the segment/datagram reliably. The transport layer **adds this**

## information to the segment/datagram.

- An example of this would be the **checksum**, which helps ensure that the message is correctly delivered to the destination, i.e. that it's not corrupted and changed to something else on the way.
- When additional information is added to the **start** of a segment/datagram, it's called a **header**.
- When additional information is appended to the **end** it's called a **trailer**.

## Network Layer #

- Network layer messages are termed as **packets**.
- They facilitate the **transportation of packets** from one end system to another and help to **determine the best routes** that messages should take from one end system to another.
- **Routing protocols** are applications that run on the network layer and exchange messages with each other to develop information that helps them route transport layer messages.
- **Load Balancing** There are many links (copper wire, optical fiber, wireless) in a given network and one objective of the network layer is to keep them all roughly equally utilized. Otherwise, if some links are under-utilized, there will be concerns about the economic sense of deploying and managing them.

## Data Link Layer #

- Allows directly connected hosts to communicate. Sometimes these hosts are the only two things on a physical medium. In that case, the challenges that this layer addresses include **flow control** and **error detection/correction**.
- **Encapsulates packets** for transmission across a single link.
- **Resolves transmission conflicts** i.e., when two end systems send a message at the same time across one singular link.
- **Handles addressing** If the data link is a broadcast medium, addressing is another data link layer problem,
- **Multiplexing & Demultiplexing:**
  - Multiple data links can be multiplexed into something that appears

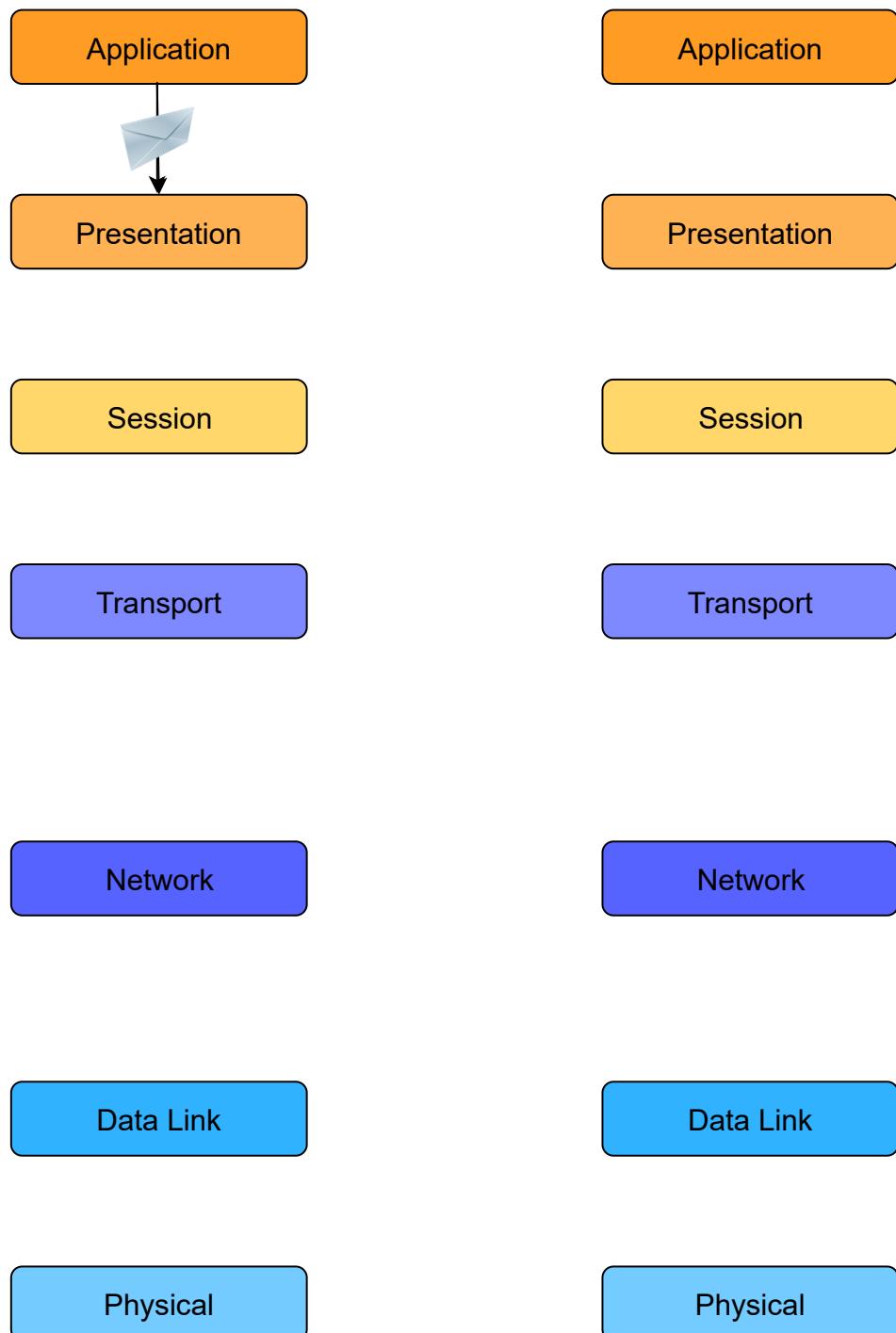
- Multiple data links can be multiplexed into something that appears like one, to integrate their bandwidths.
- Likewise, sometimes we disaggregate a single data link into virtual data links which appear like separate network interfaces.

## Physical Layer #

- Consists largely of hardware.
- Provides a solid electrical and mechanical medium to transmit the data.
- Transmits bits. Not logical packets, datagrams, or segments.
- Also has to deal with mechanical specifications about the makeup of the cables and the design of the connectors.

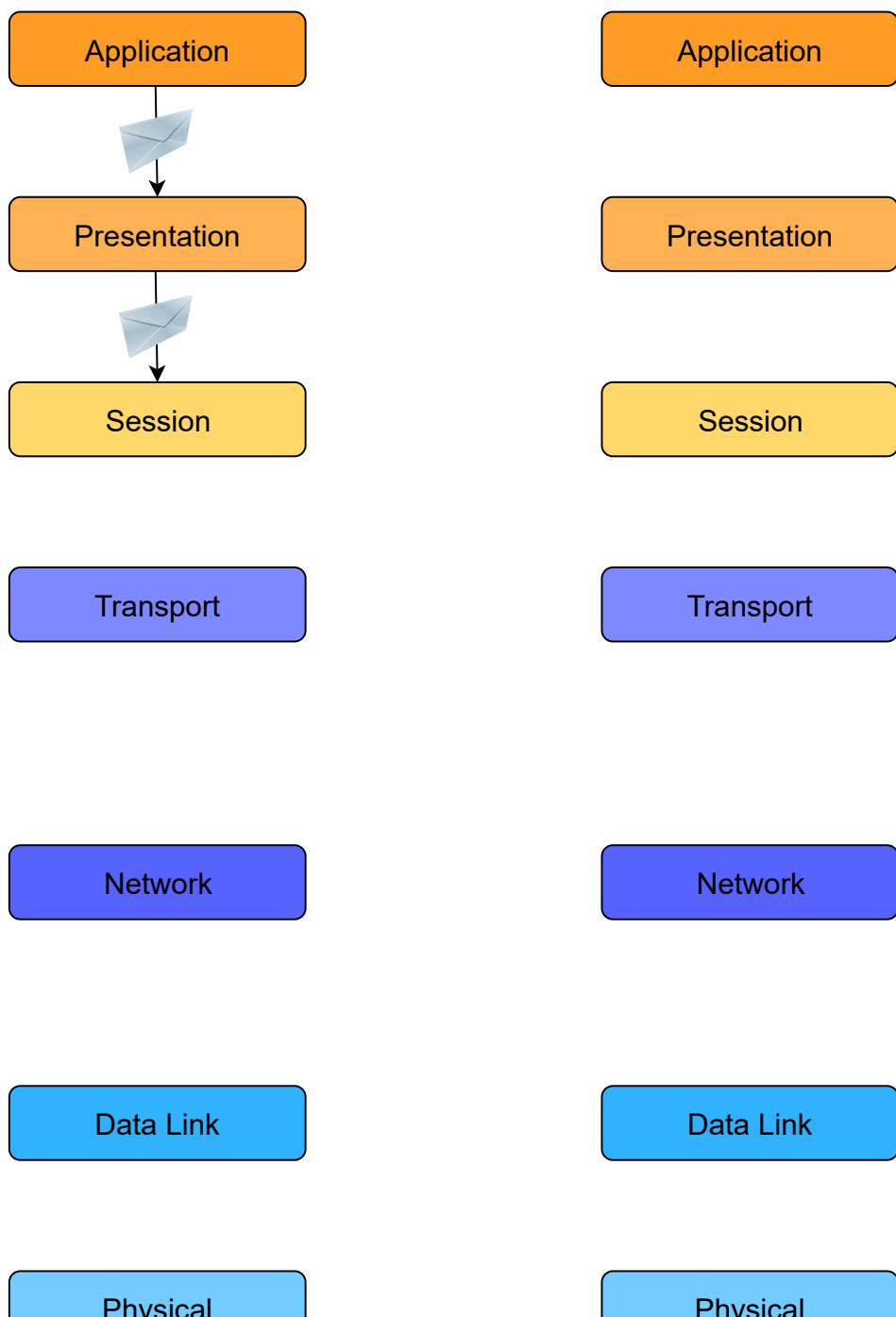
We've mostly already studied what constitutes the physical layer. We don't need to know more than what we've looked at in the [Physical Communication Media chapter](#).

Have a look at the following slides to understand how data would conceptually travel through the layers.



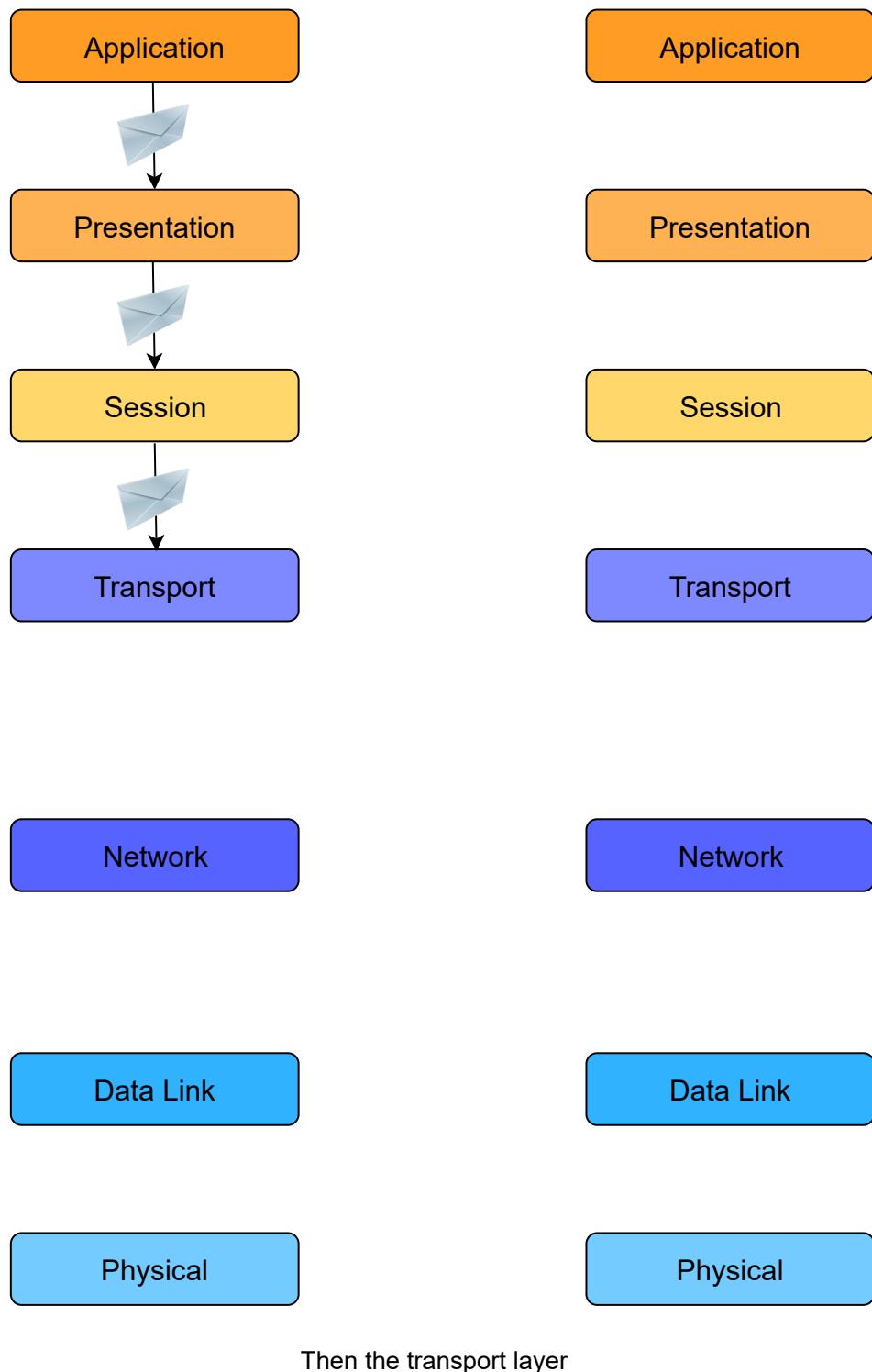
The application layer writes out streams  
of data to the presentation layer

How data conceptually travels through the layers

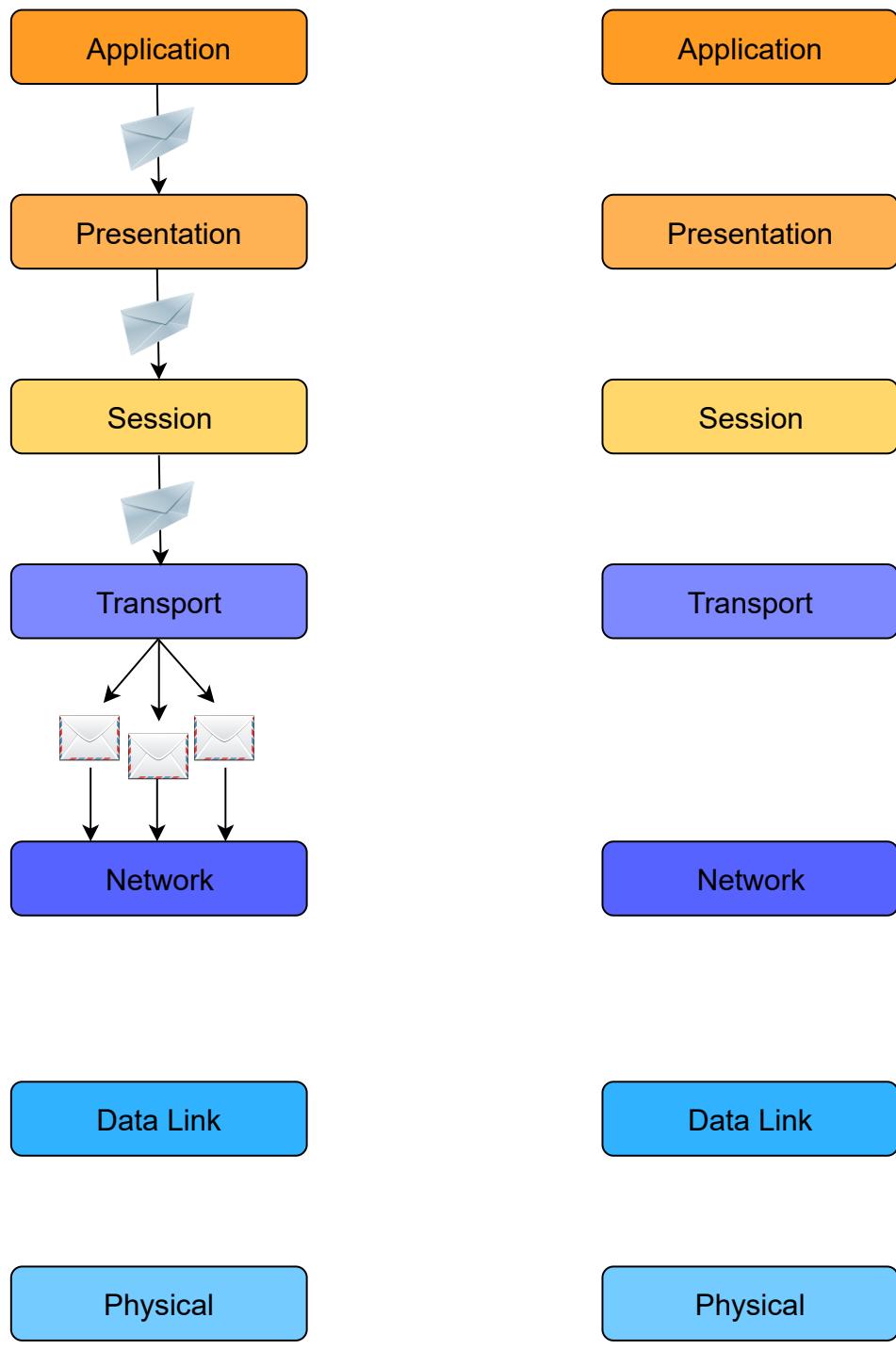


The presentation layer then hands it off to the session layer

How data conceptually travels through the layers

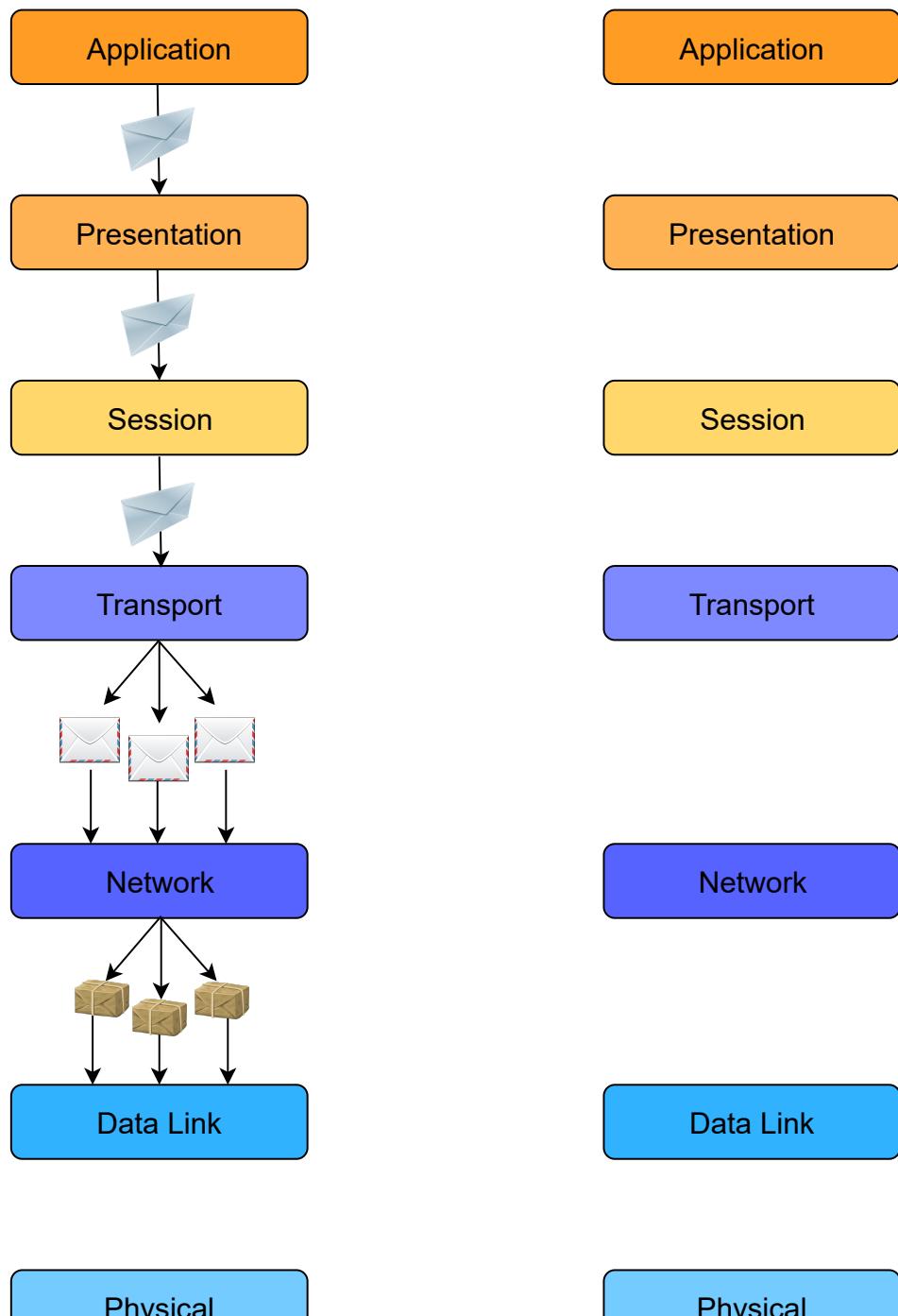


How data conceptually travels through the layers



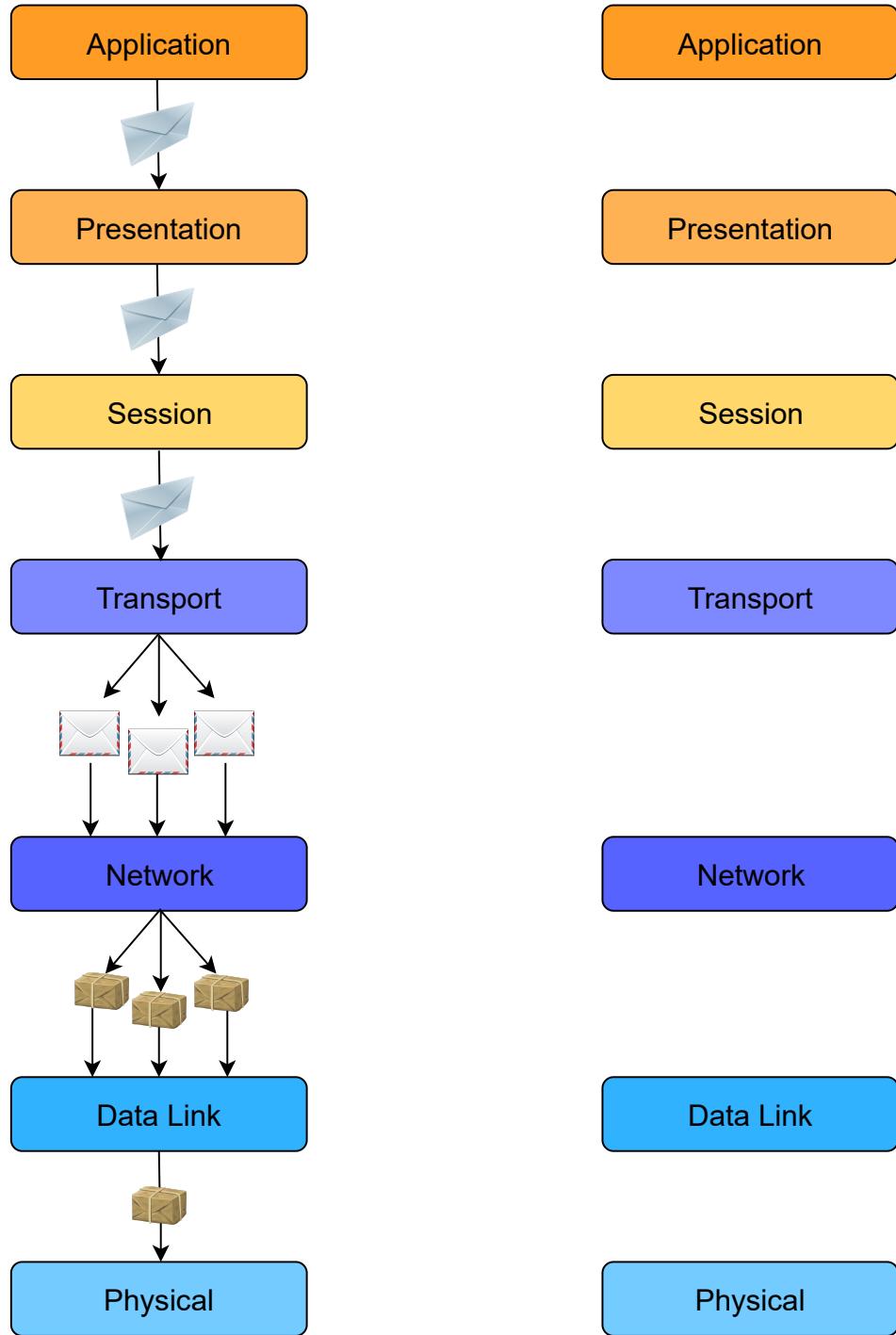
The transport layer segments this data

How data conceptually travels through the layers



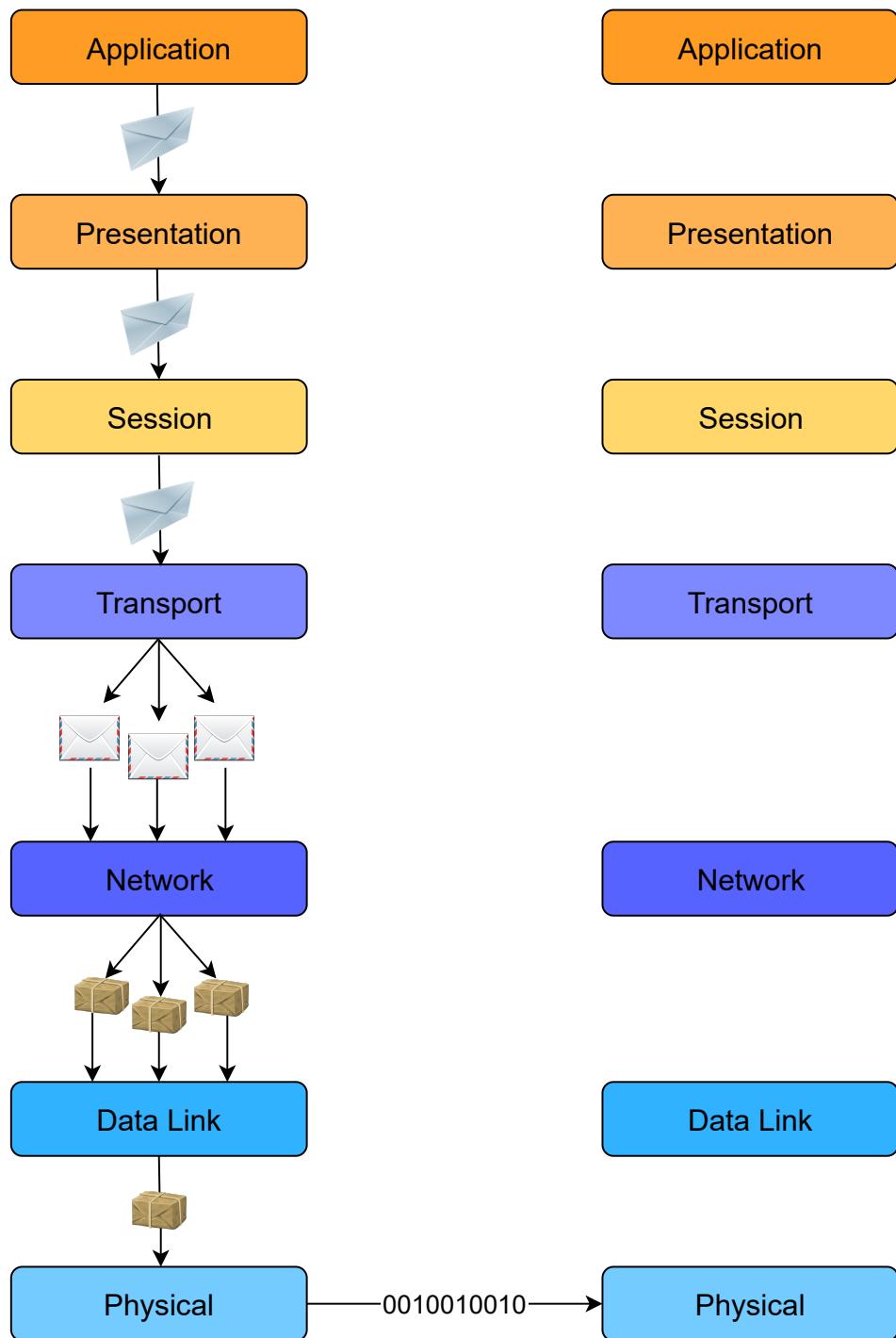
The network layer further packetizes these segments

How data conceptually travels through the layers



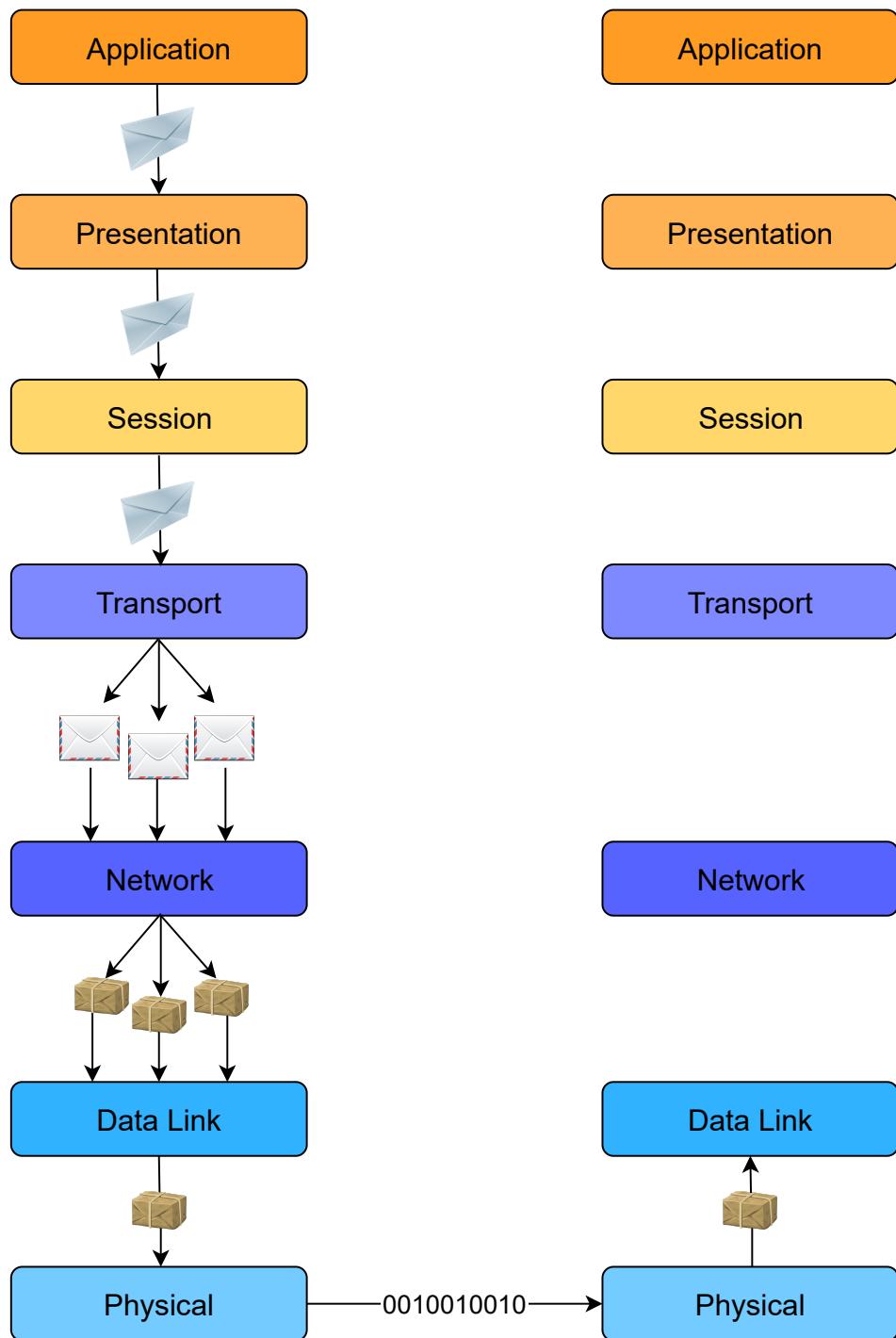
The physical layer then carries these in terms of bits represented physically in a medium

## How data conceptually travels through the layers



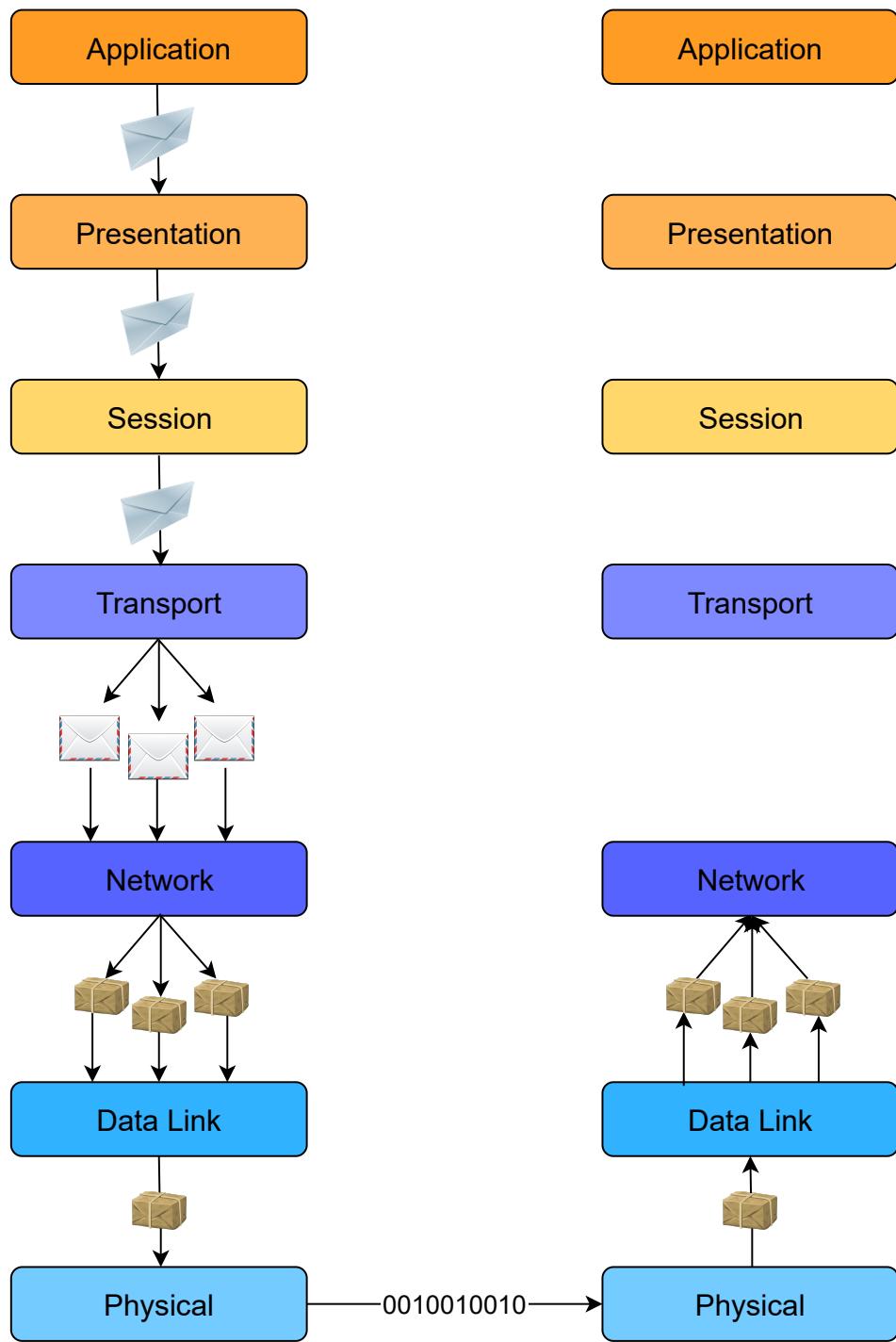
The physical layer then carries these in terms of bits represented physically in a medium

How data conceptually travels through the layers

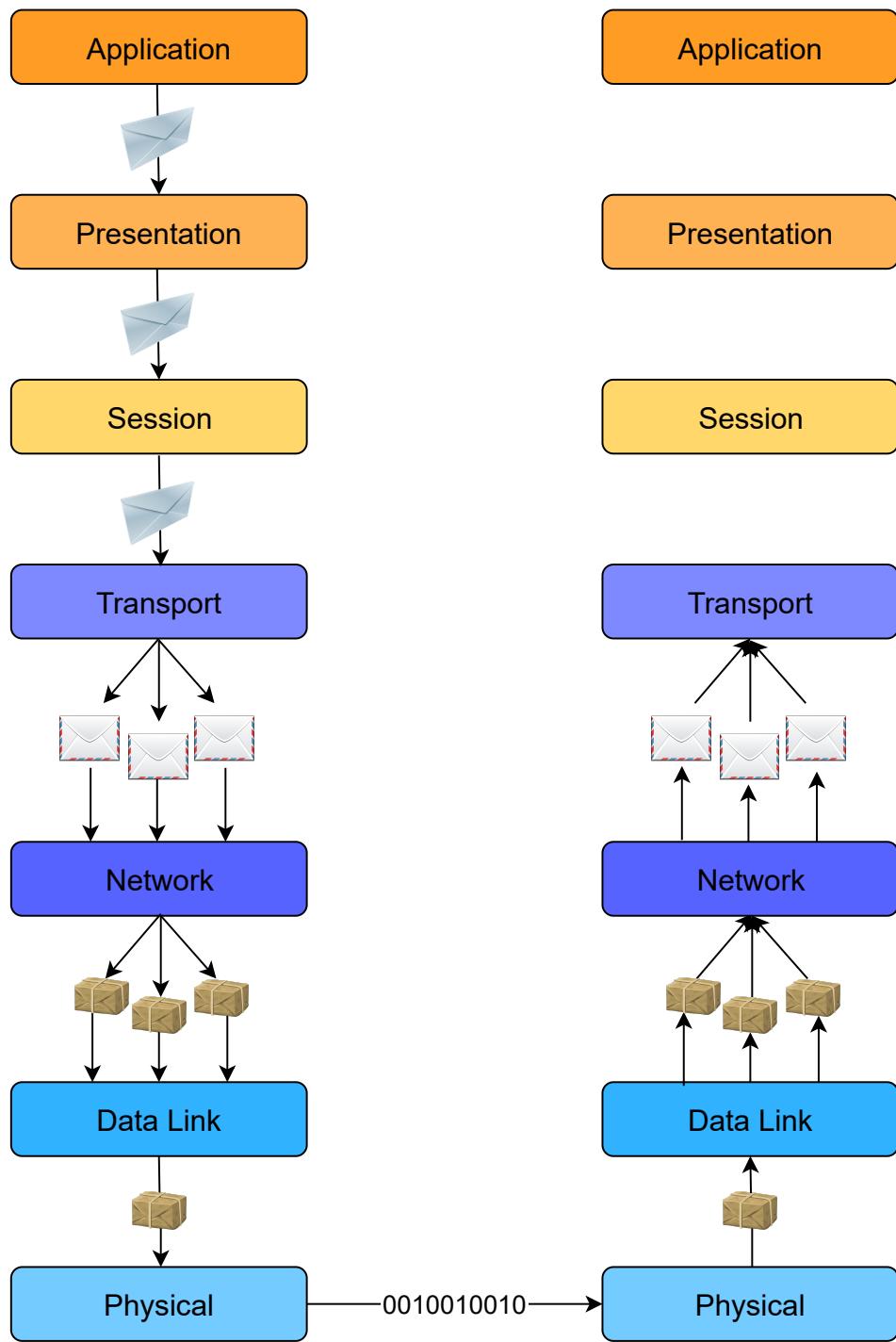


The reverse process happens from here

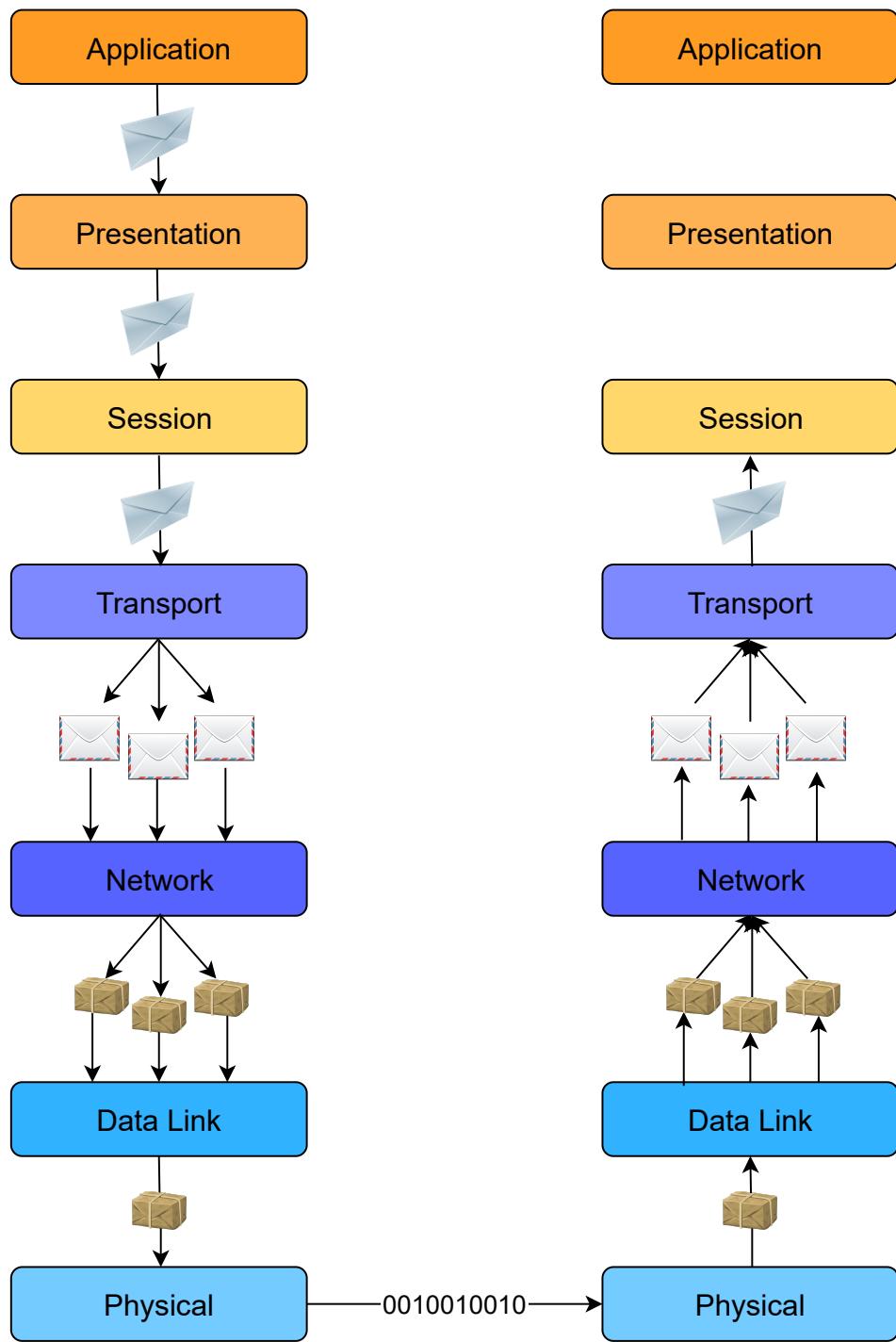
How data conceptually travels through the layers



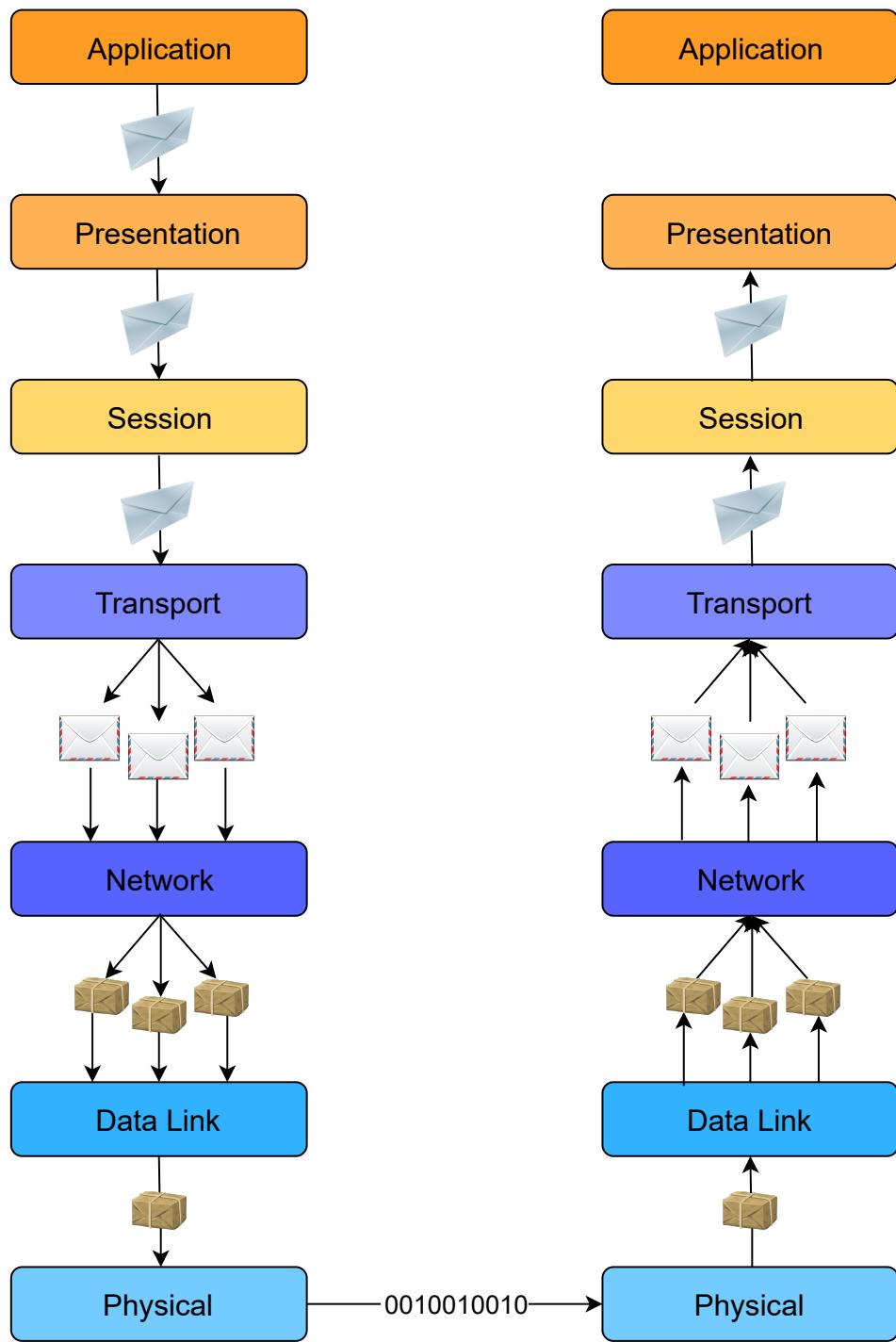
How data conceptually travels through the layers



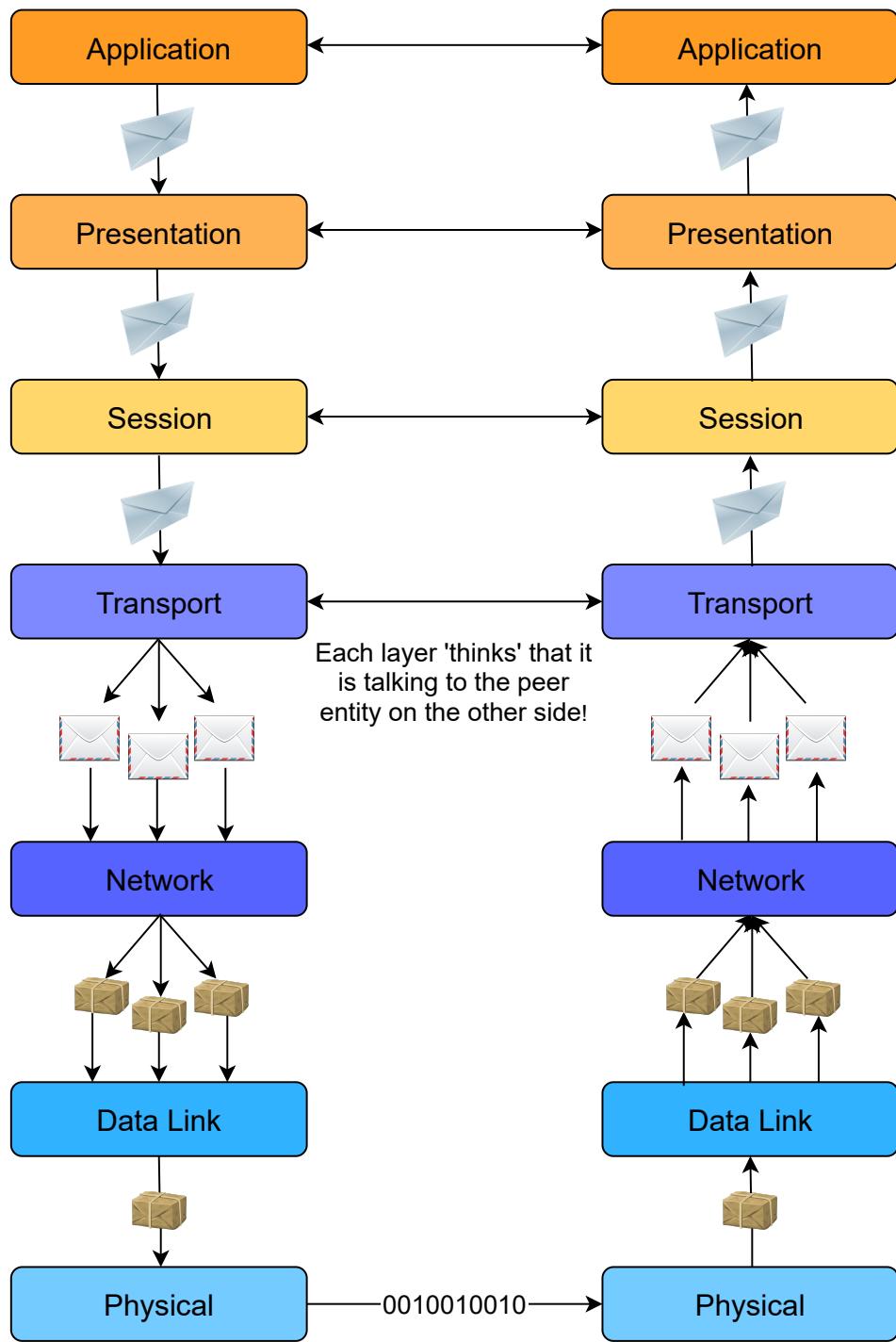
How data conceptually travels through the layers



How data conceptually travels through the layers



How data conceptually travels through the layers



How data conceptually travels through the layers

13 of 13



## Quick Quiz! #

1

How many layers does the OSI model have?

COMPLETED 0%

1 of 5



Conceptualizing networks into layers like this had a significant impact on networking education, which is why it's still taught as the primary model in a lot of courses. However, now that protocols have matured, a better way to teach networks is to take a more protocol-oriented approach. This is where the **TCP/IP model** comes in which is what we'll look at in the next lesson.

# The TCP/IP Model

Let's now have a look at the TCP/IP Model

## WE'LL COVER THE FOLLOWING



- Introduction
- The Layers of The TCP/IP Stack
- TCP/IP vs OSI
  - Key Differences
  - Differences in Layer Functionality
- There is No Unanimous Stack
- The End-To-End Argument in System Design
  - Packet Switched Core
- Quick Quiz!

## Introduction #

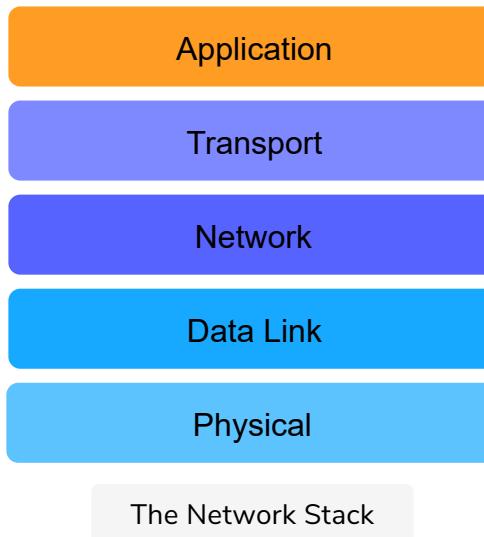
- The TCP/IP Model, also known as the **Internet protocol suite**, was **developed in 1989**.
- Its **development was funded by DARPA** (Advanced Research Projects Agency (ARPA) was renamed to the Defense Advanced Research Projects Agency (DARPA)!)
- Its technical specifications are detailed in [RFC 1122](#).
- This model is primarily based upon the most protocols of the Internet, namely the **Internet Protocol (IP)** and the **Transmission Control Protocol (TCP)**.
- The protocols in each layer are **clearly defined**, unlike in the OSI model. In this course, we'll largely adhere to the TCP/IP model and take a protocol oriented approach.

protocol-oriented approach.

## The Layers of The TCP/IP Stack #

The TCP/IP model splits up a communication system into **5 abstract layers**, stacked upon each other. Each layer performs a particular service and communicates with the layers above and below itself.

Here are the five layers of the TCP/IP model:



## TCP/IP vs OSI #

### Key Differences #

Here are some main differences between TCP/IP and OSI.

TCP/IP	OSI
<b>Is used practically</b>	The OSI model is conceptual and is <b>not practically used</b> for communication.
<b>Consists of five layers</b>	<b>Consists of seven layers</b>

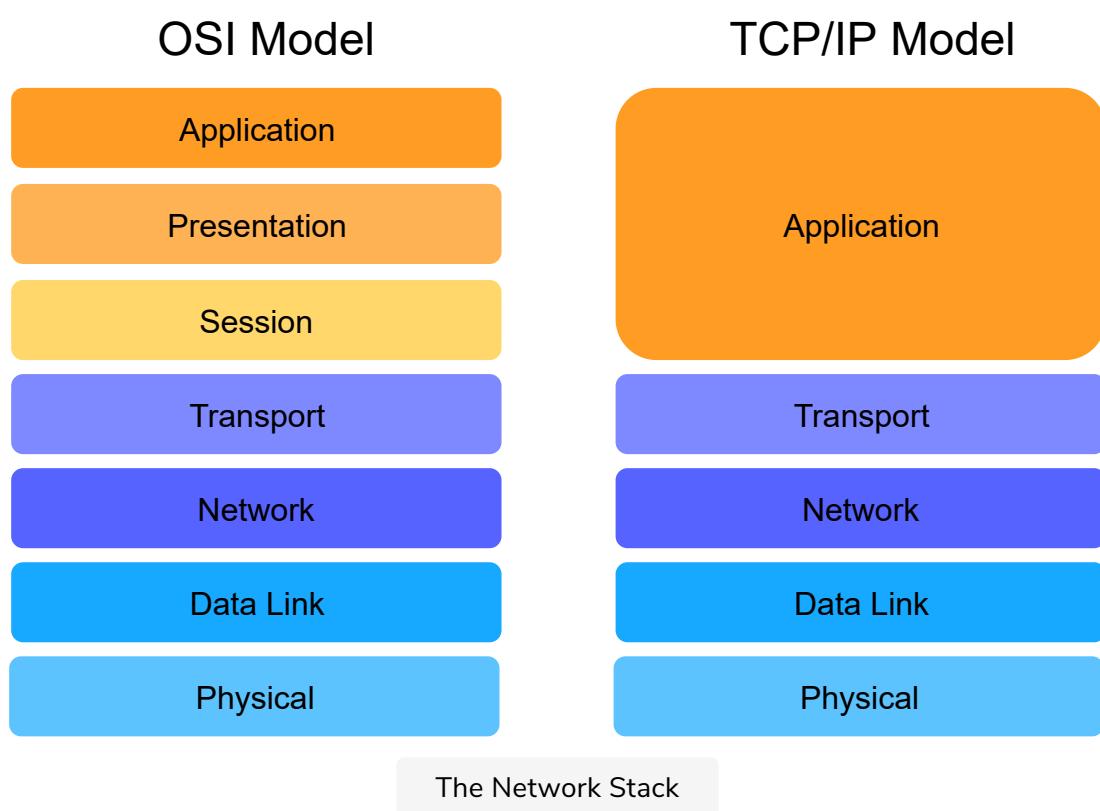
- Elaborating further on the first point, OSI is a **theoretical model** and works very well for teaching purposes, but it's far too complex for anyone to implement.

- TCP/IP, on the other hand, wasn't really a model. People just implemented it and got it to work. Then, people **reverse-engineered a reference model** out of it for theoretical and pedagogical purposes. So, something that “sounds like” a great idea might not be the eventual winner. It's de facto vs de jure standards.

## Differences in Layer Functionality #

The layers in the TCP/IP stack largely perform the same functions as their counterparts in the OSI model, except that the application layer in the TCP/IP model encompasses the functionalities of the top three layers of the OSI model.

Have a look at the following diagram for a more concrete view.



## There is No Unanimous Stack #

This is an example of where primary sources like RFCs clash with secondary sources like textbooks. There is, in fact, an entire [table on Wikipedia](#) dedicated to the prominent layer stacks! Regardless, we'll be sticking to the TCP/IP model described above.

# The End-To-End Argument in System Design #

The TCP/IP protocol suite is heavily influenced by the following design choice, also known as the **end-to-end argument**.

Implementing intelligence in the core was too expensive, therefore, intelligence was implemented at edge devices. So, the Internet's design was of **intelligent end devices** and a **dumb and fast core network**.

## Packet Switched Core #

Furthermore, the core was made **packet-switched**, which means that packets are routed **per-hop**, so they can circumvent failures because the requirement was for resilience.

With **circuit-switched networks**, however, torn connections have to be re-established, if there is still a path.

## Quick Quiz! #

1

The responsibilities of the presentation layer from the OSI model are handled by the \_\_\_\_\_ layer in the TCP/IP model.

COMPLETED 0%

1 of 2



Let's start on the application layer from the next chapter!

# What Is the Application Layer?

Here's an introduction to the application layer!

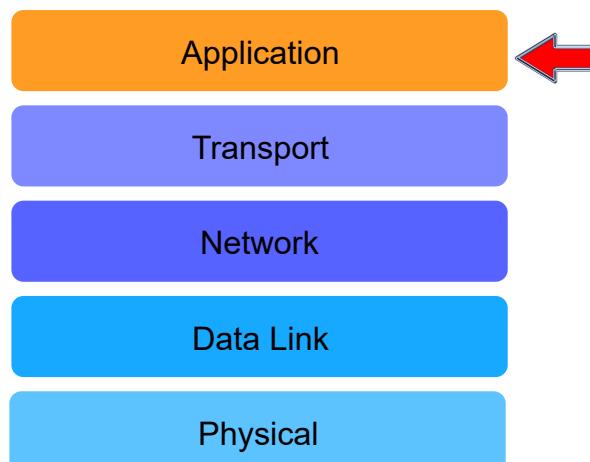
## WE'LL COVER THE FOLLOWING

- You Are Here!
- Key Responsibilities of the Application Layer
  - The Post Analogy
- Where It Exists
- Application Layer Protocols



## You Are Here! #

We're starting our study of the TCP/IP layers with the application layer.



## Key Responsibilities of the Application Layer #

The main job of the application layer is to enable end-users to access the Internet via a number of applications. This involves:

- **Writing data off to the network** in a format that is compliant with the protocol in use.
- **Reading data from the end-user.**

- **Providing useful applications** to end users.
- Some applications also ensure that the data from the end-user is in the correct format.
- Error handling and recovery is also done by some applications.

## The Post Analogy #

- Imagine you post a package across the world.
- Presumably, the post system would hand it off to an airplane or ship to transport it across the world.
- However, you would take it to the post office first to be shipped off.  
**Carrying the package to the post office** is what the application layer does in networks, except that **it carries messages to the transport layer!**

## Where It Exists #

The application layer resides entirely on end-systems. These end-systems can be any Internet-enabled device, be it a refrigerator or a tower PC.

## Application Layer Protocols #

Most would argue that **user applications are the true purpose of the Internet. If useful applications did not exist**, the Internet would not be what it is today.

- The development of the Internet in the last century started with text-based network apps such as **e-mail**.
- Then came **the app**: the **World Wide Web** which revolutionized everything.
- **Instant messaging** came at the end of the millennium, which has changed the way we communicate.
- Since then, we have come up with **voice over IP**, (WhatsApp calls), **video chat** (Skype), and **video streaming** (YouTube).
- **Social media** has also taken the world by storm resulting in complex large-scale distributed systems building on top of the protocol.

human social networks and businesses building on top of these websites.

All of these applications **run on application layer protocols**. Due to the presence of these standard protocols, client applications developed by various vendors can talk to server applications developed by others!

---

Let's uncover some of the underlying application layer protocols, in the next few lessons.

# Network Application Architectures

In this lesson, we'll learn about network application architectures.

## WE'LL COVER THE FOLLOWING ^

- Client-Server Architecture
  - Servers
  - Clients
  - An Example
- Data Centers
- Peer-to-Peer Architecture (P2P)
  - An Example
- Hybrid
- Quick Quiz!

Before we start off with application layer protocols, **it's important to understand how applications are structured across end systems**. This is called the network application's **architecture** and it's designed by application developers. The architecture lays out **how the application communicates and with what**.

Let's discuss some common application architectures.

## Client-Server Architecture #

In this architecture, a network application consists of two parts: **client-side** software and **server-side** software. These pieces of software are generally called **processes**, and they communicate with each other through **messages**.

## Servers #

The server process controls access to a centralized resource or service such as

The server process controls access to a centralized resource or service such as a website.

Servers have two important characteristics:

1. Generally, an attempt is made to keep servers online all the time, although 100% availability is impossible to achieve. Furthermore, servers set up as a hobby or as an experiment may not need to be kept online. Nevertheless, the client must be able to find the server online when needed, otherwise, communication wouldn't take place.
2. They have at least one reliable IP address with which they can be reached.

A good analogy is a 24/7 pizza delivery place. They are always open and have a phone number with which they can always be reached.

## Clients #

Client processes use the Internet to consume content and use the services. Client processes almost always initiate connections to servers, while server processes wait for requests from clients.

## An Example #

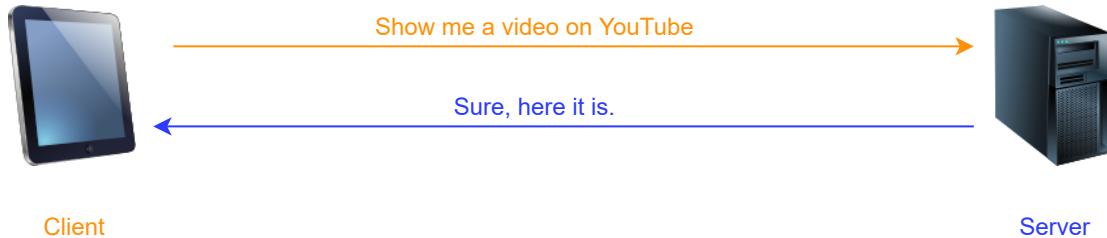
A good example of the client-server architecture is the **web**.

Take **Google** for instance. Google has several servers that control access to videos. So when a [google.com](http://google.com) is accessed, a client process (a browser) requests Google's homepage from one of Google's servers. That server was presumably online, got the request, and granted access to the page by sending it.

## Data Centers #

Now, you might have noticed that we mentioned that Google has *servers* and not one server. That's because, as mentioned previously, when client-server applications scale, one or even two servers can't handle the requests from a large number of clients. Additionally, servers may crash due to any reason and might stop working. Most applications have several servers in case one fails. Therefore, several machines host server processes (these machines are called servers too), and they reside in **data centers**.

Data centers are buildings that house servers. [Facebook](#), for example, has “nearly 15 million square feet of data center space completed or under construction, with several million more feet in the planning stages” as of 2018.

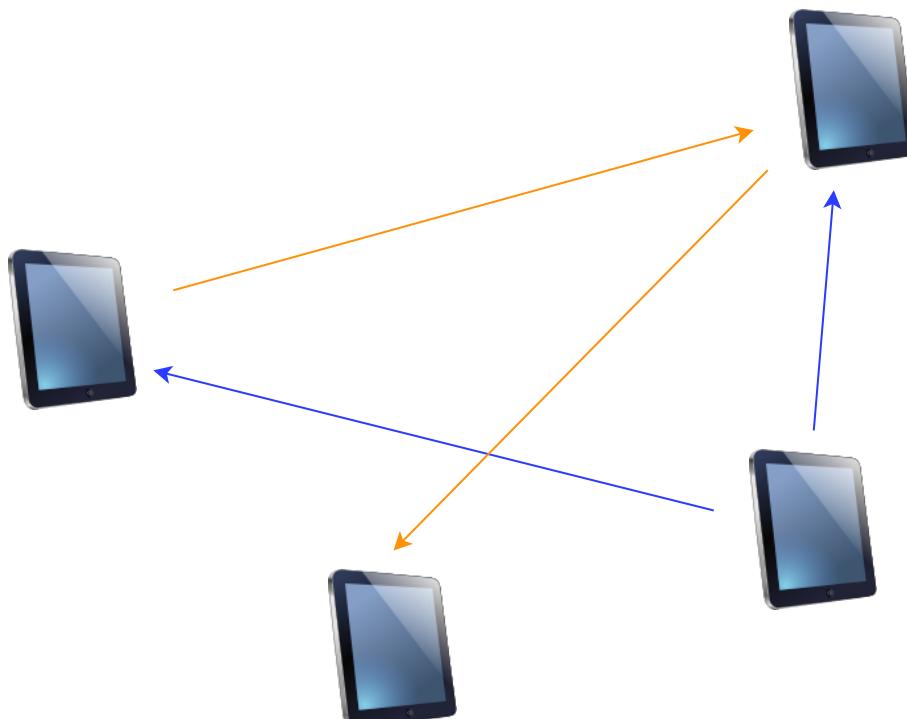


Clients obtain content (such as videos) and/or services (such as an online calculator) from servers.

## Peer-to-Peer Architecture (P2P) #

In this architecture, applications on end-systems called ‘peers’ communicate with each other. No dedicated server or large data center is involved. Peers mostly reside on PCs like laptops and desktops in homes, offices, and universities.

The key advantage of the P2P architecture is that it can scale rapidly – without the need of spending large amounts of money, time or effort.



Peers communicate with each other without a dedicated server

Regardless of P2P’s decentralized nature, each peer *can* be categorized as

servers or clients i.e., every machine is capable of being a client as well as a

server. Strictly speaking, the peer that initiates a connection is the client, and the other one is called the server.

## An Example #

A lot of popular applications today, like **BitTorrent**, are based on P2P architectures.

When a file is downloaded via BitTorrent, the downloading party accesses **bits** of the file on several other users' computers and puts them together on its end. No traditional 'server' is involved in this scenario.



**Note: P2P Is Not the Same as File Sharing!** Some early P2P applications were used for file sharing. For example, [Napster](#) and [Gnutella](#). Because of the massive impact of these P2P applications, a lot of people associate file sharing exclusively with P2P.

**File sharing** is a specific application. Whereas **P2P** is a design principle for distributed systems and an application architecture.

Also, file sharing is not the only application of P2P. Other examples include: streaming media, telephony, content distribution, routing, and volunteer computing.

## Hybrid #

The hybrid architecture involves server involvement to *some* degree. It's essentially a combination of the P2P and client-server architectures.

## Quick Quiz! #

1

What architecture is the web based on?

COMPLETED 0%

1 of 5



Let's look at how processes communicate across machines in the next lesson!

# P2P vs. Client-Server

Before we move on with the details of BitTorrent, it's useful to do a quantitative comparison of the hybrid architecture with the client-server architecture.

## WE'LL COVER THE FOLLOWING ^

- Quantitative Comparison of P2P with Client-Server
  - Client-Server
  - P2P
- Quick Quiz!

## Quantitative Comparison of P2P with Client-Server #

Let's calculate how long it will take to transmit a file from one server to a number of clients based on both the P2P and server-client architectures. The calculations will be performed based on the following givens.

- A **server** that can upload at a rate of  $up_s$  where  $up_s$  is the upload speed in bits/second.
- There are  $N$  **clients** all wanting to download the same file from the server. Client  $i$  can upload at a rate of  $up_i$  bits/second and download at a rate of  $dwn_i$  bits/second.
- The size of the file that all the peers want is  $S$ .

## Client-Server #

Let's start with the **client-server** architecture. The following can be observed.

- Since  $N$  clients each want a file of size  $S$ , the server will have to upload  $NS$  bits. The upload rate of the server is  $up_s$  so the server will take at least  $\frac{NS}{up_s}$  time to transmit the file to all  $N$  clients.
- The client with the lowest download rate ( $dwn_{min} = \min(dwn_i)$ ) will

take at least  $\frac{S}{dwn_{min}}$  time to download the full file.

So, in total the time taken to transmit the file will be the maximum of both of the times above, i.e.:

$$\max \left\{ \frac{NS}{up_s}, \frac{S}{dwn_{min}} \right\}$$

## P2P #

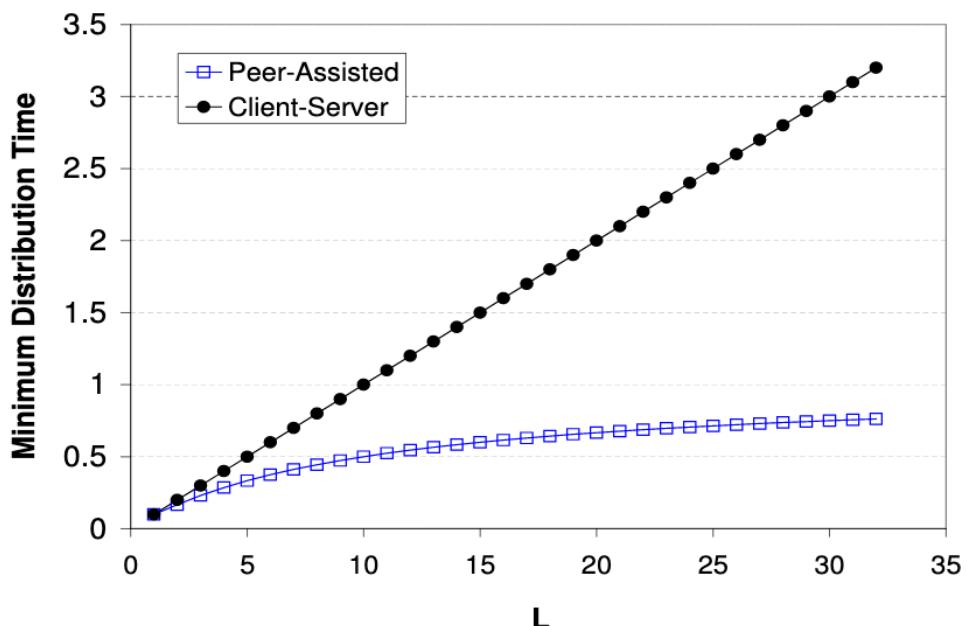
We can make the following observations:

- Initially, only the server has the file. It has to throw the file out into the network and to do that, it will take at least  $\frac{S}{up_s}$  time. While the file is being sent out into the network of peers, they start to distribute it amongst themselves.
- The peer with the lowest download rate ( $dwn_{min}$ ) will take at least  $\frac{S}{dwn_{min}}$  time to download the full file.
- The file cannot be transmitted faster than the total upload speed of the entire network: ( $up_{sum} = \{up_1 + up_2 + up_3 + \dots + up_N\}$ ). Since the file has to be distributed to all  $N$  peers,  $NS$  bits have to be transmitted, that will take  $\frac{NS}{up_{sum}}$  time.

Therefore, the time taken in total to distribute a file of size  $S$  to  $N$  peers is:

$$\max \left\{ \frac{S}{up_s}, \frac{S}{dwn_{min}}, \frac{NS}{up_{sum}} \right\}$$

Note that as the number of clients/peers,  $N$ , grows, the time taken by the client-server architecture also grows. Here is a graph of how the distribution time grows for each architecture as the number of clients/peers grow:



Graph of How p2p Scales vs Client-Server attributed to:

<https://pdfs.semanticscholar.org/3de3/1a9b45a3d071c638574117af8e046b578004.pdf>

**P2P networks are extremely mathematically scalable.** The resources of a P2P system grows with the number of peers in the system. Thus, applications with P2P architecture are self-scaling.

## Quick Quiz! #

1

In a client-server model, the rate at which a client can download a file is limited by the \_\_\_\_\_.

---

Let's now get into how processes communicate!

# How Processes Communicate

Let's have a quick look at the technical aspect of how applications communicate

WE'LL COVER THE FOLLOWING



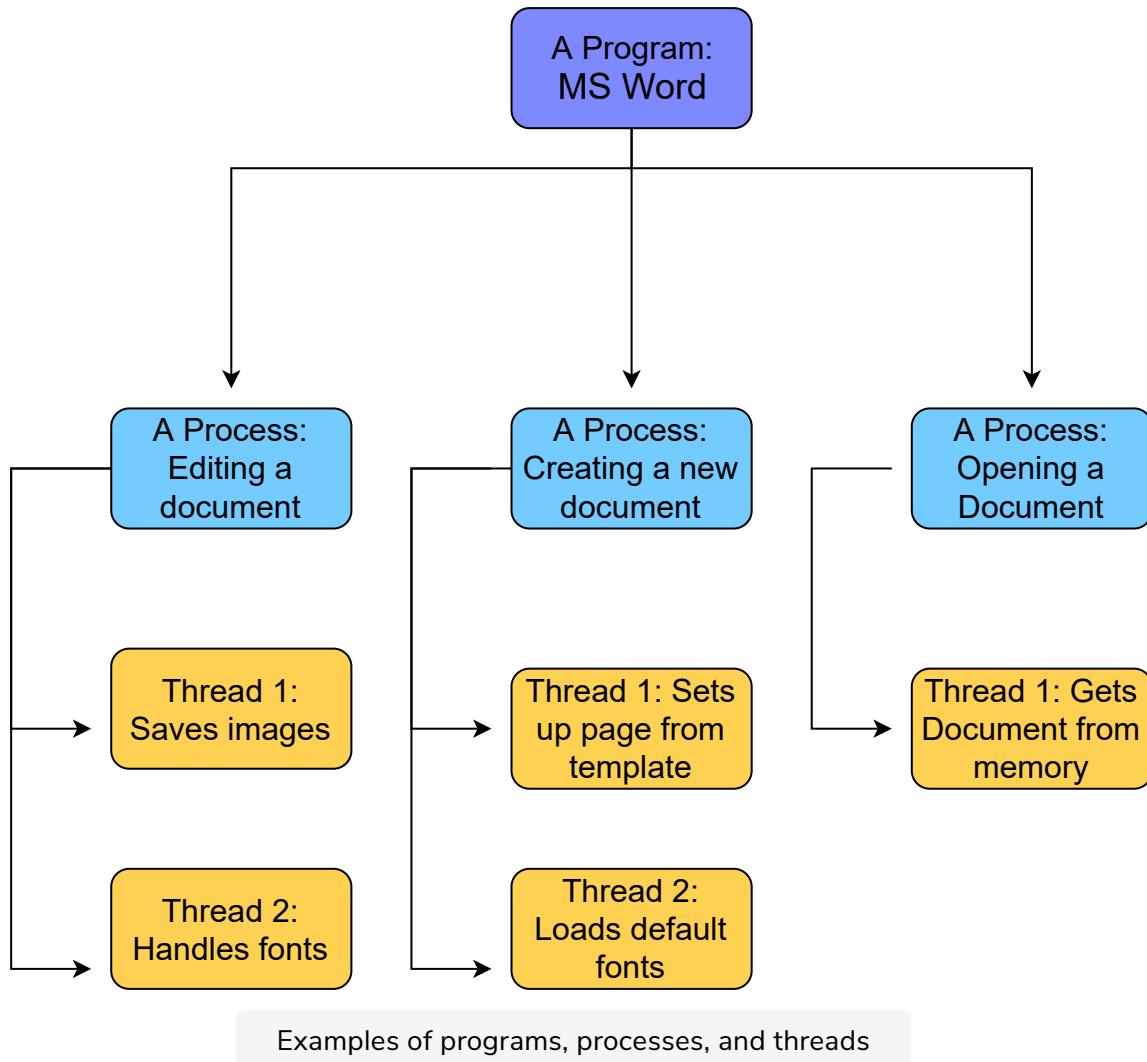
- Program vs. Process vs. Thread
- Sockets
- Addressing
- Quick Quiz!

## Program vs. Process vs. Thread #

We've loosely used the term 'process' pretty much interchangeably with the term 'application' in the last few chapters. Now, let's now get a finer definition.

- A **program** is simply an executable file. An application such as MS Word is one example.
- A **process** is any currently running instance of a program. So one program can have several copies of it running at once. One MS Word program can have multiple open windows.
- A **thread** is a lightweight process. One process can have multiple running threads. The difference between threads and processes is that threads do lightweight singular jobs.

Here's the MS Word example illustrated:



Processes that exist on the same machine can and do regularly communicate with each other following the rules of the machine's OS. However, we are more interested in how processes that run on *different* machines communicate.

## Sockets #

Processes on different machines send messages to each other through the computer network. The *interface* between a process and the computer network is called a **socket**. Note that sockets do not have anything to do with hardware – they are software interfaces.

Processes simply direct their messages to sockets and don't worry about it after that.

## Addressing #

Messages have to be addressed to a certain application on a certain end

system. How is it done with potentially millions of end systems and hundreds of applications on each of them?

Well, it's done via addressing constructs like **IP addresses and ports**. While both were touched upon [previously](#), we would like to reintroduce ports a bit more in-depth.

## Ports

Since every end-system may have a number of applications running, **ports** are used to address the packet to specific applications. As stated previously, some ports are reserved such as port 80 for HTTP and port 443 for HTTPS.



### An Analogy: Post

Continuing with our post analogy, you can think of an end-system like a large apartment complex. Each apartment in the complex is an application.

The mailing address of the complex is like the IP address of the end-system. All running applications share it, just like all apartments share the street address. Each application running on a host has a different port number, just like each apartment has a different apartment number.

### Ephemeral Ports

The port that an application will use is usually predefined by its application developers. So an application can have port 3000 reserved for it. But what if **several instances (processes) of an application are running at once**? How will the system address those processes?

Well, the answer lies in [Ephemeral Ports](#). Different port numbers are dynamically generated for each instance of an application. The port is freed once the application is done using it.

Furthermore, **server processes need to have well defined and fixed port numbers** so that clients can connect to them in a systematic and predictable way. However, **clients don't need to have reserved ports**. They can use ephemeral ports. Servers can also use ephemeral ports **in addition** to the reserved ones. For instance, a client makes the initial connection to the server on a well-known port and the rest of the communication is carried out by connecting to an ephemeral port on the server.

## Quick Quiz! #

1

A process is a running instance of a program

COMPLETED 0%

1 of 3



Now that we are familiar with some basic application layer terms, in the next lesson, let's get into the finer details of some key application layer protocols!

# HTTP: The Basics

Welcome to the core of this course! We are finally getting started with protocols, the first of which is HTTP.

## WE'LL COVER THE FOLLOWING



- Introduction
- Objects
- The Anatomy of a URL
- HTTP
- HTTP Requires Lower Layer Reliability
- Types of HTTP Connections
- Non-persistent HTTP
- Persistent HTTP
- Quick Quiz!

## Introduction #

The Internet was an obscure set of methods for file transfer and email used by academics and researchers. The World Wide Web was invented to allow the European research organization CERN to present documents linked by hypertexts. All of that changed though when it caught the public's eye and popularized the Internet. The web was different from other services such as cable television, because it served content based on demand. People could watch what they wanted. **HTTP** or **HyperText Transfer Protocol** is the protocol at the core of the web.

## Objects #

- Web pages are objects that consist of other **objects**.
- An **object** is simply a file like an HTML file, PNG file, MP3 file, etc.
- Each object has a URL

- The **base object** of a web page **is often an HTML file** that has **references to other objects** by making requests for them via their URL.



**Note:** HTML or HyperText Markup Language is the standard markup language to build webpages.

## The Anatomy of a URL #

A **URL**, or **Universal Resource Locator**, is used to locate files that exist on servers. URLs consist of the following parts:

- **Protocol** in use
- The **hostname** of the server
- The **location of the file**
- **Arguments** to the file

http://www.educative.io/allourses/course.php?auth=44&user=5

1 of 5

http://www.educative.io/allourses/course.php?auth=44&user=5  
Protocol

2 of 5

http://www.educative.io/allourses/course.php?auth=44&user=5  
Protocol  
Hostname of server

3 of 5

http://www.educative.io/allourses/course.php?auth=44&user=5  
Protocol  
Hostname of server  
path to resource

4 of 5

Hostname of server  
Protocol

http://www.educative.io/

path to resource

Arguments to course.php

5 of 5

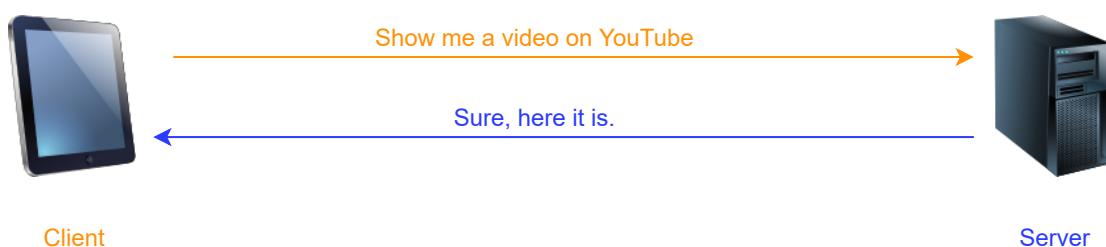


## HTTP #

Let's get back into **HTTP**. It's a client-server protocol that specifies how Web clients request Web pages from Web servers and how Web servers send them.

Remember the following diagram from the lesson on [Network Application Architectures](#)? Well, it was actually outlining HTTP in general.

- The **orange** arrow represents an **HTTP request**
- The **blue** arrow represents an **HTTP response**



Clients obtain content (such as videos) and/or services (such as an online calculator) from servers.

The first message is called an **HTTP request** and the second one an **HTTP response**. There's a whole class of protocols that are considered **request-response protocols**. HTTP is one of them. We will look in more detail at the HTTP request message in the next lesson and response after that!

Note that HTTP is a **stateless protocol**: servers do not store any information about clients by default. So if a client requests the same object multiple times in a row, the server would send it and would not know that the same client is requesting the same object repeatedly.

## HTTP Requires Lower Layer Reliability #

- Application layer protocols rely on underlying transport layer protocols called **UDP** (User Datagram Protocol) and **TCP** (Transmission Control Protocol).
- For now, all you need to know is that **TCP ensures that messages are always delivered**. Messages get delivered in the order that they are sent.
- **UDP does not ensure that messages get delivered**. This means that some messages may get dropped and so never be received.
- **HTTP uses TCP** as its underlying transport protocol so that messages are guaranteed to get delivered in order. This allows the application to function without having to build any extra reliability as it would've had to with UDP.

This sort of reliance on other layers for certain jobs is one of the key advantages of a layered architecture!

- **TCP is connection-oriented**, meaning a connection has to be initiated with servers using a series of starting messages.
- Once the connection has been made, the client exchanges messages with the server until the connection is officially closed by sending a few ending messages.

## Types of HTTP Connections #

There are two kinds of HTTP connections:

- **Non-persistent HTTP connections**
- **Persistent HTTP connections**

These two kinds of HTTP connections use TCP differently. Let's discuss the key advantages and disadvantages of each.

## Non-persistent HTTP #

**Non-persistent HTTP** connections use **one TCP connection per request**. Assume a client requests the base HTML file of a web page. Here is what happens:

1. The client initiates a TCP connection with a server

2. The client sends an HTTP request to the server

3. The server retrieves the requested object from its storage and sends it

4. The client receives the object which in this case is an HTML file. If that file has references to more objects, steps 1-4 are repeated for each of those

5. The server closes the TCP connection

For each HTTP request, more requests tend to follow, as well to fetch images, javascript files, CSS files, and other objects.

The underlying TCP connection requires three TCP messages are sent between the client and server. Similarly, when the connection is closed, three TCP messages are sent back and forth between the client and server.

## Persistent HTTP #

An HTTP session typically involves multiple HTTP request-response pairs, for which separate TCP connections are established and then torn down between the same client and server. This is inefficient. Later on, **Persistent HTTP** was developed, which used a single client-server TCP connection for all the HTTP request-responses for a session.

Typically, if there have been no requests for a while, the server closes the connection. The duration of time before the server closes the connection is configurable.

## Quick Quiz! #

1

What does HTTP stand for?

COMPLETED 0%

1 of 8



---

In the next lesson, we'll discuss HTTP request messages in more detail.

# HTTP: Request Messages

HTTP request messages are a pivotal part of the protocol. Let's have a close look at them!

## WE'LL COVER THE FOLLOWING



- Introduction
- HTTP Request Messages
- The Anatomy of an HTTP Request Line
  - HTTP Methods
  - URL
  - Version
  - The Anatomy of HTTP Header Lines

## Introduction #

There are two types of HTTP messages as discussed previously:

- **HTTP request messages**
- **HTTP response messages**

We'll study request messages in this one.

## HTTP Request Messages #

Let's look at request messages first. Here is an example of a typical HTTP message:

```
GET /path/to/file/index.html HTTP/1.1
Host: www.educative.io
Connection: close
User-agent: Mozilla/5.0
Accept-language: fr
Accept: text/html
```



It should be noted that,

- HTTP messages are in **plain ASCII text**
- **Each line** of the message **ends with** two control characters: a **carriage return and a line feed**: `\r\n`.
  - The last line of the message also ends with a carriage return and a line feed!
- This particular message has 6 lines, but HTTP messages can have **one or as many lines as needed**.
- The first line is called the **request line** while the rest are called **header lines**.

## The Anatomy of an HTTP Request Line #

The HTTP request line is followed by an HTTP header. We'll look at the request line first. The request line consists of three parts:

- **Method**
- **URL**
- **Version**

Let's discuss each.

```
GET path/to/file HTTP 1.1
```

HTTP request line

1 of 4

Request Method

```
GET path/to/file HTTP 1.1
```

HTTP request method

2 of 4

Request Method

GET path/to/file HTTP 1.1

URL

URL to resource

3 of 4

Request Method

GET path/to/file HTTP 1.1

URL

HTTP Version

4 of 4

-

[ ]

## HTTP Methods #

HTTP methods tell the server what to do. There are a lot of HTTP methods but we'll study the most common ones: **GET**, **POST**, **HEAD**, **PUT**, or **DELETE**.

- **GET** is the most common and **requests data**.
  - This method is generally used when the client is not sure where the new data would reside. The server responds with the location of the object.
  - The data posted can be a message for a bulletin board, newsgroup, mailing list, a command, a web form, or an item to add to a database.
  - The POST method technically requests a page but that depends on what was entered.
- **HEAD** is similar to the **GET** method except that **the resource requested does not get sent in response. Only the HTTP headers are sent instead.**
  - This is useful for quickly retrieving meta-information written in

response headers, without having to transport the entire content. In other words, it's useful to check with minimal traffic if a certain object still exists. This includes its meta-data, like the last modified date. The latter can be useful for caching.

- This is also useful for testing and debugging.

- **PUT** **uploads an enclosed entity under a supplied URI.** In other words, it **puts** data at a specific location. If the URI refers to an already existing resource, it's replaced with the new one. If the URI does not point to an existing resource, then the server creates the resource with that URI.
- **DELETE** **deletes an object** at a given URL.

Note that while most forms are sent from the POST method, the GET method is also used sometimes with the entries of the form appended to the URL, as in arguments like this:

```
http://www.website.com/form.php/?Name=PostMan?Age=45?Interest=Post
```

forms with GET requests

However, sending forms with a POST request is generally better because:

1. The amount of data that can be sent via a post request is unlimited.
2. The form's fields are not shown in the URL.



### Note: URIs & URLs

- **Uniform Resource Locators (URLs)** URLs are used to identify an object over the web. [RFC 2396](#). A URL has the following format:  
`protocol://hostname:port/path-and-file-name`
- **Uniform Resource Identifiers (URIs)** can be more specific than URLs in such a way that they can locate fragments within objects too [RFC 3986](#). A URI has the following format:  
`http://host:port/path?request-parameters#nameAnchor`. For instance,  
<https://www.educative.io/collection/page/10370001/6105520698032128/6460983855808512/#http-methods> is a URI.

## URL #

This is the location that any HTTP method is referring to.

## Version #

The HTTP version is also specified in the request line. The latest version of HTTP is [HTTP/2](#).

## The Anatomy of HTTP Header Lines #

The HTTP request line is followed by an HTTP header. A lot of HTTP headers exist! We'll be covering the most important ones in this lesson. However, if you're interested, you can [read further](#) about all of them.

- The first header line specifies the `Host` that the request is for.
- The second one defines the type of HTTP `Connection`. It's Non-persistent in the case of the following drawing as the connection is specified to be closed.
- The `user-agent` line specifies the client. This is useful when the server has different web pages that exist for different devices and browsers.
- The `Accept-language` header specifies the language that is preferred. The server checks if a web page in that language exists and sends it if it does, otherwise the server sends the default page.
- The `Accept` header defines the sort of response to accept. It can be anything like HTML files, images, and audio/video.

```
GET/path/to/file/index.html HTTP/1.1
Host: www.educative.io
Connection: close
User-agent:Mozilla/5.0
Accept-language: fr
Accept: text/html
```

Request  
line

```
GET/path/to/file/index.html HTTP/1.1
Host: www.educative.io
Connection: close
User-agent:Mozilla/5.0
Accept-language: fr
Accept: text/html
```

2 of 7

**host** specifies  
the domain that  
the request is for,  
This one was for  
educative.io

```
GET/path/to/file/index.html HTTP/1.1
Host: www.educative.io
Connection: close
User-agent:Mozilla/5.0
Accept-language: fr
Accept: text/html
```

3 of 7

```
GET/path/to/file/index.html HTTP/1.1
Host: www.educative.io
Connection: close
User-agent:Mozilla/5.0
Accept-language: fr
Accept: text/html
```

**Connection** indicates  
if the connection is to  
be persistent or not.  
In this case, it is not  
persistent and will be  
closed after every  
message.

4 of 7

```
GET/path/to/file/index.html HTTP/1.1
Host: www.educative.io
Connection: close
User-agent:Mozilla/5.0
Accept-language: fr
Accept: text/html
```

The **user-agent** line specifies the client. In this case, it specifies the user's browser

5 of 7

```
GET/path/to/file/index.html HTTP/1.1
Host: www.educative.io
Connection: close
User-agent:Mozilla/5.0
Accept-language: fr
Accept: text/html
```

The **Accept-language** header specifies the language that is preferred. In this case, French.

6 of 7

```
GET/path/to/file/index.html HTTP/1.1
Host: www.educative.io
Connection: close
User-agent:Mozilla/5.0
Accept-language: fr
Accept: text/html
```

The **Accept** header defines the sort of response to accept. In this case, it will accept text or HTML

7 of 7



In the next lesson, we'll conduct an exercise to look at real HTTP request messages!



# Exercise: Looking at a Real HTTP Request

In this lesson, you will be looking at real HTTP messages right from your browser!

## WE'LL COVER THE FOLLOWING



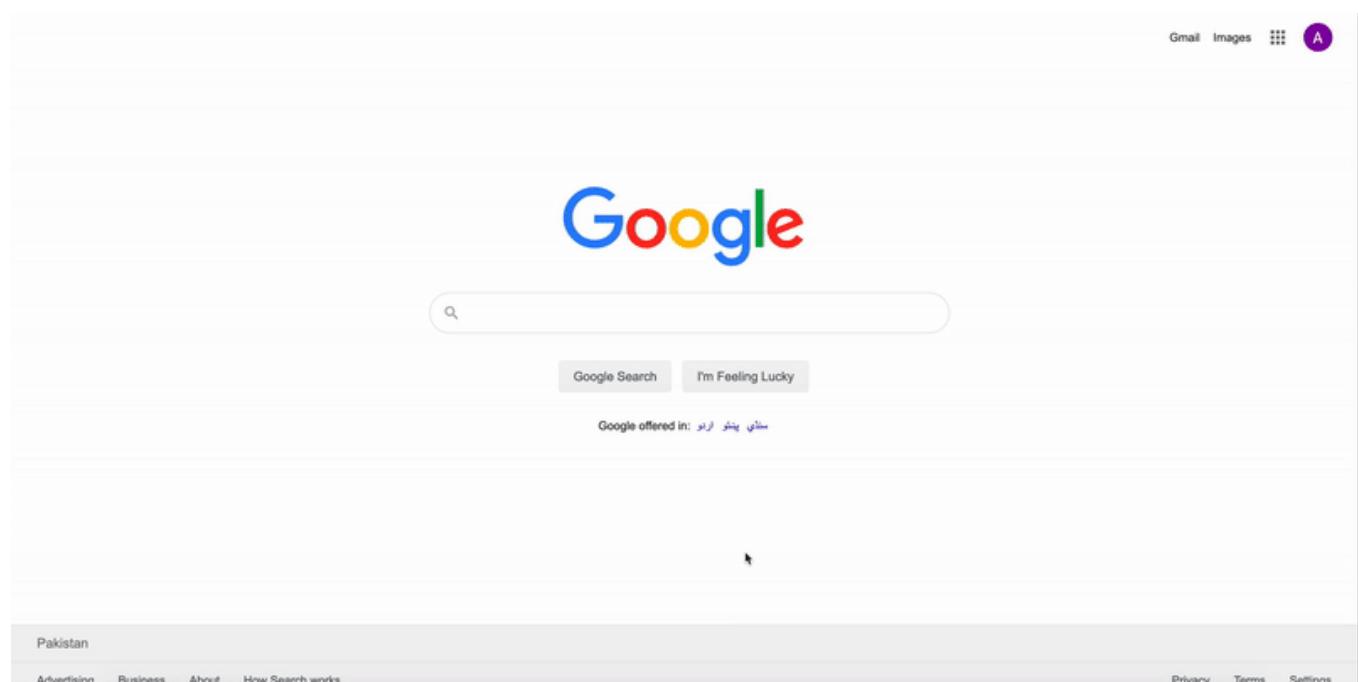
- Open up the Developer Tools on Your Browser
- Go to the Network Tab
- Click on Any Entry
- An Example of an Entry

## Open up the Developer Tools on Your Browser #

Have a look at this GIF. We were on **Firefox** here.

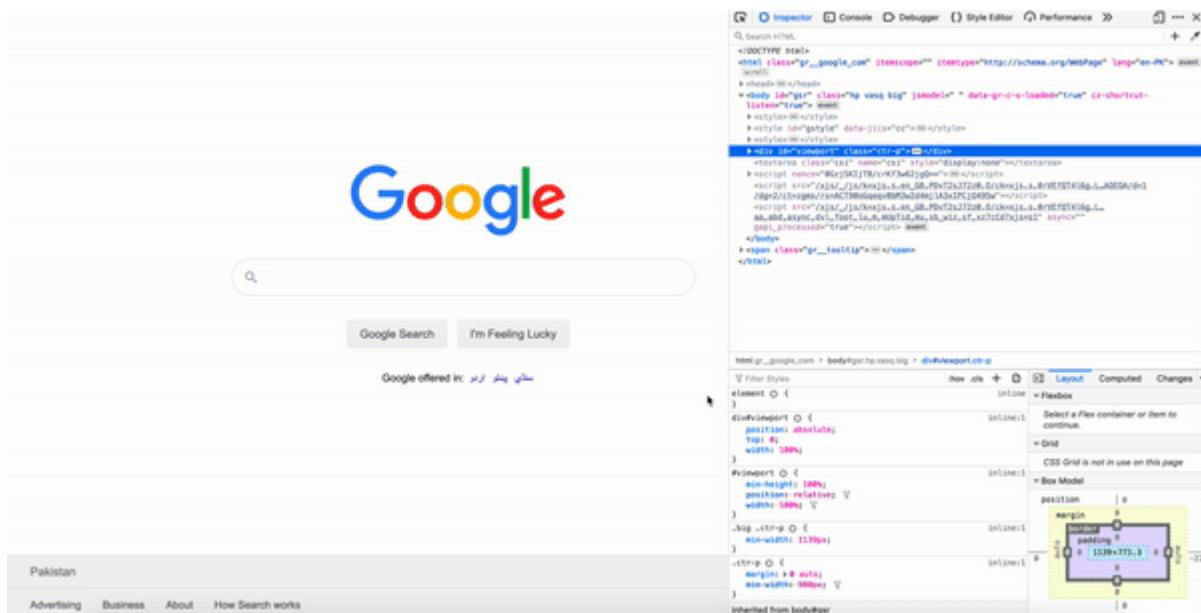
1. Navigate to any website. We picked [google.com](https://www.google.com).
2. Right-click anywhere.
3. Click on ‘inspector tools’ in the drop-down menu.

The process should be similar for other browsers.



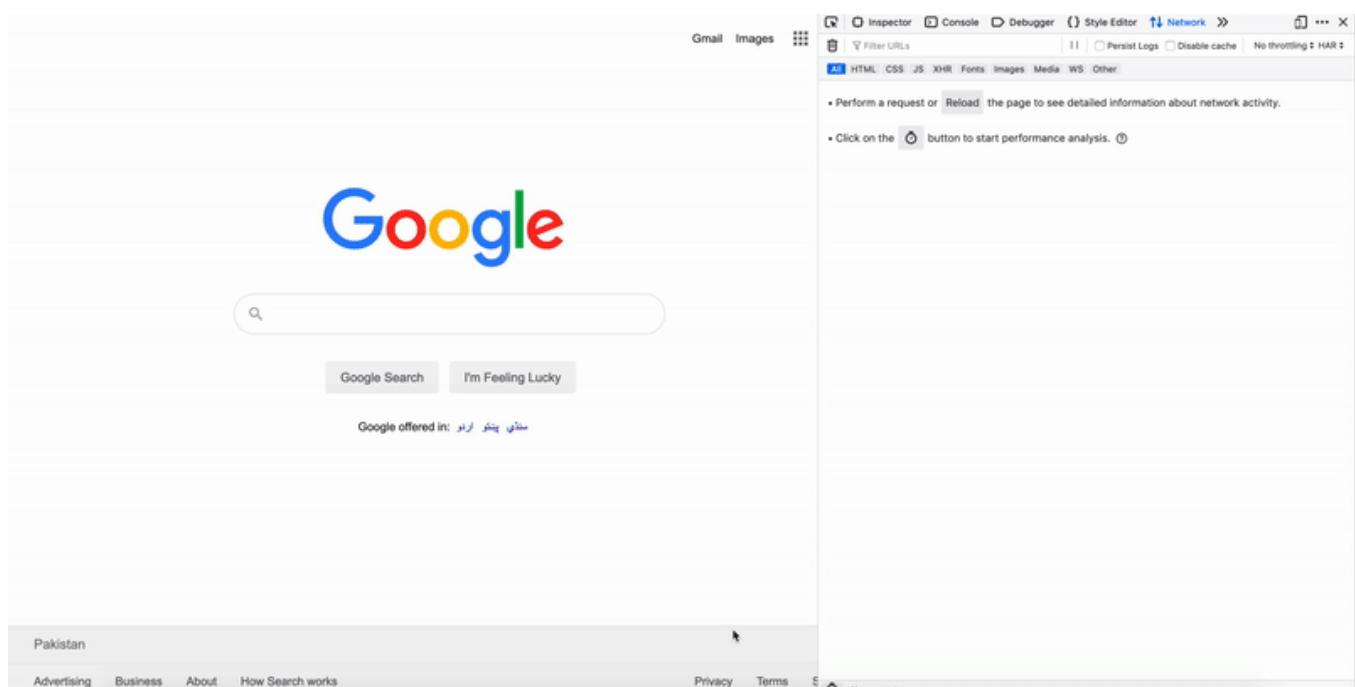
# Go to the Network Tab #

1. The network tab should be one of the tabs on the top-bar (or sidebar in some browsers).
2. Find it and click on it.
3. There may be a chance that your browser hasn't logged any network calls. In that case, just reload the page.



# Click on Any Entry #

1. You'll see a bunch of calls. Click on any one of them.
2. You'll see details about the HTTP message including the request headers, the kind of request, and the headers. We encourage you to spend some time exploring this.



# An Example of an Entry #

Headers Cookies Params Response Timings Stack Trace Security

Request URL: <https://www.google.com/>

Request method: GET

Remote address: 172.217.19.164:443 IP address of remote server. Notice the port number is 443 which is reserved for HTTPS

Status code: **200** OK ? Status code: 200 ok means the resource was found.

Version: HTTP/2.0

Edit and Resend

Filter headers

▼ Response headers (645 B) Raw headers

```
HTTP/2.0 200 OK
date: Mon, 23 Sep 2019 05:15:01 GMT
expires: -1
cache-control: private, max-age=0
content-type: text/html; charset=UTF-8
strict-transport-security: max-age=31536000
content-encoding: br
server: gws
content-length: 60432
x-xss-protection: 0
x-frame-options: SAMEORIGIN
set-cookie: 1P_JAR=2019-09-23-05; expires=Wed, 23-Oct-2019 05:15:01 GMT; path=/;
set-cookie: SIDCC=AN0-TYtW6jZBB-jHA24xjV8ayjZVFUZHeV1_2hu58SsLTaC6yrAkkCAF_bLA2e
alt-svc: quic=":443"; ma=2592000; v="46,43,39"
X-Firefox-Spdy: h2
```

▼ Request headers (0.985 KB) The response headers. Try tallying each with what we learned in the last lesson. Raw headers

```
Host: www.google.com
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.14; rv:68.0) Gecko/20100101
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
DNT: 1
Connection: keep-alive
Cookie: NID=somehiddenstring
Upgrade-Insecure-Requests: 1
Cache-Control: max-age=0
TE: Trailers
```

Now that we have a clear idea of what HTTP request messages look like, let's study the response messages in the next lesson.

# HTTP: Response Messages

Let's look at what HTTP response messages look like!

## WE'LL COVER THE FOLLOWING



- Introduction
- Status Line
  - Status Code
- Header Lines
  - How HTTP Headers Are Chosen
- Quick Quiz on HTTP!

## Introduction #

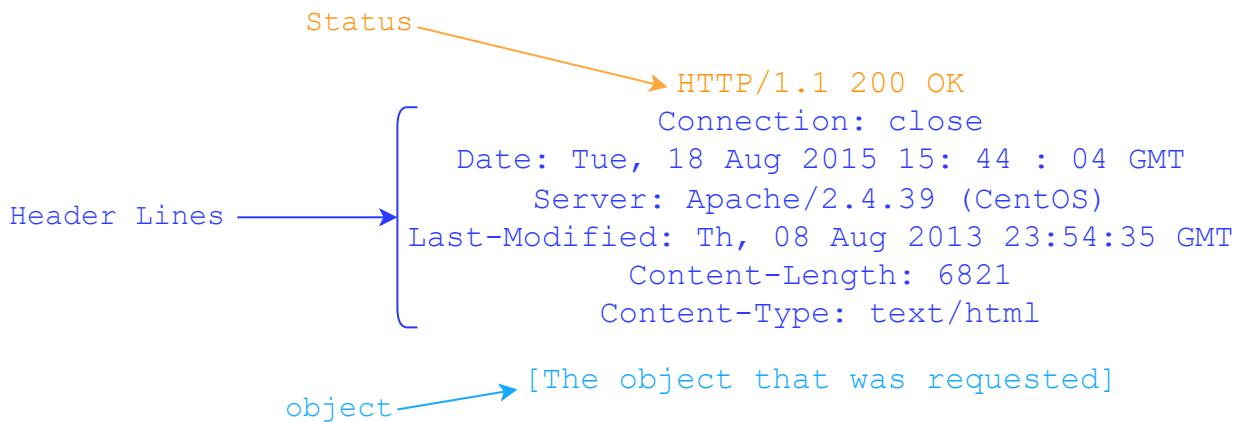
Let's start with a typical example of an HTTP response message:

```
HTTP/1.1 200 OK
Connection: close
Date: Tue, 18 Aug 2015 15:44:04 GMT
Server: Apache/2.2.3 (CentOS)
Last-Modified: Tue, 18 Aug 2015 15:11:03 GMT
Content-Length: 6821
Content-Type: text/html

[The object that was requested]
```



It has 3 parts: an initial **status line**, some **header lines** and an **entity body**.



An HTTP response message



**Note:** HTTP response messages don't have the URL or the method fields. Those are strictly for request messages.

## Status Line #

- HTTP response status lines start with the **HTTP version**.

## Status Code #

- The **status code** comes next which tells the client if the request succeeded or failed.
- There are a lot of status codes:
  - 1xx codes fall in the informational category
  - 2xx codes fall in the success category
  - 3xx codes are for redirection
  - 4xx is client error
  - 5xx is server error

Here is a list of some common status codes and their meanings:

- **200 OK** : the request was successful, and the result is appended with the response message.
- **404 File Not Found** : the requested object doesn't exist on the server.

- **400 Bad Request**: generic error code that indicates that the request was in a format that the server could not comprehend.
- **500 HTTP Internal Server Error**: the request could not be completed because the server encountered some unexpected error.
- **505 HTTP Version Not Supported**: the requested HTTP version is not supported by the server.

Have a look at pages 39 and 40 of [RFC 2616](#) for a comprehensive list.

## Header Lines #

Let's study the header lines.

- **Connection type**. In this case, indicates that the server will **close** the TCP connection after it sends the response.
- **Date**. The date at which the response was generated.
- **Server**. Gives server software specification of the server that generated the message. Apache in this case.
- **Last-Modified**. The date on which the object being sent was last modified.
- **Content-Length**. The length of the object being sent in 8-bit bytes.
- **Content-Type**. The type of content. The type of the file is not determined by the file extension of the object, but by this header.

The **response** body contains the file requested.

## How HTTP Headers Are Chosen #

Lastly, you must be wondering how browsers decide which HTTP headers to include in requests and how servers decide which headers to return in the response. That **depends on a complex mix of factors such as the browser, the user configurations and products**.

## Quick Quiz on HTTP! #

1

## What is HTTP?

COMPLETED 0%

1 of 5



In the next lesson, we'll look at real HTTP responses via a simple command-line tool!

# Exercise: Looking at a Real HTTP Response

In this lesson, we'll look at and study real HTTP responses via cURL.

## WE'LL COVER THE FOLLOWING ^

- `cURL`
- Explanation
- Sample Output
- Quick Quiz!

## cURL #

Run the following command to look at a real HTTP response.

```
curl http://example.org --head -silent
```



**cURL** (pronounced ‘curl’) is a command-line tool that transfers data to or from a server. The transfer can be based on a vast set of protocols, so we’ll be seeing cURL a lot. It’s perfect for our purposes because it doesn’t require live user interaction. cURL stands for “Client URL.” You can read more about cURL on its [manpage](#).

## Explanation #

Let’s learn about all of its components.

- `curl` is the name of the command that tells the terminal that this is a curl command.
- The `--head` flag or `-I` in short, tells cURL to send an HTTP request with the `head` method. In other words, the entity-body of the HTTP message is

the `head` method. In other words, the entity body of the HTTP message is not fetched.

- The `-silent` flag tells cURL to not display the progress meter. The progress meter is interpreted as an error on our platform, which is why we decided to remove it. The command is perfectly fine without this flag otherwise.

We encourage you to explore the cURL command. You can find a list of all the flags under the ‘options’ heading on cURL’s [manpage](#). Try different websites and different flags and see what you get!

## Sample Output #

The output of this command is an HTTP response such as the following. Notice the HTTP response code and the headers.

```
HTTP/1.1 200 OK
Content-Encoding: gzip
Accept-Ranges: bytes
Cache-Control: max-age=604800
Content-Type: text/html; charset=UTF-8
Date: Mon, 23 Sep 2019 06:48:39 GMT
Etag: "1541025663"
Expires: Mon, 30 Sep 2019 06:48:39 GMT
Last-Modified: Fri, 09 Aug 2013 23:54:35 GMT
Server: ECS (ord/5726)
X-Cache: HIT
Content-Length: 606
```

## Quick Quiz! #

1

What is cURL?

COMPLETED 0%

1 of 2



---

In the next lesson, let's have a look at one of the key concepts of computer networks – cookies!

# Cookies

Let's discuss another key concept of computer networking, cookies!

## WE'LL COVER THE FOLLOWING

- **Set-cookie** Header
  - Example
  - Blocking Third-Party Cookies Is Not Enough!
- Quick Quiz!

## Introduction

You might have heard of the term ‘cookie’ used a lot in the context of computer networks and privacy. Let’s have a closer look at what they are.

HTTP is a stateless protocol, but we often see websites where session state is needed. For instance, imagine you are browsing for products on an e-commerce website. How does the server know if you are logged in or not, or if the protocol is stateless? How does the server know what’s in your shopping cart when checking out if the protocol is stateless? Cookies allow the server to keep track of this sort of information.



## How Cookies Work

- Cookies are **unique string identifiers** that can be stored on the client’s browser.
- These identifiers are **set by the server through HTTP headers** when the

client first navigates to the website.

- After the cookie is set, it's sent along with subsequent HTTP requests to the same server. This **allows the server to know who is contacting it** and hence serve content accordingly.

So the HTTP request, the HTTP response, the cookie file on the client's browser, and a database of cookie-user values on the server's end are all involved in the process of setting and using cookies.

## Set-cookie Header #

Let's look at how cookies work in a bit more detail. When a server wants to set a cookie on the client-side, it includes the header `Set-cookie: value` in the HTTP response. This value is then **appended to a special cookie file stored on your browser**. The cookie file contains:

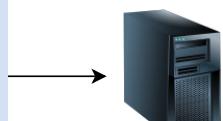
- The website's domain
- The string value of the cookie
- The date that the cookie expires (yes, much like actual cookies, they do expire)

Have a look at the following slides to see how cookies work in practice.

**HTTP Request**  
First ever request from a client to educative.io does not contain any cookies.



```
GET /index.html HTTP 1.1
Accept: image/gif, image/x-bitmap, image/jpeg, */
        Accept-Language: en
        Connection: Keep-Alive
        User-Agent: Mozilla/1.22
        Host: educative.io
Referrer: http://www.google.com?q=best+courses+ever
```



## HTTP Response

can include a session identifier, i.e., a cookie via the `set-cookie` header that tracks a user once they have authenticated



```
HTTP/1.1 200 OK
Date: Sat, 19 Feb 2011 02:32:58 GMT
Server: Apache/2.2.3 (CentOS)
Connection: Keep-alive
Last-Modified: Tue, 18 Aug 2015 15:11:03 GMT
Set-cookie: session=44ecb091; path=/servlets
Content-Length: 6821

<HTML> website content ... </HTML>
```

2 of 3

## Follow-up HTTP Request

Cookies sent along with requests



```
GET /login.html/user=postman&pass=secret HTTP/1.1
Accept: image/gif, image/x-bitmap, image/jpeg, */*
Accept-Language: en
Connection: Keep-Alive
User-Agent: Mozilla/1.22
Host: educative.io
Cookie: session=44ecb091; path=/servlets
```

3 of 3



## The Dangers of Cookies 💀

While cookies seem like a great idea to make HTTP persistent when needed, cookies have been **severely abused in the past**.

If a website has stored a cookie on your browser, it knows exactly when you visit it, what pages you visit and in what order. This itself makes s



cookie monster; image attribution:

## Third-party Cookies

Also, websites may not necessarily know personally identifiable information about you such as your name (by the way, websites that require you to sign-up *do* know your name), and they may only know the value of your cookie. But what if **websites can track what you do on *other* websites?** Well, they can. Welcome to the concept of third-party cookies.

While we can't go into too much detail, it suffices to know that **third-party cookies are cookies set for domains that are not being visited.**

### Example #

1. A user visits [amazon.com](http://amazon.com).
2. A cookie for [free-stats.com](http://free-stats.com) is subsequently set on their browser because free-stats has placed an advertisement on Amazon. Notice that this is a **third-party cookie!**
3. Suppose, the user visits [ebay.com](http://ebay.com), and **eBay also has placed an advertisement for free-stats.com**.
4. The **same cookie set on the Amazon site will be reused** and sent to free-stats along in an HTTP request with the name of the host that the user is on.
5. Free-stats **can in this way track every website the user visits** that they are advertising on and create more targeted ads in order to generate greater revenue.

Also, the public has largely considered third-party cookies to be a breach of privacy and so rejected them. Most modern browsers come with the in-built option to block third-party cookies.

## Blocking Third-Party Cookies Is Not Enough! #

However, firms have come up with several workarounds including but not

limited to:

- Respawning cookies
- Flash cookies
- Entity tags
- Canvas fingerprinting

## Quick Quiz! #

1

What is a cookie?

COMPLETED 0%

1 of 2



Now that we know the basics of cookies, let's look at them in practice with a quick exercise!

# Exercise: View and Manage Your Cookies

In this lesson, we'll go through an exercise to manage your browser's cookies.

## WE'LL COVER THE FOLLOWING ^

- Viewing Cookies For a Page
- Managing Cookies
  - Chrome
  - Safari
  - Firefox

## Viewing Cookies For a Page #

For most browsers, while you can't view your entire cookie file, you *can* view cookies for individual websites.

1. Open up any **website of your choice**.
2. Open up the **developer tools**. All browsers have some form of developer tools.
3. Click on the **storage tab** to view details of the cookies that the website has set.

## Managing Cookies #

Most modern browsers allow some degree of cookie management (viewing, editing and deleting), which can be integral to protecting your privacy on the Internet. Here are some links to instructions on how to manage cookies for popular browsers. Unfortunately, for most browsers, all cookies cannot be viewed at once, but they can be managed.

## Chrome #

Google has provided some [instructions](#) to view cookies on Chrome. Have a

Google has provided some [instructions](#) to view cookies on Chrome. Here's a look!

## Safari #

This [Apple Support page](#) has some instructions on managing cookies. While you can't view the content of the cookie files, you can view which websites have stored cookies on your browser.

## Firefox #

Here's an [official page](#) to manage cookies on Firefox.

---

Now that we have a good understanding of HTTP, and HTTP specific constructs, let's move on to a new protocol!

# Email: SMTP

Let's now discuss some important protocols that make email what it is.

## WE'LL COVER THE FOLLOWING ^

- History of SMTP
- How SMTP Works
- Error Handling
- Quick Quiz!

## Introduction

Email has been a **key application** of the Internet since its early days.

Today, almost **all formal or official correspondences occur on email**. In fact, a lot of major corporations communicate primarily through email both externally and internally despite the advent of instant messaging.

Let's see how this evergreen application really works!

## SMTP

There are many protocols associated with email. One popular choice is a combination of **POP3** and **SMTP**. One is used to send emails that are stored in a user's inbox and the other is used to retrieve emails sent to you. However, the very core of electronic mail is the **Simple Mail Transfer Protocol (SMTP)**.

SMTP uses TCP, which means that transfers are **reliable**. The connection is established at **port 25**.





**Note** A good mnemonic to remember what SMTP does is **S**ending **M**ail **T**o **P**eople.

Also, for ease and consistency, we are defining **User Agents** as agents that allow users to compose, view, delete, reply to, and forward emails. Applications such as Apple Mail, Microsoft Outlook, and Gmail's webmail are examples of user agents.

## History of SMTP #

Let's delve a bit into the history of SMTP which is incredibly important to understand why it's designed the way it is.

**SMTP predates HTTP** by quite a margin and therefore has some antiquated design properties.

For example, **all SMTP messages have to be encoded into 7-bit ASCII**. This made sense in the early days when computer networks did not have the capacity to email large images, audio, or videos, and when email was primarily text that could fit in ASCII characters. We needed 7-bit encoding and decoding because mostly US-ASCII characters were being used in which the MSB (most significant bit) of the byte was 0, so there was no point in sending 8 bits per character. Instead 7 bits per character were transmitted.

Sending text with characters that require a greater number of bits per character, or binary data became challenging. Therefore, all email attachments are encoded into 7-bit ASCII even today when sending, and then decoded upon receiving. This requires additional computational work.

## How SMTP Works #

Let's look at how this ubiquitous protocol works.

1. When an email is sent, it's sent to the sender's **SMTP server** using the **SMTP protocol**.

○ The SMTP server is configured in your email client. The general

• The SMTP server is configured in your email client. The general format of the domain of the SMTP server is `smtp.example.com` where the main email address of the sender is `user@example.com`. But it's not mandatory to adhere to this format. We could set up, say, `zeus.example.com` to serve as our SMTP server, if we wanted. From a security point of view, it is probably a good idea, since people are unlikely to guess it as easily.

2. The **email is now placed on a message queue in the sending SMTP server.**
3. Then, the SMTP server initiates a connection with the recipient server and will conduct an initial SMTP handshake.
  - If the recipient is on the same SMTP server as the sender (for instance `alice@gmail.com` sending to `bob@gmail.com`), then the SMTP server doesn't need to connect to the recipient's server.
4. The SMTP server will finally send the message to the recipient's email server.
5. The **email is then downloaded from the recipient's SMTP server** using other protocols when the recipient logs in **to their email account or 'user agent.'** In other words, the recipient's SMTP server copies the email to the recipient's mail-box.



**Note** SMTP is a **push protocol** because the email client sends the email out to the server when it needs to. Which means it only *sends* data to servers. Other protocols called **Mail Access Protocols** such as **POP** and **IMAP** are used for *getting* email from a server and are called **pull** protocols because the client asks their POP/IMAP server if they have any new messages whenever they feel like.

Have a look at the following slides to get a clearer picture.



Sender's SMTP Server



Recipient's SMTP Server



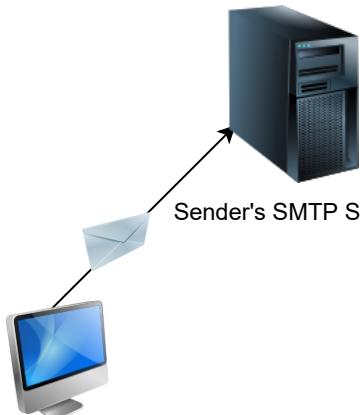
Sender



Recipient

The sender wants to send the receiver an email

1 of 7



Recipient's SMTP Server



Recipient

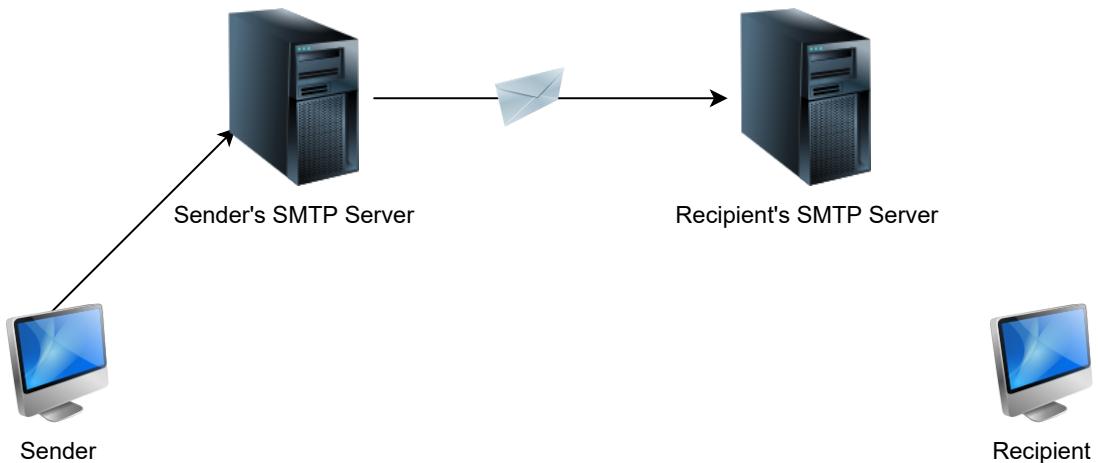
The sender composes the email and sends it

2 of 7



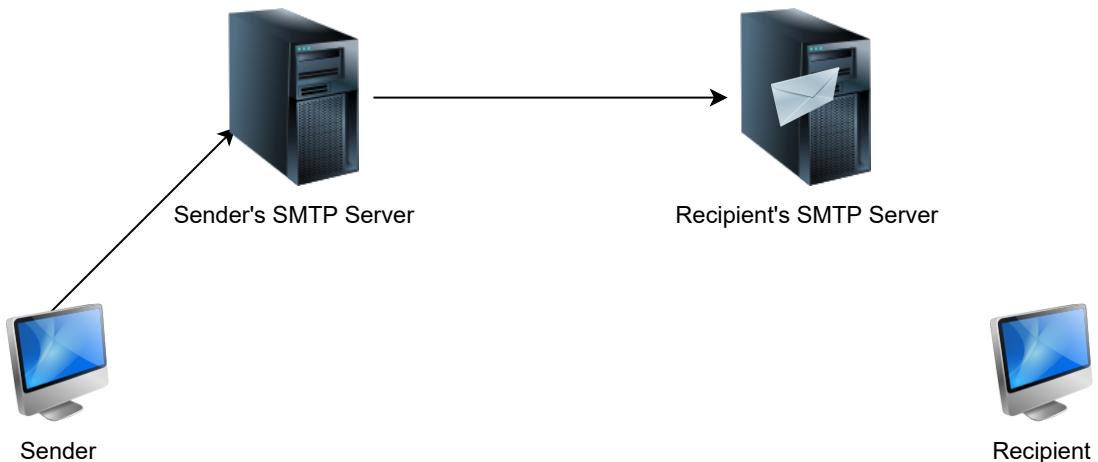
The email resides in a queue on the sender's SMTP server.

3 of 7



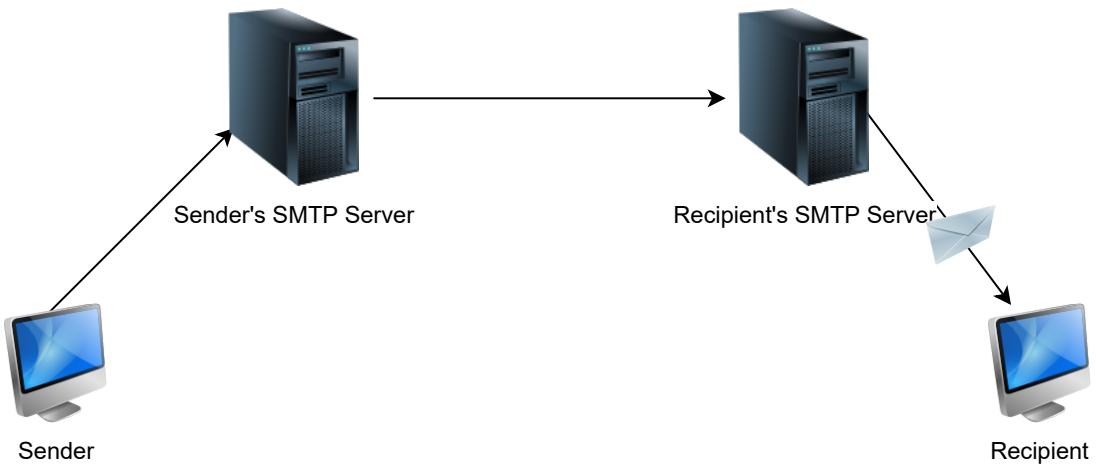
After opening an initial connection and conducting an SMTP handshake with the recipient's server, the sender's server sends the email.

4 of 7



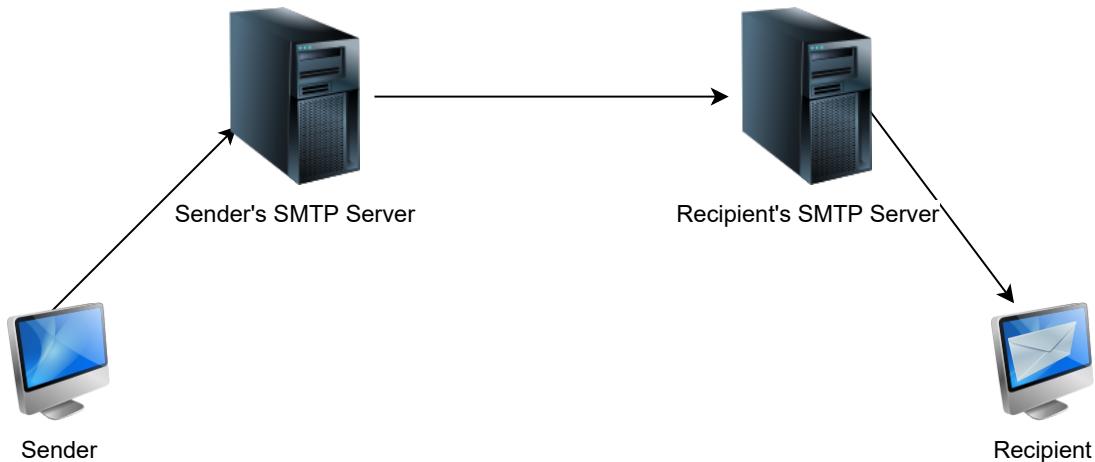
The email resides on the recipient's server in the recipient's inbox. The email will be downloaded to the recipient's user agent once they log in.

5 of 7



The recipient logs in and receives the email.

6 of 7



The recipient logs in and receives the email.

7 of 7



## Error Handling #

There are many scenarios where sending an email may fail. Here are a few.

- **If the email is not sent** for any reason such as a misspelled recipient address, the email is returned to the sender with a “**not delivered**” message.
- **If the recipient SMTP server is offline**, the sending SMTP server keeps trying to send the email, say, every 30 minutes or so. It stops trying after a few days and alerts the sender about the mail not delivered error.
- Also, SMTP does not use any intermediate servers. So, even if an attempt to send an email fails because of any reason such as the receiving server being down, the email won’t be stored on an intermediary server. It will be stored on the sending SMTP server.

## Quick Quiz! #

1

SMTP is a pull protocol

COMPLETED 0%

1 of 2



---

Let's do an exercise with another popular command-line tool to check what your domain's mail server is in the next lesson!

# Exercise: Checking a Domain's Mail Server with nslookup

In this lesson, we'll use nslookup to look up a domain's mail server.

## WE'LL COVER THE FOLLOWING ^

- **nslookup**
- Outlook
- Gmail
- Yahoo!

## nslookup #

**nslookup**, or **name server lookup**, is a command-line tool that can be used to find the name and IP address of the SMTP server for a domain like [live.com](https://live.com) or [gmail.com](https://gmail.com). Have a look at the following command.

## Outlook #

```
nslookup -type=mx https://outlook.live.com
```



The **mx** in the **-type=mx** flag stands for **Mail Exchanger** record, which essentially means the SMTP server.

There is a lot more that **nslookup** can be used for. Here's the [manpage](#) for **nslookup** if you want to learn more.

Ignore what authoritative and non-authoritative mean for now. You'll understand them when we get to [DNS](#).

Try any other domain of your choice! Here are a couple of very popular ones.

## Gmail #

```
nslookup -type=mx gmail.com
```



In this case, one of the SMTP servers for Gmail is [alt1.gmail-smtp-in.l.google.com](#).

## Yahoo! #

```
nslookup -type=mx mail.yahoo.com
```



---

Let's study pull protocols like POP and IMAP in some detail in the next lesson.

# Email: POP & IMAP

Let's now discuss the other side of the coin for how email works.

## WE'LL COVER THE FOLLOWING ^

- POP
  - POP Phases
  - POP Modes
- IMAP
- Quick Quiz!

POP and IMAP are used to retrieve email from an email server. Either one can be used. Let's discuss both.

## POP #

The most commonly used version of the **Post Office Protocol (POP)** is version 3, or **POP3**. This is how it works:

## POP Phases #

Emails are simply downloaded from the server in **4 phases: connect, authorize, transaction, update**.

- 1. Connect:** The user agent first connects to the POP3 server on TCP using **port 110**.
- 2. Authorize:** The user agent authenticates the user with a username and a password.
- 3. Transaction:** The user can now retrieve emails and mark emails for deletion.
- 4. Update:** After the user agent quits and closes the POP3 session, the server makes updates based on the user's commands. So if the user marked an email for deletion, it will delete it. No copy of a deleted email is kept on the server.

email for deletion, it will delete it. No copy of a deleted email is kept on the server.

- Note that only what's in the user's inbox is downloaded. Other folders such as sent items, outbox, or drafts are not synced. So **POP3** does not synchronize the folders.

## POP Modes #

POP works in two modes.

- **Download and delete:** Once emails are downloaded from the server to the user agent, they are all deleted from there.
- **Download and keep:** Emails are not deleted from the server once they are downloaded onto the user agent.

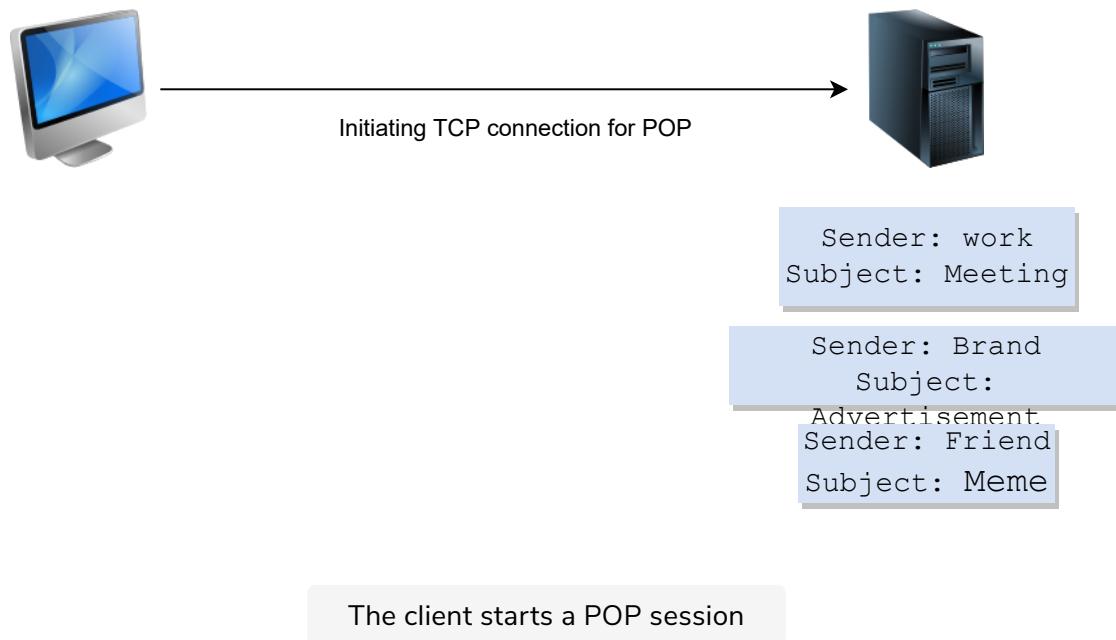
However, with the download and delete model, you can only use one client to check your emails. If you use multiple devices to check your email, this method is not appropriate because emails will not look the same across devices at different times. Also, users won't be able to reread emails from different devices.

Have a look at the following slides for an example of how emails might not be in sync on multiple devices with POP.

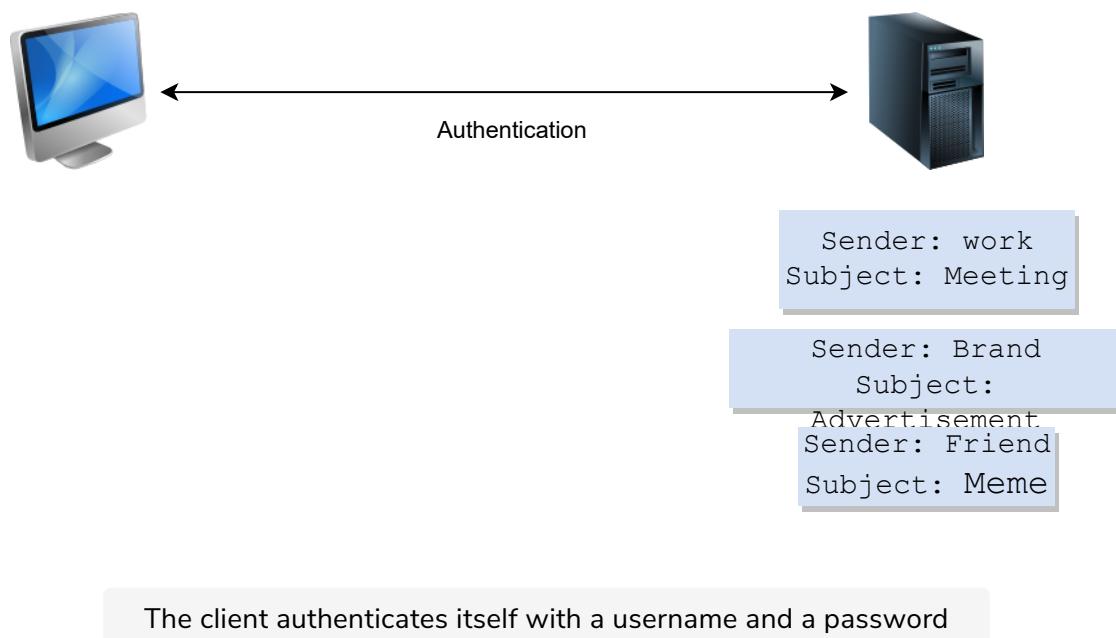


Sender: work  
Subject: Meeting

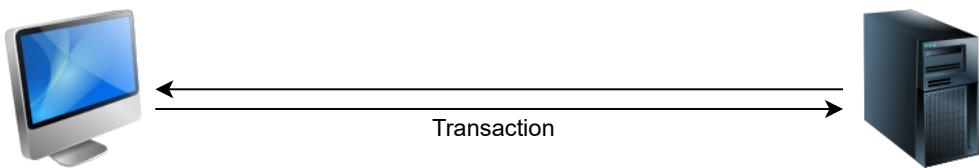
Sender: Brand  
Subject:  
Advertisement  
Sender: Friend  
Subject: Meme



2 of 10



3 of 10

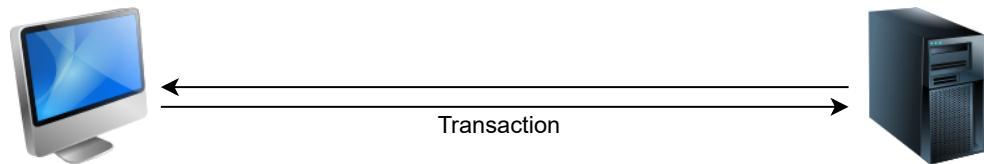


Sender: work  
Subject: Meeting

Sender: Brand  
Subject:  
Advertisement  
Sender: Friend  
Subject: Meme

The client agent requests all emails

4 of 10



Sender: work  
Subject: Meeting

Sender: Brand  
Subject:  
Advertisement  
Sender: Friend  
Subject: Meme

Sender: work  
Subject: Meeting

Sender: Brand  
Subject:  
Advertisement  
Sender: Friend  
Subject: Meme

The client requests all emails

5 of 10



Transaction

Sender: work  
Subject: Meeting

Sender: Brand  
Subject:  
Advertisement  
Sender: Friend  
Subject: Meme

Sender: work  
Subject: Meeting

Sender: Brand  
Subject:  
Advertisement  
Sender: Friend  
Subject: Meme

The client requests all emails

6 of 10



Quit

Sender: work  
Subject: Meeting

Sender: Brand  
Subject:  
Advertisement  
Sender: Friend  
Subject: Meme

Sender: work  
Subject: Meeting

Sender: Brand  
Subject:  
Advertisement  
Sender: Friend  
Subject: Meme

The client now quits the session

7 of 10



Sender: work  
Subject: Meeting

Sender: Brand  
Subject:  
Advertisement  
Sender: Friend  
Subject: Meme

During the update phase, the server updates its records. In this case, all emails are wiped clean because no new ones are received

8 of 10



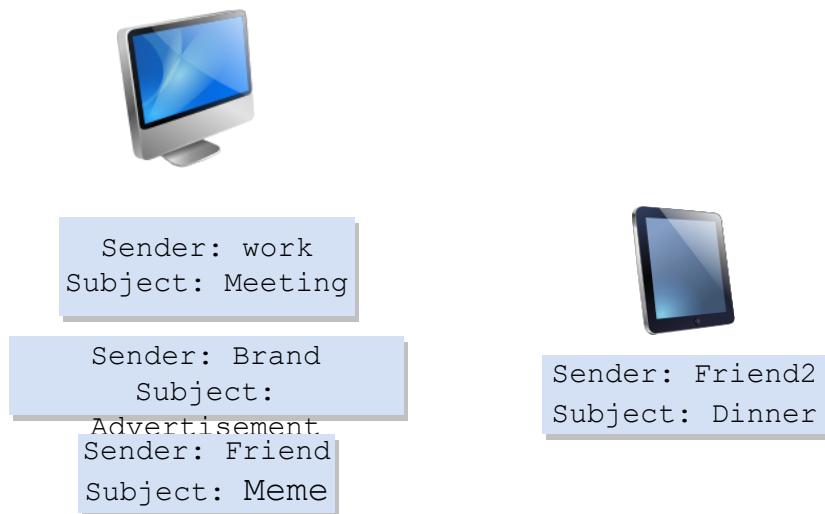
Sender: work  
Subject: Meeting

Sender: Friend2  
Subject: Dinner

Sender: Brand  
Subject:  
Advertisement  
Sender: Friend  
Subject: Meme

Suppose another email is received sometime later.

9 of 10



Suppose that the user checks their email via another device, say their cell phone. All 4 steps would occur again but they won't be able to view their previous emails.

**10 of 10**



## IMAP #

The **Internet Message Access Protocol (IMAP)**, like POP, is also a mail access protocol used for retrieving email. It is a bit more complex than POP and hence allows you to view your email from multiple devices. With IMAP, though:

- Emails are **kept** on the server and **not deleted**.
- Local copies of the emails are cached on each device.
- It **syncs** up all of the **user's folders** including custom folders.
- The **inbox** would look exactly the **same on all clients**.
- If an email is deleted from one user agent, it will be **deleted off the server**.

- Deleted emails **won't be visible** from other devices either.

## Quick Quiz! #



Which of the following are valid differences between POP3 and IMAP?

COMPLETED 0%

1 of 1



# Email: Message Format

Let's study the exact format of an email message!

## WE'LL COVER THE FOLLOWING ^

- Introduction
- Header Lines
- Message Body
- Exercise: View Raw Emails

## Introduction #

Email messages have a format the same way that HTTP request and response messages do. Let's dive right into it.

## Header Lines #

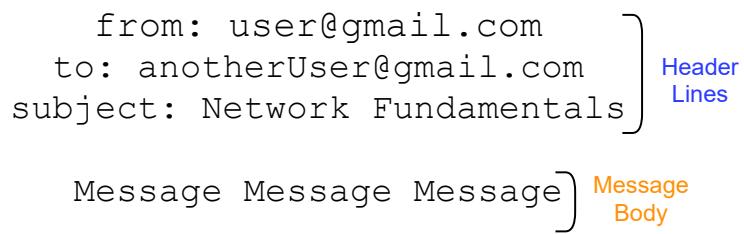
Email messages start with header lines, much akin to HTTP. The header lines contain important metadata about the email.

- The header lines consist of keywords followed by a colon, followed by a value.
- Every header line is separated by a new line with a carriage return (`\r`).
- Every header must have the `To:` and `From:` header lines.
- The rest of the headers, including the `subject:` header line, are optional.

## Message Body #

The message body of the email follows the header lines after a blank line.

Here is an example of what an email message looks like:



## Exercise: View Raw Emails #

Open up your favorite email agent. Google search “view headers with *name of agent*.” For example, I could Google “view headers with Gmail.” Here are some instructions for common clients:

1. [Gmail](#)
2. [Outlook](#)
3. [Yahoo!](#)

Once you have the instructions, study the headers. Can you figure out what each does? For your reference, here is a sample of email headers. Note that they are a bit simplified for your ease.

```

Delivered-To: user@gmail.com
Date: Thu, 16 May 2019 03:36:28 +0000 (UTC)
From: Fahim from Educative <fahim+newsletter@educative.io>
Mime-Version: 1.0
Reply-to: fahim@educative.io
Subject: Data analysis with R
To: user@gmail.com
  
```

```

Content-Transfer-Encoding: quoted-printable
Content-Type: text/plain; charset=UTF-8
Mime-Version: 1.0
  
```

Hey User,

With the way technology is evolving, more and more data is being produced and tracked every day. And because of that, the skills to work with that data, to make sense of it and turn into useful insight, are more in-demand than ever.

n ever before.

If recent trends are anything to go by, in the future the ability to work with large quantities of data won't be the field of just data scientists - it's going to become a necessary skill across industries, kind of like using a word processor.

For years, R has been at the forefront of the data science revolution. It's beloved by data scientists and statisticians for its robust statistical functionality, outstanding graphing ability, and extensibility through packages. The recent data science craze has just breathed new life into it.

Learn R from Scratch [https://www.educative.io/collection/6151088528949248/5=357220915052544?utm\\_source=3Dsendgrid&utm\\_medium=3Demail&utm\\_campaign=3Dlearn-r-from-scratch&utm\\_content=3D](https://www.educative.io/collection/6151088528949248/5=357220915052544?utm_source=3Dsendgrid&utm_medium=3Demail&utm_campaign=3Dlearn-r-from-scratch&utm_content=3D) is designed to get you up to speed writing code in R as quickly as possible. You'll start with the very basics and work your way up to advanced concepts like exception handling. By the time you're done, you'll be able to write detailed, useful code in R yourself.

Get started with R, stay on top of the data science craze, and solve real-world problems with data.

Happy learning!

-- Fahim

CEO & co-founder, educative.io

1203 114th Ave SE,  
Bellevue, WA 98004



Note the headers are from a received email and not the headers

when it was sent, which is what we discussed initially. So, the **Delivered-To** header is derived from the **To:** header in the originally sent email. The SMTP or the POP server probably make this transformation. More likely the SMTP server.

If you wish to study each of these headers and the format in detail, have a look at [RFC5322](#).

1

Why is SMTP not used for transferring emails from the recipient's mail server to the recipient's user agent?

COMPLETED 0%

1 of 7



Now that we have a good idea of email, let's move on to the directory of the web: DNS.



# DNS: Introduction

DNS is the Internet's directory service. Let's jump right in!

## WE'LL COVER THE FOLLOWING



- How Do We Find Things on The Internet?
- Distributed Hierarchical Database
- DNS Namespaces
  - Root DNS Servers
  - Top-level Servers
  - Authoritative Servers
  - Local DNS Cache
  - Local DNS Servers
- Quick Quiz!

## How Do We Find Things on The Internet? #

Generally, using one of the following ways:

- **Addresses or locations** that specify where something is.
  - Just like with a physical address, we still may need a map to get to the address. On the Internet, the addresses are typically IP addresses, and routers know the map.
- **Names.** In particular, domain names, or the unique name that identifies a websites, are mapped into IP addresses based on lookup service that uses a database. The most well-known lookup service is the **Domain Name System (DNS)**. So when you enter the URL '[educative.io](https://educative.io)' into your browser, it uses DNS to find the actual IP address of the server that hosts it.
- **Content-based addressing.**
  - The content itself is used to look up its location

• The content itself is used to look up its location.



**Note A Useful Analogy:** Domain names are like actual people names and IPs are like phone numbers.

In this lesson, our focus will be on **DNS**, the client-server application layer protocol that translates hostnames on the Internet to IP addresses.

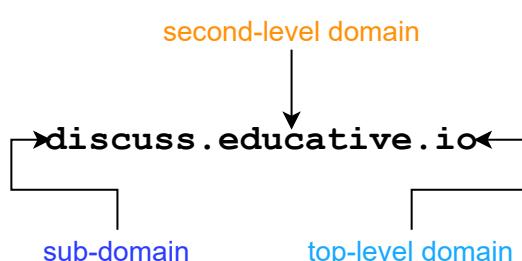
At the core, the Internet operates on IP addresses, but these are difficult to remember for humans. So, DNS names are preferably used at the application layer for which the DNS provides a mapping to IP addresses. For example, HTTP first translates the DNS hostname provided by the user in the URL to its IP address and then attempts to connect to the server. Furthermore, DNS is not just a protocol. It also consists of a distributed database of names that map to IP addresses. So essentially it's a directory service.

## Distributed Hierarchical Database #

One single database on one single server does not scale for reasons such as:

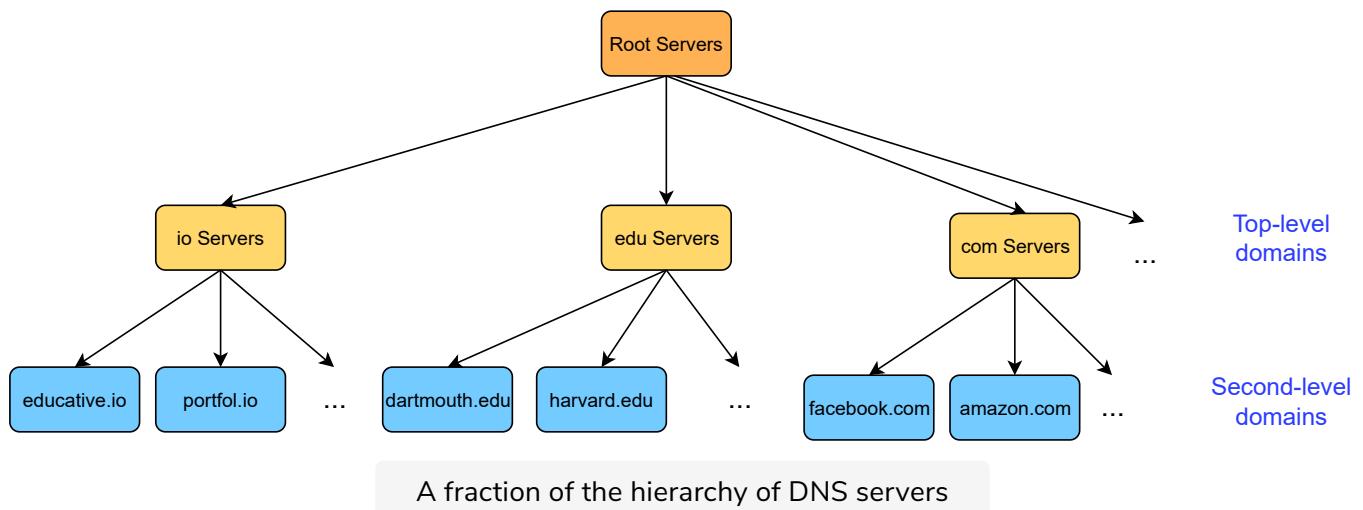
- Single point of failure. If the server that has the database crashes, DNS would stop working entirely, which is too risky.
- Massive amounts of traffic. Everyone would be querying that one server. It will not be able to handle that amount of load.
- Maintenance. Maintaining the server would become critical to the operation of DNS.
- Location. Where would the server be located?

This is why DNS employs several servers, each with part of the database. Also, the servers exist in a hierarchy. To understand this hierarchy better, you need to understand how URLs are broken down into their hierarchies. Have a look at the following diagram.



# DNS Namespaces #

The parts of the URL above roughly map to DNS servers. These servers manage the abstract space of domains. The servers all exist in a *hierarchy*. Have a look at the following diagram.



## Root DNS Servers #

Root DNS servers are the first point of contact for a DNS query (after the client's local cache of names and IP addresses). They exist at the top of the hierarchy and point to the appropriate TLD server in reply to the query. So a query for `educative.io` would return the IP address of a server for the top-level domain, `io`.

As of the writing of this course, there are 1017 instances of root servers operated by 12 different organizations. To get a full list and an interactive map, have a look at [root-servers.org](http://root-servers.org).

## Top-level Servers #

Servers in the top-level domain hold mappings to DNS servers for certain domains. Each domain is meant to be used by specific organizations only. Here are some common domains:

- **com**: This TLD was initially meant for **commercial** organizations only - but it has now been opened for general use.
  - Example: [codinginterview.com](http://codinginterview.com)
- **edu**: Used by educational institutions.

- Example: [stanford.edu](http://stanford.edu)
- **gov:** Only used by the U.S. government.
  - Example: [nasa.gov](http://nasa.gov)
- **mil:** Used by U.S. military organizations.
  - Example: army.mil
- **net:** It was initially intended for use by organizations working in network technology such as ISPs, but it is now a general purpose domain like com.
  - Example: [doubleclick.net](http://doubleclick.net)
- **org:** This domain was intended for non-profit organizations but has been opened for general use now.
  - [mozilla.org](http://mozilla.org)
- **pk, uk, us,...:** Country suffixes. 244 two-letter ones exist.
- Some new and uncommon suffixes include: **name, mobi, biz, pro.**
- **International domains:** 中國

Today, the set of top-level domain-names is managed by the [Internet Corporation for Assigned Names and Numbers](#) (ICANN).

Once one of these servers is contacted, it points to the authoritative name server of that organization.

## Authoritative Servers #

Every organization with a public website or email server provides DNS records. These records have hostname to IP address mappings stored for that organization. These records can either be stored on a dedicated DNS server for that organization or they can pay for a service provider to store the records on their server.

This is the next link in the chain. If this server has the answer that we are looking for, the IP address that it has is finally returned to the client. However, this server may not have the sought after answer if the domain has a **sub-domain**. In that case, this server *may* point to a server that has records of the subdomain.

For instance, if the DNS record for [cs.stanford.edu](http://cs.stanford.edu) is being looked for, a DNS server separate from ‘[stanford.edu](http://stanford.edu)’ may hold records for the sub-domain ‘cs.’

## Local DNS Cache #

- DNS mappings are often also cached locally on the client end-system to avoid repetitive lookups and saves time for often visited websites.
- This is done via an entity called the **local resolver library**, which is part of the OS. The application makes an API call to this library. This library manages the local DNS cache.
- If the local resolver library does not have a cached answer for the client, it will contact the organization's local DNS server.
- This local DNS server is typically configured on the client machine either using a protocol called **DHCP** or can be configured statically.
- So, if it's configured manually, any local DNS server of the client's choice can be chosen. A few open DNS servers are incredibly popular, such as the ones by [Google](#).

## Local DNS Servers #

There is one type of server that we ignored — the **local DNS Server**. Local DNS servers are usually the first point of contact after a client checks its local cache. These servers are generally hosted at the ISP and contain some mappings based on what websites users visit.



**Security Warning:** ISPs have a record of which IP address they assigned to which customers. Furthermore, their DNS server has the IP addresses of who contacts it and what hostname they were trying to resolve. So your **ISP technically has a record of all of the websites you visit. Yikes!** P.S. if this makes you uncomfortable, you can change your DNS server to any open public DNS server.

For an overview of how a query is resolved, have a look at the following slides.



Root Server



A user



The user's local DNS server



Server for 'io'



Server for educative

A user opens up a browser to start working on the networks course that they purchased

1 of 12



Root Server



A user



The user's local DNS server



Server for 'io'



The user enters 'educative.io' into their browser

2 of 12



Root Server



A user



The user's local DNS server



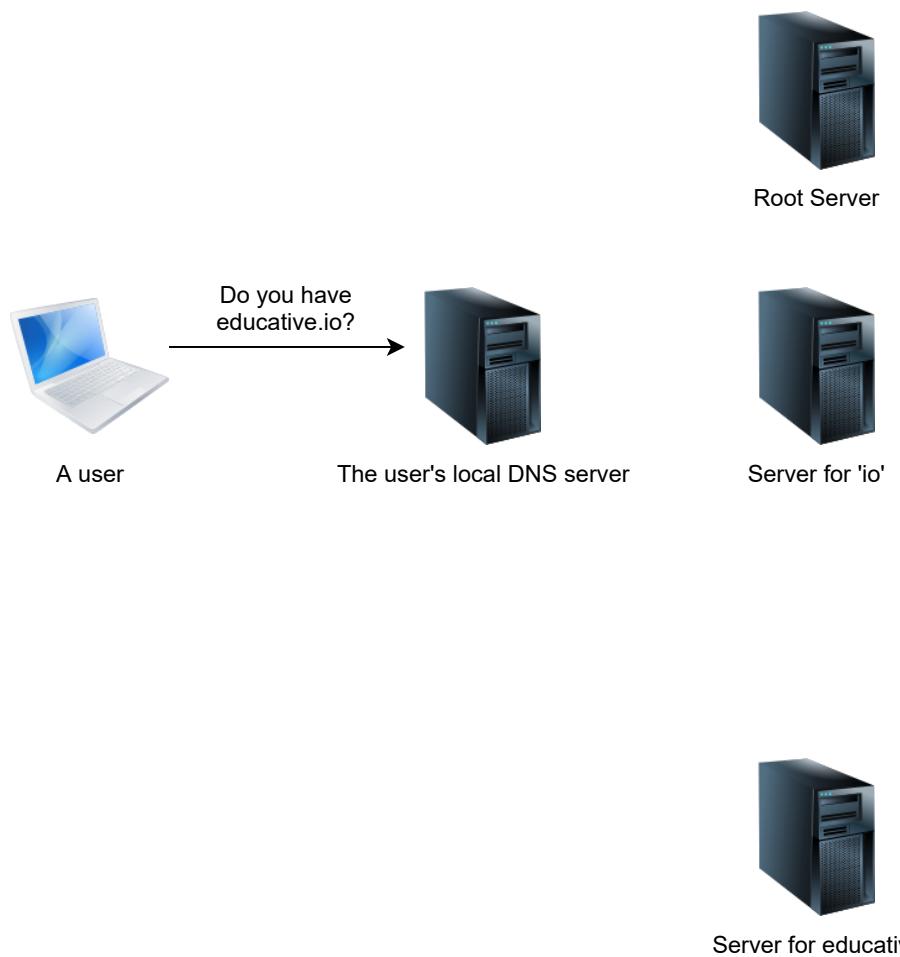
Server for 'io'



Server for educative

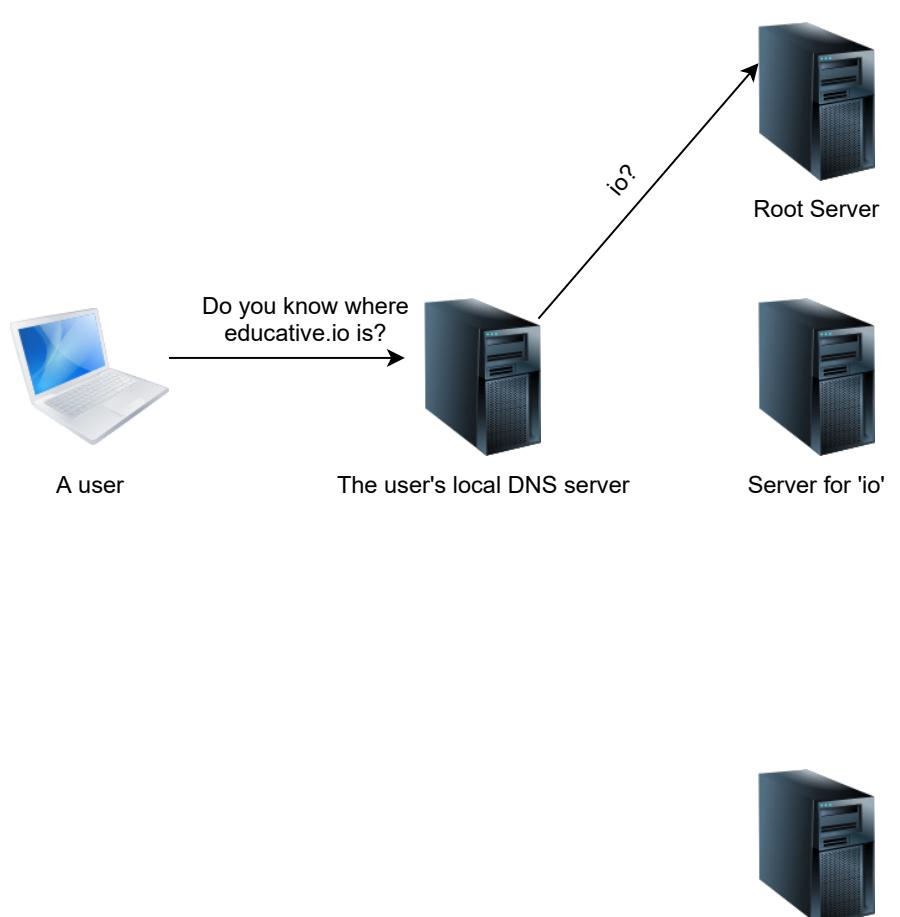
DNS starts. The first place to look for is the local cache.

3 of 12



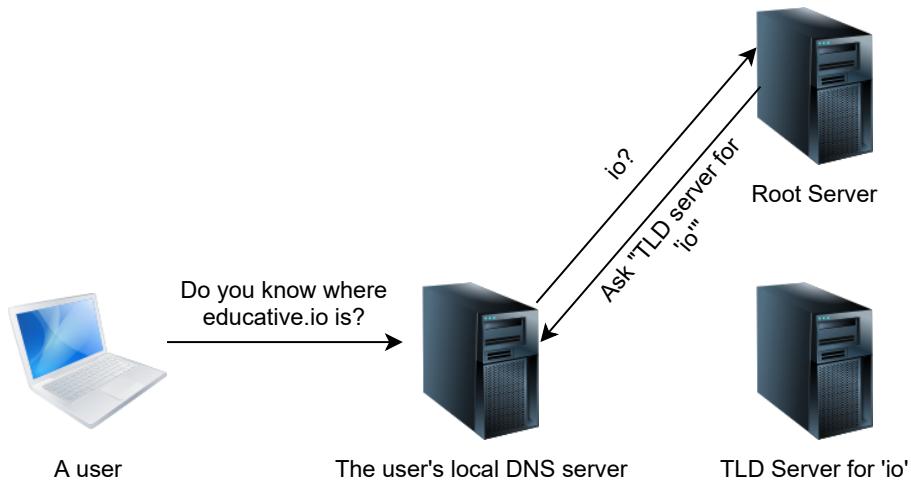
Assuming the IP address was not found in the local cache, the client contacts the local DNS server.

4 of 12



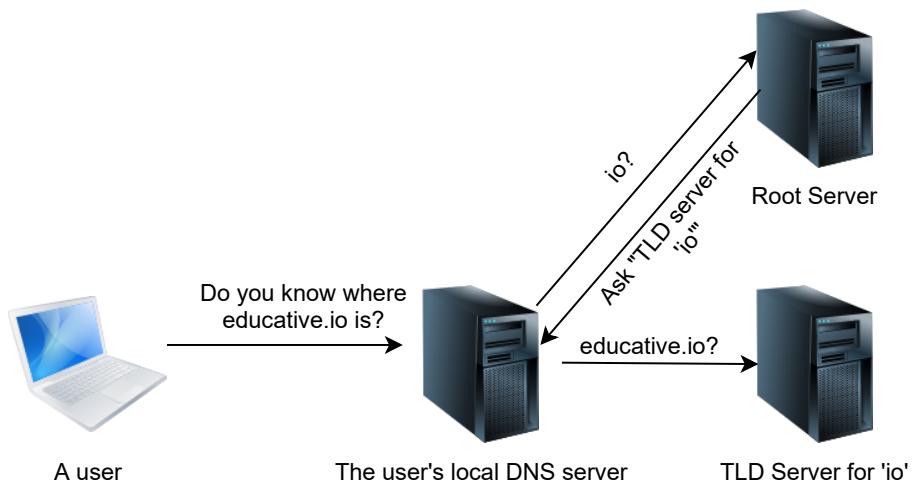
If the local DNS server does not have a record for the website requested, it asks a root server

5 of 12



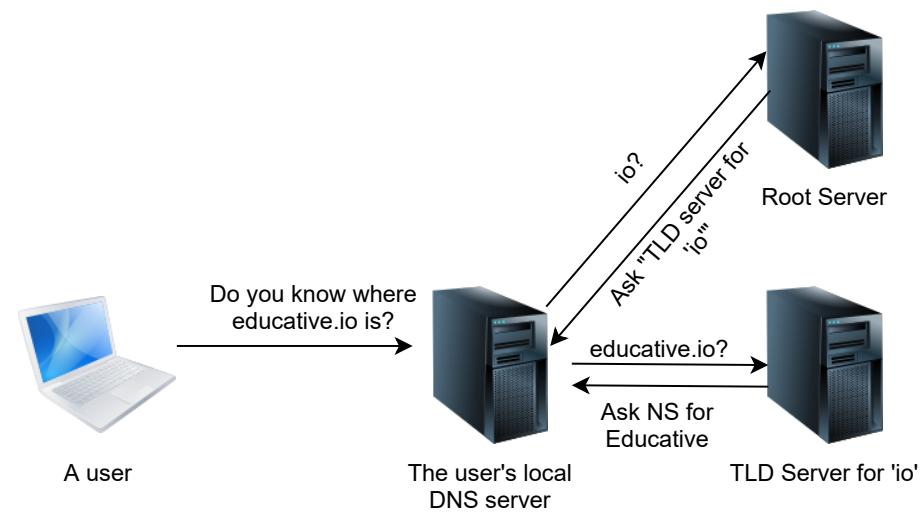
The root server points to the top-level domain server for 'io'

6 of 12



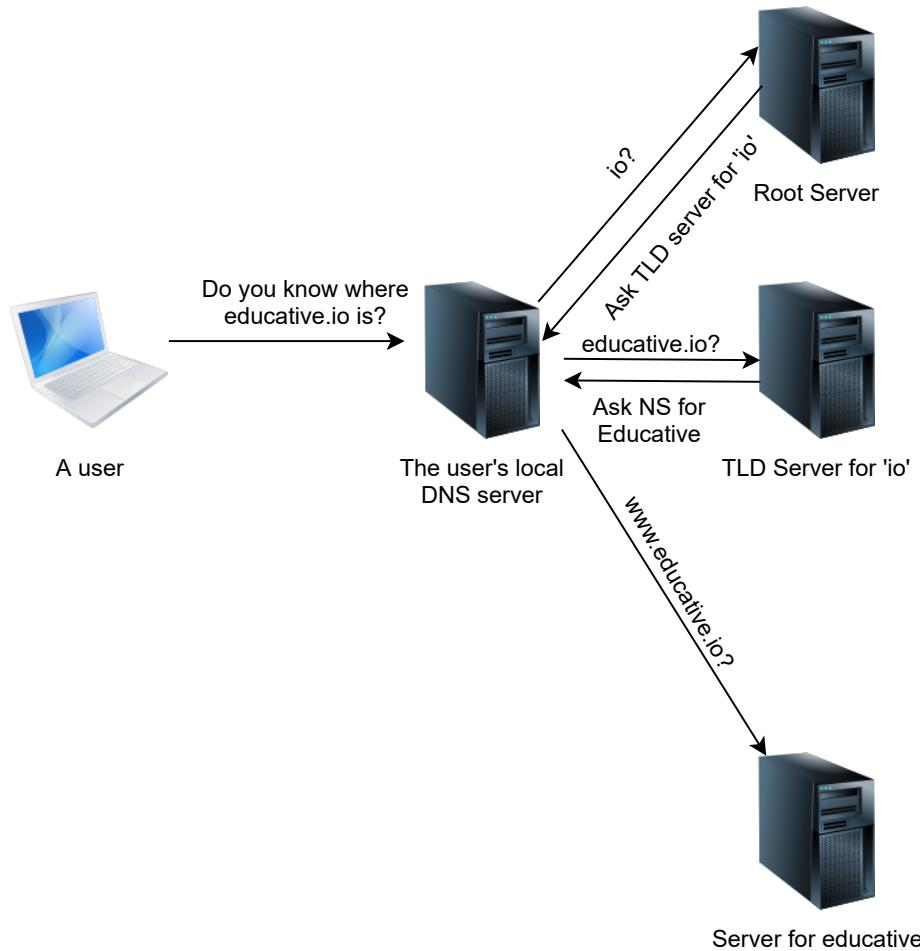
The local DNS server asks the TLD server for the authoritative name server that knows about educative.io

7 of 12



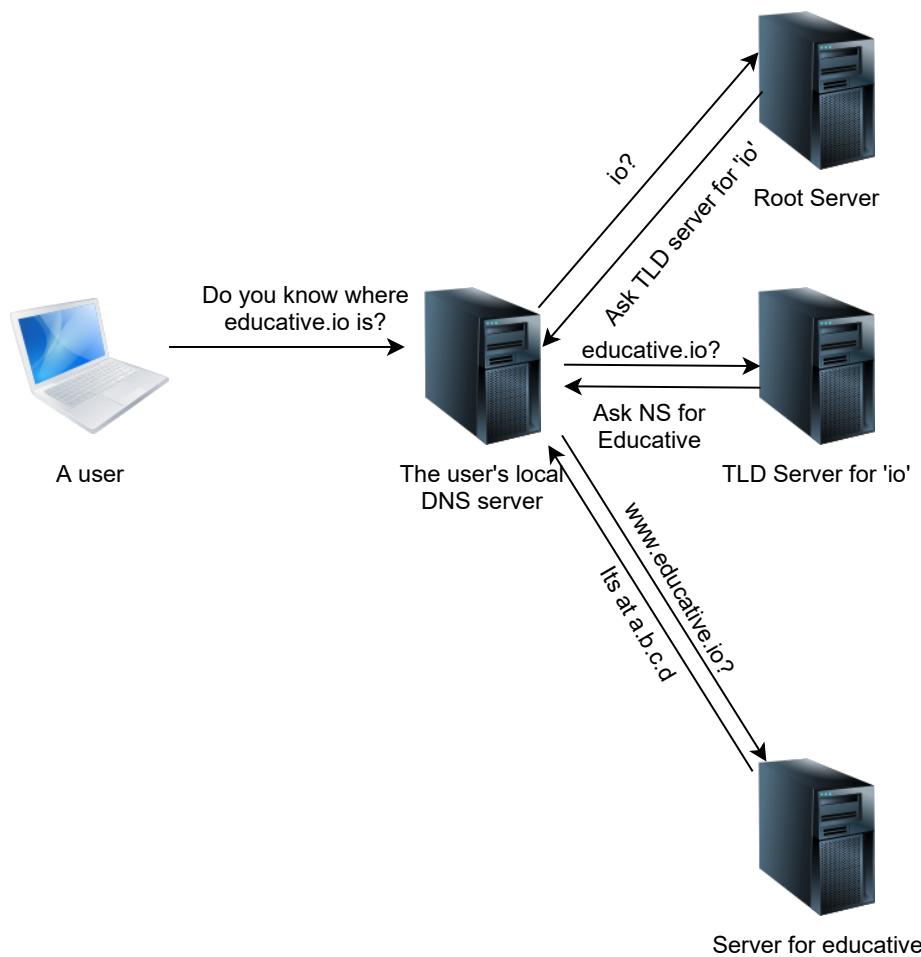
The TLD server gives the address of the name server for educative

8 of 12



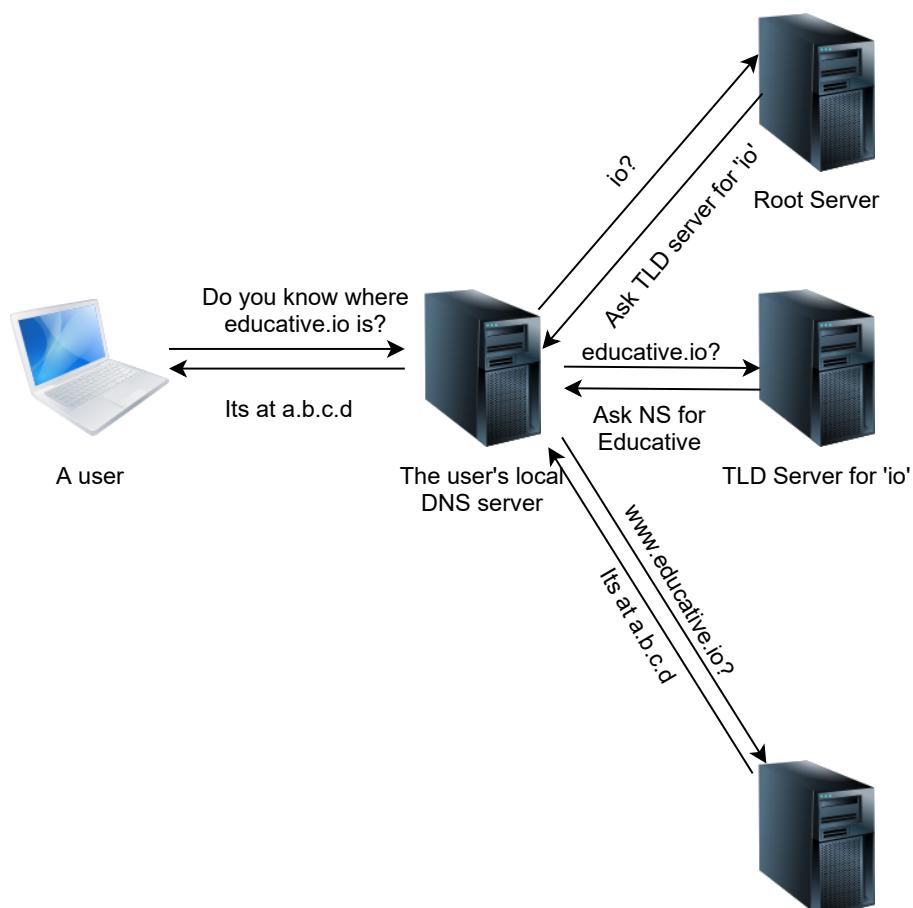
The local DNS server now contacts the NS for educative

9 of 12



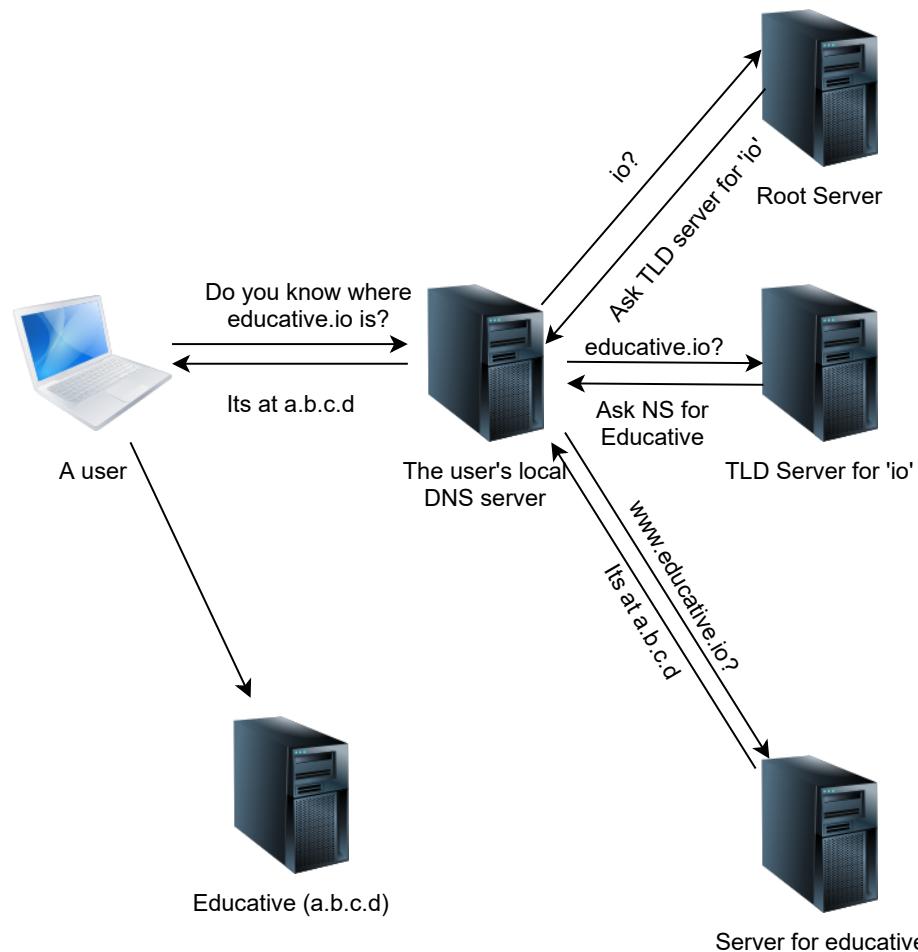
The authoritative name server gives Educative's location to the local DNS server

10 of 12



The local DNS server returns the IP address of Educative it to the client

11 of 12



The client can now connect to the server using other protocols like HTTP

12 of 12

## Quick Quiz! #

1

Which entity is contacted first when resolving a DNS name?

COMPLETED 0%

1 of 2



---

Now that we have an idea of how DNS works, let's look at an exercise in the next lesson!

# Exercise: Finding Name Servers

In this lesson, we will look at a few command-line tools to lookup domain name servers.

## WE'LL COVER THE FOLLOWING



- Finding The Authoritative Name Server Using `host`
- Checking What Your Local DNS Server Is
- Output

## Finding The Authoritative Name Server Using `host` #

```
host -t ns google.com
```



`host` is a DNS lookup utility. It is normally used to map hostnames to IP addresses. However, with a combination of flags, it can be programmed to perform a myriad of DNS-related tasks.

The syntax of the command above, for instance, is

```
host -t ns hostname.com
```



- `host` invokes the host command
- `-t` is the `type` flag. It is used to specify the type of the command. Check out [host's manpage](#) for a list of all the types available.
- `ns` specifies the type. It stands for the name server in this case.
- `hostname.com` can be any website of your choice.

We encourage you to experiment, explore, and get creative with the tool!

# Checking What Your Local DNS Server Is #

To check the IP address of your local DNS server, run the following command on UNIX based machines. If you're on a mobile machine, try [www.whatismydnsserver.com](http://www.whatismydnsserver.com). There are a lot of instructions available for Windows machines online.

```
cat /etc/resolv.conf
```



## Output #

Here is what the output may look like

```
# Dynamic resolv.conf(5) file for glibc resolver(3) generated by resolvconf(8)
#       DO NOT EDIT THIS FILE BY HAND -- YOUR CHANGES WILL BE OVERWRITTEN
nameserver 169.254.169.254
search c.educative-exec-env.internal google.internal
```

You can safely ignore the first two lines since they are comments. On the third line, `nameserver` shows the IP address of the local DNS server. On the last line, `search` represents the default search domain that is used to resolve a query for a domain with no suffix (for example, www.facebook).



**Note:** Educative's code widgets run on a remote server. The code runs there, the output is then sent back to your machine and displayed. So, the DNS server shown here is local to the server that runs Educative's code.

Now that we have a good idea of how DNS works, let's study DNS records and messages in the next lesson.



# DNS: Records and Messages

Let's now get into what DNS records and messages look like.

## WE'LL COVER THE FOLLOWING ^

- Resource Records
  - Format
  - Types of RRs
- DNS Messages

## Resource Records #

The DNS distributed database consists of entities called **RRs**, or **Resource Records**.

### Format #

RRs are 4-tuples with the following entries:

```
(name, value, type, ttl)
```

Every resource record has a **type** and a **TTL** along with a **name-value** pair. The TTL specifies **how long an RR entry can be cached by the client**. The remaining fields are described for each RR type below.

## Types of RRs #

- **Address**
  - Type **A** addresses are used to map IPv4 addresses to hostnames.
  - **name** is the hostname in question.
  - **value** is the IP address of the hostname.
  - **Example:** `educative.io. 299 IN A 104.20.7.183` where 299 is the TTL, `educative.io` is the name, `A` is the type, and `104.20.7.183` is the value.

value.

- **Canonical name**

- Type **CNAME** records are records of alias hostnames against actual hostnames. For example if, `ibm.com` is really `servereast.backup2.com`, then the latter is the canonical name of `ibm.com`.
- `name` is the alias name for the real or ‘canonical’ name of the server.
- `value` is the canonical name of the server.
- **Example:** `bar.example.com. CNAME foo.example.com.`

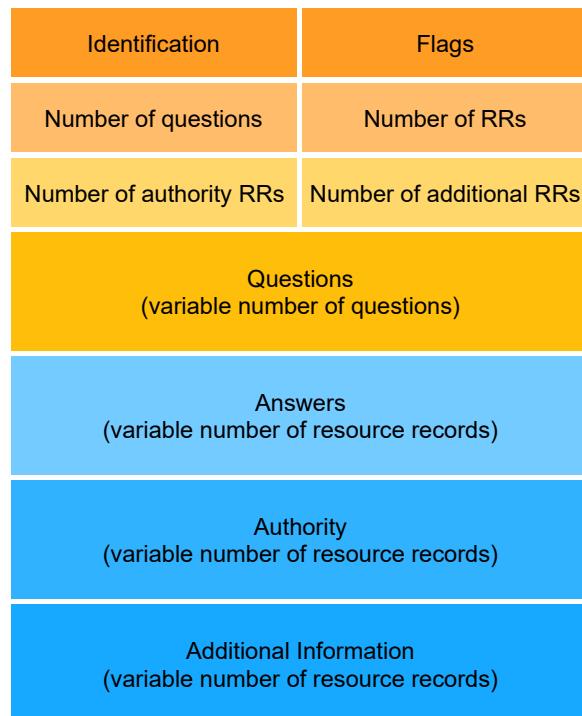
- **Mail Exchanger**

- We have seen this one before! Type **MX** records are records of the server that accepts email on behalf of a certain domain.
- The `name` is the name of the host.
- `value` is the name of the mail server associated with the host.
- **Example:** `educative.io mail exchanger = 10 aspmx2.googlemail.com.`

These resource records are stored in text form in special files called **zone files**.

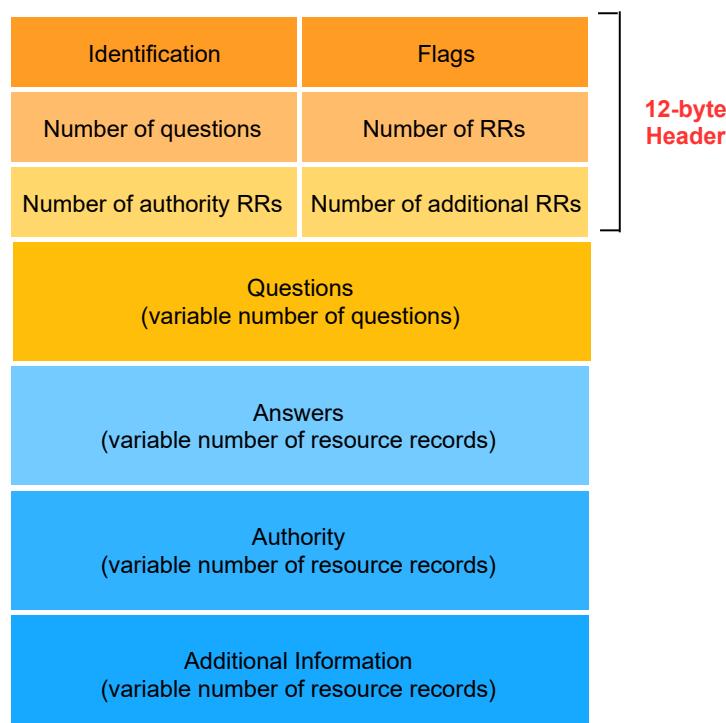
## DNS Messages #

There are a few kinds of DNS messages, out of which the most common are **query** and **reply**, and both have the same format. Study the following slides for a detailed overview of a DNS message.



Here is a generic DNS message

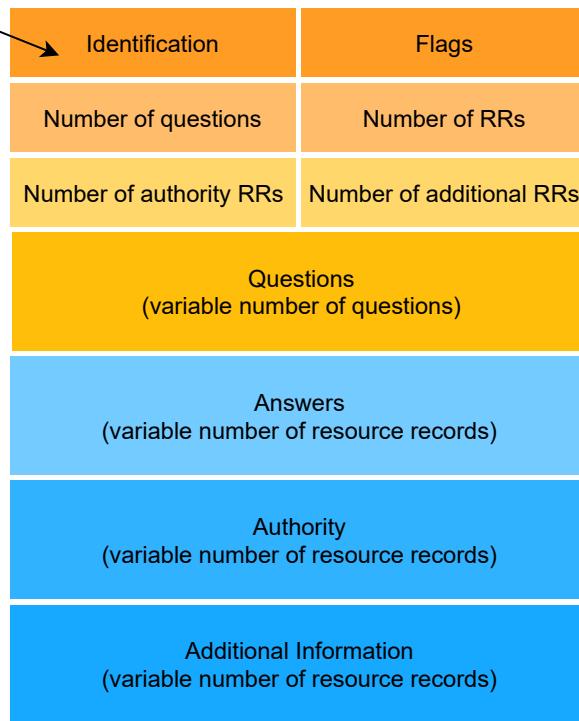
1 of 10



Let's discuss the header first. It is 12-bytes long and contains a number of fields.

2 of 10

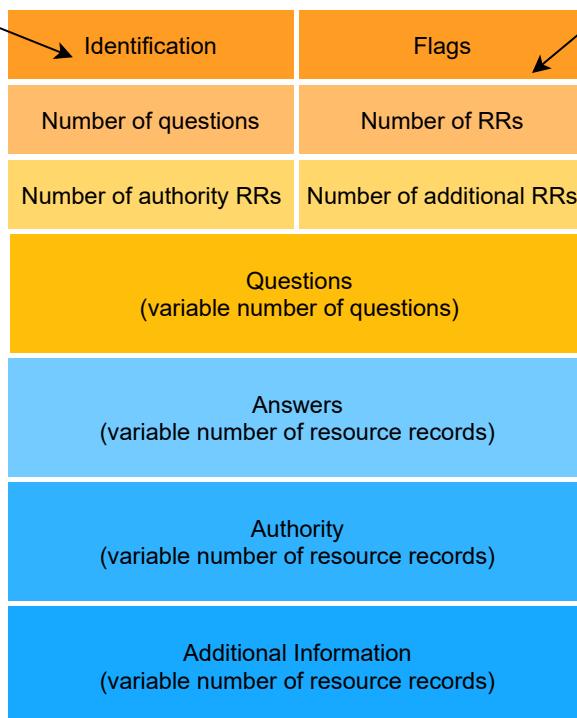
The Identification field is a 16-bit number that identifies the query. The number is copied into reply messages so that end-systems can identify what query this was meant to be a reply for.



The identification field.

3 of 10

The Identification field is a 16-bit number that identifies the query. The number is copied into reply messages so that end-systems can identify what query this was meant to be a reply for.



The flag field contains a number of 1-bit flags:

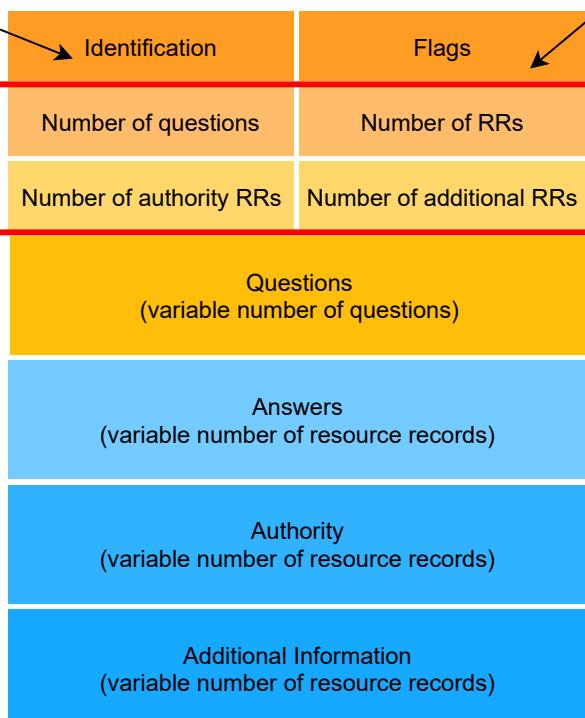
- query or reply
- recursion desired
- recursion available
- reply is authoritative

The flags field.

4 of 10

The Identification field is a 16-bit number that identifies the query. The number is copied into reply messages so that end-systems can identify what query this was meant to be a reply for.

The Indicate number of instances of the data fields that follow



The flag field contains a number of 1-bit flags:

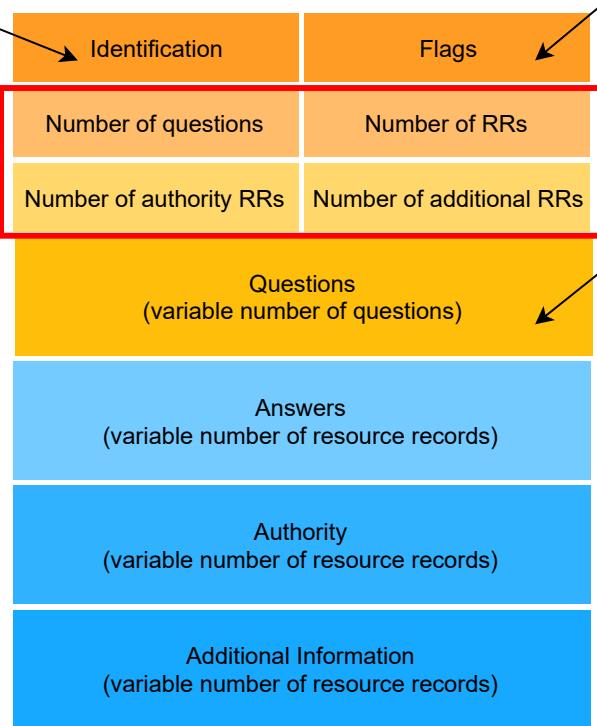
- query or reply
- recursion desired
- recursion available
- reply is authoritative

Number-of fields. These indicate the number of instances of the 4 data sections that follow.

5 of 10

The Identification field is a 16-bit number that identifies the query. The number is copied into reply messages so that end-systems can identify what query this was meant to be a reply for.

The Indicate number of instances of the data fields that follow



The flag field contains a number of 1-bit flags:

- query or reply
- recursion desired
- recursion available
- reply is authoritative

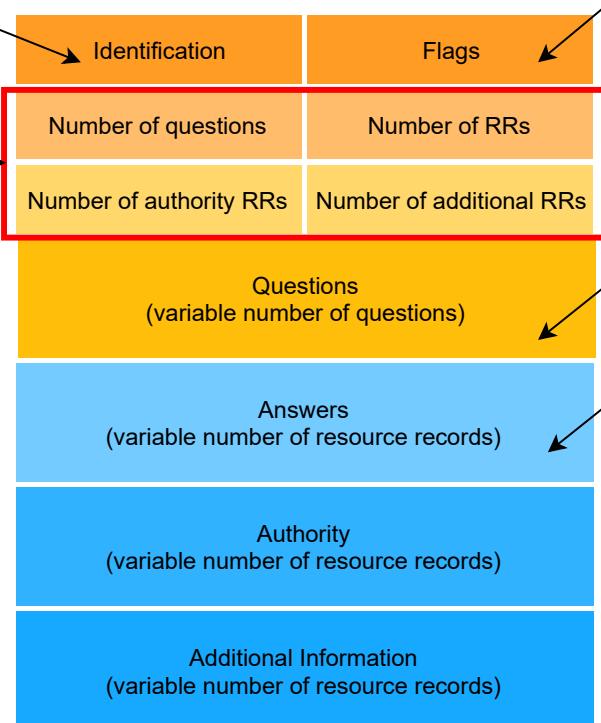
The questions field contains information about the query itself: the name (google.com) and the type (MX, A, etc)

The questions field.

6 of 10

The Identification field is a 16-bit number that identifies the query. The number is copied into reply messages so that end-systems can identify what query this was meant to be a reply for.

The Indicate number of instances of the data fields that follow



The flag field contains a number of 1-bit flags:

- query or reply
- recursion desired
- recursion available
- reply is authoritative

The questions field contains information about the query itself: the name (google.com) and the type (MX, A, etc)

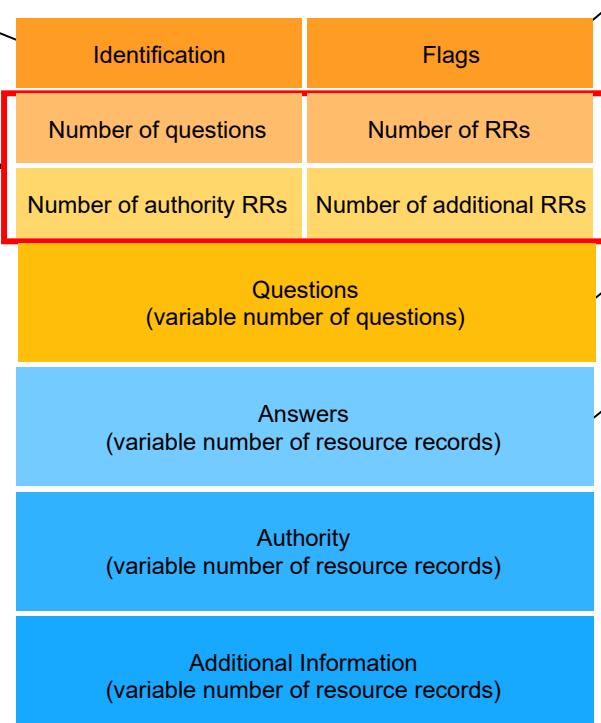
The answers field contains the resource records in response to a query

### The Answers field.

7 of 10

The Identification field is a 16-bit number that identifies the query. The number is copied into reply messages so that end-systems can identify what query this was meant to be a reply for.

The Indicate number of instances of the data fields that follow



The flag field contains a number of 1-bit flags:

- query or reply
- recursion desired
- recursion available
- reply is authoritative

The questions field contains information about the query itself: the name (google.com) and the type (MX, A, etc)

The answers field contains the resource records in response to a query. Multiple records can be returned as you have seen from the commands above.

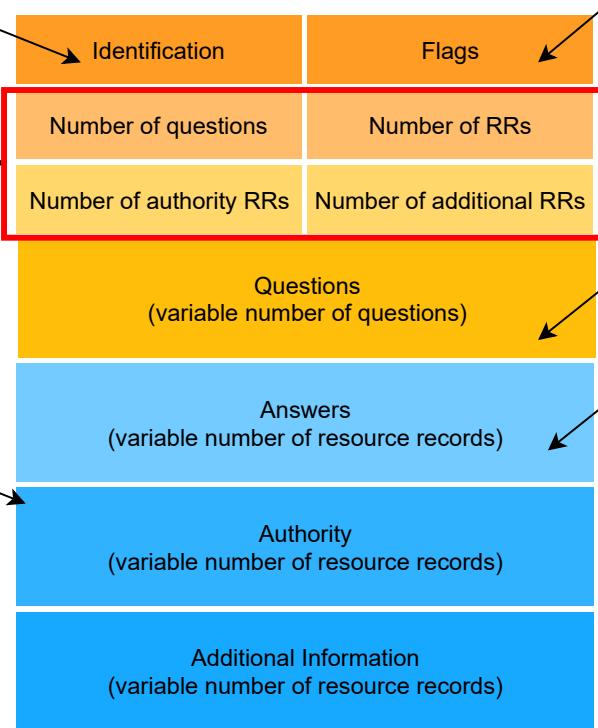
## The Answers field.

8 of 10

The Identification field is a 16-bit number that identifies the query. The number is copied into reply messages so that end-systems can identify what query this was meant to be a reply for.

The Indicate number of instances of the data fields that follow

The authority section contains the records of authoritative servers



The flag field contains a number of 1-bit flags:

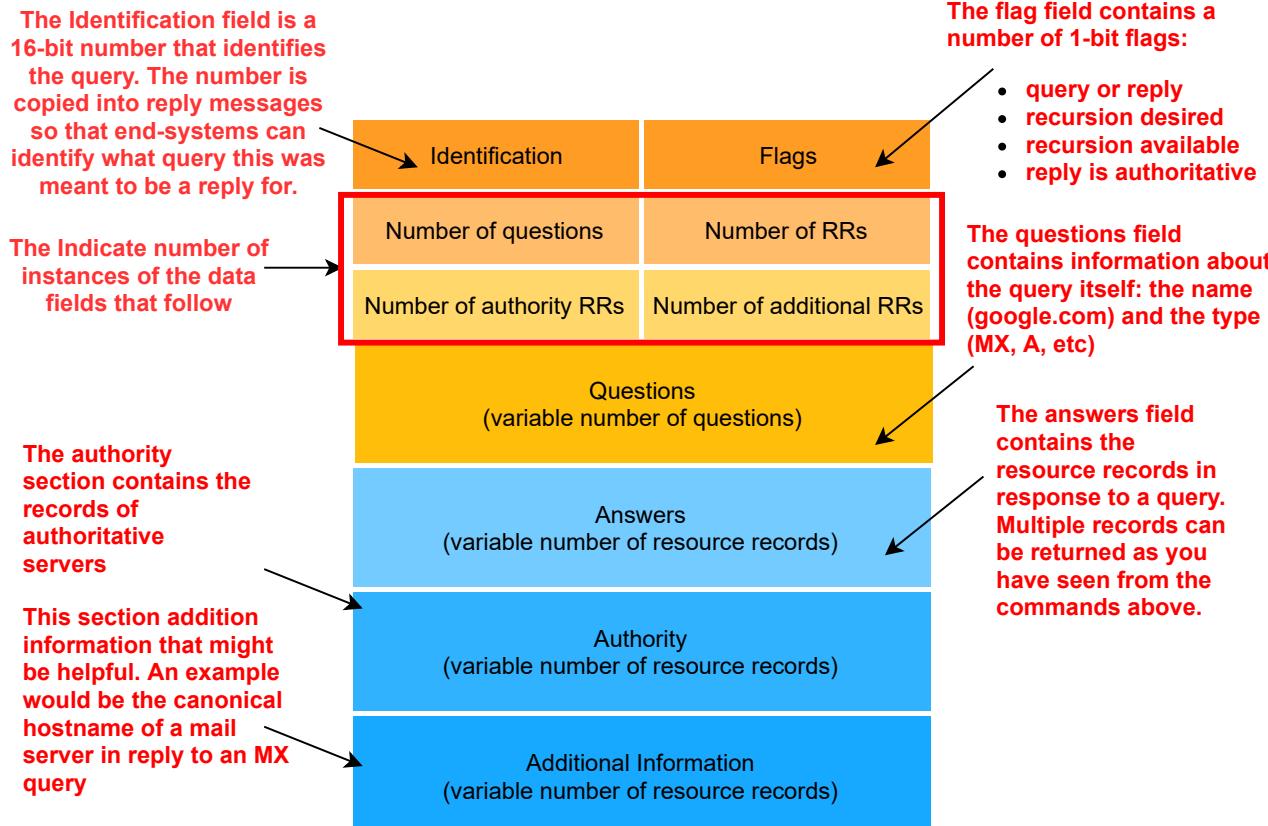
- query or reply
- recursion desired
- recursion available
- reply is authoritative

The questions field contains information about the query itself: the name (google.com) and the type (MX, A, etc)

The answers field contains the resource records in response to a query. Multiple records can be returned as you have seen from the commands above.

## The Authority field.

9 of 10



The Additional Information field.

10 of 10



There are also **zone transfer request and response**. But, those are not used by common clients. Backup or secondary DNS servers use them for **zone transfers**, which are when zone files are copied from one server to another. This takes place over TCP.

1

Which of the following are valid DNS record entry types?

COMPLETED 0%

1 of 3



---

In the next lesson, we'll use command-line tools to look at DNS response messages and resource records!

# Exercise: Looking At DNS Response Messages and Resource Records

In this lesson, we'll use command-line tools to look at DNS response messages and resource records!

## WE'LL COVER THE FOLLOWING ^

- Revisiting `Nslookup`
  - Output
- Looking At Real DNS Response Messages With `dig`

## Revisiting `Nslookup` #

```
nslookup -type=A educative.io
```



`nslookup` is a versatile tool for DNS lookups. The `type` flag determines the type of RR that you want to look into!

## Output #

`nslookup` can be used to look at DNS records. In this example, we looked up `educative.io`.

Here's what the output may look like:

```
Server:      169.254.169.254
Address:     169.254.169.254#53
```

```
Non-authoritative answer:
```

```
Name:      educative.io
```

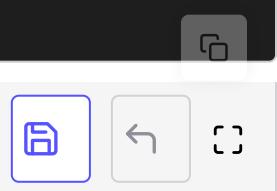
```
Address:   104.20.7.183
```

```
Name:      educative.io
```

- The first two lines are the IP address of the local DNS server which is **169.254.169.254** in our case.
- The last few lines return the type A RR that maps **educative.io** to the IP address **104.20.6.183**. It says ‘non-authoritative’ because the answer is coming from a local DNS server’s cache, and not from Educative’s authoritative DNS server.

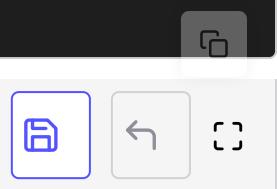
If you’re wondering what **TTL** values look like, run the following command. The value in the **TTL** field is in seconds, so a value of **279** is **4** minutes and **39** seconds.

```
nslookup -debug educative.io
```



## Looking At Real DNS Response Messages With **dig** #

```
dig educative.io
```



**dig** is a command-line tool used to query DNS servers. **dig** stands for **domain information groper**, and it displays the actual messages that were received from DNS servers. You can decipher the output for yourself now that you know what a DNS message looks like.

As always, we encourage you to read the [dig manpage](#) and explore the command for yourself!

---

We have now studied the Internet’s directory in detail. Let’s move on to another protocol. See you in the next lesson!



# BitTorrent

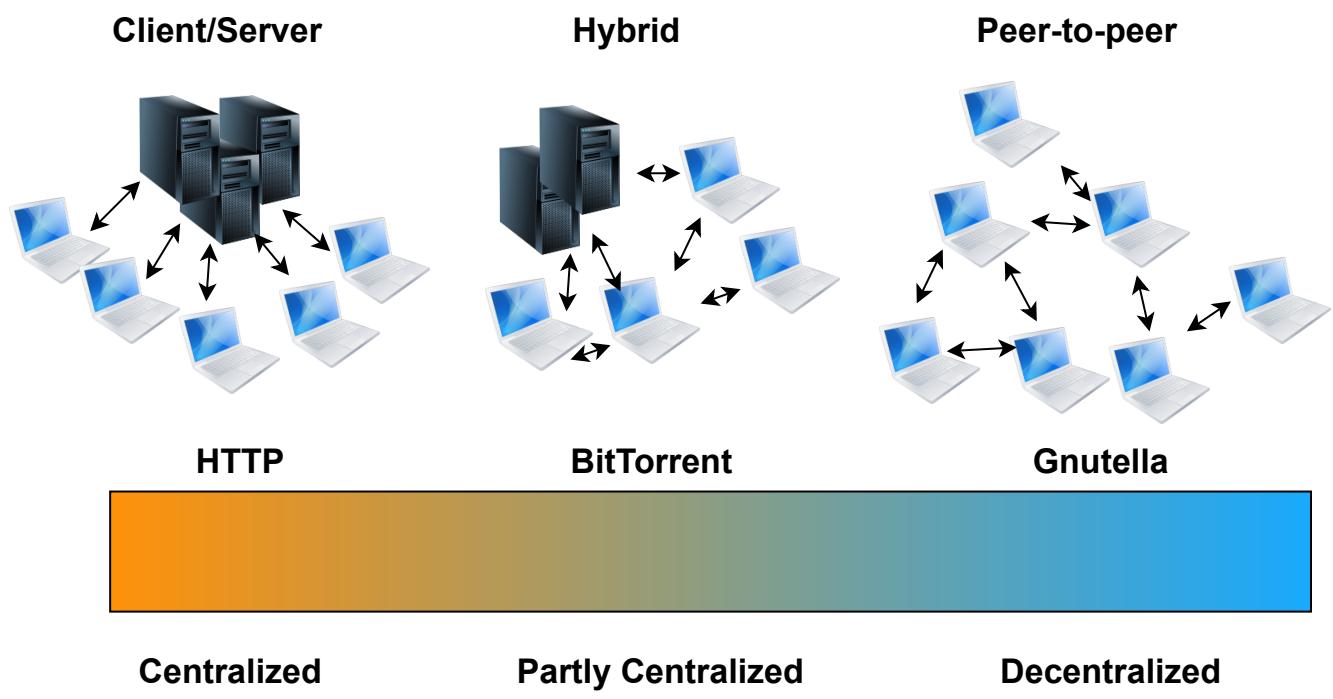
BitTorrent is a key protocol and has millions of users simultaneously and actively sharing and downloading hundreds of thousands of files of all types: music, movies, books, and so on.

## WE'LL COVER THE FOLLOWING

- Overview of BitTorrent
- How It Works
  - Trackers and torrent files
  - A Simplified BitTorrent Session
- Swarming
- Tit-for-tat



We have already had an overview of what the [Peer-to-peer](#) and [Hybrid](#) architectures are. BitTorrent falls more in the **hybrid** category than pure P2P. Here it is on a spectrum of decentralization:



On a spectrum of decentralization, BitTorrent falls in the middle.

# Overview of BitTorrent #

BitTorrent is a protocol for peer-to-peer file sharing. A **BitTorrent Client** is an application that uses this protocol.

Since BitTorrent is based on a hybrid architecture, it retains some centralized components.

- For example, a **central controller** that maintains a list of participating nodes is involved.
- But the centralized component is not involved in resource-intensive operations. So there will never be too much load on it.
- Data is instead **downloaded or uploaded directly to and by peers**.
- The file is first supplied to a peer in pieces called chunks, and then they also distribute the file to other peers.
- This is sometimes called a **peer-assisted** system.

## How It Works #

### Trackers and torrent files #

How do clients find peers to connect to? Well, clients connect to a special tracker node first. The tracker responds with the IP and the port of a few other peers who are downloading the same file.



**Note:** Modern BitTorrent clients are trackerless and use a Distributed Hash Table instead, but that's beyond the scope of this course.

So clients can find peers through trackers. But how do clients find the tracker in the first place? Clients begin by downloading a ‘torrent file’ from a web server which has the URL of the tracker. The torrent file also contains a SHA1 hash of each file chunk. Can you guess why?

## A Simplified BitTorrent Session #

1. Download the ‘torrent file’ from a web server.

2. Connect to the tracker and get a list of peers.

3. Connect to the peers - initially as a 'leecher.'

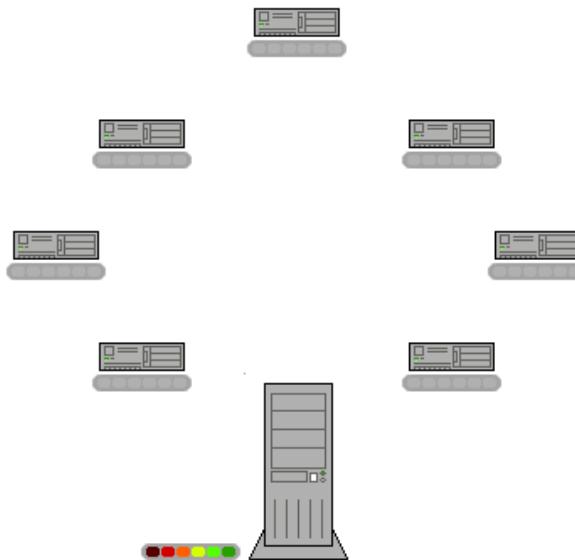
4. While the file is not yet fully downloaded:

- Advertise to peers which blocks are available locally.
  - Request blocks from peers.
  - Compare hash of downloaded blocks to the hash in the torrent file.
- Can you guess why?

5. Turn into a 'seeder,' i.e., continue uploading to peers without downloading.

## Swarming #

Study the following animation to get an idea of how a file is distributed in chunks. Note that it starts with one peer that has the file in its entirety and then the peers start distributing the file to each other.



[CC BY-SA 3.0 via Wikimedia Commons](#)

Once a peer has the entire file it can choose to leave (selfishly) or choose to stay as a seeder.

The distinct chunks of the file are represented by different colors in the diagram above, as are the bits being transferred. The initial blank gray boxes represent that none of the machines has any bit of the file. Eventually, all the machines have the entire file.

So what this protocol needs really is to give peers the incentive to upload. That is where the **tit-for-tat** scheme comes in.

## Tit-for-tat #

- Every ten seconds, a peer in the network will calculate which four peers are supplying data at the **highest rate** to it. It will then supply data to them in return. These 4 peers are said to be **unchoked** in the sense that they are now receiving data in return.



**Note:** This list of top four peers may change every 10 seconds.

- A peer in the network will randomly pick another peer every thirty seconds and supply data to them. The best-case scenario would be that the peer becomes one of the randomly picked peer's top 4 suppliers. Naturally, that random peer would start supplying data in return. Then if the randomly picked peer is sending data at a fast enough rate, it may also become part of the peer's top four suppliers. In other words, two peers partnered randomly will continue working with each other if they are satisfied with the trading. This randomly picked partner is said to be **optimistically unchoked**.

The result of this scheme is that everyone has an incentive to upload. The scheme is an instance of an old successful idea that stems from [Axelrod's tournament](#).

### Quiz on BitTorrent

1

What category does BitTorrent fall in?

COMPLETED 0%

1 of 2



That concludes our study of application layer protocols! Let's move on to the next layer at last.

# What Is the Transport Layer?

We finished the application layer, and now we'll study the transport layer.

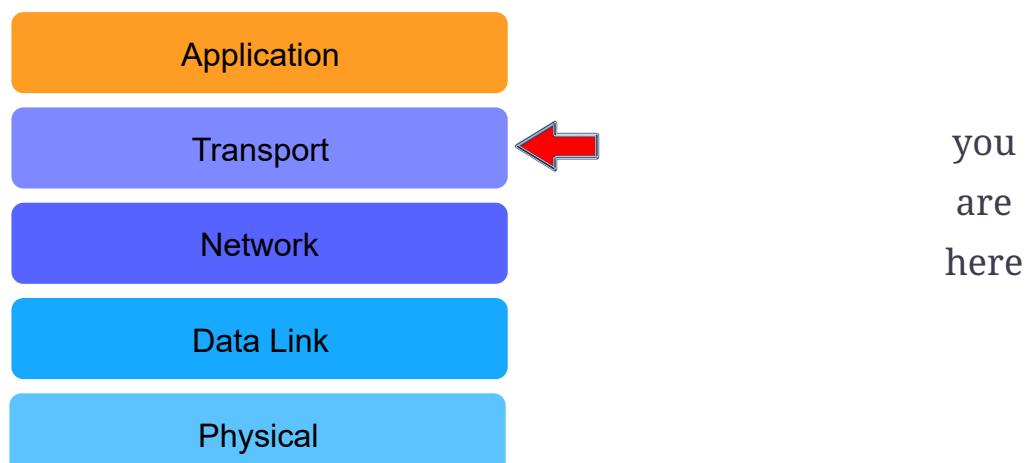
## WE'LL COVER THE FOLLOWING

- You Are Here!
- Key Responsibilities of the Transport Layer
  - The Post Analogy
- Where It Exists
- Transport Layer Protocols
- Quick Quiz!



## You Are Here! #

Let's zoom out and have a look at the big picture.



## Key Responsibilities of 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 Post Analogy #

Imagine you post a package across the world. Presumably, a ship or an airplane would carry the message to the relevant country. However, the post system of that country would take it to the relevant address. The **plane/ship** is the *network layer* and the **post system** is the *transport layer*.

Have a look at the slides for a clearer explanation.



Host 1



Host 2

suppose two end-systems are communicating with each other on Skype via the internet

1 of 4



Host 1



Host 2

and one end-system sends off a message to the other

2 of 4



Host 1



Host 2



however, delivering the message to the relevant application on the other end system is handled by the transport layer



Here are some other responsibilities of the transport layer.

- **Logical application-to-application delivery**, the transport layer makes it so that applications can address other applications on other end-systems directly. This is true even if it exists halfway across the world. So it provides a layer of **abstraction**.
- **Segments data**. The transport layer also divides the data into manageable pieces called ‘segments’ or ‘datagrams.’
- **Can allow multiple conversations**. Tracks each application to application connection or ‘conversation’ separately, which can allow multiple conversations to occur at once.
- **Multiplexes & demultiplexes data**. It ensures that the data reaches the relevant application *within* an end-system. So if multiple packets get sent to one host, each will end up at the correct application.

## Where It Exists #

- 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.

Therefore, the transport layer and its protocols **reside on end-systems!** It is also the first layer in the OSI reference model (from the bottom) that distinguishes between applications.

## 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. We'll discuss the intricacies of each in detail in upcoming chapters, but here is a quick overview.

### TCP

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

### UDP

- Does not ensure in-order delivery of messages that we call ‘datagrams.’
- Detects any modifications that may have been introduced in the packets during delivery but does not correct them by default.
- Does not ensure reliable delivery.
- Generally faster than TCP because of the reduced overhead of ensuring uncorrupted delivery of packets in order.
- Applications that use UDP include: Domain Name System (DNS), live video streaming, and Voice over IP (VoIP).

# Quick Quiz! #

1

The transport layer in the OSI reference model uses the services of \_\_\_\_\_ layer.

COMPLETED 0%

1 of 2



In the next lesson, we'll have a more in-depth look at multiplexing and demultiplexing!

# Multiplexing and Demultiplexing

Let's discuss how the transport layer handles so many simultaneous connections over one network!

## WE'LL COVER THE FOLLOWING



- What are Multiplexing & Demultiplexing?
  - What is Demultiplexing?
  - What is Multiplexing?
- How Do They Work in the Transport Layer?
- Quick Quiz!

## What are Multiplexing & Demultiplexing? #

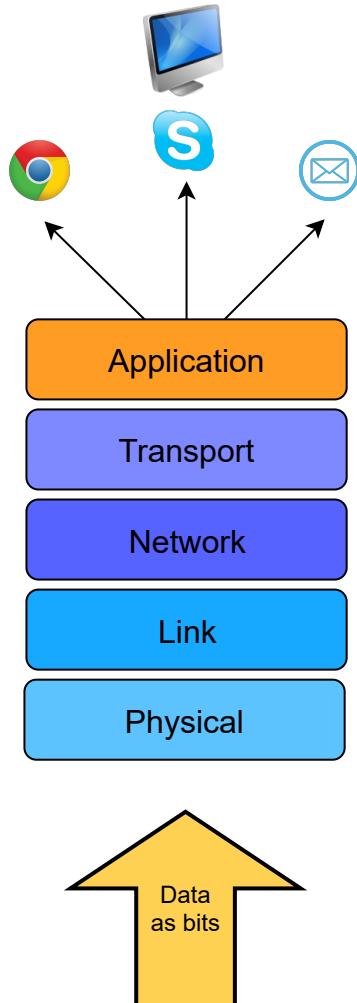
End-systems typically run a variety of applications at the same time. For example, at any given time a browser, a music streaming service, and an email agent could be running.

So how does the end-system know which process to deliver packets to? Well, that's where the transport layer's **demultiplexing** comes in.

## 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.

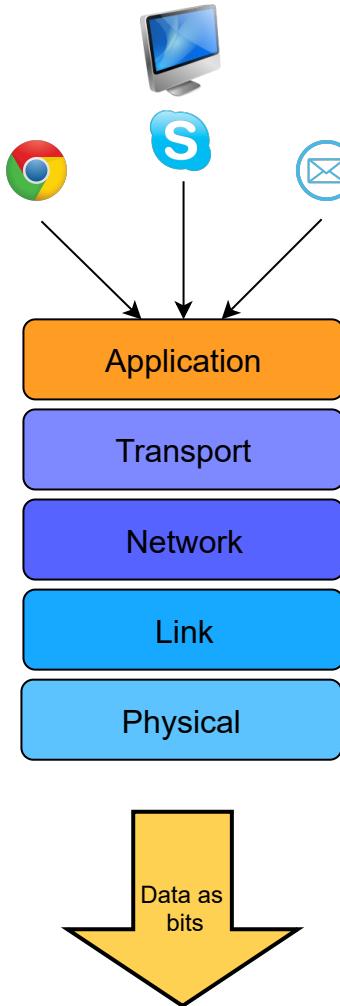


End systems run many programs at once which leaves us with the question: what process to deliver which packet to?

## What is Multiplexing? #

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

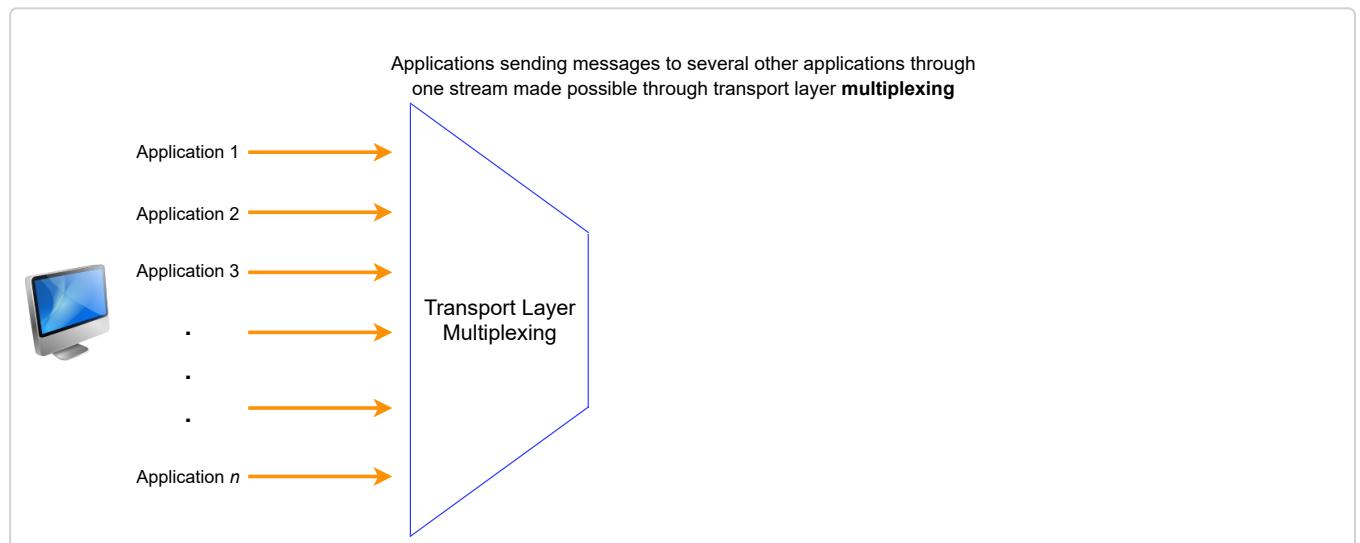
An analogy would be when several packages to several different locations are mailed out from one house.

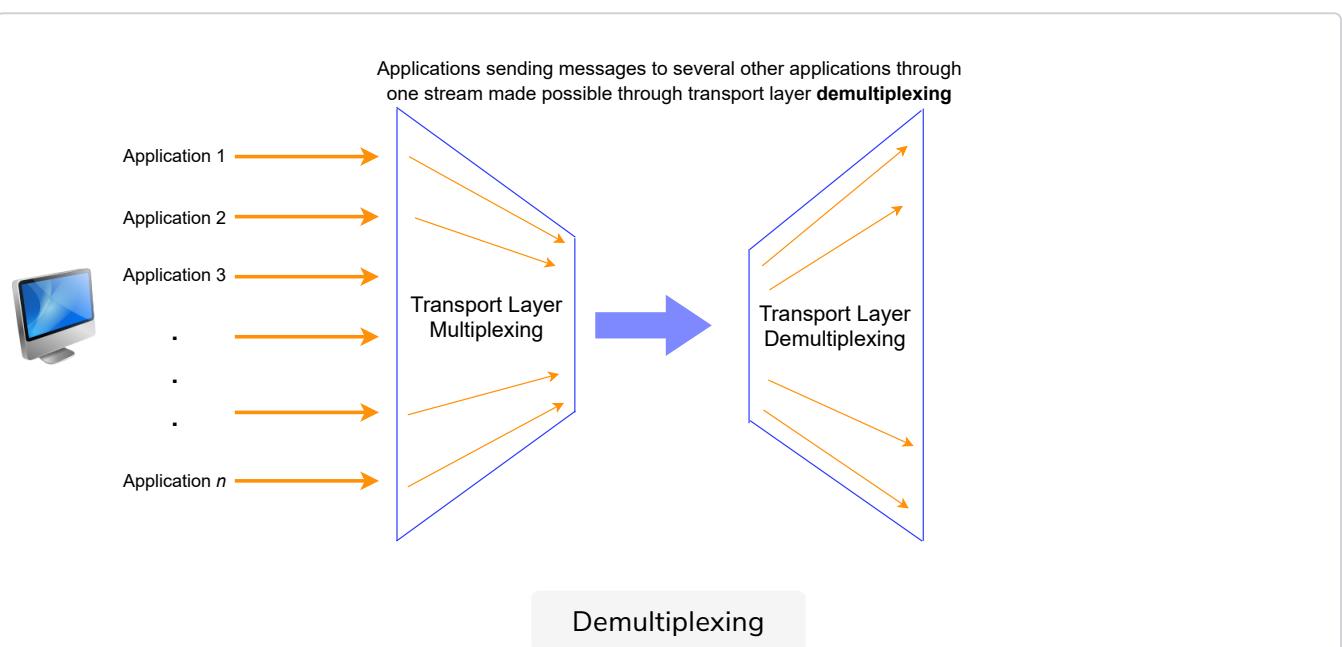
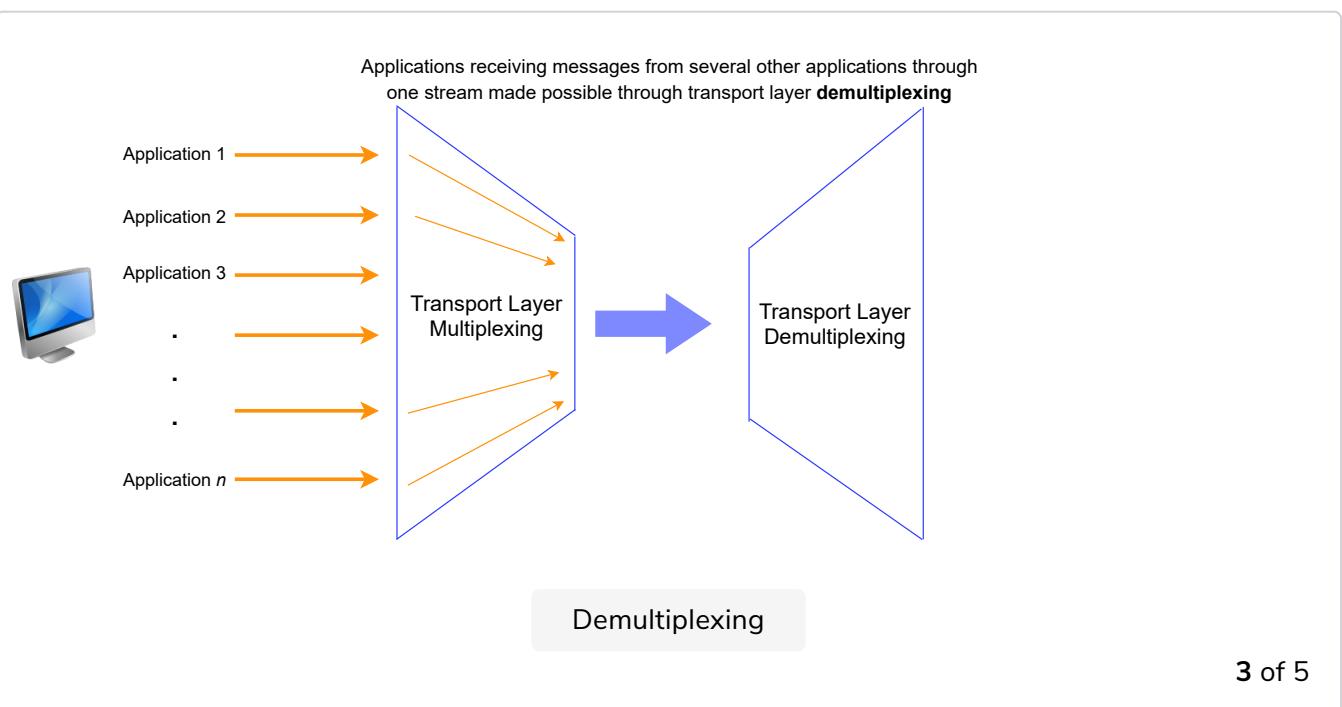
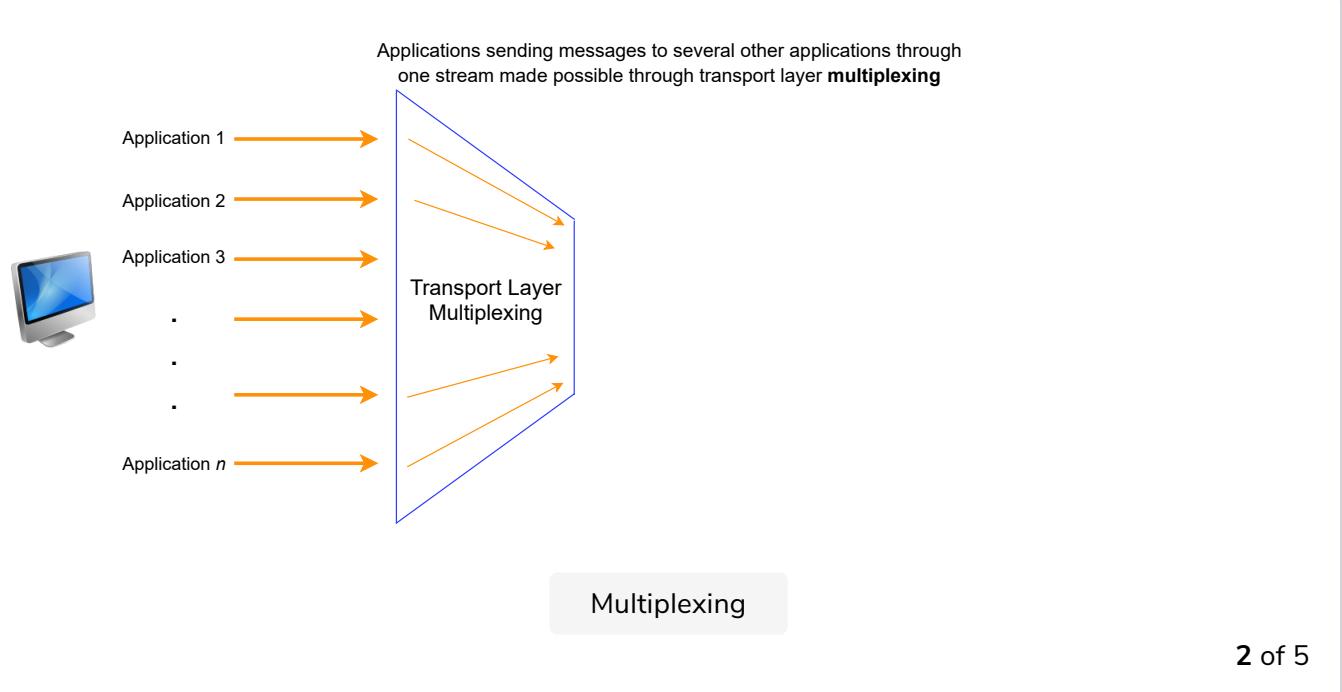


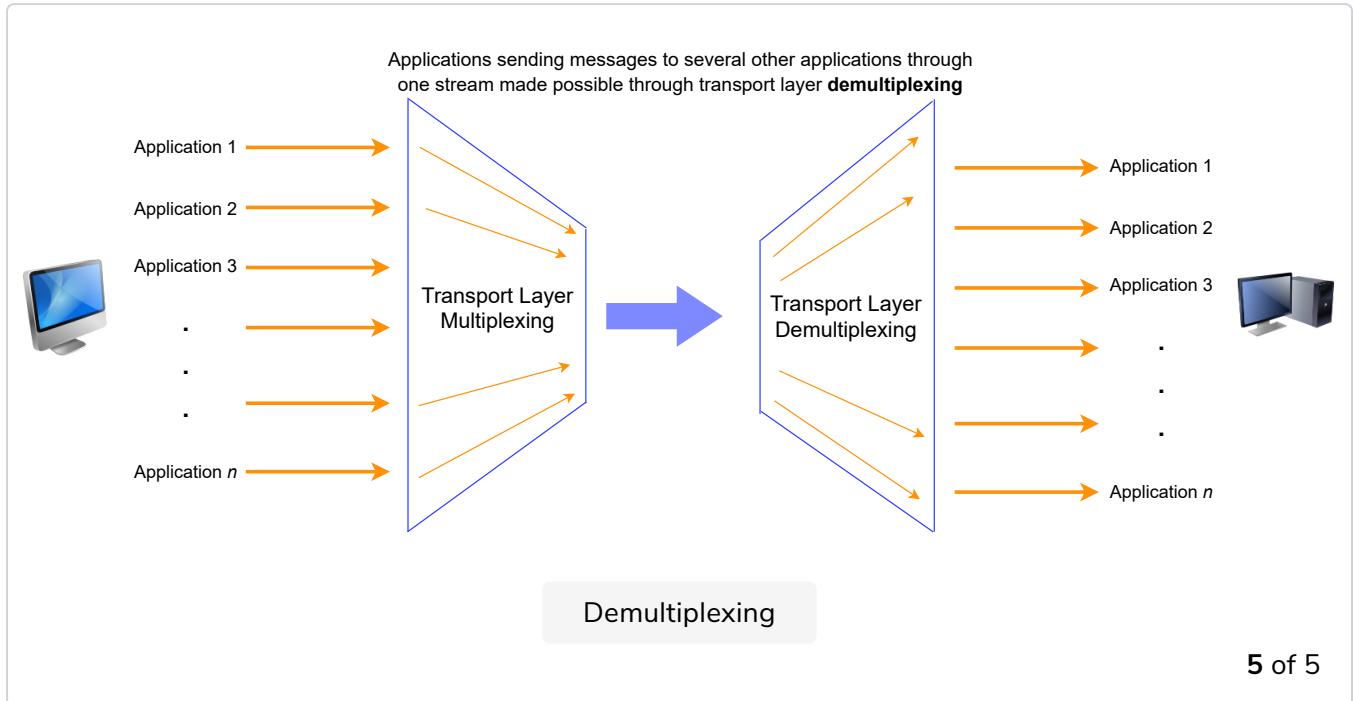
End systems can be talking to many other applications at once which leaves us with the question:  
how to send messages such that they arrive to the correct process?

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.

Here's a simplified view of what multiplexing and demultiplexing are.







## 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.

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.

## Quick Quiz! #



The transport layer in the TCP/IP reference model uses \_\_\_\_\_ to multiplex/demultiplex messages for different applications.

COMPLETED 0%

1 of 1



---

Let's look at connectionless multiplexing and demultiplexing in the next lesson!

# Multiplexing & Demultiplexing in UDP

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

## WE'LL COVER THE FOLLOWING



- Ports
- Multiplexing & Demultiplexing in UDP
- On Port Assignment in UDP
- Quick Quiz!

## Ports #

Here's a quick refresher on what ports are because that needs to be crystal clear in order for you to understand multiplexing and demultiplexing.

- **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.



Sockets have associated port numbers!

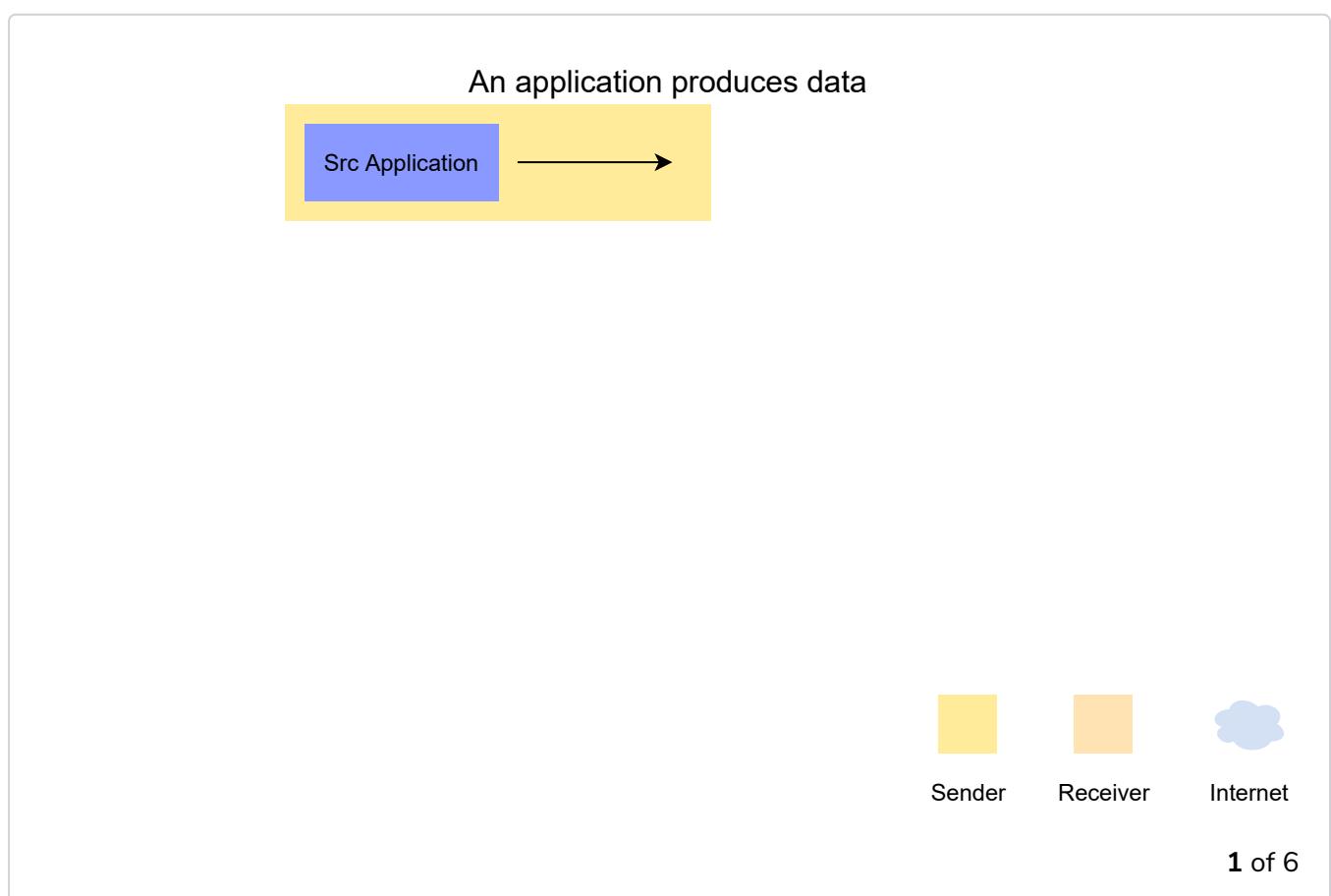
- Out of these, the port numbers 0 – 1023 are **well-known** and are reserved for certain standard protocols. Port 80, for instance, is reserved for HTTP whereas port 22 is reserved for SSH.

- Refer to page 16 of [RFC 1700](#) for more details regarding what port number is assigned to what protocol.

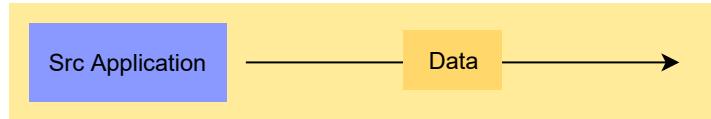
## Multiplexing & Demultiplexing in UDP #

- 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.

Here are some slides to give you a quick overview of what happens.

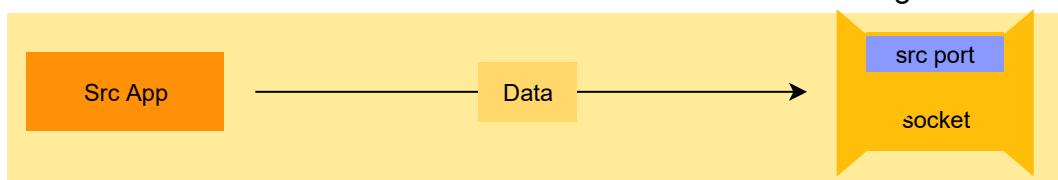


An application produces data



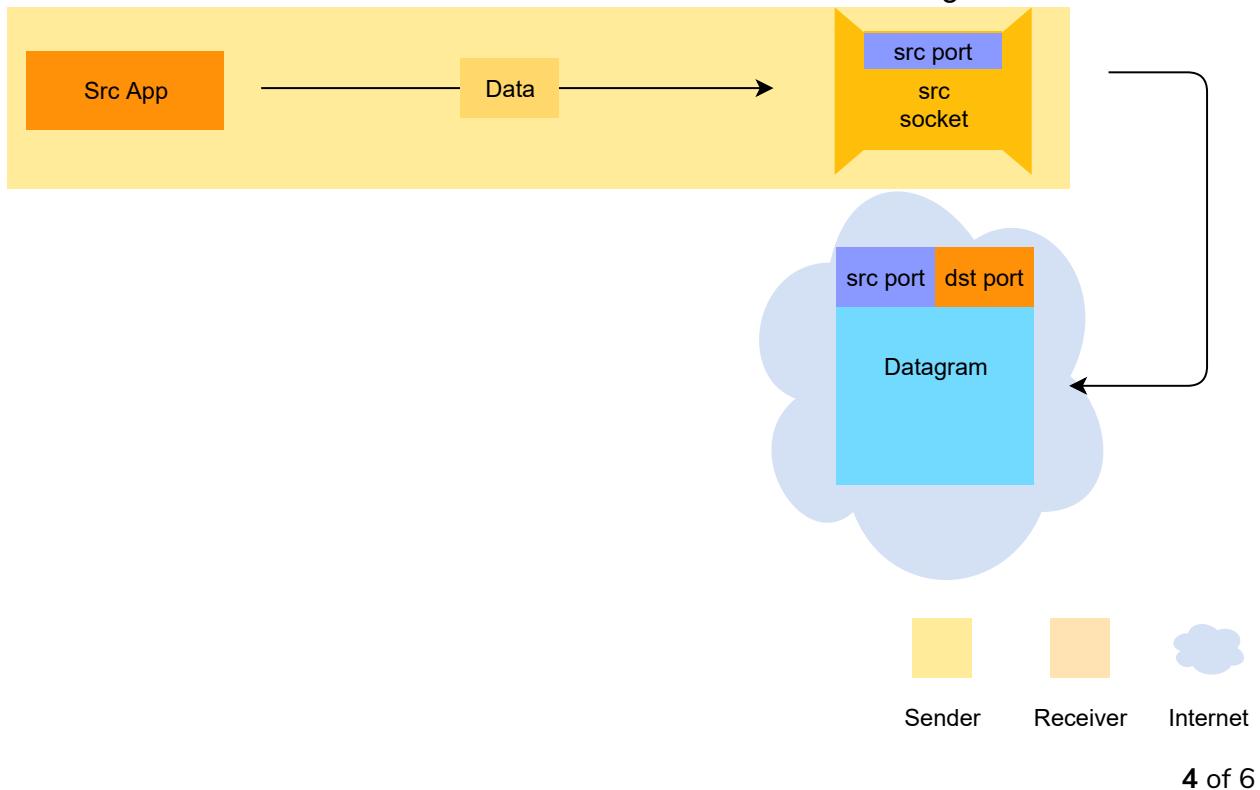
2 of 6

It writes the data out to its socket and it turns into a datagram

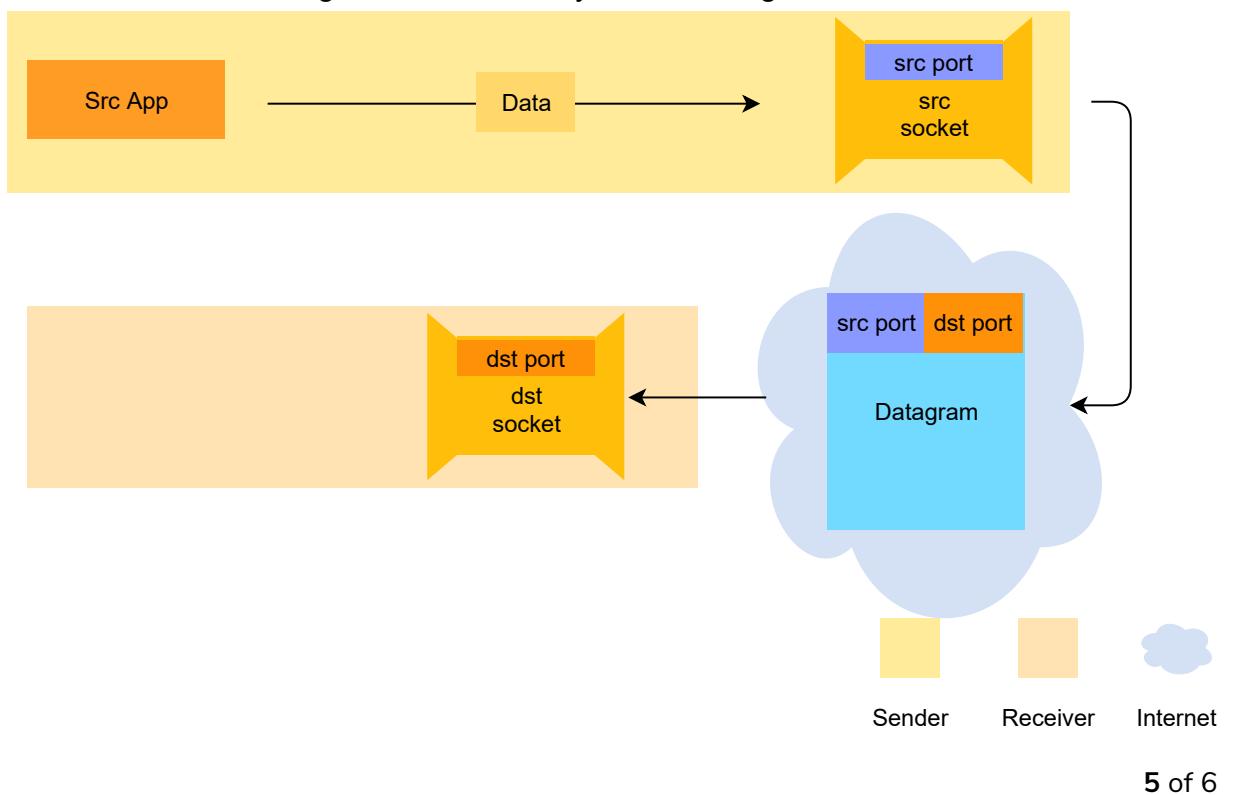


3 of 6

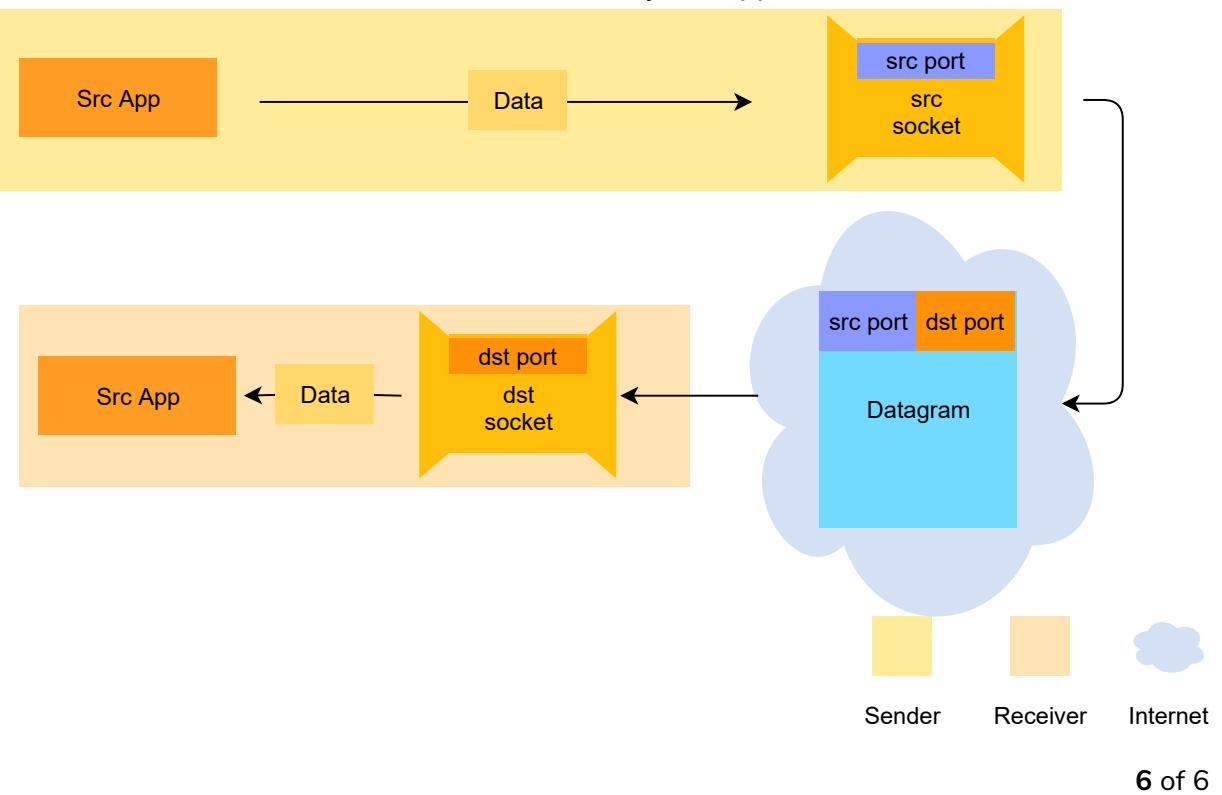
It writes data out to its socket and turns it into a datagram



The datagram is received by the receiving end's socket



The data is then received by the application



## 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.

## Quick Quiz! #

COMPLETED 0%

1 of 3



Let's look at the principles of reliable data transfer in the next lesson. This is key to building a good foundation for later lesson on TCP!

# Introduction to Congestion Control

In this lesson, we'll look at congestion control!

## WE'LL COVER THE FOLLOWING

- Bandwidth Allocation Principles
  - Should Allocation Be on a per Host or per Connection Basis?
  - Efficiency & Power
    - Bursts of Traffic
    - Transmission Threshold
- Quick Quiz!

## What Is Congestion?

When more packets than the network has bandwidth for are sent through, some of them start getting dropped and others get delayed. This phenomenon leads to an overall drop in performance and is called **congestion**.



Traffic congestion at Times Square, NYC.

This is analogous to vehicle traffic congestion when too many vehicles drive on the same road at the same time. This slows the overall traffic down.

## How Do We Fix It?

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**.

Congestion control is really just congestion avoidance. Here's how the transport layer controls congestion:

1. It 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. Let's also look at a few principles of bandwidth allocation before moving on.

## Bandwidth Allocation Principles #

### Should Allocation Be on a per Host or per Connection Basis? #

Before we get into the key principles of bandwidth allocation, we have to answer the question: should bandwidth be allocated to each host or to each *connection* made by a host?

Not all hosts are created equal; Some can send and receive at a higher data rate than others. Furthermore, if the bottleneck bandwidth was allocated equally to all hosts, some of them wouldn't be able to use the bandwidth to its full capacity and some wouldn't have enough. For example, if an Internet-enabled doorbell and a busy server had the same bandwidth, the doorbell would have too much and the server would likely not have enough.

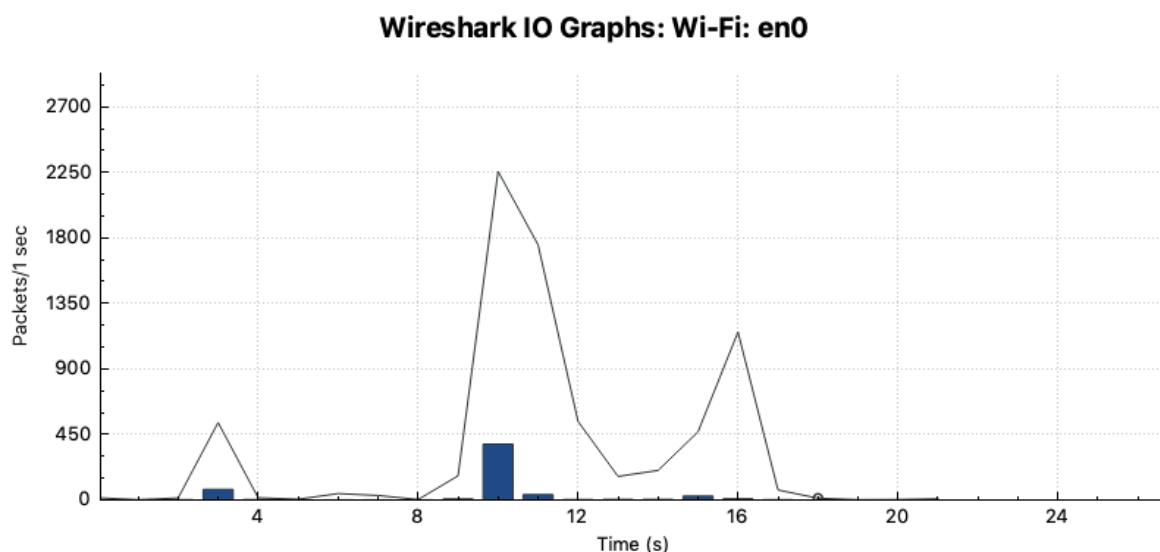
The per-connection allocation, on the other hand, can be exploited by hosts opening multiple connections to the same end-system.

Usually, bandwidth is allocated per connection.

## Efficiency & Power #

### Bursts of Traffic #

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.,  $\frac{200}{4} = 50$  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. Why? Because real traffic is transmitted in **bursts** and not in one continuous stream. Have a look at the following plot of a Wireshark traffic capture for a clearer picture.



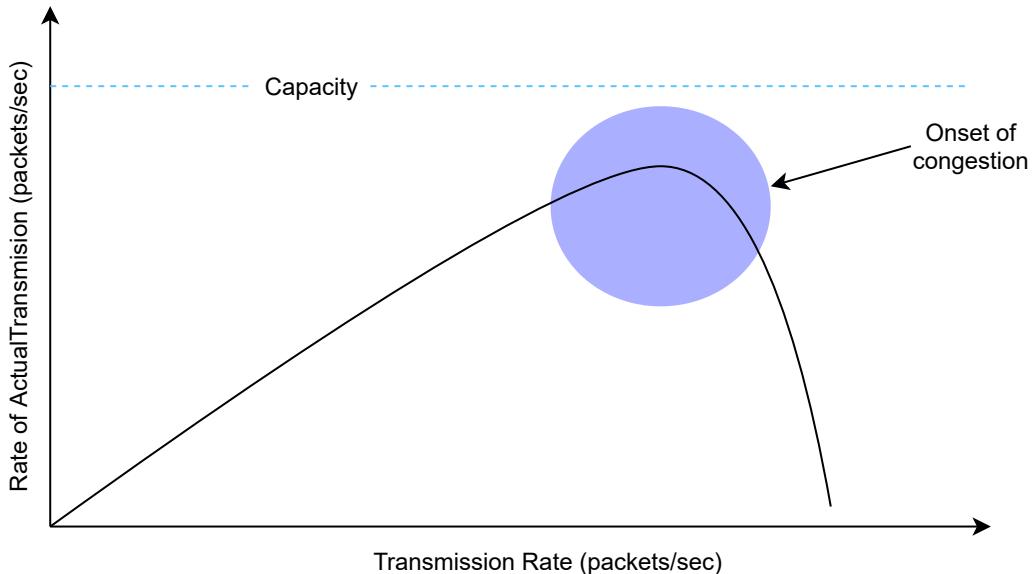
Plot of a Wireshark traffic capture depicting three prominent bursts of traffic

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!**

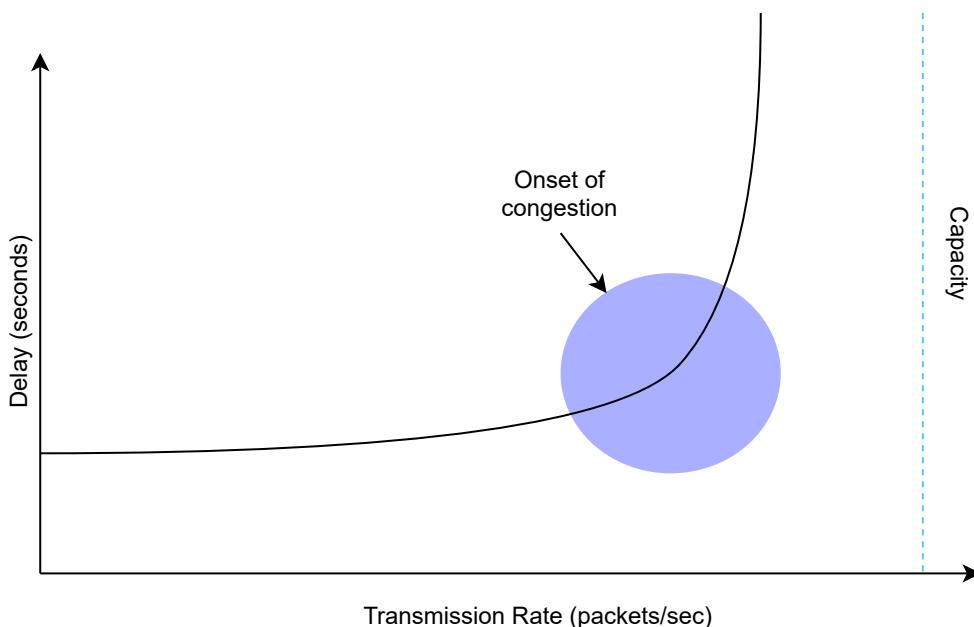
### Transmission Threshold #

The graph below depicts 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 effect of the increase in transmission rate on useful traffic

The following graph shows that 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.



The effect of the increase in transmission rate on delay



**Note:** **congestion collapse** occurs when all end-systems are sending a lot of traffic but nothing is being received, for example, when all or most packets are dropped. There are a few causes for this, including but not limited to **Spurious retransmissions**. Spurious retransmissions occur

when a retransmission timer times out for packets that are not lost but

have not yet reached the destination. So, much of the network's bandwidth ends up being consumed by a small number of packets.

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

Lastly, according to [Kleinrock's 1979 paper](#), the optimal transmission rate is such that *power*, given by the following equation, is maximized:

$$\frac{\text{Power}}{\text{Delay}} = \frac{\text{Transmission Rate}}{\text{Delay}}$$

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

## Quick Quiz! #

1

If the applications increase sending rate, the throughput \_\_\_\_\_ at first, then \_\_\_\_\_.

We've now learned the first principle of congestion control: the entire bandwidth of a network should not be allocated. Let's continue with the next two in the next lesson!

# More on Principles of Congestion Control

Let's look at a couple of more key principles that congestion control algorithms adhere to!

## WE'LL COVER THE FOLLOWING ^

- Max-min Fairness
- Convergence
- Quick Quiz!

What we know so far is that congestion control schemes **must avoid congestion**. In practice, this means that the bottleneck link (the link with the lowest bandwidth) cannot be overloaded. This means on average, the sum of the transmission rate allocated to all hosts at any given time should be less than or equal to the bottleneck link's bandwidth.

Additionally, the congestion control scheme must be **efficient**. The bottleneck link is usually both a shared and an expensive resource. Usually, bottleneck links are wide-area links that are much more expensive to upgrade than the local area networks. The congestion control scheme should, therefore, ensure that such links 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.

## Max-min Fairness #

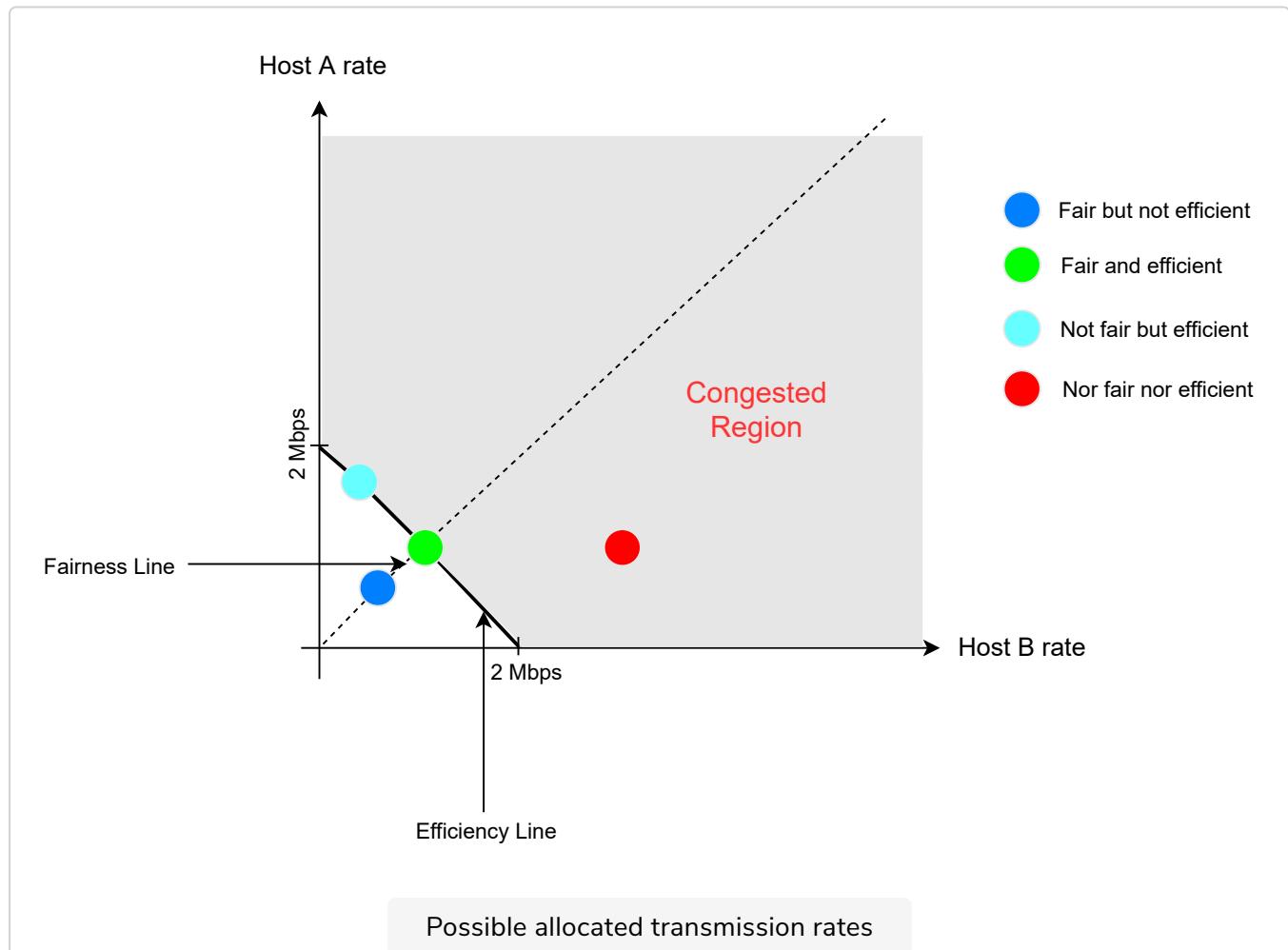
Furthermore, the congestion control scheme should be **fair**. Most congestion schemes aim at achieving **max-min fairness**. An allocation of transmission rates to sources is said to be **max-min fair** if:

1. No link in the network is congested
2. The rate allocated to a source  $j$  cannot be increased without decreasing the rate allocated to another source  $i$ , whose allocation is smaller than

the rate allocated to another source  $i$ , whose allocation is smaller than the rate allocated to the source  $j$ .

In other words, this principle postulates that **increasing the transmission rate of one end-system necessarily decreases the transmission rate allocated to another end-system with an equal or smaller allocation.**

To visualize the different rate allocations, it's useful to consider the graph shown below. Consider hosts A and B that share a bottleneck link.



In this graph, we plot the **rate allocated to host B** on the x-axis and we plot the **rate allocated to host A** on the y-axis . A point in the graph  $(r_B, r_A)$  corresponds to a possible allocation of the transmission rates. Since there is a 2 Mbps bottleneck link in this network, the graph can be **divided into regions**:

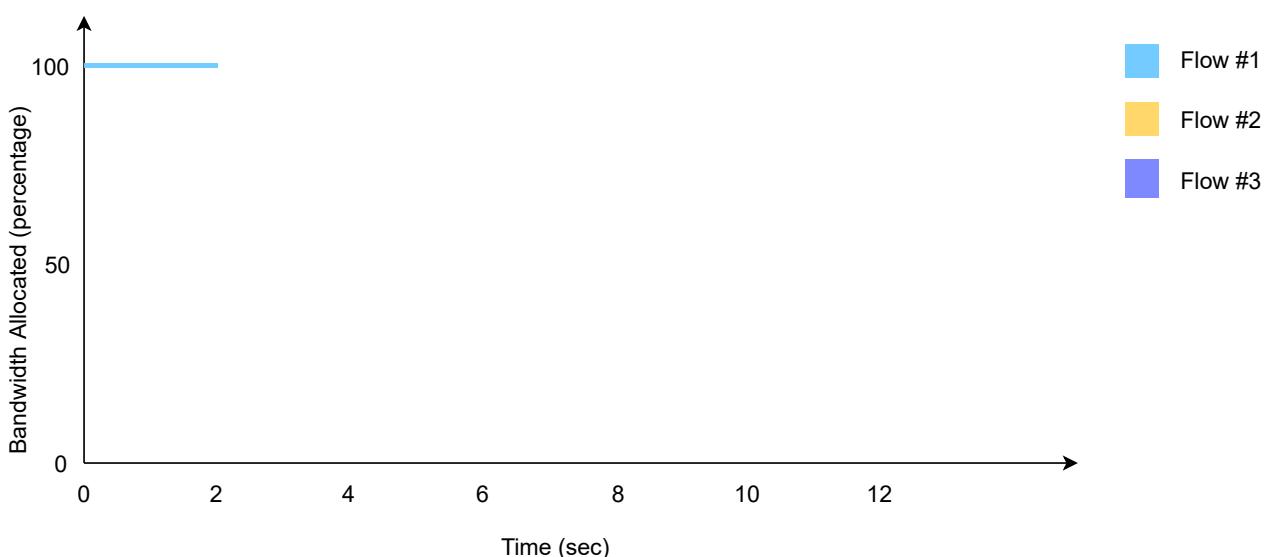
1. The white-colored part of the graph contains all allocations  $(r_B, r_A)$  such that the bottleneck link is not congested  $(r_A + r_B < 2)$ .
2. The right border of this region is the efficiency line or the set of allocations that completely utilize the bottleneck link  $(r_A + r_B = 2)$ .
3. Finally, the fairness line is the set of fair allocations.

Depending on the network, a max-min fair allocation may not always exist. In practice, max-min fairness is an ideal objective that cannot necessarily be achieved. When there is a single bottleneck link as in the example above, max-min fairness implies that each source should be allocated the same transmission rate.

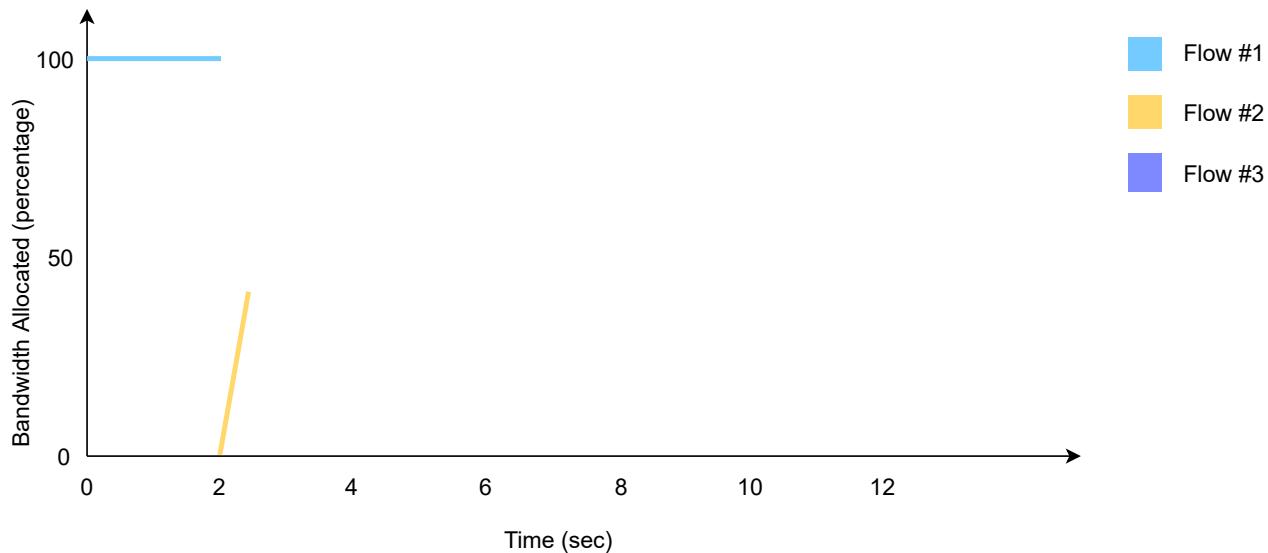
Ideally, the allocation should be both fair and efficient. Unfortunately, maintaining such an allocation with fluctuations in the number of flows that use the network is a challenging problem. Furthermore, there might be several thousands of TCP connections or more that pass through the same link.

## Convergence #

Additionally, bandwidth should be allocated such that it converges to a fair (a host does not hog all of it), and efficient value. This means it should not oscillate and most of it will be used. Furthermore, it should also change in response to changes in the demands of the network over time. Here are some slides that demonstrate this convergence.

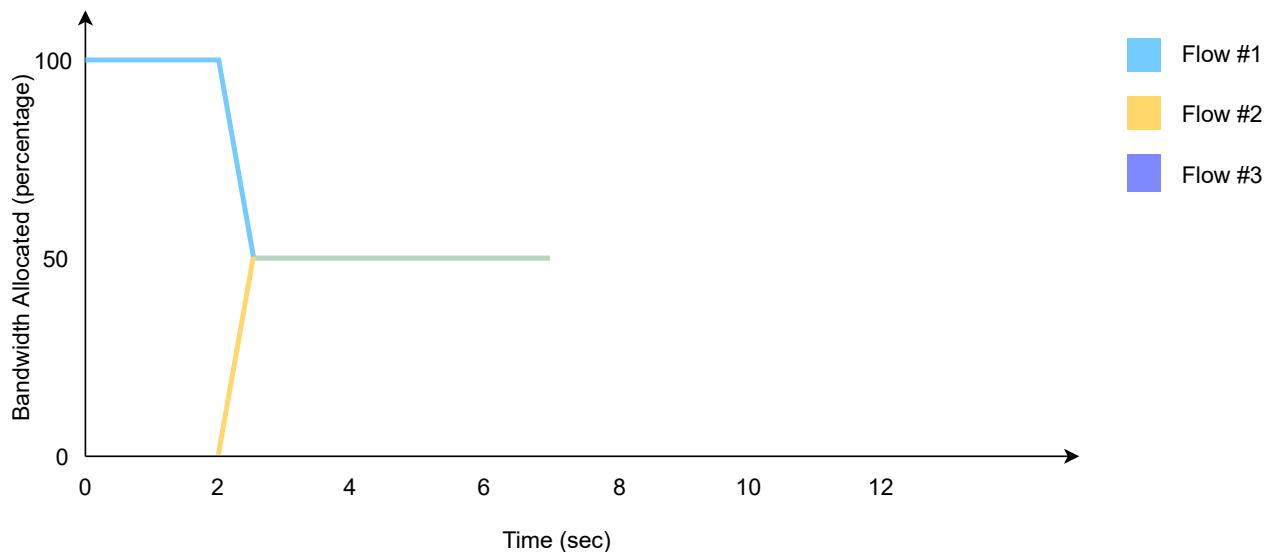


Flow #1 starts and continues for 2 seconds. Flow #1 is using all of the bandwidth.



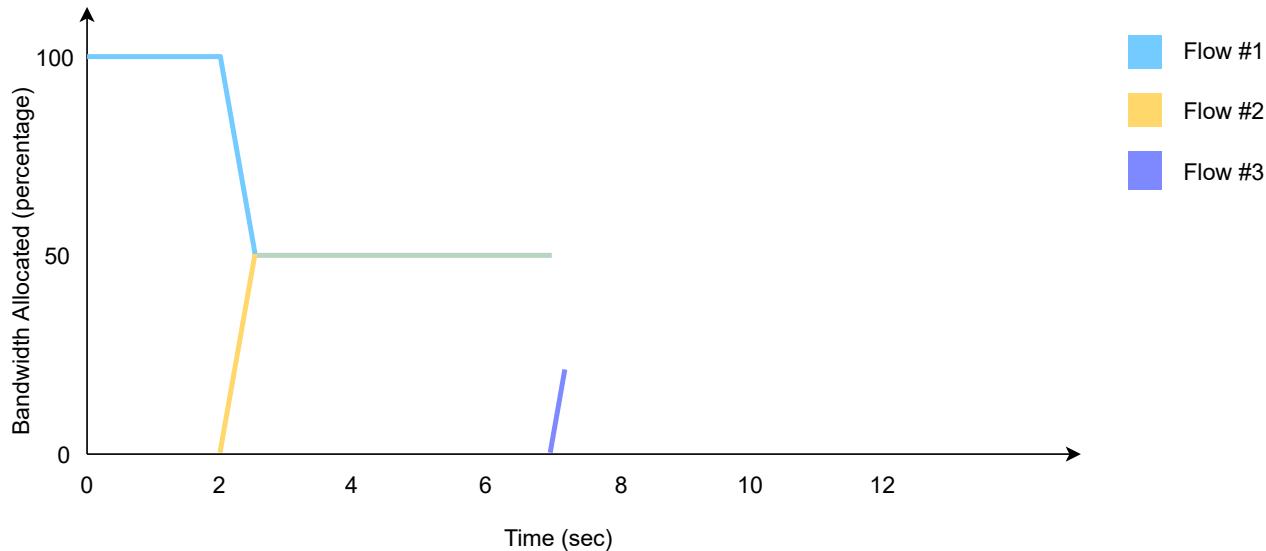
At the 2 second mark, another flow, flow #2 arrives

2 of 7



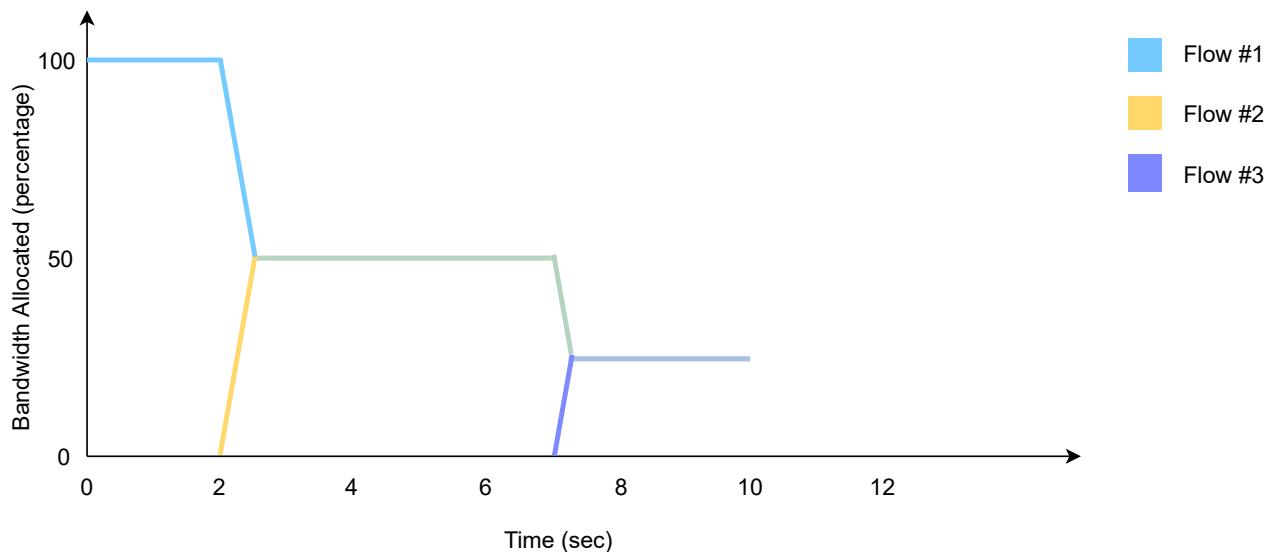
Flow #1 rapidly converges and drops its bandwidth usage to half of the available bandwidth. Both flows use 50% of the available bandwidth and continue for another 4 seconds or so.

3 of 7



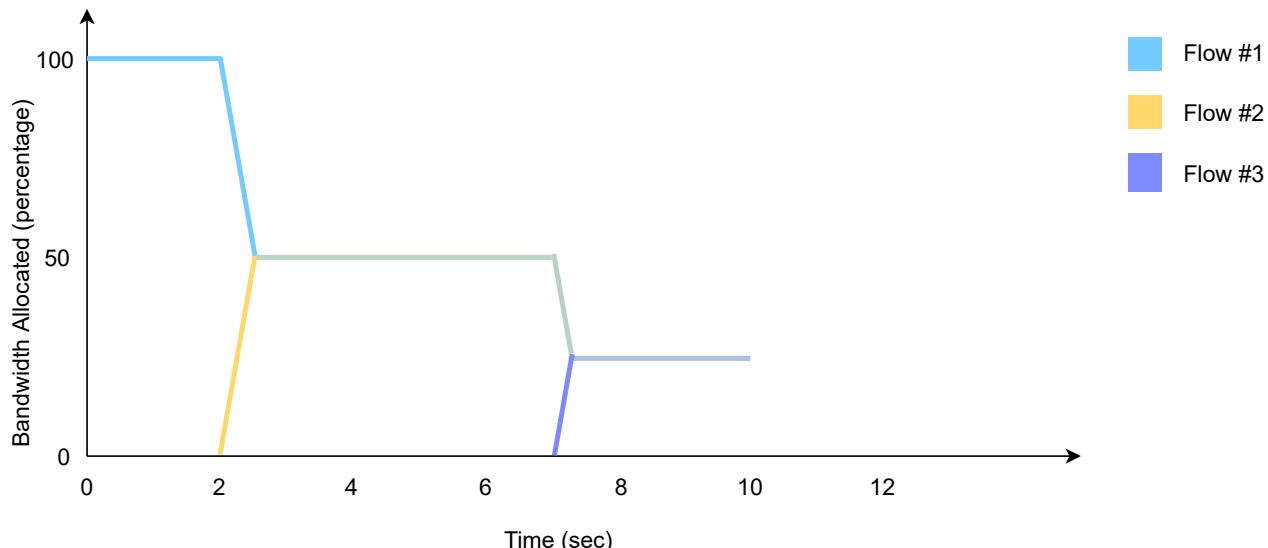
At the 7 second mark, another flow, flow #3 starts.

4 of 7



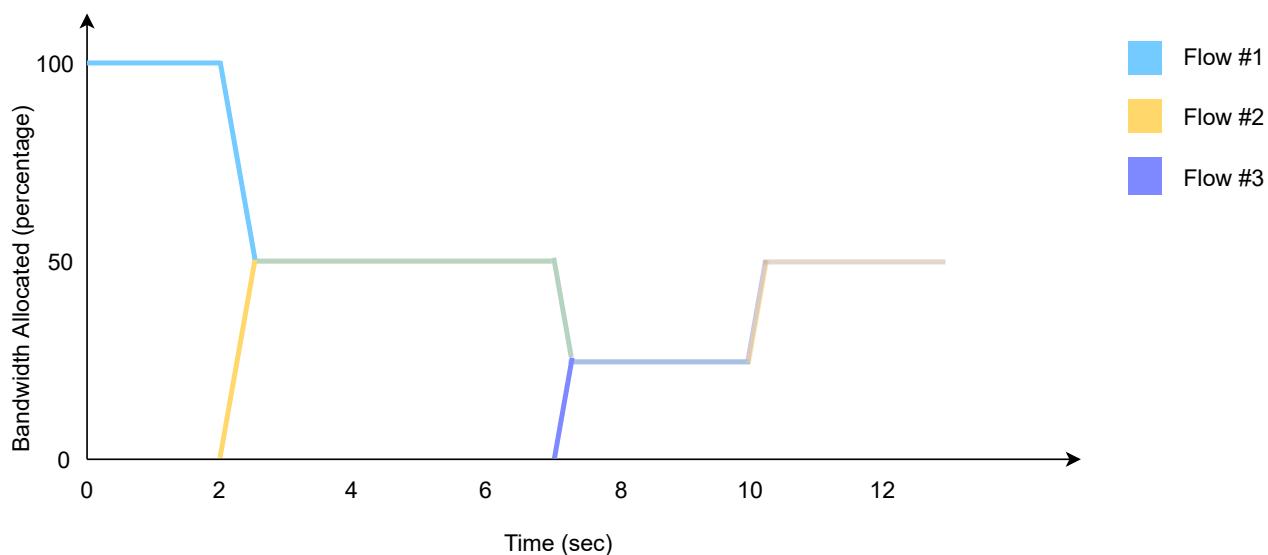
All flows converge to 1/3 of the bandwidth

5 of 7



At the 10-second mark, all flow #1 drops out.

6 of 7



At the 10-second mark, all flow #1 drops out, so flows #2 and #3 converge back to half the available bandwidth

7 of 7



**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

## Quick Quiz! #

1

In the context of congestion control, an efficient rate allocation is desirable because \_\_\_\_.

COMPLETED 0%

1 of 3



We now understand some basics of congestion control and bandwidth allocation. Now that we have an in-depth understanding of how UDP works, let's move on to perhaps one of the most important protocols of this entire course: TCP.

# Principles of Reliable Data Transfer

WE'LL COVER THE FOLLOWING

^

- Network Layer Imperfections
- Checksums
- Retransmission Timers
  - Limitations of Retransmission Timers
- Sequence Numbers
- Quick Quiz!

## Network Layer Imperfections #

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**

Let's look at some workarounds for these problems that the transport layer employs.

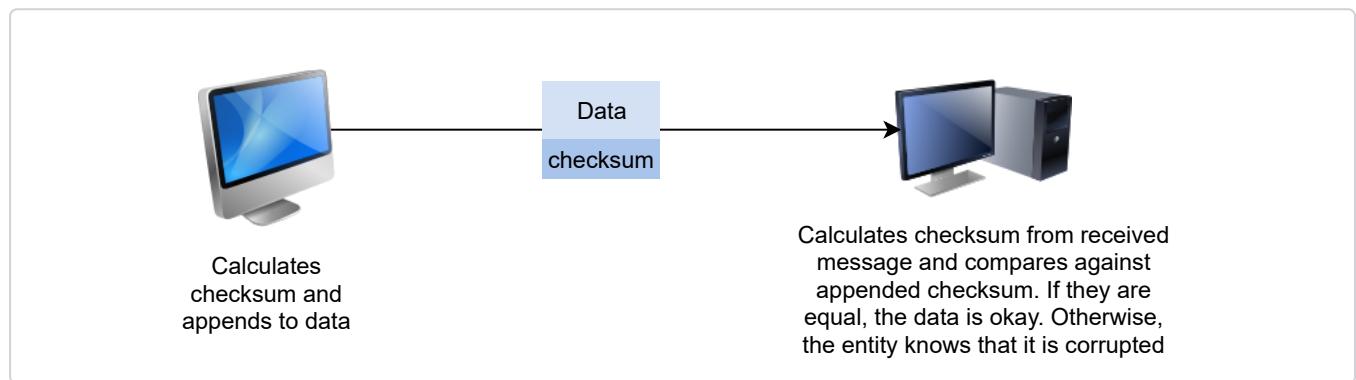
## Checksums #

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

A checksum can be based on a number of schemes. One possible scheme is an arithmetic sum of all the bytes of a segment. Checksums are computed by the

sender and attached with the segment. The receiver verifies it upon reception

and can choose what to do in case it is not valid. Quite often, the segments received with an invalid checksum are **discarded**.

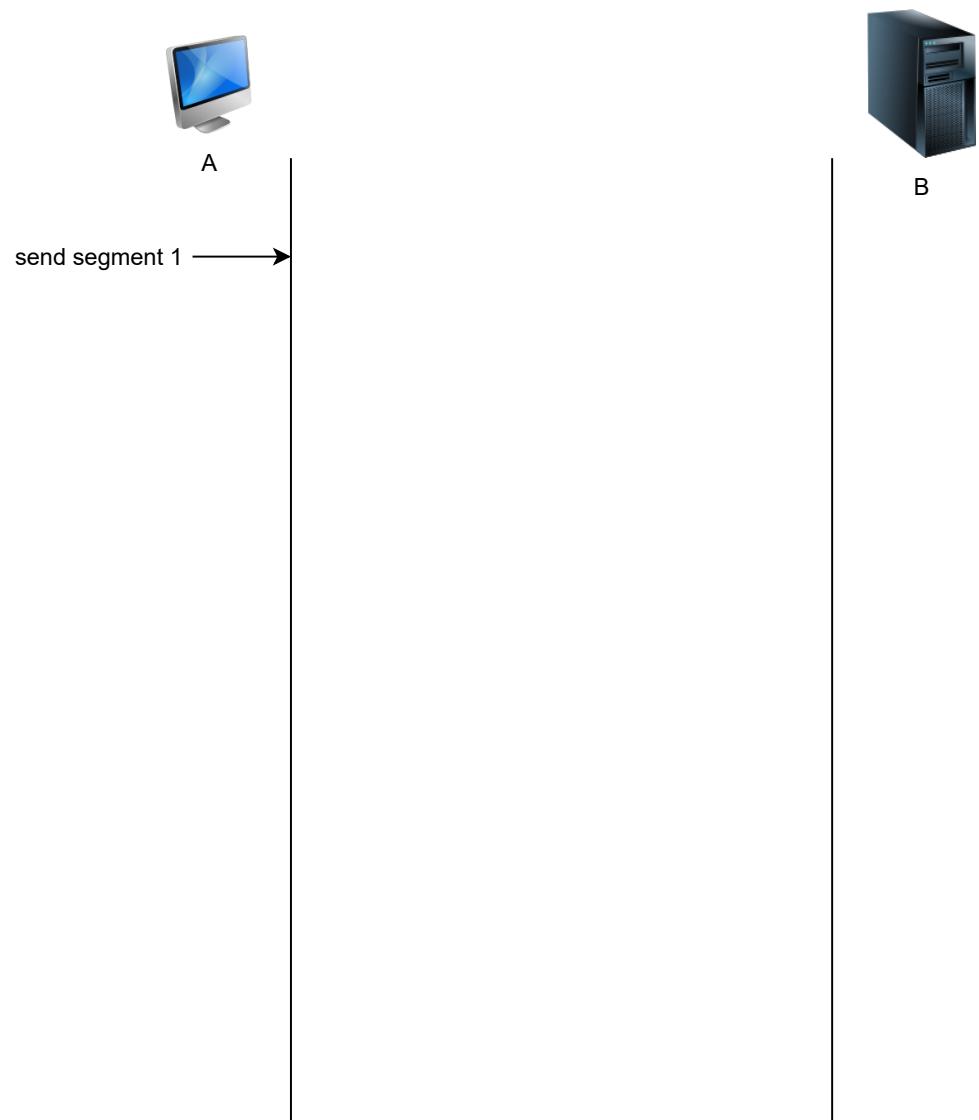


## Retransmission Timers #

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**.

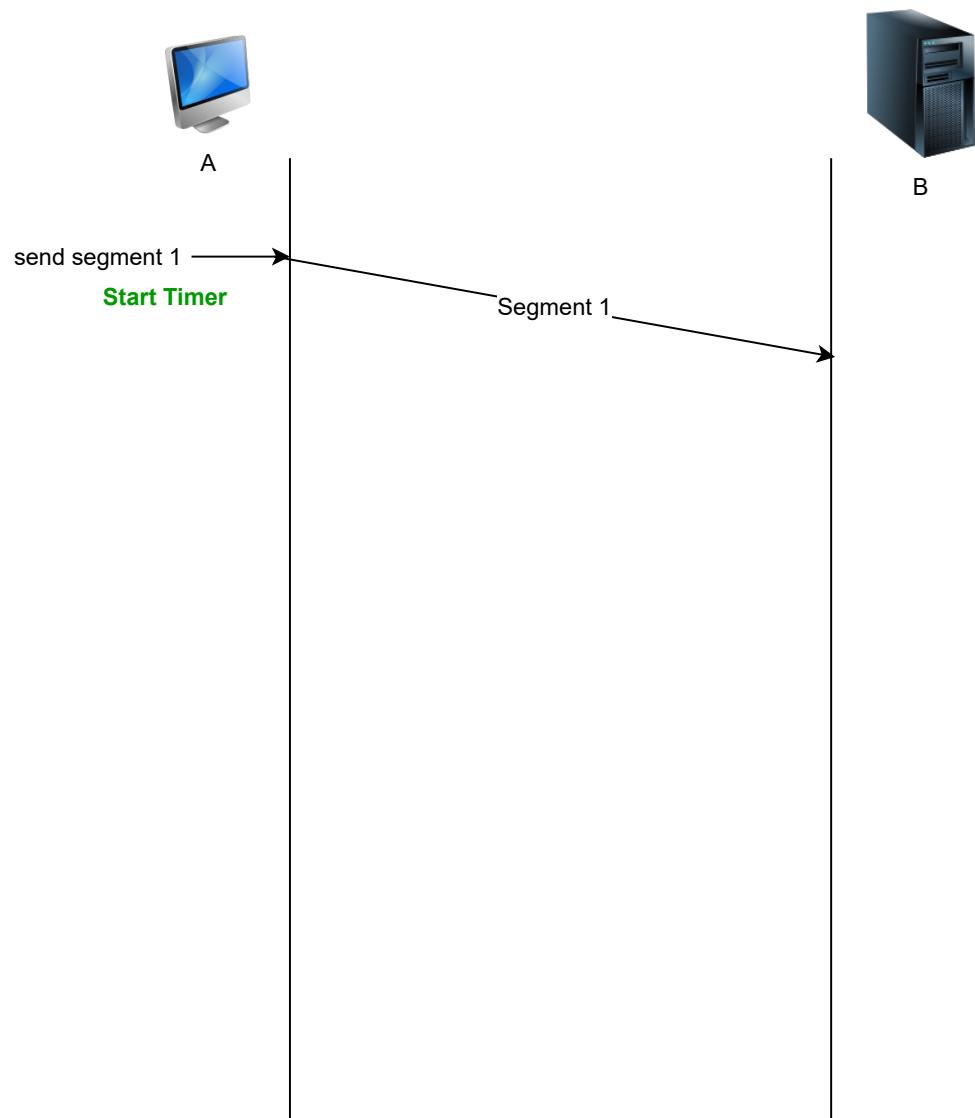
A retransmission timer starts when the sender sends a segment. The value of this retransmission timer should be *greater* than the **round-trip-time**, for example, the delay between the transmission of a data segment and the reception of the corresponding acknowledgment. Note that TCP sends an acknowledgment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

This is illustrated in the figure below:



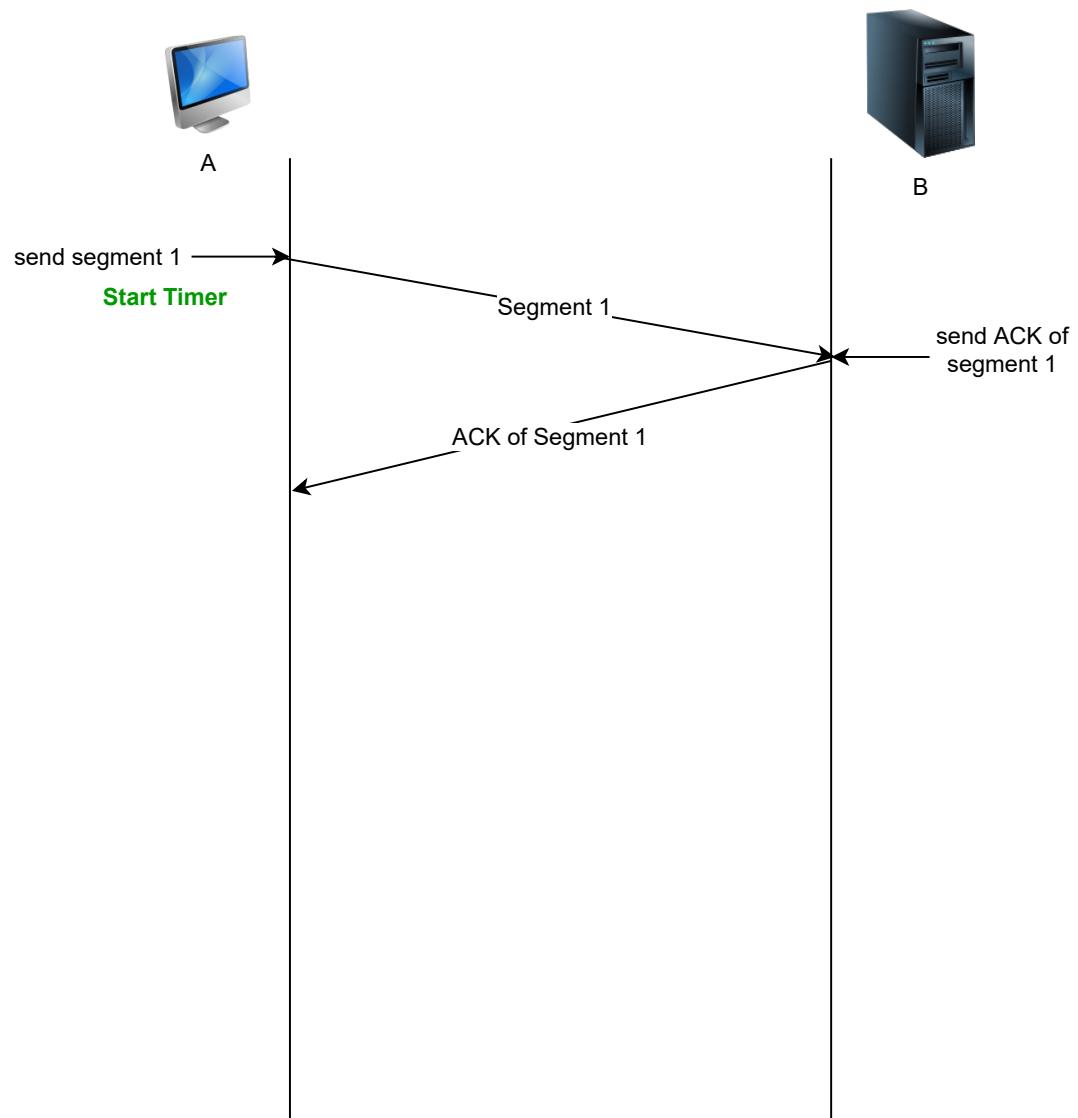
Using retransmission timers to recover from segment losses

1 of 7



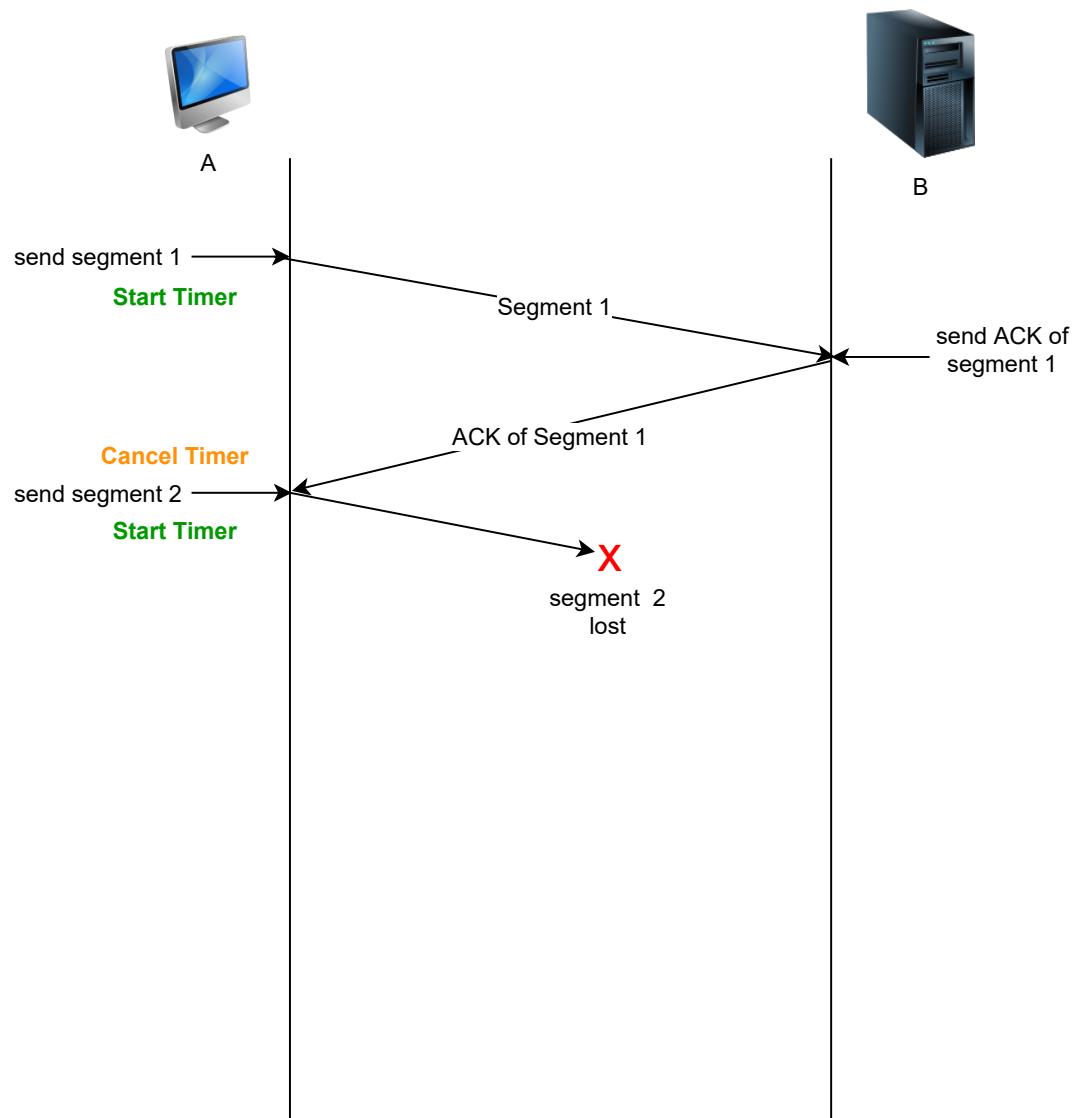
Using retransmission timers to recover from segment losses

2 of 7

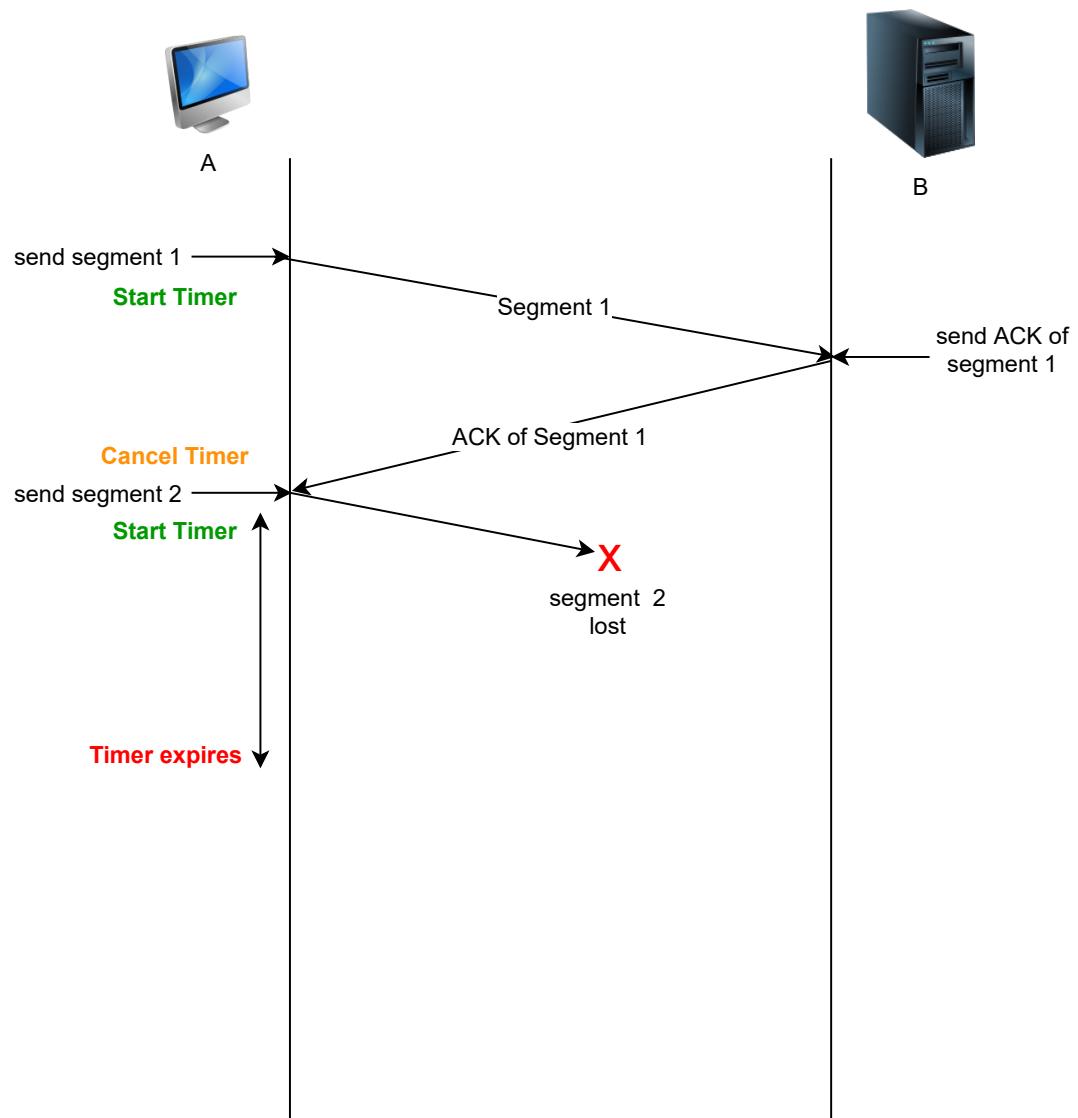


Using retransmission timers to recover from segment losses

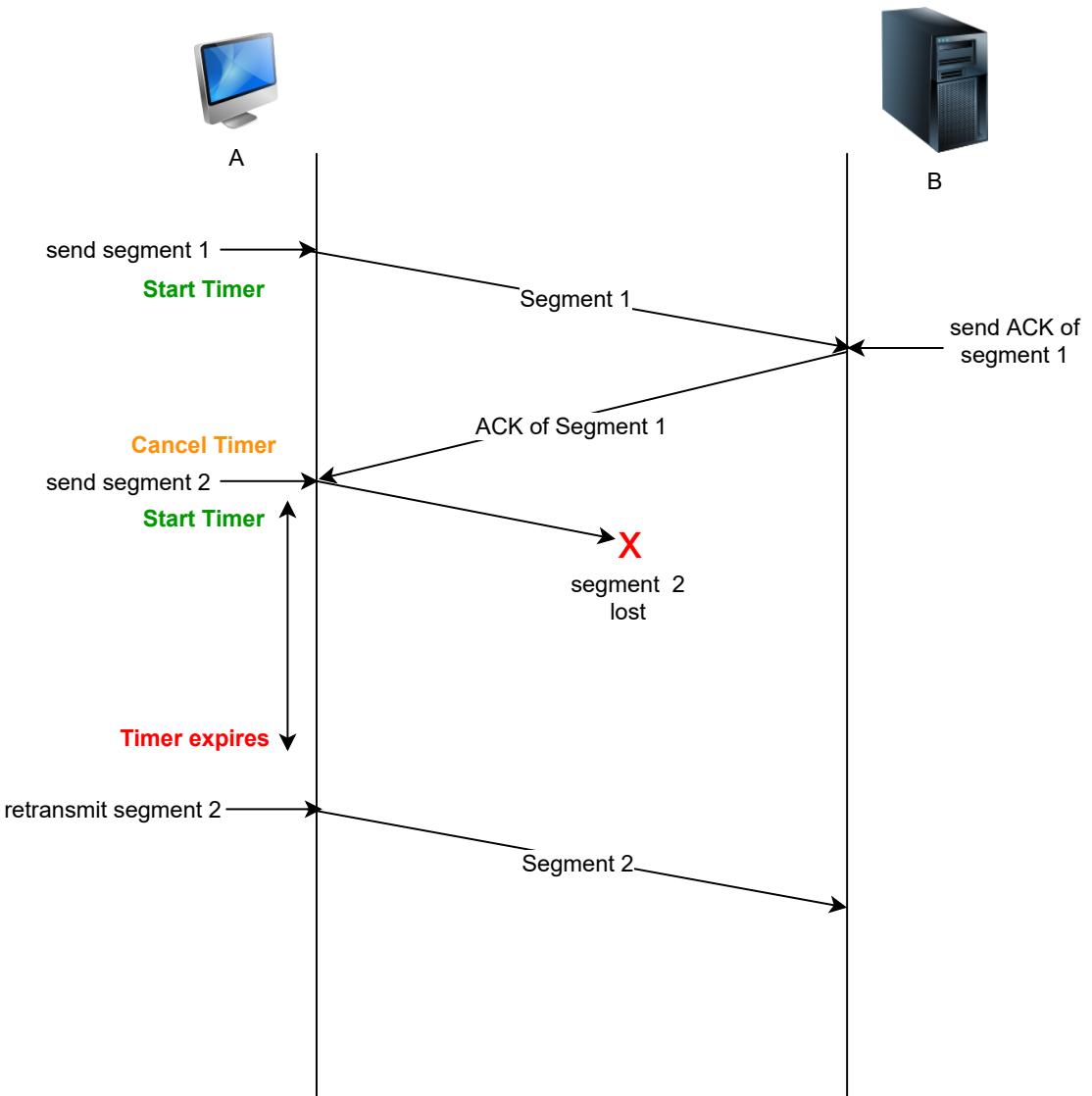
3 of 7



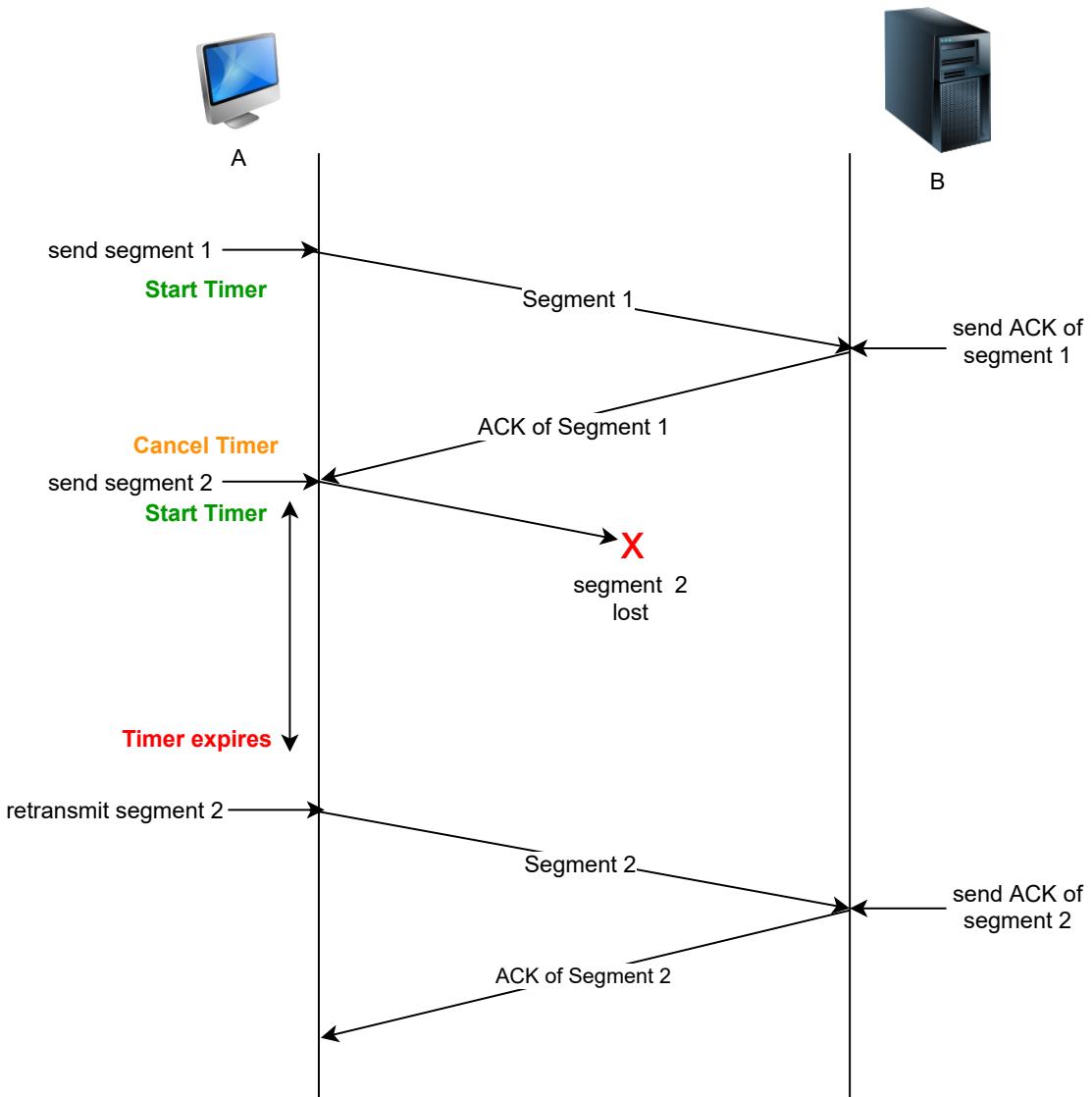
Using retransmission timers to recover from segment losses



Using retransmission timers to recover from segment losses



Using retransmission timers to recover from segment losses



Using retransmission timers to recover from segment losses

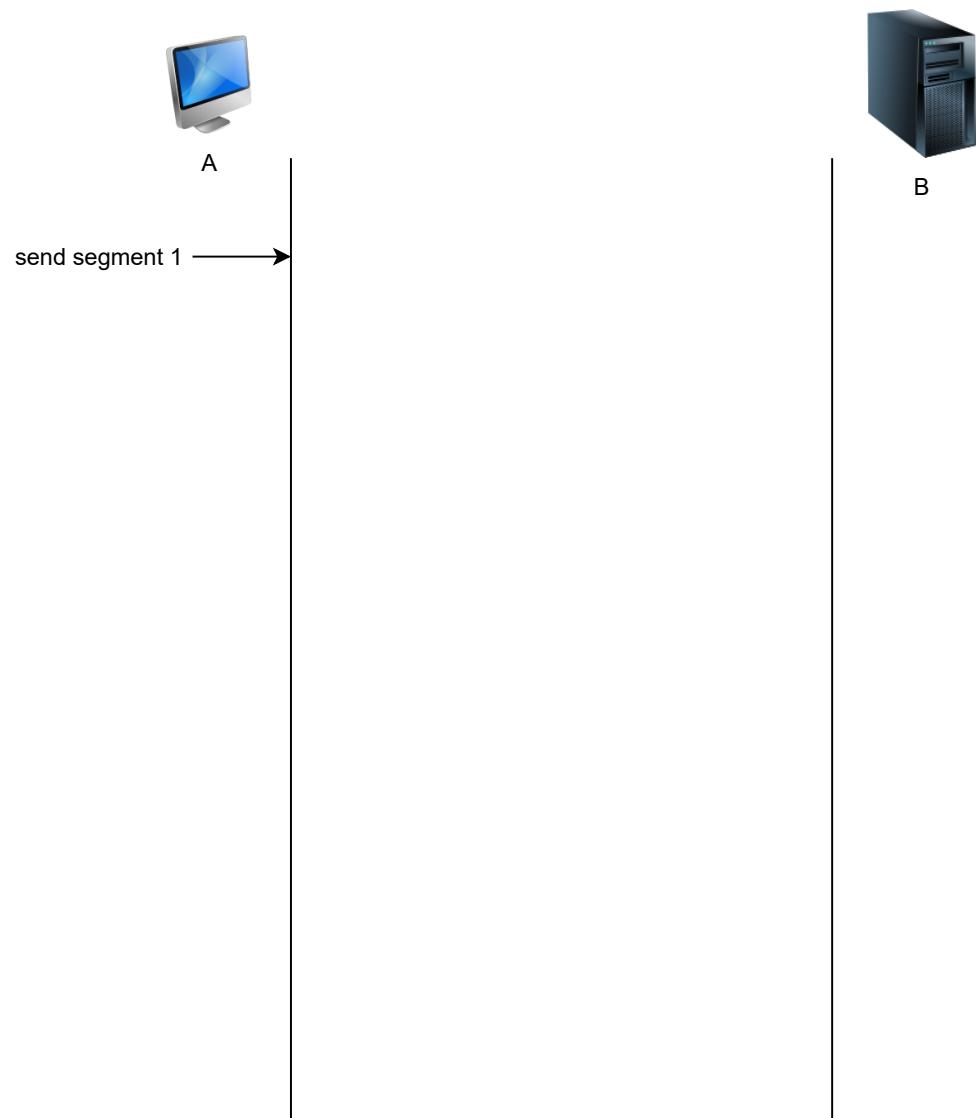
7 of 7



## Limitations of Retransmission Timers #

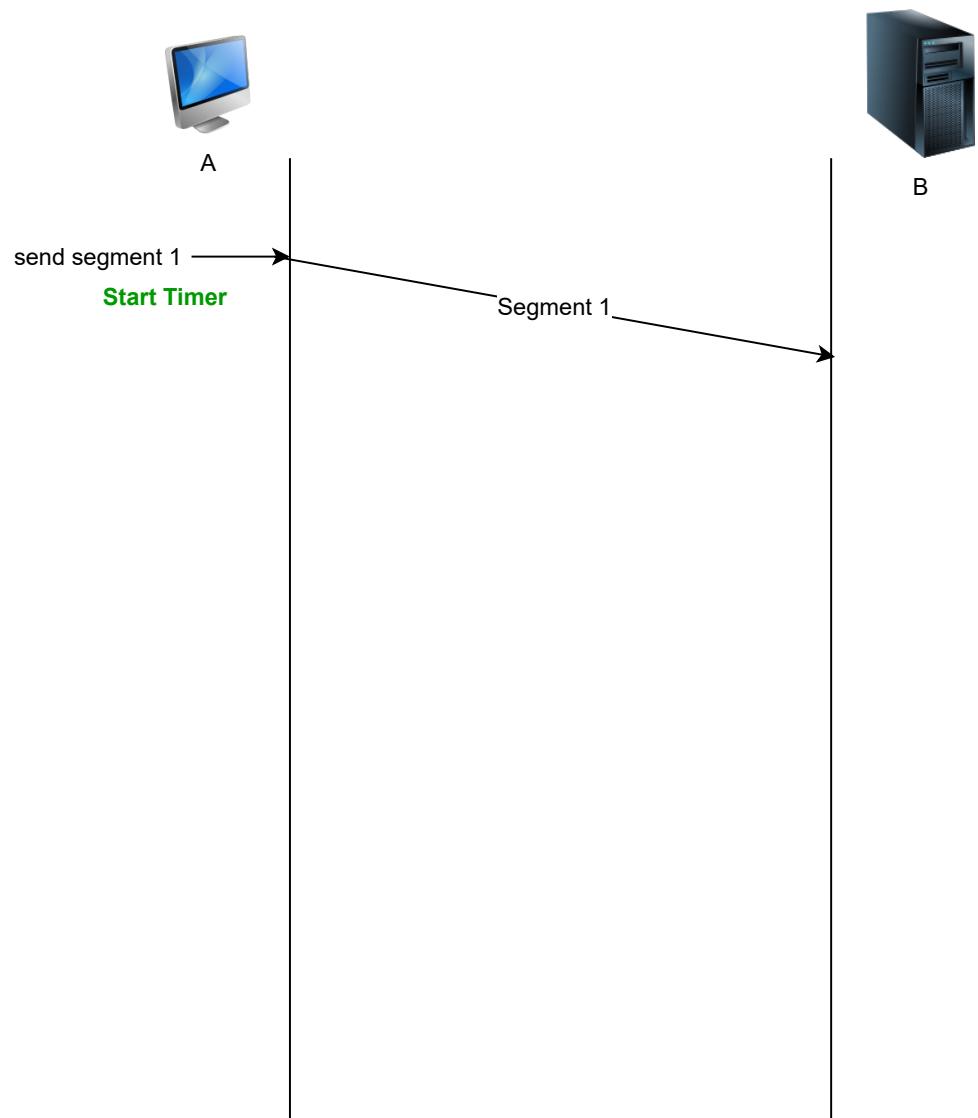
Unfortunately, retransmission timers alone are not sufficient to recover from segment losses. Let us consider the situation depicted below where **an acknowledgement is lost**. In this case, the sender retransmits a data segment that has been received correctly, but not properly acknowledged.

Unfortunately, as illustrated in the figure below, the **receiver considers the retransmission as a new segment** effectively and the segment is *duplicated*.



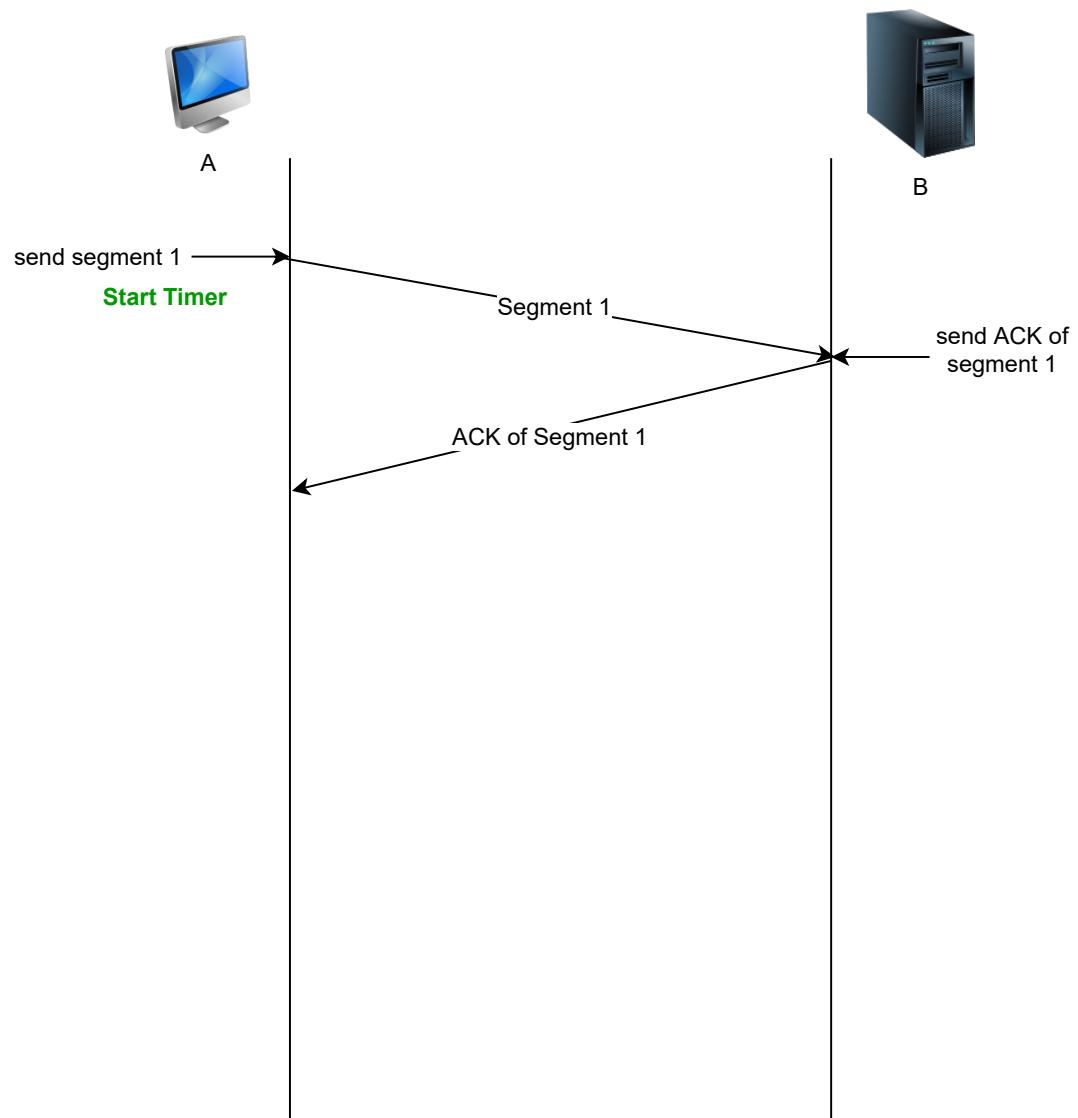
Using retransmission timers to recover from segment losses

1 of 7



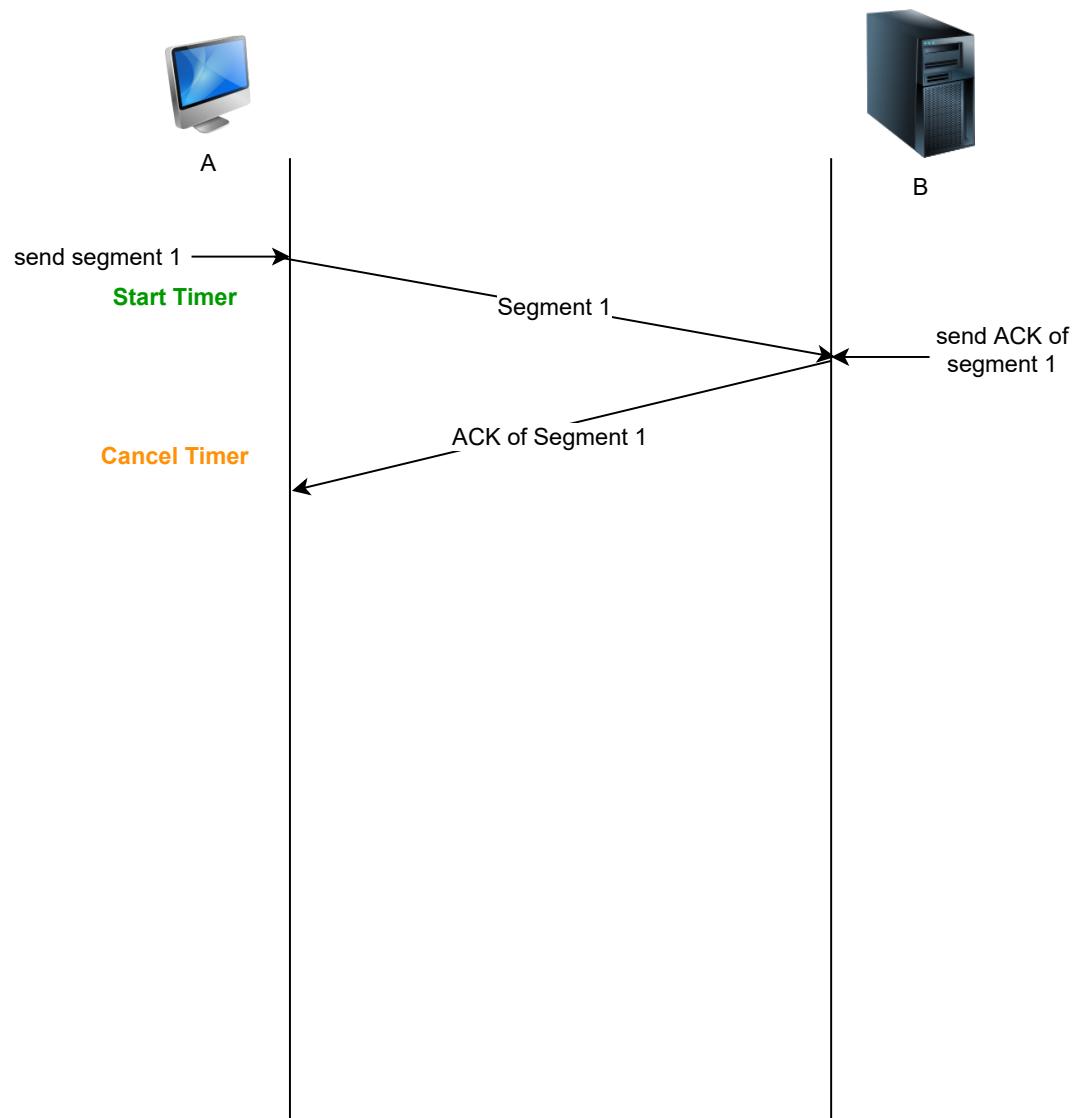
Using retransmission timers to recover from segment losses

2 of 7

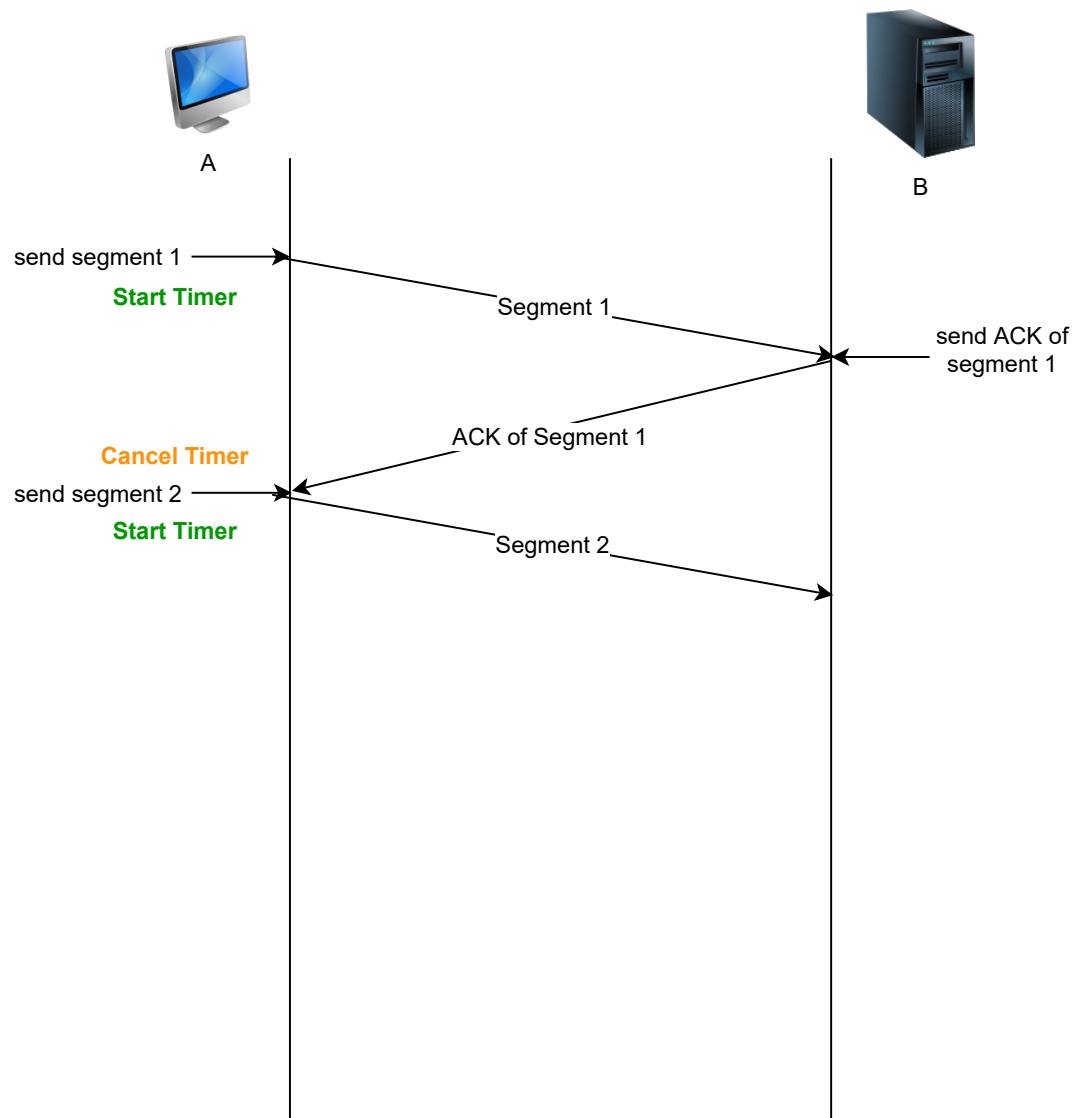


Using retransmission timers to recover from segment losses

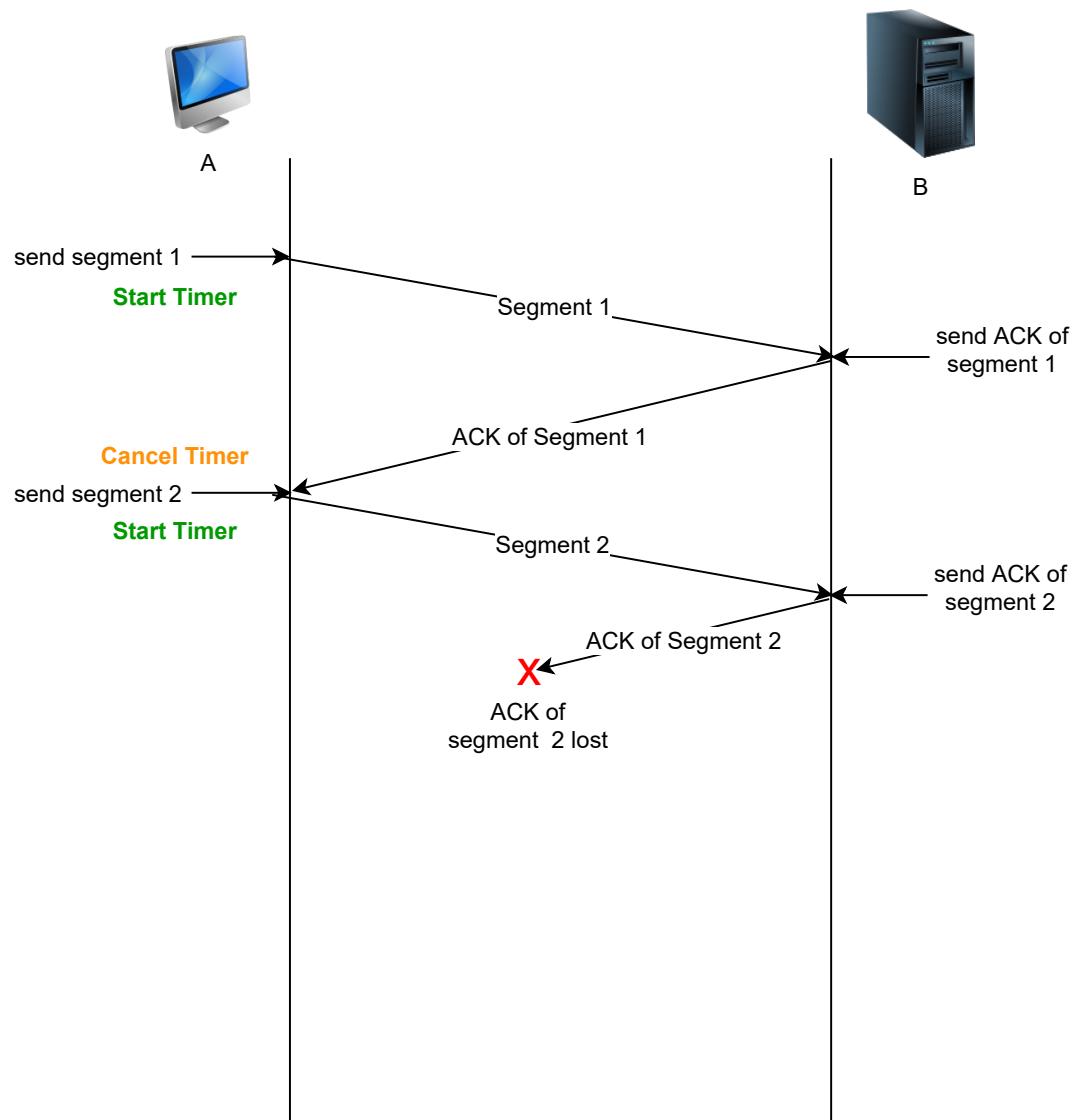
3 of 7



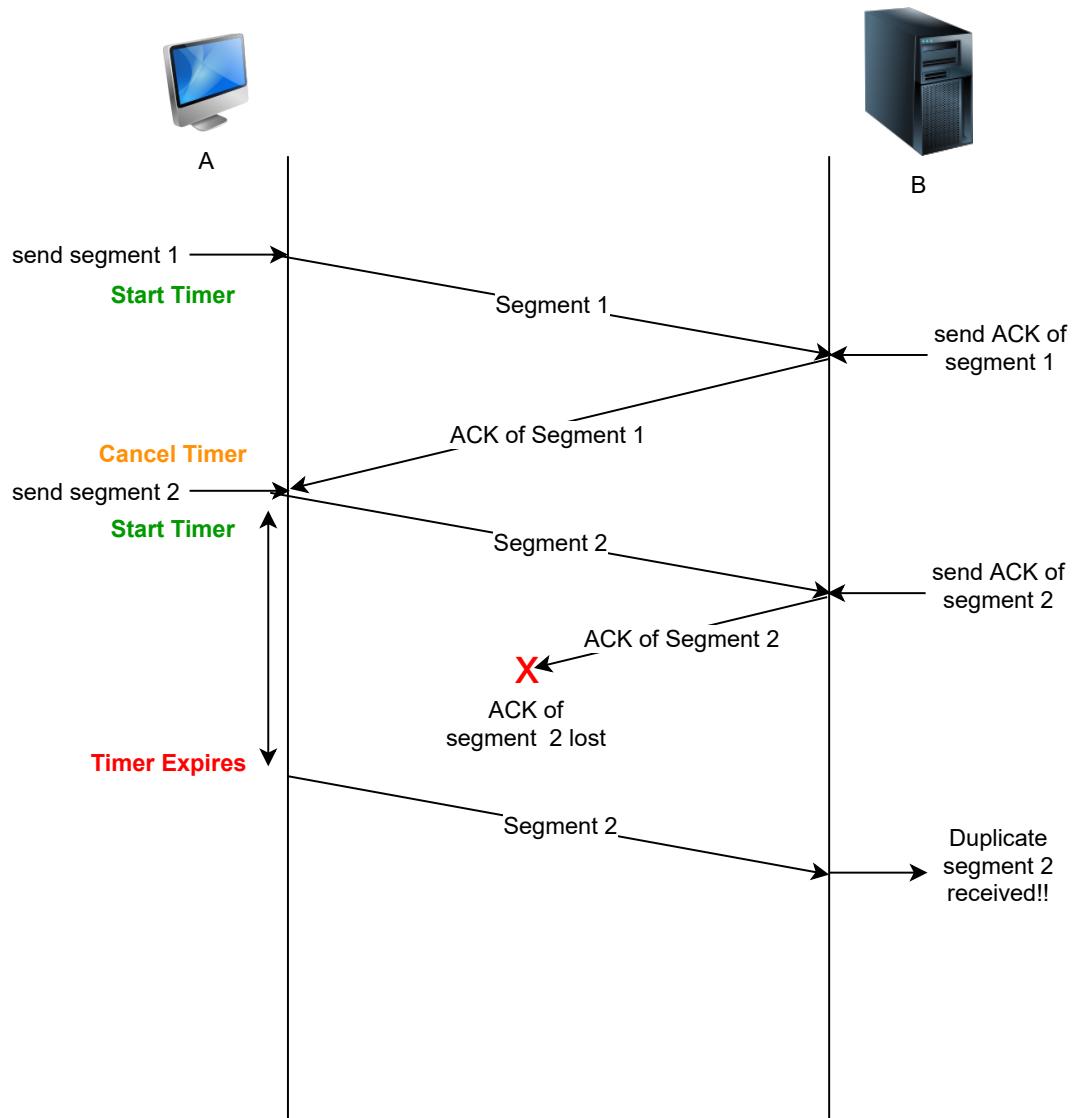
Using retransmission timers to recover from segment losses



Using retransmission timers to recover from segment losses



Using retransmission timers to recover from segment losses



Using retransmission timers to recover from segment losses

7 of 7



## 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.

## Quick Quiz! #

1

Checksums address which imperfection of the network layer?

COMPLETED 0%

1 of 5



Let's continue our discussion of reliable transfer data in the next lesson!

# Reliable Data Transfer: Sliding Window

In this lesson, we'll study sliding windows

## WE'LL COVER THE FOLLOWING ^

- Pipelining
  - Sliding Window
- Quick Quiz!

## 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 processor emits the message to be sent, the network carries the message to the destination.
2. The receiver processor receives and emits an acknowledgment message.
3. 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.

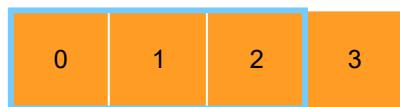
## Sliding Window #

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.

The figure below illustrates the operation of the sliding window. The sliding window shown contains three segments. The sender can thus transmit three segments before being forced to wait for an acknowledgment. The sliding window moves to the higher sequence numbers upon reception of acknowledgments. When the first acknowledgment (of segment 0) is received, it allows the sender to move its sliding window to the right, and sequence number 3 becomes available.

In practice, as the segment header encodes the sequence number in a  $n$  bits string, only the sequence numbers between 0 and  $2^n - 1$  can be used. There's a problem here. What happens when an application has transmitted  $2^n - 1$  messages and still has more data to send? Well, the same sequence number can be used for different segments and that the sliding window can wrap. This is illustrated in the figure below assuming that 2 bits are used to encode the sequence number in the segment header. Note that upon reception of the acknowledgment for the first segment, the sender slides its window and can use sequence number 0 again.

**Sending Window**



A



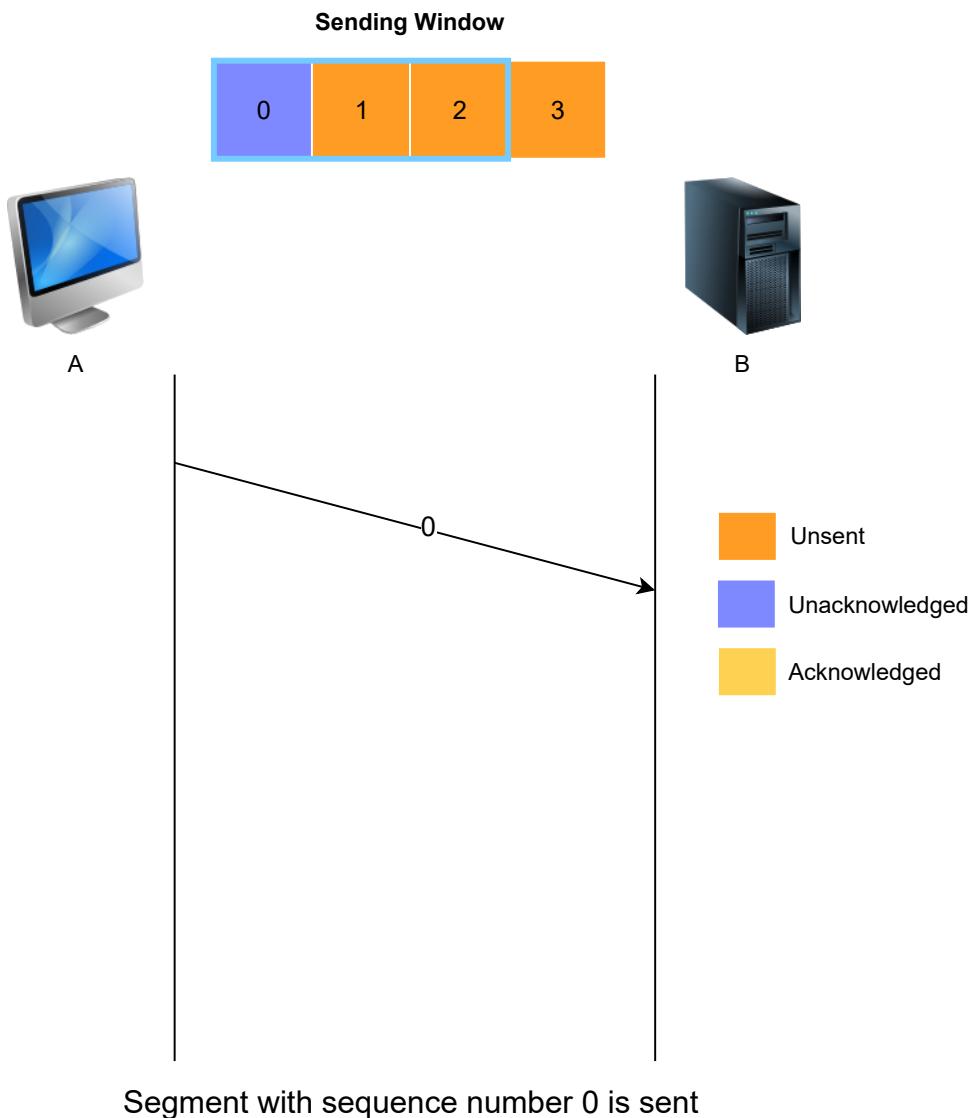
B

- █ Unsent
- █ Unacknowledged
- █ Acknowledged

A sending sliding window of size 3 is initiated

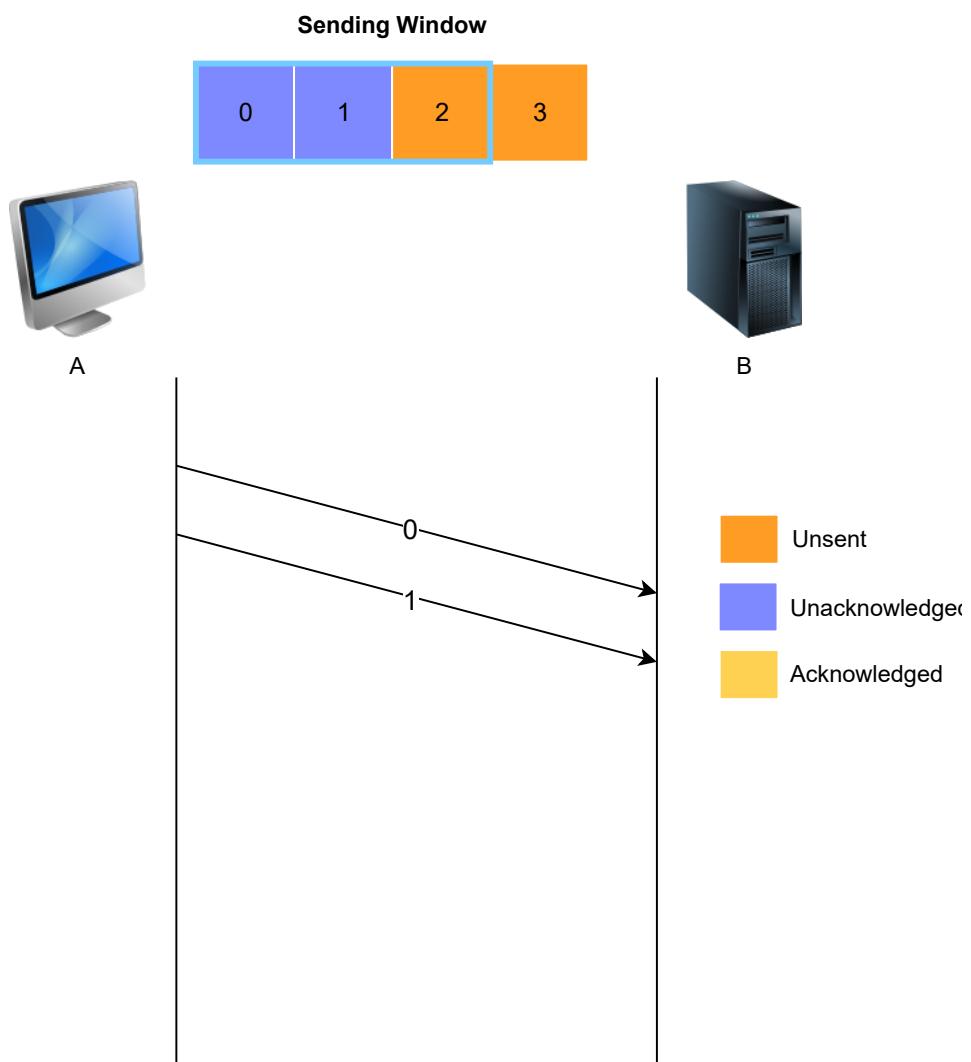
The sliding window

1 of 11



The sliding window

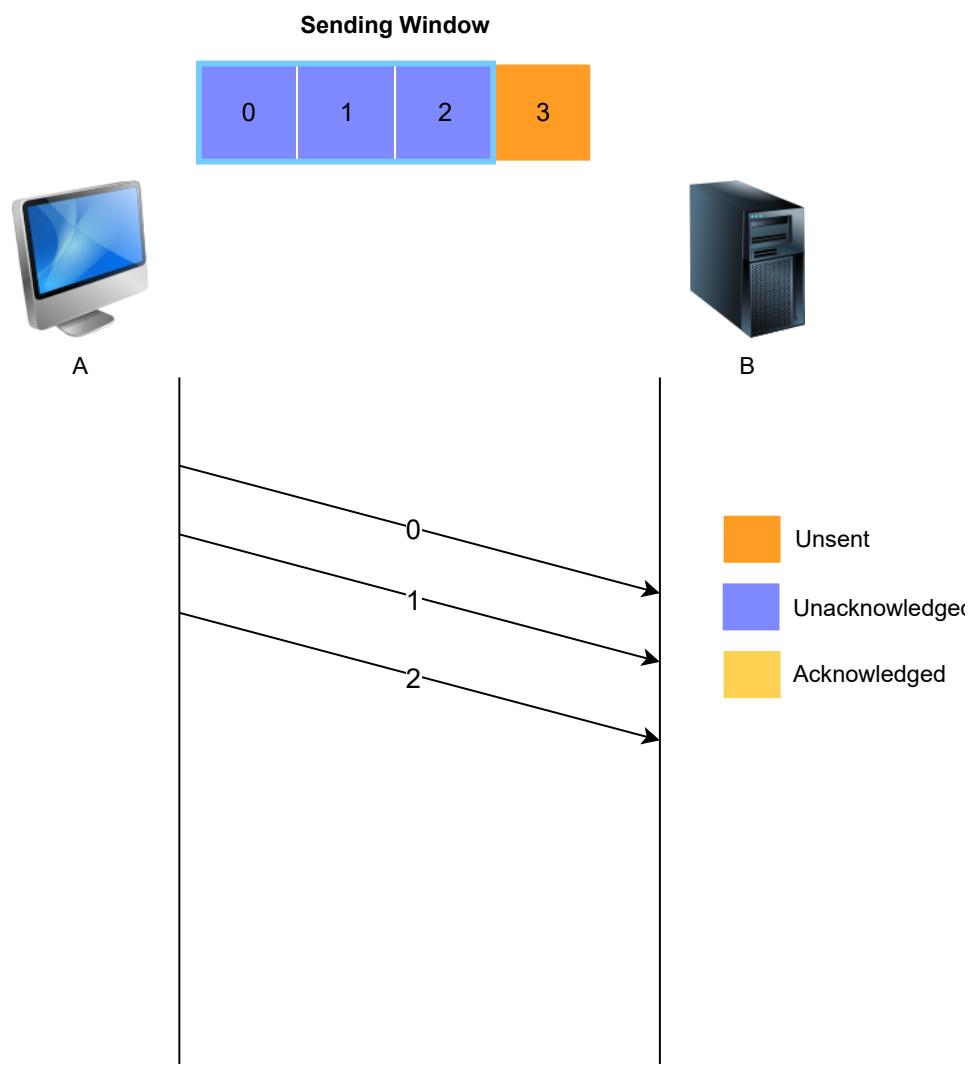
2 of 11



Segment with sequence number 1 is sent.  
No acknowledgement is received yet

The sliding window

3 of 11

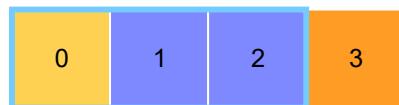


Segment with sequence number 2 is sent.  
No acknowledgement is received yet.

The sliding window

4 of 11

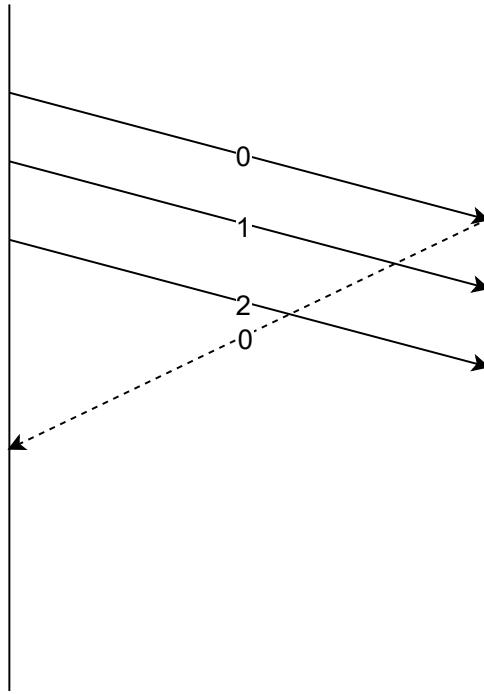
**Sending Window**



A



B

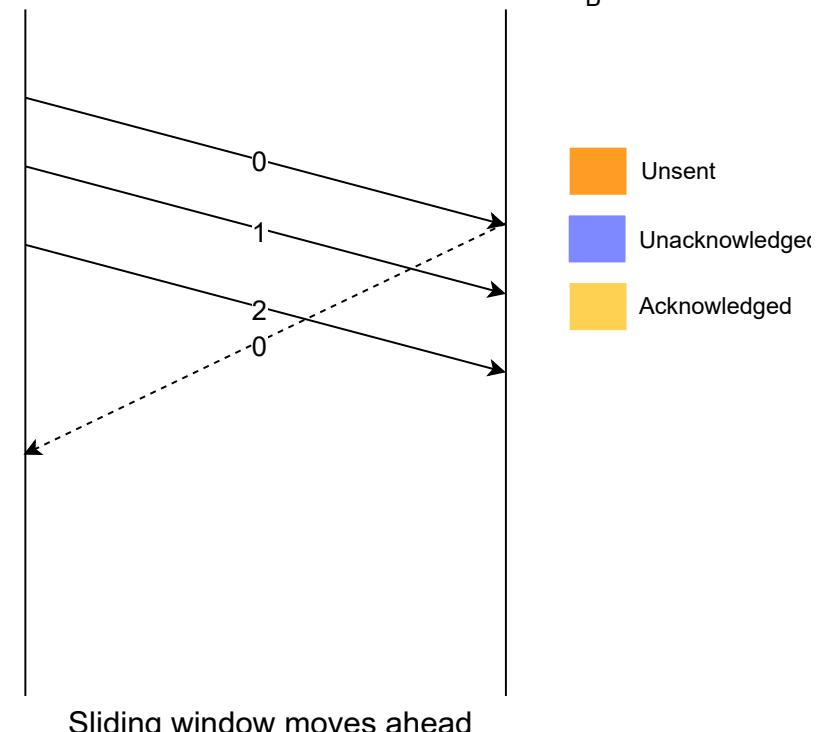
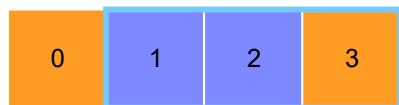


Acknowledgement for segment with sequence number 0 received

The sliding window

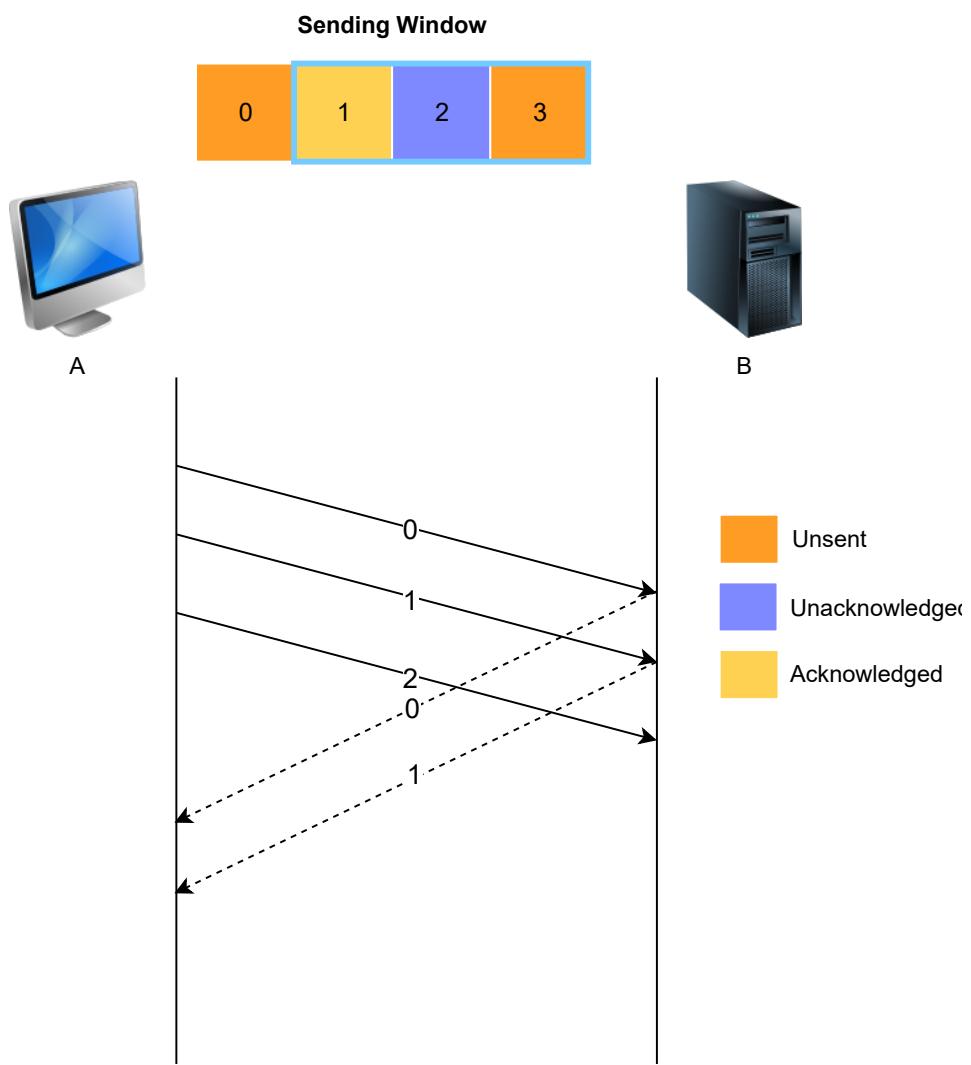
5 of 11

**Sending Window**



The sliding window

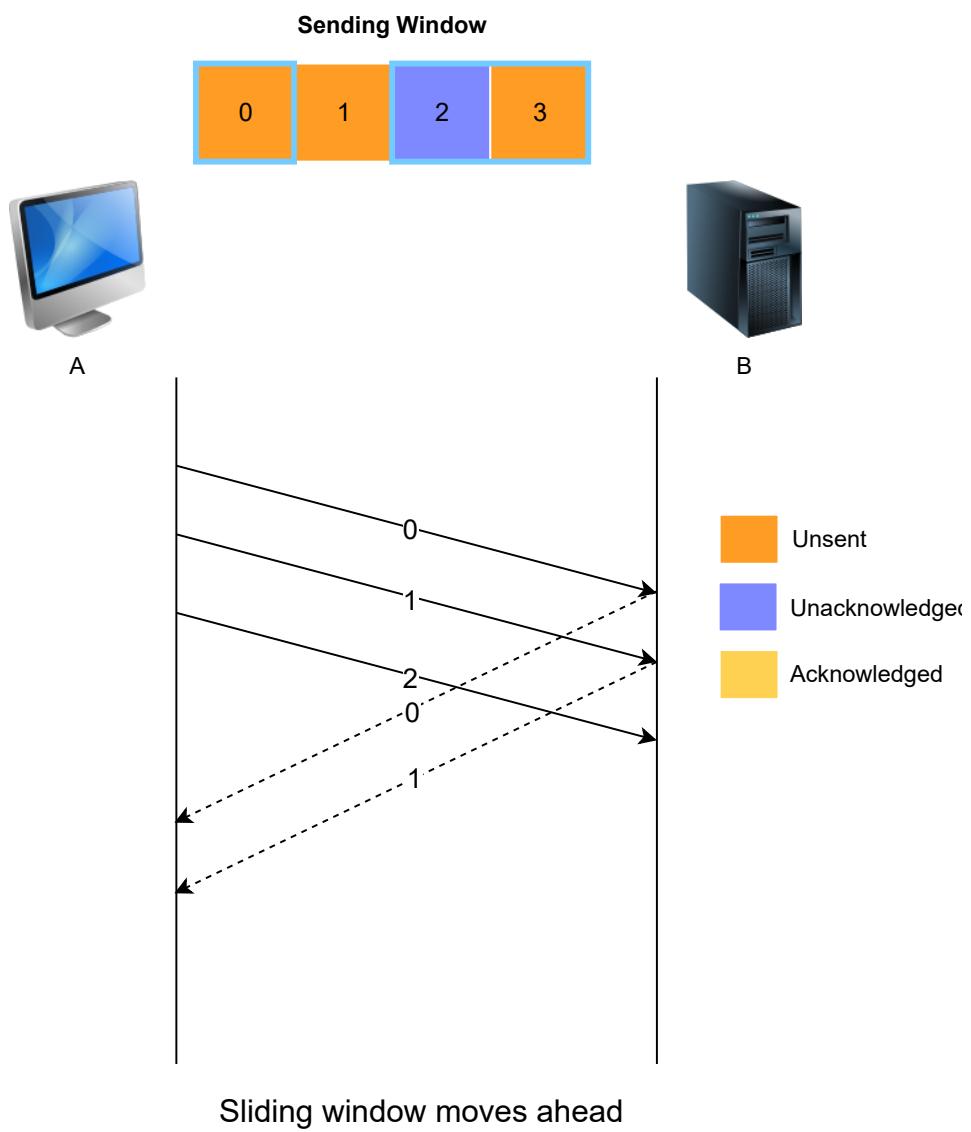
6 of 11



Acknowledgement of segment with sequence number 1 received

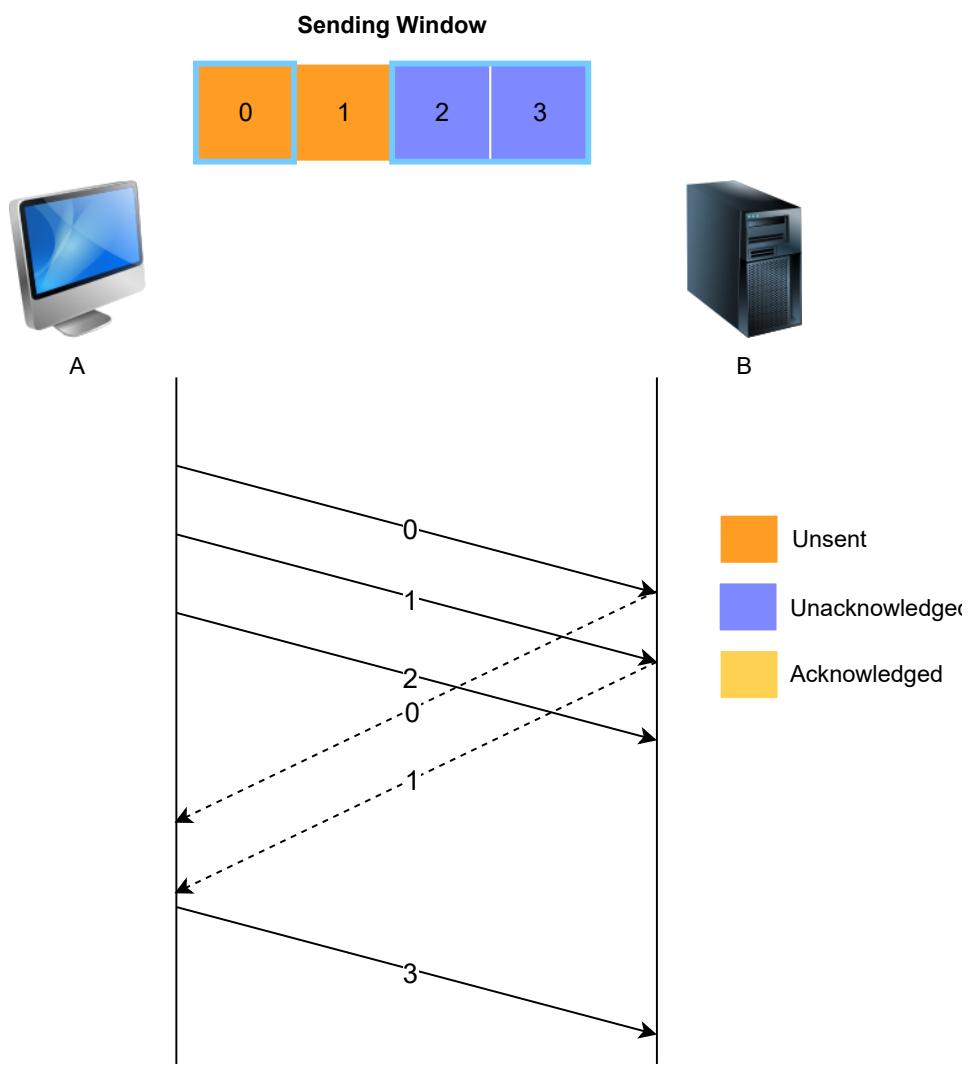
The sliding window

7 of 11



The sliding window

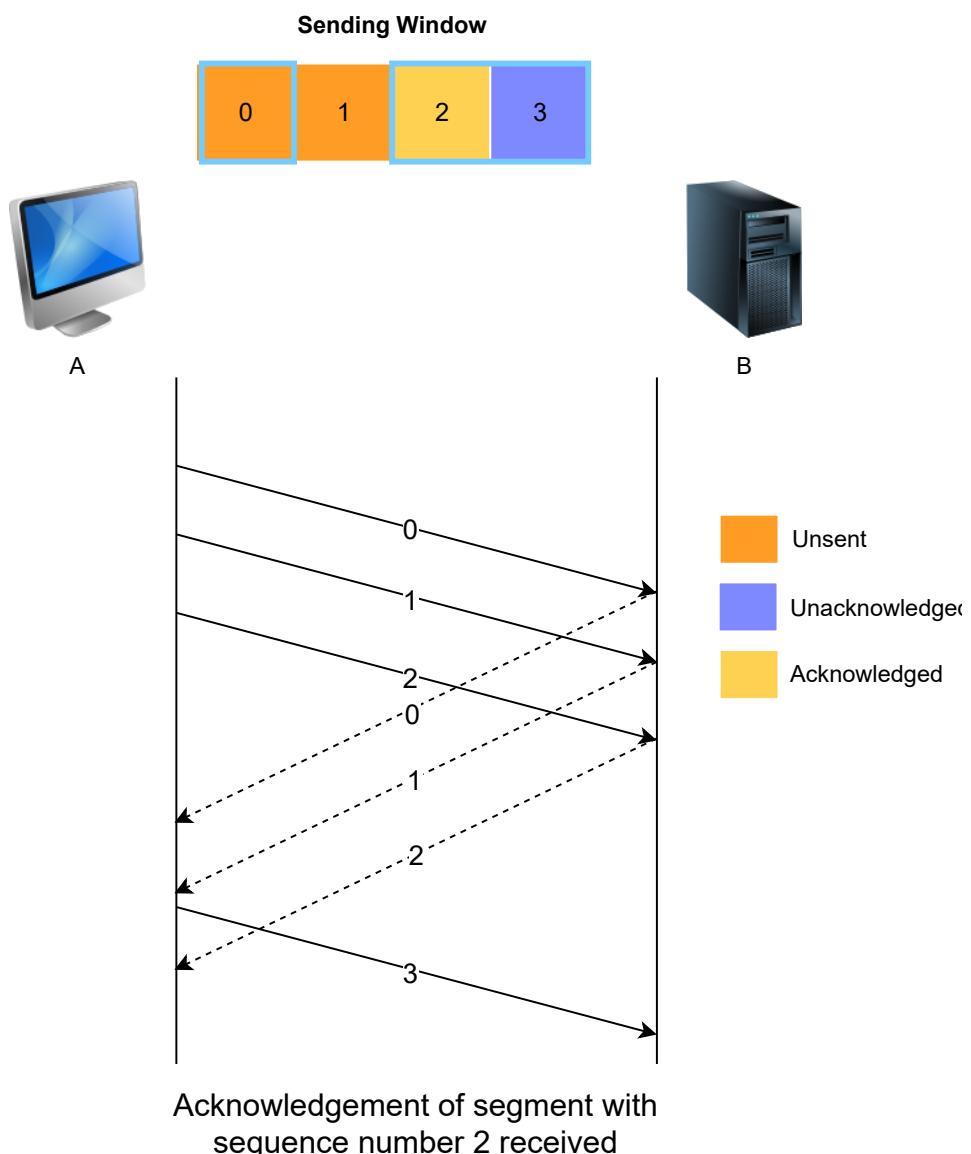
8 of 11



data for segment 3 has just arrived from the application (it wasn't available earlier) and data with sequence 3 is sent

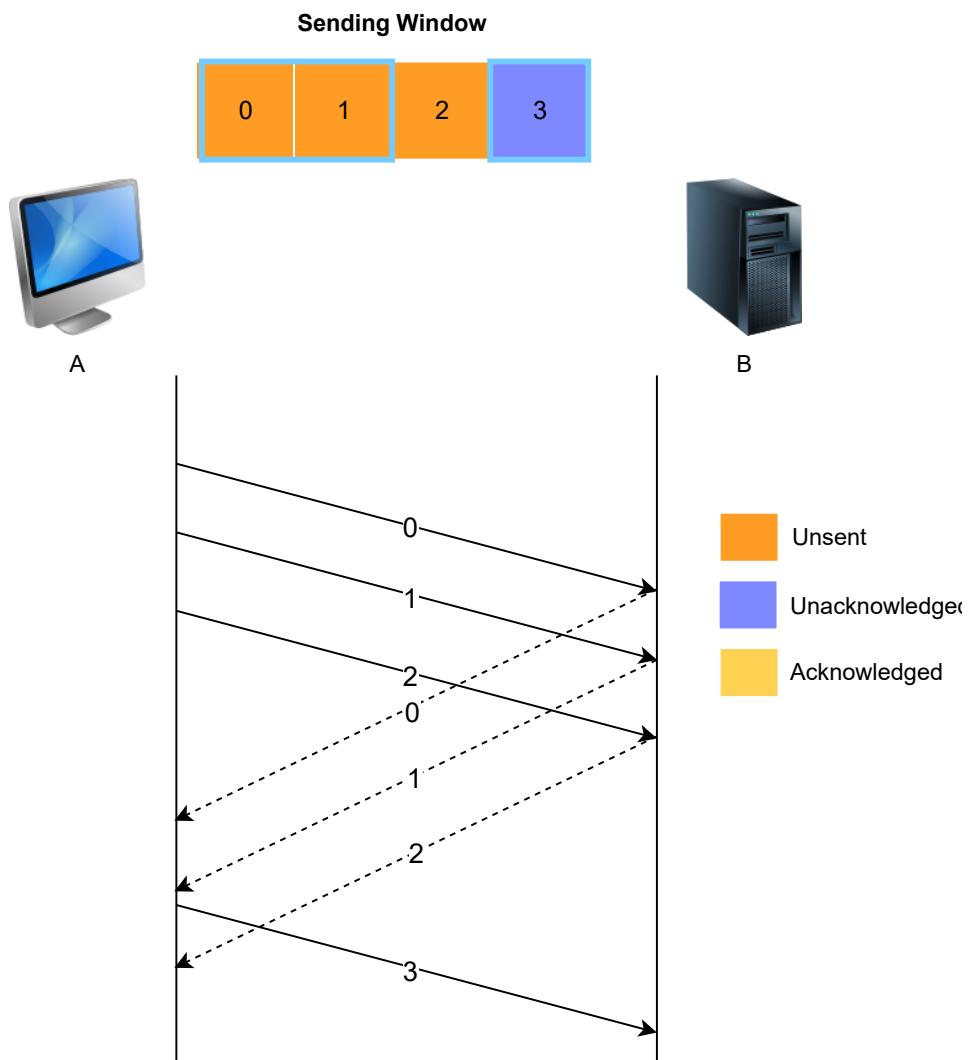
The sliding window

9 of 11



The sliding window

10 of 11



Sliding window moves ahead. The sending/receiving continues in this fashion.

The sliding window

11 of 11



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.

# Quick Quiz! #

1

Pipelining and congestion control are competing objectives.

COMPLETED 0%

1 of 2



Let's look at both in the next lesson.

# Reliable Data Transfer: Go-back-n

In this lesson, we'll study go-back-n: a simple protocol to ensure detection and retransmission of lost packets.

## WE'LL COVER THE FOLLOWING



- Go-back-n
  - Go-back-n Receiver
    - Cumulative Acknowledgements
  - Go-back-n Sender
    - Retransmission Timer
  - Advantages of Go-back-n
- Selective Repeat
- Comparing to go-back-n
- Quick Quiz!

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**

## Go-back-n #

The simplest sliding window protocol uses **go-back-n** recovery.

## Go-back-n Receiver #

Intuitively, go-back-n receiver operates as follows:

1. It only accepts the segments that arrive in-sequence.
2. It discards any out-of-sequence segment that it receives.
3. When it receives a data segment, it always returns an acknowledgment.

- When it receives a data segment, it always returns an acknowledgement containing the sequence number of the **last in-sequence segment** that it has received.

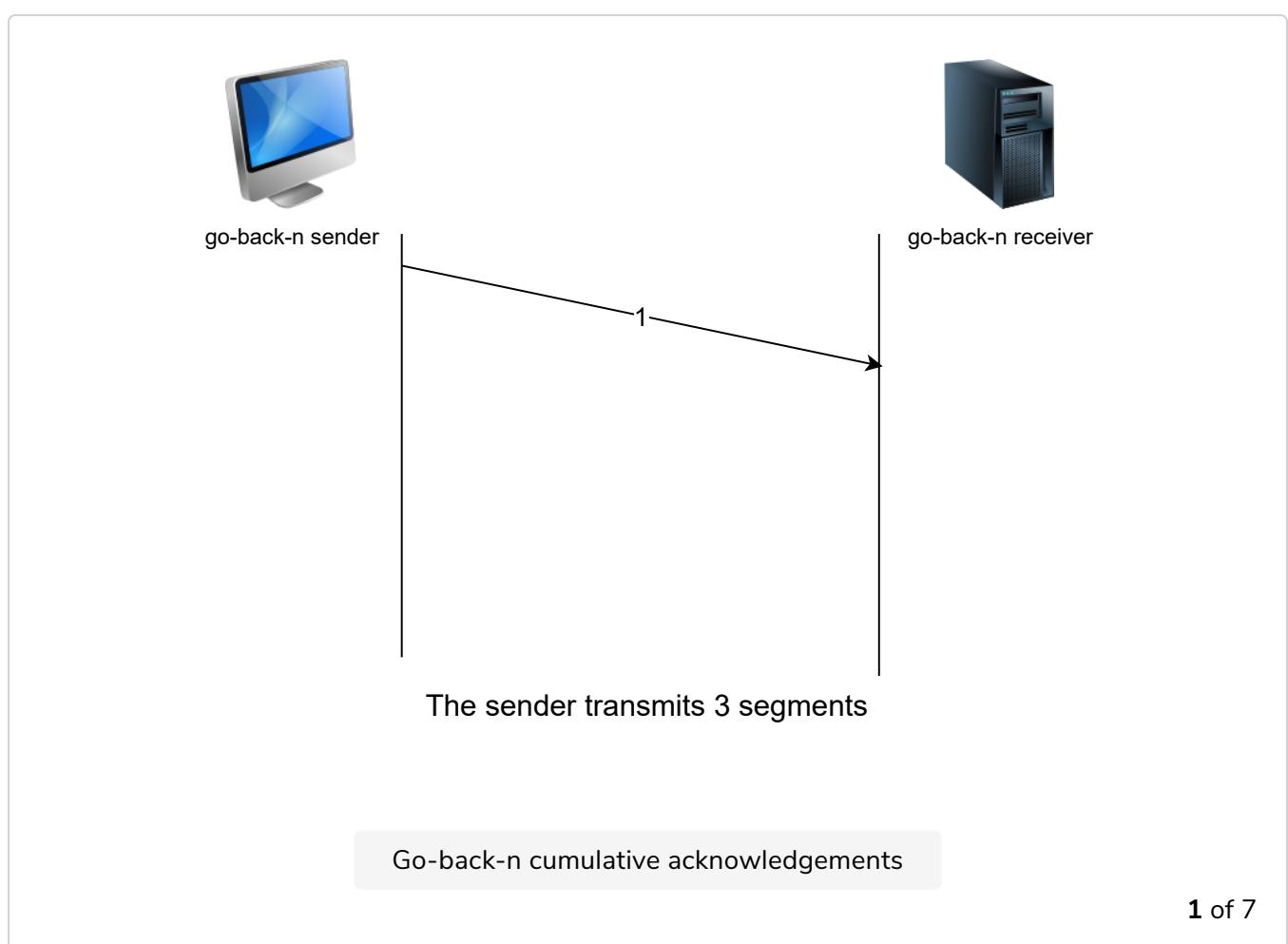
### Cumulative Acknowledgements #

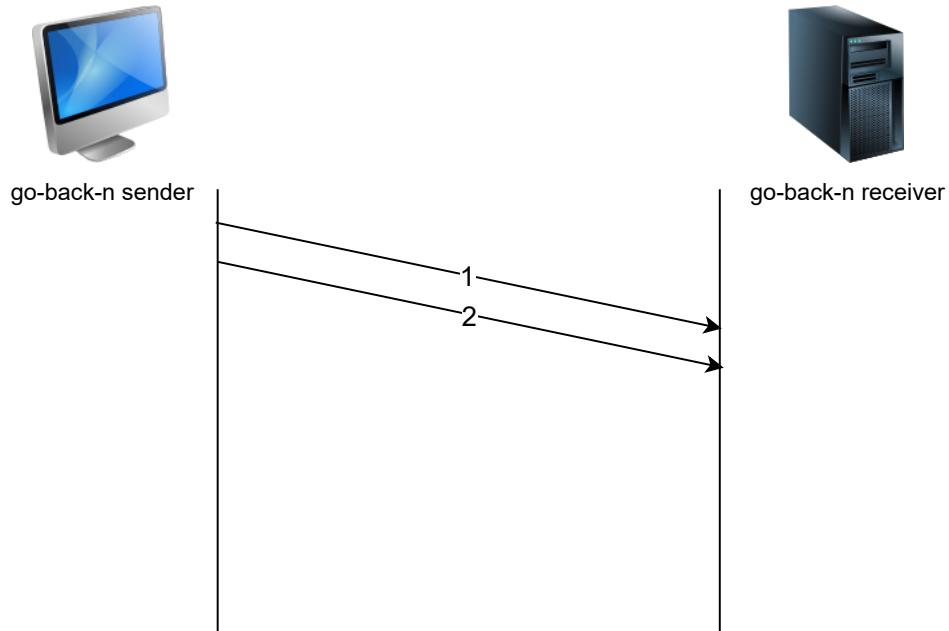
This acknowledgment is said to be **cumulative**. When a go-back-receiver sends an acknowledgment for sequence number  $x$ , it implicitly acknowledges the reception of all segments whose sequence number is smaller than or equal to  $x$ .

A key advantage of these cumulative acknowledgments is that it's **easy to recover from the loss of an acknowledgment**.

Consider for example a go-back-n receiver that received segments 1, 2 and 3.

- It sent acknowledgments for all three segments.
- Unfortunately, acknowledgments of the first two were lost.
- Thanks to the cumulative acknowledgments, the receiver receives the acknowledgment for the last segment and so it knows that all three have been correctly received.

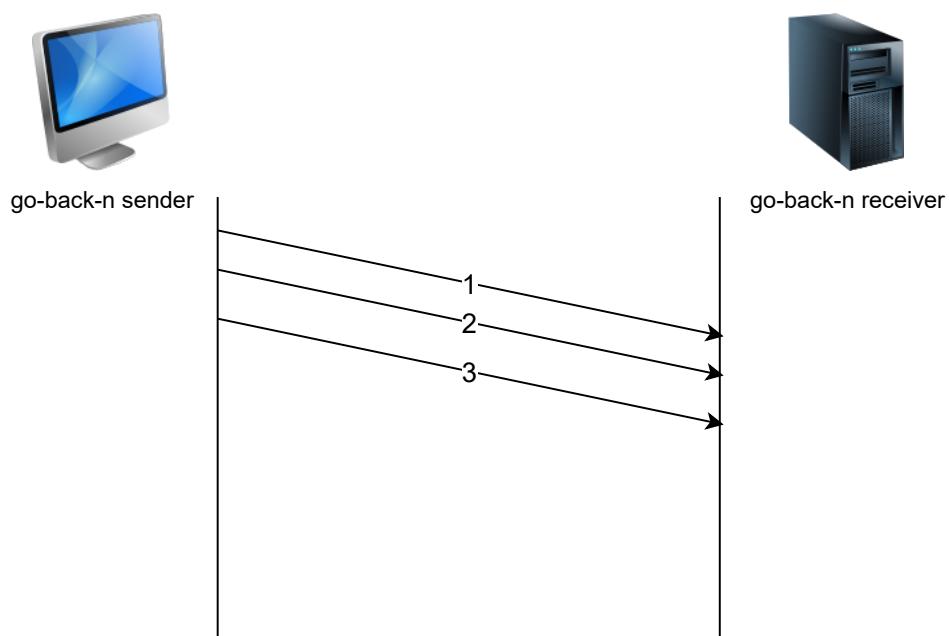




The sender transmits 3 segments

Go-back-n cumulative acknowledgements

2 of 7



The sender transmits 3 segments

Go-back-n cumulative acknowledgements

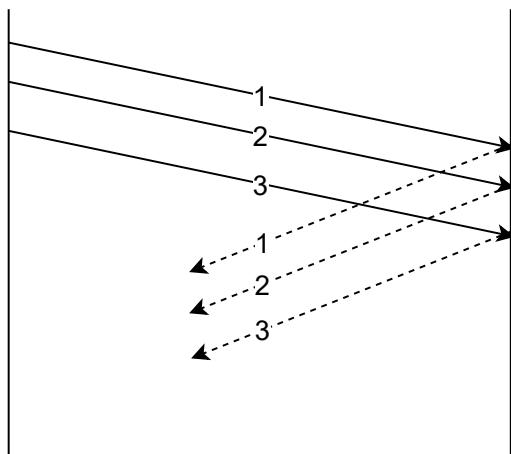
3 of 7



go-back-n sender



go-back-n receiver



The receiver sends acknowledgements  
for all 3 segments

Go-back-n cumulative acknowledgements

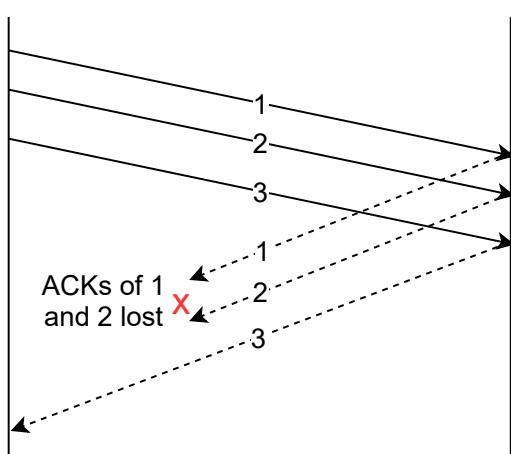
4 of 7



go-back-n sender



go-back-n receiver



However, the acknowledgments for the  
first and second segments are lost

Go-back-n cumulative acknowledgements

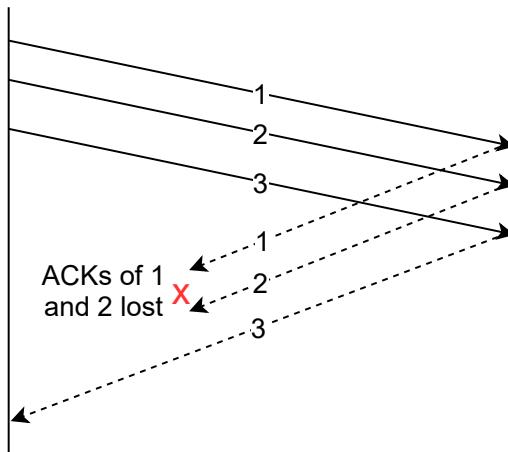
5 of 7



go-back-n sender



go-back-n receiver



The acknowledgment for the third segment gets passed through though

Go-back-n cumulative acknowledgements

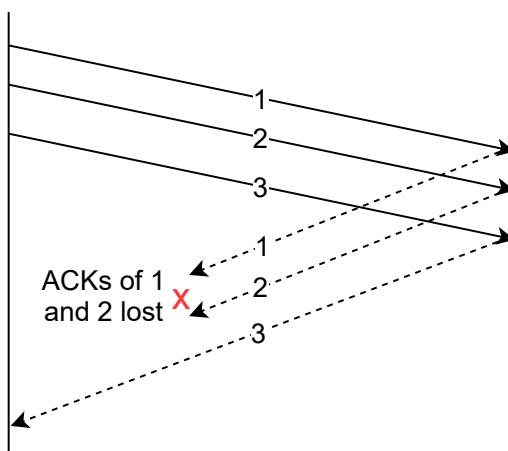
6 of 7



go-back-n sender



go-back-n receiver



Hence the sender knows that the first two were received correctly too and does not retransmit them

Go-back-n cumulative acknowledgements

7 of 7

## Go-back-n Sender #

A go-back-n sender uses a sending buffer that can store an entire sliding window of segments.

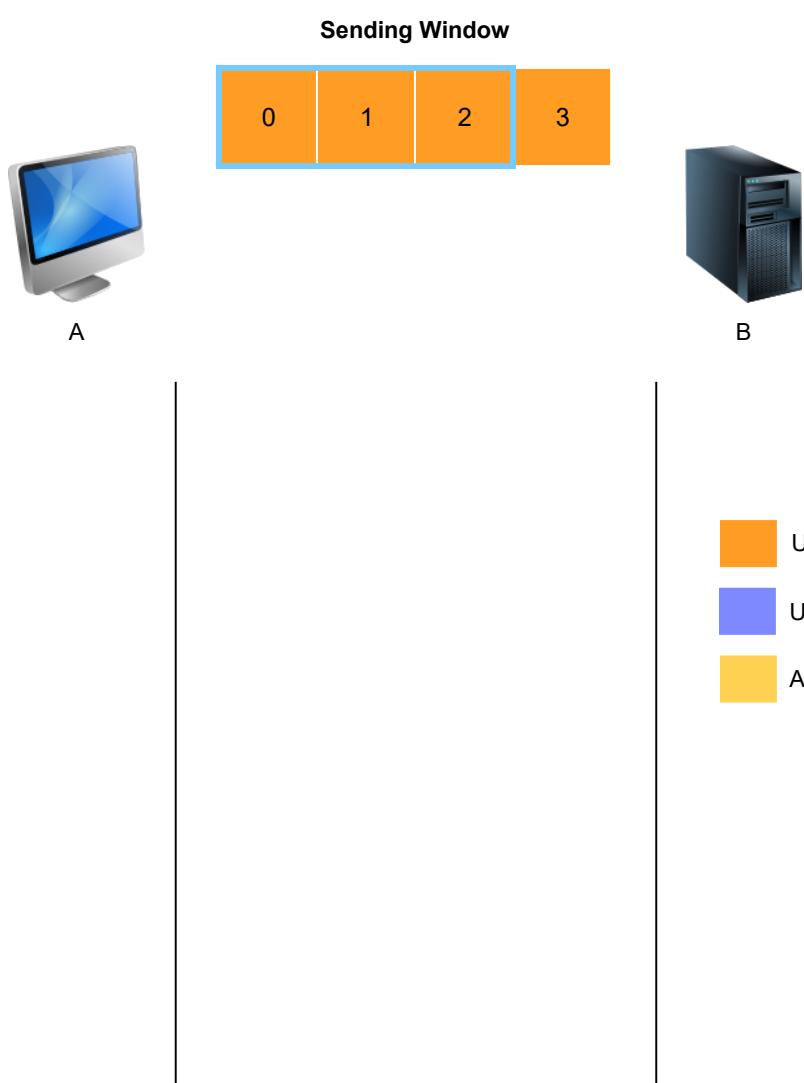
- The segments are sent with a sending sliding window that we looked at in the last lesson.
- The sender must wait for an acknowledgment once its sending buffer is full.
- When a go-back-n sender receives an acknowledgment, it removes all the acknowledged segments from the sending buffer.

## Retransmission Timer #

A go-back-n sender uses a **retransmission timer** to detect segment losses. It maintains one retransmission timer per connection. Here's how it works:

1. This timer is started when the first segment is sent.
2. When the go-back-n sender receives an acknowledgment, it restarts the retransmission timer, but only if any unacknowledged segments exist in its sending window.
3. When the retransmission timer expires, the go-back-n sender assumes that all of the unacknowledged segments currently stored in its sending buffer have been lost. It thus retransmits all the unacknowledged segments in the buffer and restarts its retransmission timer.

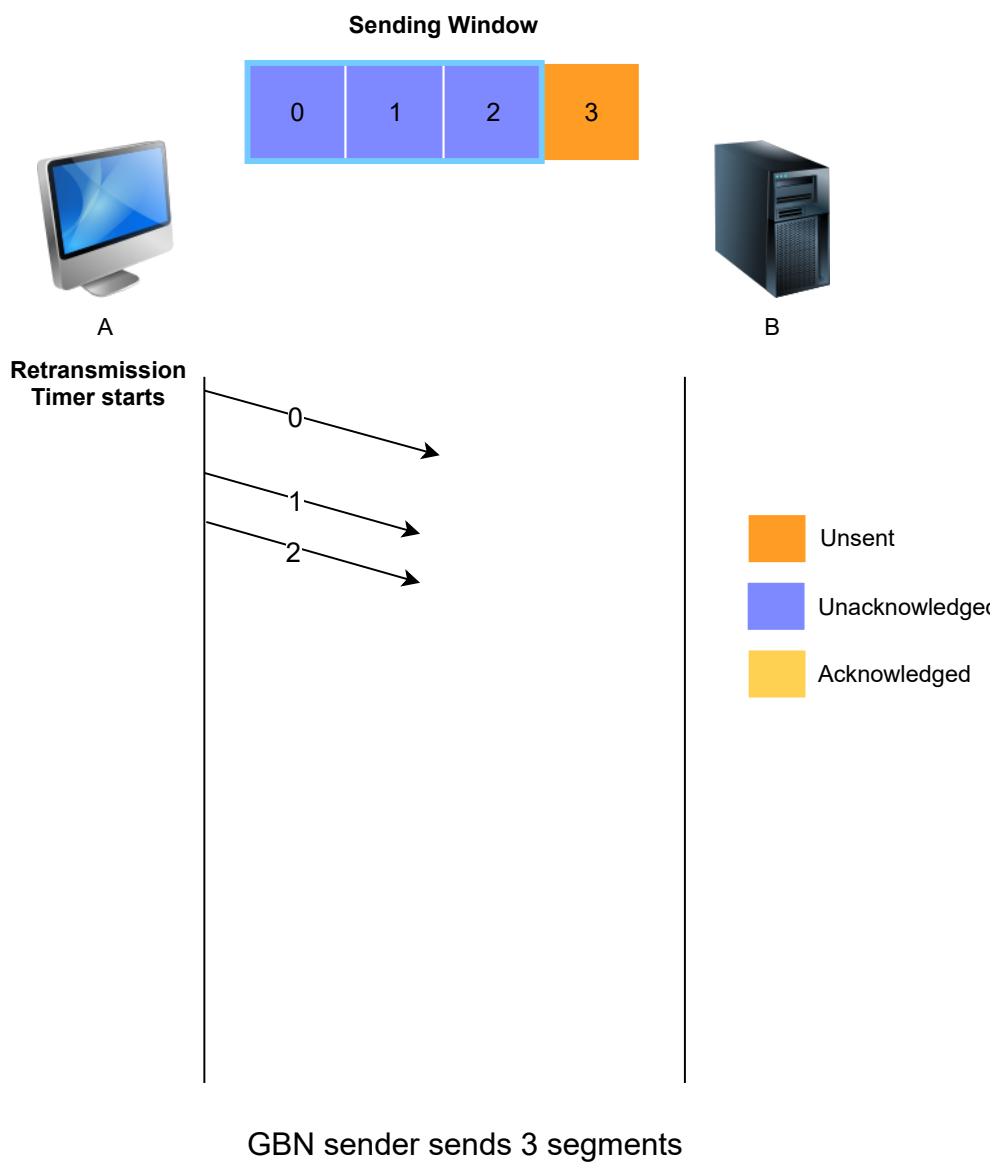
The general operation of go-back-n is illustrated in the figure below.



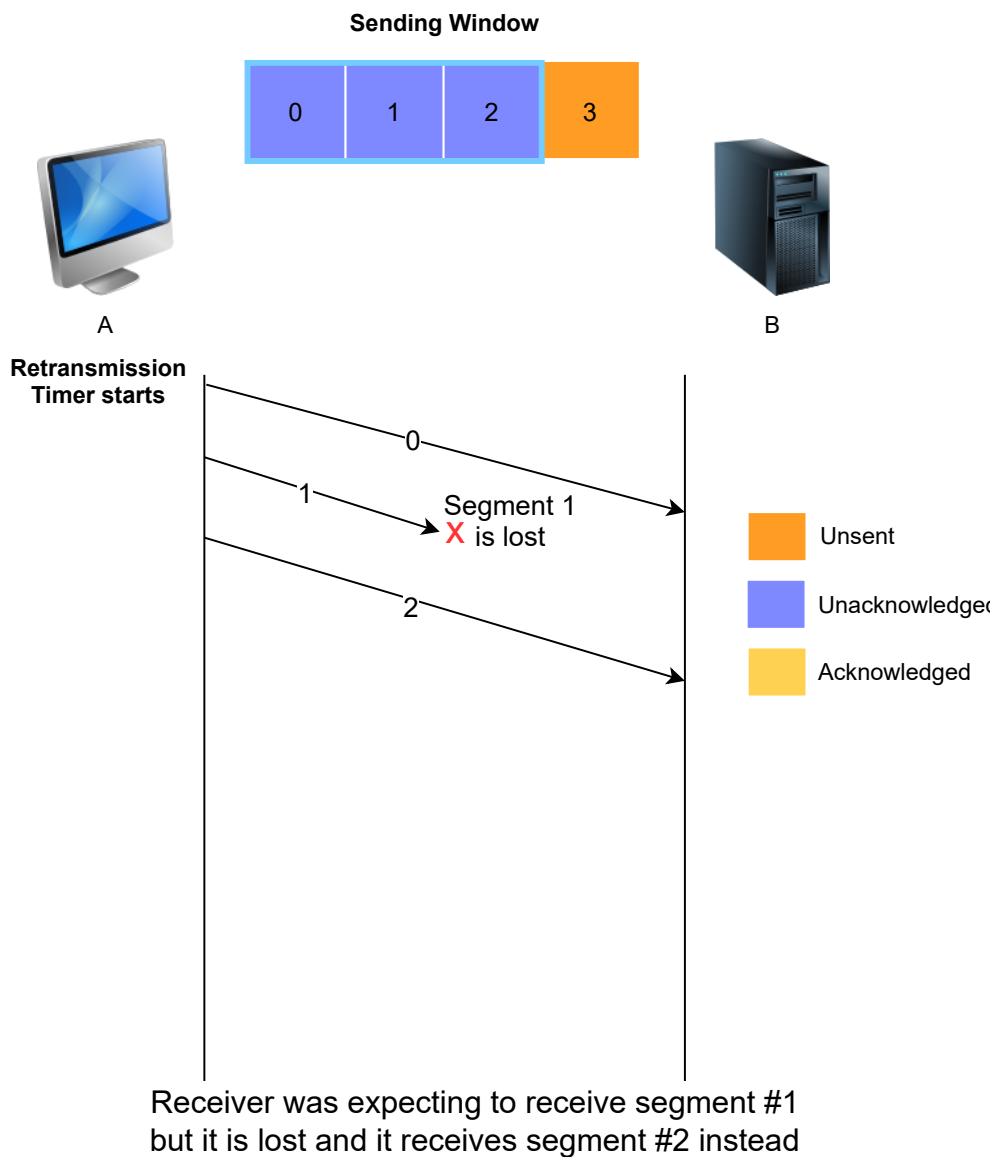
A sending sliding window of size 3 is initiated

Go-back-n: example

1 of 7

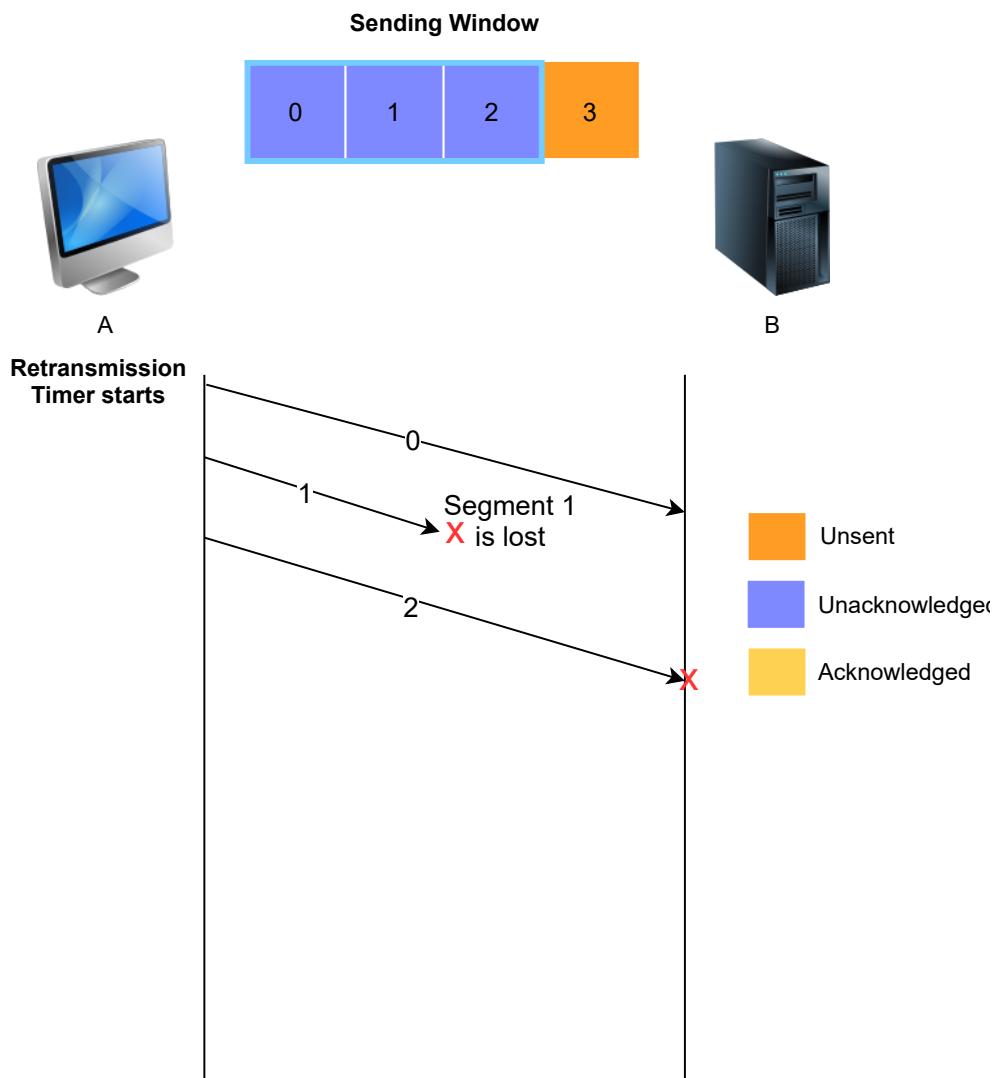


Go-back-n: example



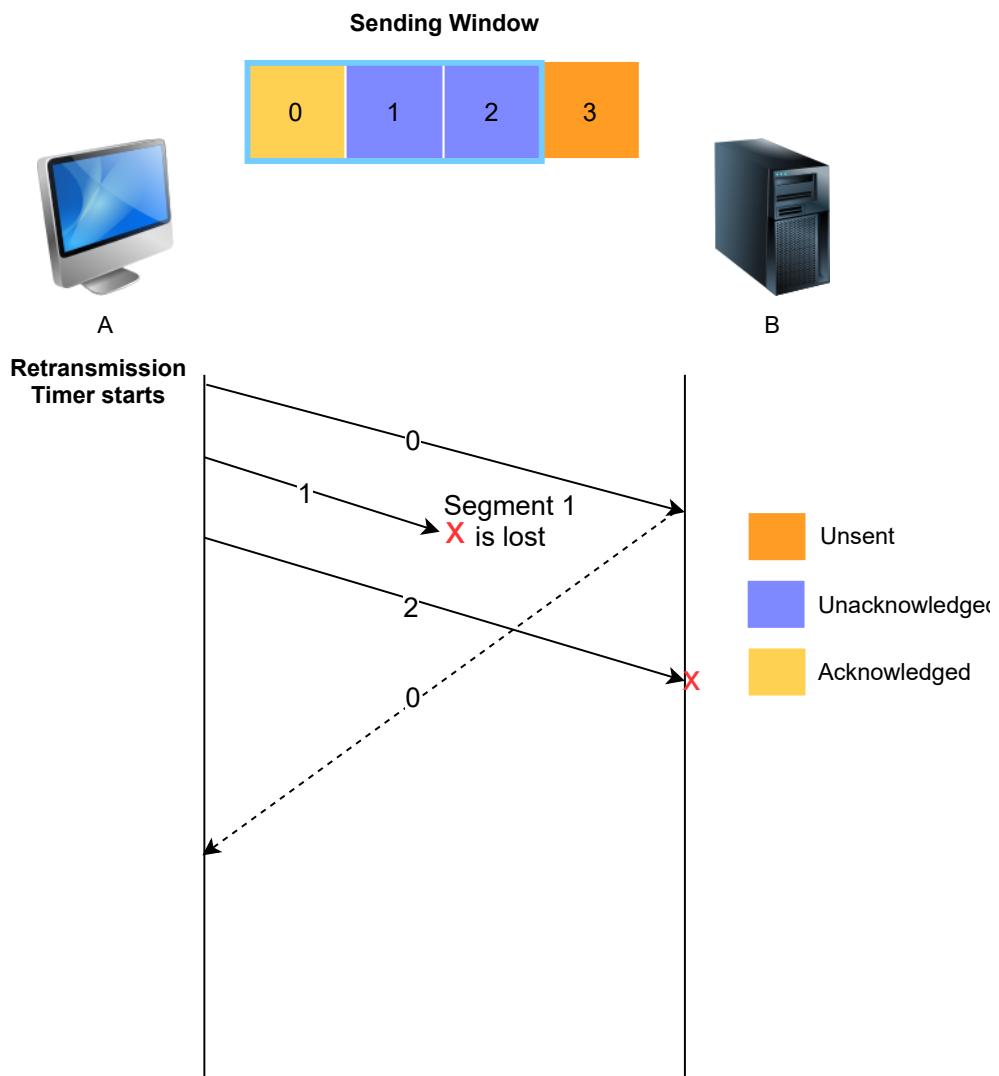
Go-back-n: example

3 of 7



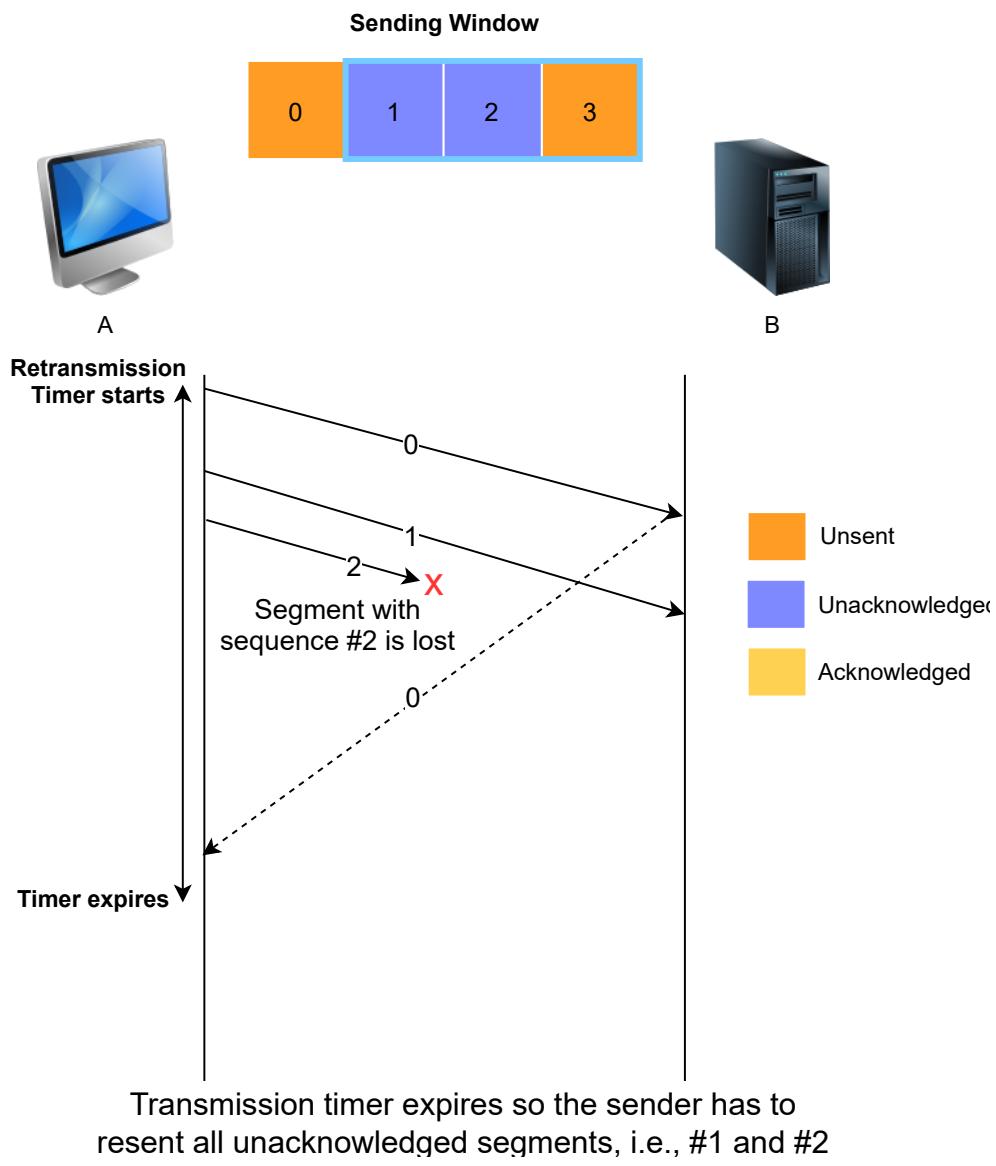
Segment #2 is discarded because that sequence number was not expected. #1 was expected

Go-back-n: example

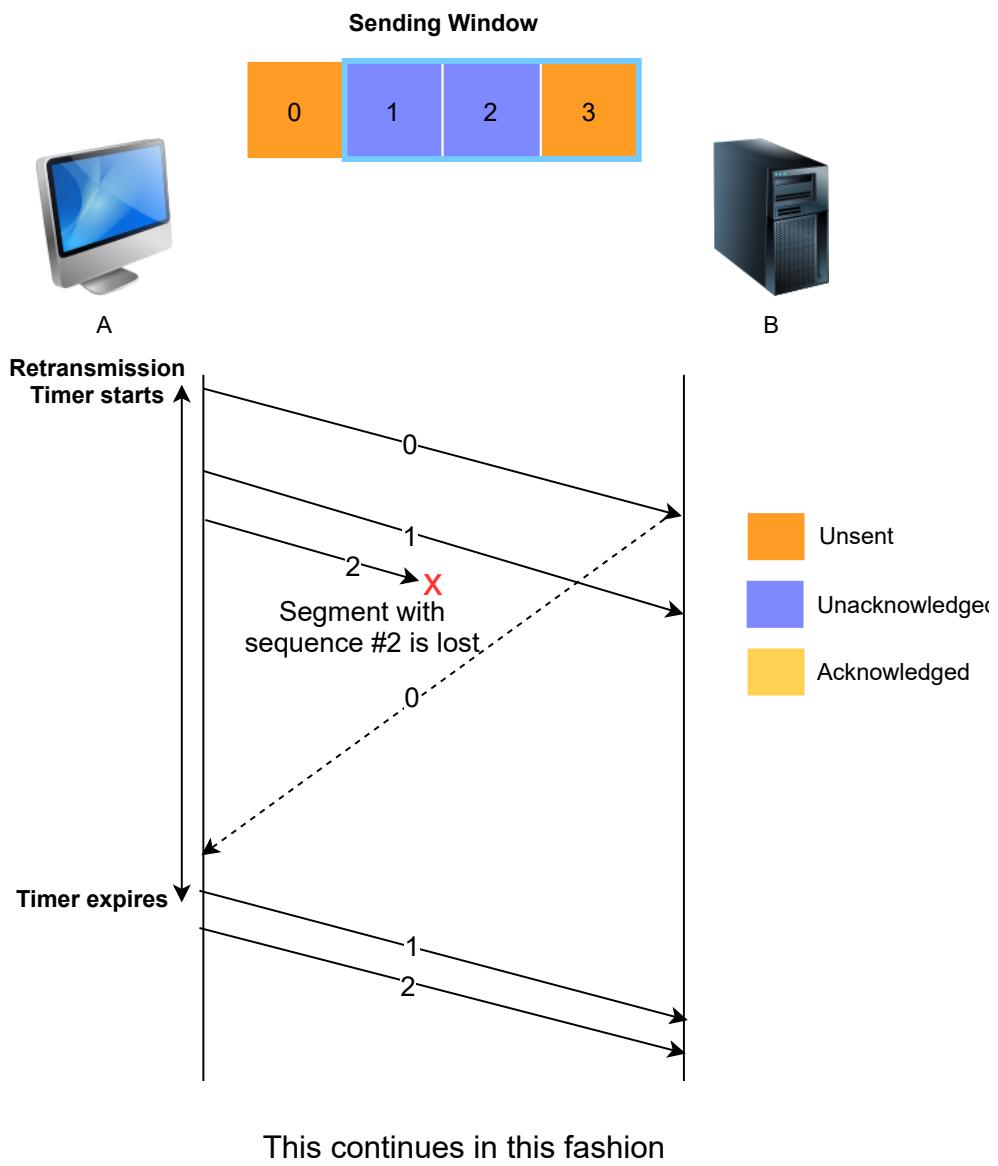


Acknowledgement for segment #0 is sent

Go-back-n: example



Go-back-n: example



Go-back-n: example

7 of 7



## Advantages of Go-back-n #

The main advantage of go-back-n is that it can be easily implemented, and it can also provide good performance when only a few segments are lost. But when there are many losses, the performance of go-back-n quickly drops for two reasons:

- The go-back-n receiver does not accept out-of-sequence segments.
- The go-back-n sender retransmits all unacknowledged segments once it has detected a loss.

Since the go-back-n protocol does not accept out of order segments, it can waste a lot of bandwidth if segments are frequently lost.

The Selective Repeat protocol attempts to remedy this by accepting out of order packets and only retransmitting packets that are corrupted or lost in the network.

## Selective Repeat #

- Uses a sliding window protocol just like go-back-n.
- The window size should be less than or equal to half the sequence numbers available. This avoids packets being identified incorrectly. Here's an **example**: suppose the window size is greater than half the buffer size.
  - Segment '1' is lost, hence the receiver expects a segment with sequence number 1 to be retransmitted.
  - Meanwhile, the window wraps around back to sequence number '1.'
  - The sender sends a new packet with sequence number 1 and the receiver perceives it to be the original one that it was expecting.
- Senders retransmit unacknowledged packets after a timeout or if a *NAK* (*negative acknowledgment/not acknowledged*) is received.
- The receiver acknowledges all correct packets.
- The receiver stores correct packets until they can be delivered in order to the upper application layer.



**Note:** NAKS Negative acknowledgements can be realized using positive acknowledgements (ACKs) of the last correctly received segment!

# Comparing to go-back-n #

In comparison to the go-back-n protocol, selective repeat conserves bandwidth, which is, the rate of data transfer across a path.

# Quick Quiz! #

- 1

A cumulative acknowledgement for sequence number n acknowledges the receipt of all sequences numbers upto and including \_\_\_\_.

COMPLETED 0%

1 of 5



In the next lesson, we'll start with UDP, a transport layer protocol.

# The User Datagram Protocol

This lesson gives an introduction to one of the protocols at the heart of the transport layer: UDP!

## WE'LL COVER THE FOLLOWING



- What is UDP?
- How It Works
- Structure of A UDP Datagram
  - Header
  - Data
- Quick Quiz!

## What is UDP? #

UDP, or **User Datagram Protocol**, is a transport layer protocol that works over the network layer's famous **Internet protocol** (which we'll look at in-depth in the [next chapter](#)). [RFC 768](#) is the official RFC for UDP.

## 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.

## Structure of A UDP Datagram #

### Header #

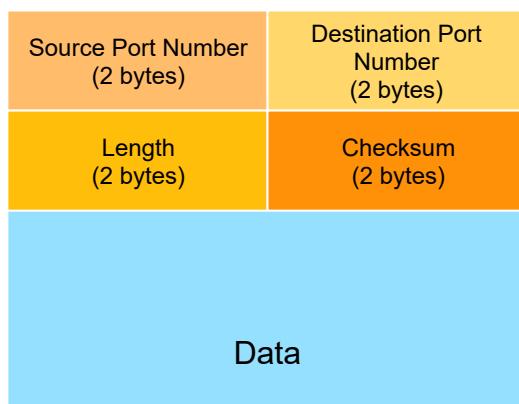
UDP prepends **four** 2-byte header fields to the data it receives from the application layer. So in total, a UDP header is **8 bytes** long. The fields are:

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!](#)

### 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 8-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:



## Quick Quiz! #

1

An application layer hands down 10 bytes to be sent in a UDP datagram. The value in the length field in the datagram will be \_\_\_\_.

COMPLETED 0%

1 of 4



Let's go on to look at how UDP does error detection, why UDP is used, and what well-known protocols are built on top of it in the next lesson!

# UDP Checksum Calculation & Why UDP?

Let's look at how the UDP checksum is calculated in-depth, why UDP would ever get used, and applications that use UDP.

## WE'LL COVER THE FOLLOWING



- Checksum Calculation
  - What if the checksum field gets corrupted?
- Why UDP?
- Well-Known Applications That Use UDP
  - Xbox Live
  - Name Translation
  - Network Management
- Quick Quiz!

UDP detects if any changes were introduced into a message while it traveled over the network. To do so, it appends a ‘checksum’ to the packet as a field that can be checked against the message itself to see if it was corrupted. It’s calculated the same way as in TCP. Here’s a refresher with some extra information:

## Checksum Calculation #

1. The payload and some of the headers (including some IP headers) are all divided into 16-bit words.
2. These words are then added together, wrapping any overflow around.
3. Lastly, the one’s complement of the resultant sum is taken and appended to the message as the checksum.



**Note** Also, note that if a message cannot be perfectly divided into 16-bit chunks, then the last word is padded to the right with zeros. This is

bit chunks, then the last word is padded to the right with zeros. This is

only for checksum calculation though! The actual message does not have these zeros.

Here's a visual of how the checksum for a datagram is calculated:

```
1110011001100110  
0101010101010101  
1100010000100010
```

Assume a message can be divided into these 3 16-bit words

1 of 24

```
1110011001100110  
+0101010101010101  
_____
```

We will start off by adding the first 2

2 of 24

$$\begin{array}{r} 1110011001100110 \\ +0101010101010101 \\ \hline 1 \end{array}$$

0 + 1 = 1

3 of 24

$$\begin{array}{r} 1110011001100110 \\ +0101010101010101 \\ \hline 11 \end{array}$$

1 + 0 = 1

4 of 24

$$\begin{array}{r} 1110011001100110 \\ +0101010101010101 \\ \hline 011 \end{array}$$

1 + 1 = 10 where the 1 is carried

5 of 24

$$\begin{array}{r} 1110011001100110 \\ +0101010101010101 \\ \hline 1011 \end{array}$$

0 + 0 + 1 (from the carry) = 1

6 of 24

$$\begin{array}{r} 1110011001100110 \\ +0101010101010101 \\ \hline 11011 \end{array}$$

$$0 + 1 = 1$$

7 of 24

$$\begin{array}{r} 1110011001100110 \\ +0101010101010101 \\ \hline 111011 \end{array}$$

$$1 + 0 = 1$$

8 of 24

$$\begin{array}{r} 1110011001100110 \\ +0101010101010101 \\ \hline 0111011 \end{array}$$

1 + 1 = 10 where 1 is carried

9 of 24

$$\begin{array}{r} 1110011001100110 \\ +0101010101010101 \\ \hline 110111011 \end{array}$$

0 + 1 = 1

10 of 24

$$\begin{array}{r} 1110011001100110 \\ +0101010101010101 \\ \hline 1110111011 \end{array}$$

$$1 + 0 = 1$$

11 of 24

$$\begin{array}{r} 1110011001100110 \\ +0101010101010101 \\ \hline 01110111011 \end{array}$$

$$1 + 0 = 1$$

12 of 24

$$\begin{array}{r} 1110011001100110 \\ +0101010101010101 \\ \hline 101110111011 \end{array}$$

$$1 + 0 + 0 = 1$$

13 of 24

$$\begin{array}{r} 1110011001100110 \\ +0101010101010101 \\ \hline 1101110111011 \end{array}$$

$$0 + 1 = 1$$

14 of 24

$$\begin{array}{r} \overset{1}{1}11001100\overset{1}{1}100110 \\ +0101010101010101 \\ \hline \textcolor{orange}{1}1101110111011 \end{array}$$

$$1 + 0 = 1$$

15 of 24

$$\begin{array}{r} \overset{1}{1}11001100\overset{1}{1}100110 \\ +0101010101010101 \\ \hline \textcolor{blue}{1}\textcolor{orange}{0}011101110111011 \end{array}$$

1 + 1 = 10. Now, notice that the 1 is an overflow, so we can wrap it around!

16 of 24

$$\begin{array}{r} \overset{1}{1} \overset{1}{1} \overset{1}{1} \overset{1}{1} \\ 1110011001100110 \\ +0101010101010101 \\ \hline 10011101110111011 \end{array}$$

We bring the '1' to the right and add it to the rest of the word!

17 of 24

$$\begin{array}{r} \overset{1}{1} \overset{1}{1} \overset{1}{1} \overset{1}{1} \\ 1110011001100110 \\ +0101010101010101 \\ \hline 0011101110111011 \\ \quad \quad \quad 1 \end{array}$$

We bring the '1' to the right and add it to the rest of the word!

18 of 24

$$\begin{array}{r}
 \overset{1}{1} \overset{1}{1} \overset{1}{1} \overset{1}{0} 0110011001100110 \\
 + 010101010101010101 \\
 \hline
 \textcolor{orange}{0011101110111011} \\
 + \textcolor{blue}{1} \\
 \hline
 \end{array}$$

We bring the '1' to the right and add it to the rest of the word!

**19** of 24

$$\begin{array}{r}
 \overset{1}{1} \overset{1}{1} \overset{1}{1} \overset{1}{0} 0110011001100110 \\
 + 010101010101010101 \\
 \hline
 \textcolor{orange}{0011101110111011} \\
 + \textcolor{blue}{1} \\
 \hline
 \textcolor{orange}{0011101110111100}
 \end{array}$$

We sum using the same rules as before to get this

**20** of 24

$$\begin{array}{r} 0011101110111100 \\ + 1100010000100010 \\ \hline \end{array}$$

← third word

We sum using the same rules as before to get this

21 of 24

$$\begin{array}{r} 0011101110111100 \\ + 1100010000100010 \\ \hline 0111111111011110 \end{array}$$

← Final sum

We sum using the same rules as before to get this

22 of 24

$$\begin{array}{r} 0011101110111100 \\ + 1100010000100010 \\ \hline 0111111111011110 \end{array} \quad \text{Final sum}$$

↓

$$100000000010000 \quad \text{Final checksum}$$

We sum using the same rules as before to get this

23 of 24

100000000010001

The final checksum!

24 of 24

At the receiving end, UDP sums the message in 16-bit words and adds the sum to the sent checksum. If the result is **1111111111111111**, the message was not corrupted. If the result is otherwise, it was.

## What if the checksum field gets corrupted? #

If the checksum itself gets corrupted, UDP will assume that the message has an error.

## 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](#).
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. Going on points 3 and 4, **UDP allows custom protocols to be built on top of it.**
  - In fact, Google's transport layer protocol, **Quick UDP Internet Connections (QUIC)**, pronounced *quick*, is an experimental transport layer network protocol built on top of UDP and designed by Google. The overall goal is to reduce latency compared to that of TCP. It's used by most connections from the Chrome web browser to Google's servers!
5. 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 #

# Well-known Applications That Use UDP #

## Xbox Live #

Xbox live is built on UDP.



Yes, Xbox live runs on UDP!

## Name Translation #

Yes, [DNS](#) uses UDP! In the case of failed message delivery, DNS either:

1. Resends the message.
2. Sends the message to some other server.
3. Gives a failure message.

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

## Network Management #

Network management and network monitoring is done using a protocol called [Simple Network Management Protocol](#) and it runs on UDP as well.

## Quick Quiz! #

1

An application implements its own mechanisms of checksums and retransmissions. UDP is the ideal choice of transport layer protocol for this application.

COMPLETED 0%

1 of 4



Let's look at some actual live UDP packets in the next lesson with TCPDUMP!

# Exercise: Capturing UDP Packets

We'll now look at a command-line tool that allows us to capture UDP packets.

## WE'LL COVER THE FOLLOWING ^

- What is `tcpdump` ?
  - Sample Output
  - Counting Packets with `-c`
  - Printing PCAP Files With `-r`
- Looking at Real UDP Packet Headers
- Try it Yourself!

Let's get into viewing real packets.

## What is `tcpdump`? #

`tcpdump` 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!

```
tcpdump
```

Packets will start getting printed rapidly to give a comprehensive view of the traffic.

## Sample Output #

However, some might not find it to be very helpful because it does not allow for a more **zoomed-in and fine-grained dissection of the packets**, which is the main purpose of `tcpdump` (it's technically a packet *analyzer*). So you might want to consider using some flags to filter relevant packets out.

```
win 1419, options [nop,nop,TS val 3469904026 ecr 41304754], length 0
08:12:55.043775 IP ed-live-vm-gl-small-024668f6-3cbb-4480-ae19-04ae92fe20b8.c.educative-exec-env.internal.8890 > reverse-proxy-instance-group-j619.c.educative-exec-env.internal.49280: Flags [P.], seq 168563
:169182, ack 1, win 229, options [nop,nop,TS val 41304765 ecr 3469904026], length 619
08:12:55.049253 IP ed-live-vm-gl-small-024668f6-3cbb-4480-ae19-04ae92fe20b8.c.educative-exec-env.internal.8890 > reverse-proxy-instance-group-j619.c.educative-exec-env.internal.49280: Flags [P.], seq 169182
:169522, ack 1, win 229, options [nop,nop,TS val 41304770 ecr 3469904026], length 340
08:12:55.049887 IP reverse-proxy-instance-group-j619.c.educative-exec-env.internal.49280 > ed-live-vm-gl-small-024668f6-3cbb-4480-ae19-04ae92fe20b8.c.educative-exec-env.internal.8890: Flags [.], ack 169522,
win 1419, options [nop,nop,TS val 3469904037 ecr 41304765], length 0
08:12:55.055275 IP ed-live-vm-gl-small-024668f6-3cbb-4480-ae19-04ae92fe20b8.c.educative-exec-env.internal.8890 > reverse-proxy-instance-group-j619.c.educative-exec-env.internal.49280: Flags [P.], seq 169522
:170141, ack 1, win 229, options [nop,nop,TS val 41304776 ecr 3469904037], length 619
08:12:55.060738 IP ed-live-vm-gl-small-024668f6-3cbb-4480-ae19-04ae92fe20b8.c.educative-exec-env.internal.8890 > reverse-proxy-instance-group-j619.c.educative-exec-env.internal.49280: Flags [P.], seq 170141
:170481, ack 1, win 229, options [nop,nop,TS val 41304782 ecr 3469904037], length 340
08:12:55.061384 IP reverse-proxy-instance-group-j619.c.educative-exec-env.internal.49280 > ed-live-vm-gl-small-024668f6-3cbb-4480-ae19-04ae92fe20b8.c.educative-exec-env.internal.8890: Flags [.], ack 170481,
win 1419, options [nop,nop,TS val 3469904048 ecr 41304776], length 0
08:12:55.065727 IP ed-live-vm-gl-small-024668f6-3cbb-4480-ae19-04ae92fe20b8.c.educative-exec-env.internal.8890 > reverse-proxy-instance-group-j619.c.educative-exec-env.internal.49280: Flags [P.], seq 170481
:171100, ack 1, win 229, options [nop,nop,TS val 41304787 ecr 3469904048], length 619
08:12:55.071194 IP ed-live-vm-gl-small-024668f6-3cbb-4480-ae19-04ae92fe20b8.c.educative-exec-env.internal.8890 > reverse-proxy-instance-group-j619.c.educative-exec-env.internal.49280: Flags [P.], seq 171100
```

... what??

## Useful **tcpdump** Flags

Here are some flags that you might find useful in your exploration of this tool. You can find more details about each on [tcpdump's Manpage](#)

### Saving **tcpdump** Output to a File with **-w**

Instead of having all the output print to the console, we can save it to view at a later date or to feed into another program to analyze.

```
tcpdump -w filename.ext
```



Let's zoom into the traffic a bit

Try using this tool in the following code executable.

```
tcpdump -w output.pcap # Saving output to a file called 'output.pcap'
```



The file **output.pcap** will have all the packets saved to it. Try running this command in the terminal below. Note that the process does not exit without a

keyboard interrupt. The next flag will help us stop packet capture in a predetermined fashion.



**Note** .pcap files are used to store the packet data of a network.

Packet analysis programs such as [Wireshark](#) (think of it like tcpdump with a GUI) export and import packet captures in pcap files.

## Counting Packets with `-c #`

This flag makes `tcpdump` capture a defined number of packets. Here's how it's used.

```
tcpdump -w output.pcap -c 10 # Capturing 10 packets
```



You can't view the file just yet. Let's do it next.

## Printing PCAP Files With `-r #`

Great! Let's actually **read** `.pcap` files now. Here's how to do it.

```
tcpdump -w output.pcap -c 10 # Capturing 10 packets  
tcpdump -r output.pcap # Printing the captured packets in a PCAP file
```



We've gotten pretty far with this. There are plenty of other flags and arguments you could give to `tcpdump` to make it capture packets precisely as per your requirements.

## Looking at Real UDP Packet Headers #

Here's a script to capture and print one UDP packet.

Note that the code *may* time out before it actually captures a packet. We would suggest running this one on the [terminal](#).

```
tcpdump udp -X -c 1 # Capturing 1 UDP packet
```



The `-X` flag just prints the payload of the packet (the data) in both hex and ASCII.

Here's what the output is depicting.

```
root@educative:/# tcpdump udp -X -c 1
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on ens4, link-type EN10MB (Ethernet), capture size 262144 bytes
07:39:03.760504 IP ed-live-vm-g1-small-aa299037-fa26-4cd9-aaa8-e28d46ffadb5.c.educative-exec-env.intern
al.ntp > metadata.google.internal.ntp: NTPv4, Client, length 48
    0x0000: 45b8 004c fdfe 4000 4011 dd42 0a80 0031 E..L..@.B...
    0x0010: a9fe a9fe 007b 0038 5ef7 2303 07e8 ....{.{.8^.#...
    0x0020: 0000 004d 0000 03ac a9fe a9fe e0e9 1f09 ...M.....
    0x0030: a49a 060d e0e9 2095 c299 lcde e0e9 2095 .....
    0x0040: c2de ecd6 e0e9 2117 c2ad 9902 ..!....
```

The command that starts tcpdump is on the first line

1 of 11

```
root@educative:/# tcpdump udp -X -c 1
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on ens4, link-type EN10MB (Ethernet), capture size 262144 bytes
07:39:03.760504 IP ed-live-vm-g1-small-aa299037-fa26-4cd9-aaa8-e28d46ffadb5.c.educative-exec-env.intern
al.ntp > metadata.google.internal.ntp: NTPv4, Client, length 48
    0x0000: 45b8 004c fdfe 4000 4011 dd42 0a80 0031 E..L..@.B...
    0x0010: a9fe a9fe 007b 0038 5ef7 2303 07e8 ....{.{.8^.#...
    0x0020: 0000 004d 0000 03ac a9fe a9fe e0e9 1f09 ...M.....
    0x0030: a49a 060d e0e9 2095 c299 lcde e0e9 2095 .....
    0x0040: c2de ecd6 e0e9 2117 c2ad 9902 ..!....
```

Some tcpdump output including the interface being monitored (ens4) and the link type (ethernet)

Here's some general output from tcpdump itself

2 of 11

```

root@educative:/# tcpdump udp -X -c 1
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on ens4, link-type EN10MB (Ethernet), capture size 262144 bytes } Some tcpdump output
07:39:03.760504 IP ed-live-vm-g1-small-aa299037-fa26-4cd9-aaa8-e28d46ffadb5.c.educative-exec-env.internal.nntp > metadata.google.internal.ntp: NTPv4, Client, length 48
    0x0000: 45b8 004c fdf8 4000 4011 dd42 0a80 0031 E..L..@.e..B...1
    0x0010: a9fe a9fe 007b 0038 5ef7 2303 07e8 .....{.8^.#...
    0x0020: 0000 004d 0000 03ac a9fe a9fe e0e9 1f09 ...M.....
    0x0030: a49a 060d e0e9 2095 c299 1cde e0e9 2095 .....
    0x0040: c2de ecd6 e0e9 2117 c2ad 9902 .....!.....

```

Time stamp of the packet

hostname was resolved IP address does this

The format of the next line is like so 'IP address of sender > IP address of receiver'. Notice that the IP addresses have been resolved into hostnames. tcpdump does this by default. If you wish to see the actual IP address, pass in the '-n' flag. Also notice the time stamp.

3 of 11

```

root@educative:/# tcpdump udp -X -c 1
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on ens4, link-type EN10MB (Ethernet), capture size 262144 bytes } Some tcpdump output
07:39:03.760504 IP ed-live-vm-g1-small-aa299037-fa26-4cd9-aaa8-e28d46ffadb5.c.educative-exec-env.internal.nntp > metadata.google.internal.ntp: NTPv4, Client, length 48
    0x0000: 45b8 004c fdf8 4000 4011 dd42 0a80 0031 E..L..@.e..B...1
    0x0010: a9fe a9fe 007b 0038 5ef7 2303 07e8 .....{.8^.#...
    0x0020: 0000 004d 0000 03ac a9fe a9fe e0e9 1f09 ...M.....
    0x0030: a49a 060d e0e9 2095 c299 1cde e0e9 2095 .....
    0x0040: c2de ecd6 e0e9 2117 c2ad 9902 .....!.....

```

Time stamp of the packet

IP address of receiver

hostname was resolved address. tc this by

IP address (or hostname really) of the receiver

4 of 11

```

root@educative:/# tcpdump udp -X -c 1
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on ens4, link-type EN10MB (Ethernet), capture size 262144 bytes } Some tcpdump output
07:39:03.760504 IP ed-live-vm-g1-small-aa299037-fa26-4cd9-aaa8-e28d46ffadb5.c.educative-exec-env.internal.nntp > metadata.google.internal.ntp: NTPv4, Client, length 48
    0x0000: 45b8 004c fdf8 4000 4011 dd42 0a80 0031 E..L..@.e..B...1
    0x0010: a9fe a9fe 007b 0038 5ef7 2303 07e8 .....{.8^.#...
    0x0020: 0000 004d 0000 03ac a9fe a9fe e0e9 1f09 ...M.....
    0x0030: a49a 060d e0e9 2095 c299 1cde e0e9 2095 .....
    0x0040: c2de ecd6 e0e9 2117 c2ad 9902 .....!.....

```

Time stamp of the packet

IP address of receiver

hostname It was reso an IP ad tcpdump c by def

The message in hexadecimal

The message in ASCII

Let's dissect the datagram now

5 of 11

```

root@educative:/# tcpdump udp -X -c 1
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on ens4, link-type EN10MB (Ethernet), capture size 262144 bytes } Some tcpdump output
07:39:03.760504 IP ed-live-vm-g1-small-aa299037-fa26-4cd9-aaa8-e28d46ffadb5.c.educative-exec-env.internal.nntp > metadata.google.internal.ntp: NTPv4, Client, length 48
    0x0000: 45b8 004c fdf8 4000 4011 dd42 0a80 0031 E..L..e..e..B...1
    0x0010: a9fe a9fe 007b 007b 0038 5ef7 2303 07e8 .....{.{.8^.#...
    0x0020: 0000 004d 0000 03ac a9fe a9fe e0e9 1f09 ....M.....
    0x0030: a49a 060d e0e9 2095 c299 1cde e0e9 2095 .....
    0x0040: c2de ecd6 e0e9 2117 c2ad 9902 ..!.....

```

Time stamp of the packet  
IP address of receiver  
10 word IP Header  
The message in Hex

hostname was resolved IP address does this

The first 160 bits are the IP header. Note that a single hex digit is exactly 4 bits so that means the header is of  $160/4 = 40$  hex digits or  $40/4 = 10$  blocks. We can safely ignore it for now!

6 of 11

```

root@educative:/# tcpdump udp -X -c 1
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on ens4, link-type EN10MB (Ethernet), capture size 262144 bytes } Some tcpdump output
07:39:03.760504 IP ed-live-vm-g1-small-aa299037-fa26-4cd9-aaa8-e28d46ffadb5.c.educative-exec-env.internal.nntp > metadata.google.internal.ntp: NTPv4, Client, length 48
    0x0000: 45b8 004c fdf8 4000 4011 dd42 0a80 0031 E..L..e..e..B...1
    0x0010: a9fe a9fe 007b 007b 0038 5ef7 2303 07e8 .....{.{.8^.#...
    0x0020: 0000 004d 0000 03ac a9fe a9fe e0e9 1f09 ....M.....
    0x0030: a49a 060d e0e9 2095 c299 1cde e0e9 2095 .....
    0x0040: c2de ecd6 e0e9 2117 c2ad 9902 ..!.....

```

Time stamp of the packet  
IP address of receiver  
The message in Hex

4 block UDP Header

hostname was resolved IP address does this

The UDP header is of 64 bits 4 blocks. Each block represents one UDP field.

7 of 11

```

root@educative:/# tcpdump udp -X -c 1
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on ens4, link-type EN10MB (Ethernet), capture size 262144 bytes } Some tcpdump output
07:39:03.760504 IP ed-live-vm-g1-small-aa299037-fa26-4cd9-aaa8-e28d46ffadb5.c.educative-exec-env.internal.nntp > metadata.google.internal.ntp: NTPv4, Client, length 48
    0x0000: 45b8 004c fdf8 4000 4011 dd42 0a80 0031 E..L..e..e..B...1
    0x0010: a9fe a9fe 007b 007b 0038 5ef7 2303 07e8 .....{.{.8^.#...
    0x0020: 0000 004d 0000 03ac a9fe a9fe e0e9 1f09 ....M.....
    0x0030: a49a 060d e0e9 2095 c299 1cde e0e9 2095 .....
    0x0040: c2de ecd6 e0e9 2117 c2ad 9902 ..!.....

```

Time stamp of the packet  
IP address of receiver  
The message in Hex

The source and destination ports in hex. These ports are '123' in decimal. This is an example of the source and destination both using well known port numbers.

4 block UDP Header

hostname It was resolved an IP address by tcpdump default

The UDP header is of 64 bits i.e., 4 blocks. Each block represents one UDP field. The first two fields are the source and destination ports which are both 007b or port numbers 123 in decimal.

8 of 11

```

root@educative:/# tcpdump udp -X -c 1
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode } Some tcpdump output
listening on ens4, link-type EN10MB (Ethernet), capture size 262144 bytes
07:39:03.760504 IP ed-live-vm-g1-small-aa299037-fa26-4cd9-aaa8-e28d46ffadb5.c.educative-exec-env.internal.nntp > metadata.google.internal.ntp: (NTPv4) Client, length 48
    0x0000: 45b8 004c fdfe 4000 4011 dd42 0a80 0031 E..L..@.B...1
    0x0010: a9fe a9fe 007b 0038 5ef7 2303 07e8 .....{.8^.#...
    0x0020: 0000 004d 0000 03ac a9fe a9fe e0e9 1f09 ...M.....
    0x0030: a49a 060d e0e9 2095 c299 1cde e0e9 2095 .....
    0x0040: c2de ecd6 e0e9 2117 c2ad 9902 .....!

```

Time stamp of the packet  
 IP address of receiver  
 The message in Hex  
 The message is part of the NTP protocol as can be inferred from the UDP header as well  
 These first two hex blocks represent source and destination ports. These ports are '123' in decimal.  
 4 block UDP Header  
 hostame It was resc an IP ac tcpdump dc def

Note that port 123 is reserved for the NTP protocol (which runs on UDP) as shown by the output here.

9 of 11

```

root@educative:/# tcpdump udp -X -c 1
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode } Some tcpdump output
listening on ens4, link-type EN10MB (Ethernet), capture size 262144 bytes
07:39:03.760504 IP ed-live-vm-g1-small-aa299037-fa26-4cd9-aaa8-e28d46ffadb5.c.educative-exec-env.internal.nntp > metadata.google.internal.ntp: (NTPv4) Client, length 48
    0x0000: 45b8 004c fdfe 4000 4011 dd42 0a80 0031 E..L..@.B...1
    0x0010: a9fe a9fe 007b 0038 5ef7 2303 07e8 .....{.8^.#...
    0x0020: 0000 004d 0000 03ac a9fe a9fe e0e9 1f09 ...M.....
    0x0030: a49a 060d e0e9 2095 c299 1cde e0e9 2095 .....
    0x0040: c2de ecd6 e0e9 2117 c2ad 9902 .....!

```

Time stamp of the packet  
 IP address of receiver  
 The message in Hex  
 The message is part of the NTP protocol as can be inferred from the UDP header as well  
 These first two hex blocks represent source and destination ports. These ports are '123' in decimal.  
 4 block UDP Header  
 hostame resolved fr tcpdump dc  
 The length of the message is '0038' or 56 in decimal whereas the checksum is 5ef7

The next two fields are the length and the checksum!

10 of 11

```

root@educative:/# tcpdump udp -X -c 1
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode } Some tcpdump output
listening on ens4, link-type EN10MB (Ethernet), capture size 262144 bytes
07:39:03.760504 IP ed-live-vm-g1-small-aa299037-fa26-4cd9-aaa8-e28d46ffadb5.c.educative-exec-env.internal.nntp > metadata.google.internal.ntp: (NTPv4) Client, length 48
    0x0000: 45b8 004c fdfe 4000 4011 dd42 0a80 0031 E..L..@.B...1
    0x0010: a9fe a9fe 007b 0038 5ef7 2303 07e8 .....{.8^.#...
    0x0020: 0000 004d 0000 03ac a9fe a9fe e0e9 1f09 ...M.....
    0x0030: a49a 060d e0e9 2095 c299 1cde e0e9 2095 .....
    0x0040: c2de ecd6 e0e9 2117 c2ad 9902 .....!

```

Time stamp of the packet  
 IP address of receiver  
 The message in Hex  
 The message is part of the NTP protocol as can be inferred from the UDP header as well  
 These first two hex blocks represent source and destination ports. These ports are '123' in decimal.  
 4 block UDP Header  
 hostame i was resolv IP address: does this  
 The length of the message is '0038' or 56 in decimal whereas the checksum is 5ef7

That concludes our inspection of a UDP packet. Explore this more! Try capturing a packet on the command line below and try dissecting it!



## Try it Yourself! #

You can try all the commands in this terminal. [Click here to go back](#)

● Terminal



---

In the next lesson, we'll learn about the transmission control protocol!

# The Transmission Control Protocol

In this lesson, we'll look at a quick overview of TCP and some famous applications that use it.

## WE'LL COVER THE FOLLOWING



- Introduction to TCP
- An Analogy: Talking on a Cell Phone
- What TCP Does
- Well-Known Applications That Use TCP
  - File Transfer
  - Secure Shell **SSH**
  - Email
  - Web Browsing
- Quick Quiz!

## Introduction to TCP #

TCP, or the **transmission control protocol**, is one of the two key protocols of the transport layer. TCP is what makes most modern applications as enjoyable and reliable as they are. HTTP's implementation, for example, would be very complex, if it weren't for TCP.

TCP is a robust protocol meant to adapt to a diverse range of network topologies, bandwidths, delays, message sizes, and other varying factors that exist in the network layer.

## An Analogy: Talking on a Cell Phone #

TCP is a **connection-oriented protocol** unlike UDP. The connection orientedness is like a **phone call** because a connection is established before communication takes place, and then we hang up. Some scenarios that might occur on a phone call with a friend are relevant to connection-oriented

occur on a phone call with a friend are relevant to connection-oriented protocols in the Internet, too. For example,

- What if you **can't understand** what your friend is saying?
  - Should you ask them to **repeat what they said?**
- What if you haven't **heard them speak for a while**?
  - Does your friend simply **not have anything to say?**
  - Has the phone lost reception and **disconnected**?
  - Should you **keep talking**? If so, **for how long**?
  - Should both you and your friend periodically say “mhmm” to indicate that you are present and listening?

TCP, being THE connection-oriented transport layer protocol for the Internet, has mechanisms to solve these problems.

## What TCP Does #

Here are some key responsibilities of the protocol.

1. **Send data** at an appropriate transmission rate. It should be a fast enough 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.

## Application Layer

The application layer sends data in an unfragmented stream

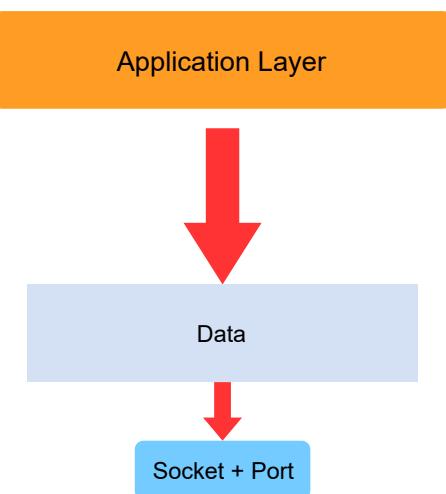
1 of 5

## Application Layer



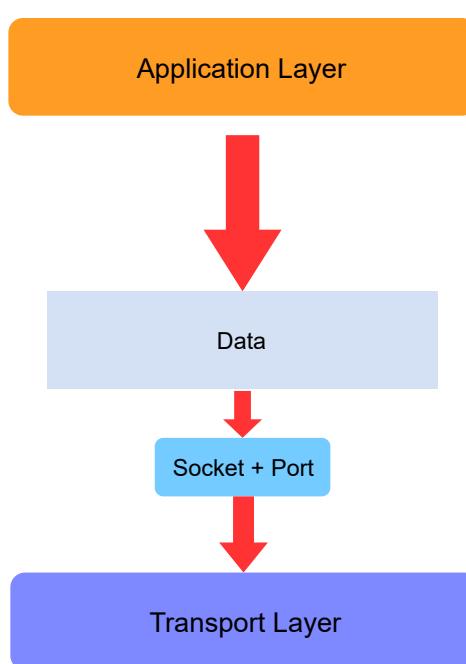
## Data

The application layer sends data in an unfragmented stream

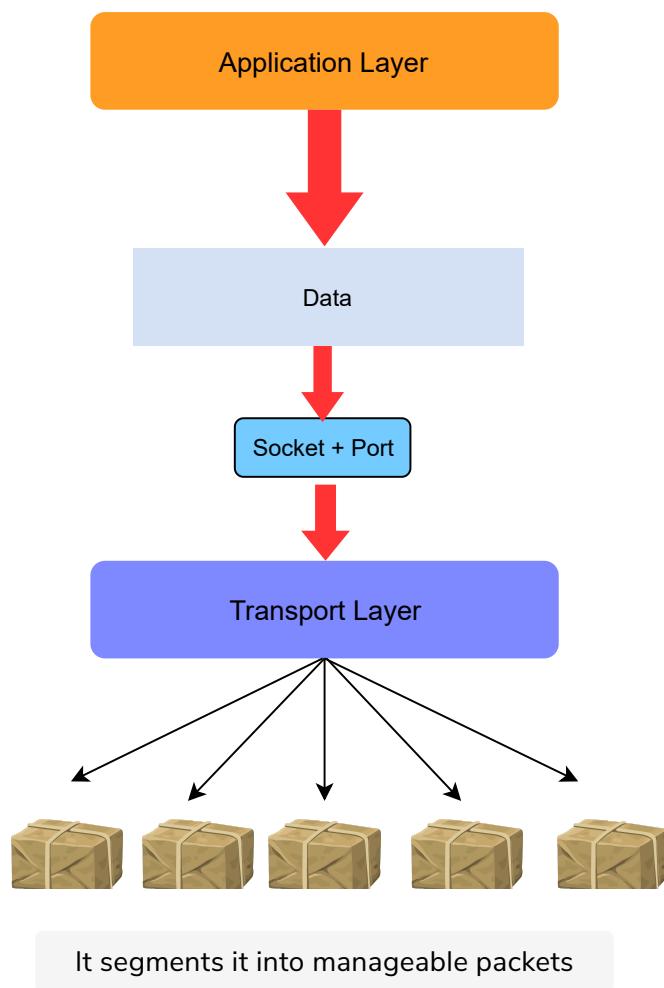


It sends it through to the transport layer through a socket and port

3 of 5



The transport layer receives the stream of unfragmented data



5 of 5



3. **Identify and retransmit messages** that do not get delivered. The network layer cannot be relied upon to deliver messages.
4. **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.

## Well-Known Applications That Use TCP #

### 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** #

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 #

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 #

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

## Quick Quiz! #

1

The fact that TCP is connection oriented guarantees in-order delivery of application messages.

COMPLETED 0%

1 of 4



Now that we have an overview of what TCP is and what it does, let's look at some of its key properties in the next lesson!

# Key Features of the Transmission Control Protocol

Here are some key properties of TCP. These are important to know to understand the design of the protocol.

## WE'LL COVER THE FOLLOWING ^

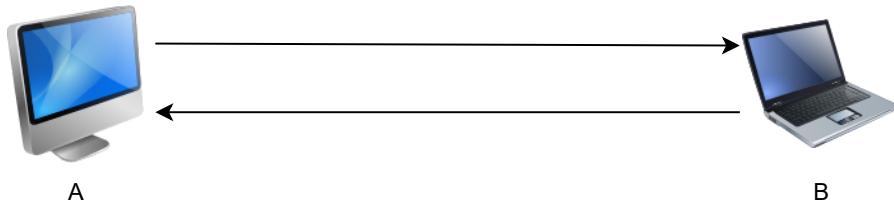
- Connection Oriented
- Full Duplex
- Point-to-point Transmission

## Connection Oriented #

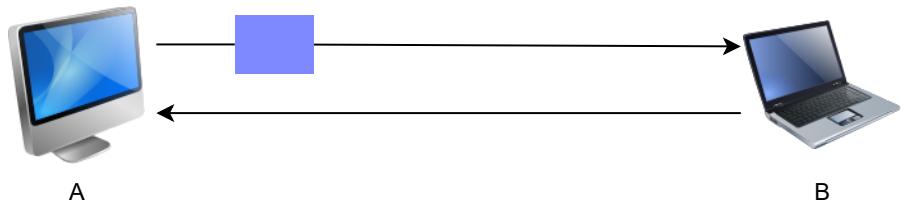
TCP itself is **connection-oriented** and creates a long term connection between hosts. The connection remains until a certain termination procedure is followed.

## Full Duplex #

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

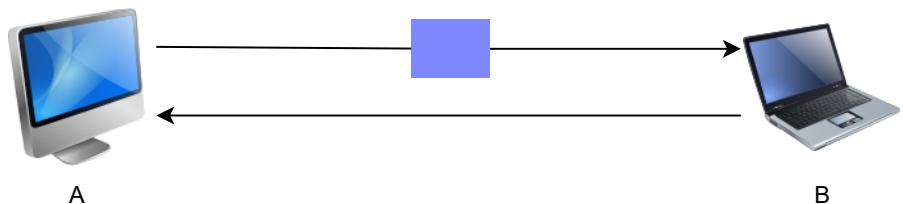


Suppose two hosts are connected to each other over TCP



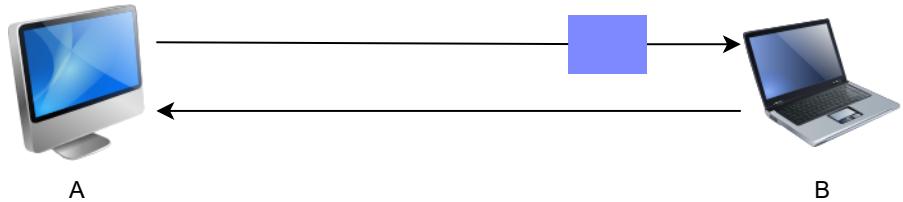
Host A can send a message to host B

2 of 12



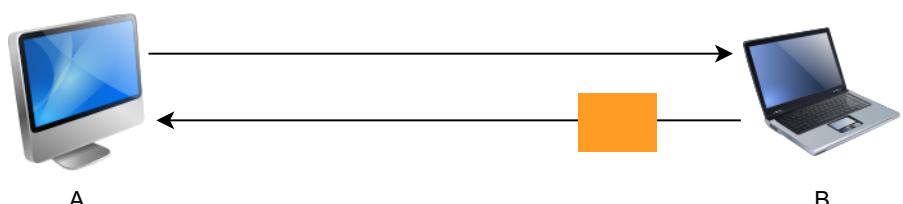
Host A can send a message to host B

3 of 12



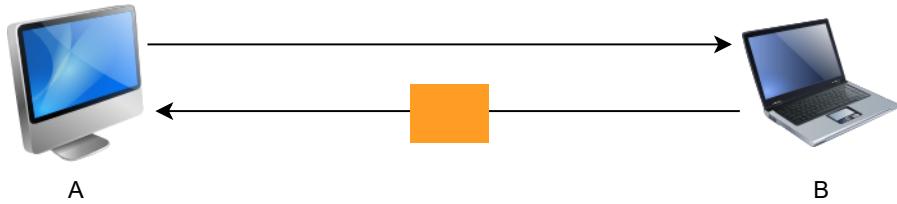
Host A can send a message to host B

4 of 12



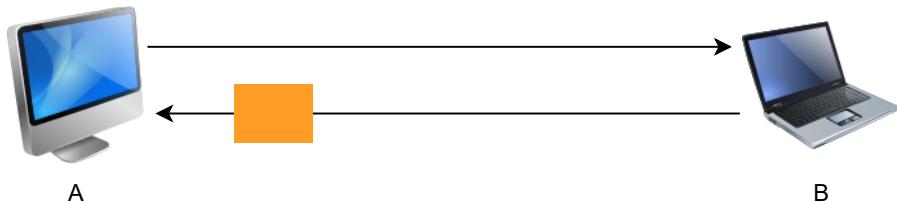
Similarly, host B can send a message to host A

5 of 12



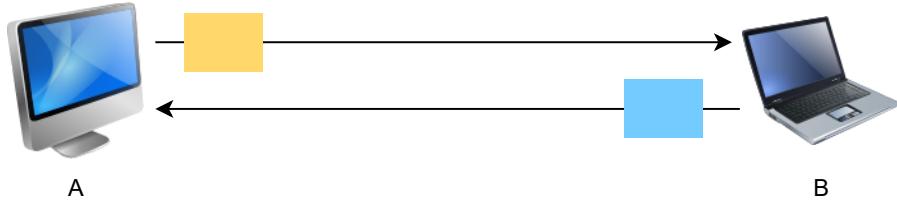
Similarly, host B can send a message to host A

6 of 12



Similarly, host B can send a message to host A

7 of 12



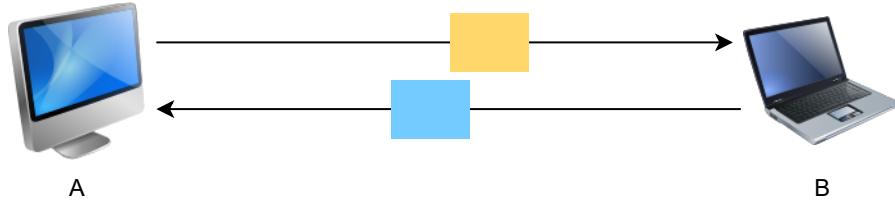
Also, both hosts can send each other messages simultaneously because TCP is full duplex!

8 of 12



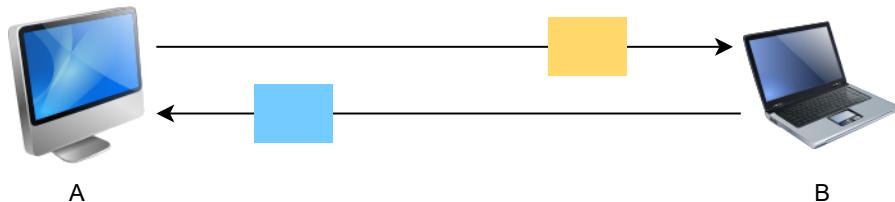
Also, both hosts can send each other messages simultaneously because TCP is full duplex!

9 of 12



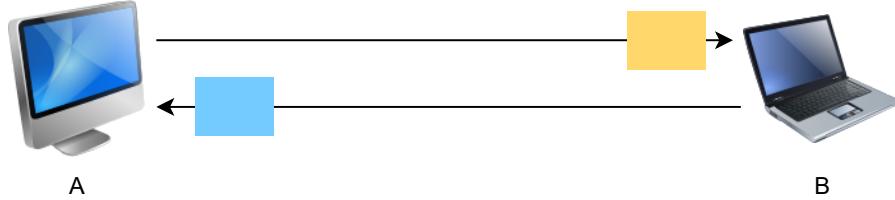
Also, both hosts can send each other messages simultaneously because TCP is full duplex!

10 of 12



Also, both hosts can send each other messages simultaneously because TCP is full duplex!

11 of 12



Also, both hosts can send each other messages simultaneously because TCP is full duplex!

12 of 12



## Point-to-point Transmission #

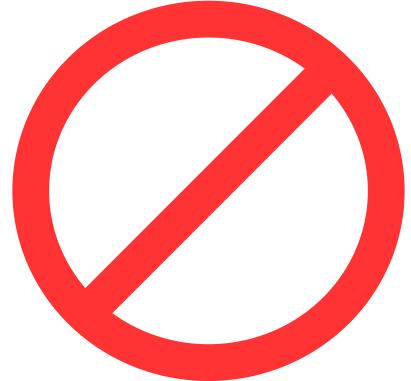
TCP connections have **exactly two endpoints!** This means that **broadcasting** or **multicasting** is not possible with TCP.

## 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.



TCP detects and handles errors!

## Congestion Control

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

---

Now we understand what TCP is and its general design principles. Let's study TCP headers in the next lesson!

# TCP Segment Header

We'll now study TCP headers. They're are far more complex than UDP headers and really are what allow for TCP to work properly!

## WE'LL COVER THE FOLLOWING ^

- Introduction
- Source and Destination Ports
- Sequence Number
- Acknowledgement Number
  - Example
- Header Length
- Reserved Field
- Quick Quiz!

## Introduction #

TCP headers play a crucial role in the implementation of the protocol. In fact, TCP segments without actual data and with headers are completely valid. They're actually used quite often!

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

## Source and Destination Ports #

Source Port Number (2 bytes)		Destination Port Number (2 bytes)			
Sequence Number (4 bytes)					
Acknowledgment Number (4 bytes)					
Header Length (4 bits)	Reserved (4 bits)	8 flags (8 bits)	Window Size (2 bytes)		
Checksum (2 bytes)		Urgent Pointer (2 bytes)			
Options and Padding (40 bytes)					

The source and destination ports are the first fields of the TCP header.

The source and destination port numbers are self-explanatory. They are exactly like the source and destination ports in UDP. Just for a refresher though, the source port is the port of the socket of the application that is **sending** the segment and the **destination port** is the port of the socket of the **receiving** application. The size of each field is **two bytes**.

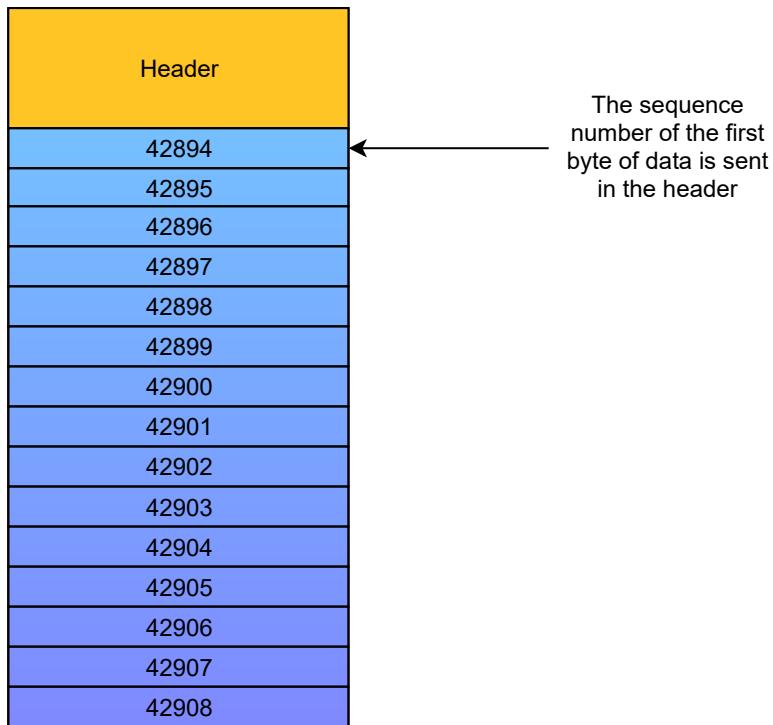
## Sequence Number #

Source Port Number (2 bytes)		Destination Port Number (2 bytes)			
Sequence Number (4 bytes)					
Acknowledgment Number (4 bytes)					
Header Length (4 bits)	Reserved (4 bits)	8 flags (8 bits)	Window Size (2 bytes)		
Checksum (2 bytes)		Urgent Pointer (2 bytes)			
Options and Padding (40 bytes)					

The sequence number is the second field of the TCP header. It represents the first byte of data in the TCP segment.

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

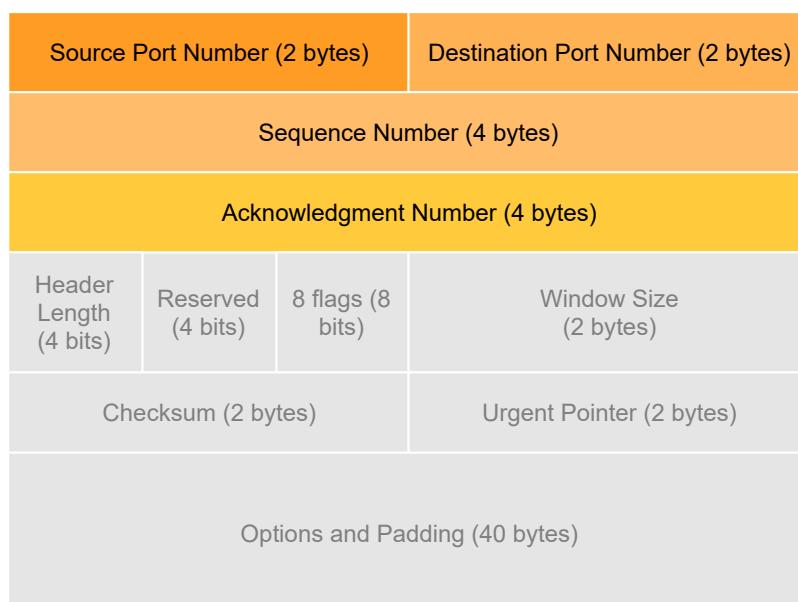
sequence number of the first byte of data in the segment.



Each byte of a TCP segment is labeled with a sequence number.

 Note The initial sequence number is a randomly generated number between 0 and  $2^{32} - 1$ .

## Acknowledgement Number #



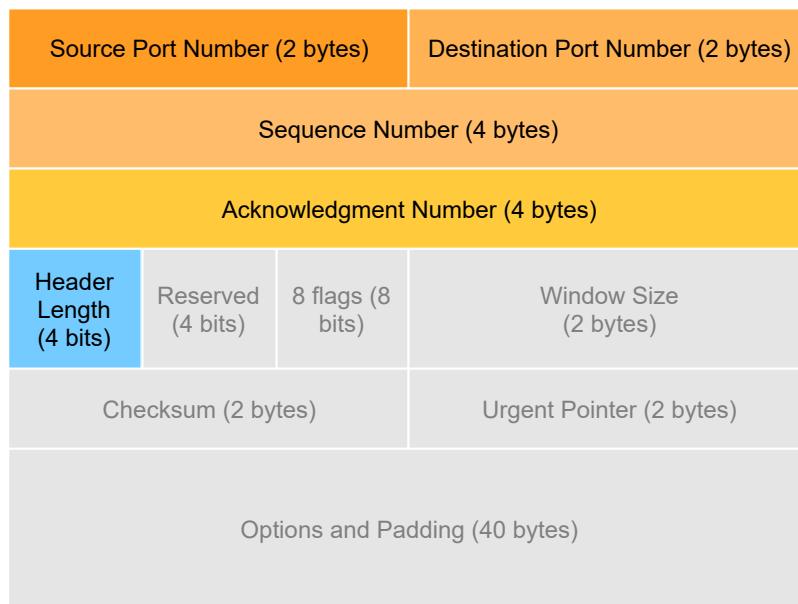
The third field in the TCP header is the acknowledgement number

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

## Example #

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 42908. This helps TCP to identify if a segment was missing or out of order.

## Header Length #

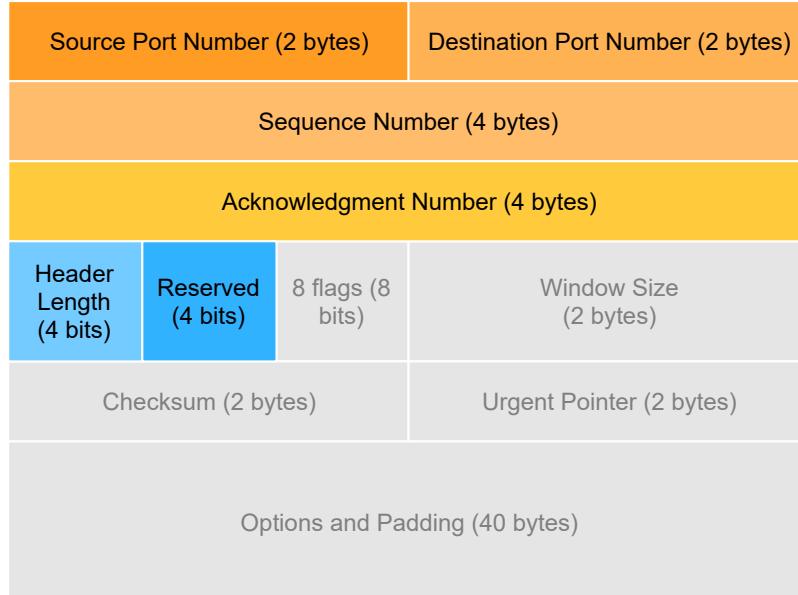


The length of the TCP header is specified here. This helps the receiving end to identify where the header ends and the data starts from.



**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

## Reserved Field #



The reserved bits serve as an offset and are left for potential future use

The header has a 4-bit field that is reserved and is always set to 0. This field aligns the total header size to be in multiples of 4 (as we saw was necessary for the header length to be processed).

## Quick Quiz! #

1

Given an initial sequence number of 255 and 50 bytes sent in a TCP segment, what will be the value of the sequence number field in the next TCP packet header?

COMPLETED 0%

1 of 4



We'll continue dissecting the TCP headers in the next lesson!

# TCP Header Flags

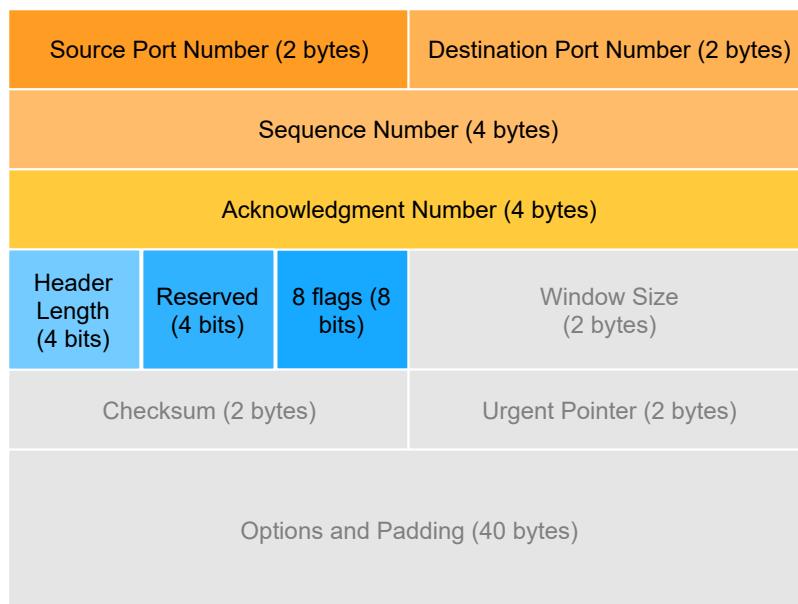
In the last lesson, we discussed eight fields of the TCP header. Let's now discuss the last few!

## WE'LL COVER THE FOLLOWING

- Flags
  - ACK
  - RST
  - SYN
  - FIN
    - TCP Connection Establishment & Termination
  - CWR & ECN
  - PSH
  - URG
- Quick Quiz!



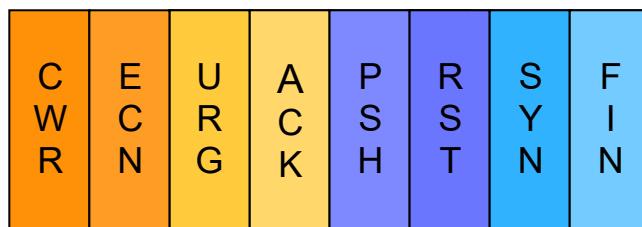
TCP headers have eight 1-bit flags that are imperative to signaling in the protocol.



Each 1-bit flag is crucial to make the protocol work.

# Flags #

Let's have a quick look at what each flag is meant for.



The eight flags used in TCP headers.

The first four discussed below, namely **ACK**, **RST**, **SYN**, and **FIN** are used in the establishment, maintenance, and tear-down of a TCP connection.

## ACK #

This flag is set to 1 in a segment to **acknowledge** a segment that was received previously. This is an important part of the protocol. In other words, 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.



**Note** that these acknowledgment messages can and often do contain data as well!

## 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.

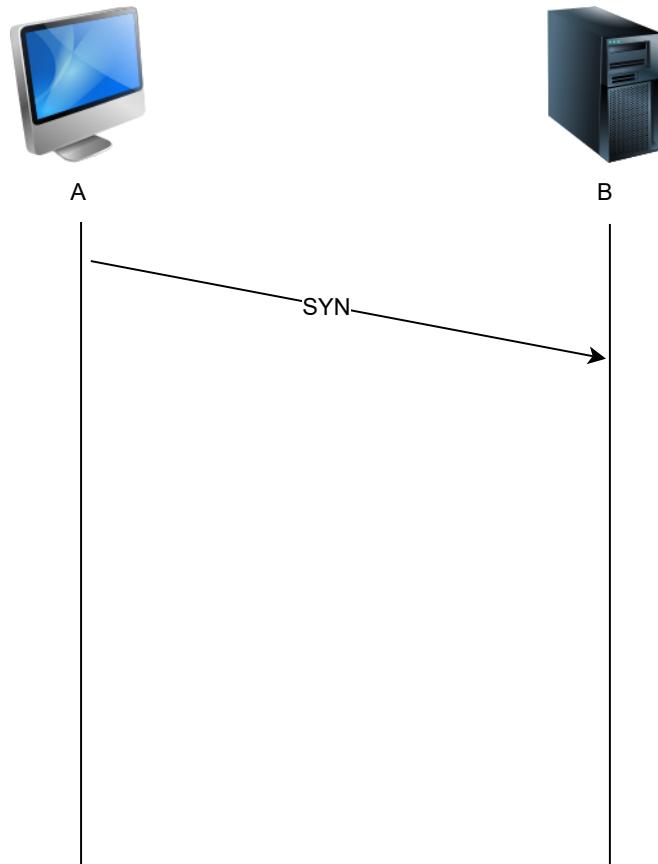
## SYN #

The **synchronization** flag initiates a connection establishment with a new host. The details will be covered later in the lesson on [connection establishment](#).

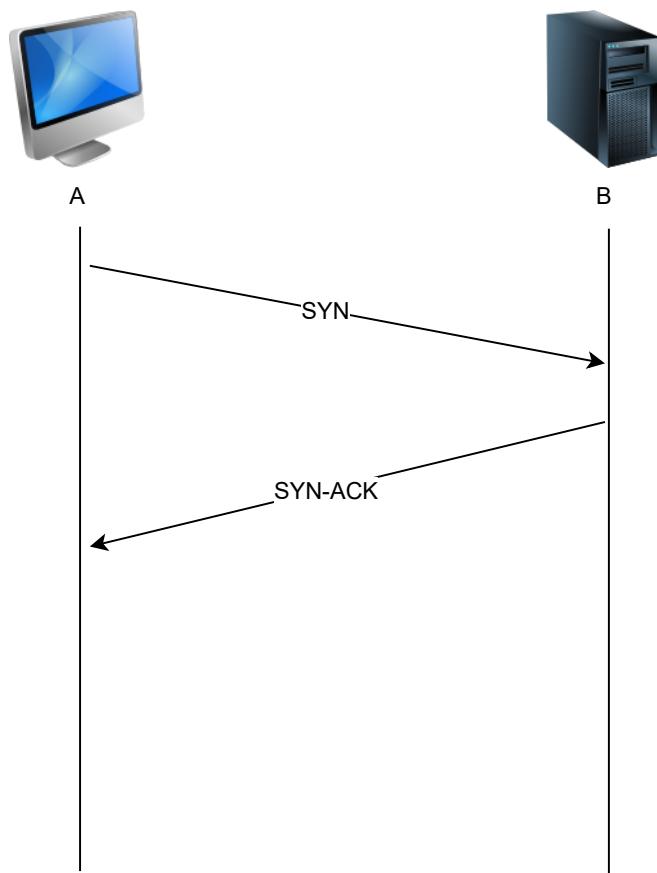
This flag is used to terminate or **finish** a connection with a host.

### TCP Connection Establishment & Termination #

The slides below give a very high level overview of how these flags are used to establish and terminate a TCP connection.

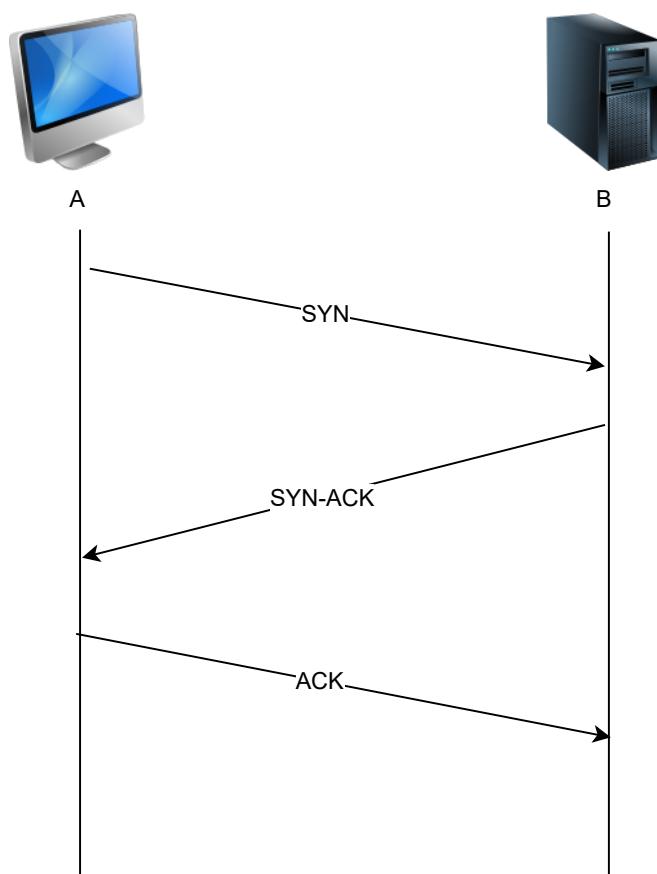


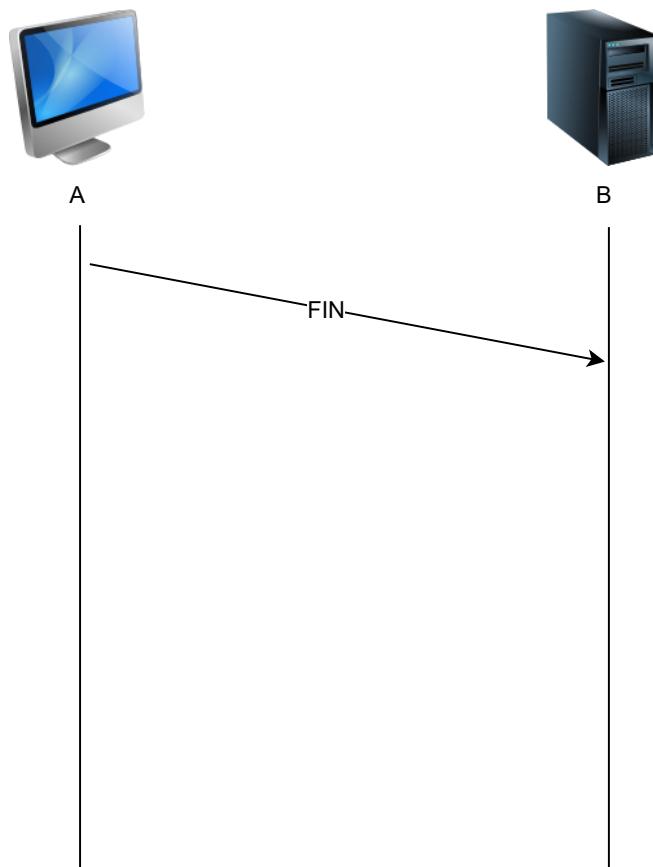
Host A wants to establish a connection with host B and so sends a segment with the SYN flag set



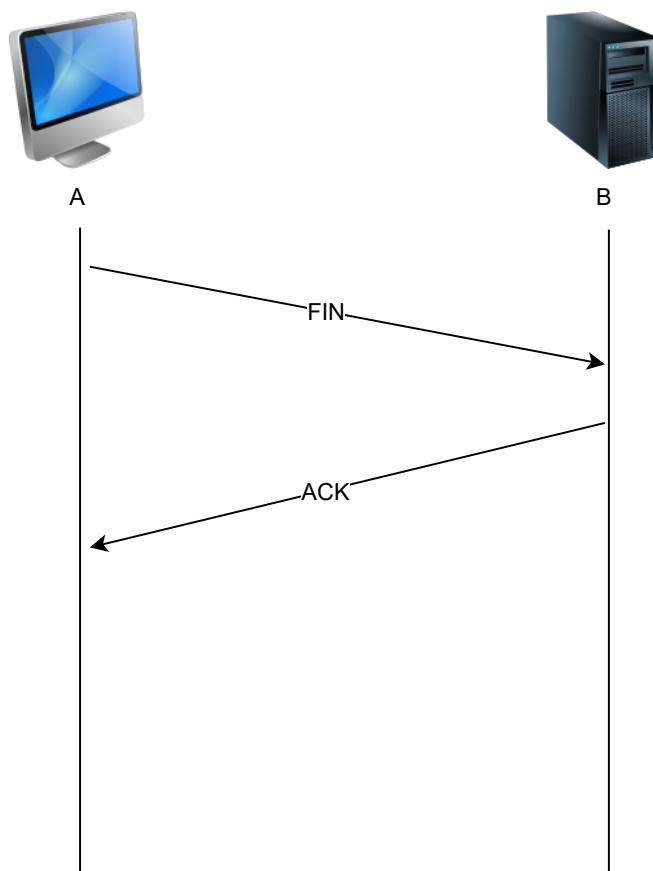
Host B acknowledges the SYN with a segment with an ACK flag along with a SYN flag of its own set in the same message

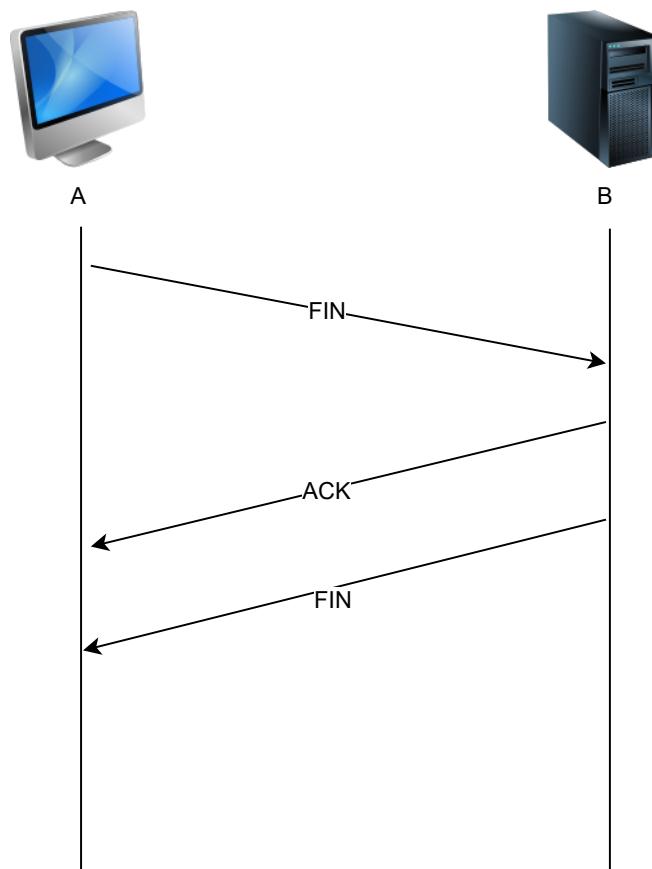
2 of 8



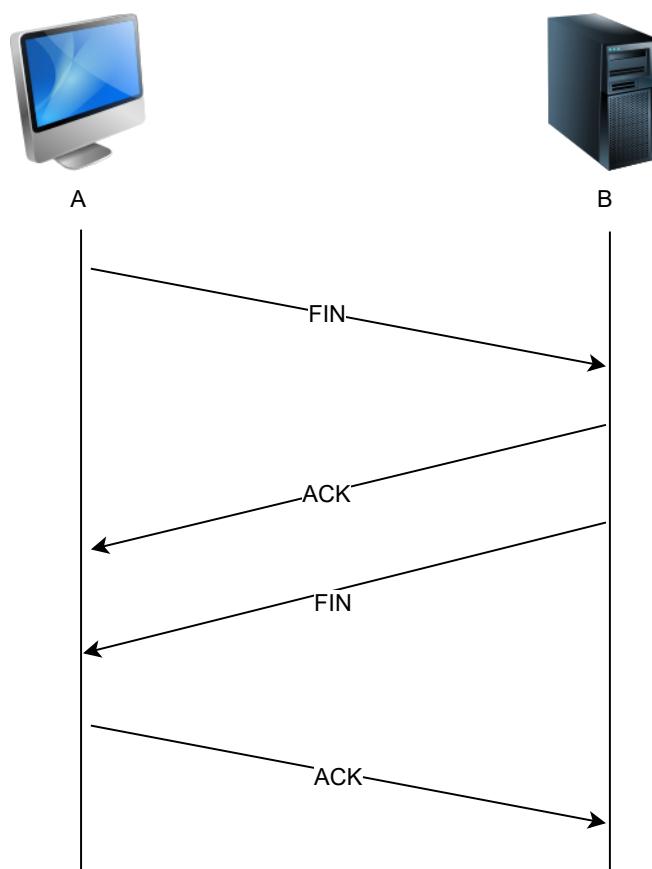


Host A wants to TERMINATE a connection with host B and so sends a segment with the FIN flag set to 1

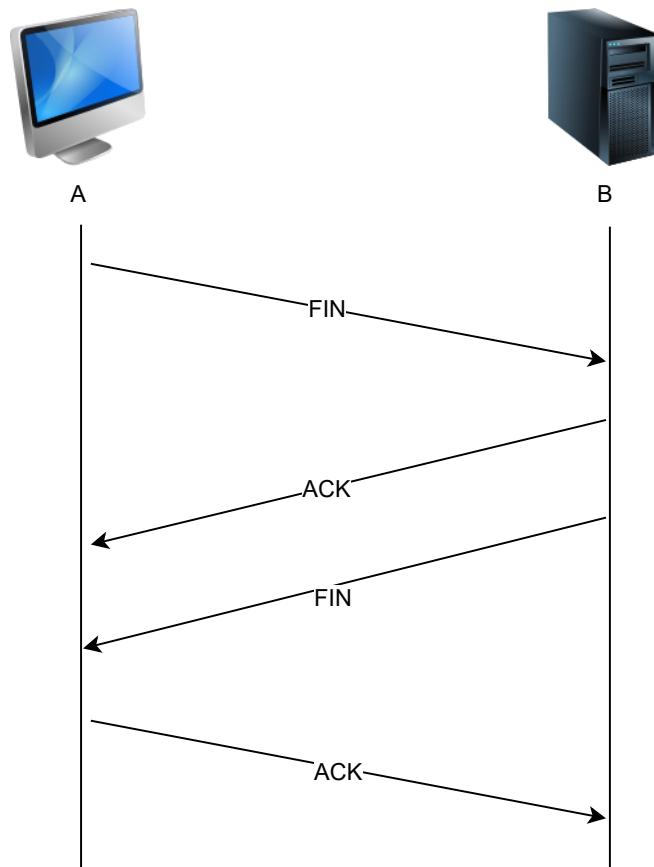




Host B sends a FIN flag



Host A acknowledges host B's FIN segment with an ACK



The connection is terminated

8 of 8



The rest of the flags, given below, are not very well-known. However, it doesn't hurt to know about them.

## 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 #

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)

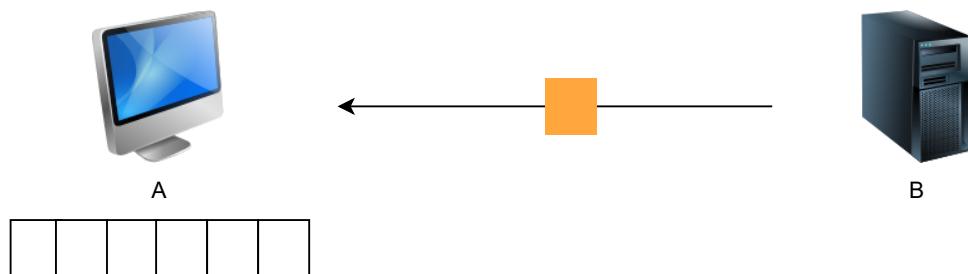
flag is set, the receiving end immediately flushes the data from its buffer to the application instead of waiting for the rest of it to arrive.

This is usually used for applications like Telnet, where every keystroke is a command. It would not make sense to say, buffer 50 keystrokes and send them to the application layer at once, so, every keystroke is pushed.



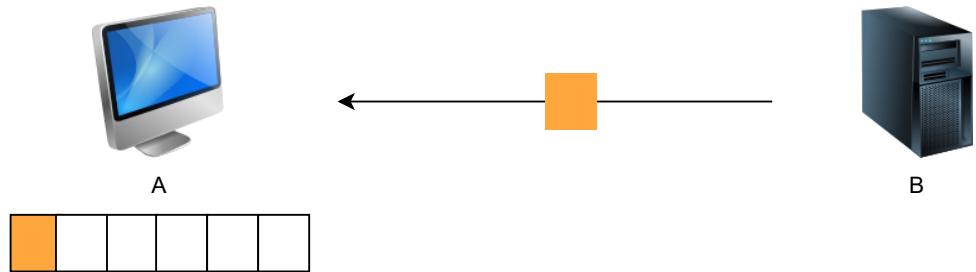
Host A has a buffer for the data it receives from host B

1 of 9



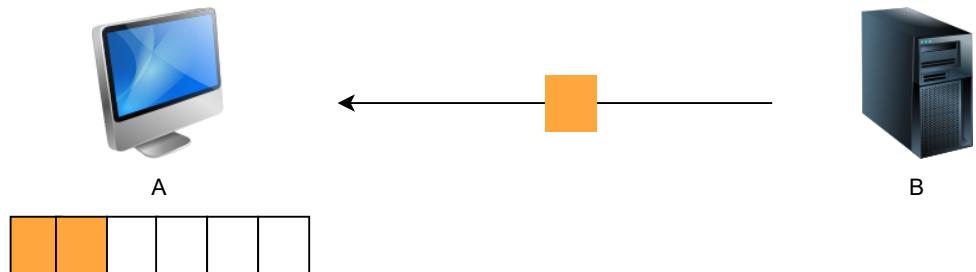
Host A will 'flush' that data out to the application when it is full

2 of 9



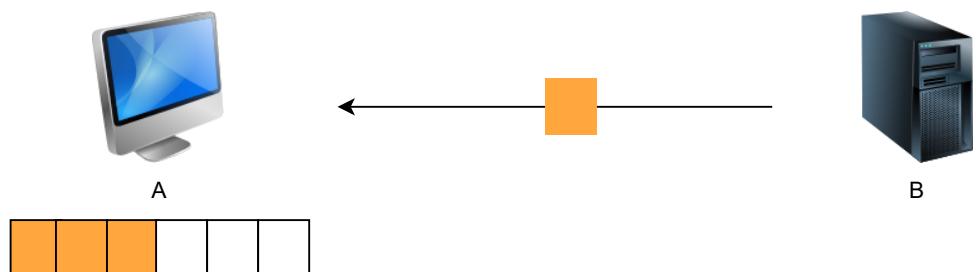
Host A receives a TCP segment from host B

3 of 9



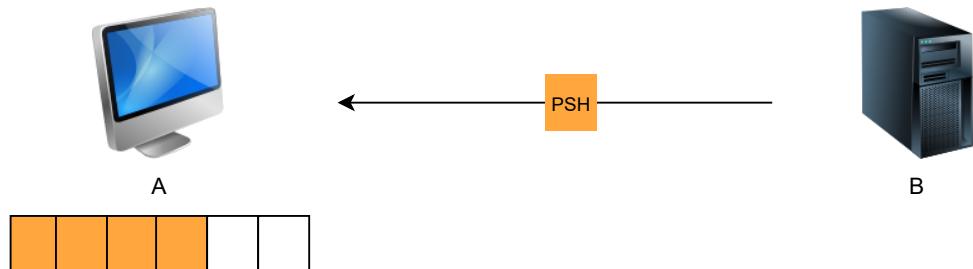
Host A receives a TCP segment from host B

4 of 9



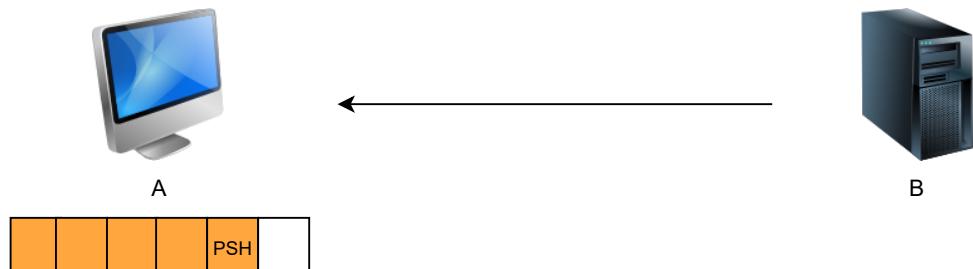
Host A receives a TCP segment from host B

5 of 9



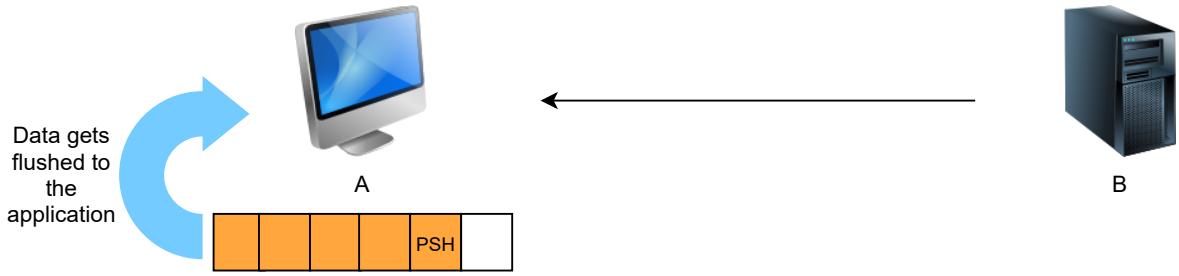
Host B sends a segment with the PSH flag on

6 of 9



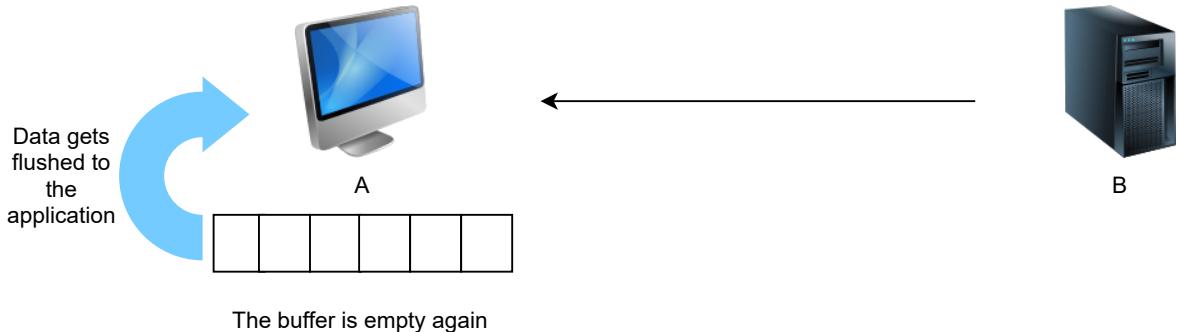
Host A receives said segment

7 of 9



and flushes the buffer out to the application even though the buffer is not full yet

8 of 9



The buffer is empty again

The buffer gets flushed, is empty and is ready to receive data again

9 of 9



## 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. The rest of the data in the segment is processed normally.

This would be used when suppose a large file is being transferred but the

This would be used when suppose a large file is being transferred but the sender realizes that it's the wrong file and sends a command to stop transfer.

It wouldn't make sense to have the file finish transferring first, hence the command to stop transfer is marked as **urgent** and is executed before the file is done transferring.

## Quick Quiz! #

1

What functionality does the urgent flag allow that the push flag does not?

COMPLETED 0%

1 of 3



Let's finish off looking at the rest of the headers in the next lesson!

# TCP Headers: Window Size, Checksum & More

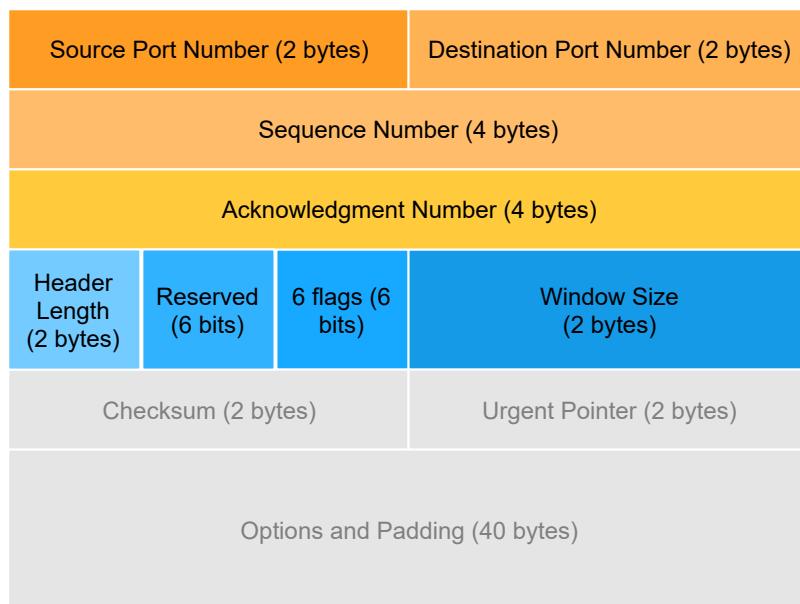
We'll continue our discussion of TCP headers in this lesson

## WE'LL COVER THE FOLLOWING ^

- Window Size
- Checksum
- Urgent Pointer
- Options & Padding
  - Common Options
- Quick Quiz!

## Window Size #

Remember the ‘buffer’ we discussed in the [last lesson](#)? 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**.

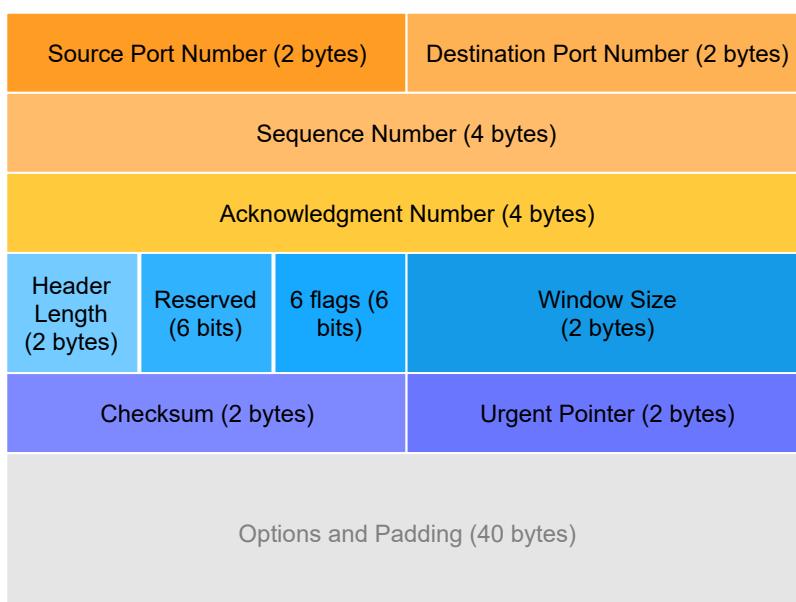


The window size is a critical element of TCP

The window size can be up to **2 bytes** in size hence, the numerical range of the window size is from  $2^0 \rightarrow 2^{16} - 1$  or  $0 \rightarrow 65535$ .

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.



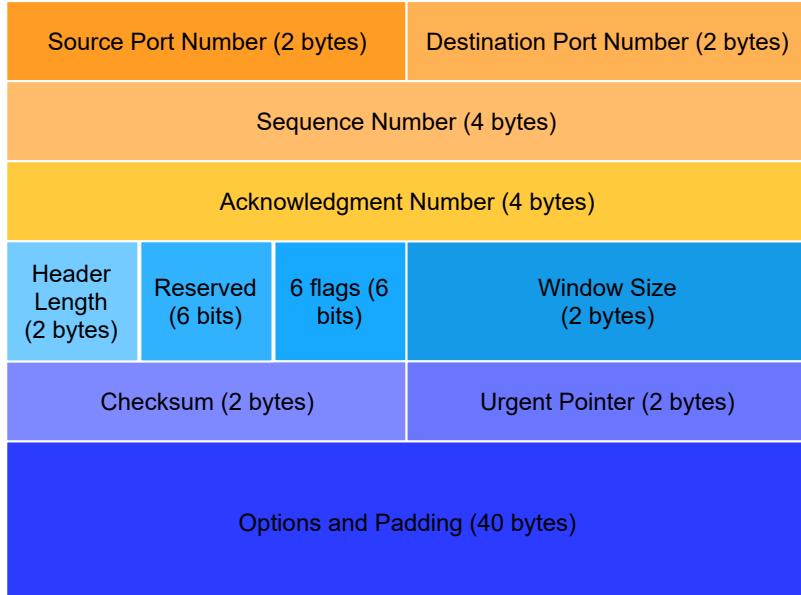
The checksum allows TCP to be reliable, and the urgent pointer identifies the range of bytes in the packet that are urgent.

## Checksum #

The **checksum** is calculated exactly like in UDP except that the checksum calculation is mandatory in TCP!

## Urgent Pointer #

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 field allows to build in extra functionality that the regular header does not cover

## Options & Padding #

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.

### Common Options #

Some options are commonly used and are well-defined. Here's a table that discusses each.

Option	Explanation
MSS	Defines the maximum-sized payload a host can handle at one time. Larger payload sizes are preferable because the header would be slightly different in certain fields for each segment. The redundancy will be significant and overhead will therefore be high. If this option is not used, the 536-byte payload is used as default.

## Timestamp

Allows senders to timestamp segments. This is a logical extension of sequence numbers. We'll have a closer look at this one in upcoming lessons!

## Window Scale

Allows the host to 'scale up' its window size by a factor in situations where sending data to the sender from the receiver takes longer than sending to the receiver. This, for example, allows the sender to keep sending data without having to wait long periods of time to hear back for acknowledgments. This option essentially allows the communicating parties to use larger than default buffers - larger by a multiple factor (power of 2 using shift operations). So, if the scale factor is 4 and the window size otherwise specified is 20 kB, the actual window size is 80 kB.

## Quick Quiz! #

1

The window size represents the amount of buffer available at the receiver. This should be constant and only communicated in the first TCP segment sent from the receiver to the sender. What do you think?

We're finally done with the TCP header! Let's learn more about the protocol itself starting from the next lesson.

# TCP Connection Establishment: Three-way Handshake

In this lesson, we'll discuss how a TCP connection is established!

## WE'LL COVER THE FOLLOWING



- Initiating a Connection
- Responding to an Initial Connection Message
- Acknowledging The Response
- Quick Quiz!

A TCP connection is established by using a **three-way handshake**, which we briefly touched upon in a previous lesson. The connection establishment phase uses the **sequence number**, the **acknowledgment number**, and the **SYN flag**.

## Initiating a Connection #

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

- The **SYN** flag set
- The sequence number set to a random initial value. So the sequence numbers **do not start with 0!** Can you guess why?

 Why sequence numbers are set to random values

## Responding to an Initial Connection Message #

Upon reception of this segment (which is often called a **SYN** segment), the server host replies with a segment containing:

- the **SYN** flag set

- 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  $1 \bmod 2^{32}$ , 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 allows the number to cycle back and start from 0.



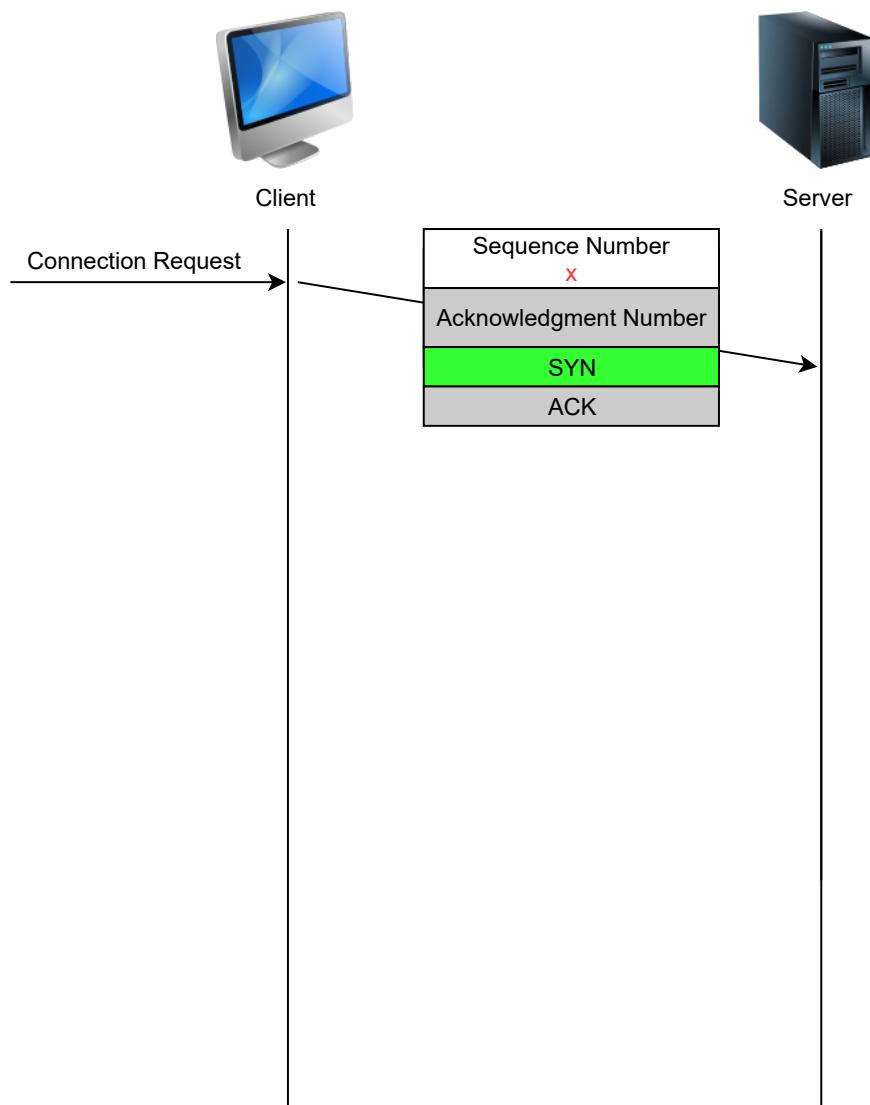
**Note** When a TCP entity sends a segment with  $x + 1$  as the acknowledgment number, it means that it has received all the segments up to and including the segments with the sequence number  $x$ , and that it's expecting data having sequence number  $x + 1$ .

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.

## Acknowledging The Response #

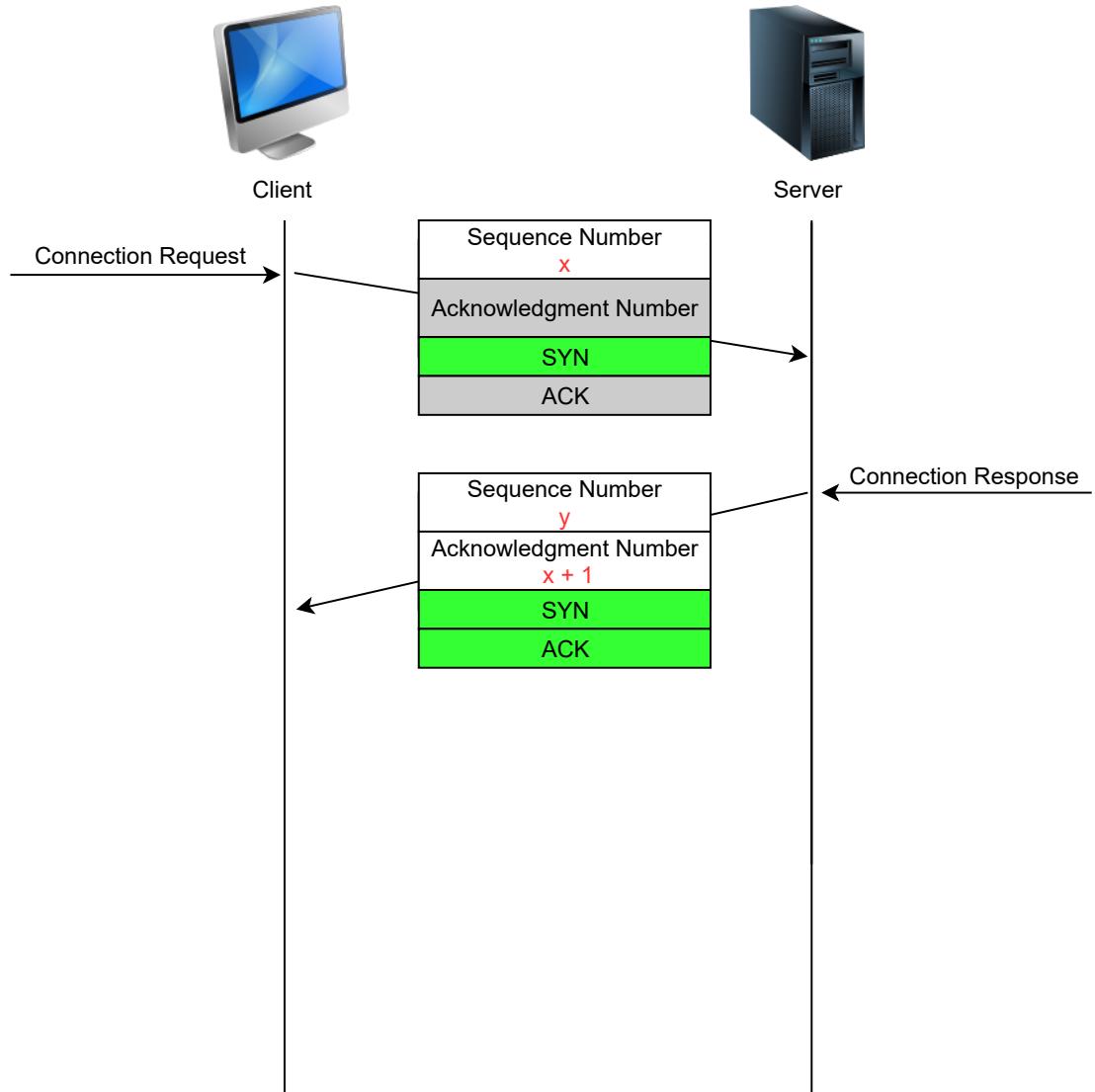
Upon reception of the *SYN+ACK* segment, the client host replies with a segment containing:

- 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. This is illustrated in the figure below:



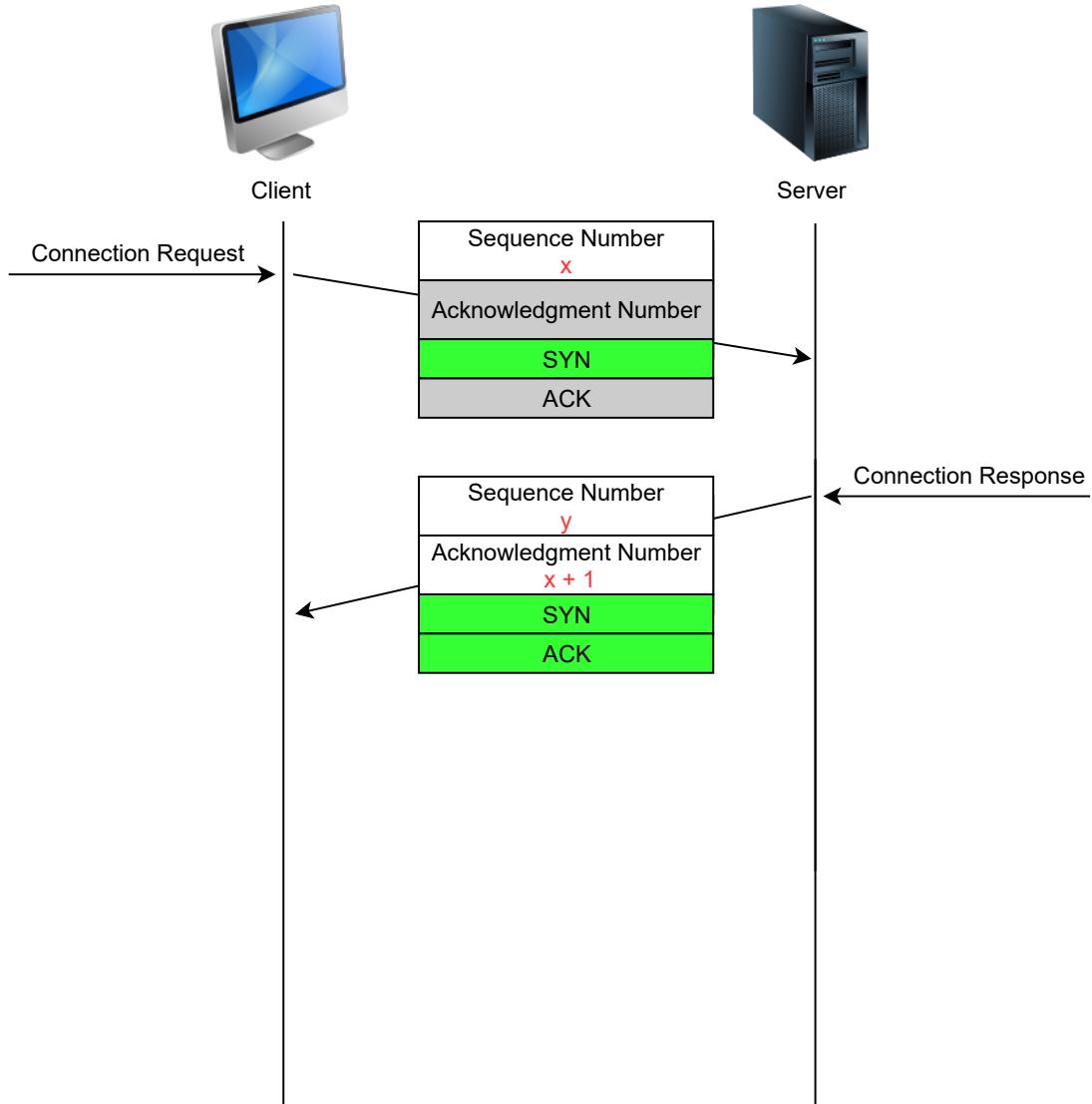
Initiating a Connection

1 of 4



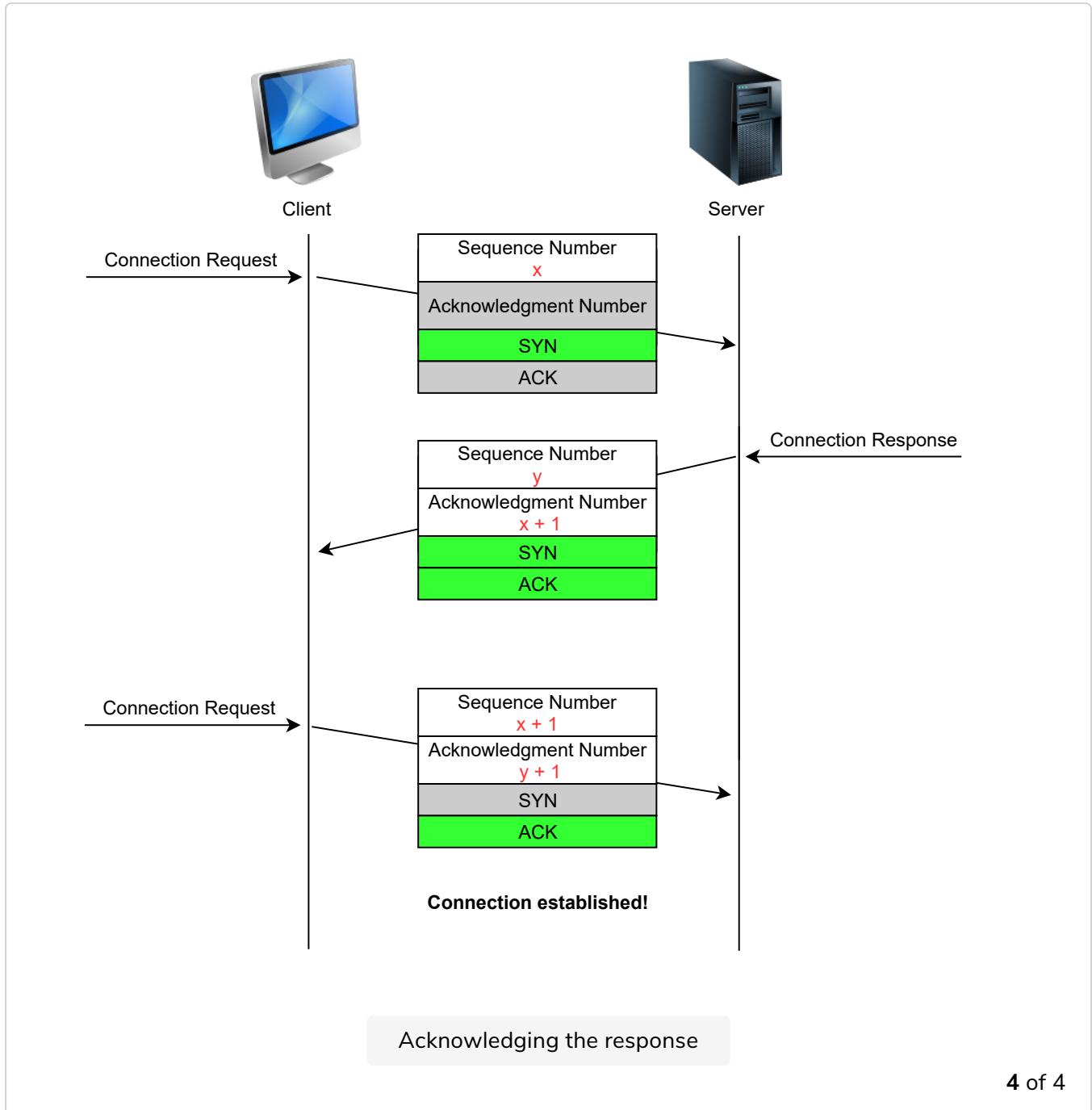
Responding to the initial connection message

2 of 4



Acknowledging the response

3 of 4



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.

## Quick Quiz! #

1

- A client is establishing a connection with a server. The first segment sent from the client to the server contains \_\_\_\_\_ in the flags field.

COMPLETED 0%

1 of 3



---

A server could, of course, refuse to open a TCP connection upon reception of a SYN segment. This refusal may be due to various reasons. Let's discuss the details in the next lesson.

# Other TCP Connection Establishment Methods

In this lesson, we'll look at some unconventional ways that connection establishment may occur.

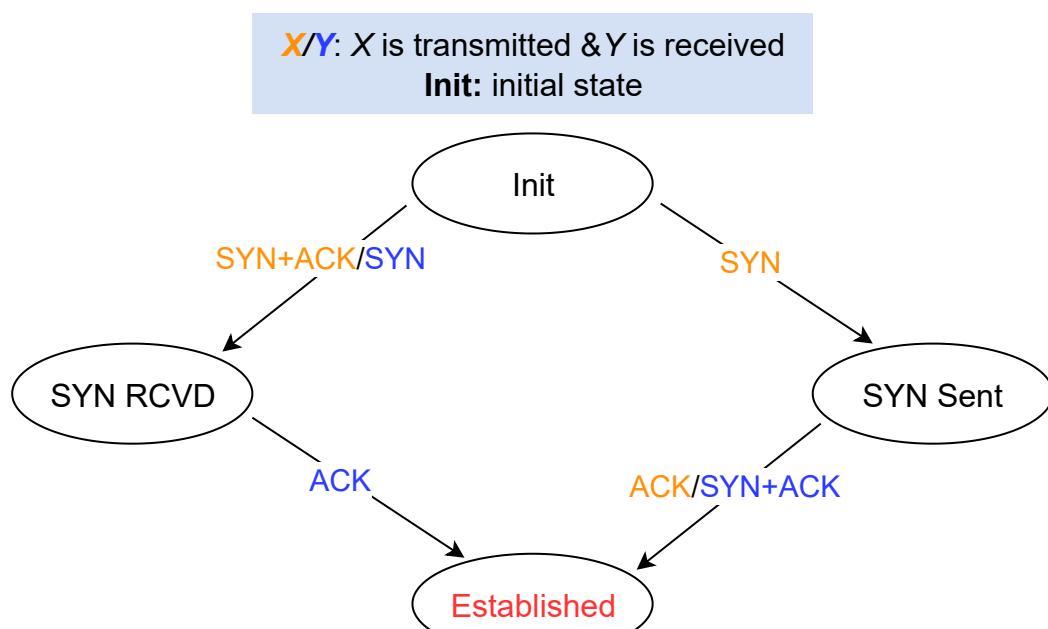
## WE'LL COVER THE FOLLOWING

- A TCP Three-way Handshake FSM
  - Client-Side
  - Server-Side
- Simultaneous Connection Establishment
- Quick Quiz!

In the last lesson, we looked at the most common way that TCP connection establishment could occur. Now, let's look at some other ways it can successfully occur.

## A TCP Three-way Handshake FSM #

TCP connection establishment can be described with a four-state Finite State Machine (FSM) as shown below. In this FSM,  $X/Y$  indicates that segment  $X$  was transmitted and segment  $Y$  was received. *Init* is the initial state



## Client-Side #

Let's carve out the paths in this FSM. Here's the three-way handshake path.

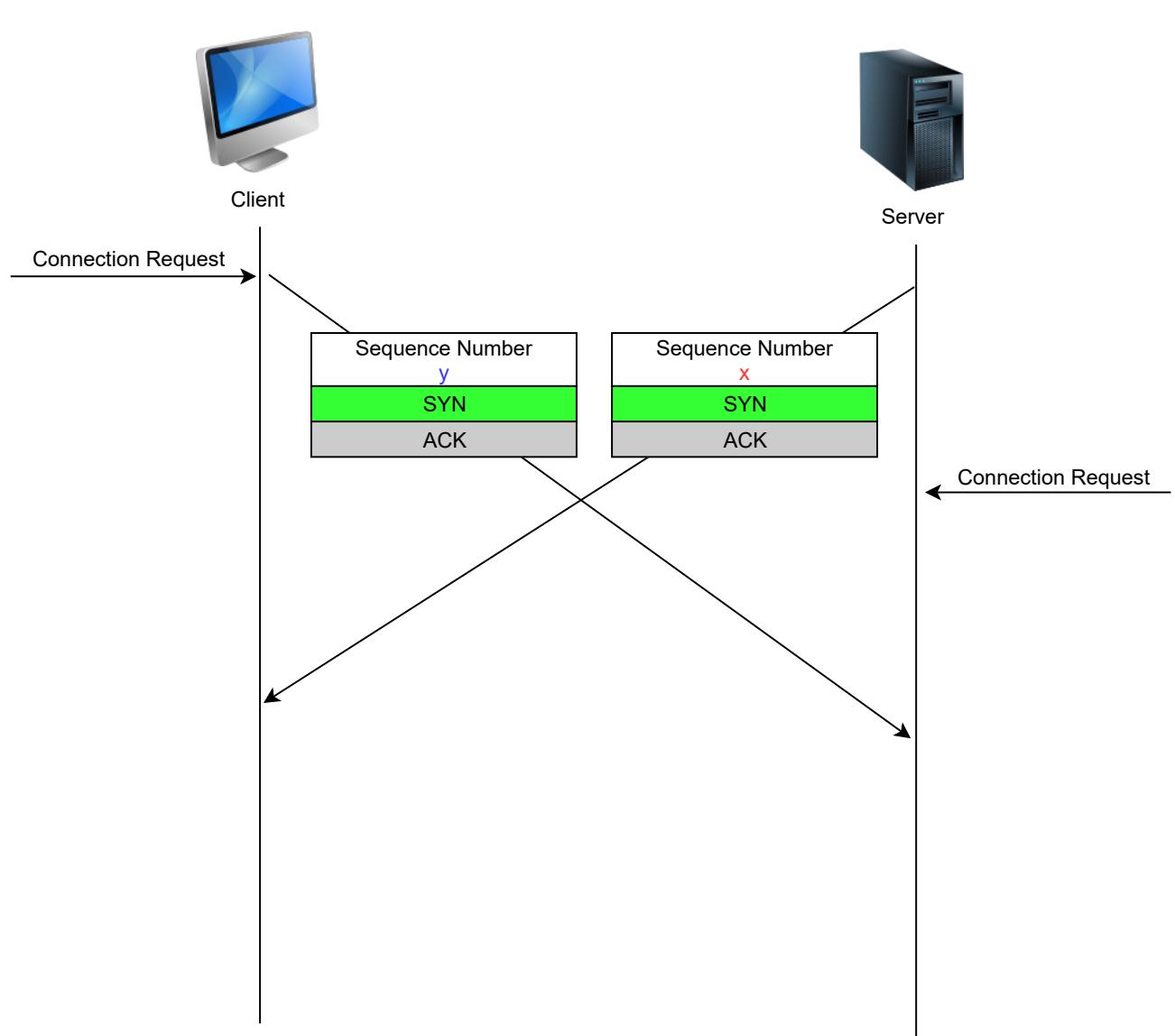
1. A client host starts in the **Init** state.
2. It then sends a *SYN* segment and enters the **SYN Sent** state where it waits for a *SYN+ACK* segment.
3. When a *SYN+ACK* is received in the **SYN SENT** state, it replies with an *ACK* segment and enters the **Established** state where data can be exchanged.

## Server-Side #

1. On the other hand, a server host starts in the **Init state**.
2. When a server process starts to listen to a destination port, the underlying TCP entity creates a TCP control block and a queue to process incoming *SYN* segments. Upon reception of a *SYN* segment, the server's TCP entity replies with a *SYN+ACK* and enters the **SYN RCVD** state.
3. It remains in this state until it receives an *ACK* segment that acknowledges its *SYN+ACK* segment, and with this it then enters the **Established** state.

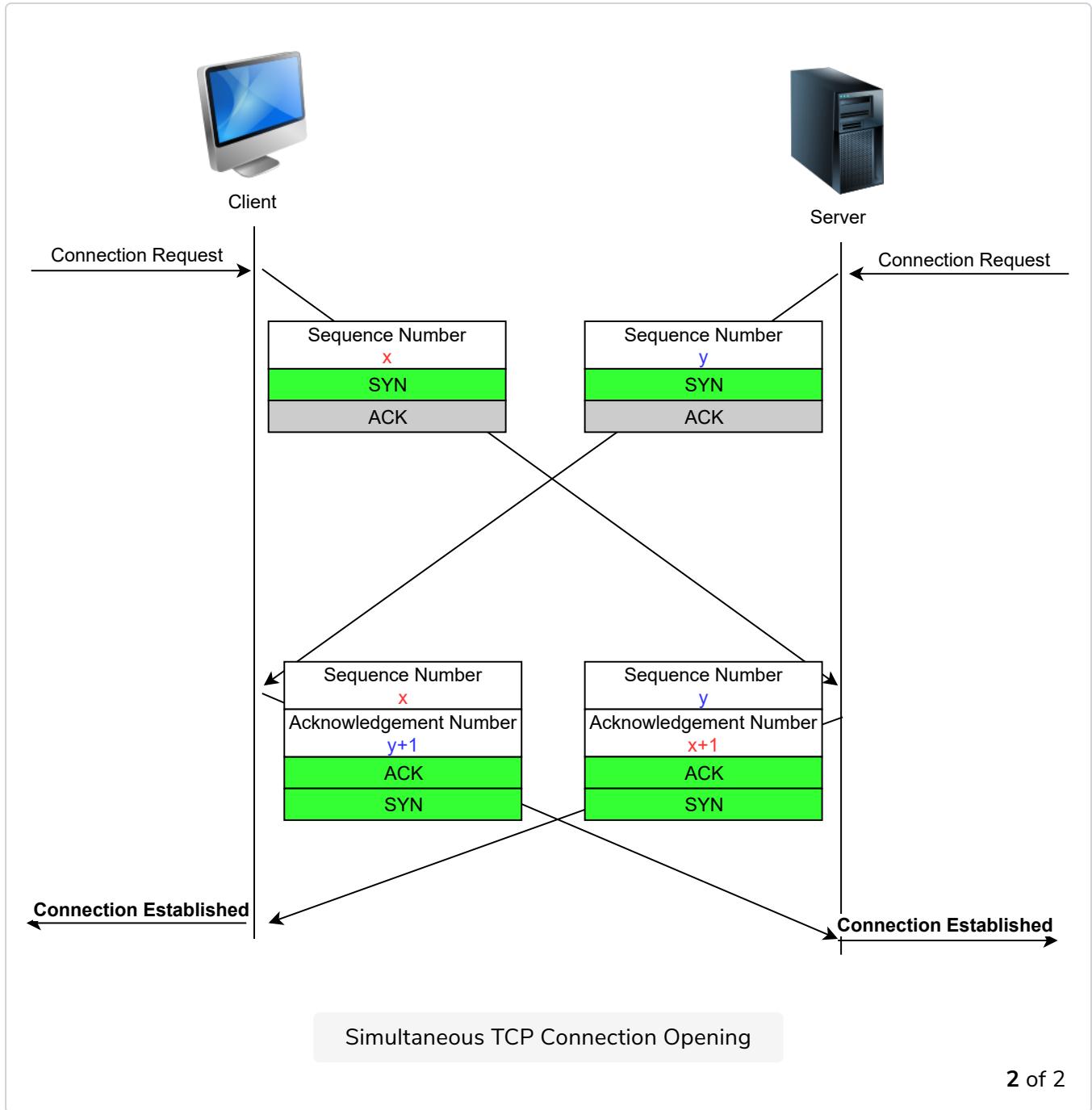
## Simultaneous Connection Establishment #

Apart from these two paths in the TCP connection establishment, shown in the above FSM, there is a third way that a connection can be established: when both the client and the server send a *SYN* segment to open a TCP connection.



Simultaneous TCP Connection Opening

1 of 2



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.

## Quick Quiz! #

1

- Simultaneous connection may result in more segments being exchanged than a regular three-way handshake.

COMPLETED 0%

1 of 2



---

Coming up, we'll look at a few scenarios in which connection establishment can go wrong and how TCP handles it.

# When Connection Establishment Fails: Syn Floods & Retransmission

In this lesson, we'll look at a couple of loopholes in TCP's implementation and how modern fixes took care of them.

## WE'LL COVER THE FOLLOWING



- Hosts Can Refuse Connection Requests
- Syn Flood Attacks
  - Syn Cookies
- Retransmitting Lost Segments
- Quick Quiz!

## Hosts Can Refuse Connection Requests #

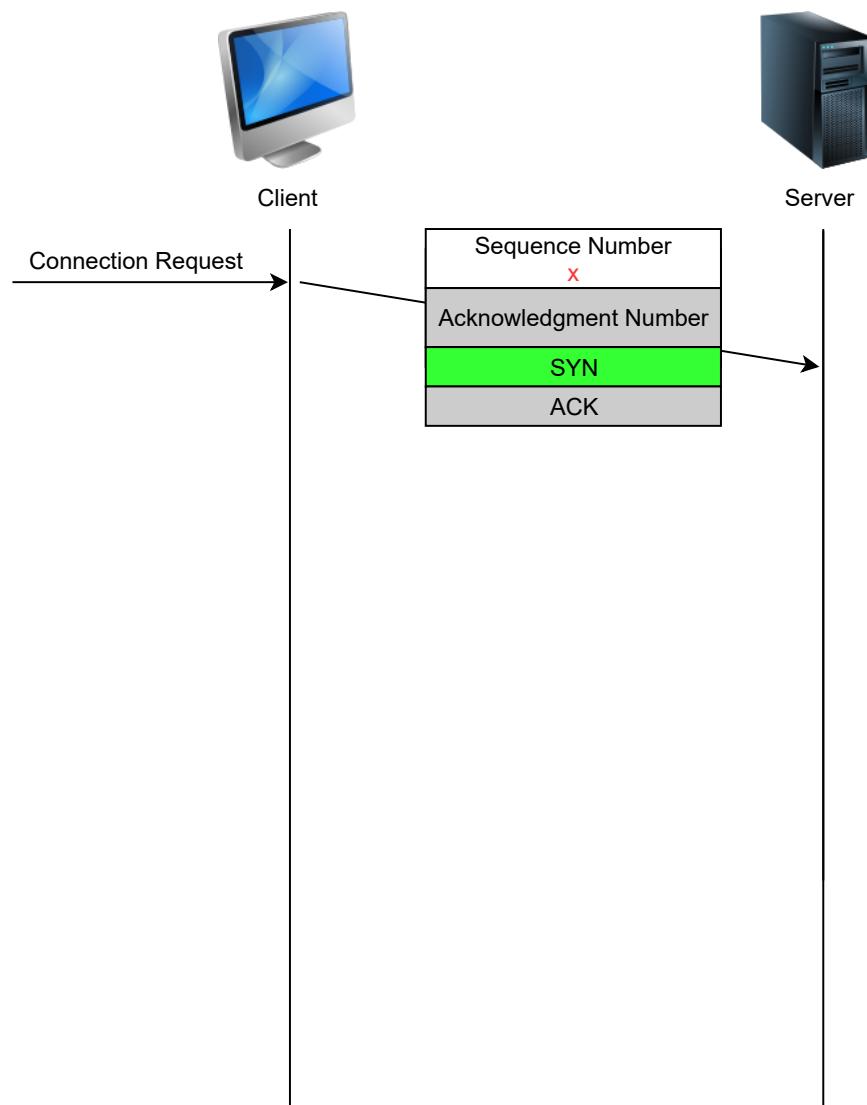
A host could refuse to open a TCP connection upon reception of a *SYN* segment. This refusal may be due to various reasons, for example:

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:

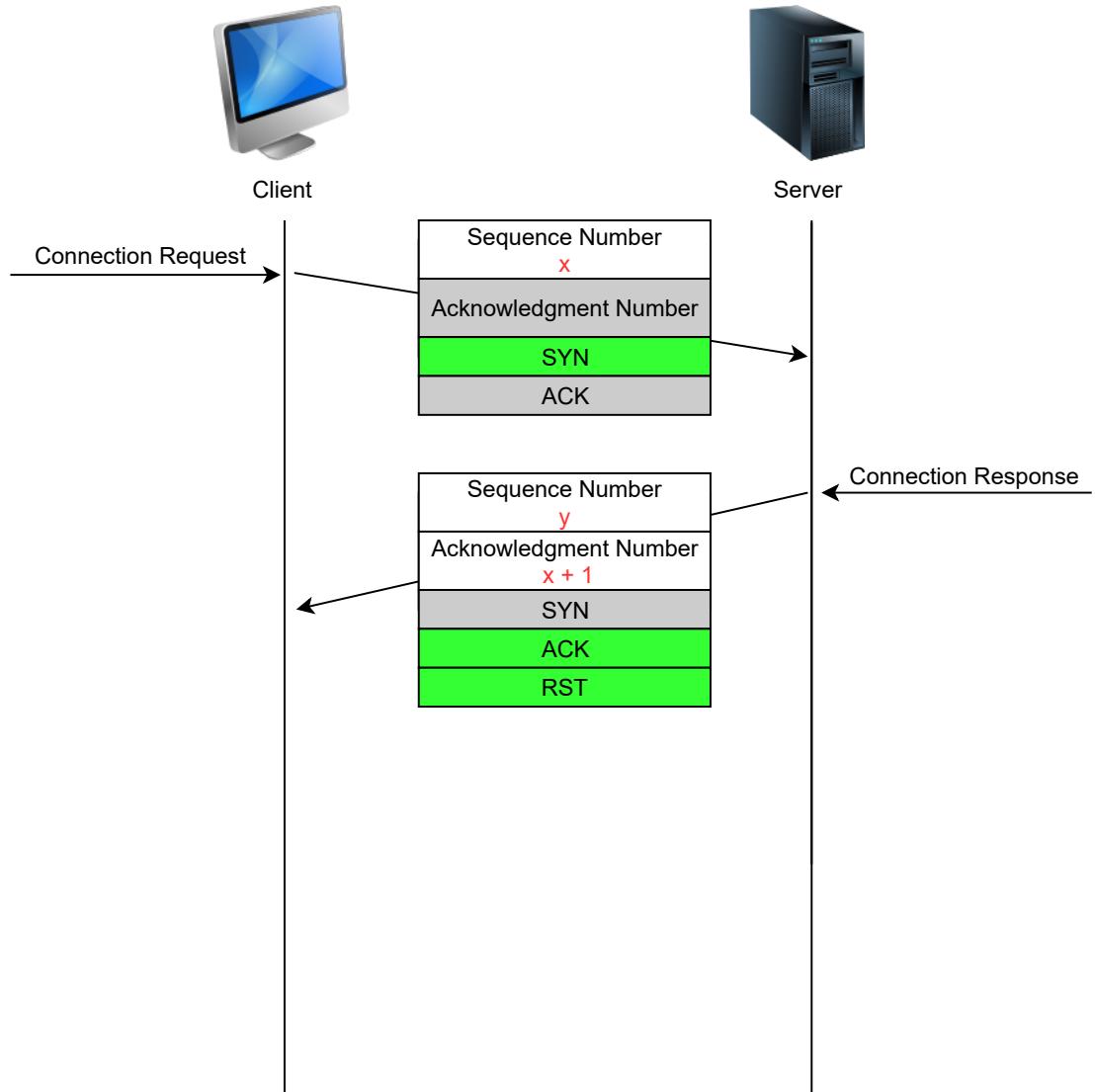
- Has its *RST* flag set
- Contains the sequence number of the received *SYN* segment as its acknowledgment number.

This is illustrated in the slides below. We will discuss the other utilizations of the TCP *RST* flag later in the TCP connection release lesson.

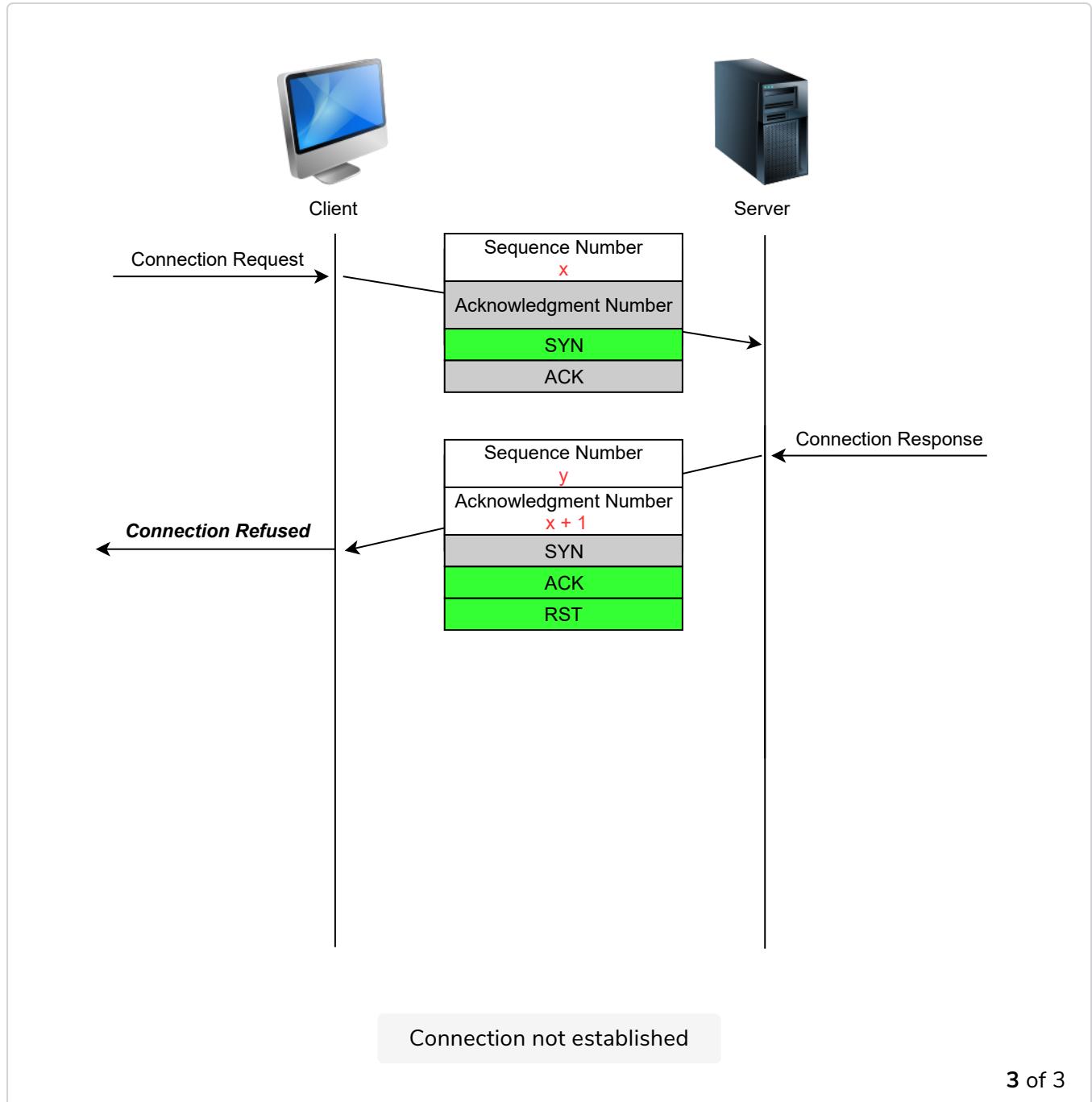


Initiating a Connection

1 of 3



The server refuses the connection with a rst segment



3 of 3



## Syn Flood Attacks #

When a TCP entity opens a TCP connection, it creates a **Transmission Control Block (TCB)** that contains the entire state of the connection, including the local sequence number and sequence number sent by the remote client. Until the mid-1990s, TCP implementations had a limit on the number of 'half-open' TCP connections (TCP connections in the **SYN RCVD** state) which was most commonly at 100. So a machine could only have a 100 'half-open' TCP connections. This was meant to avoid overflowing the entity's memory with TCBs. When the limit was reached, the TCP entity would stop accepting any

*new SYN segments.*



**Note: The Transmission Control Block:** For each established TCP connection, a TCP implementation must maintain a Transmission Control Block (TCB). A TCB contains all the information required to send and receive segments. This includes:

- the local IP address
- the remote IP address
- the local TCP port number
- the remote TCP port number
- the current state of the TCP FSM
- the maximum segment size (MSS)

Here's an illustration of what should happen.



Client

Assuming that  
the TCB limit is 6



Server

Assuming that  
the TCB limit is 6



Client

Assuming that  
the TCB limit is 6

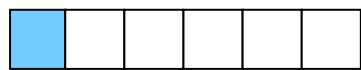


Server

Connection Request + Create TCB

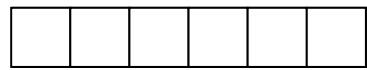


Assuming that  
the TCB limit is 6



Client

Assuming that  
the TCB limit is 6



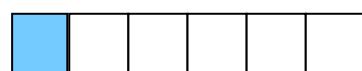
Server

Connection Request + Create TCB



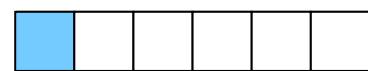
Normal behavior with TCBs

Assuming that  
the TCB limit is 6



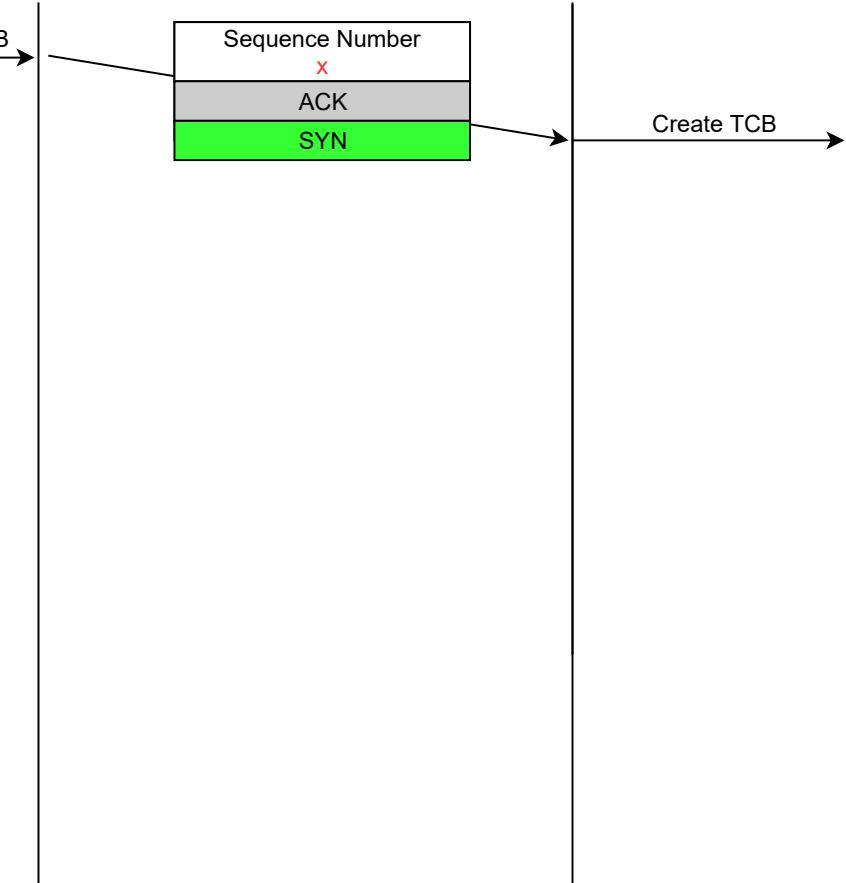
Client

Assuming that  
the TCB limit is 6

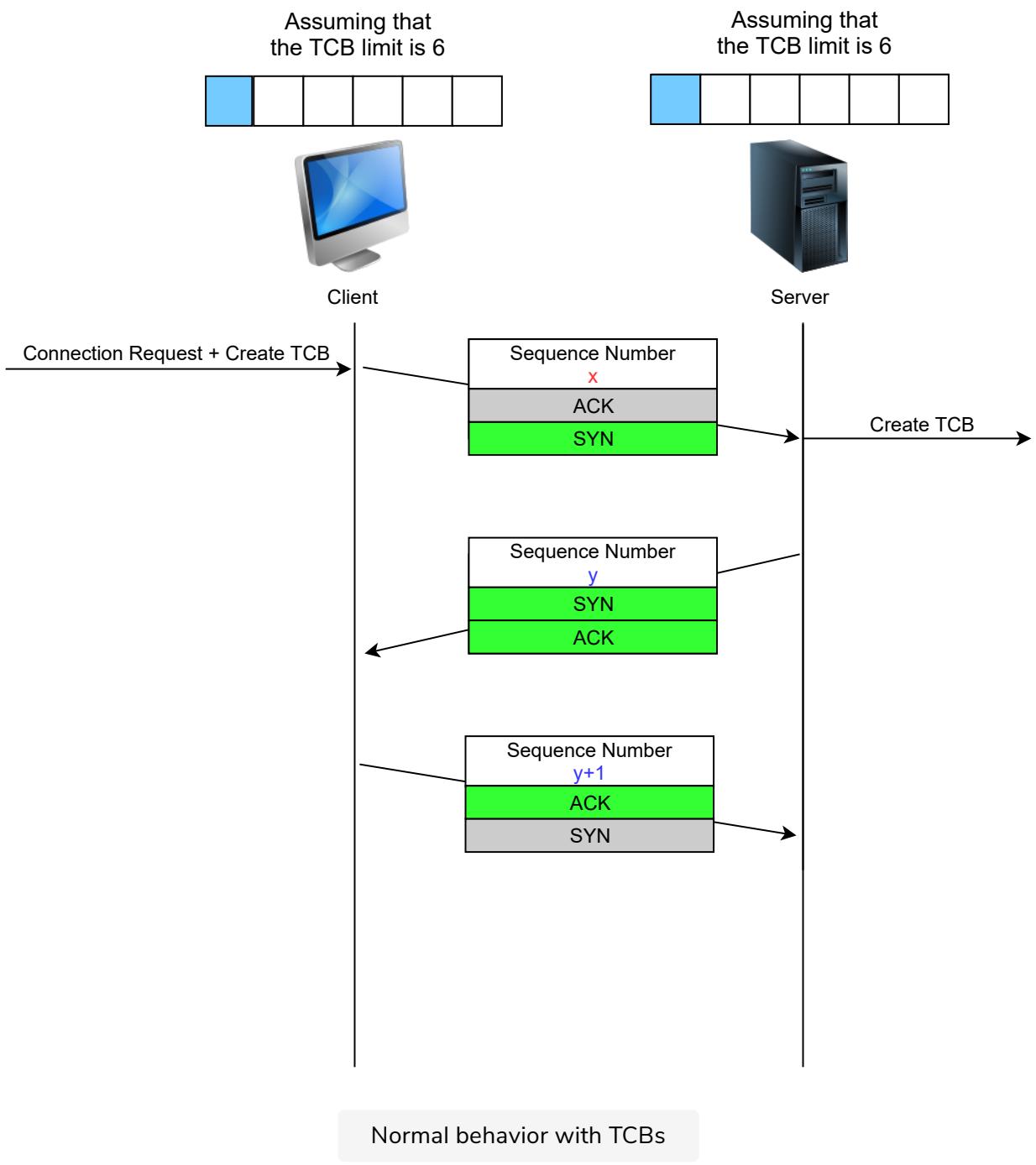


Server

Connection Request + Create TCB



Normal behavior with TCBs



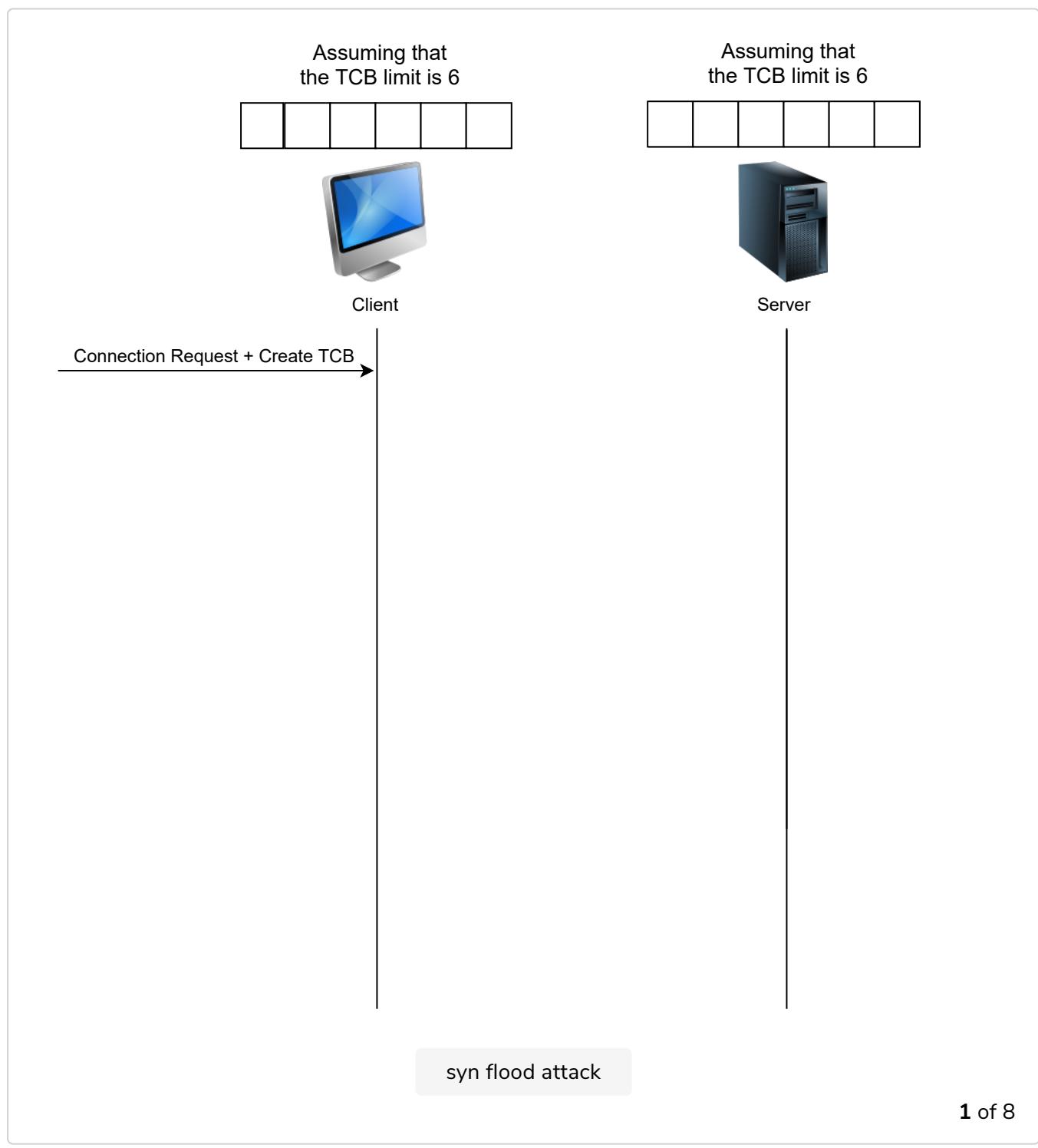
## Syn Cookies #

However, this allowed attackers to carry out an attack where they could render a resource unavailable in the network by sending it valid messages. Such attacks are called **Denial of Service (DoS)** attacks because they deny the user(s) a service. Here's how this one was carried out:

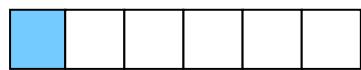
1. The attacker would send a few 100 SYN segments every second to a server

2. The attacker would not reply to any received **SYN+ACK** segments
3. To avoid being caught, the attacker would send these **SYN** segments with a different IP address from their own IP address.
4. Once a server entered the **SYN RCVD** state, it would remain in that state for several seconds, waiting for an ACK and not accepting any new, possibly genuine connections, thus being rendered unavailable.

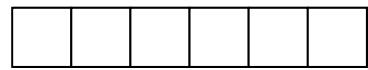
Here are some slides depicting a **SYN flood attack**:



Assuming that  
the TCB limit is 6



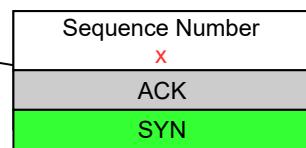
Assuming that  
the TCB limit is 6



Client

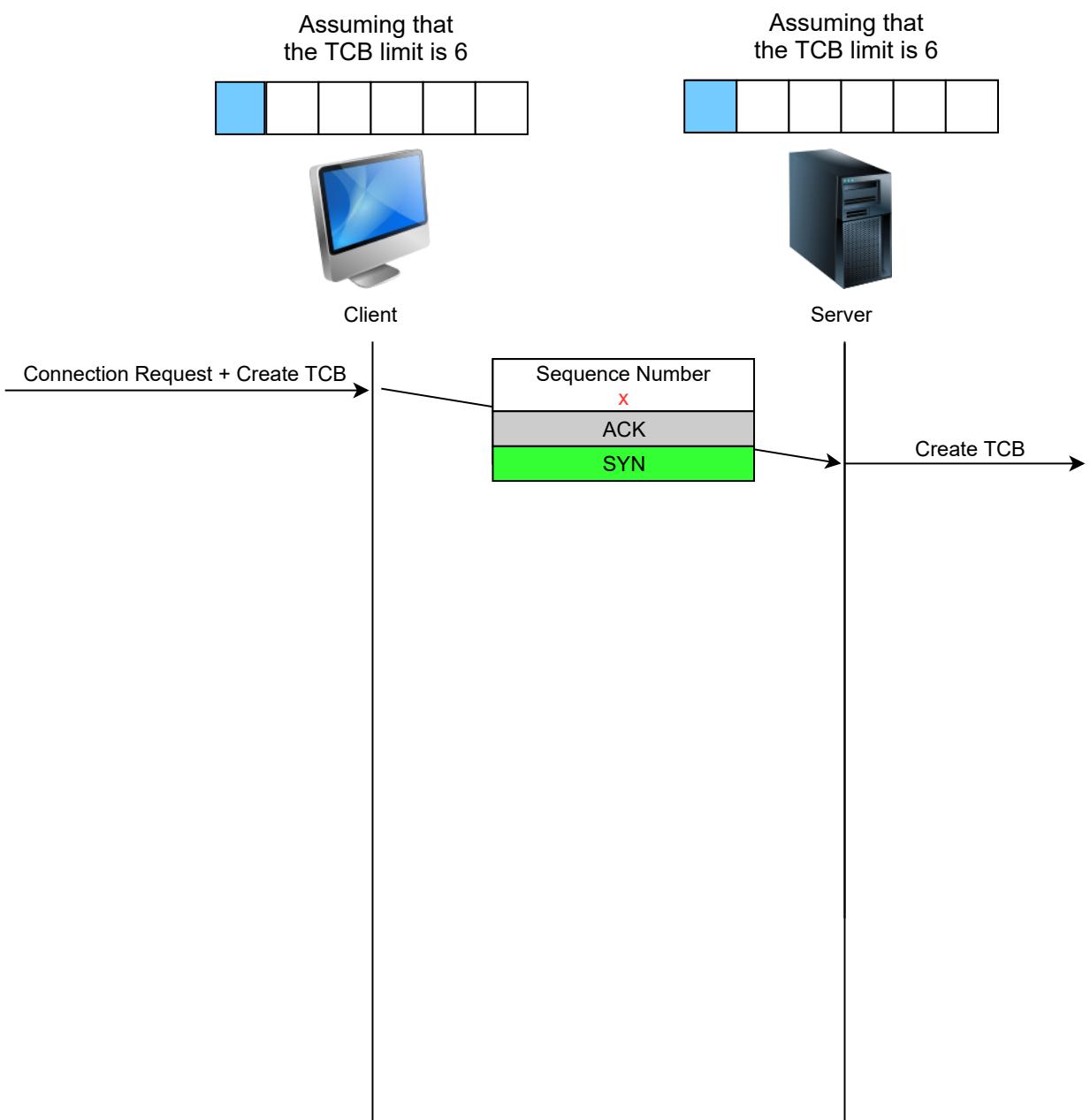
Server

Connection Request + Create TCB



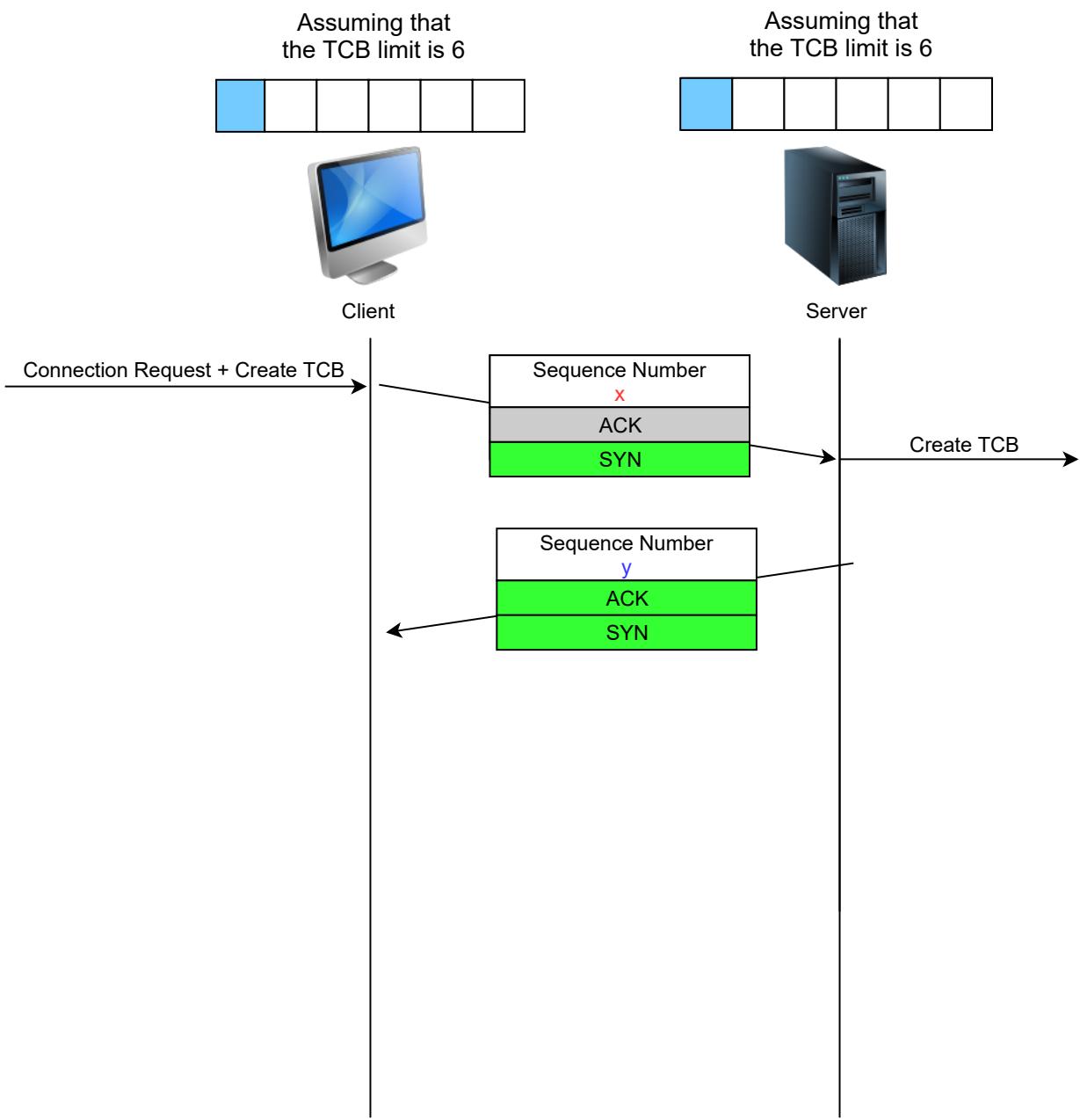
syn flood attack

2 of 8

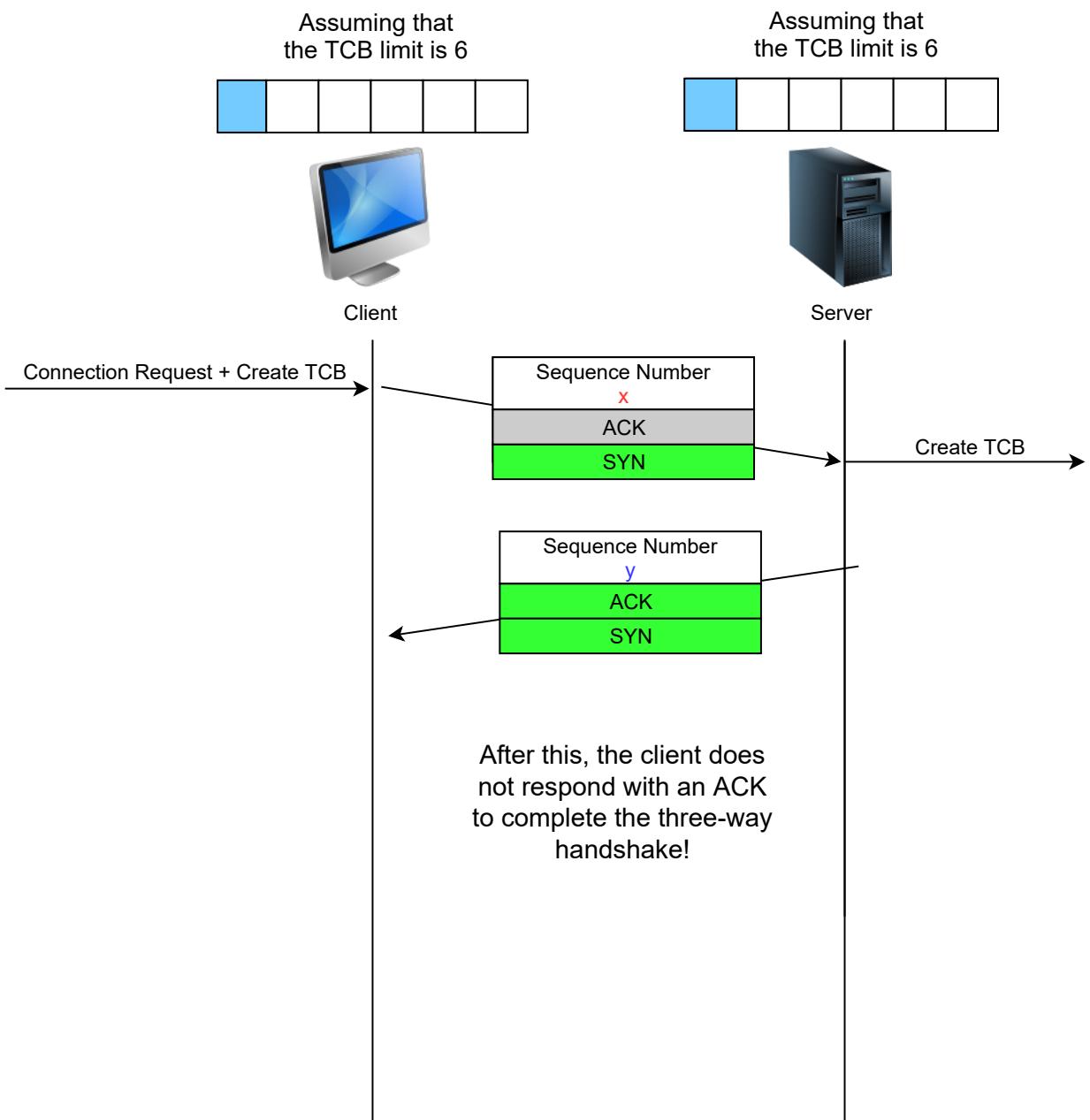


syn flood attack

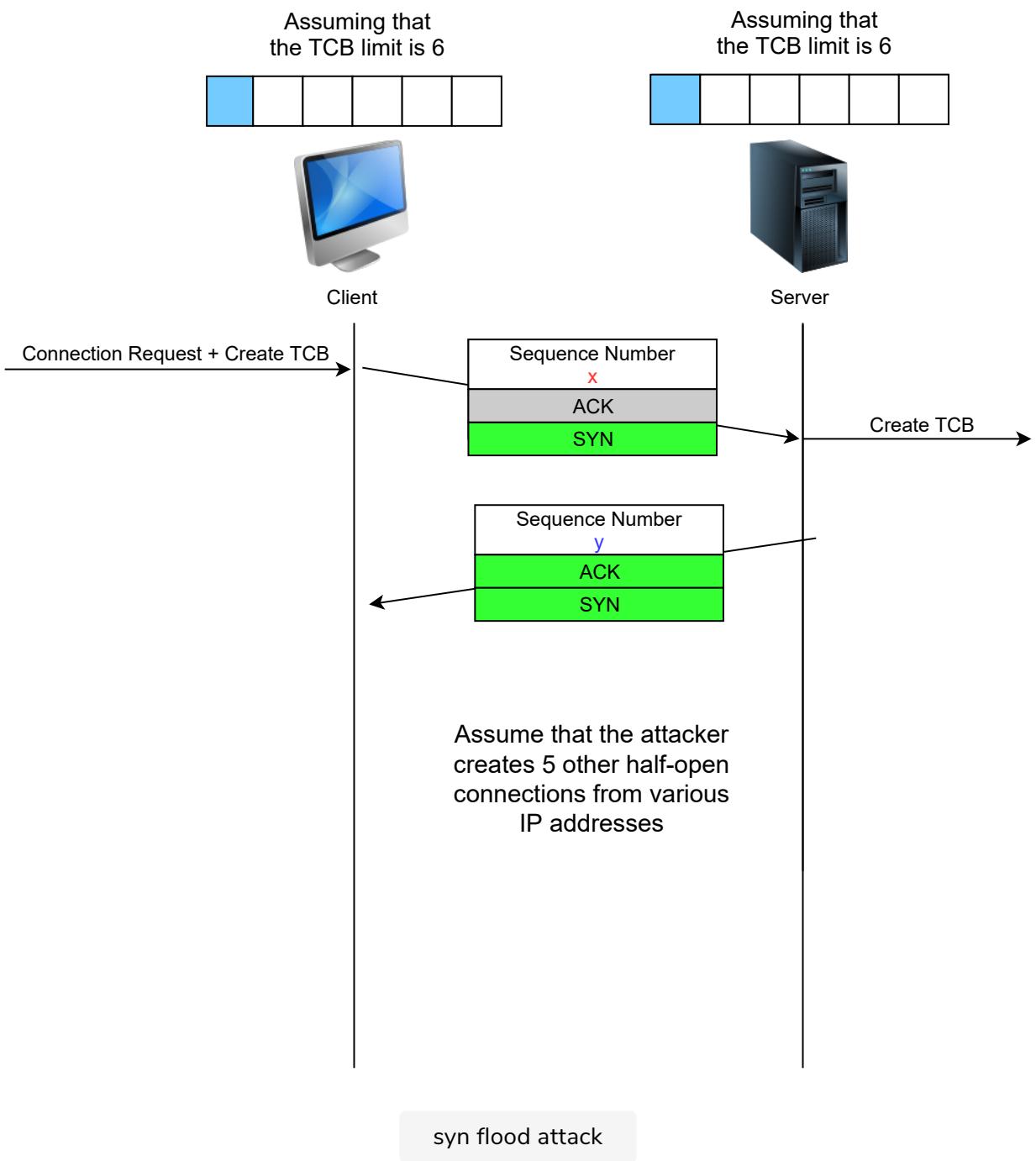
3 of 8

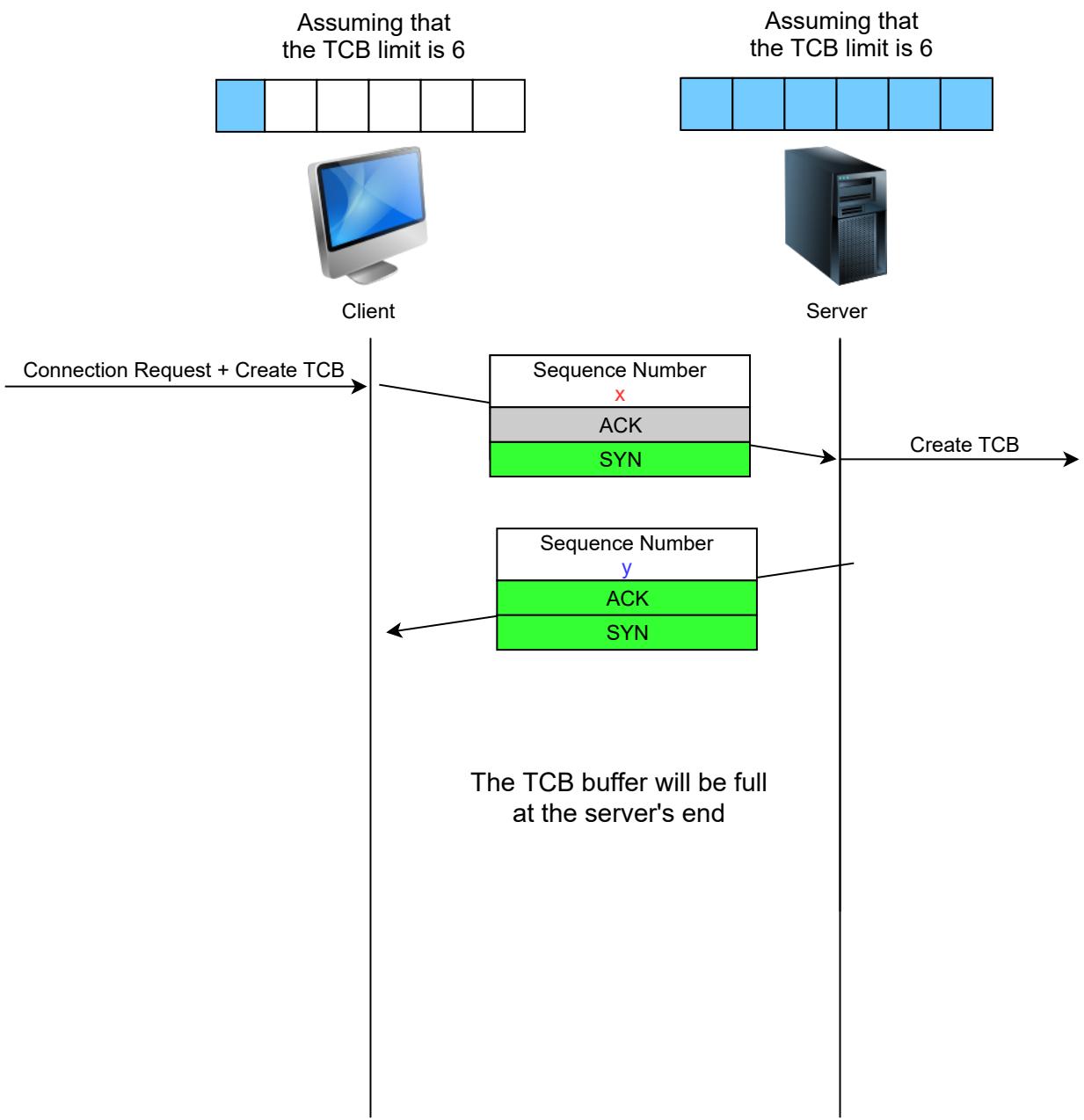


syn flood attack

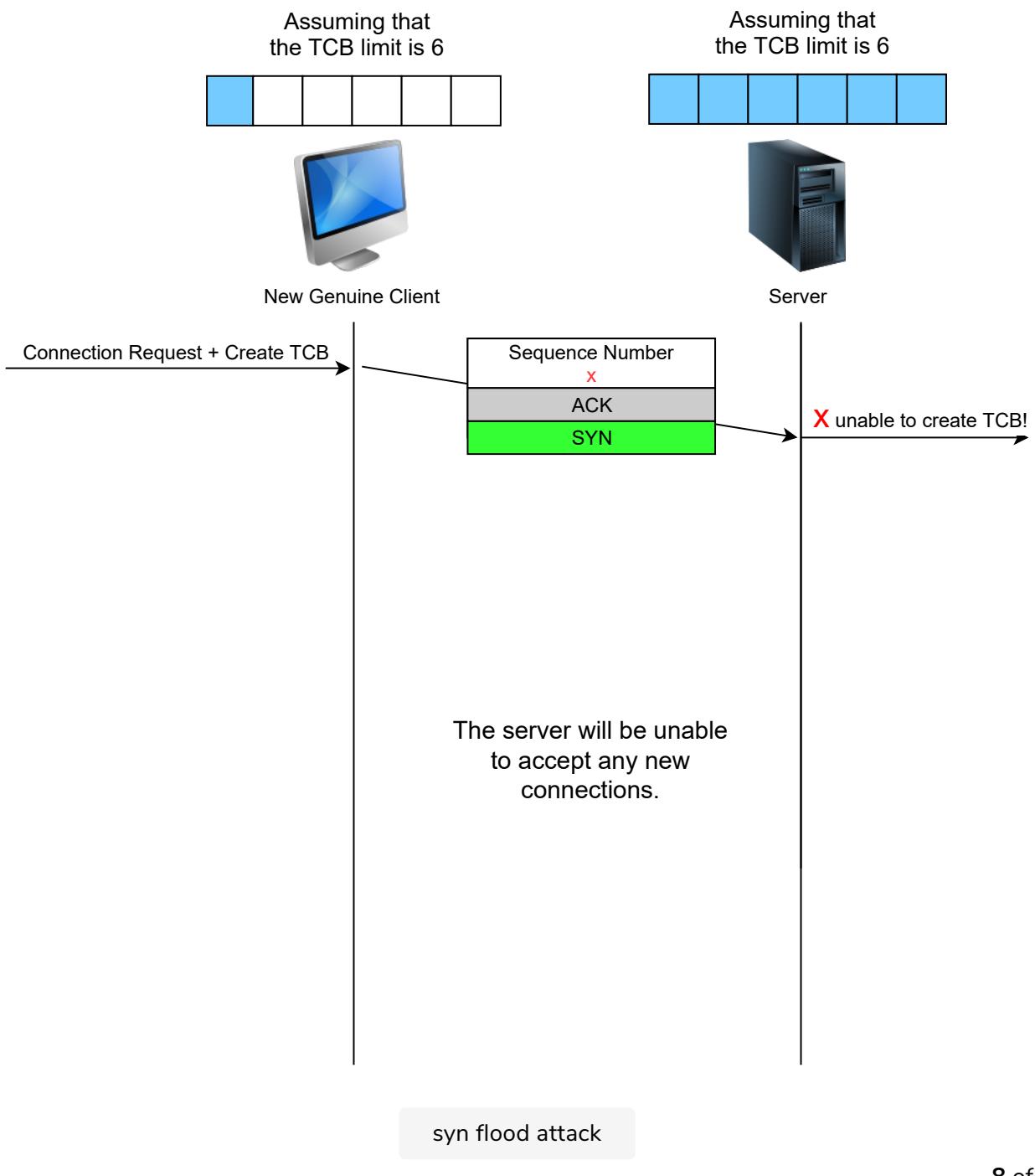


syn flood attack





syn flood attack



To avoid these **SYN flood attacks**, newer TCP implementations reply directly with SYN+ACK segments and wait until the reception of a valid ACK to create a TCB.

The goal is to not store connection state on the server immediately upon reception of a *SYN* packet. But, without this information, the server cannot tell if a subsequent *ACK* it receives is from a legitimate client that had sent a benign *SYN* packet. One way to do it is to verify that if the acknowledgement

number contained in the *ACK* packet is  $y$ , then the server had sent a sequence number  $y - 1$  in the *SYN+ACK* packet. But, again, if we are remembering the initial sequence number for each *SYN* packet, we are back to square one - remembering connection state. The way **SYN Cookie** solves this problem is to use a function that uses some information from the client's *SYN* packet and some information from the server side to calculate a random initial sequence number. This number, say,  $y - 1$  is sent to the client in a *SYN + ACK* message. If an *ACK* packet is later received with a sequence number  $y$ , using some packet header fields and some server side information, a reverse function can verify that the acknowledgement number is valid. If not, the connection is refused, otherwise a TCB is created and a connection is established.

The advantage of *SYN* cookies is that the server would not need to create and store a TCB upon reception of the *SYN* segment.

## Retransmitting Lost Segments #

Since the underlying Internet protocol provides an unreliable service, the *SYN* and *SYN+ACK* segments sent to open a TCP connection could be lost. Current TCP implementations start a **retransmission timer** when the first *SYN* segment is sent. This timer is often set to three seconds for the first retransmission, and then doubles after each retransmission ([RFC 2988](#)). When the timer expires, the segment is retransmitted. TCP implementations also enforce a maximum number of retransmissions for the initial *SYN* segment. Note that the same technique is used for all TCP segments, not just connection establishment segments.

We'll look at this in detail when we study TCP reliability in an upcoming lesson!

## Quick Quiz! #

1

A TCP connection establishment request may be refused due to which of the following reasons?

COMPLETED 0%

1 of 3



---

That is it for TCP connection establishment. What we'll cover next is how TCP releases a connection.

# TCP Connection Release

In this lesson, we'll discuss how TCP terminates established connections.

## WE'LL COVER THE FOLLOWING ^

- Abrupt Connection Release
- Graceful Connection Release
  - FSM
  - Receiving a FIN
    - Tracing through the FSM
  - Sending a FIN
    - Slides of Path 1A
- Quick Quiz!

TCP, like most connection-oriented transport protocols, supports two types of connection releases:

1. **Graceful** connection release, where the connection is not closed until both parties have closed their sides of the connection.
2. **Abrupt** connection release, where either one user closes both directions of data transfer or one TCP entity is forced to close the connection.

## Abrupt Connection Release #

We've already had a brief overview of abrupt connection release with *RST* segments in a previous lesson. Let's have a closer look.

An abrupt release is executed when a *RST* segment is sent. A *RST* can be sent for the following reasons:

- A non-*SYN* segment was received for a **non-existing TCP connection** ([RFC 793](#)).

- Some implementations send a *RST* segment when a segment with an **invalid header** is received on an open connection ([RFC 3360](#)). This causes the corresponding connection to be closed and has prevented attacks ([RFC 4953](#)).
- Some implementations send an *RST* segment when they need **to close an existing TCP connection** for any reason such as:
  - There are **not enough resources** to support this connection
  - The remote **host has stopped responding and is now unreachable**.

When a *RST* segment is sent by a TCP entity, it should contain the current value of the sequence number for the connection (or 0 if it does not belong to any existing connection), and the acknowledgment number should be set to the next expected in-sequence sequence number on this connection.

## Graceful Connection Release #

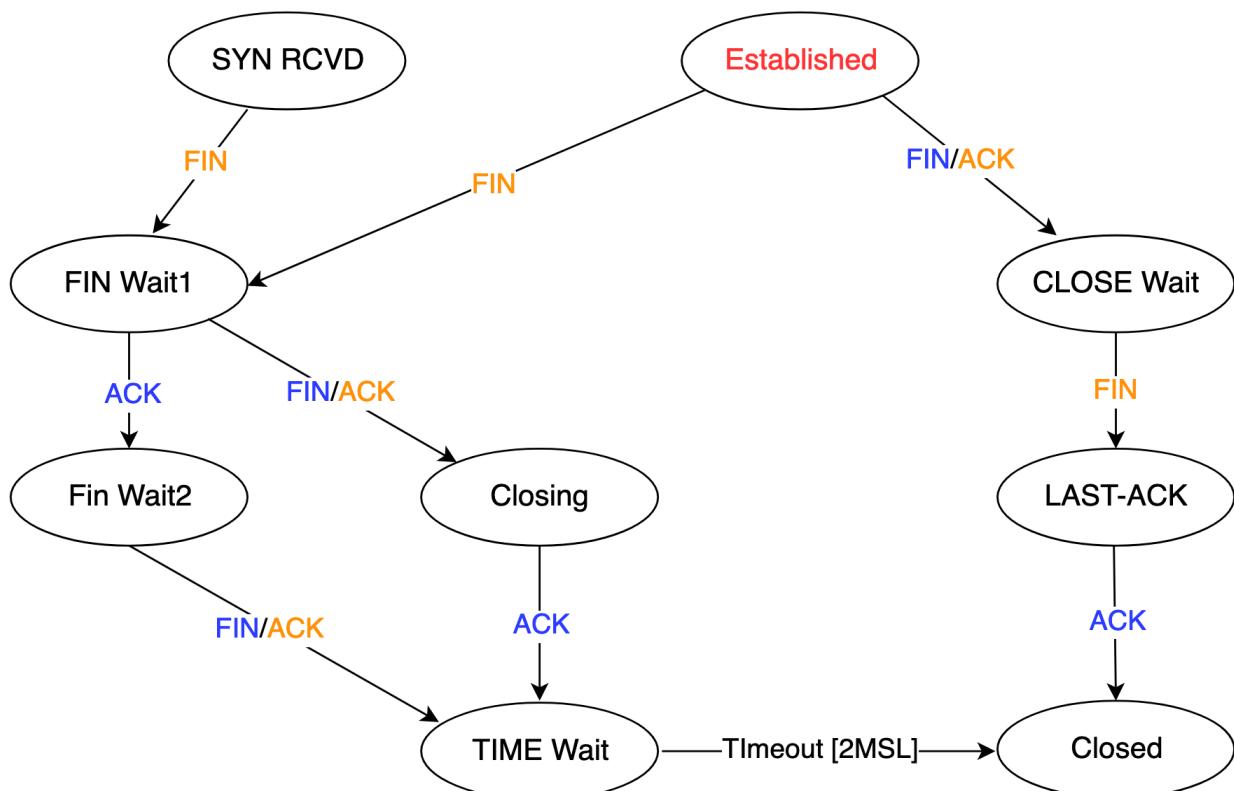
The normal way of terminating a TCP connection is by using the *FIN* flag of the TCP header. This ‘graceful mechanism’ allows each host to release its own side of the connection individually. The utilization of the *FIN* flag in the TCP header consumes one sequence number.

### FSM #

The following figure shows an FSM that depicts the various ‘graceful’ ways that a TCP connection can be released.

Don’t feel overwhelmed if you don’t understand it yet, we’ll study each possible path individually.

**X/Y:** X is transmitted & Y is received  
**Init:** initial state



Connection Release FSM

Starting from the **Established** state, there are two main paths through this FSM.

## Receiving a FIN #

Throughout the rest of this lesson we'll refer to the two hosts as **client** and **server**. In the case of this path, the client receives a *FIN* segment. Let's trace it.

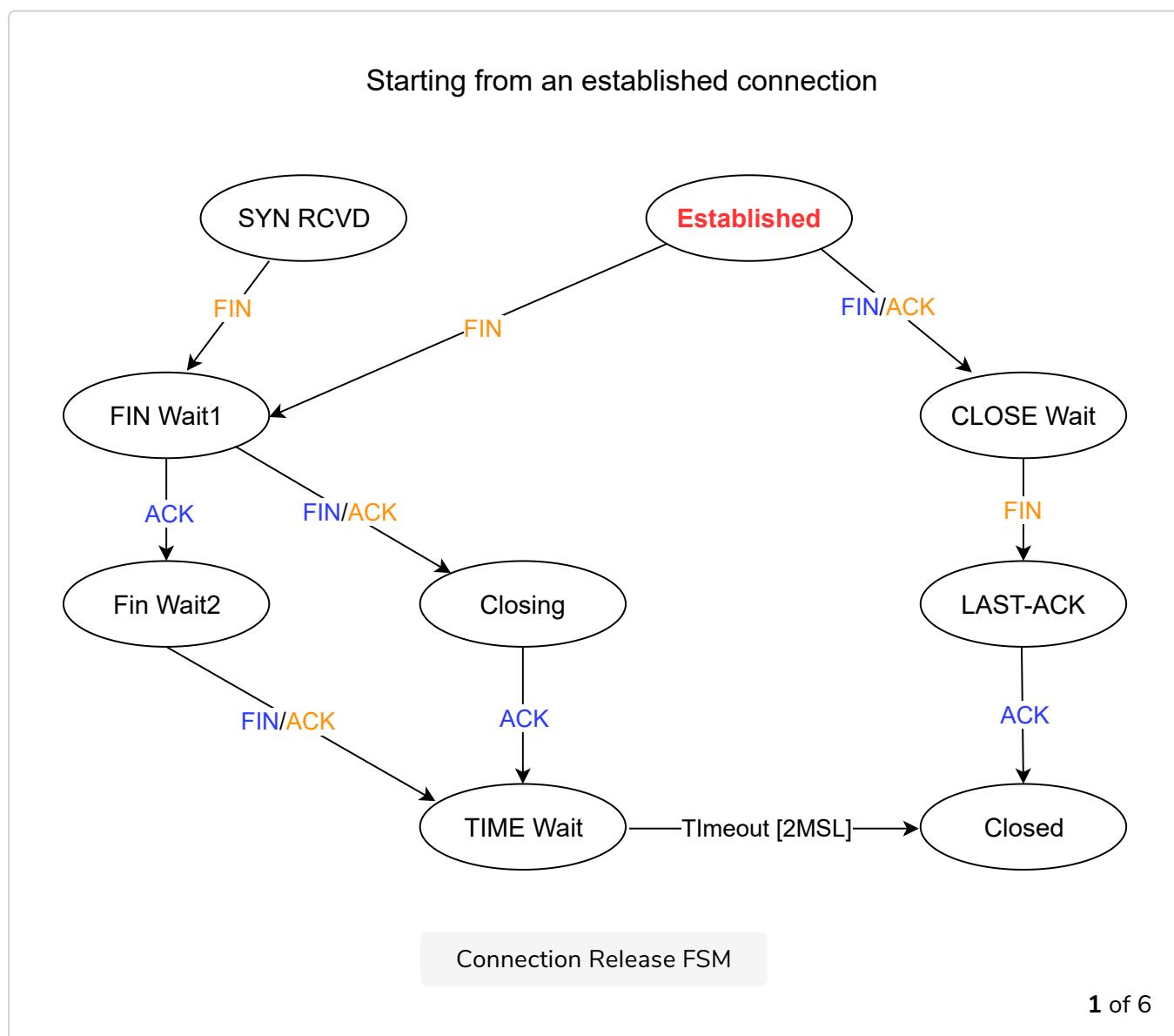
1. The client receives a segment with sequence number  $x$  and the *FIN* flag set. The utilization of the *FIN* flag indicates that the byte before sequence number  $x$  was the last byte of the byte stream sent by the server. The *FIN* segment is subject to the same retransmission mechanisms as a normal TCP segment. In particular, its transmission is protected by the retransmission timer that we'll look at in the next few lessons.
2. Once all of the data has been delivered to the application layer entity, the TCP entity sends an *ACK* segment to acknowledge the *FIN* segment it received in step (1), whose **acknowledgment number** field is set to

$$(x + 1) \bmod 2^{32}.$$

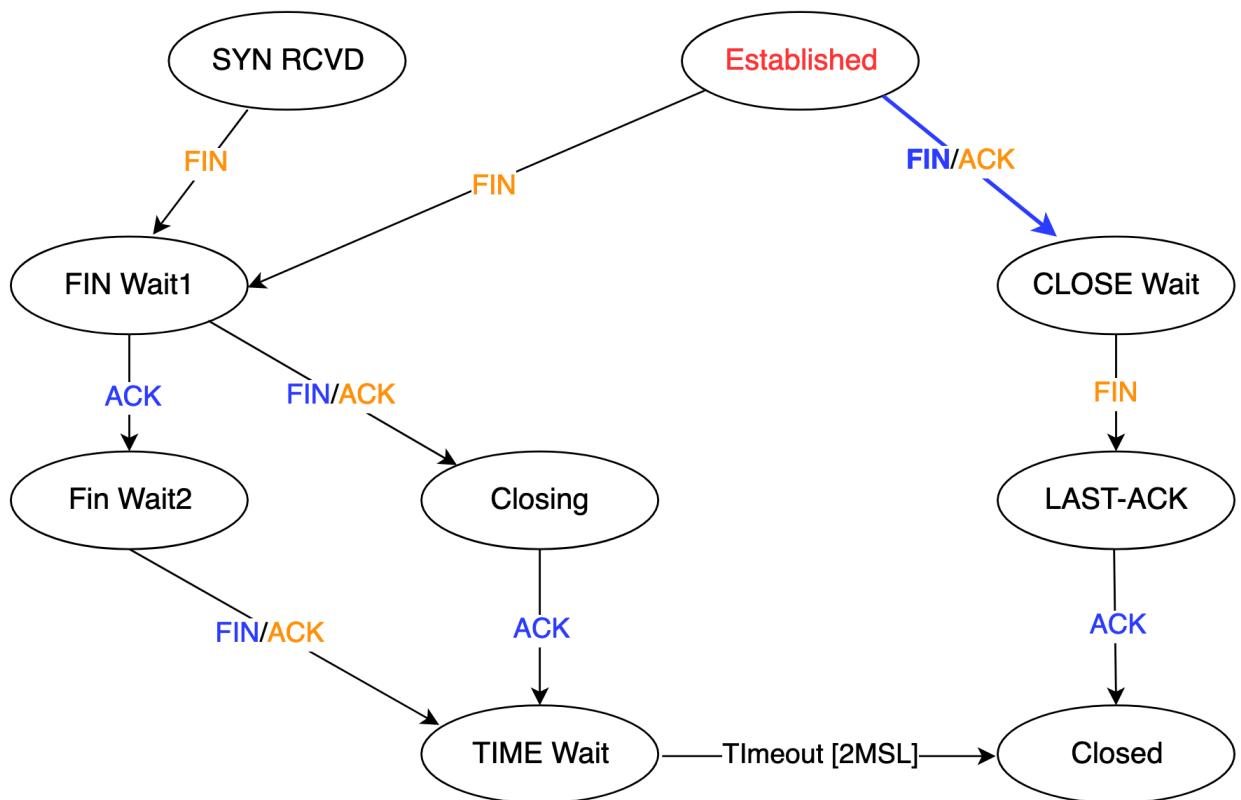
3. At this point, the TCP connection enters the **CLOSE\_WAIT** state. In this state, the client can still send data to the server.
4. Once the client has sent all the data that it was supposed to, it sends a *FIN* segment and enters the **LAST\_ACK** state. In this state, the client waits for the acknowledgment of its *FIN* segment. It may still retransmit unacknowledged data segments, e.g. if the retransmission timer expires.
5. Upon reception of the acknowledgment for the *FIN* segment, the TCP connection is completely closed and its TCB can be discarded.

### Tracing through the FSM #

Here are some slides tracing this path through the FSM.



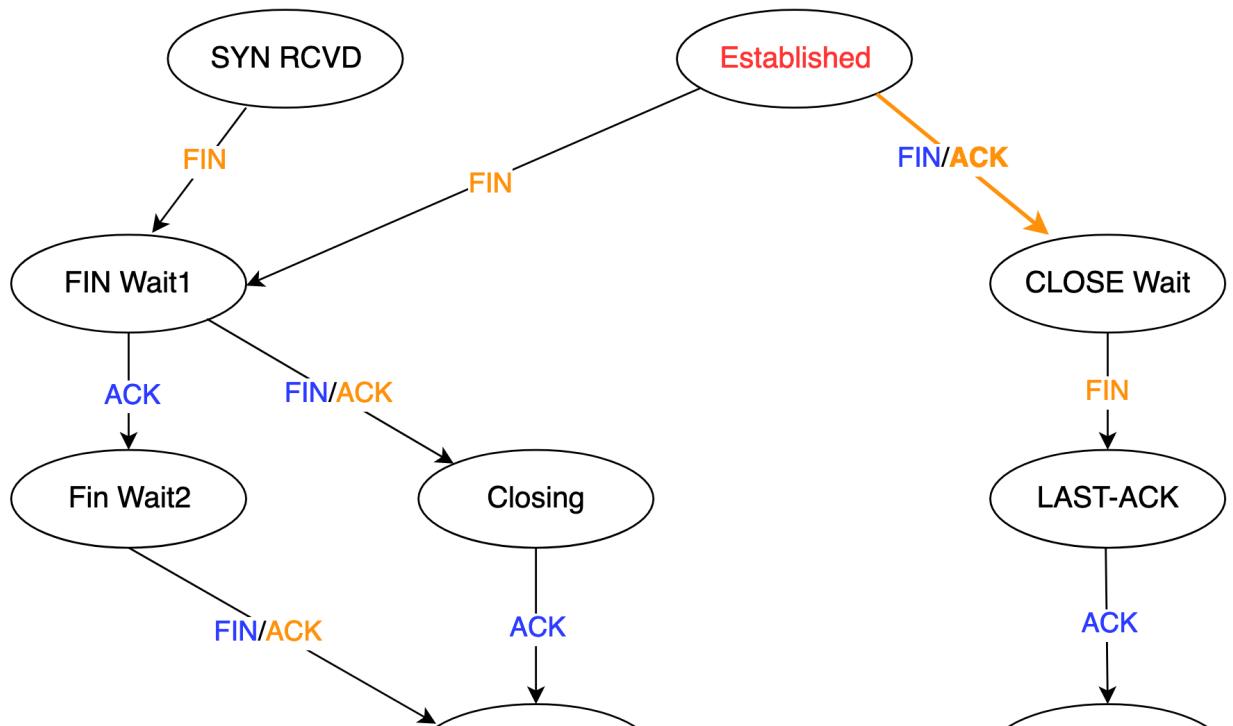
A FIN Segment is received from the server

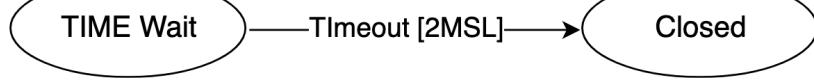


Connection Release FSM

2 of 6

An ACK Segment is sent back to acknowledge the FIN

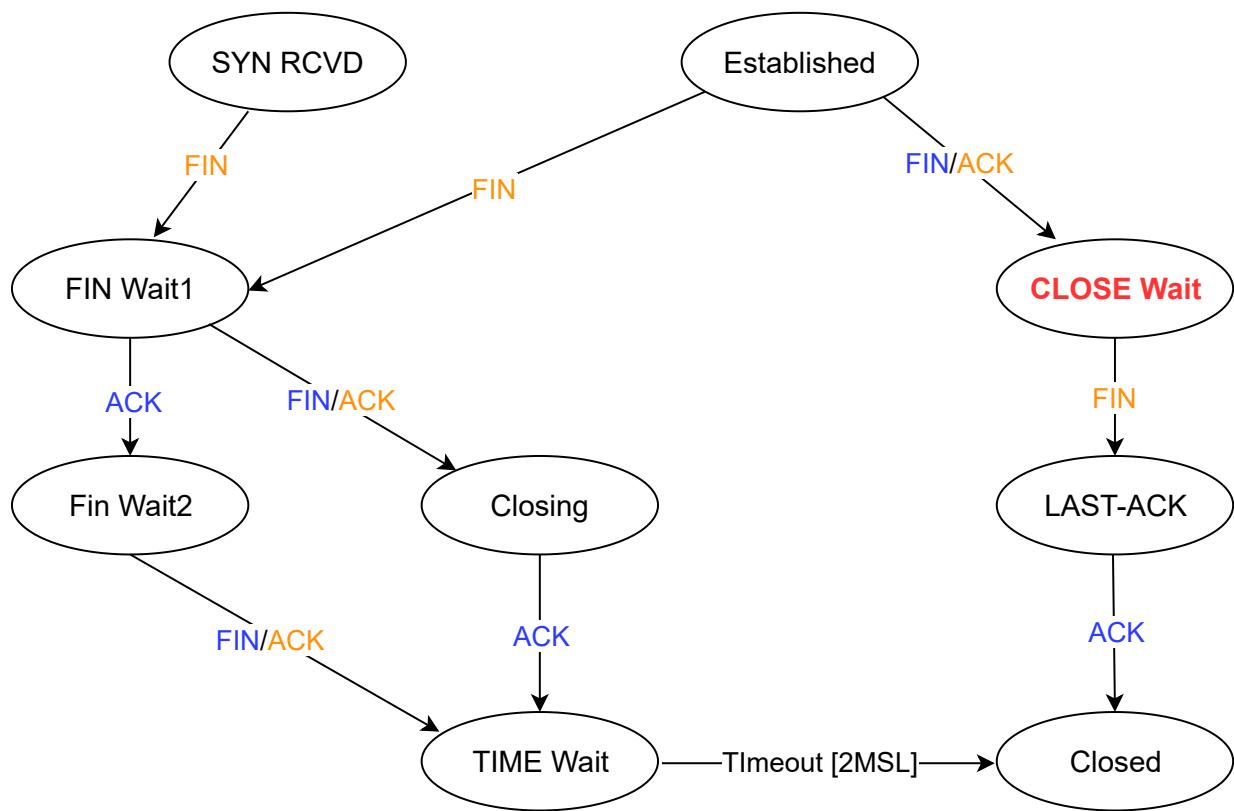




Connection Release FSM

3 of 6

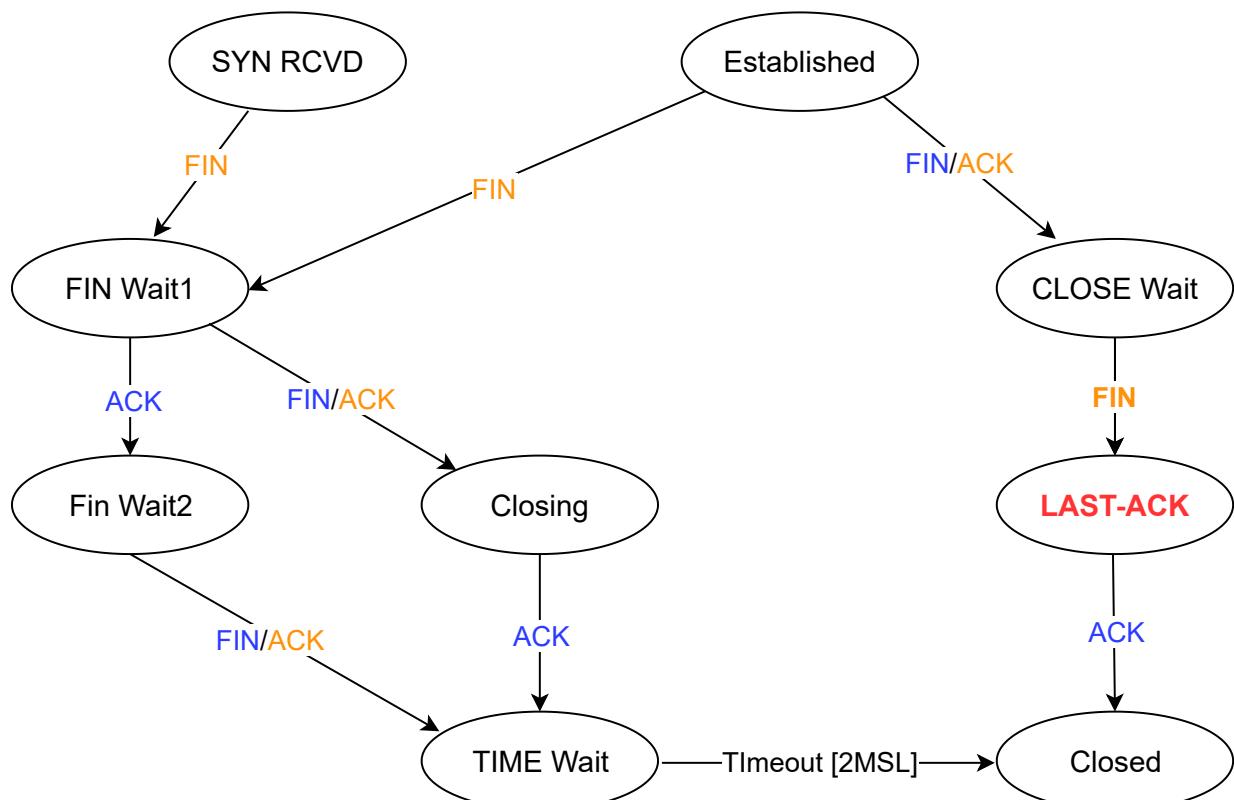
Now in CLOSE Wait state



Connection Release FSM

4 of 6

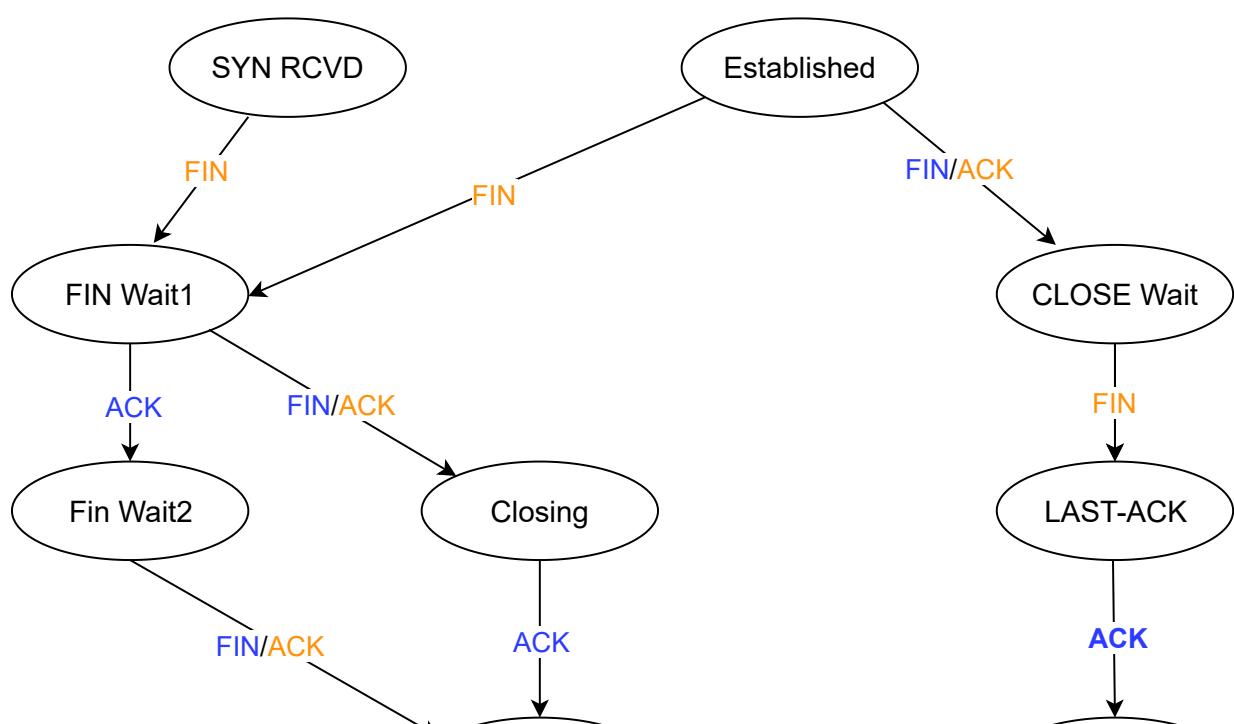
The client now sends a FIN to the server  
and enters the LAST-ACK state

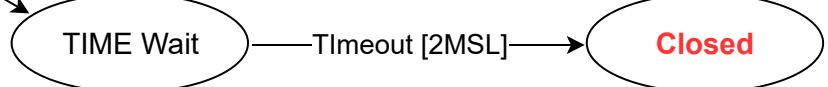


Connection Release FSM

5 of 6

The client now receives an ACK from the  
server and the connection is closed





### Connection Release FSM

6 of 6



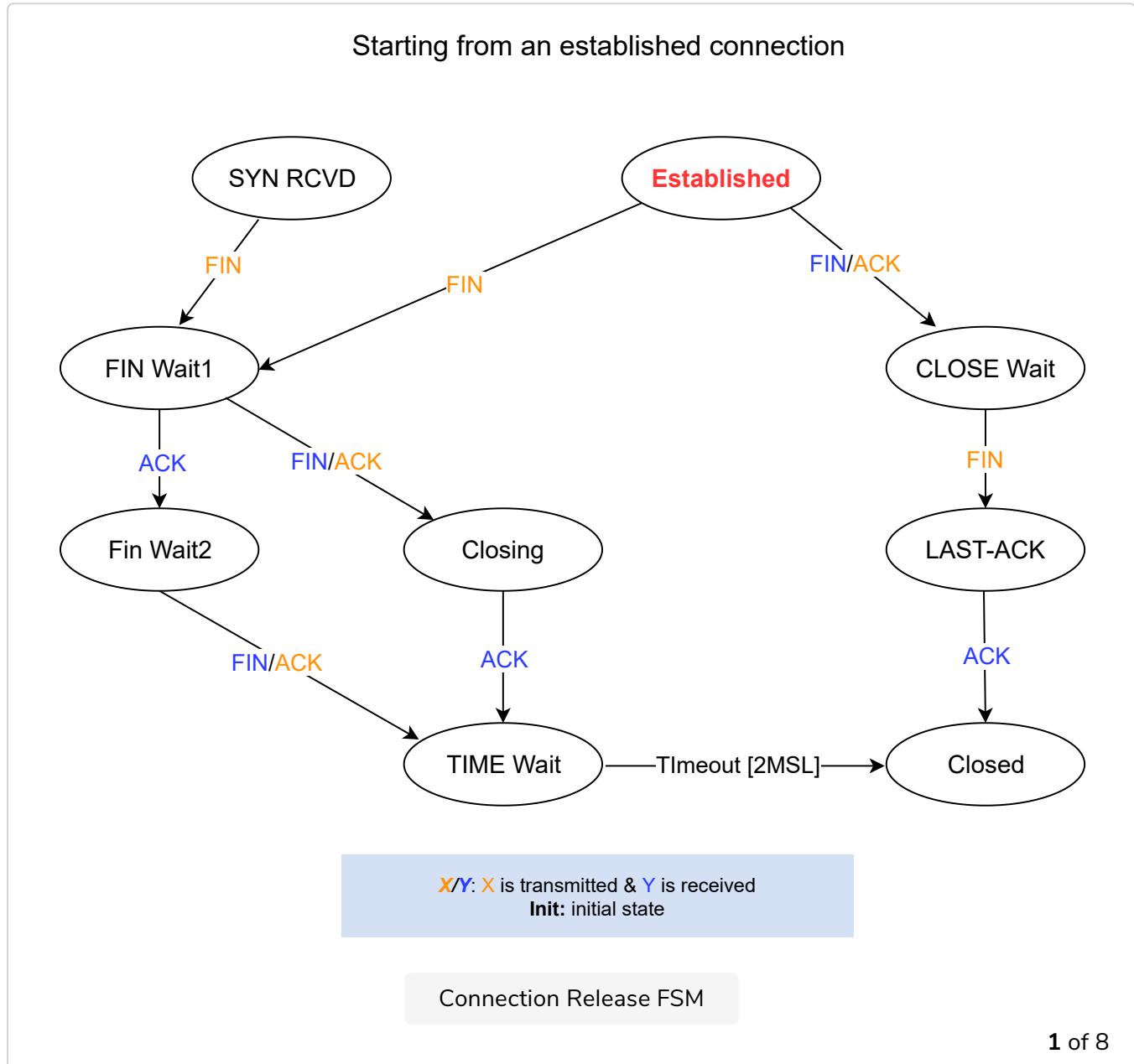
## Sending a FIN #

The second path is when the client decides first to send a *FIN* segment.

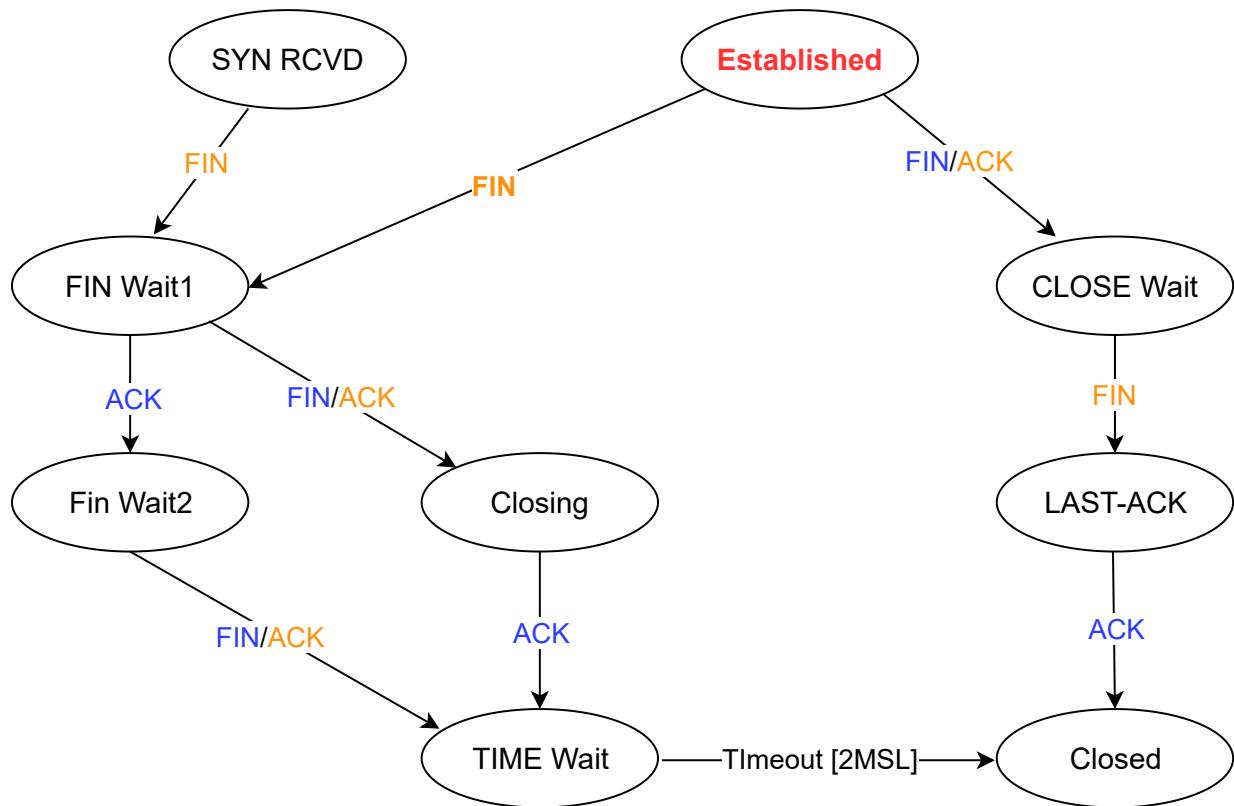
1. Upon sending the *FIN*, the connection enters the **FIN\_WAIT1** state. In this state, the client can retransmit unacknowledged segments, but cannot send *new* data segments. There are two paths that this one can split into after this:
  - A. The client receives an ACK segment in response to its *FIN*. The TCP connection enters the **FIN\_WAIT2** state in which new data segments from the server are still accepted until the reception of a *FIN* segment. The acknowledgment for this segment is sent once all the data before the *FIN* segment has been delivered to the client. After this, the connection enters the **TIME\_WAIT** state.
  - B. In the second case, a *FIN* segment is received from the server. The connection enters the **Closing** state once all the data from the server has been delivered to the client. In this state, no new data segments can be sent and the client waits for an acknowledgment of its *FIN* segment before entering the **TIME\_WAIT** state.
2. A TCP connection enters the **TIME\_WAIT** state after the client sends the last *ACK* segment to a server. This segment indicates to the server that all the data that it's sent has been correctly received and that it can safely release the TCP connection and discard the corresponding TCB.
3. The connection remains in the **TIME\_WAIT** state for  $2 \times$  a certain length of time called the **maximum segment lifetime** (MSL) seconds. The TCP standard defines MSL as 120 seconds (2 minutes). Although, this value is flexible in modern implementations. During the  $2 \times$  MSL period, the TCB of the connection is maintained on both ends to:

- allow retransmission of the sent ACK segments if any are lost.
- handle duplicate segments on the connection correctly without causing the transmission of a RST segment

## Slides of Path 1A #



The client sends a FIN to the server

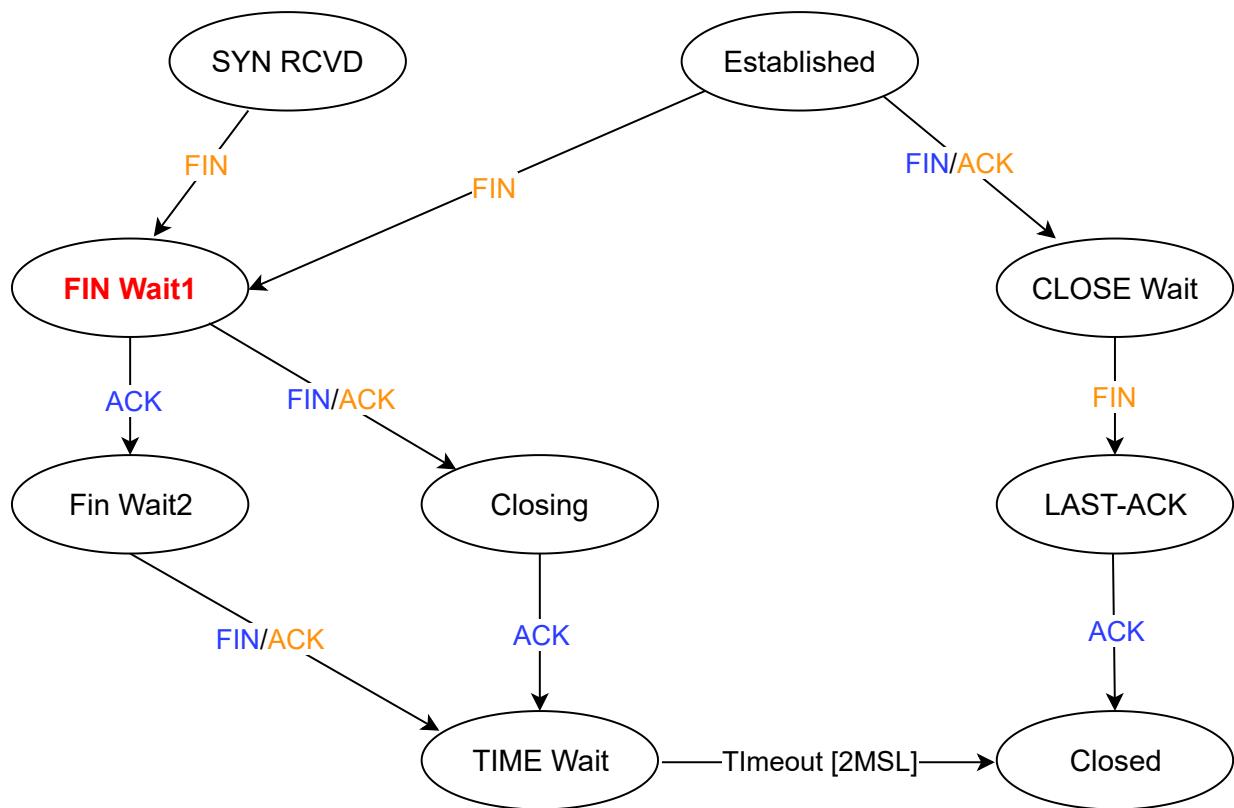


X/Y: X is transmitted & Y is received  
Init: initial state

Connection Release FSM

2 of 8

Now in the FIN WAIT1 state

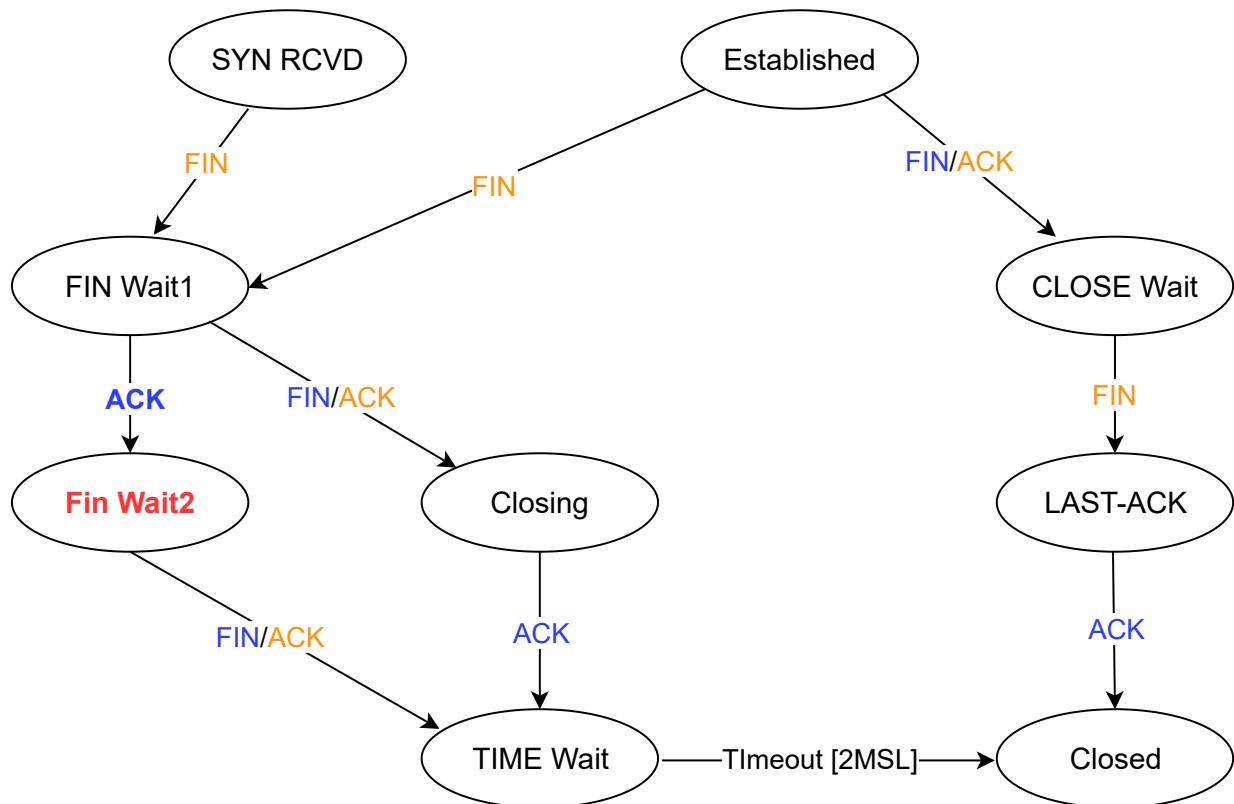


X/Y: X is transmitted & Y is received  
Init: initial state

Connection Release FSM

3 of 8

The client receives an ACK from the server.  
Now in the FIN Wait2 state.

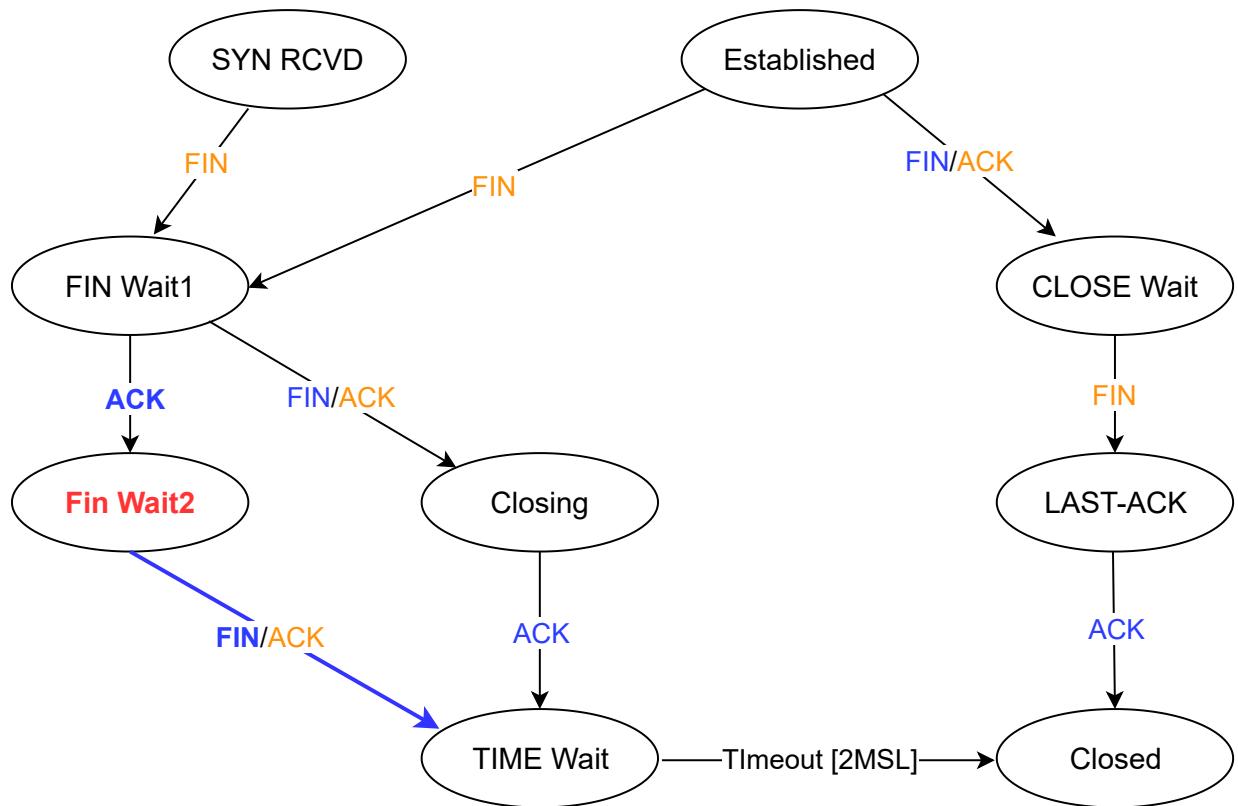


X/Y: X is transmitted & Y is received  
Init: initial state

Connection Release FSM

4 of 8

The client receives a FIN from the server.

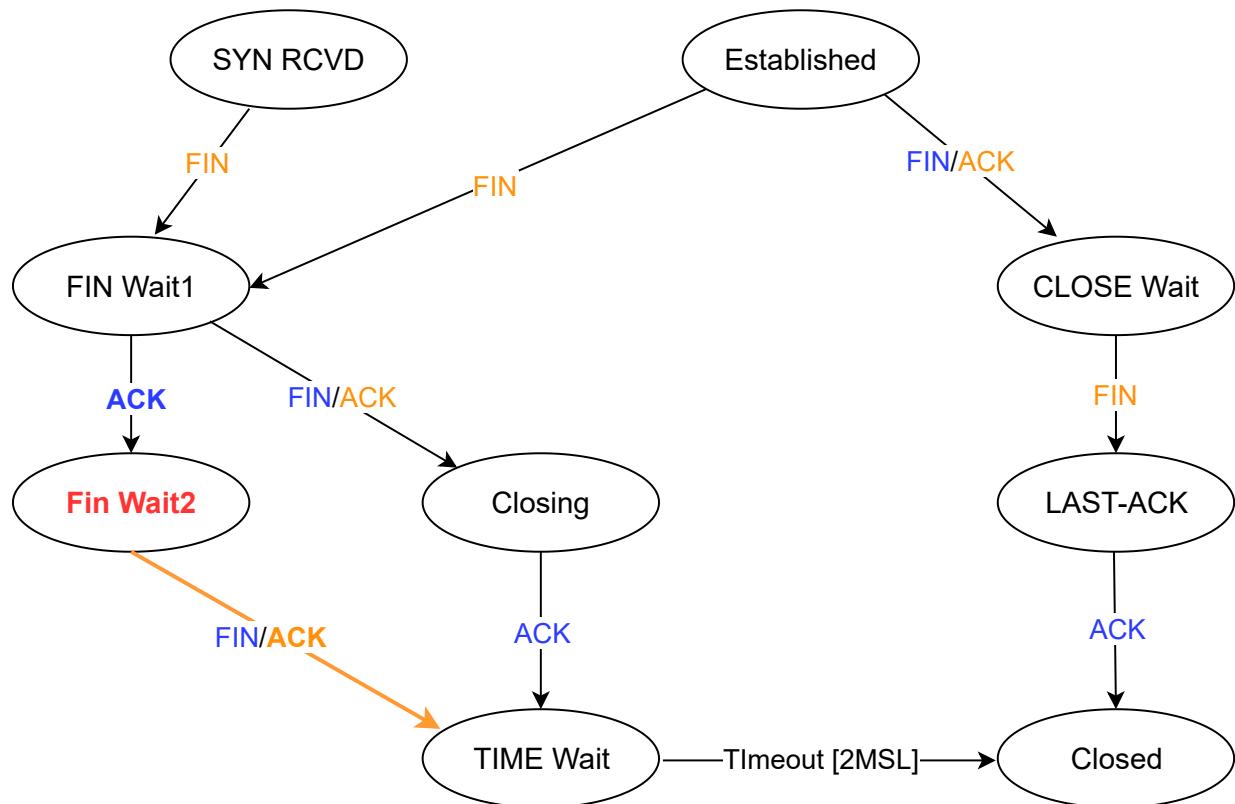


X/Y: X is transmitted & Y is received  
Init: initial state

Connection Release FSM

5 of 8

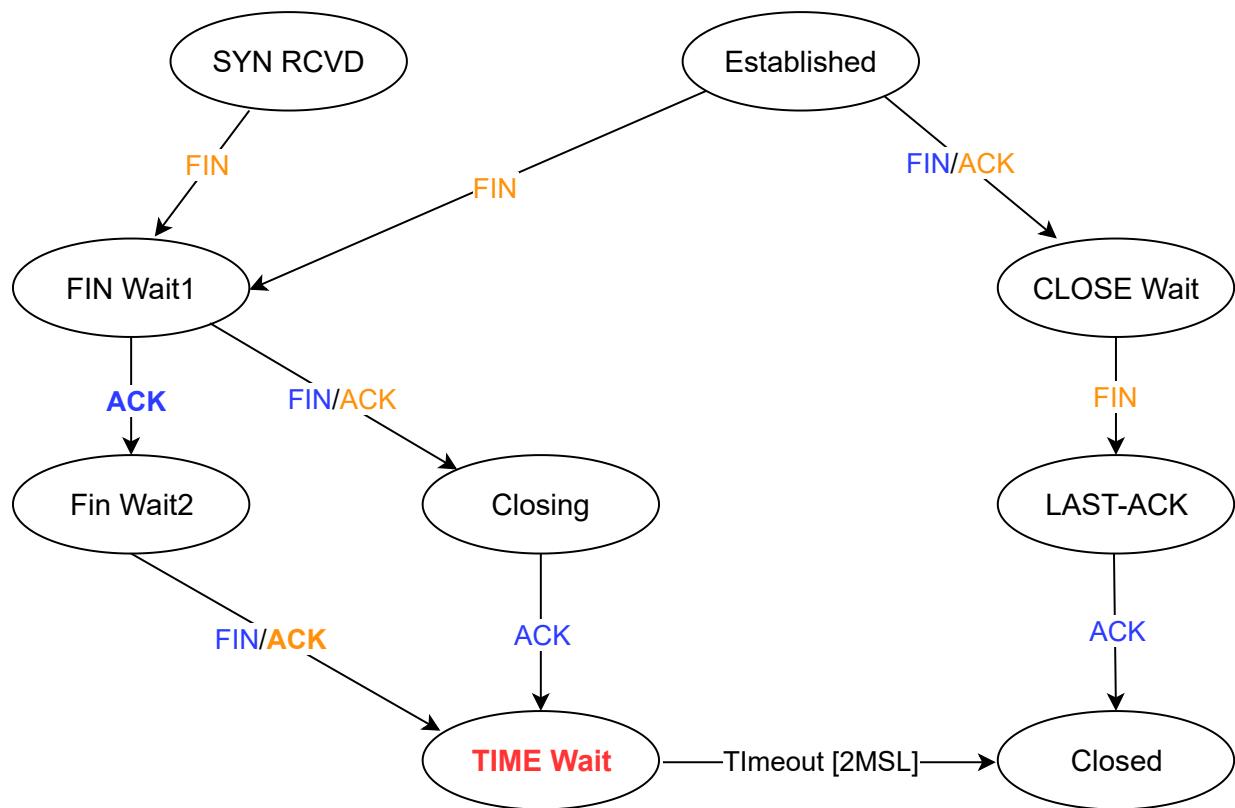
The client sends an ACK in response to the FIN sent from the server.



X/Y: X is transmitted & Y is received  
Init: initial state

Connection Release FSM

Now in the TIME Wait state.

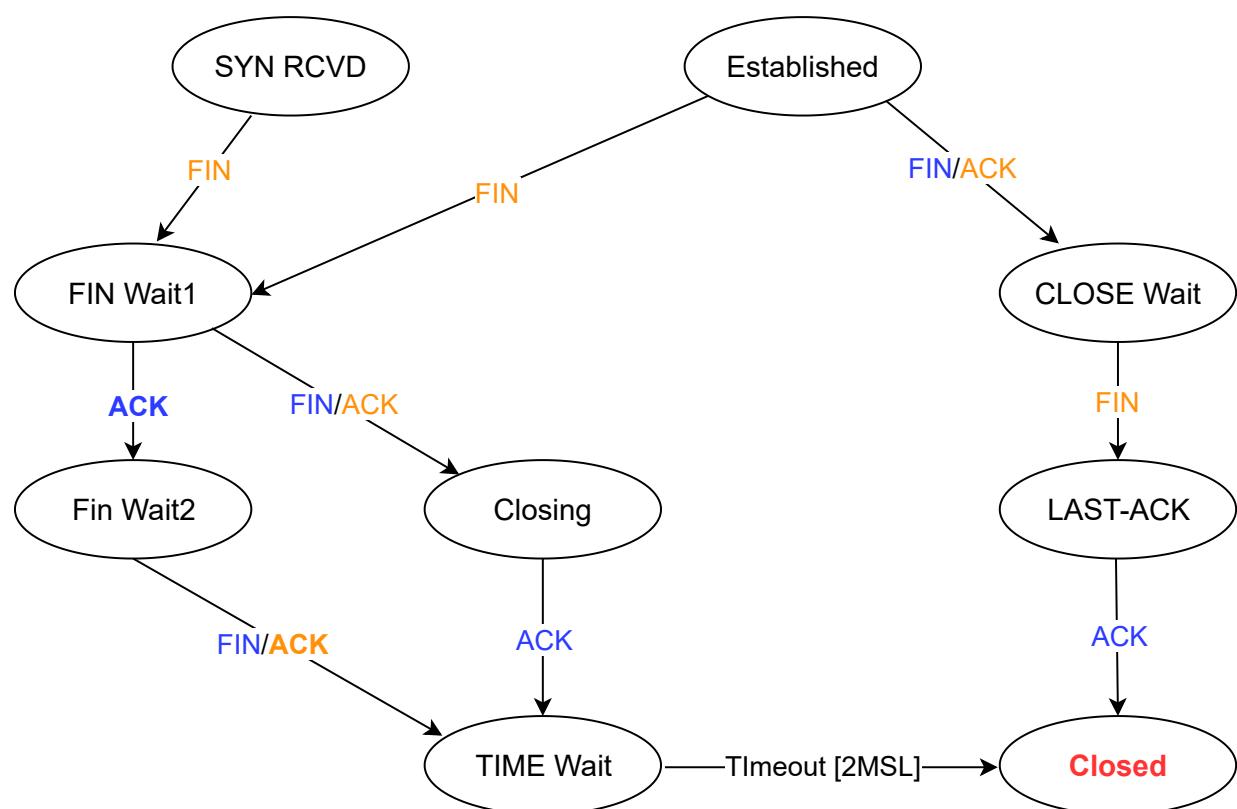


X/Y: X is transmitted & Y is received  
Init: initial state

Connection Release FSM

7 of 8

Finally closes after 2MSL.



X/Y: X is transmitted & Y is received  
Init: initial state

Connection Release FSM

8 of 8

## Quick Quiz! #

1

What are some reasons why a connection may get abruptly terminated?

COMPLETED 0%

1 of 3



Now that we're done with what connection release looks like in TCP, let's move on to efficient data transmission with TCP using Nagle's algorithm in the next lesson!

# Efficient data transmission with TCP

In this lesson, we'll study the main data transfer mechanisms used by TCP.

## WE'LL COVER THE FOLLOWING



- Segment Transmission Strategies
- Nagle's Algorithm
  - Algorithm
  - Limitations
- Quick Quiz!

## Segment Transmission Strategies #

In a transport protocol such as TCP that offers a byte stream, a practical issue that was left as an implementation choice in [RFC 793](#) was *when* a new TCP segment should be sent.

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](#).
  - **Advantage: Reduced overhead**
  - **Disadvantage:** Potentially at the cost of a **very high delay**, which may be unacceptable for interactive applications.

# Nagle's Algorithm #

An elegant solution to this problem was proposed by John Nagle in [RFC 896](#) called **Nagle's Algorithm**.

In essence, as long as there are unacknowledged packets, Nagle's algorithm keeps collecting application layer data up to maximum segment size, to be sent together in a single packet. This helps reduce the packetization overhead by reducing small packets.

## Algorithm #

```
if window size >= MSS and available data >= MSS:  
    send one MSS-sized segment  
else:  
    if there is unacknowledged data:  
        place data in buffer until acknowledgement is received  
    else:  
        send one TCP segment containing all buffered data
```

Nagle's Algorithm

The algorithm takes up 2-3 lines in a TCP implementation. 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.

## Limitations #

Nagle's has a few limitations:

1. Nagle's algorithm is only supported by TCP and no other protocols like UDP.
2. TCP applications that require **low latency** and **fast response times** such as internet phone calls or real-time online video games, do not work well when Nagle's is enabled. The delay caused by the algorithm triggers a noticeable lag. These applications usually *disable* Nagle's with an interface called the **TCP\_NODELAY** option.

3. The algorithm was originally developed at a time when computer networks supported much less bandwidth than they do today. It saved bandwidth and made a lot of sense at the time, however, the algorithm is much less frequently used today.
4. The algorithm also works poorly with **delayed ACKS**, a TCP feature that is used now. With both algorithms enabled, applications experience a consistent delay because Nagle's algorithm doesn't send data until an ACK is received and delayed ACKs feature doesn't send an ACK until after a certain delay.

## Quick Quiz! #

1

Consider the following scenario:

- The last segment received by a TCP entity had an ACK value of 10.
- The MSS value is 536.
- There are 50 bytes in the buffer waiting for transmission.
- The window size is 500.
- The application sends a 500 byte message.
- All the data sent so far has been acknowledged.

Will the data be buffered or sent on if Nagle's was active?

COMPLETED 0%

1 of 3



For now, that's all on Nagle's. Let's move on to TCP window-scaling mechanisms.

# TCP Window Scaling

In this lesson, we'll discuss TCP window scaling!

## WE'LL COVER THE FOLLOWING ^

- Problem: Small Windows Result in Inefficient Use of Bandwidth
  - Round Trip Time vs. Bandwidth vs. Throughput
- Transmission and Propagation Delays: Example
- Solution: Larger Windows
  - Scaling Factor
  - Deciding a Scaling Factor
- Improvement
- Quick Quiz!

## Problem: Small Windows Result in Inefficient Use of Bandwidth #

From a performance point of view, one of the main limitations of the original TCP design is that the 16-bit **window size** header limits the TCP receive window size to  $2^{16}$  bytes i.e., 65,535 bytes. That means the sender can send at most 65,535 bytes before it has to wait for an acknowledgment.

## Round Trip Time vs. Bandwidth vs. Throughput #

- **Round Trip Time** is the amount of time it takes to send a packet and receive its acknowledgment.
- **Bandwidth** is the rate at which the network can transport the bits.
- **Throughput** is the amount of data that is *actually* transferred from one end-system to another.

So what if the round trip time is short enough to accommodate sending more data without having to wait for acknowledgments? The table below shows the

rough maximum throughput that can be achieved by a TCP connection with a 64 Kbytes window as a function of the connection's round-trip-time:

RTT	Maximum Throughput
1 msec	524 Mbps
10 msec	52.4 Mbps
100 msec	5.24 Mbps
500 msec	1.05 Mbps

This limitation was not a severe problem when TCP was designed, because at the time the available bandwidth was 56 kbps at best. However, in today's networks where the bandwidth can be in order gigabytes, this limitation is not acceptable.

## Transmission and Propagation Delays: Example #

- Consider a **1.544 Mbps** link from one end host to another over a **satellite** link that is **36000 km** above the surface of the earth.
- It takes  $\frac{2 \times 72000 \times 1000}{3 \times 10^8} = 480$  ms to get to the end host and then for the acknowledgement to return (ignoring the transmission time of the ack). This is the total distance to be covered by two segments divided by the speed of light.
- In the same amount of time, the sender could put  $\frac{480 \times 1544000}{1000} = 741120$  bits = 92640 bytes = 90.5 kB on the network.
- Now, even if the sliding window is at its maximum of 64 kB, the sender will only transmit 64 kB and then wait, sitting idle until an acknowledgement is received from the other side.
- That is a network utilization of  $\frac{90.5}{64} = 70.71\%$ . If we factor in the overheads of headers from various layers, the utilization drops further.

Note that we don't use satellite links much these days, but effectively,

the round trip delay is still a significant fraction of the transmission time.

whenever the product of bandwidth and delay is high, which is common with today's high speed networks, we face the same problem. So, if the delay is small, but the bandwidth is high, the sender can still put out a lot of bytes really quickly on the wire and still have to wait for an ACK sitting idle.

## Solution: Larger Windows #

To solve this problem, a backward-compatible extension that allows TCP to use larger receive windows was proposed in [RFC 1323](#).

**Basic idea:** instead of storing the size of the sending window and receiving window as 16-bit integers in the TCB, we keep the 16-bit window size, but introduce a multiplicative scaling factor.

### Scaling Factor #

As the TCP segment header only contains 16 bits to place the window field, it is impossible to copy the size of the sending window in each sent TCP segment. Instead, the header contains:

$$\text{sending window} \ll S$$

where  $S$  is the scaling factor ( $0 \leq S \leq 14$ ) negotiated during connection establishment and is specified in the header options field. ' $\ll$ ' is the bitwise shift operator. This operation pads zeros at the right and discards the bits on the left essentially multiplying by 2 for each shift.

### Deciding a Scaling Factor #

The client adds its proposed scaling factor as a TCP option in the SYN segment. If the server supports scaling windows, it sends the scaling factor in the SYN+ACK segment when advertising its own receive window. The local and remote scaling factors are included in the TCB. If the server does not support scaling windows, it ignores the received option and no scaling is applied. So scaling only applies when both parties support it.

Note that the protection mechanism of not maintaining state from the SYN packet via SYN cookies has the disadvantage that **the server wouldn't remember the proposed scaling factor**.

### Improvement #

By using the window scaling extensions defined in [RFC 1323](#), TCP implementations can use a receive buffer of up to 1 GByte. With such a receive buffer, the maximum throughput that can be achieved by a single TCP entity is delineated in the following table:

RTT	Maximum Throughput
1 msec	8590 Gbps
10 msec	859 Gbps
100 msec	86 Gbps
500 msec	17 Gbps

These throughputs are acceptable in today's networks as there are already servers with 10 Gbps interfaces.

## Quick Quiz! #

1

Given the window scaling factor of 6 and window size of 3125, how many bytes can the sender send without waiting for an acknowledgement?

Did you know that you can measure the round trip time on your network?  
Yes, it's true. Let's see how in the next lesson.

# Exercise: Measuring RTT with Ping

WE'LL COVER THE FOLLOWING ^

- Round-trip-time
- Pinging Google
- Dissecting The Output

## Round-trip-time #

Just to recap the last lesson, the **Round Trip Time** of a connection is the amount of time it takes to send a packet and receive its acknowledgment.

The `ping` command can be used to measure the round-trip-time to send and receive packets from a remote host. We're just pinging google from here because the location of the server that actually runs these commands from our website may change over time. However, if you try this locally, chose a remote destination which is far from your current location, e.g., a small web server in a distant country.

Checkout [ping's manpage](#) for more details!

## Pinging Google #

● Terminal



## Dissecting The Output #

```

PING google.com (172.217.212.102) 56(84) bytes of data.
64 bytes from 172.217.212.102: icmp_seq=1 ttl=53 time=1.15 ms
64 bytes from 172.217.212.102: icmp_seq=2 ttl=53 time=0.694 ms
64 bytes from 172.217.212.102: icmp_seq=3 ttl=53 time=0.617 ms
64 bytes from 172.217.212.102: icmp_seq=4 ttl=53 time=0.649 ms
64 bytes from 172.217.212.102: icmp_seq=5 ttl=53 time=0.598 ms
64 bytes from 172.217.212.102: icmp_seq=6 ttl=53 time=0.719 ms
64 bytes from 172.217.212.102: icmp_seq=7 ttl=53 time=0.695 ms
64 bytes from 172.217.212.102: icmp_seq=8 ttl=53 time=0.653 ms
64 bytes from 172.217.212.102: icmp_seq=9 ttl=53 time=0.629 ms
64 bytes from 172.217.212.102: icmp_seq=10 ttl=53 time=0.647 ms
64 bytes from 172.217.212.102: icmp_seq=11 ttl=53 time=0.634 ms
64 bytes from 172.217.212.102: icmp_seq=12 ttl=53 time=0.610 ms
64 bytes from 172.217.212.102: icmp_seq=13 ttl=53 time=0.683 ms
64 bytes from 172.217.212.102: icmp_seq=14 ttl=53 time=0.613 ms Sequence
64 bytes from 172.217.212.102: icmp_seq=15 ttl=53 time=0.636 ms number and
64 bytes from 172.217.212.102: icmp_seq=16 ttl=53 time=0.634 ms TTL of the
64 bytes from 172.217.212.102: icmp_seq=17 ttl=53 time=0.644 ms packet
64 bytes from 172.217.212.102: icmp_seq=18 ttl=53 time=0.632 ms
64 bytes from 172.217.212.102: icmp_seq=19 ttl=53 time=0.708 ms
64 bytes from 172.217.212.102: icmp_seq=20 ttl=53 time=0.656 ms
64 bytes from 172.217.212.102: icmp_seq=21 ttl=53 time=0.620 ms

```

IP address of domain that was looked up

Size of the packet

Round trip time

Note that the **Time To Live (TTL)** is the number of routers a packet can hop. So a ttl of 53 means it can jump 53 more ‘hops’ before being discarded. Each router or intermediary forwarding device decreases the TTL by one. This was a rather simplified definition of the term. We will study it in detail in the next chapter.

---

Now that we know some quirks and alterations added to optimize TCP over time, let’s get into TCP congestion control algorithms!

# TCP Congestion Control: AIMD

The last part of TCP that we're going to study is congestion control.

## WE'LL COVER THE FOLLOWING



- Requirements of a Congestion Control Algorithm
- TCP Congestion Control Algorithms
  - Additive Increase Multiplicative Decrease
- Quick Quiz!

We've already looked at what congestion control is in a [previous lesson](#). Let's get into some TCP-specific congestion control algorithms now!

But first:

## Requirements of a Congestion Control Algorithm



1. The congestion control scheme **must avoid congestion**. In practice, this means that the algorithm should ensure that the bandwidth allocated to a certain host at any given time does not exceed the bandwidth of the bottleneck link.
2. The congestion control scheme **must be efficient**. The congestion control scheme should ensure that the available bandwidth is efficiently used. Namely that the bandwidth allocated to a certain host at any given time doesn't fall too far below the bandwidth of the bottleneck link.
3. The congestion control scheme **should be fair**. Most congestion schemes aim at achieving max-min fairness, which we had a lesson dedicated to.

## TCP Congestion Control Algorithms



To implement most TCP congestion control algorithms, a TCP host must be

To implement most TCP congestion control algorithms, a TCP host must be able to control its **transmission rate**. In order to do so, it can constrain its

**sending window**. Recall that after transmitting data equal to the window size, the sender must pause and wait at least one RTT for the *ACK* before it can transmit more data. Thus the maximum data rate is:  $\frac{\text{window}}{\text{rtt}}$  where *window* is the maximum between the host's sending window and the window advertised by the receiver.

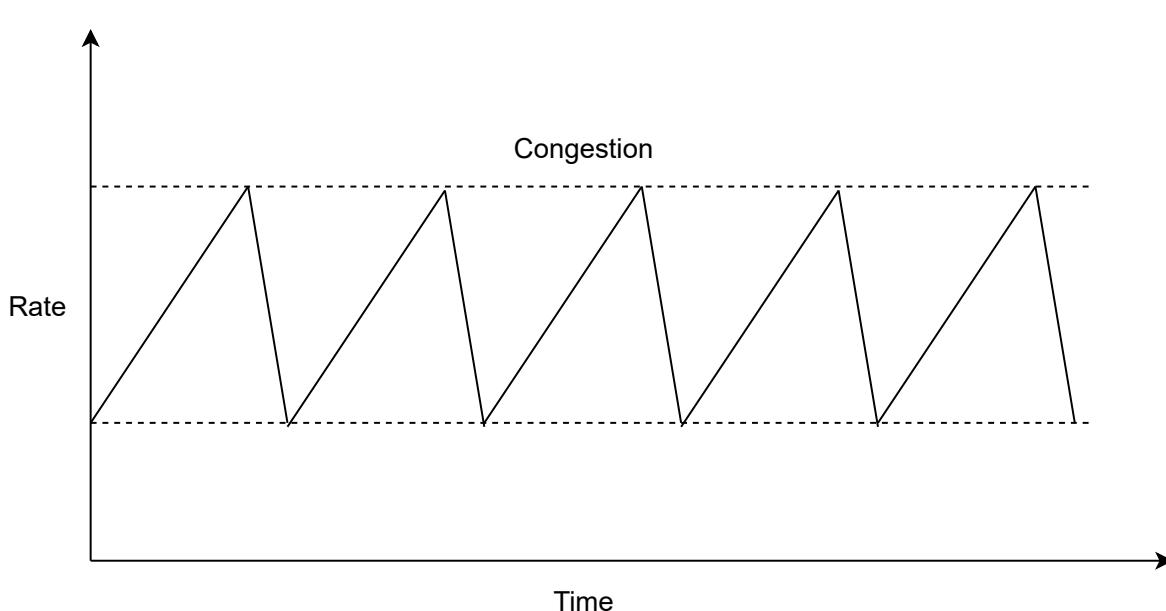
Hence, TCP's congestion control scheme can restrict sending windows based on a **congestion window**. The current value of the congestion window is stored in the TCB of each TCP connection. The value of the window that can be used by the sender is

$$\min(\text{congestion window}, \text{receiving window}, \text{sending window}).$$

## Additive Increase Multiplicative Decrease #

The **Additive Increase Multiplicative Decrease** algorithm *decreases* the transmission rate of a host when congestion has been detected in the network. Congestion is detected when acknowledgments for packets do not arrive before the transmission timer times out. Furthermore, it *increases* their transmission rate when the network is *not* congested. Hence, the rate allocated to each host fluctuates with time, depending on the feedback received from the network.

The figure below illustrates the evolution of the transmission rates allocated to a host in a network.



The **Additive Increase** part of the TCP congestion control increments the congestion window by  $MSS$  bytes every round-trip time. In the TCP literature, this phase is often called the *congestion avoidance* phase. Once congestion is detected, the **Multiplicative Decrease** part of the TCP congestion control reacts by multiplying the current value of the congestion window with a number greater than 0 and less than 1.

However, since the increase in the window is so slow, the TCP connection may have to wait for many round-trip times before being able to efficiently use the available bandwidth. To avoid this, the TCP congestion control scheme includes the **slow-start** algorithm.

## Quick Quiz! #

1

Which of the following is not a requirement of a congestion control algorithm?

COMPLETED 0%

1 of 2



Let's have a look at the slow start algorithm in the next lesson.



# TCP Congestion Control: Slow Start

In this lesson, we're going to look at the slow start algorithm.

## WE'LL COVER THE FOLLOWING



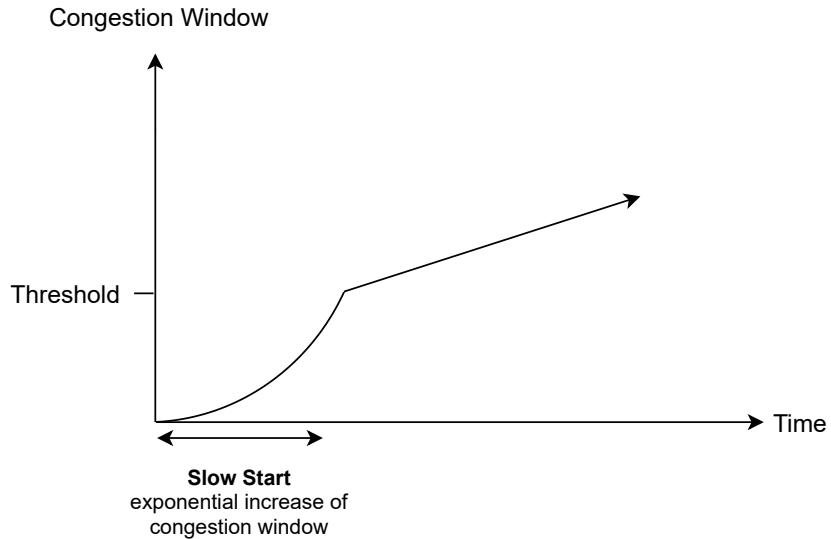
- How Slow Start Works
- Not All Congestion Is the Same!
  - Severe Congestion
    - Figure
  - Mild Congestion
    - Figure
- Quick Quiz!

## How Slow Start Works #

The objective of TCP slow-start is to quickly reach an acceptable value for the congestion window.

During slow-start:

1. The congestion window is **doubled every round-trip time**.
2. The slow-start algorithm uses an additional variable in the TCB to maintain the **slow-start threshold**.
  - The slow-start threshold is an estimation of the last value of the congestion window that did *not* cause congestion.
  - It is initialized at the sending window and is updated after each congestion event.



Congestion Window Size Change with Slow Start

## Not All Congestion Is the Same! #

The TCP congestion control scheme distinguishes between two types of congestion:

1. Severe Congestion
2. Mild Congestion

Let's discuss these one by one.

### Severe Congestion #

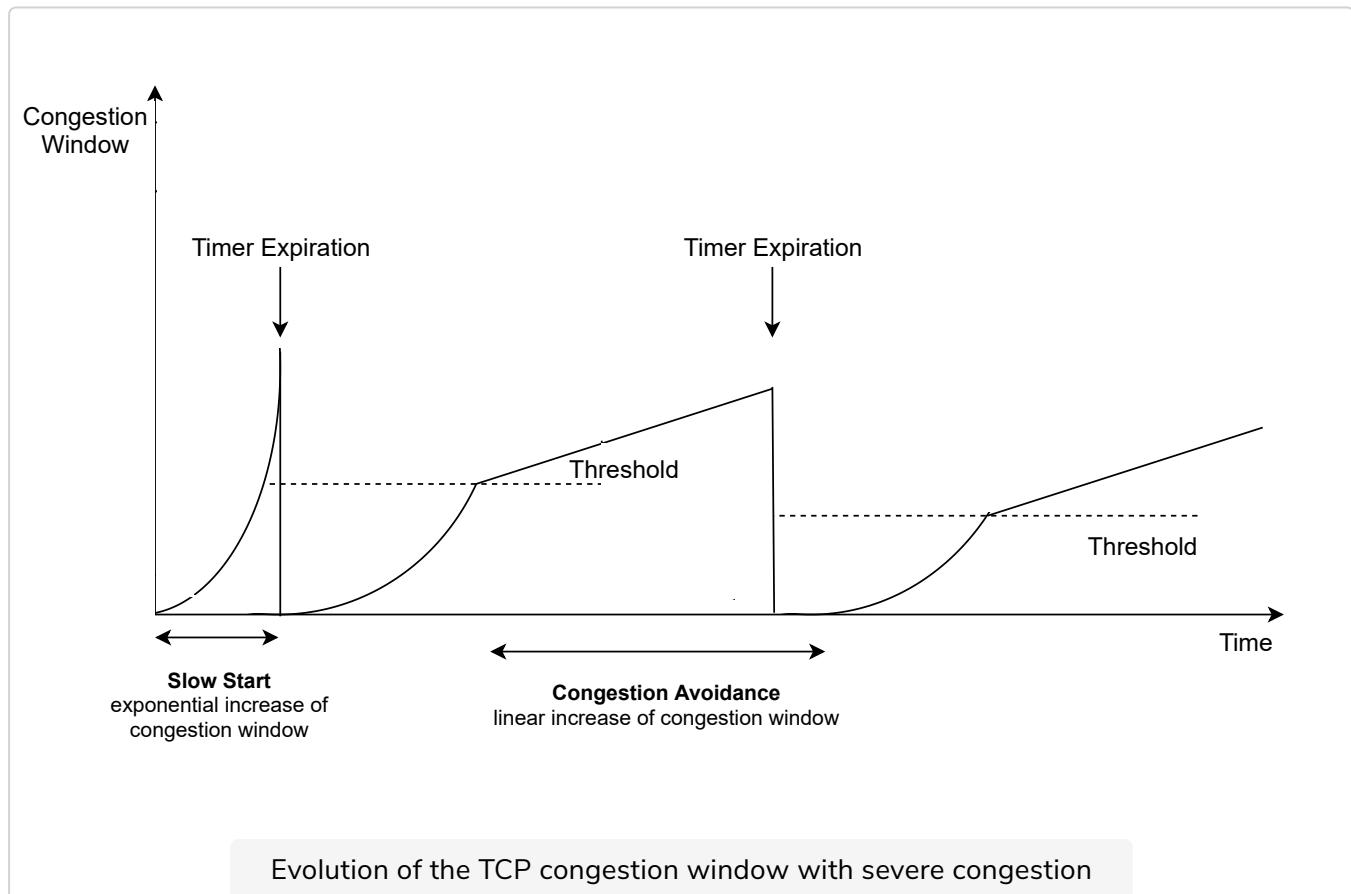
TCP considers that the network is severely congested when its retransmission timer expires. The following process is followed accordingly:

1. The sender performs slow-start until the first segments are lost and the retransmission timer expires.
2. At this time, TCP retransmits the first segment and the slow start threshold is set to half of the current congestion window. Then the congestion window is reset at one segment.
3. The last segments are retransmitted as the sender again performs slow-start.

3. The lost segments are retransmitted as the sender again performs slow-start until the congestion window reaches the slow start threshold.
4. It then switches to congestion avoidance and the congestion window increases linearly until segments are lost and the retransmission timer expires.

### Figure #

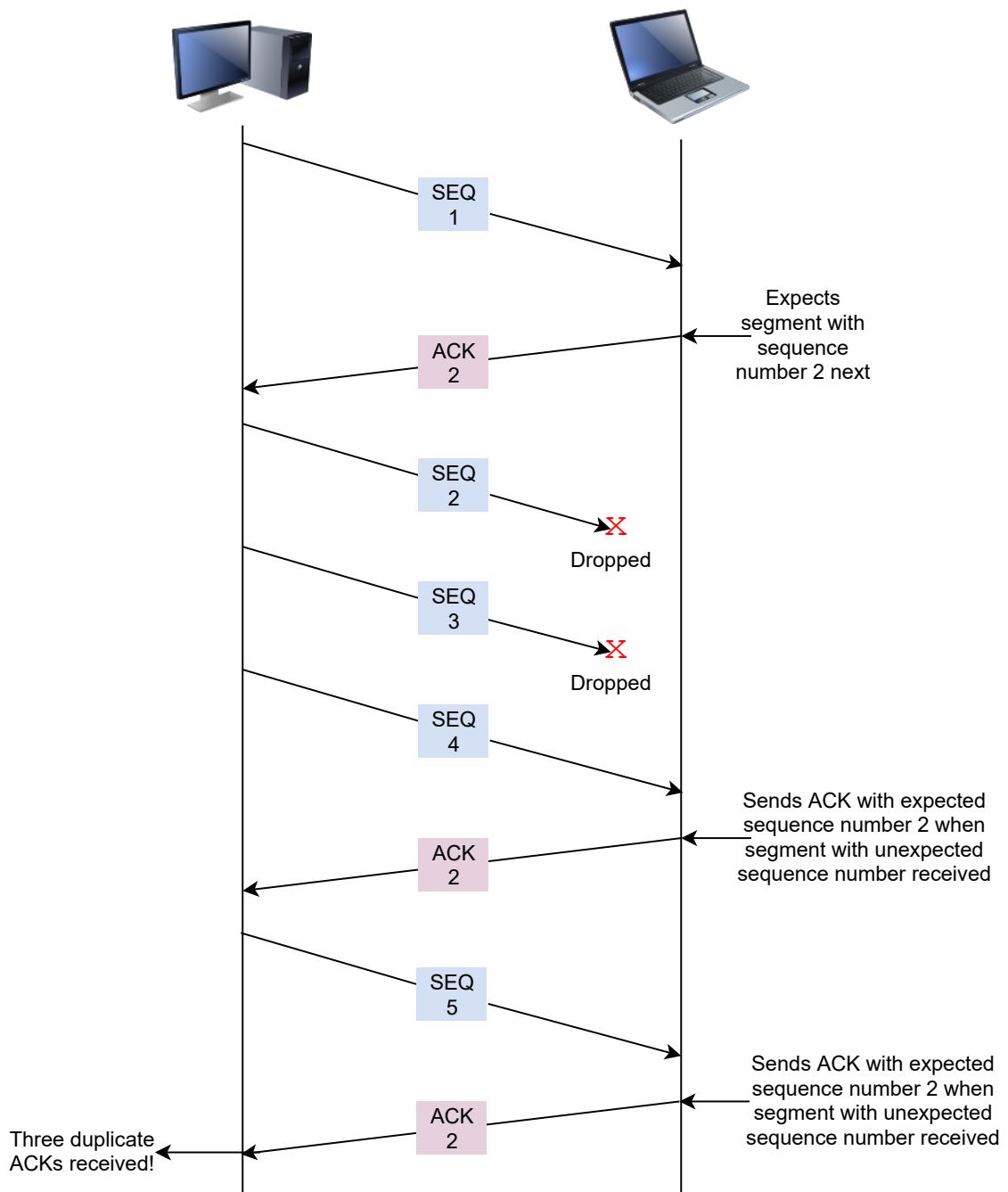
The figure below illustrates the evolution of the congestion window when there is severe congestion:

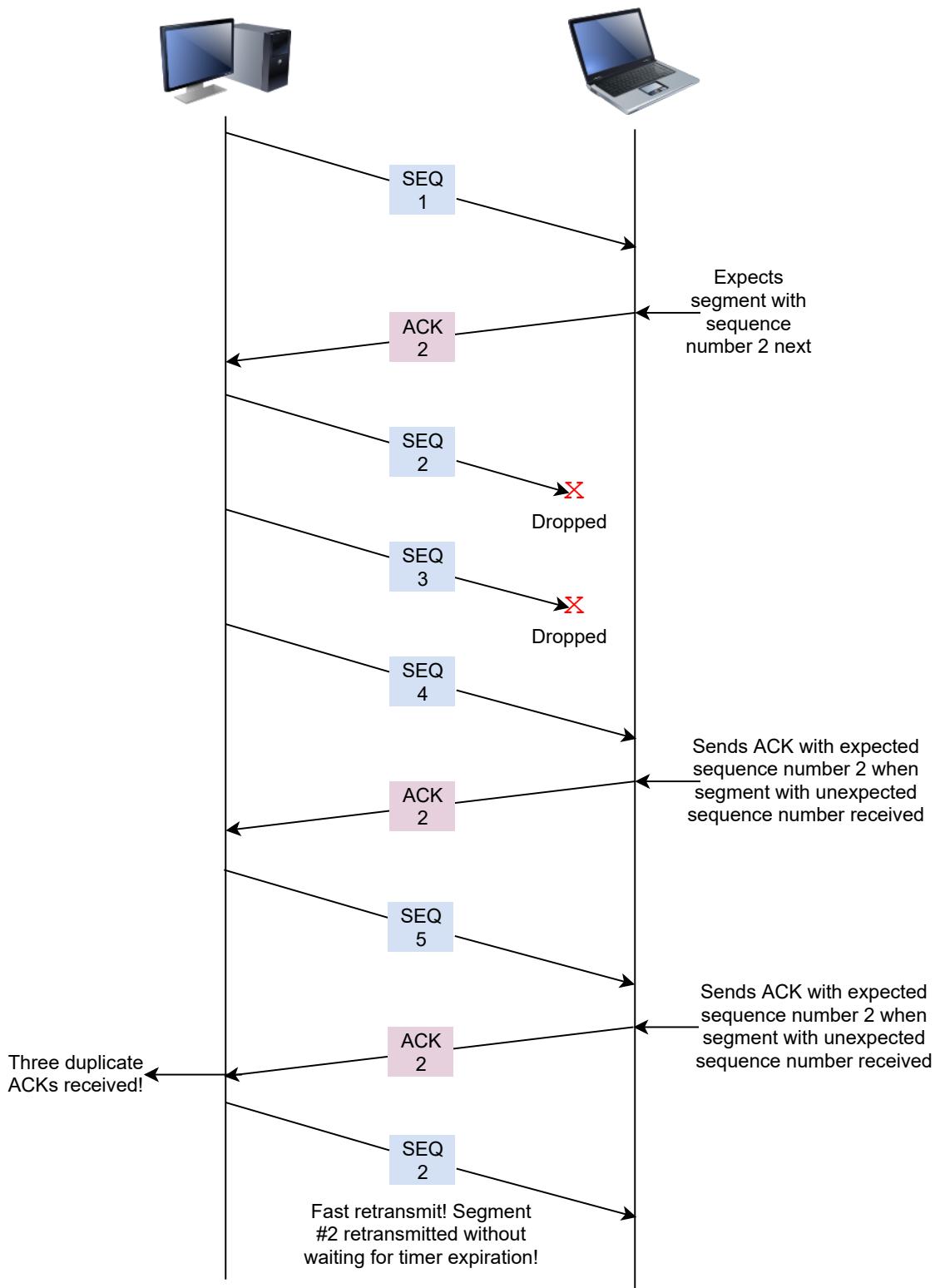


### Mild Congestion #

TCP considers that the network is lightly congested if it receives three duplicate acknowledgments.

1. The sender begins with a slow-start.
2. If 3 duplicate ACKs arrive, the sender performs a **fast retransmit** (retransmits without waiting for the retransmission timer to expire).
  - o Have a look at the following slides to see when 3 duplicate acknowledgments could arrive and when a fast retransmit happens.





2 of 2



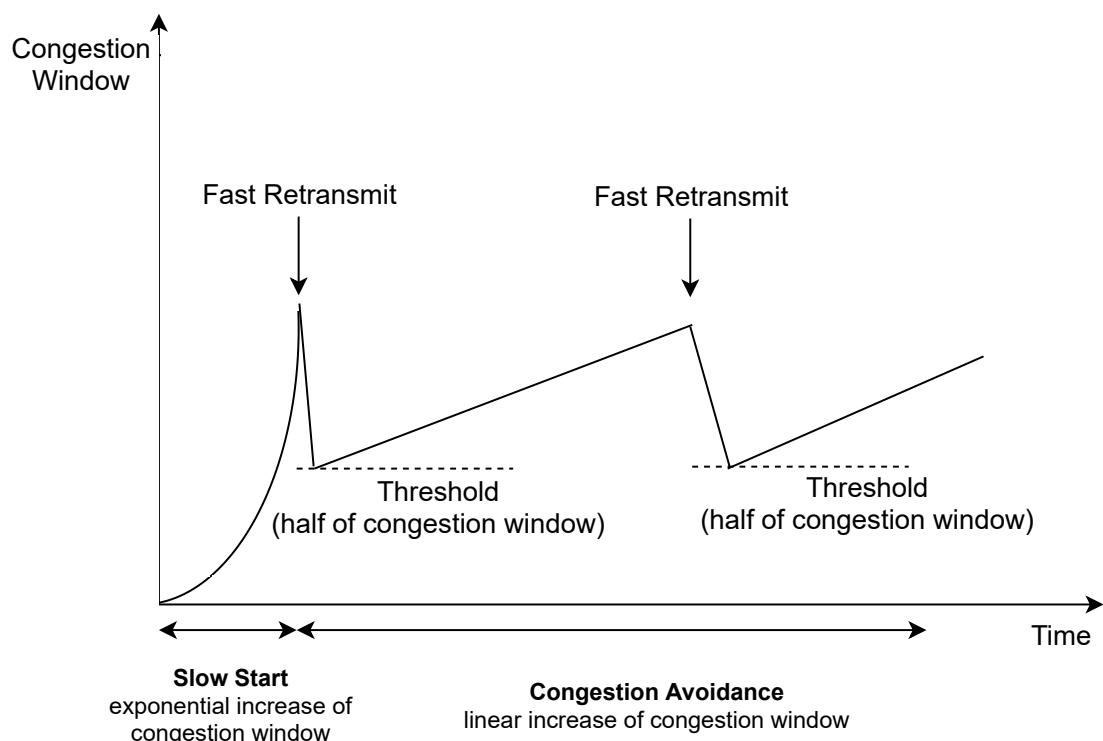
3. If the fast retransmit is successful, this implies that only one segment has been lost.

- In this case, TCP performs multiplicative decrease and the congestion window is divided by 2.

- The slow-start threshold is set to the new value of the congestion window.
4. The sender immediately enters congestion avoidance as this was mild congestion.

Figure #

The figure below illustrates the evolution of the congestion window when the network is lightly congested and all lost segments can be retransmitted using fast retransmit.



Evolution of the TCP congestion window when the network is lightly congested

## Quick Quiz! #

1

Slow start increases the congestion window size exponentially, whereas congestion avoidance increases the congestion window size linearly.

COMPLETED 0%

1 of 3



That's it for the transport layer! Let's look at socket programming in Python next!

# The Basics

Congratulations, we made it! We can finally get to writing some code.

## WE'LL COVER THE FOLLOWING



- Socket Programming
  - Types of Network Applications
  - Socket Programming in Python

## Socket Programming #

- Recall that network applications usually consist of two programs: the **server** program and the **client** program. These programs reside on two separate end systems.
- When any of these programs want to communicate with another, they write the data they want to send to their **sockets**. The underlying protocols then deliver the data to the appropriate destination.

## Types of Network Applications #

- **Standard:** applications that use well-known protocols based on meticulous standards laid down by standards documents.
  - Pros: any other developer can write applications that are compatible with standard ones.
  - Cons: some customizability will be compromised.
- **Proprietary:** applications that use protocols of the developer's own design. The source code for such applications is generally not disclosed.
  - Pros: Extremely customizable, which allows for optimizing the application for particular use cases.
  - Cons: It's incredibly difficult to write applications compatible with a

proprietary one. Also, designing protocols can lead to security

loopholes.

 **Did You Know?** Writing applications compatible with a proprietary one is doable but hard. A case in point is **Windows network file sharing**. Windows had a proprietary Server Message Block (SMB) protocol for file sharing. Only Windows clients could use Windows shared files, folders and printers. But then some hackers collected packet traces and reverse-engineered the protocol and came up with an open-source implementation named SAMBA, using which, Linux clients could access Windows file shares and even share files, folders, and printers with a Windows domain. A challenge with this approach is that when Windows updates their proprietary protocol, the open-source clients and servers don't talk to the new version of Windows and the hackers have to pull up their sleeves all over again.

## Socket Programming in Python #

Anyways, we're going to be learning the very basics of writing an application in Python. It's a great way to truly apply what you've learned.

Here's what to expect from the rest of this chapter:

1. We'll start with the very basics. You'll learn how to set up a UDP socket using the `socket` library.
2. We'll then write a complete server and a client program that runs on UDP. We'll also see them in action.
3. We'll then look at some improvements on the UDP server.
4. You'll get to do a project next! You'll be writing your very own chat app.
5. Lastly, we'll see how a few tweaks can change the program into one that runs on TCP.

---

Let's start with setting up a socket next!



# Setting up a UDP Socket

We're now going to write some basic server code in TCP. Let's get right into it.

## WE'LL COVER THE FOLLOWING ^

- Purpose of the Program
- Importing `socket`
- Creating a `socket` Object
- Binding The Socket
  - Port Number
  - Hostname

Remember that sockets are just software endpoints that processes write and read data from. They are bound to an IP address and a port. As we will see, the sending process attaches the IP address and port number of the receiving application. The IP address and port number of the sending process are also attached to the packets as headers, but that's not done manually in the code of the application itself. Networking libraries are provided with nearly all programming languages and they take responsibility for lots of plumbing.

## Purpose of the Program #

Let's set up a socket for a **UDP server program** that works like so:

1. The client will send a line of text to the server.
2. The server will receive the data and convert each character to **uppercase**.
3. The server will send the uppercase characters to the client.
4. The client will receive and display them on its screen.

We are building the code line-by-line to make it easier to understand.

Moreover, we will highlight new additions at every step.

## Importing `socket` #

The first step when writing a network application in Python is to import the `socket` library. It's generally already part of the Python bundle, so no extra library will have to be manually installed.

```
import socket
```



## Creating a `socket` Object #

We now create a **socket object** called `s` with a call to `socket.socket()`.

```
import socket  
  
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)  
print(s)
```



The syntax is as follows.

```
socket.socket(family, type, proto, fileno)
```

The syntax is shown here for completion, however, our main focus will be on explaining the **family** and **type** properties.

**1. Family.** The address family property is used to assign the type of addresses that a socket can communicate with. Only then can the addresses of that type be used with the socket. There are **three main options** available for this:

- `AF_INET`. This family is used with **IPV4** addresses. When we first introduced IP addresses, we were talking about **IPV4** addresses. IP addresses are most commonly used.
- `AF_INET6` Another address scheme, IPv6, was introduced since IPV4 is limited to about 4 billion addresses which are not sufficient, considering the exponential growth of the Internet. IPV6 provides

considering the exponential growth of the internet. IPv6 provides 340 undecillion addresses ( $340 \times 10^{36}$ ). It's slowly being adopted.

`AF_INET6` is used for IPV6 addresses. We'll introduce IPV6 more formally in a [later lesson](#).

- `AF_UNIX` This family is used for **Unix Domain Sockets (UDS)**, an interprocess communication endpoint for the same host. It's available on POSIX-compliant systems. Most operating systems today like Windows, Linux and Mac OS are POSIX compliant! So processes on a system can communicate with each other directly through this instead of having to communicate via the network.

2. **Type**. The type specifies the transport layer protocol:

- `SOCK_DGRAM` specifies that the application is to use **User Datagram Protocol** (UDP). Recall that **UDP** is less reliable but requires no initial connection establishment. We are building these server and client programs pair in UDP.
- `SOCK_STREAM` specifies that the application is to use **Transmission Control Protocol** (TCP). Recall that while TCP requires some initial setup, it's more reliable than UDP.

If you want to study the rest of the fields, have a look at the [documentation](#). Default arguments are being used for the remaining arguments, which is fine in our example.



**Note: INET socket vs UNIX socket** UNIX sockets are bound to a special file on your machine, whereas INET sockets are bound to an IP address and port. So UNIX sockets can only be accessed by the processes running on the machine and are therefore used for interprocess communication. INET sockets, on the other hand, can be accessed by remote machines.

## Binding The Socket #

```
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
port = 3000
hostname = '127.0.0.1'
```



```
hostname = "127.0.0.1"
s.bind((hostname, port))
```



Now, we bind the socket to an IP address and a port using the `bind()` function. It's given a certain n-tuple as an argument, where `n` depends on the family of the socket. In the case of `SOCK_DGRAM`, it's a tuple of the IP address and the port like the following.

```
socketObject.bind((IP address, port))
```

## Port Number #

We define the hostname and the port as variables on **lines 4 and 5**. The `port` is **3000** in this case, and as mentioned previously, the ports 0 – 1024 should be avoided as they're reserved for other system-defined processes. Binding to them may generate an error as it may already be in use by another application.

## Hostname #

The hostname is the IP address that your server will listen on. You can set it to one of three options:

1. If you're following along on your local machine, set it to `127.0.0.1` which is the `localhost` address, for IPV4. This address is called the loopback or localhost address and you should use this because you'll be writing the client code on the same machine as the server. We're doing the same here.
2. You could also set it to the empty string `''` which represents the `INADDR_ANY`. This specifies that the program intends to receive packets sent to the specified port destined for any of the IP addresses configured on that machine.
3. Lastly, you could set it to a specific IP address assigned to your machine.

---

Now that we know how to set up a basic socket, which is necessary for any kind of network application using sockets, let's write some code that is specific

to servers.

# Writing a UDP Server

In the last lesson, we'd written code to setup an IPV4 socket on TCP. Let's now get into writing a program for a basic server.

## WE'LL COVER THE FOLLOWING



- Setting a Purpose for the Server
- UDP IPV4 Socket
- Listening Infinitely
- Receiving Messages from Clients
- Capitalizing the Data
- Printing the Client's Message & Encoding
- Sending the Message Back to the Client

## Setting a Purpose for the Server #

In this lesson, we'll finish writing our server program. Our server will reply to every client's message with **a capitalized version of whatever a client program sends to it**. This is just what we're requiring of our server to do, there are other functions that such a server can perform as well.

So, in particular, the server will:

1. Print the original message received from the client.
2. Capitalize the message.
3. Send the capitalized version back to the client.

Let's code!

## UDP IPV4 Socket #

For reference, here is the code for a UDP socket.

```
import socket

# Setting up a socket
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
port = 3000
hostname = '127.0.0.1'
s.bind((hostname, port)) # Binding the socket to a port and IP address
print('Listening at {}'.format(s.getsockname())) # Printing the IP address and port of socket
```



You can use the `getsockname()` method on an object of the `socket` class to find the current IP address and port that a socket is bound to.

## Listening Infinitely #

Next, we set up a while loop (**lines 10 and 11**) so that the server listens infinitely. If the rest of this code weren't in this infinite while loop, the server would exit after dealing with one client.

```
import socket

# Setting up a socket
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
port = 3000
hostname = '127.0.0.1'
s.bind((hostname, port)) # Binding the socket to a port and IP address
print('Listening at {}'.format(s.getsockname())) # Printing the IP address and port of socket

while True:
    # The code to handle clients will go here
```

## Receiving Messages from Clients #

The server can now receive data from clients! The `recvfrom()` method (**line 12**) here accepts data of `MAX_SIZE_BYTES` length (declared on **line 3**) which is the size of one UDP datagram in bytes. This is to make sure that we receive the entirety of each packet. It also returns the IP address of the client that sent the data. We store the data and the client's IP address in the variables `data` and `clientAddress` respectively.

Note that the code stops and waits at `recvfrom()` until some data is received.

```
import socket
```



```
MAX_SIZE_BYTES = 65535 # Maximum size of a UDP datagram

# Setting up a socket
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
port = 3000
hostname = '127.0.0.1'
s.bind((hostname, port)) # Binding the socket to a port and IP address
print('Listening at {}'.format(s.getsockname())) # Printing the IP address and port of socket
while True:
    data, clientAddress = s.recvfrom(MAX_SIZE_BYTES) # Receive at most 65535 bytes at once
```

## Capitalizing the Data #

Next, we decode the message from the byte stream to ASCII (**lines 13 and 14**). Then we capitalize whatever the client sent.

```
import socket

MAX_SIZE_BYTES = 65535 # Maximum size of a UDP datagram

# Setting up a socket
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
port = 3000
hostname = '127.0.0.1'
s.bind((hostname, port)) # Binding the socket to a port and IP address
print('Listening at {}'.format(s.getsockname())) # Printing the IP address and port of socket
while True:
    data, clientAddress = s.recvfrom(MAX_SIZE_BYTES)
    message = data.decode('ascii')
    upperCaseMessage = message.upper()
```

## Printing the Client's Message & Encoding #

We print the client's IP address next since it's always a good idea after a connection has been made in order to keep track of it (**line 15**). We also print the message they sent along with it.

We then encode the capitalized ASCII message to bytes (**line 16**).

```
import socket

MAX_SIZE_BYTES = 65535 # Maximum size of a UDP datagram

# Setting up a socket
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
port = 3000
hostname = '127.0.0.1'
s.bind((hostname, port))
print('Listening at {}'.format(s.getsockname()))
while True:
    data, clientAddress = s.recvfrom(MAX_SIZE_BYTES)
    message = data.decode('ascii')
```

```
message = data.decode('ascii')
upperCaseMessage = message.upper()
print('The client at {} says {!r}'.format(clientAddress, message))
data = upperCaseMessage.encode('ascii')
```

## Sending the Message Back to the Client #

Lastly, we send the capitalized message back to the client using the `sendto()` function (**line 17**).

```
import socket

MAX_SIZE_BYTES = 65535 # Maximum size of a UDP datagram

# Setting up a socket
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
port = 3000
hostname = '127.0.0.1'
s.bind((hostname, port))
print('Listening at {}'.format(s.getsockname()))
while True:
    data, clientAddress = s.recvfrom(MAX_SIZE_BYTES)
    message = data.decode('ascii')
    upperCaseMessage = message.upper()
    print('The client at {} says {!r}'.format(clientAddress, message))
    data = upperCaseMessage.encode('ascii')
    s.sendto(data, clientAddress)
```

---

Now we have a basic server that accepts messages from clients, has a defined purpose (capitalization), and responds to the client's messages. Let's write code for a client to go with this in the next lesson.

# Writing a UDP Client Program

Let's now write a client to go with the server we wrote.

## WE'LL COVER THE FOLLOWING



- The Server
- Creating a Client Socket
- Reading Data
- Sending It to the Server
- Receiving the Server's Response
- Decoding & Printing the Capitalized Message

## The Server #

Here's the server code that we have so far for reference.

```
import socket

MAX_SIZE_BYTES = 65535 # Maximum size of a UDP datagram

# Setting up a socket
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
port = 3000
hostname = '127.0.0.1'
s.bind((hostname, port))
print('Listening at {}'.format(s.getsockname()))
while True:
    data, address = s.recvfrom(MAX_SIZE_BYTES)
    message = data.decode('ascii')
    upperCaseMessage = message.upper()
    print('The client at {} says {!r}'.format(address, message))
    data = upperCaseMessage.encode('ascii')
    s.sendto(data, address)
```



## Creating a Client Socket #

Instead of explicitly binding the socket to a given port and IP as we did previously, we can let the OS take care of it. Remember [ephemeral ports](#)? Yes

previously, we can let the OS take care of it. Remember [ephemeral ports](#)? Yes,

the OS will bind the socket to a port dynamically. So all we really need is to create a UDP socket (**line 3**).

```
import socket  
  
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
```



In fact, we can check what address and port the OS assigned to the socket using the following line of code on **line 4**:

```
import socket  
  
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)  
print('The OS assigned the address {} to me'.format(s.getsockname()))
```



You'll always get `(0.0.0.0, 0)` for now because we haven't actually used the socket. You'll get the correct answer when we use the socket to send data in the next lesson.

## Reading Data #

Remember that the goal of this client and server was for the client to send a string to the server that it would capitalize and send back? Well, we'll get that string from the user's keyboard using the `Python3` function, `input()`. The function displays whatever prompt we specify, in this case the message 'Input lowercase sentence:' and waits for the user to provide an input from the keyboard (**line 4**). The user can then type a string of their choice and hit enter. The string will be stored in the variable `message`.

Next, we encode the ASCII encoded data to bytes using the `encode()` function (**line 5**).

```
import socket
```



```
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
message = input('Input lowercase sentence: ')
data = message.encode('ascii')
```

## Sending It to the Server #

We now send the message to the server using the `sendto()` function. In addition to the data, this function takes an IP address and a port number (**line 6**). We give it the IP address `127.0.0.1` and the port `3000` which we assigned to the server previously.

Also, notice that the `getsockname()` function will give us a useful answer at this point (**line 7**).

```
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
message = input('Input lowercase sentence: ')
data = message.encode('ascii')
s.sendto(data, ('127.0.0.1', 3000))
print('The OS assigned the address {} to me'.format(s.getsockname()))
```



## Receiving the Server's Response #

We next receive the server's response and limit it to the size `MAX_SIZE_BYTES` which we saw previously.

```
import socket

MAX_SIZE_BYTES = 65535 # Maximum size of a UDP datagram

s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
message = input('Input lowercase sentence: ')
data = message.encode('ascii')
s.sendto(data, ('127.0.0.1', 3000))
print('The OS assigned the address {} to me'.format(s.getsockname()))
data, address = s.recvfrom(MAX_SIZE_BYTES)
```



## Decoding & Printing the Capitalized Message #

Lastly, we decode and print the capitalized message that the server sent us.

```
import socket

MAX_SIZE_BYTES = 65535 # Maximum size of a UDP datagram

s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
message = input('Input lowercase sentence: ')
data = message.encode('ascii')
```



```
data = message.encode('ascii')
s.sendto(data, ('127.0.0.1', 3000))
print('The OS assigned the address {} to me'.format(s.getsockname()))
data, address = s.recvfrom(MAX_SIZE_BYTEx)
text = data.decode('ascii')
print('The server {} replied with {!r}'.format(address, text))
```

---

Now we have both our server and client programs. Let's see them live in action next!

# Running The UDP Server & Client Together

We've spent the last few lessons writing code for a very basic client and a server. Let's see these in action in this lesson!

## WE'LL COVER THE FOLLOWING ^

- Connecting the Two

## Connecting the Two #

We'll run both together in one file called `udp.py` instead of running them separately. We've written some python code in the main function that allows you to specify which function you want the code to run, the `server` or the `client`.

To run the code, you would need to follow these steps:

1. Type your code and when you are ready to run the program, click on **Run**. The server code should start up automatically.
2. Open another terminal by clicking on +
3. Type the command `python3 /usercode/udp.py client` Note that it can be `server` in place of `client`.
4. Enter the text in the client window and see the effect.
5. If the program is not running to your satisfaction:
  1. Kill the running server program by typing the break sequence `ctrl+c` or `command+c` in both of the terminal windows.
  2. Change the code
  3. Click on **Run**.
  4. Type command `python3 /usercode/udp.py server` and `python3 /usercode/udp.py client` in the first and second terminal window, respectively. Go back to step 4.

Every time you make a change to the code you must click **run** for the changes to take effect.

```
import argparse, socket

MAX_SIZE_BYTEx = 65535 # Maximum size of a UDP datagram

def server(port):
    s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    hostname = '127.0.0.1'
    s.bind((hostname, port))
    print('Listening at {}'.format(s.getsockname()))
    while True:
        data, clientAddress = s.recvfrom(MAX_SIZE_BYTEx)
        message = data.decode('ascii')
        upperCaseMessage = message.upper()
        print('The client at {} says {}'.format(clientAddress, message))
        data = upperCaseMessage.encode('ascii')
        s.sendto(data, clientAddress)

def client(port):
    s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    message = input('Input lowercase sentence: ')
    data = message.encode('ascii')
    s.sendto(data, ('127.0.0.1', port))
    print('The OS assigned the address {} to me'.format(s.getsockname()))
    data, address = s.recvfrom(MAX_SIZE_BYTEx)
    text = data.decode('ascii')
    print('The server {} replied with {}'.format(address, text))

if __name__ == '__main__':
    funcs = {'client': client, 'server': server}
    parser = argparse.ArgumentParser(description='UDP client and server')
    parser.add_argument('functions', choices=funcs, help='client or server')
    parser.add_argument('-p', metavar='PORT', type=int, default=3000,
                       help='UDP port (default 3000)')
    args = parser.parse_args()
    function = funcs[args.functions]
    function(args.p)
```

---

In the next lesson, we're going to look at some possible improvements to our current UDP program.

# Improvements to UDP Programs: Avoiding Arbitrary Servers

There are a few improvements that can easily be made to our UDP program. Let's have a look.

## WE'LL COVER THE FOLLOWING



- Problem: Replies From Arbitrary Servers
  - Fix with `connect()`
  - Disadvantages
  - Fix with Address Matching
- Quick Quiz!

## Problem: Replies From Arbitrary Servers #

Note that at the moment, our UDP client accepts replies from *any* machine and assumes that it's the one that it sent the initial message to, evident in the following line,

```
data, address = s.recvfrom(MAX_SIZE_BYTES)
```

Note how the client does not check **who** it is receiving the message from. It just receives a message.

## Fix with `connect()` #

There are two quick ways to go about fixing this. The first of which is to use the `connect()` method to forbid other addresses from sending packets to the client.

```
import socket  
  
MAX_SIZE_BYTES = 65535 # Maximum size of a UDP datagram  
  
def client(port):
```



```
host = '127.0.0.1'
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.connect((host, port))
message = input('Input lowercase sentence: ')
data = message.encode('ascii')
s.send(data)
print('The OS assigned the address {} to me'.format(s.getsockname()))
data = s.recv(MAX_SIZE_BYTES)
text = data.decode('ascii')
print('The server replied with {!r}'.format(text))
```

With the `sendto()` method, we had to specify the IP address and port of the server every time the client wanted to send a message. However, with the `connect()` method we used, we just use `send()` and `recv()` without passing any arguments about which address to send to because the program **knows** that.

This also means that no server other than the one the client *connected* to can send it messages. The operating system discards any of those messages by default.

## Disadvantages #

The main disadvantage of this method is that the **client can only be connected to one server at a time**. In most real life scenarios, singular applications connect to *multiple servers*!

## Fix with Address Matching #

A better, though more tedious approach, to handle multiple servers would be to **check the return address of each reply against a list of addresses that replies are expected from**.

Let's implement it!

```
import socket
MAX_SIZE_BYTES = 65535 # Maximum size of a UDP datagram

def client(port):
    s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    hosts = []
    while True:
        host = input('Input host address: ')
        hosts.append((host, port))
        message = input('Input message to send to server: ')
        data = message.encode('ascii')
        s.sendto(data, (host, port))
        print('The OS assigned the address {} to me'.format(s.getsockname()))
```

```
print('The OS assigned the address {} to me'.format(s.getsockname()))
data, address = s.recvfrom(MAX_SIZE_BYTEx)
text = data.decode('ascii')

if(address in hosts):
    print('The server {} replied with {!r}'.format(address, text))
    hosts.remove(address)
else:
    print('message {!r} from unexpected host {}!'.format(text, address))
```

As you can see, we created a list called `hosts` which contains tuples like (*IP addresses, port numbers*) of any host that the client connects to. Upon receiving every message, it checks whether the message is from a host it expects to receive a reply from. As soon as a reply is received, it removes the host from the list.

## Quick Quiz! #

1

Why do we remove the host's address from `hosts` once a reply is received?

COMPLETED 0%

1 of 2



In the next lesson, you're going to try out an exercise for yourself: write a chat app in UDP!

# Project: Write a UDP Chat App!

Welcome to your UDP project in this course!

## WE'LL COVER THE FOLLOWING ^

- Instructions

## Instructions #

Writing a chat app is not so different from the capitalization code we saw in the last lesson. The idea is very simple. The client sends a message to the server and the server should respond with one. Both messages should be taken as input from the user. We've given you some basic starter code for it.

Here are some other factors you would want to consider to write your app:

1. Your client and server both need to stay alive and not exit after each message sent.
2. Both the client and the server need to print every message received from the other party.
3. The server should not be chatting with more than one client.

```
import argparse, socket

MAX_SIZE_BYTES = 65535 # Maximum size of a UDP datagram

def server(port):
    pass
    # Your code goes here

def client(port):
    pass
    # Your code goes here

if __name__ == '__main__':
    funcs = {'client': client, 'server': server}
    parser = argparse.ArgumentParser(description='UDP client and server')
    parser.add_argument('functions', choices=funcs, help='client or server')
    parser.add_argument('-p', metavar='PORT', type=int, default=3000,
```

```
parser.add_argument('--port', type=int, default=3000, help='UDP port (default 3000)')  
args = parser.parse_args()  
  
function = funcs[args.functions]  
function(args.p)
```

Note that to run the code, you would need to follow these steps:

1. Type your code and when you are ready to run the program, click on **Run**. The server code should start up automatically.
2. Open another terminal by clicking on +
3. Type the command `python3 /usercode/udp.py client` Note that it can be `server` in place of `client`.
4. Enter the text in the client window and see the effect.
5. If the program is not running to your satisfaction:
  1. Kill the running server program by typing the break sequence `ctrl+c` or `command+c` in both of the terminal windows.
  2. Change the code.
  3. Click on **Run**.
  4. Type command `python3 /usercode/udp.py server` and `python3 /usercode/udp.py client` in the first and second terminal window, respectively. Go back to step 4.

Every time you make a change to the code you must click **run** for the changes to take effect

In the next lesson, we'll look at the solution to this project.

# Solution: Write a UDP Chat App!

WE'LL COVER THE FOLLOWING ^

- Client
- Server

## Client #

The client program uses a `while` loop to keep the conversation with the server alive. Furthermore, it uses `connect()` to ensure that only one server is connected to, and only replies from that server are received.

```
import argparse, socket

MAX_SIZE_BYTES = 65535 # Maximum size of a UDP datagram

def client(port):
    s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    host = '127.0.0.1'
    while True:
        s.connect((host, port))
        message = input('Input message to send to server: ')
        data = message.encode('ascii')
        s.send(data)
        data = s.recv(MAX_SIZE_BYTES)
        text = data.decode('ascii')
        print('The server replied with {!r}'.format(text))
```

## Server #

```
import argparse, socket

MAX_SIZE_BYTES = 65535 # Maximum size of a UDP datagram

def server(port):
    s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    hostname = '127.0.0.1'
    s.bind((hostname, port))
    print('Listening at {}'.format(s.getsockname()))
```

```
while True:  
    data, clientAddress = s.recvfrom(MAX_SIZE_BYTES)  
    message = data.decode('ascii')  
  
    print('The client at {} says {!r}'.format(clientAddress, message))  
    msg_to_send = input('Input message to send to client: ')  
    data = msg_to_send.encode('ascii')  
    s.sendto(data, clientAddress)
```

---

Great! Let's look at how server and client programs can be written to run on TCP in Python3 in the next lesson!

# A TCP Client-Server Program

In the last chapter, we studied TCP theory in detail. Now, we'll look at how we can code up TCP programs in Python.

## WE'LL COVER THE FOLLOWING



- Introduction
- A TCP Server & Client Program
  - Handling Fragmentation
    - `sendall()`
    - `recvall()`

## Introduction #

There are a few key points to be noted about TCP programs:

- TCP `connect()` calls induce full TCP three-way handshakes! As we saw in [the last chapter](#), **three-way handshakes can fail and so can `connect()` calls.**
- **A new socket gets created for every new connection** in a TCP architecture.
- **One socket** on TCP servers is dedicated to **continuously listen for new incoming connections**.
- When a connection is successful, that listening socket creates a **new socket exclusively for that connection**.
- When the **connection terminates**, the associated **socket** gets **deleted**.
- Every socket, and hence **each connection**, is identified by the **unique 4-tuple: (`local_ip`, `local_port`, `remote_ip`, `remote_port`)**. All incoming TCP packets are examined to see whether their source and destination

addresses belong to any such currently connected sockets.

- Unlike UDP, **TCP segments will be delivered as long as the sender and receiver are connected by a path and they are both live.**
- A sending TCP entity might **split TCP segments into packets** and so, receiving TCP entities would have to reassemble them. This is unlikely in our small program but happens all the time in the real world. So we need to take care of when there is data leftover in the buffer to send or to receive after one call.

## A TCP Server & Client Program #

```
import argparse, socket

def recvall(sock, length):
    data = b''
    while len(data) < length:
        more = sock.recv(length - len(data))
        if not more:
            raise EOFError('was expecting %d bytes but only received'
                           ' %d bytes before the socket closed'
                           % (length, len(data)))
        data += more
    return data

def server(port):
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    sock.bind(('127.0.0.1', port))
    sock.listen(1)
    print('Listening at', sock.getsockname())
    while True:
        print('Waiting for a new connection')
        sc, sockname = sock.accept()
        print('Connection from', sockname)
        print('  Socket name:', sc.getsockname())
        print('  Socket peer:', sc.getpeername())
        message = recvall(sc, 16)
        print('  message from client:', repr(message))
        sc.sendall(b'Goodbye, client!')
        sc.close()
        print('  Closing socket')

def client(port):
    host = '127.0.0.1'
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.connect((host, port))
    print('Client has been assigned the socket: ', sock.getsockname())
    sock.sendall(b'Greetings, server')
    reply = recvall(sock, 16)
    print('Server: ', repr(reply))
    sock.close()
```

```
if __name__ == '__main__':
    choices = {'client': client, 'server': server}
    parser = argparse.ArgumentParser(description='Send and receive over TCP')
    parser.add_argument('role', choices=choices, help='which role to play')
    parser.add_argument('-p', metavar='PORT', type=int, default=3000, help='TCP port (default 3000)')
    args = parser.parse_args()
    function = choices[args.role]
    function(args.p)
```

As you can see, the client program is pretty much the same as a UDP client program. There are a few key differences which we will explore now:

## Handling Fragmentation #

```
sendall() #
```

One of three things may happen at every `send()` call:

1. All the data you passed to it gets sent immediately.
2. None of the data gets transmitted.
3. *Part* of the data gets transmitted.

The `send()` function returns the length of the number of bytes it successfully transmitted, which can be used to check if the entire segment was sent.

Here's what code to handle partial or no transmission would look like:

```
bytes_sent = 0 # No bytes initially sent
while bytes_sent < len(message): # If number of bytes sent is less than the amount of data
    message_left = message[bytes_sent:] # Indexing and storing the part of the message remaining
    bytes_sent += sock.send(message_left) # Sending remaining message
```

Luckily, Python has its own implementation of this in a function called `sendall()`. It ensures that all of the data gets sent. Check out line 37 in the TCP server-client program above to see how it is used.

```
recvall() #
```

Unfortunately, no equivalent to automatically handle fragmentation exists for the receiving end. Hence, we'd have to cater for the cases when:

- Part of the sent data arrives
- None of the sent data arrives

We do this by defining a function called `recvall()` on line 3.

---

That's the end of this chapter. In the next one, we'll start on the network layer.

# What Is the Network Layer?

In this lesson, we'll get a quick introduction to the network layer.

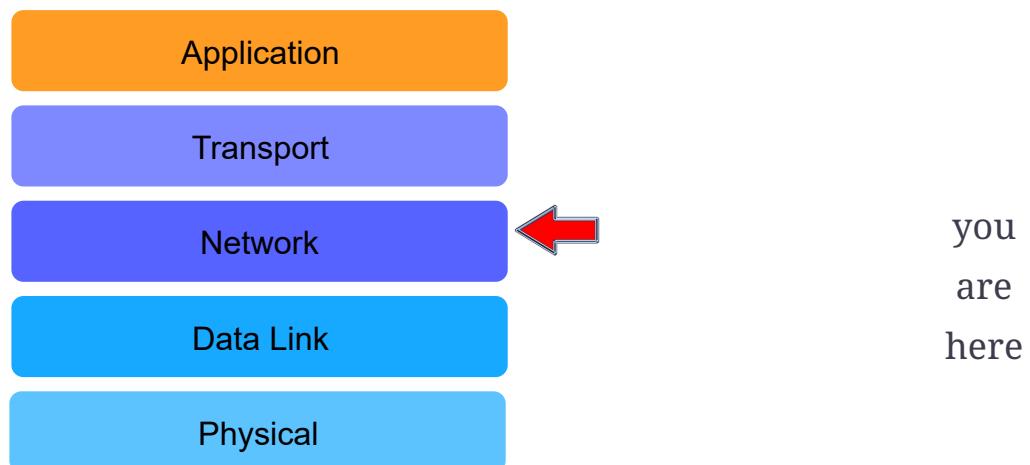
## WE'LL COVER THE FOLLOWING

- You Are Here!
- Main Objectives & Key Responsibilities
  - Limitations Of The Underlying Data Link Layer
  - Principles of the Network Layer
  - Network Layer Services
- Quick Quiz!



## You Are Here! #

Let's zoom out and have a look at the big picture.



## Main Objectives & Key Responsibilities #

The main objective of the network layer is to allow end systems to exchange information through intermediate systems called **routers**. The unit of information in the network layer is called a **packet**.

## Limitations Of The Underlying Data Link Layer #

Messages at the data link layer are called **frames**. There are more than a dozen different types of data link layers.

1. Every data link layer technology has a limit on maximum frame size.
2. Most of them use a different maximum frame size.
3. Furthermore, each interface on an end system in the data link layer has a link layer address. This means the link layer has to have an addressing system of its own.

The network layer must cope with this heterogeneity of the data link layer.

## Principles of the Network Layer #

The network layer relies on the following principles:

1. Each network layer entity is identified by a **network layer address**. This address is independent of the data link layer addresses that the entity may use.
2. The service provided by the network layer **does not depend on** the service or the internal organization of **the underlying data link layers**. This independence ensures:
  - **Adaptability.** The network layer can be used by hosts attached to different types of data link layers.
  - **Independent Evolution.** The data link layers and the network layer evolve independently from each other.
  - **Forward Compatibility.** The network layer can be easily adapted to new data link layers when a new type is invented.
3. The network layer is conceptually divided into **two planes**:
  1. The **data plane**. The data plane contains the protocols and mechanisms that allow *hosts and routers to exchange packets carrying user data*.
  2. The **control plane**. The control plane contains the protocols and mechanisms that *enable routers to efficiently learn how to forward packets towards their final destination*.

## Network Layer Services #

There are two types of services that can be provided by the network layer:

- An unreliable connectionless service. This kind of service does not ensure message delivery and involves no established connections.
- A connection-oriented, reliable or unreliable, service. This kind of service establishes connections and may or may not ensure that messages are delivered.

Nowadays, most networks use an unreliable connectionless service at the network layer. This is our main focus in this chapter.

## Quick Quiz! #

1

What is the unit of information in the network layer called?

COMPLETED 0%

1 of 3



In the next lesson, we'll look at the two most common ways that the network layer is organized.

# Organization of the Network layer

We'll study the two internal organizations of the network layer in this lesson!

## WE'LL COVER THE FOLLOWING



- Datagram Organization
  - Forwarding Tables
- Virtual Circuit Organization
- Virtual Circuit Organization vs. Datagram Organization
  - Advantages of Datagram Organization
  - Advantages of The Virtual Circuit Organization
- Quick Quiz!

There are two possible internal organizations of the network layer: **datagram** and **virtual circuits**.

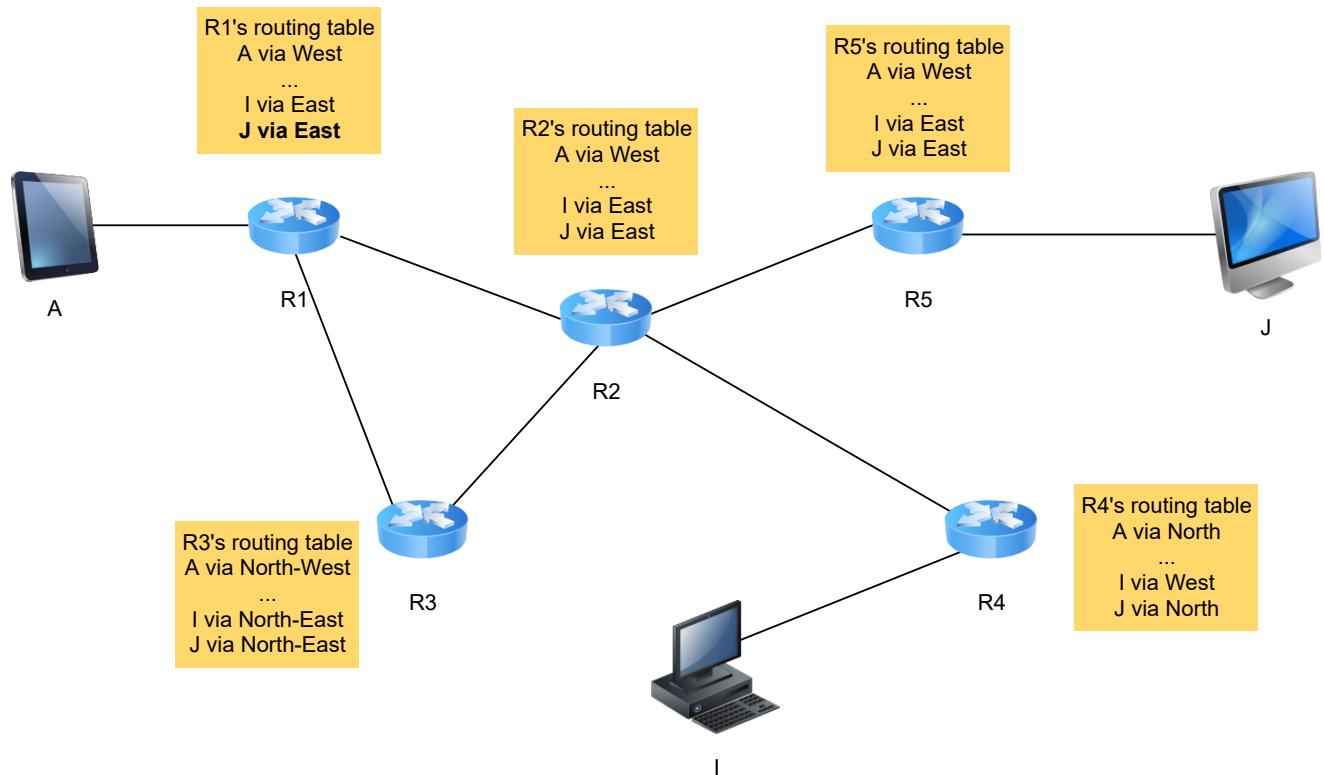
## Datagram Organization #

The datagram organization has been very popular in computer networks. Datagram-based network layers include **IPv4** and **IPv6** in the **global Internet**, CLNP defined by the ISO, IPX defined by Novell or XNS defined by Xerox.

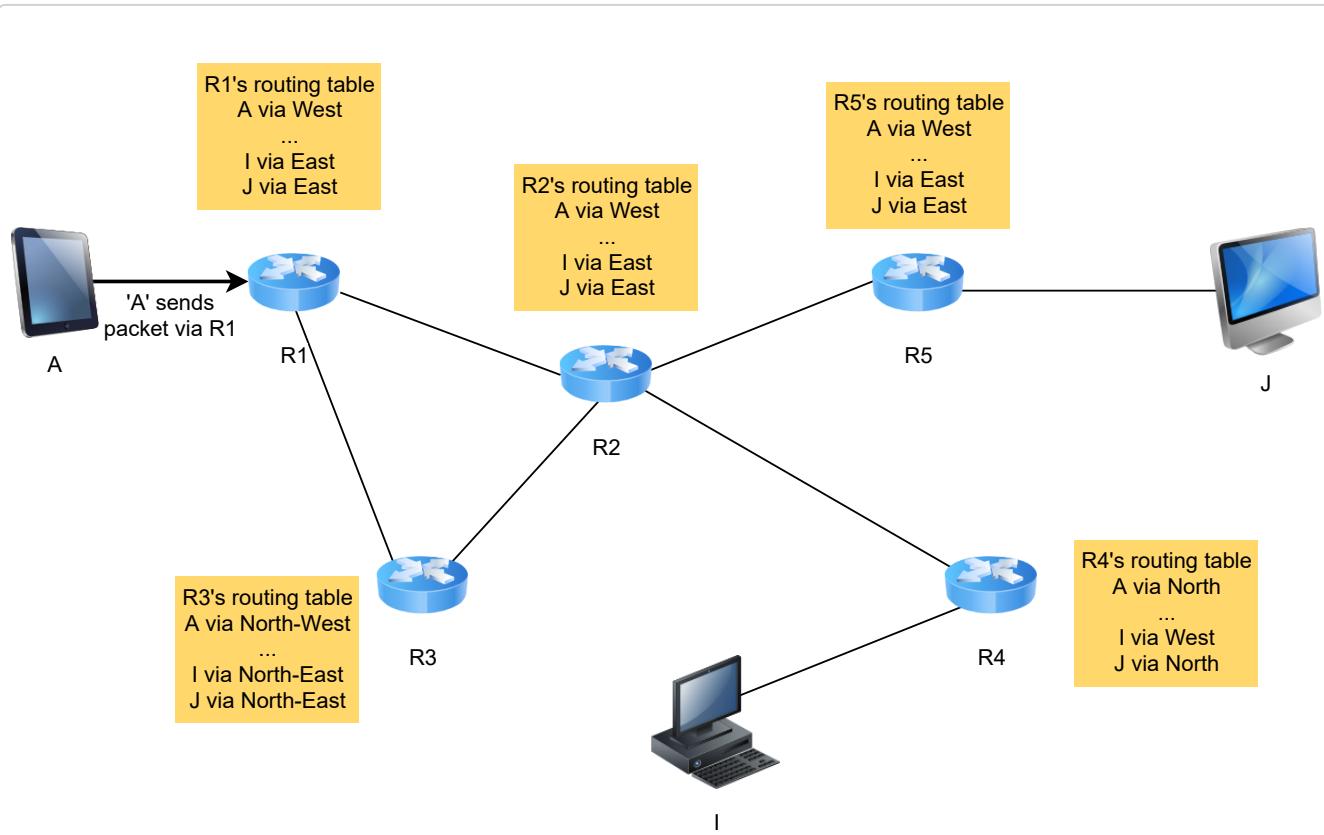
This organization is **connectionless** and hence each packet contains:

- The network layer address of the destination host.
- The network layer address of the sender.
- The information to be sent.

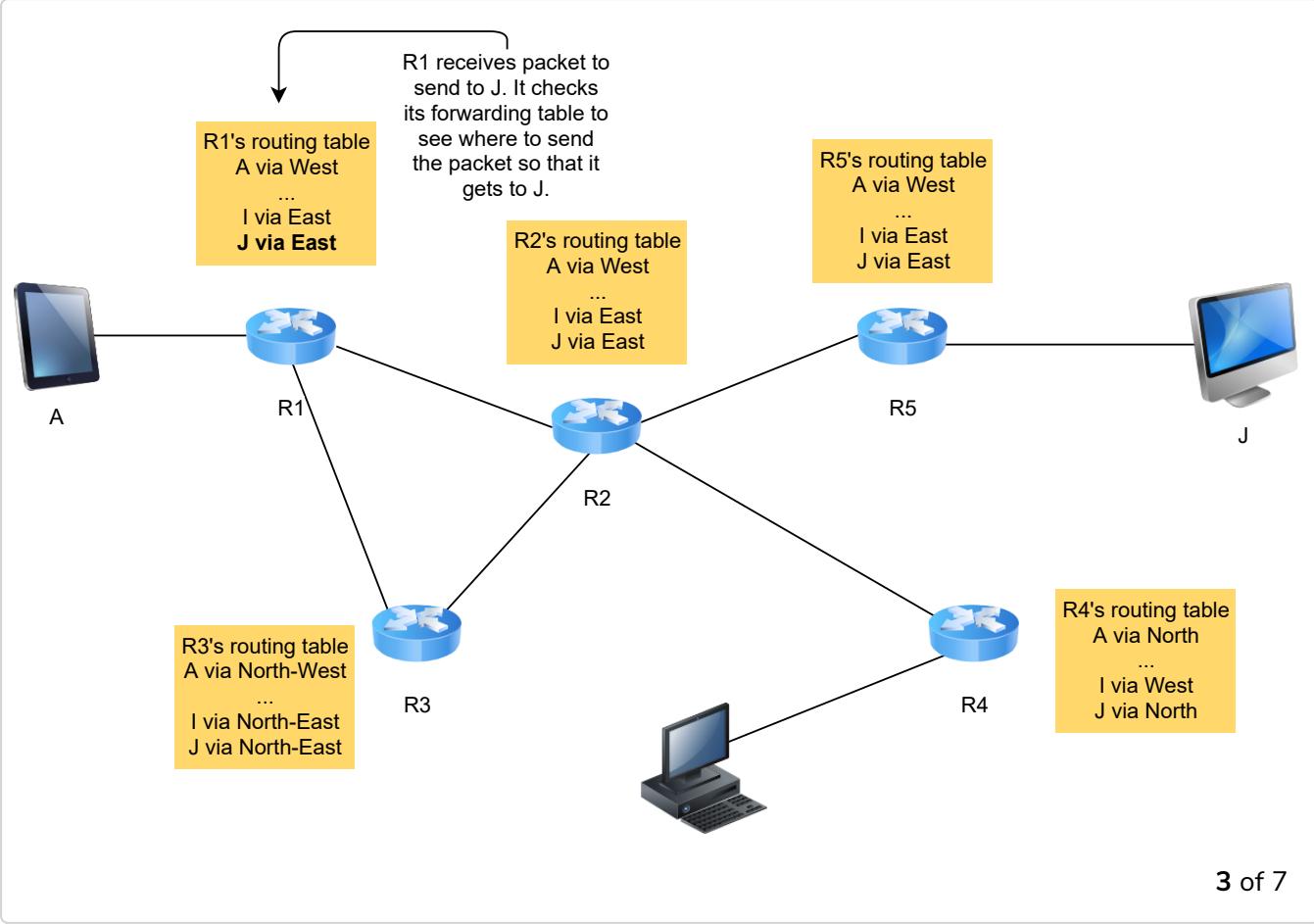
To understand the datagram organization, let's consider the slides below. A network layer address represented by a letter, has been assigned to each host and router. **Host A** wishes to send some information to **host J**.



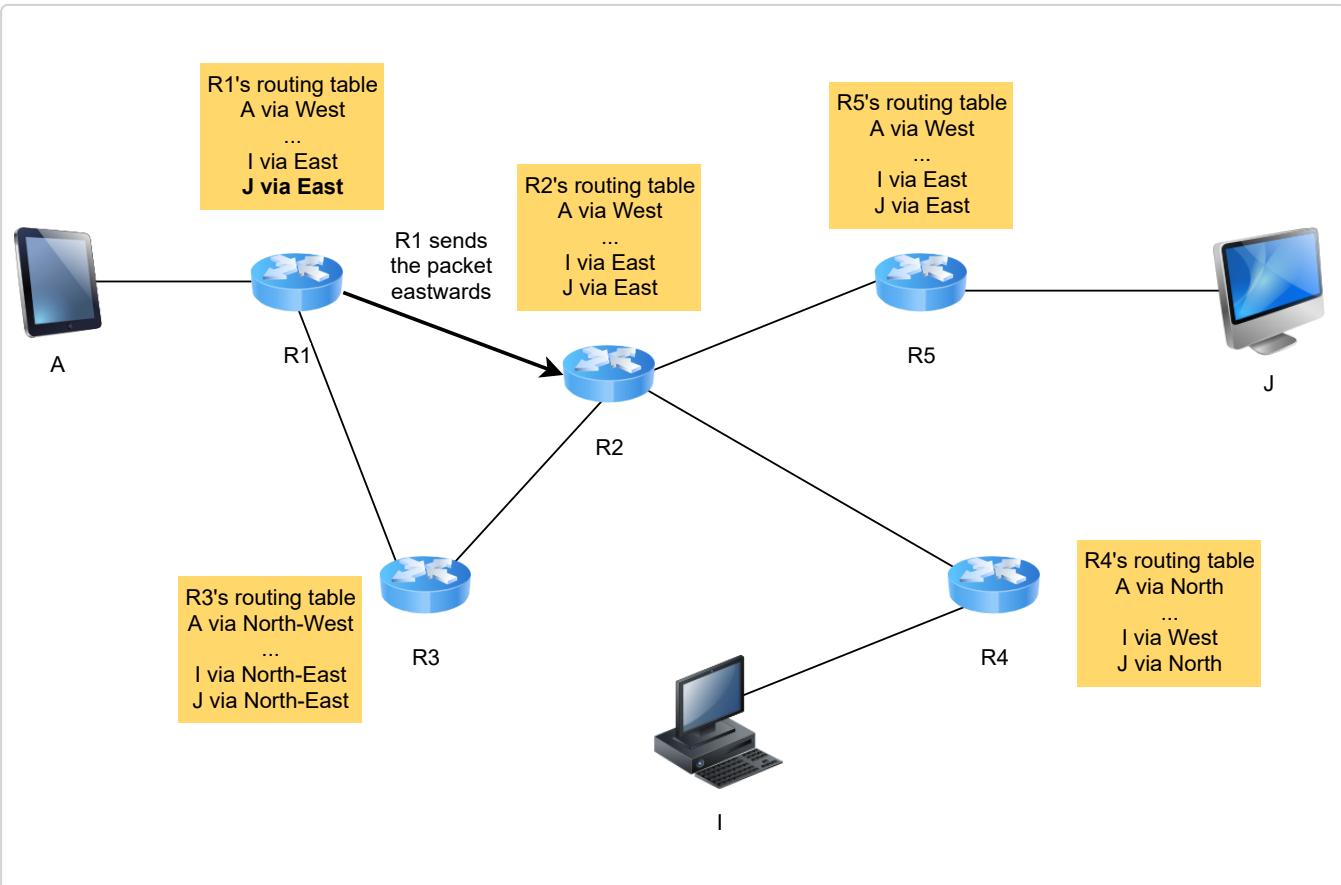
1 of 7



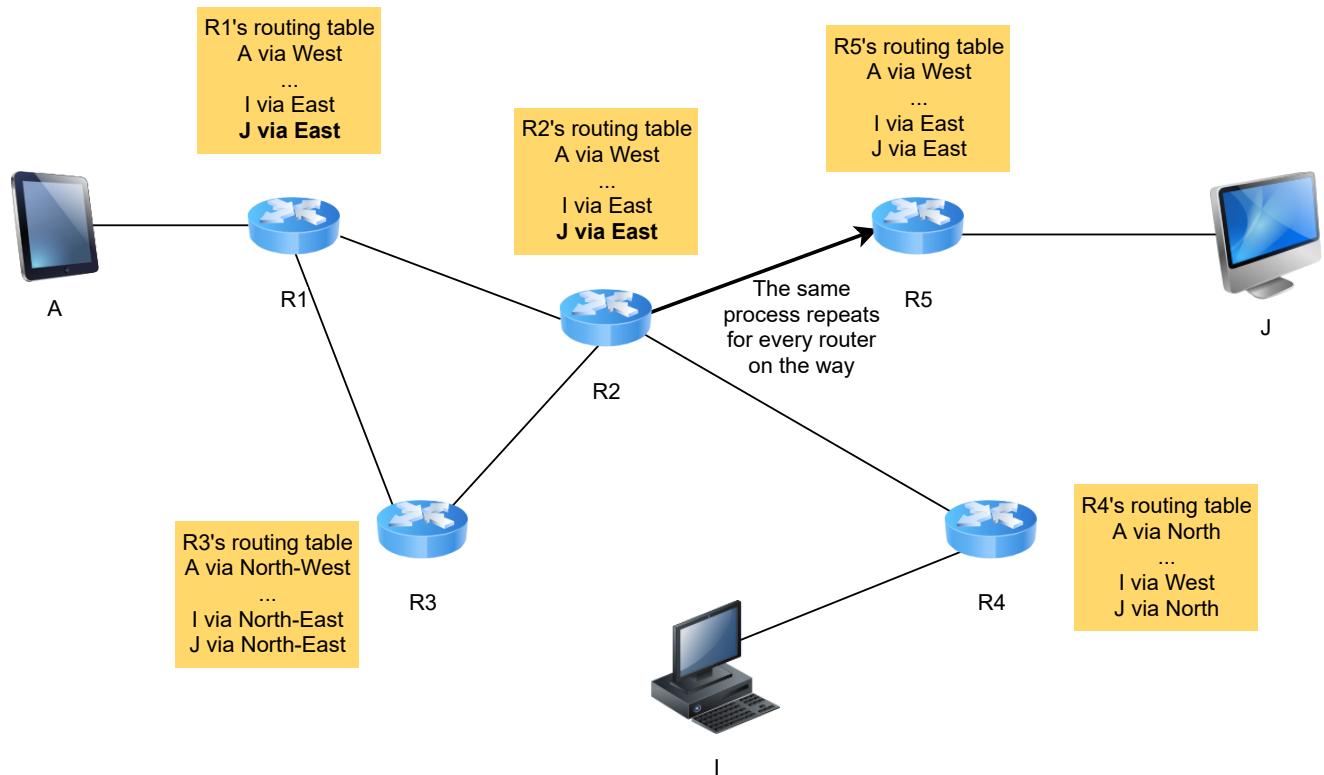
2 of 7



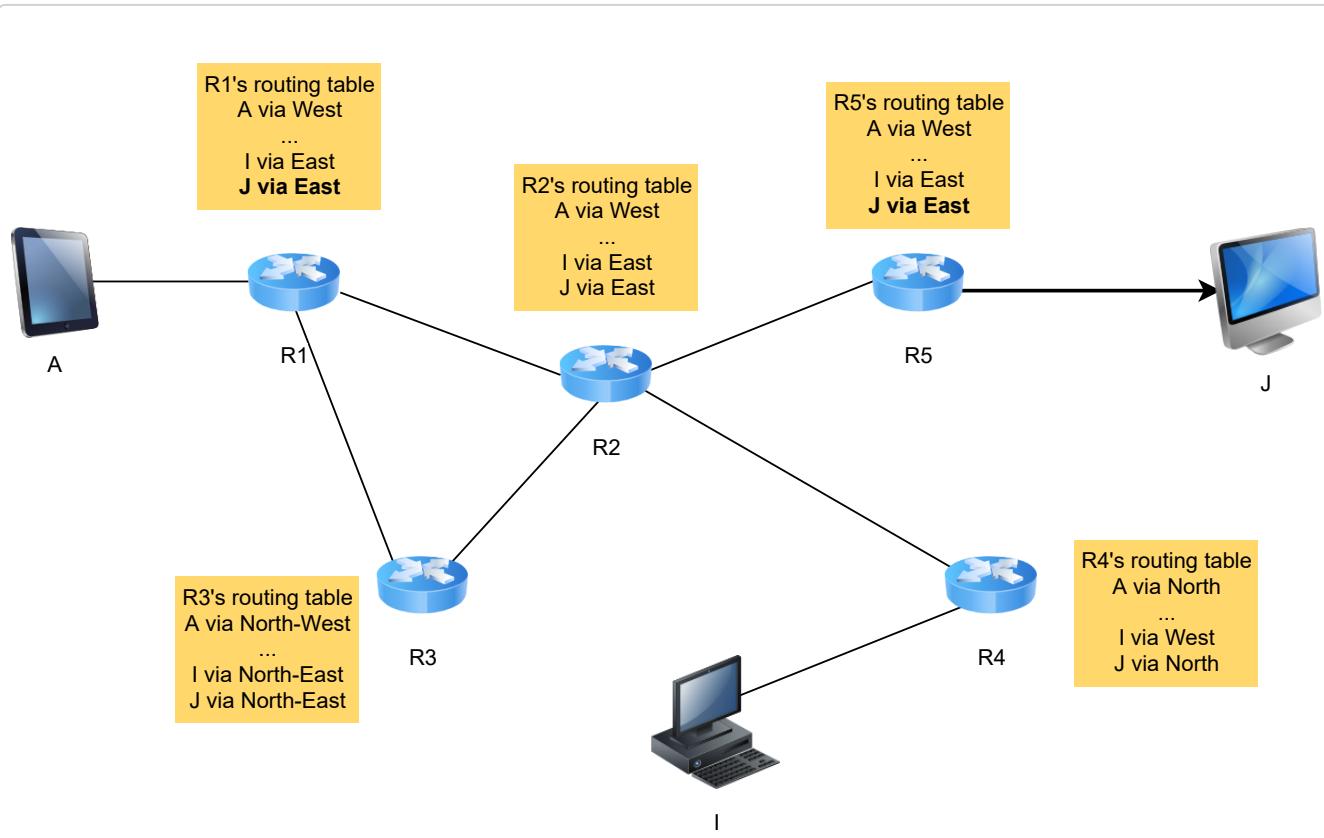
3 of 7



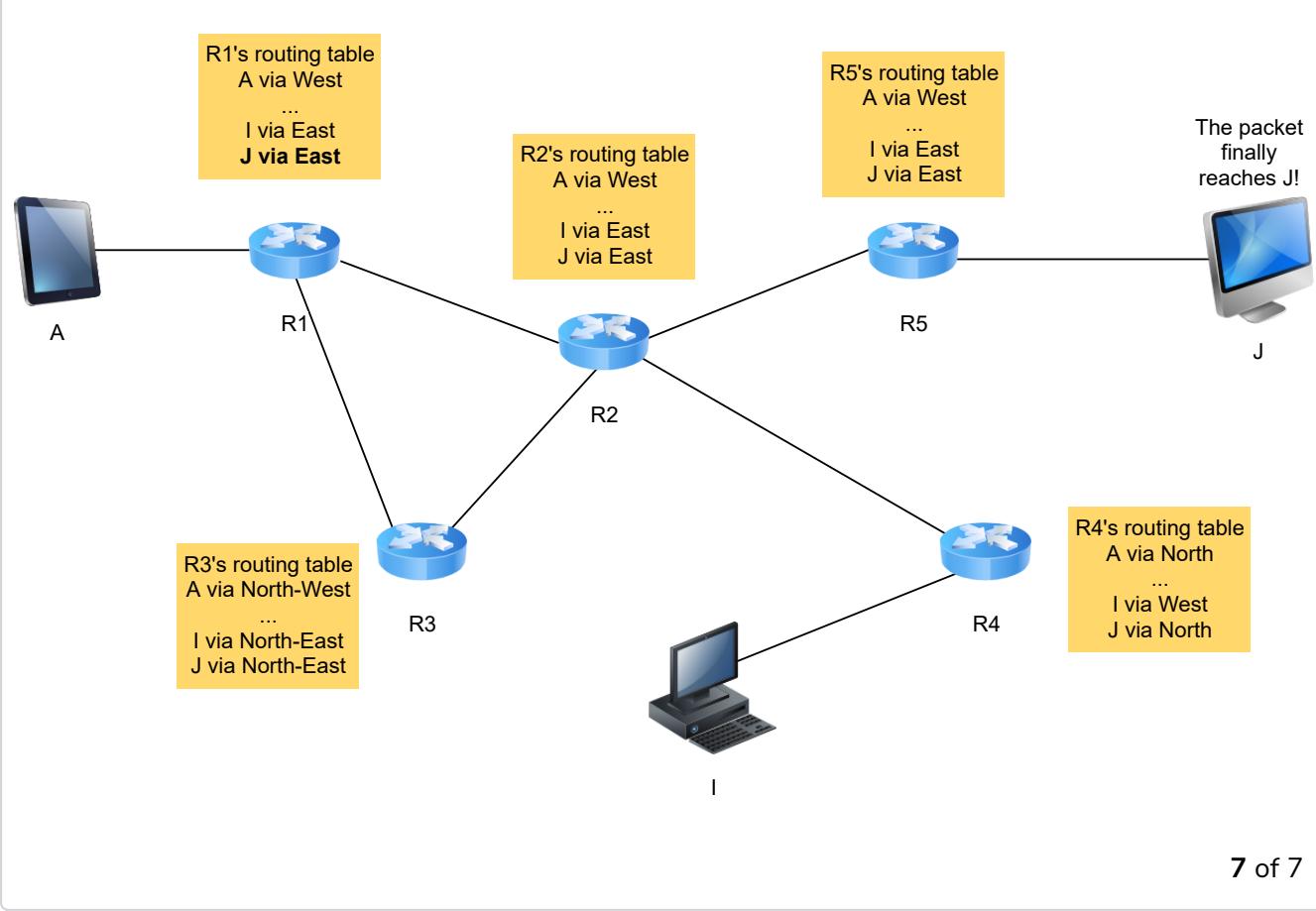
4 of 7



5 of 7



6 of 7



7 of 7



## Forwarding Tables #

Routers use *hop-by-hop* forwarding in the datagram organization. This means that when a router receives a packet that is not destined to itself, it looks up the destination address of the packet in its **forwarding table**.

A **forwarding table** is a data structure that maps each destination address to the device. Then, a packet must be forwarded for it to reach its final destination.

forwarding tables must

- Allow any host in the network to reach any other host. This implies that **each router must know a route towards each destination**.
- The paths composed from the information stored in the forwarding tables **must not contain loops**. Otherwise, some destinations would be

The **data plane** contains all the protocols and algorithms that are used by hosts and routers to create and process the packets that contain user data.

The **control plane** contains all the protocols and mechanisms that are used to compute, install, and maintain forwarding tables on the routers.



**Note: Routing Tables vs. Forwarding Tables** Routing tables are generally used to *generate* the information for a forwarding table, which is a subset of the routing table. So, a routing table may have 3 paths for one source, and destination pair generated from a few different algorithms that's perhaps also entered manually. The forwarding table, however, will only have one of those entries which is usually the preferred one based on another algorithm or criteria. The forwarding table is usually optimized for storage and lookup.

## Virtual Circuit Organization #

The second organization of the network layer, called **virtual circuits**, has been *inspired by the organization of telephone networks*.

- Telephone networks have been designed to carry phone calls that usually last a few minutes.
- Each phone is **identified by a telephone number** and is attached to a **telephone switch**.
- To initiate a phone call, a telephone first needs to send the destination's phone number to its local switch.
- The switch cooperates with the other switches in the network to create a bi-directional channel between the two telephones through the network.
- This channel will be used by the two telephones during the lifetime of the call and will be released at the end of the call.
- Until the 1960s, most of these channels were **created manually**, by telephone operators, upon request of the caller.

- Today's telephone networks use automated switches and allow several channels to be carried over **the same physical link**, but the principles remain roughly the same.

In a network using virtual circuits, all hosts are **identified with a network layer address**. However, a host must explicitly request the establishment of a virtual circuit before being able to send packets to a destination host. The request to establish a virtual circuit is processed by the control plane, which installs state to create the virtual circuit between the source and the destination through intermediate routers.

This organization is **connection-oriented** which means that resources like buffers, CPU, and bandwidth are reserved for every connection. The first packet sent reserves these resources for subsequent packets, which all follow a single path for the duration of the connection.

The virtual circuit organization has been mainly used in public networks, starting from X.25, and then Frame Relay and Asynchronous Transfer Mode (ATM) network.

## Virtual Circuit Organization vs. Datagram Organization #

Both the datagram and virtual circuit organizations have advantages and drawbacks.

### Advantages of Datagram Organization #

The main advantage of the datagram organization is that **hosts can easily send packets to any number of destinations**, while the virtual circuit organization requires the establishment of a virtual circuit before the transmission of a data packet. This can cause high overhead for hosts that exchange small amounts of data.

Another advantage of the datagram-based network layer is that **it's resilient**. If a virtual or physical circuit breaks, it has to go through the connection establishment phase, again. In case of datagram-based network layer, **each packet can be routed independently** of each other, hop-by-hop, so

intermediate routers can divert around failures.

## Advantages of The Virtual Circuit Organization #

On the other hand, the main advantage of the virtual circuit organization is that the **forwarding algorithm used by routers is simpler** than when using the datagram organization. Furthermore, the utilization of virtual circuits may allow the **load to be better spread through the network**.

Also, since the packets follow a particular dedicated path, they **reach the destination in the order they were sent**. Virtual circuits can be configured to provide a variety of services including best effort, in which case *some* packets may be dropped. However, in case of bursty traffic, there is a possibility of packet drops.

Note that the Internet uses a **datagram organization**. We'll be focusing on that for the rest of the chapter.

## Quick Quiz! #

1

Which of the following is not a network layer principle?

---

In the next lesson, we'll look at how the control plane's routing algorithms work!

# The Control Plane: Static & Dynamic Routing

In this section, we discuss the three main techniques that can be used to maintain the routing tables in a network.

## WE'LL COVER THE FOLLOWING ^

- Modeling the Network as a Graph
- Static Routing
  - Disadvantages of Static Routing
- Dynamic Routing Algorithms
  - Distance Vector Routing
  - Link-State Routing
- Quick Quiz!

The main purpose of the *control plane* is to maintain and build routing tables. This is done via a number of algorithms and protocols which we will discuss here.



**Note: Routing Algorithms vs. Routing Protocols.** The software/hardware *implementation* of routing algorithms are known as routing protocols!

## Modeling the Network as a Graph #

A network can be modeled as a **directed weighted graph**. Each router is a node, and the links between routers are the edges in the graph. A positive weight is associated with each directed edge. Routers use the shortest path to reach each destination. In practice, different types of weights can be associated with each directed edge:

- **Unit weight.** If all links have a unit weight, shortest path routing prefers the path with the fewest number of links.

the paths with the least number of intermediate routers.

- **Weight proportional to the propagation delay.** If all link weights are configured this way, shortest path routing algorithms will prefer the paths with the smallest propagation delay.
- **Weight proportional to the available bandwidth.**  $weight = \frac{C}{bandwidth}$  where  $C$  is a constant larger than the highest link bandwidth in the network. If all link weights are configured this way, shortest path routing algorithms will prefer paths with higher bandwidth.

Usually, the same weight is assigned to the two edges that correspond to a physical link (i.e. R1→R2 and R2→R1). However, it's not necessary and some asymmetric links may have different bandwidths upstream and downstream.

## Static Routing #

Manually computed routes are manually added to the routing table. This is useful if there are a few outgoing links from your network. It gets difficult when you have rich connectivity (in terms of the number of links to other networks). It also does not automatically adapt to changes – addition or removal of links or routers.

### Disadvantages of Static Routing #

1. The main drawback of static routing is that it **doesn't adapt to the evolution of the network** and hence doesn't scale well. When a new route or link is added, all routing tables must be recomputed.
2. Furthermore, when a link or router fails, the routing tables must be updated as well.

## Dynamic Routing Algorithms #

Unlike static routing algorithms, dynamic ones adapt routing tables with changes in the network. There are two main classes of dynamic routing algorithms: **distance vector** and **link-state routing algorithms**.

### Distance Vector Routing #

**Distance vector** is a simple distributed routing protocol. Distance vector routing allows routers to discover the destinations reachable inside the

network as well as the shortest path to reach each of these destinations. The shortest path is computed based on the **cost** that is associated with each link.

## Link-State Routing #

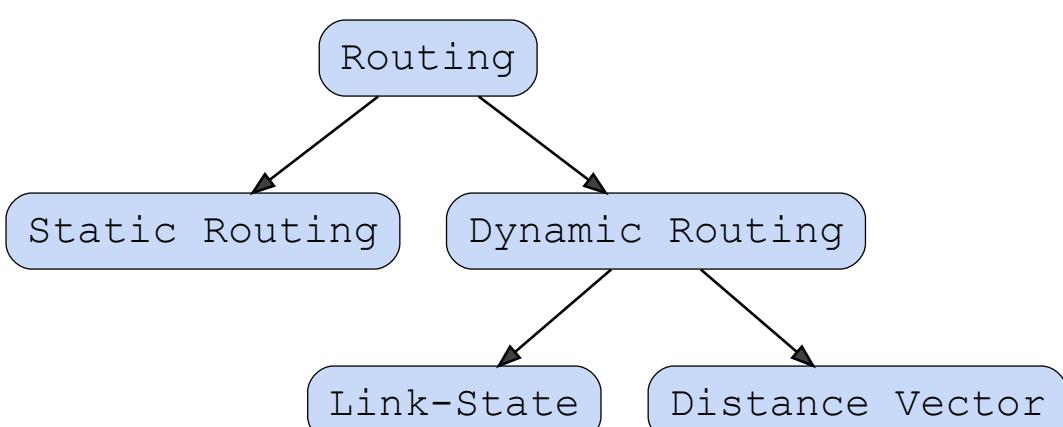
Another way to create a routing table with the most efficient path between two routers or ‘nodes’ is by using link-state routing.

Link-state routing works in two phases: **reliable flooding** and **route calculation**.

Routers running distance vector algorithms share summarized reachability information with their neighbors. Every router running link-state algorithms, on the other hand, builds a complete picture of the whole network (which is **phase I**) before computing the shortest path to all destinations.

Then, based on this learned topology, each router is able to compute its routing table by using the shortest path computation such as Dijkstra’s Algorithm. This is **phase II**.

Here’s a little roadmap of routing algorithm types.



**Note: Central Route Calculation** It’s possible that in a network all the routes are computed centrally using a specific routing algorithm, and then downloaded to all the routers in the network. This approach can be used in relatively small networks but has scalability problems. This is usually achieved via architectures based on constructs such as [Path Computation Element \(PCE\)](#).

On the other hand, routing tables can be computed in a distributed manner, whereby routers share reachability information with their neighbors. Then all routers independently run a routing algorithm to compute their local routing table. This is what we're mainly focusing on in this course.

## Quick Quiz! #

1

Given a network with 4 links with the following available bandwidth on each:

1. 100
2. 300
3. 40
4. 10

Which of the following is a possible value for  $C$  in the given equation to decide weights for each link?

$$\text{weight} = \frac{C}{\text{bandwidth}}$$

COMPLETED 0%

1 of 3



Coming up next, we'll look at distance vector routing in-depth.



# The Control Plane: Distance Vector - Routing Information Protocol

In this lesson, we'll discuss the Routing Information Protocol, a popular distance-vector algorithm, based on the famous Bellman-Ford algorithm.

## WE'LL COVER THE FOLLOWING



- Introduction
  - Initial State
    - Initial Routing Table
  - The Algorithm
    - Example
  - Count to Infinity Problem
  - Fix #1: Split Horizon
  - Fix #2: Split Horizon with Poison Reverse
- Quick Quiz!

## Introduction #

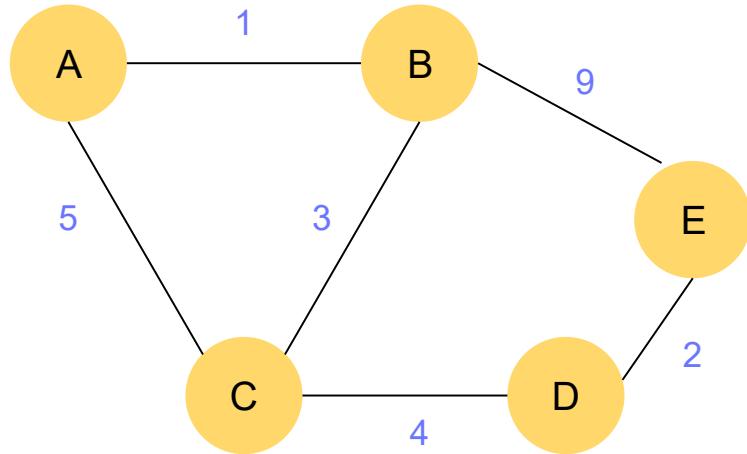
The **Routing Information Protocol (RIP)** based on the famous **Bellman-Ford** algorithm belongs to the distance vector class of routing algorithms and was used in ARPANET. While it used to be incredibly popular, it's not used very much now. There are other distance vector routing algorithms too such as Ford-Fulkerson.

## Initial State #

Each router or ‘node,’ maintains a **routing table** that initially contains the *estimated* cost to each of its neighbors.

Consider the following example of a small network where the yellow circles represent nodes, the black lines represent links, and the purple numbers

represent the cost of each link.



Sample Network

### Initial Routing Table #

What would the initial routing table look like at node **C** for example?

Destination	Cost	Next Hop
A	5	A
B	3	B
D	4	D

initial routing table at C

The table contains:

1. The names of the **destination nodes** which are the neighbors in this case.
2. The **initial cost** of the link to each of C's neighbors,
3. The '**next hop**' **node**, i.e. the node that C would have to send a packet to in order for it to reach its destination. In this case, the next hop and the destination are the same since the destinations are all C's neighbors.

Every node receives all of its neighbors' routing tables in two cases:

1. When a **trigger** happens such as a router or link failure happens.
2. Every  $N$  seconds, where  $N$  is a configurable parameter.

## The Algorithm #

Let's look at *how* the distance vector routing algorithm would arrive at the table above.

When a node **x** receives a routing table, `routing_table_y` from a neighbor **y**, the node applies the **Bellman Equation** in order to calculate/update the cost to reach each of **y**'s neighbors. Note that `routing_table_x` is the routing table of the node **x**.

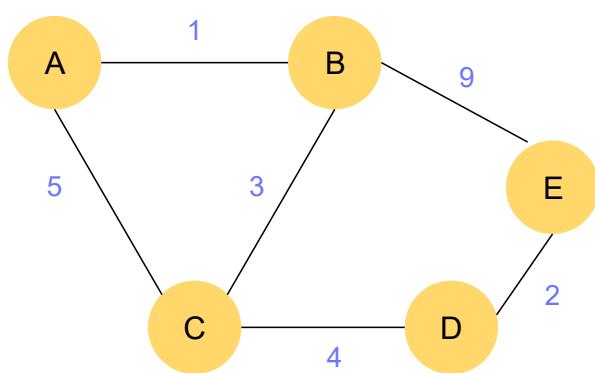
1. The Bellman Equation returns the **minimum** of the current estimated cost to reach a node, and the cost to reach a node *via y*.
2. If **x** does not have an estimated cost to reach a certain node, the cost to reach it is entered as the cost to reach the node *via y*.

Here's some pseudocode to clarify the picture.

```
for node in routing_table_y:  
    if (node in routing_table_x):  
        # Existing route, is the new one better?  
        if (routing_table_x[node].cost > routing_table_x[y].cost + node.cost): # If current cost  
            routing_table_x[node].cost = routing_table_x[y].cost + node.cost # Update  
        else:  
            # New route  
            routing_table_x[node].cost = routing_table_x[y].cost + node.cost # Add entry
```

The estimated cost will finally converge to the optimal cost after a series of these message exchanges. Have a look at the following slides for an example:

## Example #



Destination	Cost	Next Hop	Destination	Cost	Next Hop
A	5	A	B	1	B
B	3	B	C	5	C
D	4	D			

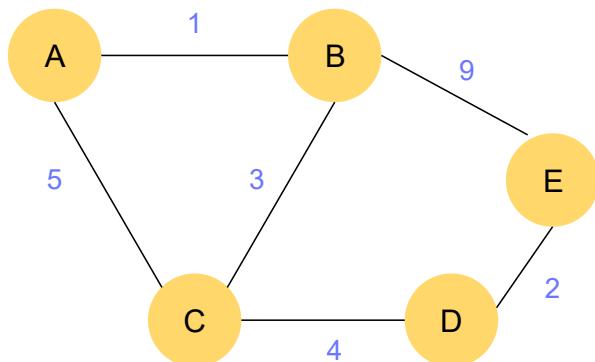
Routing table of C

Routing table of A

Node C receives a routing table from node A

### Distance Vector Routing: Table Upgrades

1 of 11



Destination	Cost	Next Hop	Destination	Cost	Next Hop
A	5	A	B	1	B
B	3	B	C	5	C
D	4	D			

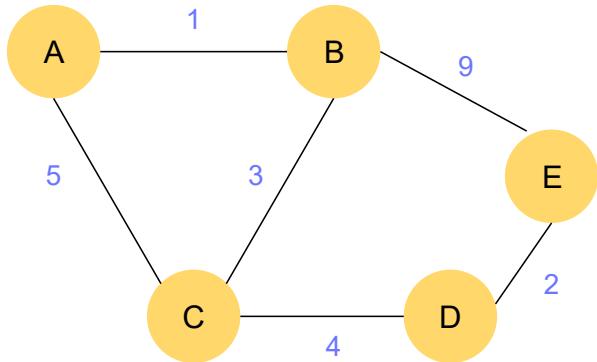
Routing table of C

Routing table of A

Node C applies Bellman's equation to each entry

### Distance Vector Routing: Table Upgrades

2 of 11



Destination	Cost	Next Hop
A	5	A
B	3	B
D	4	D
E		
C		

Routing table of C

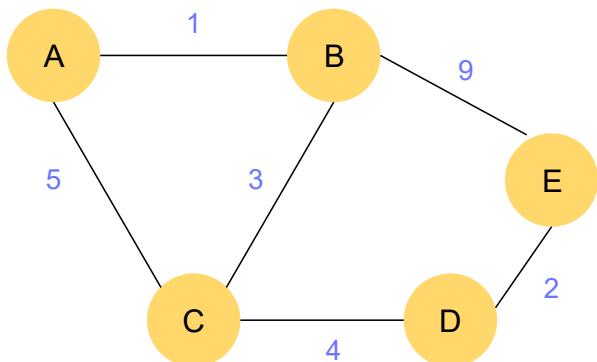
Destination	Cost	Next Hop
B	1	B
C	5	C
D		
E		

Routing table of A

Here's how **Bellman-Ford** would work in this case: **C** compares the cost of each node in **A**'s table like so: if cost of **C->node** is **greater** than cost of **C->A** + cost of **A->node** then replace the cost with the cost of **C->A** + cost of **A->node**

#### Distance Vector Routing: Table Upgrades

3 of 11



Destination	Cost	Next Hop
A	5	A
B	3	B
D	4	D
E		
C		

Routing table of C

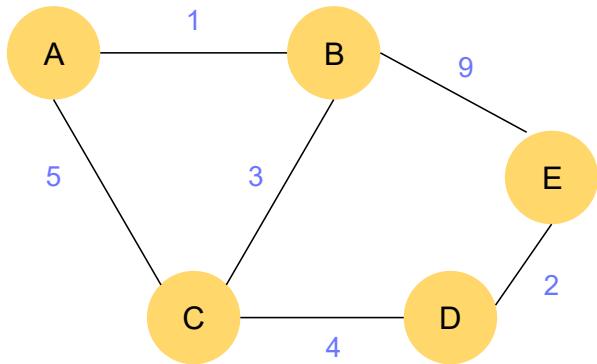
Destination	Cost	Next Hop
B	1	B
C	5	C
D		
E		

Routing table of A

Lets start with **B**

#### Distance Vector Routing: Table Upgrades

4 of 11



Destination	Cost	Next Hop	Destination	Cost	Next Hop
A	5	A	B	1	B
B	3	B	C	5	C
D	4	D			

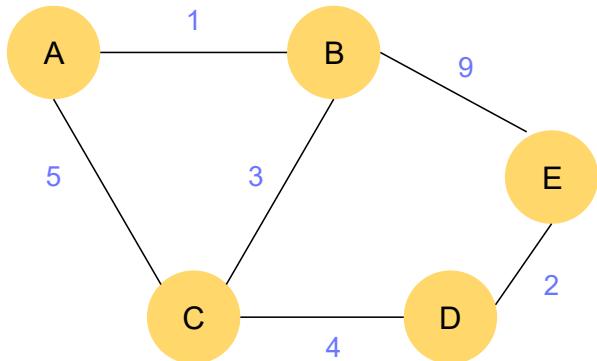
Routing table of C

Routing table of A

if cost of C->B is **greater** than cost of C->A + cost of A->B  
then replace the cost with the cost of C->A + cost of A->B

## Distance Vector Routing: Table Upgrades

5 of 11



Destination	Cost	Next Hop	Destination	Cost	Next Hop
A	5	A	B	1	B
B	3	B	C	5	C
D	4	D			

Routing table of C

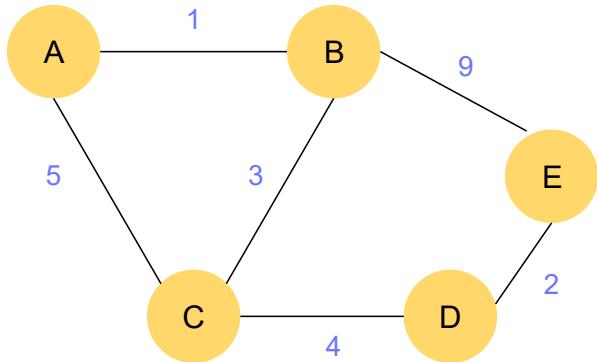
Routing table of A

Node	C->node	C->A + A->node	isGreater
B	3	5+1=6	No
C	0	5+5=10	No

Calculations to check if routing table of C should be updated

## Distance Vector Routing: Table Upgrades

6 of 11



C has received reachability information from B

Destination	Cost	Next Hop	Destination	Cost	Next Hop
A	5	A	A	1	A
B	3	B	C	3	C
D	4	D	E	9	E

Routing table of C

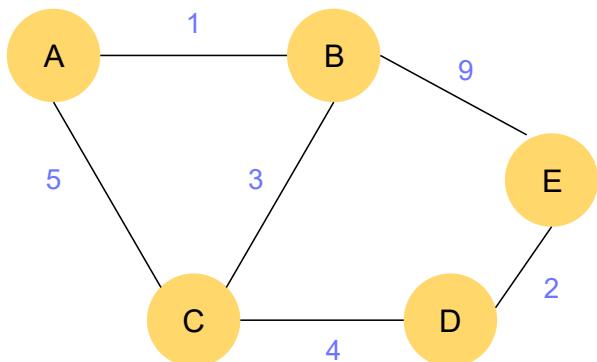
Routing table of B

Node	C->node	C->B + B->node	isGreater
A	5	3+1=4	Yes
C	0	3+3=6	No
E	$\infty$	3+9=12	Yes

Calculations to check if routing table of C should be updated

### Distance Vector Routing: Table Upgrades

7 of 11



Destination	Cost	Next Hop	Destination	Cost	Next Hop
A	4	B	A	1	A
B	3	B	C	3	C
D	4	D	E	9	E
C	12	B			

Routing table of C

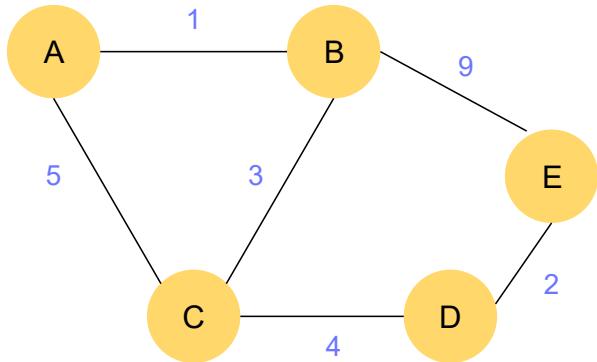
Routing table of B

Node	C->node	C->B + B->node	isGreater
A	5	3+1=4	Yes
C	0	3+3=6	No
E	$\infty$	3+9=12	Yes

Calculations to check if routing table of C should be updated

### Distance Vector Routing: Table Upgrades

8 of 11



C has received reachability information from D

Destination	Cost	Next Hop	Destination	Cost	Next Hop
A	4	B	C	4	C
B	3	B	E	2	E
D	4	D			
E	6	D			

Routing table of C

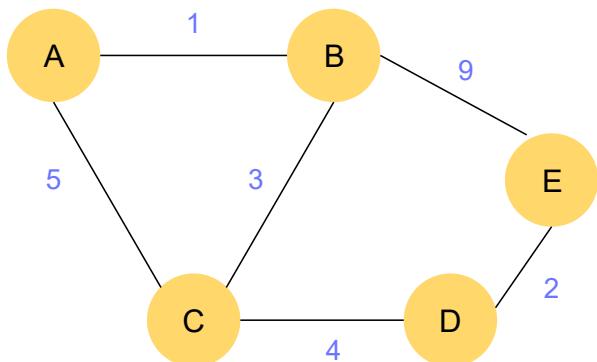
Routing table of D

Node	C->node	C->D + D->node	isGreater
E	12	4+2=6	Yes

Calculations to check if routing table of C should be updated

### Distance Vector Routing: Table Upgrades

9 of 11



C has received reachability information from D

Destination	Cost	Next Hop	Destination	Cost	Next Hop
A	4	B	C	4	C
B	3	B	E	2	E
D	4	D			
E	6	D			

Routing table of C

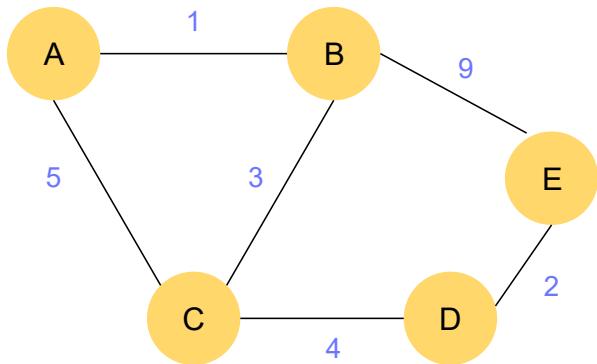
Routing table of D

Node	C->node	C->D + D->node	isGreater
E	12	4+2=6	Yes

Calculations to check if routing table of C should be updated

### Distance Vector Routing: Table Upgrades

10 of 11



Destination	Cost	Next Hop
A	4	B
B	3	B
D	4	D
E	6	D

Final routing table of **C**

Distance Vector Routing: Table Upgrades

11 of 11



That being said, this algorithm has some issues.

## Count to Infinity Problem #

Consider the following scenario based on the given graph below:

1. Suppose the link between **D** and **E** fails.
2. **D** corrects its routing table such that the cost to reach **E** is infinity  $\infty$ .
3. The network will be triggered to exchange routing tables.
4. Suppose **C** gets a chance to advertise its routing table first and sends it over to **D**.
5. Since **C** does not yet know about the link failure between **D** and **E**, its routing table has an entry for a route to **E** with the cost of 6 via **D**.
6. **D** will notice that **C** has a route to **E** and will update its routing table with a route to **E** via **C**. The cost of the route will be the sum of the cost of **D** to **C** (4) and the cost of **C** to **E** (6), which is 10.

a route to **E** via **C**. The cost of the route will be the sum of the cost of **D** to **C**, and the cost **C** has to reach **E**:  $4 + 6 = 9$ .

7. When **C** receives **D**'s routing table, it will notice that **D** has changed its cost to reach **E** from 2 to 9 and will update its table accordingly to  $4 + 9 = 13$ . Then it advertises it to other neighbors.
8. Things will eventually converge at **C** with a cost of 12 to **E** and **D** will learn that it can reach **E** at a cost of 16.
9. So, the ‘infinity’, in this case is not very high. The problem is that this convergence takes quite a while, and until the routers converge, there is a forwarding loop. Packets at **C** destined to **E** would go around in circles until their TTL expires.

There are a few improvements we can look at:

## Fix #1: Split Horizon #

This count to infinity problem occurs because a node **C** advertises to its neighbor **B** a route that it's learned from the neighbor **B** itself.

A possible solution to avoid this problem could be to change how a router creates its routing table. Instead of computing one routing table and sending it to all its neighbors, a router could create a routing table that's specific to each neighbor and **only contains the routes that have not been learned via this neighbor**. This technique is called **Split Horizon**.

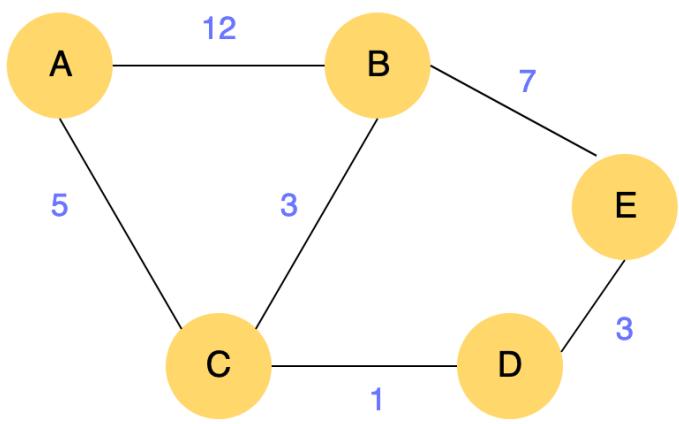
## Fix #2: Split Horizon with Poison Reverse #

Another variant of this is called **Split-Horizon with Poison Reverse**. Nodes using this variant advertise a cost of  $\infty$  for the destinations that they reach via the node to which they send their routing tables. Split Horizon with poison reverse converges faster than regular split horizon.

## Quick Quiz! #

Q

What would the initial routing table look like at **B** for the following example with distance vector routing?



COMPLETED 0%

1 of 1



The most popular routing algorithms today use **link state routing**, which we'll look at in the next lesson.

# Programming Challenge: Routing Information Protocol

In this lesson, you'll be writing code for the routing information protocol that we looked at previously.

## WE'LL COVER THE FOLLOWING ^

- Problem Statement
- Starter Code
  - `topology_reader.py`
    - Sample Input
  - `router.py`
  - What Our Test Does?
    - `port.py`
    - `rip_packet.py`
- Coding Exercise

## Problem Statement #

In this challenge, you will implement the routing information protocol that we just studied in the last lesson! You're given some starter code files.

## Starter Code #

For this coding challenge, we are providing you with a network simulator written in `python3`. The implementation of our simplified version of RIP is also required in Python. Let's look at the starter code module by module.

### `topology_reader.py` #

This is the entry point to our code. It takes a network topology in the form of a Python list as input and returns a list of `router` objects that reflect that topology. Here's what the topology looks like:

### Sample Input #

```
topology = [
    [1, [11, 2, 21, 1], [12, 4, 41, 1]], # Routers and ports
    [2,[21, 1, 11, 1],[22, 5, 53, 1], [23,3,31,1]], #[IP of router, [port of router, IP of dest
    [3,[31,2,23,1],[32,5,52,1]],
    [4,[41,1,12, 1],[42,5,51,1]],
    [5,[51,4,42,1],[52,3,32,1],[53,2,22,1]]
]
```

The list consists of sublists. Each sublist represents **one** router. So `[1, [11, 2, 21, 1], [12, 4, 41, 1]]`, for instance, represents a router.

- The first element of this sublist is the IP address of the router. It is `1` in the case of the example above.
- The rest of the elements of this sublist are other lists that represent ports and their links to ports on other routers.
- There can be as many of these sub-sublists as there are ports.
- Each port-link sub-sublist is as follows:
  - `[IP of port of router this router, IP of destination router, IP of port of destination router, cost of link]`
- So the topology looks like:
  - `[[IP of router, [IP of port of this router, IP of destination router, IP of port of destination router, cost of link], [IP of router, [IP of port of this router, IP of destination router, IP of port of destination router, cost of link], ...]`
- Note that a link between two routers **has** to be present in both. So a link to a port on a router with IP `2` from a router with IP `1` `[1, [11, 2, 21, 1], [12, 4, 41, 1]]` is reflected in the sublist of router with IP `2`, as follows: `[2, [21, 1, 11, 1], [22, 5, 53, 1], [23,3,31,1]]`

## router.py #

This file contains two classes: `router_base` and `router`.

- The `router_base` class contains the IP address, a list of RIP entries and a list of ports for each router, along with some functions that will help you implement the protocol. The IP address is self-explanatory but we'll get to the other two in a minute.
- The `router` class **inherits** the `router_base` class and is the class **you'll be**

working in. In particular, you'll be writing the functions

`send_RIP_packets()` and `receive_RIP_packets()`.

## What Our Test Does? #

Our testing code is simply a **network simulator** that supplies a number of network topologies, creates a list of routers with `topology_reader()` and calls `send_RIP_packets()` on each router `steps_to_run` number of times where `steps_to_run` is randomized. It then stores the list of routers returned from the `send_RIP_packets()` function and checks if they are as expected. Note that our testing code is not visible to you.

### port.py #

This file contains the classes `port_link` and `port`.

- The `port_link` class defines the links on each port. This class consists of the destination router's IP address (`dest_IP_address`), the destination router's port's IP addresses (`dest_port_IP`) and the cost of the link (`cost`).
- The `port` class has two attributes: the IP address of the port (`port_IP`), and the link on the port (`link`) which is an object of the class `port_link`.

The `topology_to_routers` function from the `topology_reader.py` file first creates a links, then ports and then finally passes them to router objects upon creation.

### rip\_packet.py #

This file consists of two classes: `RIP_entry` and `RIP_packet`.

- `RIP_entry`: each object of this class represents an entry of the forwarding table of the router. Each object of the `router` class consists of a list of these which makes up its forwarding table. Each `RIP_entry` object will have the following attributes: The IP address of the sending port (`port_IP`), the cost of sending on this link (`cost`), the IP address of the destination router (`dest_IP_address`), and the IP address of the next hop router (`next_hop_IP`).
- `RIP_packet`: when a router wants to forward its RIP entries, it does so by creating an object of this class. This class consists of the list of RIP entries

a few useful methods.

## Coding Exercise #

Great! Now you have some background on the code. Note that we haven't discussed the skeleton code in its entirety so you should read it to understand the methods provided. Try the challenge yourself in the widget below!

Note that `main.py` is empty. That's okay, don't worry about it.

main.py



port.py

rip\_packet.py

router.py

topology\_reader.py

```
from rip_packet import RIP_entry
from rip_packet import RIP_packet
from port import port
from port import port_link

class router_base:
    def __init__(self, IP_address, rip_entries, ports):
        self.IP_address = IP_address
        self.rip_entries = rip_entries
        self.ports = ports

    def add_port(self, prt):
        self.ports.append(prt)

    def add_RIP_entry(self, port_IP, dest_IP, cost, next_hop_IP):
        new_rip_entry = RIP_entry(port_IP, cost, dest_IP, next_hop_IP)
        self.rip_entries.append(new_rip_entry)

    def find_RIP_entry(self, destination_IP_to_find):
        for entry in self.rip_entries:
            if(entry.dest_IP_address == destination_IP_to_find):
                return entry
        return None

    def set_RIP_entry_cost(self, destination_IP_to_find, new_cost):
        for i in self.rip_entries:
            if(i.dest_IP_address == destination_IP_to_find):
                i.set_cost(new_cost)
        return None

    def delete_RIP_entry(self, destination_IP_to_find):
```

```

        for entry in self.rip_entries:
            if(entry.dest_IP_address == destination_IP_to_find):
                self.rip_entries.remove(entry)

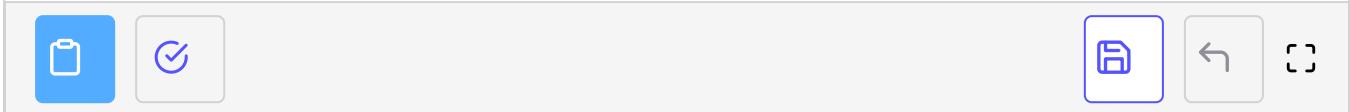
    def print_router(self):
        print("~~~~ Router IP address = " + str(self.IP_address) + "~~~~")
        print("---Ports---")
        print("Port IP | Destination Router IP | Destination Port IP | Cost")
        for p in self.ports:
            p.print_port()
        print("---RIP entries---")
        print("port IP | destination IP address | next hop | cost")
        for re in self.rip_entries:
            re.print_rip_entry()

    def return_router(self):
        r = []
        r.append("~~~~ Router IP address = " + str(self.IP_address) + "~~~~")
        r.append("---Ports---")
        r.append("Port IP | Destination Router IP | Destination Port IP | Cost")
        for p in self.ports:
            r.append(p.return_port())
        r.append("---RIP entries---")
        r.append("port IP | destination IP address | next hop | cost")
        for re in self.rip_entries:
            r.append(re.return_rip_entry())
        return r

    class router(router_base):
        def send_RIP_packets(self, routers):
            # Write your code here
            return routers

        def receive_RIP_packets(self, rip_packet, link_send_on, routers, next_hop_IP):
            # Write your code here
            return routers

```



In the next lesson, we'll look at a detailed analysis of the exercise.

# Solution Review: Routing Information Protocol

In this lesson, we'll analyze the solution to the RIP challenge.

## WE'LL COVER THE FOLLOWING ^

- Solution
- Explanation
  - `send_RIP_packets()`
  - `receive_RIP_packets()`

## Solution #

main.py



port.py

rip\_packet.py

router.py

topology\_reader.py

```
from rip_packet import RIP_entry
from rip_packet import RIP_packet
from port import port
from port import port_link

class router_base:
    def __init__(self, IP_address, rip_entries, ports):
        self.IP_address = IP_address
        self.rip_entries = rip_entries
        self.ports = ports

    def add_port(self, prt):
        self.ports.append(prt)

    def add_RIP_entry(self, port_IP, dest_IP, cost, next_hop_IP):
        new_rip_entry = RIP_entry(port_IP, cost, dest_IP, next_hop_IP)
        self.rip_entries.append(new_rip_entry)

    def find_RIP_entry(self, destination_IP_to_find):
```



```
        self.rip_entries[j].next_hop_IP = next_hop_IP
        break
    # If entry does not already exist
    if(not found):
        new_rip_entries.append(RIP_entry(link_send_on.dest_port_IP, rip_packet.rip_entries[i]))
        found = 0
    self.rip_entries.extend(new_rip_entries)
return routers
```



## Explanation #

### send\_RIP\_packets() #

Let's go through the solution line-by-line:

- **line 62:** We create the RIP packet that we'll send to all of our neighbors on this line. We pass the `rip_entries` list and the length of that list on this line.
- **lines 63-66:** we now find the neighbors of this router. We do this by iterating over the given list of routers in the network and checking to see if any of our ports have a link to them. We do this by iterating over our ports and checking each port's link's destination IP address against the router's IP address. If they match, the router is a neighbor and we send it our RIP packet by calling `receive_RIP_packets()` on it.

Finally, the routers list is returned.

### receive\_RIP\_packets() #

This function consists of the core of the Routing Information Protocol. It works as follows:

1. The receiving router checks for all RIP entries in the received RIP packet if they exist in its own RIP entries list.
2. If an entry is for a destination that's **not found** in the receiving router's RIP entries list, it adds it as done on **lines 85-88**.
3. If an entry is for a destination that **is found** in the receiving router's RIP entries list, it does one of two things:

1. If the entry is from a router whose IP address **is equal** to the next hop IP in the router's current RIP entry **and** the cost has changed, it simply sets the cost as the minimum of 16 and the new cost. Note that the new cost can be greater or lesser than the current cost.
2. Otherwise, if the cost advertised in this RIP entry is lesser than the one the router currently has, it sets its RIP entry to the one through the router that sent the RIP entry.

Finally, the routers list is returned.

# The Control Plane: Link State Routing

In this lesson, we'll study link state routing.

## WE'LL COVER THE FOLLOWING ^

- Phase I: Reliable Flooding
  - Neighbor Discovery
  - LSPs
  - Flooding Algorithm
  - Handling Link Failure
    - Two-way Connectivity Check
    - Handling Router Failure
- Quick Quiz!

-Another way to create a routing table with the most efficient path between two routers or ‘nodes’ is by using **link-state** routing.

Link state routing works in two phases: **reliable flooding** and **route calculation**. Let’s look at phase I now.

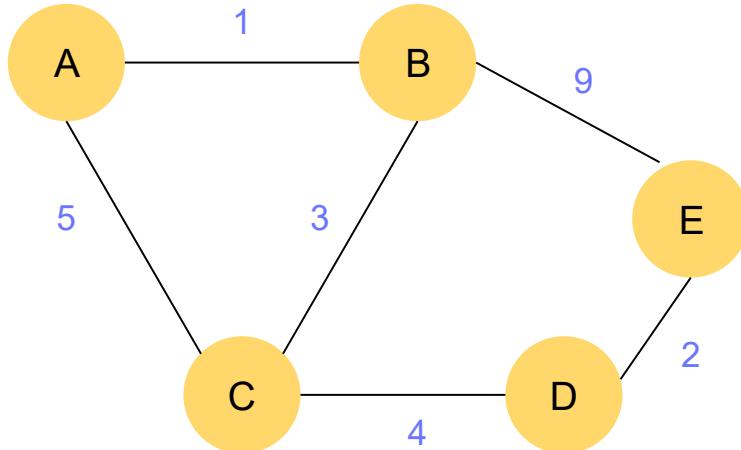
## Phase I: Reliable Flooding #

### Neighbor Discovery #

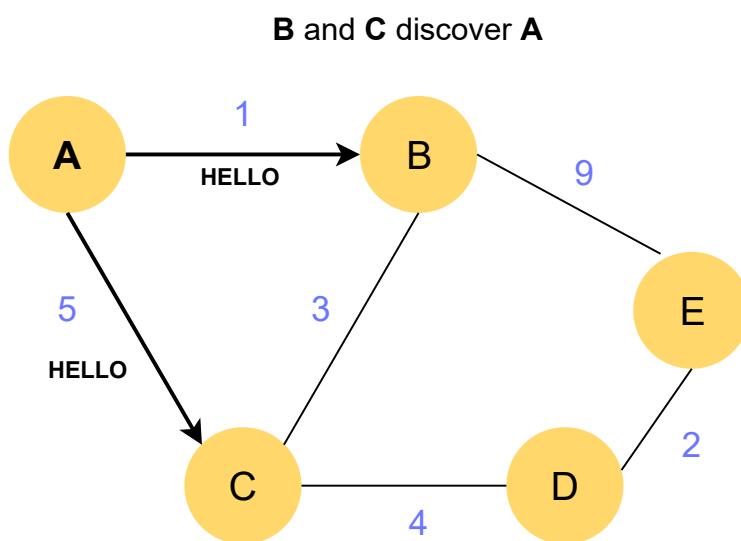
When a link-state router boots, it first needs to discover its neighbors. To do this:

- Each router sends a **HELLO** message every  $N$  seconds on all of its interfaces. This message contains the router’s unique address.
- As its neighboring routers also send **HELLO** messages, the router discovers its neighbors.
- These **HELLO** messages are only sent to neighbors that are **directly** connected.

- These **HELLO** messages are only sent to neighbors that are **directly** connected to a router, and a router never forwards the **HELLO** messages that they receive.
- **HELLO** messages are also used to detect link and router failures. A link is considered to have failed if no **HELLO** message has been received from the neighboring router for a period of  $k \times N$  seconds.

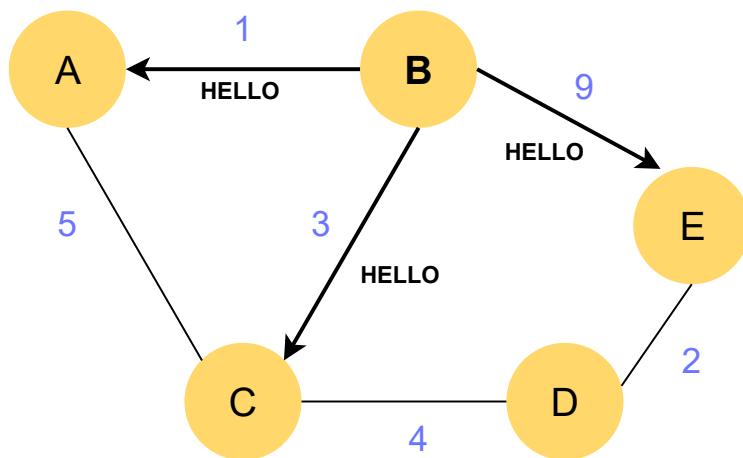


1 of 6



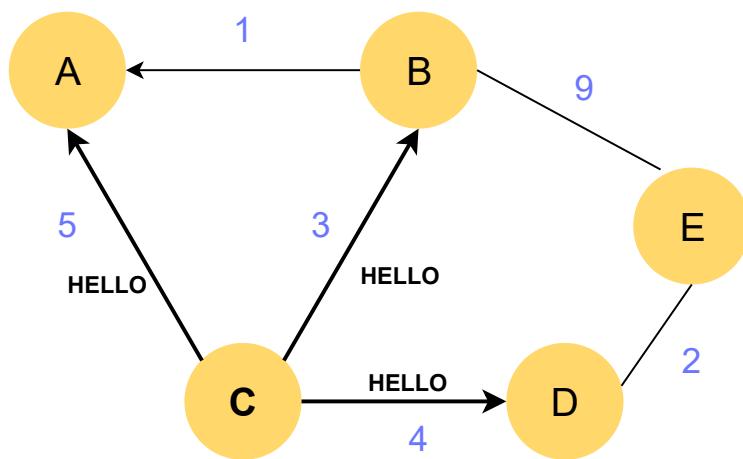
2 of 6

**A, C and E discover B**



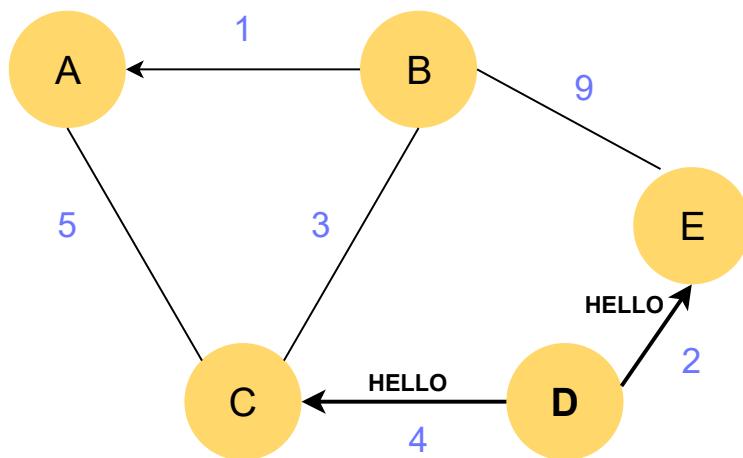
3 of 6

**A, B and D discover C**



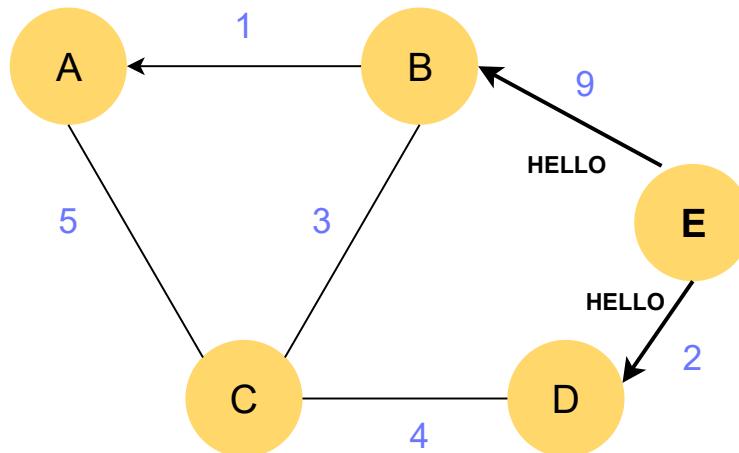
4 of 6

**E and C discover D**



5 of 6

**B and E discover E**



6 of 6



## LSPs #

Once a router has discovered its neighbours, it must reliably distribute its local links to all routers in the network to allow them to compute their local view of the network topology. For this, each router builds a link-state packet (LSP) that contains the following information:

- `LSP.Router` : identification (address) of the sender of the LSP
- `LSP.age` : age or remaining lifetime of the LSP
- `LSP.seq` : sequence number of the LSP
- `LSP.Links[]` : links advertised in the LSP. Each directed link is represented with the following information:
  - `LSP.Links[i].Id` : identification of the neighbour
  - `LSP.Links[i].cost` : cost of the link

## Flooding Algorithm #

The routers will construct their routing tables based on the LSPs.

The **Flooding Algorithm** is used to efficiently distribute the LSPs of all routers. Each router that implements flooding maintains a **link state database (LSDB)** that contains the most recent LSP sent by each router. When

a router receives an LSP, it:

- Verifies whether this LSP is already stored inside its LSDB. If so, the router has already distributed the LSP earlier and it **does not need to forward it**.
- Otherwise, **the router forwards the LSP on all links except the link over which the LSP was received.**

To ensure that all routers receive all LSPs, even when there are transmission errors, link state routing protocols use **reliable flooding**, which involves acknowledgments, and if necessary, retransmissions to ensure that all link state packets are successfully transferred to all neighbouring routers.

**What We Have so Far:** Routers combine the received LSPs with their own LSP to compute the entire network topology.

Then in phase II, the routers apply shortest path algorithms to compute the most efficient path.

## Handling Link Failure #

- When a link fails, the two routers attached to the link **detect the failure by the lack of HELLO messages** received in the last  $k \times N$  seconds.
- Once a router has detected a local link failure, it generates and **floods a new LSP that no longer contains the failed link**, and the new LSP replaces the previous LSP in the network.
- As the two routers attached to a link do not detect this failure exactly at the same time, the status of the link may be advertised differently by one of the routers.

## Two-way Connectivity Check #

- When a link is reported in the LSP of only **one** of the attached routers, the rest consider the link to have failed in **both directions** and remove it from the directed graph that they compute from their LSDB! This check allows link failures to be flooded quickly, as a single LSP is sufficient to announce such bad news.

- Furthermore, a link can only be used once the two attached routers have sent their LSps.

## Handling Router Failure #

- The two-way connectivity check also allows for dealing with **router failures**. When a router fails, all its links fail by definition. Unfortunately, it doesn't send a new LSp to announce its failure. The two-way connectivity check ensures that the failed router is removed from the graph.
- When a router fails, its LSp must be removed from the LSDB of all routers. This can be done by using the **age field** that is included in each LSp. When a router generates an LSp, it sets the **LSP.age** to a value called the LSp's **lifetime** (usually measured in seconds). All routers regularly decrement the age of the LSps in their LSDBs and an LSp is discarded once its age reaches 0.

## Quick Quiz! #

1

Why are routing loops uncommon in networks that use link state routing?



---

In the next lesson, we'll study Dijkstra's algorithm!

# The Control Plane: Route Calculation - Dijkstra's

In this lesson, we'll study Dijkstra's shortest path algorithm!

## WE'LL COVER THE FOLLOWING ^

- Phase II: Route Calculation
- Dijkstra's Algorithm
  - Algorithm
  - Finding the Shortest Path
- Visual Example
- Quick Quiz!

## Phase II: Route Calculation #

Each router then computes the spanning tree rooted at itself and calculates the entries in the routing table by using **Dijkstra's shortest path algorithm**. Dijkstra's is a common algorithm that is usually taught in *Algorithms* or *Data Structures* classes. Let's get a quick refresher of it.

## Dijkstra's Algorithm #

The goal is to find the shortest path from an **initial node** to all other nodes in the graph.

We first need to set up some data structures for us to use throughout the algorithm.

1. Create a set called the **unvisited set**. All the nodes are initially unvisited.
2. Create a set called the **visited set**. It's initially empty.
3. Create a list called the **parent** list. It will contain mappings of nodes to their parents.
4. Lastly, every node has a distance of it from the initial node. Initially, all

the nodes besides the initial node itself have a starting distance of infinity. We call this `d_node_n`,

5. Every link between two nodes in the graph has a certain weight. We call this `w_node_n_node_m`.

## Algorithm #

1. Start with the **initial node** in the graph. Mark it as the **current node**.
2. Consider each of its neighbor's that are NOT in the **visited** set.
3. If the sum of the distance of the current node and the distance to the neighbor from the current node is **lower** than the current distance of the neighbor, replace it with the new distance.
  - In other words, if `w_node_curr_node_n + d_node_curr < d_node_n`, set `d_node_n` to `w_node_curr_node_n + d_node_curr`.
  - Also, set the parent of this neighbor, `n`, to the current node.
4. Repeat step 3 for all unvisited neighbors. After that, add the current node to the visited set.
5. Repeat steps 2-4 for the neighbor with the lowest `d_node_n`. Continue until the entire graph is visited.

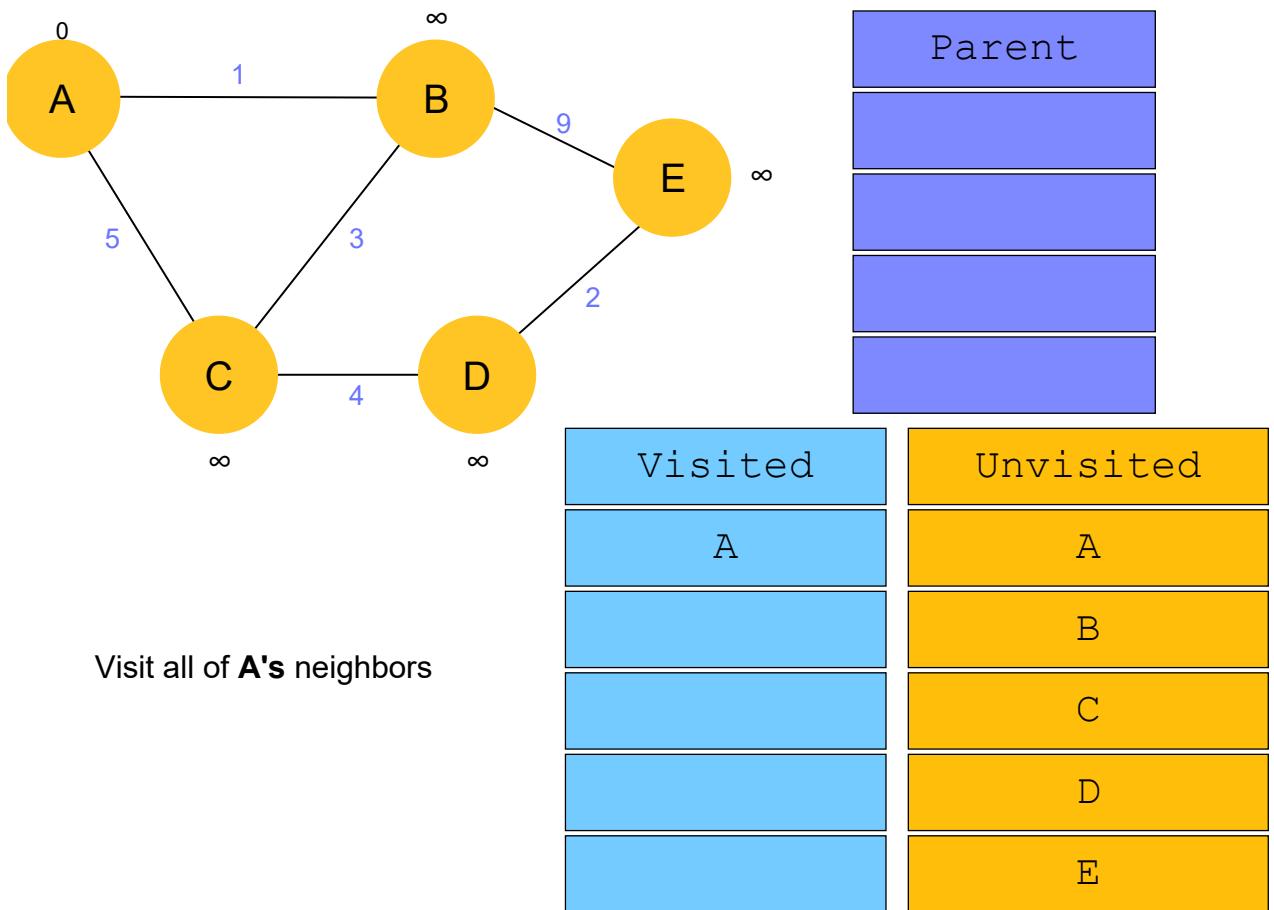
## Finding the Shortest Path #

To find the shortest path from a given node **n** to the initial node:

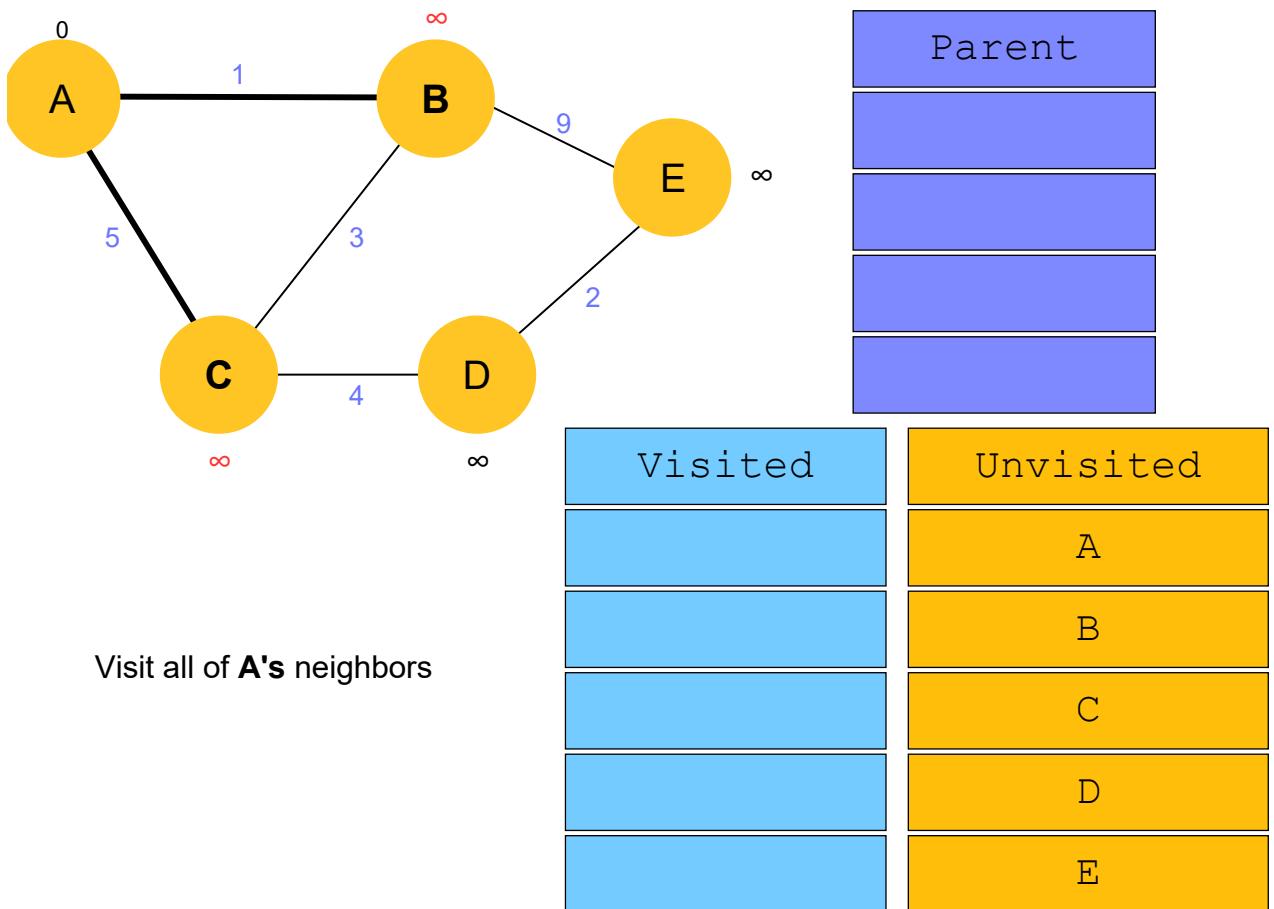
1. Find the parent of the current node. Initially the current node is **n**.
2. Set the current node to the new parent node.
3. Store each 'current node' in a stack.
4. Repeat steps 1-3 until the initial node is reached.
5. Pop and print the contents of the stack until it is empty.

## Visual Example #

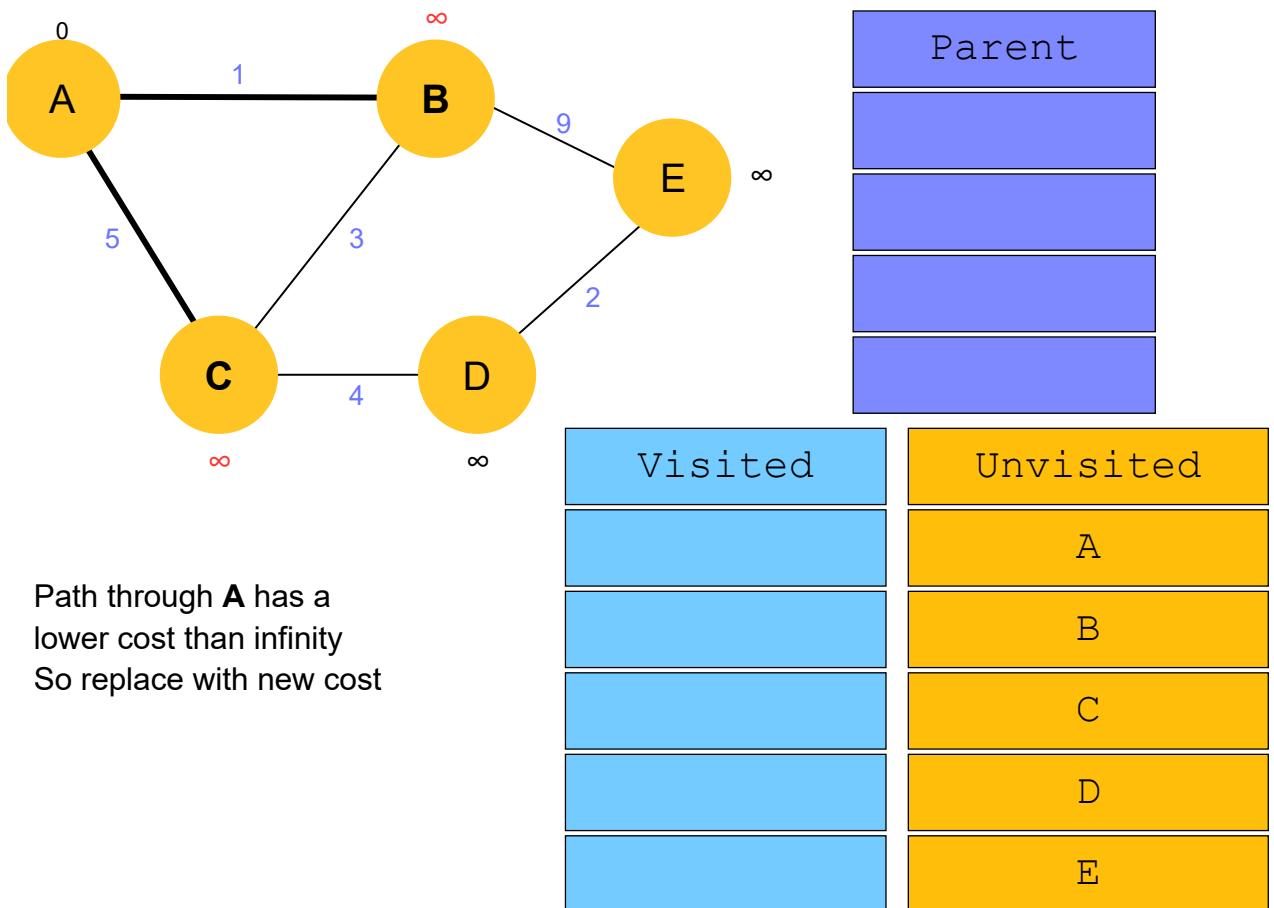
Have a look at the following example to see how Dijkstra's would apply to a graph.



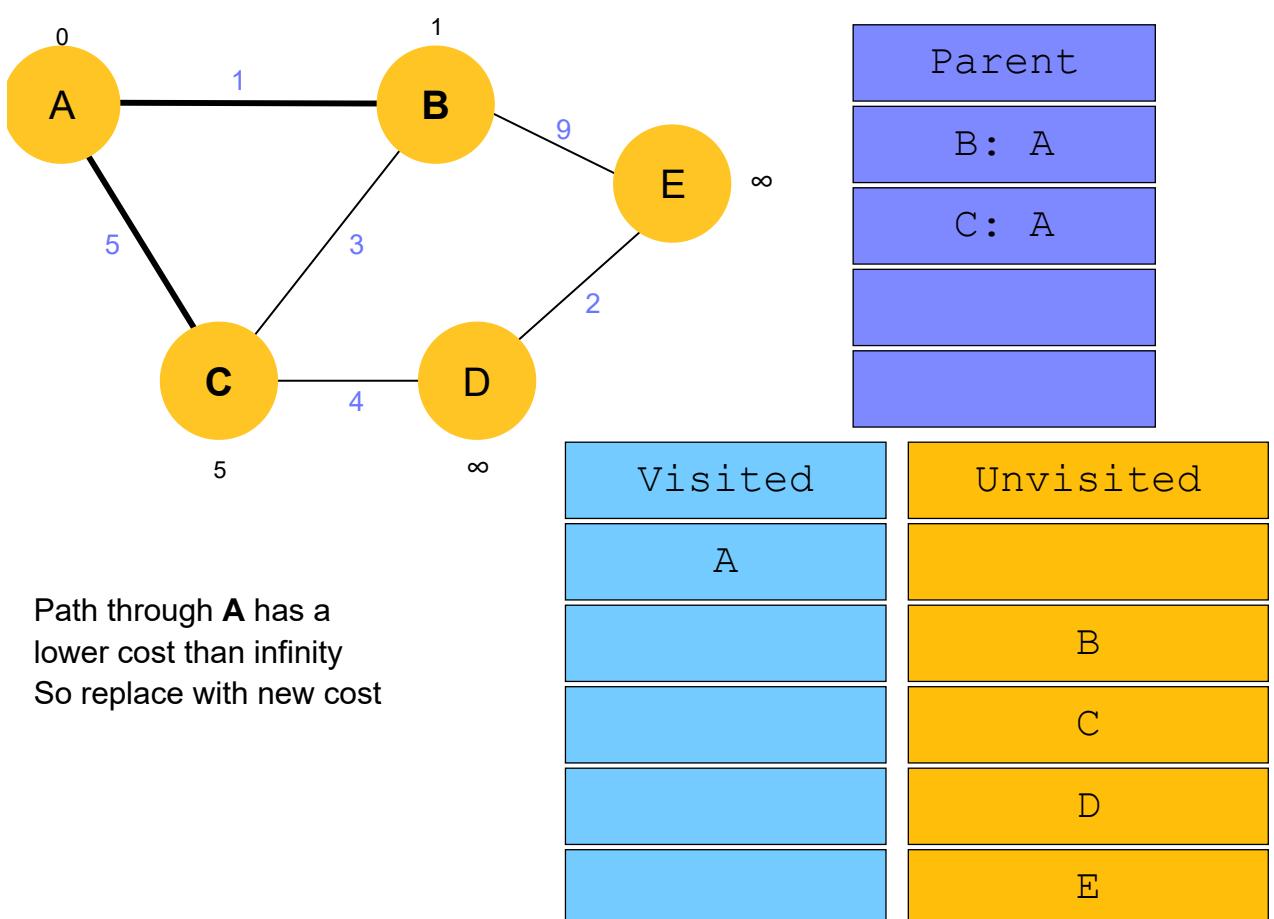
1 of 16



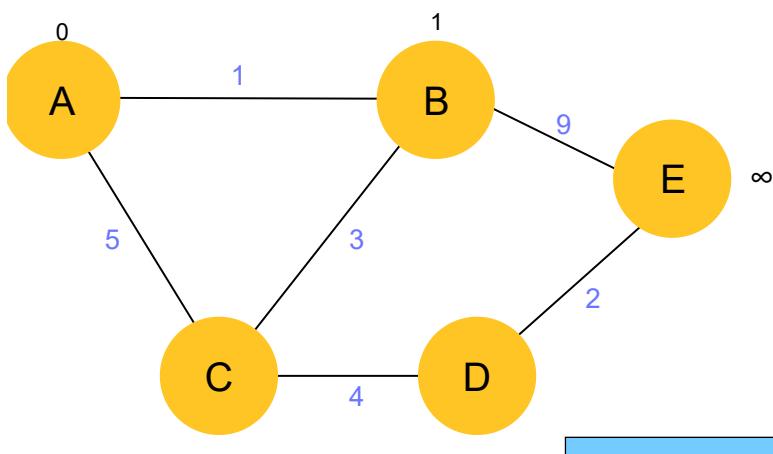
2 of 16



3 of 16



4 of 16

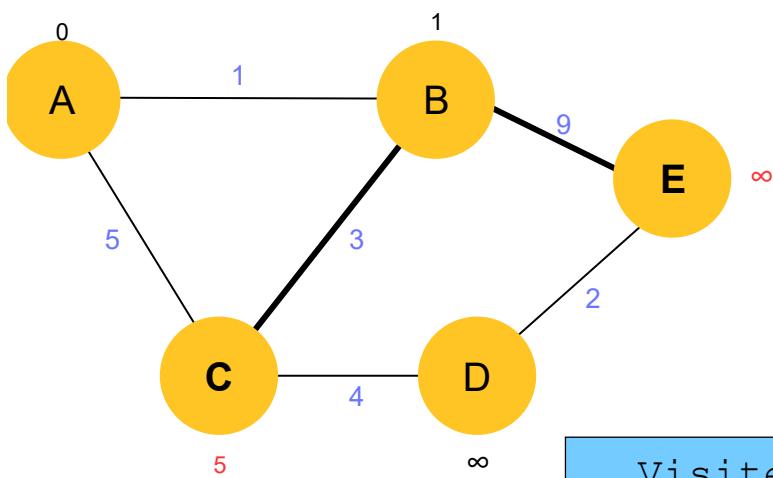


Parent
B: A
C: A

Visited	Unvisited
A	
	B
	C
	D
	E

Check paths to **B**'s unvisited neighbors next

5 of 16

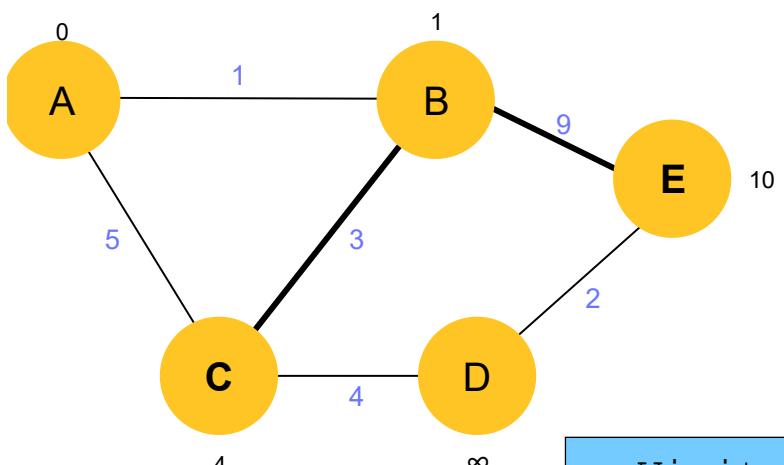


Parent
B: A
C: A

Visited	Unvisited
A	
	B
	C
	D
	E

Check paths to **B**'s unvisited neighbors next

6 of 16

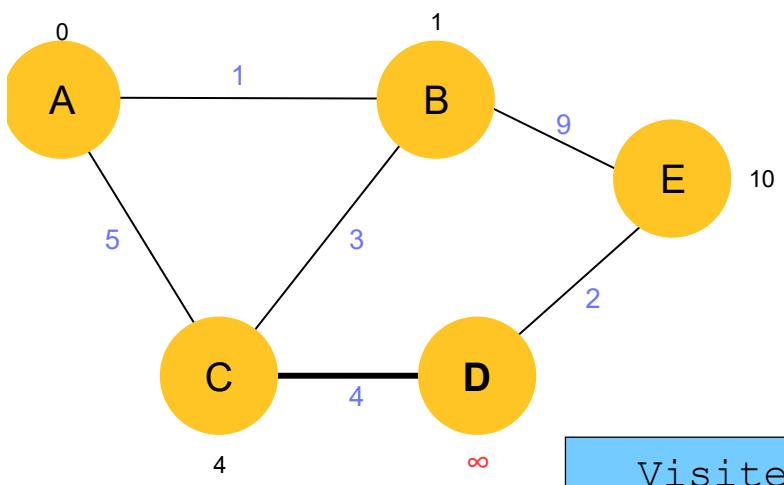


Parent
B: A
C: B
E: B

Visited	Unvisited
A	
B	
	C
	D
	E

Update costs according to the algorithm

7 of 16

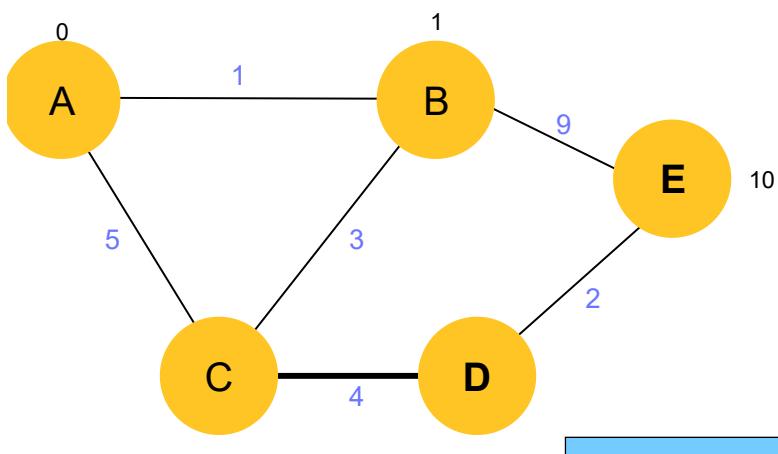


Parent
B: A
C: B
E: B

Visited	Unvisited
A	
B	
	C
	D
	E

Repeat the process for C

8 of 16

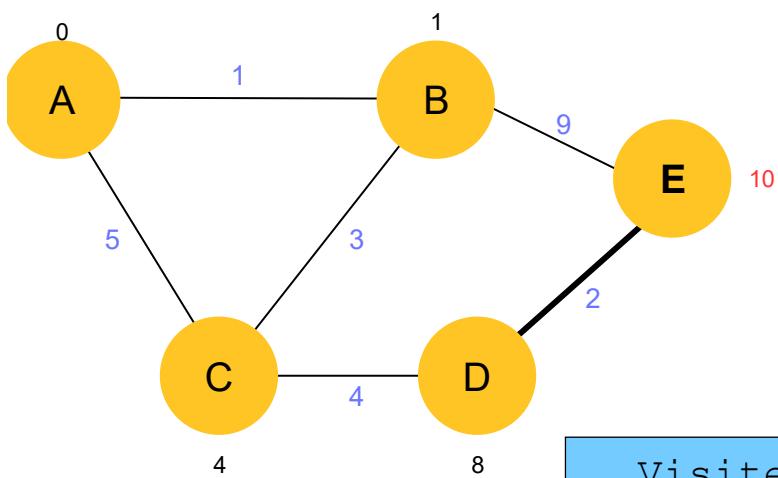


Parent
B: A
C: B
E: B
D: C

Repeat the process

Visited	Unvisited
A	
B	
C	
	D
	E

9 of 16

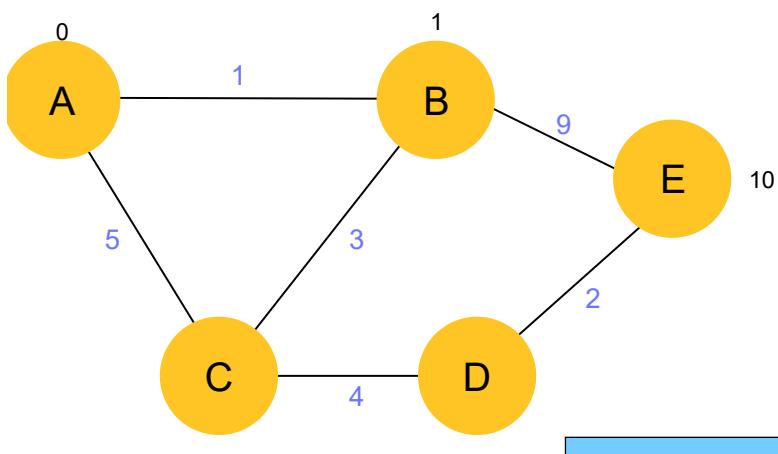


Parent
B: A
C: B
E: B
D: C

Repeat the process

Visited	Unvisited
A	
B	
C	
	D
	E

10 of 16

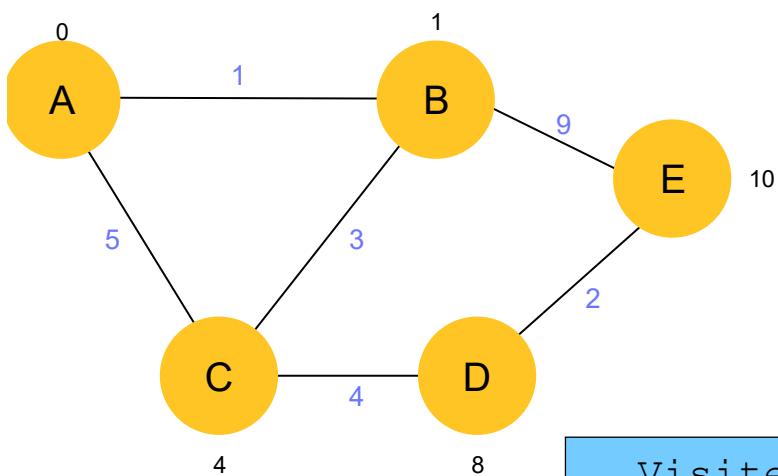


Parent
B: A
C: B
E: B
D: C

Repeat the process.  
E has no neighbors.

Visited	Unvisited
A	
B	
C	
D	
E	

11 of 16

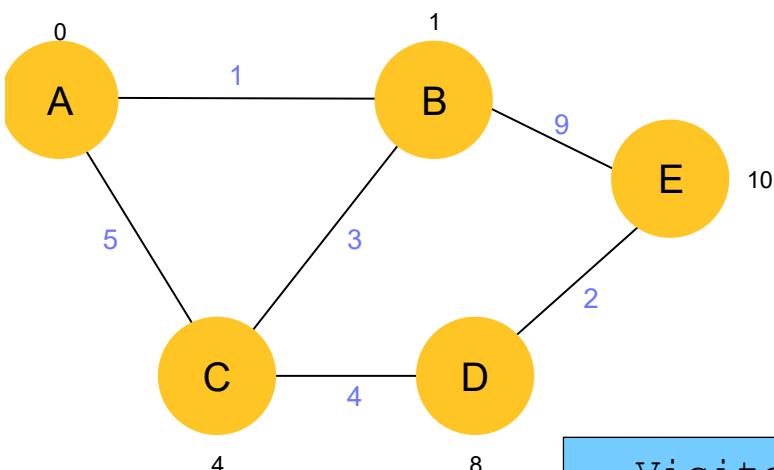


Parent
B: A
C: B
E: B
D: C

Now to find the shortest path between two vertices, we simply trace the parent array back from the destination until the source is reached. Take the A->E path for example.

Visited	Unvisited
A	
B	
C	
D	
E	

12 of 16

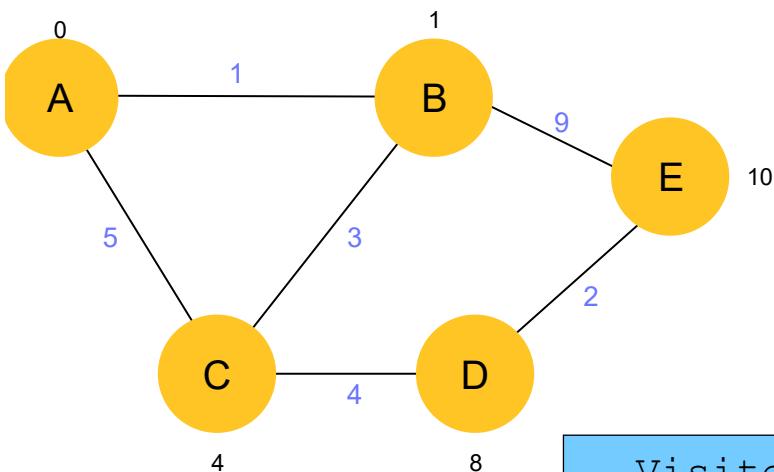


Parent
B: A
C: B
E: B
D: C

Now to find the shortest path between two vertices, we simply trace the parent array back from the destination until the source is reached. Take the A->E path for example.

Visited	Unvisited
A	
B	
C	
D	
E	

13 of 16

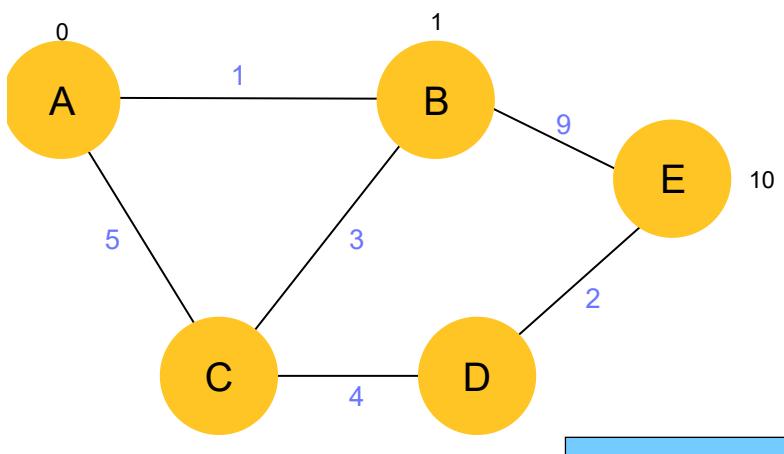


Parent
B: A
C: B
E: B
D: C

E->B

Visited	Unvisited
A	
B	
C	
D	
E	

14 of 16

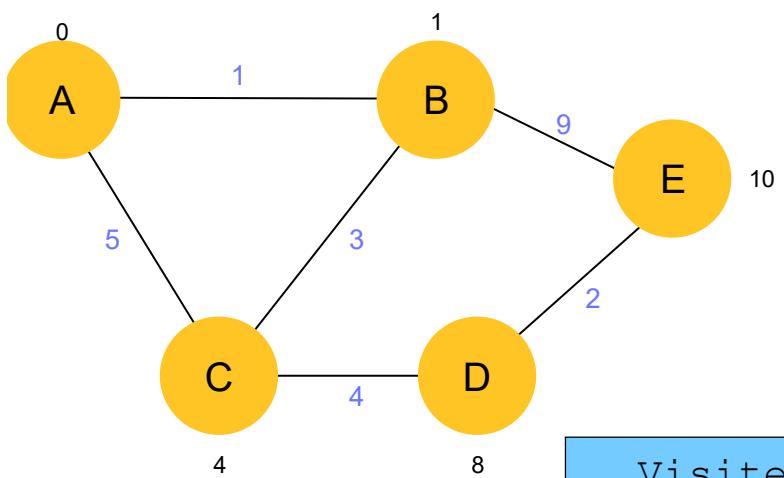


Parent
B: A
C: B
E: B
D: C

E->B->A

Visited	Unvisited
A	
B	
C	
D	
E	

15 of 16



Parent
B: A
C: B
E: B
D: C

So the shortest path from A to E is  
A->B->E

Visited	Unvisited
A	
B	
C	
D	
E	

16 of 16

# Quick Quiz! #

1

What is the aim of Dijkstra's Algorithm?

COMPLETED 0%

1 of 3



In the next lesson, you'll implement Dijkstra's Algorithm!

# Programming Challenge: Implementing Dijkstra's

In this lesson, you'll be implementing Dijkstra's Shortest Path Algorithm.

## WE'LL COVER THE FOLLOWING ^

- Problem Statement
  - Input
  - Output
  - Sample Input
  - Sample Output
- Coding Exercise

## Problem Statement #

Given an adjacency matrix in a 2D array, solve the **Single Source Shortest Path** algorithm, essentially by implementing the Dijkstra's algorithm discussed in the previous lesson. We've written some skeleton code for the function.

1. The value of the weight of the link is `graph[src][dst]`.
2. The graph is undirected so `graph[src][dst]==graph[dst][src]`.
3. A link between the `src` and `dst` exists if `-1<graph[src][dst]<16`.
4. If `graph[src][dst]>=16` the weight of the link is infinite and it does not function.

## Input #

1. An adjacency matrix, i.e., a 2D array, a source node, and a destination node.

## Output #

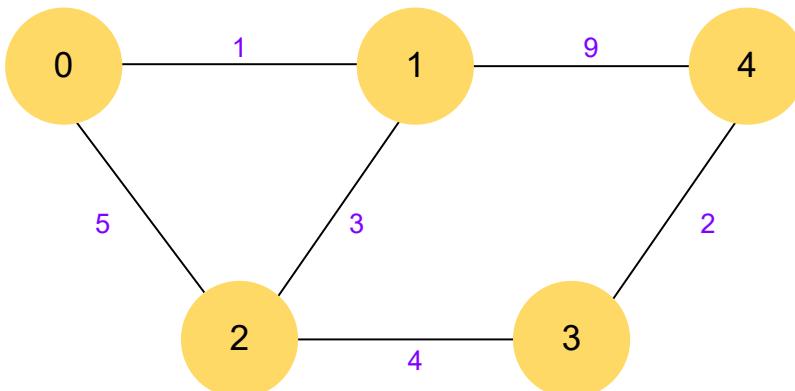
The **shortest path** between the source and destination in the form of an array

of integers where each integer represents a node and the **total weight** of the path.

## Sample Input #

```
1. graph = [  
    [0,1,5,-1,-1],  
    [1,0,3,-1,9],  
    [5,3,0,4,-1],  
    [-1,-1,4,0,2],  
    [-1,9,-1,2,0]  
]
```

This adjacency matrix represents the following graph:



2. A source and destination:

```
src = 0  
dst = 3
```

## Sample Output #

```
shortest_path = [0,1,2,3]  
cost = 8
```

## Coding Exercise #

Try it yourself below!

```
def Dijkstra(graph, src, dst):  
    pass
```



In the next lesson, we'll look at a solution to this problem.

# Solution Review: Implementing Dijkstra's

In this lesson, we'll look at a way to implement Dijkstra's Algorithm.

## WE'LL COVER THE FOLLOWING



- Solution #1: Calculating All Paths
- Explanation
- Solution #2: Stopping When Route to `dst` is Calculated
- Explanation

## Solution #1: Calculating All Paths #

```
def Dijkstra(graph, src, dst):  
    number_of_nodes = len(graph[0])    # Number of nodes in the graph  
    parent = [-1] * number_of_nodes    # Setting up various lists  
    visited = []  
    unvisited = [i for i in range(number_of_nodes)]  
    distance = [16] * number_of_nodes # The distance list initially has a distance of infinite  
    distance[src] = 0  
    shortest_path = []                # This list will have the shortest path at the end  
    current = src                    # We start with the source  
  
    while(len(unvisited)>0):  
        # Visit all neighbors of current and update distance  
        for i in range(number_of_nodes):  
            if(graph[current][i]>=0 and distance[i] > graph[current][i]+distance[current]):  
                distance[i] = graph[current][i]+distance[current] # Update distance  
                parent[i] = current # Set new parent  
  
        unvisited.remove(current) # Move current node from unvisited to visited  
        visited.append(current)  
        if(len(unvisited) != 0):  
            current = unvisited[0] # New current node is an unvisited node with the smallest 'dis  
            for n in unvisited:  
                if(distance[n] < distance[current]):  
                    current = n  
  
    curr = dst # Some code to get the shortest path from 'parent'  
    shortest_path.append(curr)  
    cost = 0  
    while curr is not src:  
        if parent[curr] == -1: # If there is no path to the source node  
            return([[],-1])  
        curr = parent[curr]
```

```

cost = cost + graph[curr][parent[curr]] # The cost is the sum of the links in a path
curr = parent[curr]
shortest_path.append(curr)

shortest_path.reverse()
return([shortest_path, cost])

def main():
    graph = [
        [0,1,5,-1,-1],
        [1,0,3,-1,9],
        [5,3,0,4,-1],
        [-1,-1,4,0,2],
        [-1,9,-1,2,0]
    ]
    src = 0
    dst = 3
    print(Dijkstra(graph,src,dst))

if __name__ == "__main__":
    main()

```



## Explanation #

Let's go through this code line by line.

- **Lines 1-9:** we set up a few variables that are important for the implementation.
  1. The `number_of_nodes` is the number of nodes in the graph. It's equivalent to the number of rows/columns of the given `graph`. This variable is not necessary for the algorithm itself, but makes calculating other variables clear and easy.
  2. The `parent` list will map each node to its 'parent' or the previous node in the shortest path to the source node. Initialized to `-1`.
  3. The `visited` list is initially empty.
  4. The `unvisited` list contains all the nodes in the graph. Since the nodes in our graph are labeled by numbers, this list is simple to generate.
  5. The `distance` list has all the current distances of all the nodes from the `src` node. Note that all the distances besides the distance of the `src` node from itself are set to infinity, i.e., `16`.
  6. The `shortest_path` is initialized to an empty list.
  7. The `current` node is set to the `src` node since we are calculating the

shortest paths from `src` to the rest of the nodes.

- **Lines 11-24:** The `while` loop.

1. We iterate through all of the neighbors of every `current` node.
2. If the current distance for a node is greater than the distance of the `current` node + the cost of the link between them, its distance is updated and its `parent` is changed to `current`.
3. Once all of the neighbors of a node have been exhausted, the next `current` node is chosen to be an unvisited node with the smallest `distance`.
4. Steps 1-3 are repeated until the while loop exits when no more nodes are left to visit.

- **Lines 26-34:** Calculating the shortest path and the cost. We traverse the `parent` list link by link until a path is generated. Since we start from the destination, the path has to be reversed. While we calculate the path, we also sum up the cost of each link that is traversed.

- **Example:** Assume the source node is `0`. IF `parent[2]` has the value `1`, the previous node from `2` is `1` as part of the shortest path to a source. Then, suppose `parent[1]` is `0`. So the shortest path will turn out to be `[2, 1, 0]`.

## Solution #2: Stopping When Route to `dst` is Calculated #

```
def Dijkstra(graph, src, dst):  
    number_of_nodes = len(graph[0])      # Number of nodes in the graph  
    parent = [-1] * number_of_nodes      # Setting up various lists  
    visited = []  
    unvisited = [i for i in range(number_of_nodes)]  
    distance = [16] * number_of_nodes    # The distance list initially has a distance of infinite  
    distance[src] = 0  
    shortest_path = []                  # This list will have the shortest path at the end  
    current = src                      # We start with the source  
  
    while(len(unvisited)>0):  
        # Visit all neighbors of current and update distance  
        for i in range(number_of_nodes):  
            if(graph[current][i]>=0 and distance[i] > graph[current][i]+distance[current]):  
                distance[i] = graph[current][i]+distance[current] # Update distance  
                parent[i] = current # Set new parent  
  
            if(current == dst):  
                break  
  
        unvisited.remove(current) # Move current node from unvisited to visted  
        visited.append(current)
```

```

if(len(unvisited) != 0):
    current = unvisited[0] # New current node is an unvisited node with the smallest 'dis'
    for n in unvisited:
        if(distance[n] < distance[current]):
            current = n

    curr = dst # Some code to get the shortest path from 'parent'
    shortest_path.append(curr)
    cost = 0
    while curr is not src:
        if parent[curr] == -1: # If there is no path to the source node
            return([[],-1])
        cost = cost + graph[curr][parent[curr]] # The cost is the sum of the links in a path
        curr = parent[curr]
        shortest_path.append(curr)
    shortest_path.reverse()
    return([shortest_path, cost])

def main():
    graph = [
        [0,1,5,-1,-1],
        [1,0,3,-1,9],
        [5,3,0,4,-1],
        [-1,-1,4,0,2],
        [-1,9,-1,2,0]
    ]
    src = 0
    dst = 3
    print(Dijkstra(graph,src,dst))

if __name__ == "__main__":
    main()

```



## Explanation #

This solution differs from the previous one because it exits the while loop as soon as the destination node is visited via the if condition on **lines 18 and 19**. Since subsequent calculations for the rest of the graph do not change the shortest path to the destination node, there is no need to visit all of them.

---

In the next lesson, we'll learn about the Internet protocol!

# The Internet Protocol: Introduction to IPv4

We're finally at the very core of the Internet. This lesson contains an introduction to the Internet protocol!

## WE'LL COVER THE FOLLOWING



- IP Version 4
- IP Addresses
- Multihoming
- Address Assignment
  - Subnetting
  - Address Classes
    - Subnet Masks
    - Network Address
    - Broadcast Address
    - Default Subnet Masks
    - Variable-Length Subnets
- Quick Quiz!

The **Internet Protocol (IP)** is the network layer protocol of the TCP/IP protocol suite. The flexibility of IP and its ability to use various types of underlying data link layer technologies is one of its key advantages. The current version of IP is **version 4** and is specified in [RFC 791](#). We first describe this version and later touch upon **IP version 6**, which is expected to replace IP version 4 in the not so distant future.

## IP Version 4 #

The design of IP version 4 was based on the following **assumptions**:

- IP should provide an *unreliable connectionless service*.
- IP operates with the *datagram transmission mode*.

- IP operates with the *datagram transmission mode*
- IP hosts must have *fixed size 32-bit addresses*
- IP must be *compatible with a variety of data link layers*
- IP hosts should be able to *exchange variable-length packets*

## IP Addresses #

The addresses are an important part of any network layer protocol. IPv4 addresses are written as 32 bit numbers in **dotted-decimal** format, such as a sequence of four integers separated by dots. Dotted decimal is a format imposed upon the 32-bit numbers for relatively easier human readability. For example:

- 1.2.3.4 corresponds to 00000001000000100000001100000100
- 127.0.0.1 corresponds to 01111110000000000000000000000001
- 255.255.255.255 corresponds to 11111111111111111111111111111111

1.2.3.4

00000001000000100000001100000100

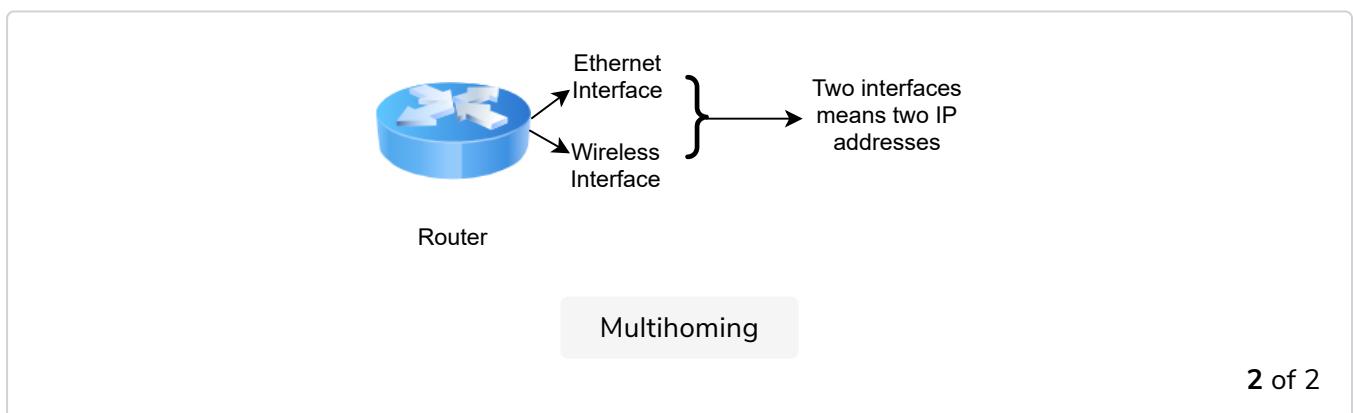
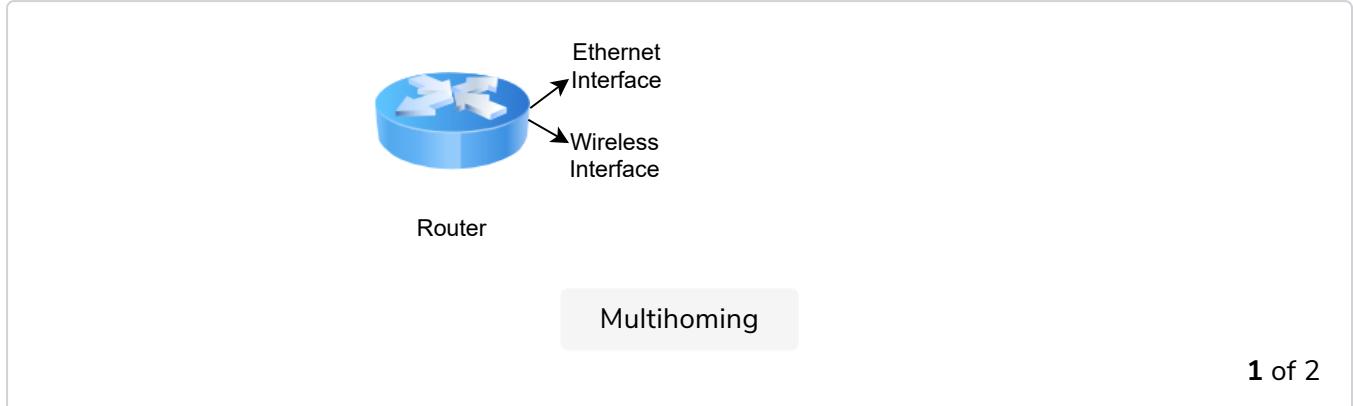
Every eight bits represent one decimal number in the IP address

## Multihoming #

An IPv4 address is used to identify an **interface on a router or an interface on a host**. Recall **network interfaces** from the physical communication media chapter.

A router has thus as many IPv4 addresses as the number of interfaces that it has in the data link layer. Most hosts have a single data link layer interface and thus have a single IPv4 address. However, with the growth of wireless more and more hosts have several data link layer interfaces (for example, an Ethernet interface and a WiFi interface). These hosts are said to be **multihomed**. A multihomed host with two interfaces has thus two IPv4

**multihomed**. A multihomed host with two interfaces has thus two IPv4 addresses.



## Address Assignment #

Appropriate network layer address allocation is key to the efficiency and scalability of the Internet.

A **naive allocation scheme** would be to provide an IPv4 address to each host when the host is attached to the Internet on a first come, first served basis. With this solution, a host in Belgium could have address 2.3.4.5 while another host located in Africa would use address 2.3.4.6. Unfortunately, this would force all routers to maintain a specific route towards all  $\approx 1$  Billion hosts on the Internet, which is **not scalable**. Hence, it's important to minimize the number of routes that are stored on each router.

## Subnetting #

One solution is that routers should only maintain routes towards **blocks of addresses** and not towards individual hosts. For this, blocks of IP addresses are assigned to ISPs. The ISPs assign sub blocks of the assigned address space in a hierarchical manner. **These sub blocks of IP addresses are called subnets**.

A typical subnet groups all the hosts that are part of the same enterprise. An enterprise network is usually composed of several LANs interconnected by routers. A small block of addresses from the Enterprise's block is usually assigned to each LAN. An IPv4 address is composed of two parts:

- A **subnetwork identifier** composed of the high order bits of the address.
- And a **host identifier** encoded in the lower order bits of the address.

This is illustrated in the figure below:

1000101000110000000110100000001

Subnetwork ID

Host ID

The subnetwork and host identifiers inside an IPv4 address

## Address Classes #

When a router needs to forward a packet, it must know the subnet of the destination address to be able to consult its routing table to forward the packet. [RFC 791](#) proposed to use the high-order bits of the address to encode the length of the subnet identifier. This led to the definition of three classes of addresses.

Class	High-order bits	Length of subnet id	Number of networks	Addresses per network
Class A	0	8 bits	128 ( $2^7$ )	16,777,216 ( $2^{24}$ )
Class B	10	16 bits	16,384 ( $2^{14}$ )	65,536 ( $2^{16}$ )
Class C	110	24 bits	2,097,152 ( $2^{21}$ )	256 ( $2^8$ )

In this **classful address scheme**, the range range of the IP addresses in each class are as follows:

- **Class A:** 0.0.0.0 to 127.255.255.255
- **Class B:** 128.0.0.0 to 191.255.255.255
- **Class C:** 192.0.0.0 to 223.255.255.255
- **Class D:** 224.0.0.0 to 239.255.255.255
- **Class E:** 240.0.0.0 to 255.255.255.255.

**Class D** IP addresses are used for multicast, whereas **class E** IP addresses are reserved and can't be used on the Internet. So classes A, B, and C are the ones used for regular purposes.

### Subnet Masks #

Every network that falls into one of these classes has a fixed number of bits in the network part to identify the network itself. The subnet mask ‘masks’ the network part of the IP address and leaves the host part open. So a subnet mask of a class C address could be 203.128.22.0, where the first 3 octets represent the subnet mask and the last octet can be used to identify hosts within this network. For instance, 203.128.22.10 can be one machine on this network.

### Network Address #

The network address is just the address with all the host bits set to 0. So 203.128.22.0 is actually a network address. It is technically not a ‘functional’ address, it’s just used for forwarding table entries.

### Broadcast Address #

The broadcast address of any network is the one where the host bits are all set to 1. So the broadcast address in our example subnet mask is 203.128.22.255. It can be used to broadcast a packet to all devices on a network.

### Default Subnet Masks #

Each class has a default mask as follows where the network ID portion has all 1s and the host ID portion has all 0s.

Class	Default Subnet Mask
Class A	255.0.0.0
Class B	255.255.0.0
Class C	255.255.255.0

However, these three classes of addresses were not flexible enough. A **class A** subnet was **too large** for most organizations and a **class C** subnet was **too small**.

### Variable-Length Subnets #

Flexibility was added by the introduction of **variable-length subnets** in [RFC 1519](#). With variable-length subnets, the subnet identifier can be any size, from 1 to 31 bits. Variable-length subnets allow the network operators to use a subnet that better matches the number of expected hosts that will use the subnet. A subnet identifier or IPv4 prefix is usually represented as **A.B.C.D/p**, where **A.B.C.D is the network address**. It's obtained by concatenating the subnet identifier with a host identifier containing only 0, and **p is the length of the subnet identifier in bits**.

The table below provides examples of variable-length IP subnets.

Subnet	Number of addresses	Lowest address	Highest address
10.0.0.0/30	4	10.0.0.0	10.0.0.3
192.0.2.0/24	256	192.0.2.0	192.0.2.255

### Quick Quiz! #

1

Suppose a new address class has a 4-bit subnet ID with one higher order bit out of the 4. How many addresses per network would that entail?

COMPLETED 0%

1 of 2



---

That's it for this lesson! We'll continue with our discussion of IP address allocation.

# The Internet Protocol: IPV4 Address Allocation

We ended the last lesson with a discussion on variable-length subnets. Let's discuss how blocks of addresses are allocated to organizations, in this lesson.

## WE'LL COVER THE FOLLOWING



- Allocating Blocks of Addresses to Organizations
- Classless Interdomain Routing
  - Who Allocates What?
  - Why CIDR?
  - Longest Prefix Match
- Classless Interdomain Routing Vs. Variable-length Subnets
- Special IPv4 addresses
- Quick Quiz!

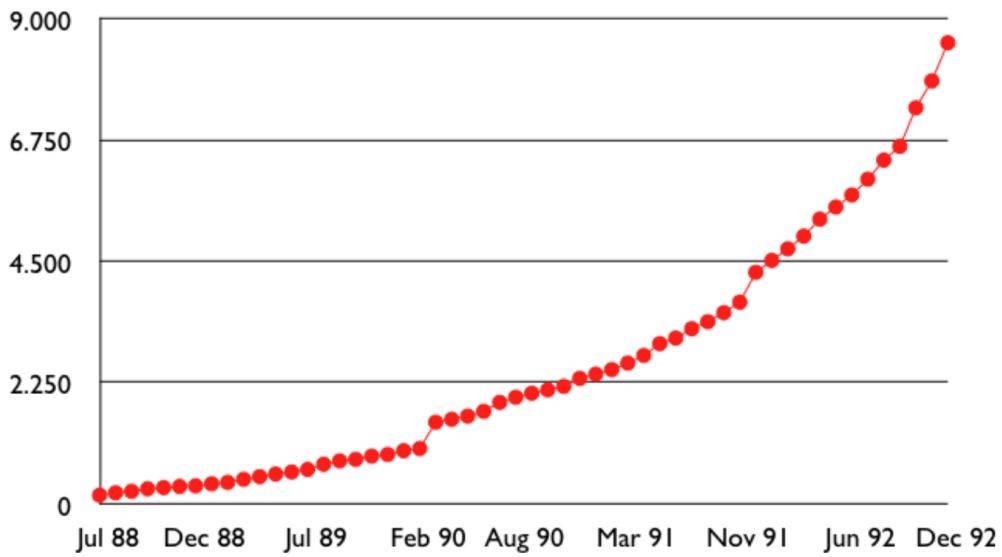
## Allocating Blocks of Addresses to Organizations



A second issue concerning the addresses of the network layer is *how to* allocate blocks of addresses to organizations.

- The first allocation scheme was to allocate class address blocks on a first come, first served basis.
- Large organizations such as **IBM**, **BBN**, as well as **Stanford** or **MIT** were able to obtain one class A address block each.
- However, **most organizations requested class B address blocks** consisting of 65,536 addresses, which was suitable for their size. Unfortunately, there were only 16,384 different class B address blocks. **This address space was being consumed quickly.** Since a disproportionate number of class B address blocks were being used, the

number of entries for class B blocks increased. So the routing tables maintained by the routers were also growing quickly, and some routers had difficulties maintaining all these routes in their limited memory. Hence, **the purpose of address space classes was being defeated.**



Evolution of the size of the routing tables on the Internet (Jul 1988-Dec 1992)

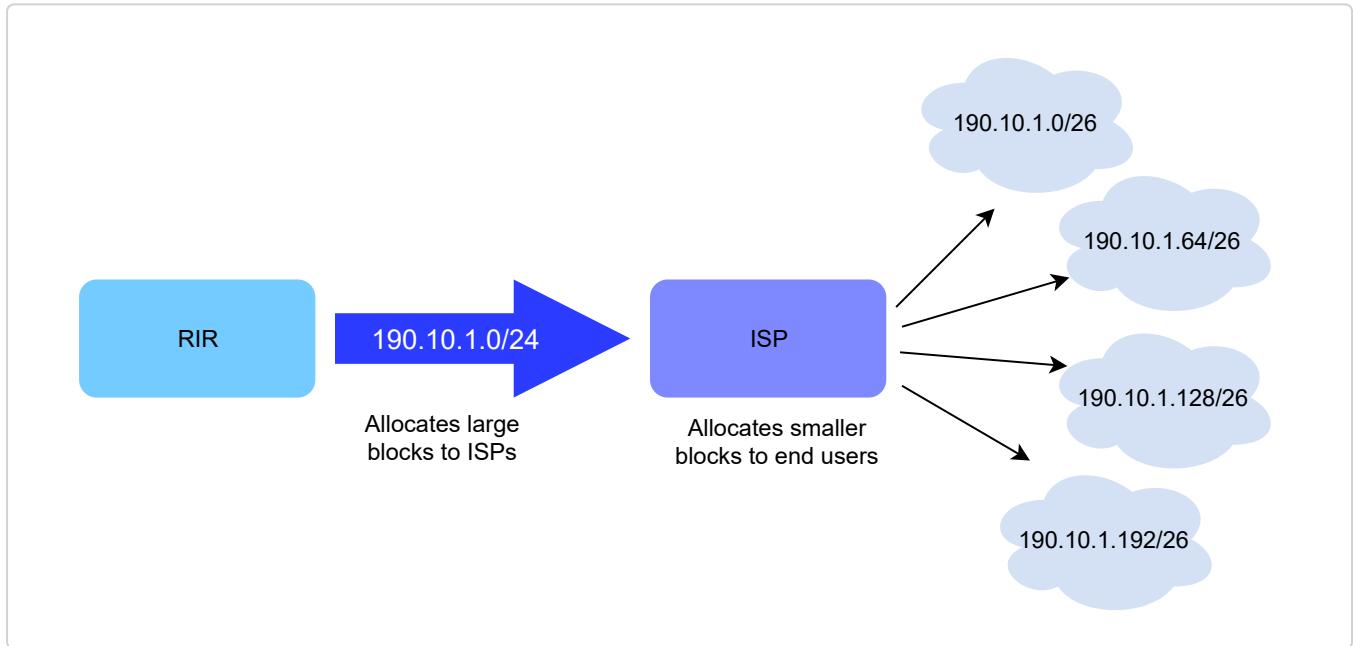
## Classless Interdomain Routing #

Faced with these two problems, the Internet Engineering Task Force decided to develop the **Classless Interdomain Routing (CIDR)** architecture [RFC 1518](#). This architecture allows IP routing to scale better than class-based architecture. CIDR contains **three** important modifications over class-based architecture:

1. IP address classes are deprecated. All IP equipment must use and support **variable-length subnets**.
2. IP address blocks are no longer allocated on a first come, first served basis. Instead, CIDR introduces a **hierarchical address allocation scheme**. The main draw-back of the first come, first served address block allocation scheme was that neighboring address blocks were allocated to very different organizations and conversely, very different address blocks were allocated to similar organizations.
3. IP routers must use **longest-prefix match** when they look up a destination address in their forwarding table.

## Who Allocates What? #

With CIDR, address blocks are allocated by **Regional IP Registries (RIR)** in an aggregatable manner. A RIR is responsible for a large block of addresses and a region. For example, [RIPE](#) is the RIR that is responsible for Europe. A RIR allocates smaller address blocks from its large block to **Internet Service Providers (ISPs)**. ISPs then allocate smaller address blocks to their customers.



## Why CIDR? #

The main advantage of this hierarchical address block allocation scheme is that it allows the routers to maintain fewer routes. For example, consider the address blocks that were allocated to some Belgian universities as shown in the table below:

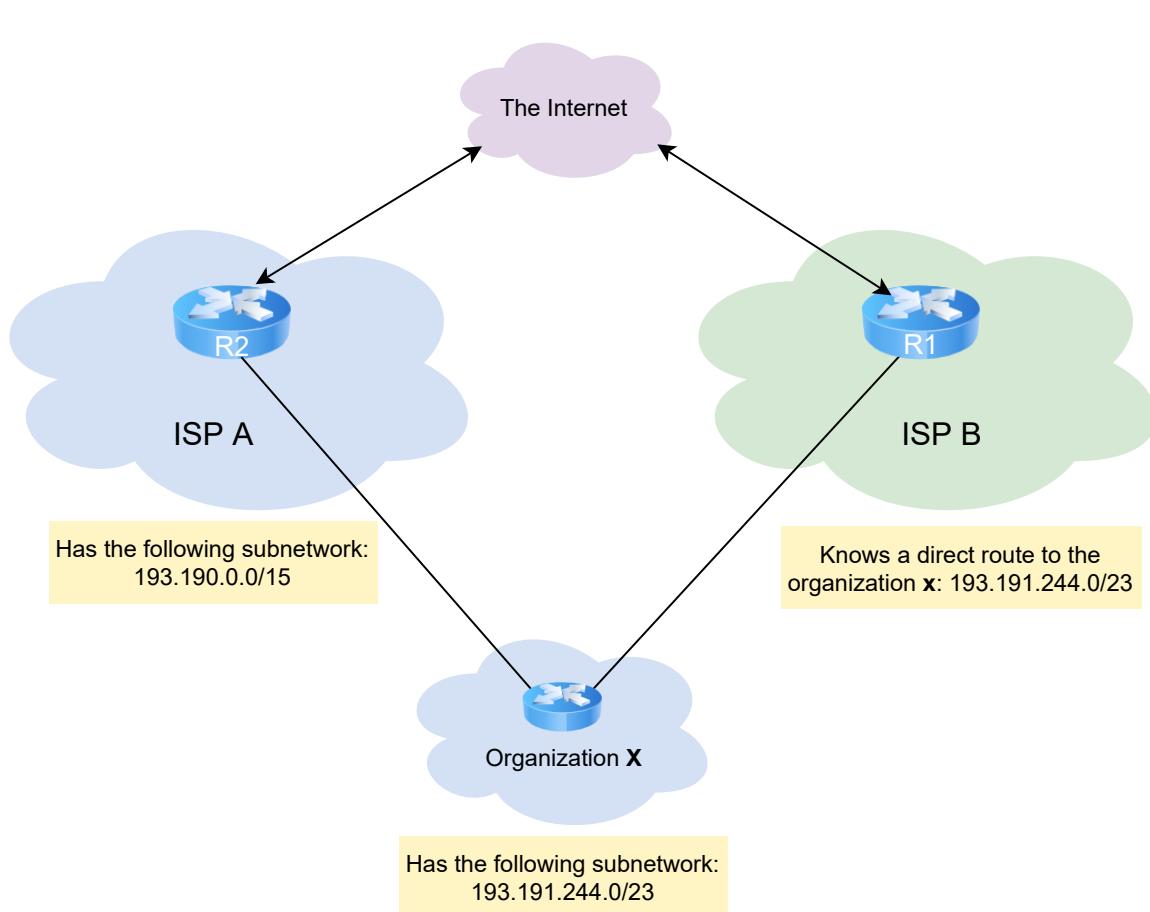
Address block	Organization
130.104.0.0/16	Universite catholique de Louvain
134.58.0.0/16	Katholieke Universiteit Leuven
138.48.0.0/16	Facultes universitaires Notre-Dame de la Paix
139.165.0.0/16	Universite de Liege

Suppose that these universities are all connected to the Internet exclusively via **ISP A**. As each university has been allocated a different address block, the routers of ISP A must announce **one route for each university**, and all routers on the Internet must maintain a route towards each university.

In contrast, suppose all the high schools and the government institutions that are connected to the Internet via ISP A are assigned one block: 193.190.0.0/15 after the introduction of CIDR. Thanks to this, ISP A has **one route for all high schools and government institutions**.

## Longest Prefix Match #

However, there is one difficulty with the aggregatable variable length subnets used by CIDR. Consider, for example, a government institution **X** that uses the 193.191.244.0/23 address block. Assume that in addition to being connected to the Internet via ISP A, **X wants to be connected to another ISP**. X's network is then said to be *multihomed*. This is shown in the figure below.



With such a multihomed network, routers in the general Internet would have **two routes** towards an address in X such as 193.191.245.88:

- One route via ISP A (193.190.0.0/15)
- One route via ISP B (193.191.244.0/23).

Both routes match IPv4 address 193.192.145.88. Since [RFC 1519](#) when a router knows several routes towards the same destination address, it must forward packets along the route with the longest prefix length. In our example:

- Our target IP address, 193.191.245.88 in binary is  
11000001.10111111.11110101.01011000
- 193.190.0.0 in binary is 11000001.10111110.00000000.00000000
- 193.191.244.0 in binary is 11000001.10111111.11110100.00000000
- Hence, 193.191.245.88 and 193.190.0.0 have a 16-bit matching prefix.
- Whereas 193.191.245.88 and 193.191.244.0 have a 23-bit matching prefix.
- In the case of 193.191.245.88, the route that will be taken will be via ISP B.

This forwarding rule is called the **longest prefix match or the more specific match**. All IPv4 routers implement this forwarding rule.

## Classless Interdomain Routing Vs. Variable-length Subnets #

Variable-length subnets steal bits from the host portion of the IP address. Classless interdomain routing also allows aggregation of smaller subnets into larger ones by making less specific subnet masks. For example, 190.10.1.0/24, 190.10.2.0/24, 190.10.3.0/24 and 190.10.4.0/24 can be summarized into 190.10.0.0/21. This reduces the number of entries that a router advertises, thereby controlling the size of the routing tables in the core of the Internet.

Furthermore, in Variable-length subnets the default subnet mask of the classes is strictly **extended**, whereas in CIDR, classes do not exist at all. So the ‘default’ length can be **extended or reduced**. Therefore, variable-length subnets are used if someone needs fewer addresses generally, whereas CIDR is for reducing routing table entries.

## Special IPv4 addresses #

Most unicast IPv4 addresses can appear as source and destination addresses in packets on the global Internet. It's worth noting though, that some blocks of IPv4 addresses have a special usage, as described in [RFC 5735](#). These include:

- **0.0.0.0/8**: reserved for self-identification. A common address in this block is **0.0.0.0**, that we saw being used in the [previous chapter!](#)
- **127.0.0.0/8**, reserved for **loopback addresses**. Each IPv4 host has a loopback interface (that's not attached to a data link layer). By convention, IPv4 address **127.0.0.1** is assigned to this interface as we saw in the previous chapter. This allows processes running on a host to use TCP/IP to contact other processes running on the same host. This is very useful for testing purposes. Furthermore, loopback interfaces can not be down. If the device is up, so are its loopback interfaces. Yes, the plural is intentional too. You can configure as many loopback interfaces as you want. In such a case, the loopback interfaces can be assigned different IP addresses. Anyway, a loopback interface address is used as a router identifier when configuring some of the routing protocols. We want the routing process to keep running even if some of the physical interfaces go down. The loopback interface(s) provide the desired stability.
- **10.0.0.0/8**, **172.16.0.0/12** and **192.168.0.0/16** are reserved for private networks that are not directly attached to the Internet. These addresses are often called private addresses.
- **169.254.0.0/16** is used for link-local addresses. Some hosts use an address in this block when they're connected to a network that does not allocate addresses as expected.

## Quick Quiz! #

1

What would the next hop for the IP address 205.135.3.2 be based on the following routing table?

Destination	Next-hop
205.135.0.0/16	R3
205.135.3.0/24	R1
205.0.0.0/8	R2
Any other	R4

If entries for a certain IP address are not available (no prefix bits match), then they get forwarded to a certain default router.

COMPLETED 0%

1 of 2



In the next lesson, we'll look at IPV4 packets!

# The Internet Protocol: IPv4 Packets

Now that we have clarified the allocation of IPv4 addresses and the utilization of the longest prefix match to forward IPv4 packets, we can have a more detailed look at IPv4 by starting with the format of the IPv4 packets.

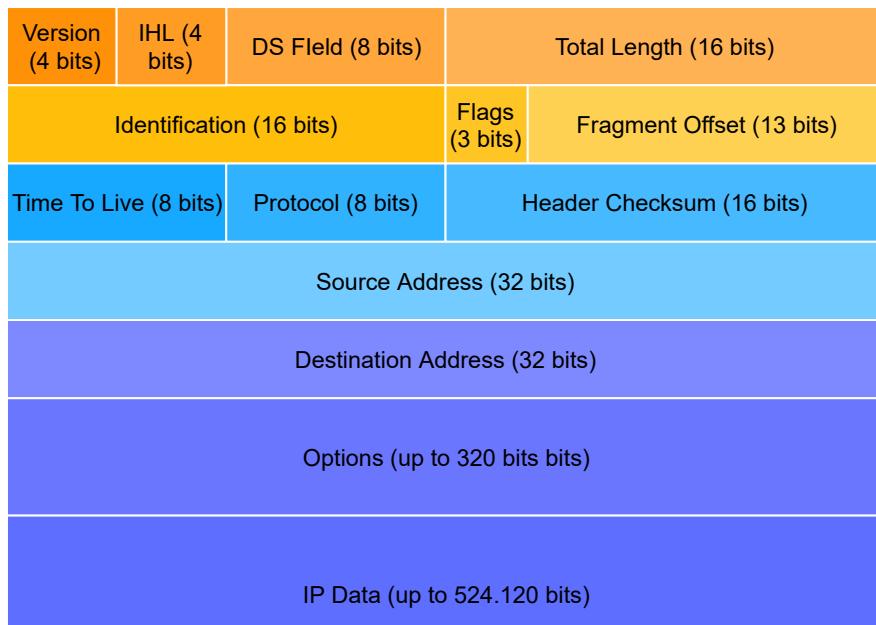
## WE'LL COVER THE FOLLOWING



- IPv4 Packet Header
- Fields of The Header
- Handling Forwarding Loops with TTL
- Handling Data Link Layer Heterogeneity
- Quick Quiz!

The IPv4 packet format was defined in [RFC 791](#). Apart from a few clarifications and some backward compatibility changes, the IPv4 packet format did not change significantly since the publication of [RFC 791](#). All IPv4 packets use a 20-byte header as shown below. Some IPv4 packets contain an optional header extension that is described later.

## IPv4 Packet Header #



# Fields of The Header #

The main fields of the IPv4 header are:

- A 4 bit **version** that indicates the version of IP used to build the header.  
Using a version field in the header allows the network layer protocol to evolve.
- A 4 bit **IP Header Length (IHL)** that indicates the length of the IP header in 32-bit words. This field allows IPv4 to use options if required, but as it is encoded as a 4 bits field, the IPv4 header cannot be longer than 64 bytes.
- An 8 bit **DS field** that is used for Quality of Service.
- A 16 bit **length field** that indicates the total length of the entire IPv4 packet (header and payload) in bytes. This implies that an IPv4 packet cannot be longer than 65535 bytes.
- **Identification** every packet has an identification number which is useful when reassembling and fragmenting a packet.
- **Flags**. There are three flags in IP headers. We'll discuss their usage in the [next lesson](#):
  - Don't Fragment
  - More Fragments
  - Reserved (must be zero)
- **Fragment Offset**: This is useful when reassembling a packet from its fragments. More details can be found in the [next lesson](#).
- **Time To Live**: This number is decremented at each hop. When it becomes 0, the packet is considered to have been in the network for too long and is dropped.
- An 8 bits **Protocol field** that indicates the transport layer protocol that must process the packet's payload at the destination. Common values for this field are 6 for TCP and 17 for UDP.
- A 16 bit **checksum** that protects only the IPv4 header against transmission errors.
- A 32 bit **source address field** that contains the IPv4 address of the source

host.

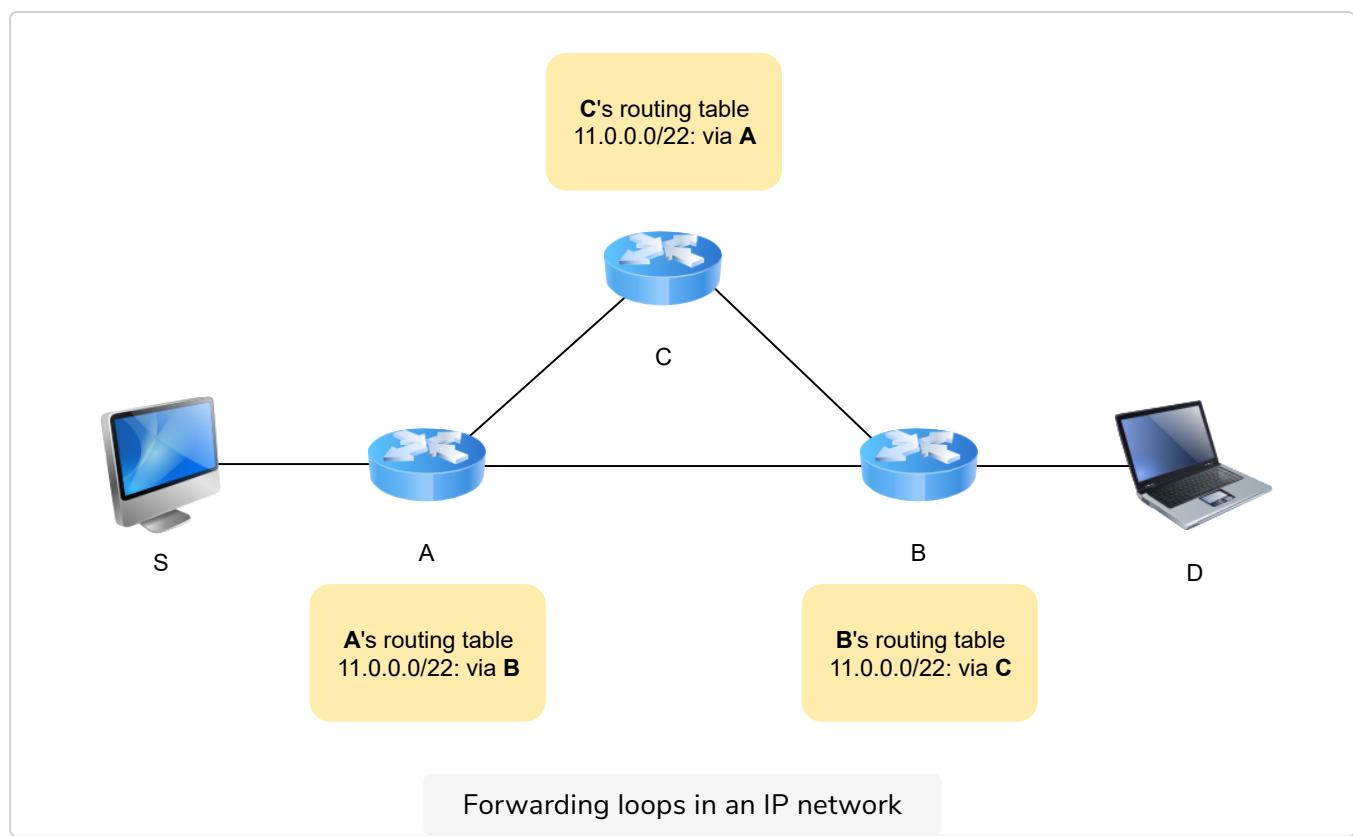
- A 32 bit **destination address field** that contains the IPv4 address of the destination host.
- **Options** this field is not used very often. It's often used to test out experimental features.
- **IP Data:** They payload. This payload is not part of the checksum.

The other fields of the IPv4 header are used for very specific purposes. We'll look at a few in this lesson.

## Handling Forwarding Loops with TTL #

The first is the 8 bit **Time To Live (TTL)** field. This field is used by IPv4 to avoid the risk of having an IPv4 packet caught in an infinite loop due to a transient or permanent error in routing tables.

Consider, for example, the forwarding loop depicted in the figure below. Destination D uses address 11.0.0.56. If S sends a packet towards this destination, the packet is forwarded to router B which forwards it to router C that forwards it back to router A, and so on.



The **TTL field** of the IPv4 header ensures that **even if there are forwarding loops in the network, packets will not loop forever**.

**loops in the network, packets will not loop forever.**

Hosts send their IPv4 packets with a positive TTL (usually 64 or more). When a router receives an IPv4 packet, it first **decrements the TTL by one**. If the **TTL becomes 0, the packet is discarded** and a message is sent back to the packet's source.

## Handling Data Link Layer Heterogeneity #

A second problem for IPv4 is the heterogeneity of the data link layer. IPv4 is used above many very different data link layers. Each of which has its own characteristics. For example, each data link layer is characterized by a **maximum frame size** or **Maximum Transmission Unit (MTU)**. The MTU of an interface is the largest IPv4 packet (including header) that it can send. The table below provides some common MTU sizes.

Data link layer	MTU
Ethernet	1500 bytes
IEEE 802.11 WiFi	2304 bytes
Token Ring (802.15.4)	4464 bytes
FDDI	4352 bytes

## Quick Quiz! #

Q

An Internet Protocol with the version number 15 is possible in theory.

COMPLETED 0%

1 of 1



In the next lesson, we'll study IPv4 fragmentation!

# The Internet Protocol: IPv4 Packet Fragmentation & Reassembly

In this lesson, we'll study IP version 4 fragmentation and reassembly.

## WE'LL COVER THE FOLLOWING

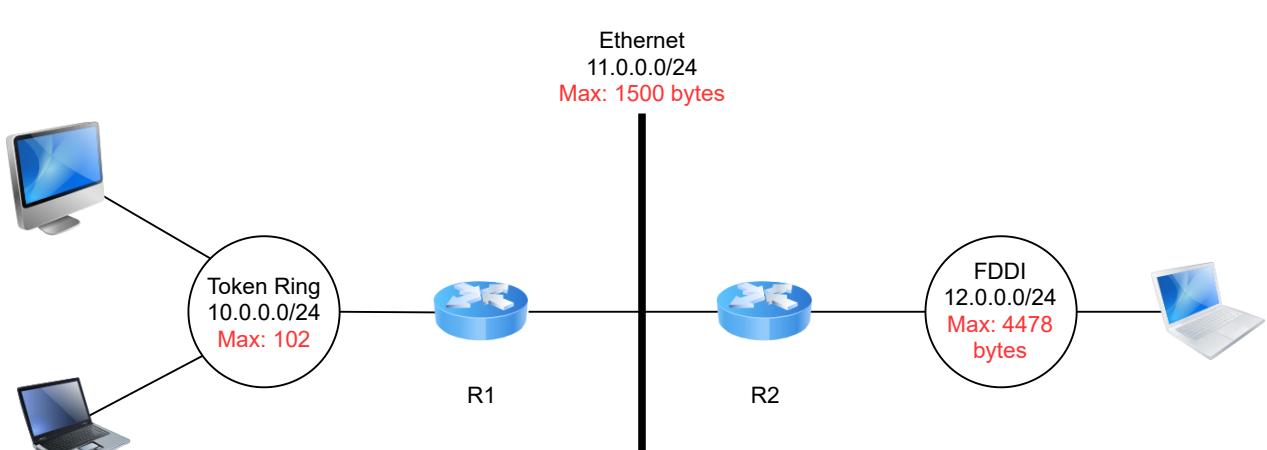


- Why Fragmentation?
- How Fragmentation Works
- How Reassembly Works
  - Handling Loss & Duplicates
- Quick Quiz!

## Why Fragmentation? #

Although IPv4 packets can be as big as 64kB, few data link layer technologies can send a 64 KB IPv4 packet inside a frame.

Furthermore, as in the figure below, if the host on the FDDI network abides by its own data link layer's maximum packet size of 4478 bytes, the resulting data link layer frame would violate the maximum frame size of the Ethernet between routers R1 and R2. Hence, a host may end up sending a packet that is too large for a data link layer technology used by (an) intermediate router(s).



To solve these problems, IPv4 includes a **packet fragmentation and reassembly mechanism** in both hosts and intermediate routers. In IPv4, fragmentation is completely performed in the IP layer and a large IPv4 packet is fragmented into two or more IPv4 packets (called fragments).

## How Fragmentation Works #

The IPv4 fragmentation mechanism relies on four fields of the IPv4 header:

- **Length**
- **Identification**
- The **flags**
  - **More fragments**
  - **Don't Fragment (DF)**. When this flag is set, it indicates that the **packet cannot be fragmented**
- **Fragment Offset.**

The **basic operation of IPv4 fragmentation** is as follows:

- A large packet is fragmented into two or more fragments where the size of all fragments, except the last one, is equal to the Maximum Transmission Unit of the link used to forward the packet.
- The Length field in each fragment indicates the length of the payload and the header **of the fragment**.
- Each IPv4 packet contains a 16 bit **Identification** field. When a packet is fragmented, the Identification of the large packet is copied in all fragments to allow the destination to reassemble the received fragments together.
- In each fragment, the **Fragment Offset** indicates, in units of 8 bytes, the position of the payload of the fragment in the payload of the original packet.
- When the **Don't Fragment (DF)** flag is set, it indicates that the **packet**

**cannot be fragmented.**

- Finally, the **More fragments** flag is set only in the last fragment of a large packet.

## How Reassembly Works #

The fragments of an IPv4 packet **may arrive at the destination in any order** since each fragment is forwarded independently in the network and may follow different paths. Furthermore, some fragments **may be lost and never reach the destination.**

The **reassembly algorithm** used by the destination host is roughly as follows:

1. First, the destination can verify whether a received IPv4 packet is a fragment or not by checking the value of the **More fragments** flag and the **Fragment Offset**. If the Fragment Offset is set to 0 and the More fragments flag is reset, the received packet has not been fragmented. Otherwise, the packet has been fragmented and must be reassembled.
2. The reassembly algorithm relies on the Identification field of the received fragments to associate a fragment with the corresponding packet being reassembled.
3. Furthermore, the Fragment Offset field indicates the position of the fragment payload in the original unfragmented packet.
4. Finally, the packet with the More fragments flag reset allows the destination to determine the total length of the original unfragmented packet.

## Handling Loss & Duplicates #

Note that the reassembly algorithm must **deal with the unreliability of the IP network**: fragments may be duplicated or may never reach the destination. The destination can easily **detect fragment duplication with the Fragment Offset**.

**To deal with fragment losses, the reassembly algorithm must bind the time during which the fragments of a packet are stored in its buffer while the packet is being reassembled.** This can be implemented by starting a timer

when the first fragment of a packet is received. If the packet has not been reassembled upon expiration of the timer, all fragments are discarded and the packet is considered to be lost.

## Quick Quiz! #

1

Given the sample MTU size of 200 and an IP datagram of size 1999, how many fragments will be created?

COMPLETED 0%

1 of 9



In the next lesson, we'll study an error-reporting protocol: **ICMP**.

# The Internet Control Message Protocol (ICMP)

In this lesson, we'll study ICMP, the network layer's error reporting protocol.

## WE'LL COVER THE FOLLOWING ^

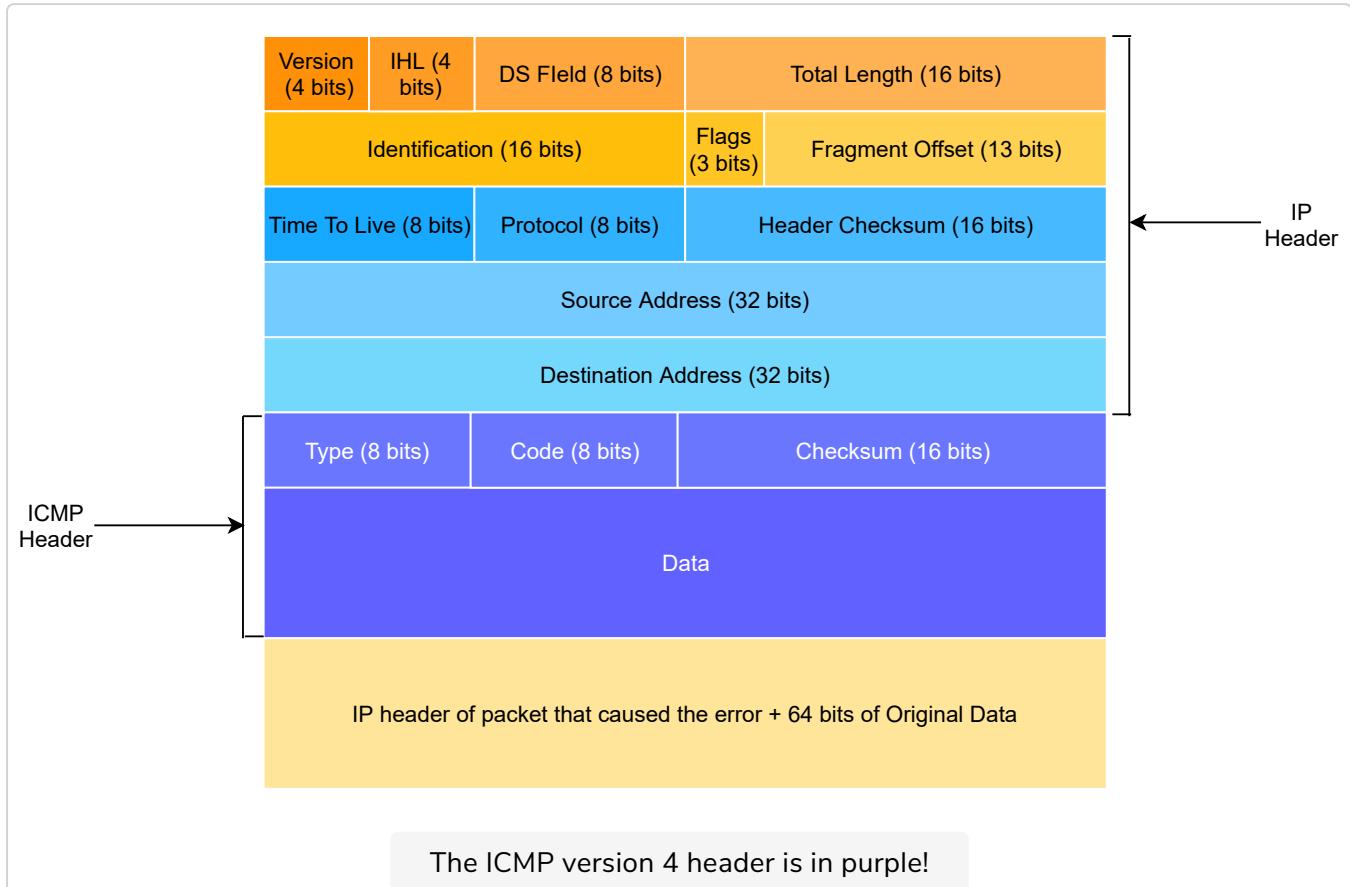
- What Is ICMP?
- ICMP Headers
- ICMP Messages
- Quick Quiz!

## What Is ICMP? #

It's sometimes necessary for intermediate routers or destination hosts to **inform the sender of a packet about any problems** that occur while processing it. In the TCP/IP protocol suite, this reporting is done by the **Internet Control Message Protocol (ICMP)**. ICMP is defined in [RFC 792](#).

## ICMP Headers #

ICMP messages are carried as the payload of IP packets (the protocol value reserved for ICMP is 1). An ICMP message is composed of an 8-byte header and a variable-length payload that usually contains the first bytes of the packet that triggered the transmission of the ICMP message.



In the ICMP header (purple in the diagram above):

- The **Type** and **Code** fields indicate the type of problem that was detected by the sender of the ICMP message.
- The **Checksum** protects the entire ICMP message against transmission errors
- The **Data field** contains additional information for some ICMP messages.

## ICMP Messages #

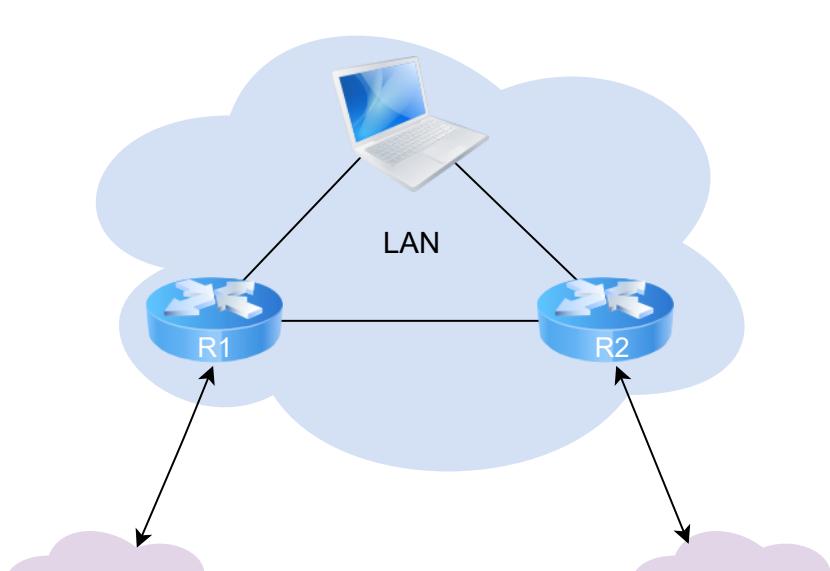
The main types of ICMP messages are:

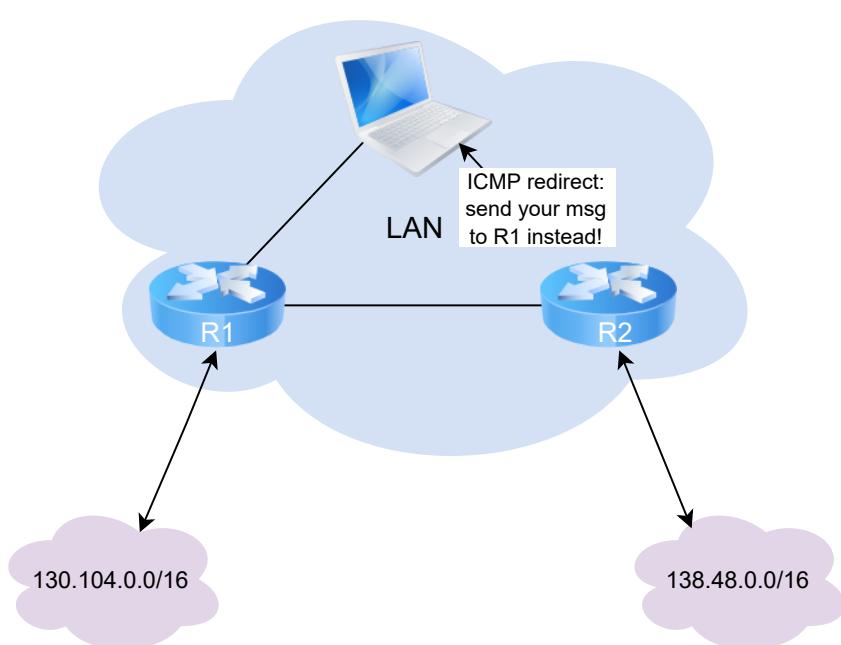
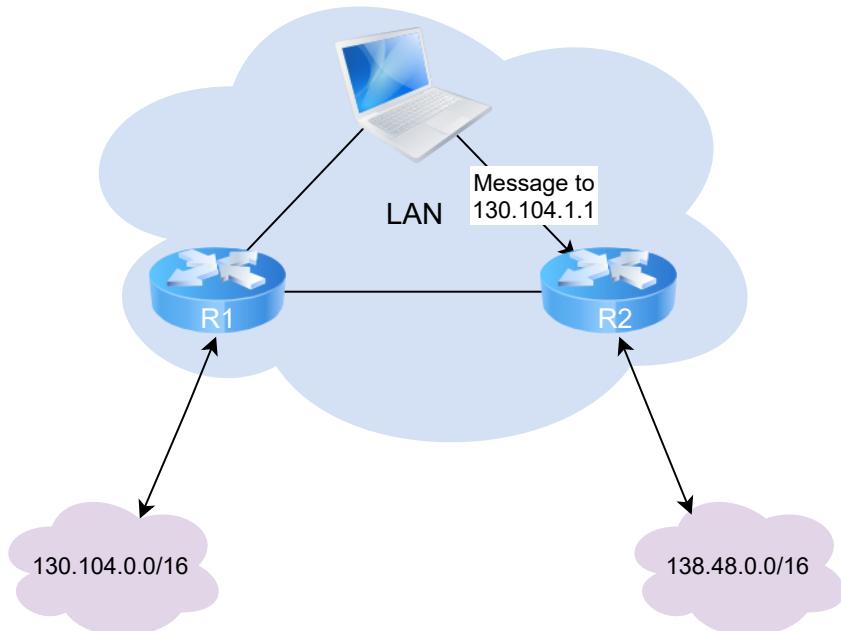
1. **Destination unreachable:** A destination unreachable ICMP message is sent when a packet can't be delivered to its destination due to routing problems. Different types of unreachability are distinguished:
  - *Network unreachable:* This ICMP message is sent by a router that does not have a route for the subnet containing the destination address of the packet.

- *Host unreachable*: This ICMP message is sent by a router that is attached to the subnet that contains the destination address of the packet, but this destination address cannot be reached at this time.
- *Protocol unreachable*: This ICMP message is sent by a destination host that has received a packet, but doesn't support the transport protocol indicated in the packet's Protocol field.
- *Port unreachable*: This ICMP message is sent by a destination host that has received a packet destined to a port number, but no server process is bound to this port.

2. **Fragmentation needed**: This ICMP message is sent by a router that receives a packet with the Don't Fragment flag set that is larger than the MTU of the outgoing interface.
3. **Redirect**: This ICMP message can be sent when there are two routers on the same LAN.

- *Example*: Consider a LAN with one host and two routers: R1 and R2. Assume that R1 is also connected to subnet 130.104.0.0/16 while R2 is connected to subnet 138.48.0.0/16. If a host on the LAN sends a packet towards 130.104.1.1 to R2, R2 needs to forward the packet again on the LAN to reach R1. This is not optimal, since the packet is sent twice on the same LAN. In this case, R2 could send an ICMP Redirect message to the host to inform it that it should have sent the packet directly to R1. This allows the host to send the other packets to 130.104.1.1 directly via R1.





**4. Parameter problem:** This ICMP message is sent when a router or a host receives an IP packet containing an error (e.g. an invalid option).

- **Source quench:** It was envisioned that a router would send this ICMP message when it had to discard packets due to congestion. However

message when it had to discard packets due to congestion. However, sending ICMP messages in case of congestion was not the best way to reduce congestion. And since the inclusion of a congestion control scheme is in TCP, this ICMP message has been deprecated.

## 5. Time Exceeded:

There are two types of Time Exceeded ICMP messages.

- *TTL exceeded*: A TTL exceeded message is sent by a router when it discards an IPv4 packet because its TTL reached 0 to the sender of the packet.
- *Reassembly time exceeded*: This ICMP message is sent when a destination has been unable to reassemble all the fragments of a packet before the expiration of its reassembly timer.

## 6. Echo request & Echo reply:

These ICMP messages are used by the `ping(8)` network debugging software. Let's have a look at `ping` next.

## Quick Quiz! #

1

An ICMP destination unreachable message is returned when \_\_\_\_.

COMPLETED 0%

1 of 3



In the next lesson, we'll send real ICMP messages with command-line tools like `ping` and `traceroute`!



# Exercise: Sending ICMP Messages With Ping & Traceroute

In this lesson, we'll look at real live ICMP packets with ping and traceroute!

## WE'LL COVER THE FOLLOWING ^

- [Ping](#)
- [Traceroute](#)
  - How It Works
  - Usage
  - Sample Output #1
  - Sample Output #2

## Ping #

When a client sends ICMP echo messages ([ping](#)), it sets a certain value in the TTL field and starts a timer. An echo server software running on the destination returns an ICMP echo reply message. Since the TTL value is decremented at each hop, the [ping](#) client can know the number of hops traversed by the packets. Also, when it receives the echo reply, it stops the timer and calculates the round trip time. There is a maximum value for the round trip time and when it's exceeded, the echo message is declared lost.

[ping](#) is also often used by network operators to verify that a given IP address is reachable.

Sample usage of [ping](#) is shown below.

• Terminal



## Traceroute #

Another very useful debugging tool is [traceroute](#). The [traceroute man page](#)

Another very useful debugging tool is `traceroute`. The [traceroute man page](#) describes this tool as “print the route packets take to network host.”

## How It Works #

Traceroute uses the TTL exceeded ICMP messages to discover the intermediate routers on the path towards a destination. The principle behind traceroute is very simple.

- When a router receives an IP packet whose TTL is set to 1, it decrements the TTL and is forced to return a TTL exceeded ICMP message to the sending host.
- To discover all routers on a network path, a simple solution is to first send a packet whose TTL is set to 1, then a packet whose TTL is set to 2, and so on. When the TTL is set to 1, the first router on the path returns a TTL expired packet, which is how its IP address can be discovered. When TTL is set to 2, the second router on the path returns a TTL expired packet, and so on. In this way, we are able to discover IP addresses of all routers on the path to the destination from the sending host. `traceroute` actually sends **three** packets with each TTL value.

Run the following call to `traceroute` to get a traceroute output of a path to [ietf.org](#) from one of Educative’s servers.

## Usage #

Terminal



### Sample Output #1 #

```
traceroute to www.ietf.org (104.20.1.85), 30 hops max, 60 byte packets
 1  216.239.63.174 27.718 ms  27.838 ms  27.998 ms
 2  108.170.244.16 157.181 ms  157.195 ms  157.714 ms
 3  141.101.73.2
```



Here’s what some simple `traceroute` output may look like. Notice that the output is organized in rows and columns where each hop is represented by one row. Here’s what each column means:

Hop	IP Address	RTT 1	RTT 2	RTT 3
-----	------------	-------	-------	-------

Number	IP Address	RTT 1	RTT 2	RTT 3
1	216.239.63. 174	27.718	27.838	27.998
2	108.170.244 .16	157.181	157.195	157.714
3	141.101.73. 2	11.648	11.650	11.721

The `traceroute` output above shows a 3-hop path (in the instance of writing this course - the number of hops and their IP addresses may be different now) between a host at Educative and one IETF's servers. For each hop, traceroute provides the IPv4 address of the router that sent the ICMP message and exactly **three** measured round-trip-times between the source and this router.

## Sample Output #2 #

```
traceroute to www.ietf.org (104.20.1.85), 30 hops max, 60 byte packets
 1  216.239.63.174 (216.239.63.174)  27.718 ms  72.14.232.108 (72.14.232.108)  11.264 ms  216.2
 2  108.170.244.16 (108.170.244.16)  157.181 ms  157.195 ms  108.170.243.196 (108.170.243.196)
 3  141.101.73.2 (141.101.73.2)  11.648 ms  104.20.1.85 (104.20.1.85)  11.610 ms  141.101.73.2
```

You may also get something slightly more complicated like the above. Here, there is more than one next-hop each packet can take. For example, the first hop shows 2 different IP addresses:

```
 1  216.239.63.174 (216.239.63.174)  27.718 ms  72.14.232.108 (72.14.232.10
 8)  11.264 ms  216.239.63.174 (216.239.63.174)  27.598 ms
```

So there are multiple routes towards the destination and probes are sent to each possible next hop.



**Note** Some routers are configured by their administrators not to respond to ICMP messages. In such cases, traceroute shows \* \* \* when it times out waiting for the response. Also, by default, the traceroute utility on our platform goes to a maximum of 30 hops.

on our platform goes to a maximum of 50 hops.

---

In the next lesson, we'll study IPv4 Data Link Layer Address Resolution

# Address Resolution Protocol (ARP)

In this lesson, we'll discuss how data link layer addresses are resolved in an IPv4 network

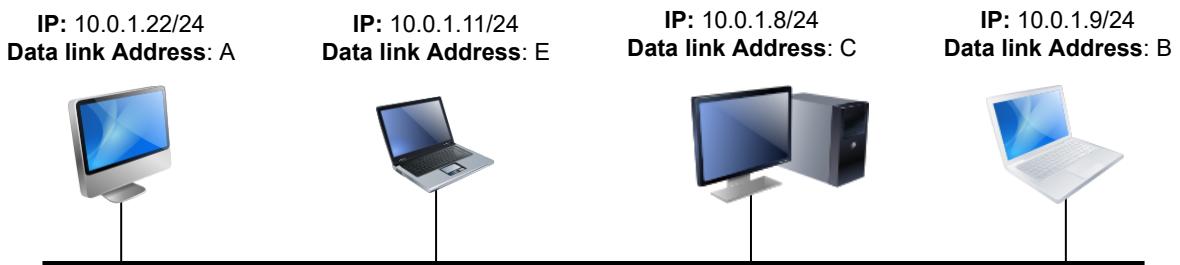
## WE'LL COVER THE FOLLOWING ^

- Introduction
  - How It Works
- Quick Quiz!

## Introduction #

While end hosts may use IP addresses to communicate with each other, the underlying data link layer uses its own naming schemes. So, end host interfaces have unique data link layer addresses. In order to get data to a host, a mechanism for converting IP addresses to the underlying data link layer address is needed. This entails that all **sending hosts must know the data link-layer address of their destination hosts** in order to send them a packet.

For example, the figure below shows four hosts attached to the same LAN configured with IPv4 addresses in the 10.0.1.0/24 subnet and data link layer addresses represented as a single character. In this network, if host 10.0.1.22/24 wants to send an IPv4 packet to the host with address 10.0.1.8, it must know that the data link layer address of this host is C.



A simple LAN

While manual configuration of the data link address of each host is possible in small networks such as the one above, it does not scale. Hence, IPv4 hosts and routers must be able to *automatically* obtain the data link layer address corresponding to any IPv4 address on the same LAN. This is the objective of the **Address Resolution Protocol (ARP)** defined in [RFC 826](#). ARP is a data link layer protocol and relies on the ability of the data link layer service to broadcast a frame to all devices attached to the same LAN.

## How It Works #

The easiest way to understand the operation of ARP is to consider the simple network shown above and:

- Assume that host 10.0.1.22/24 needs to send an IPv4 packet to host 10.0.1.8. To do so, the sending host must find the data link layer address that is attached to host 10.0.1.8.
- Each IPv4 host maintains an **ARP cache** that contains all mappings between IPv4 addresses and data link layer addresses that it knows.
- The sender, 10.0.1.22, first consults its ARP cache. As the cache does not contain the requested mapping, **the sender sends a broadcast ARP query frame on the LAN**.
- The frame contains:
  - The **data link layer address of the sender**, which is A in our example,
  - The **IPv4 address of the destination**, which is 10.0.1.8 in the example.
- This broadcast frame is received by all devices on the LAN. Every host upon receiving the ARP query inserts an entry for the sender's IP address and data link layer address into their ARP cache.
- Every host on the LAN segment receives the ARP query however, **only the host that owns the requested IPv4 address replies by returning a unicast ARP reply frame with the requested mapping**.

- Upon reception of this reply, **the sender updates its ARP cache** and sends the IPv4 packet by using the data link layer service.

Note that to deal with devices that move or whose addresses are reconfigured, most ARP implementations remove the cache entries that have not been used for a few minutes. Some implementations also revalidate ARP cache entries from time to time by sending ARP queries.

## Quick Quiz! #

1

Suppose a host, **A**, sends an ARP request to a host **C**. Immediately after that, another host, **D**, wants to send a packet to host A. Will it send an ARP request assuming that there has not been any traffic on the LAN yet?

COMPLETED 0%

1 of 2



In the next lesson, we'll look at the dynamic host configuration protocol.

# Dynamic Host Configuration Protocol (DHCP)

In this lesson, we'll discuss how IP addresses are assigned to devices on the network.

## WE'LL COVER THE FOLLOWING ^

- Introduction
- How It Works
- Quick Quiz!

## Introduction #

In the early days of the Internet, IP addresses were manually configured on both hosts and routers and almost never changed. However, this manual configuration can be complex and often causes errors that can be difficult to debug.

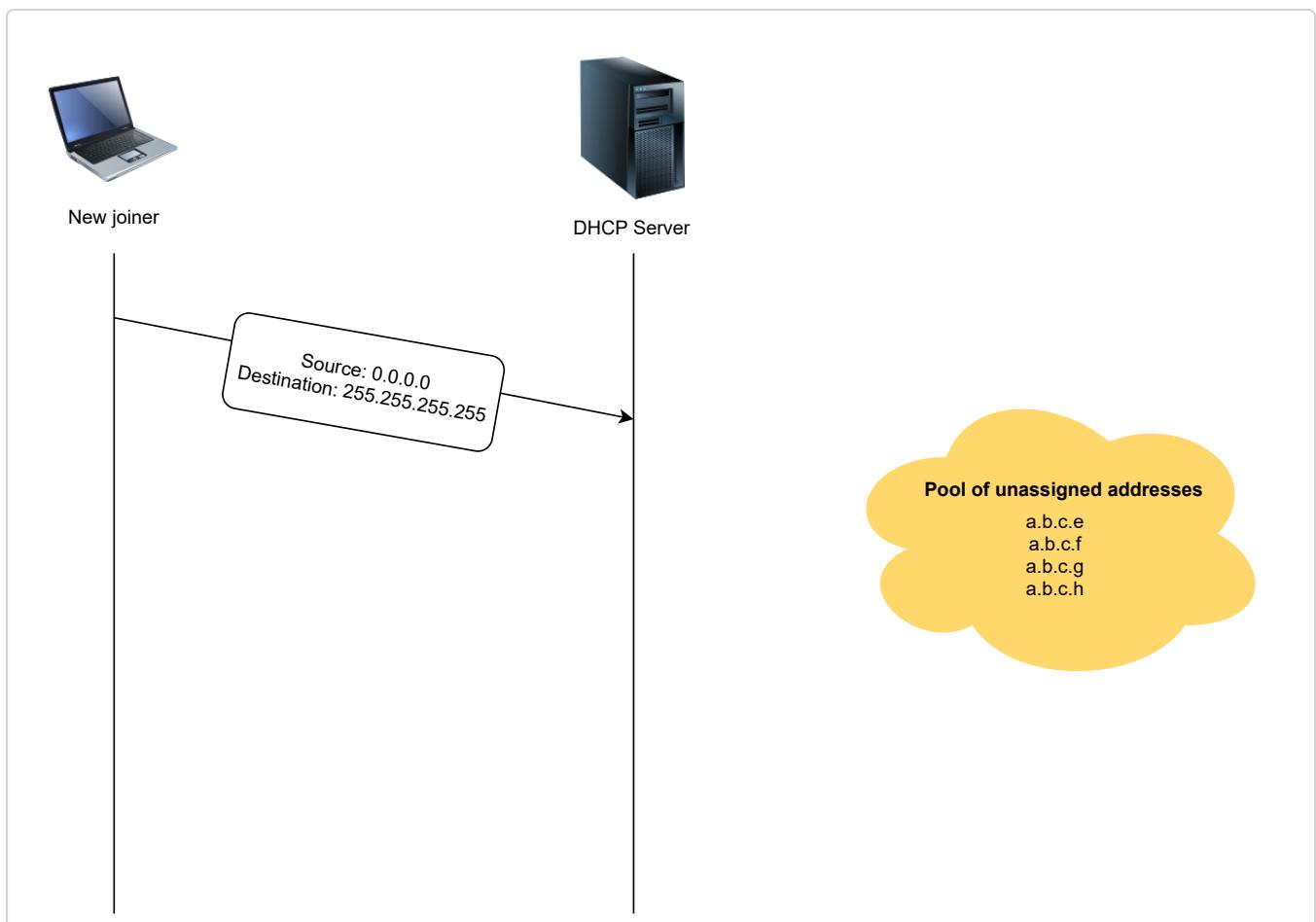
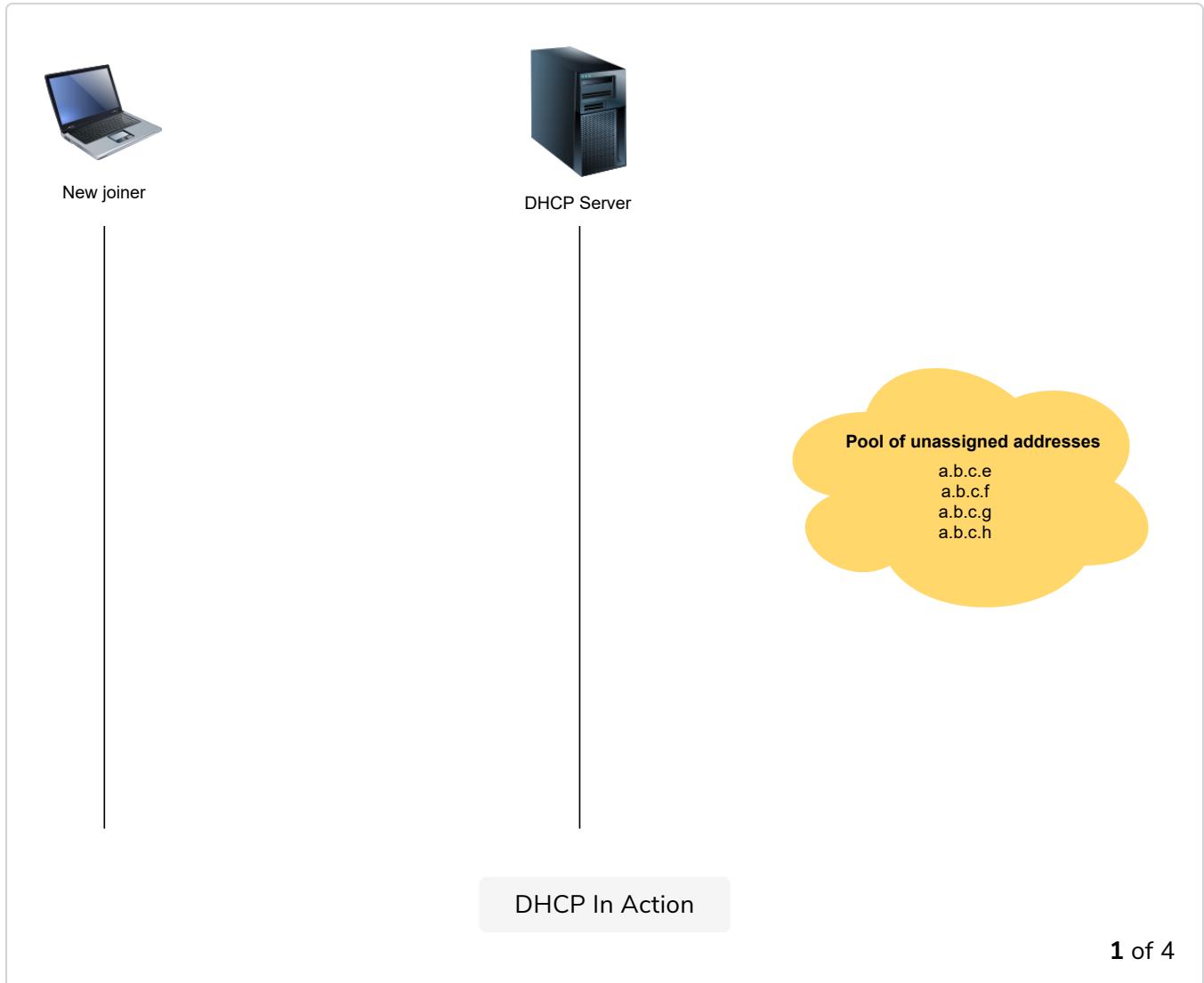
To ease the attachment of hosts to subnets, most networks now support the **Dynamic Host Configuration Protocol (DHCP)** [RFC 2131](#). DHCP allows a host to automatically retrieve its assigned IPv4 address. A DHCP client actually can retrieve other network parameters too, including subnet mask, default gateway and DNS server addresses from the DHCP server.

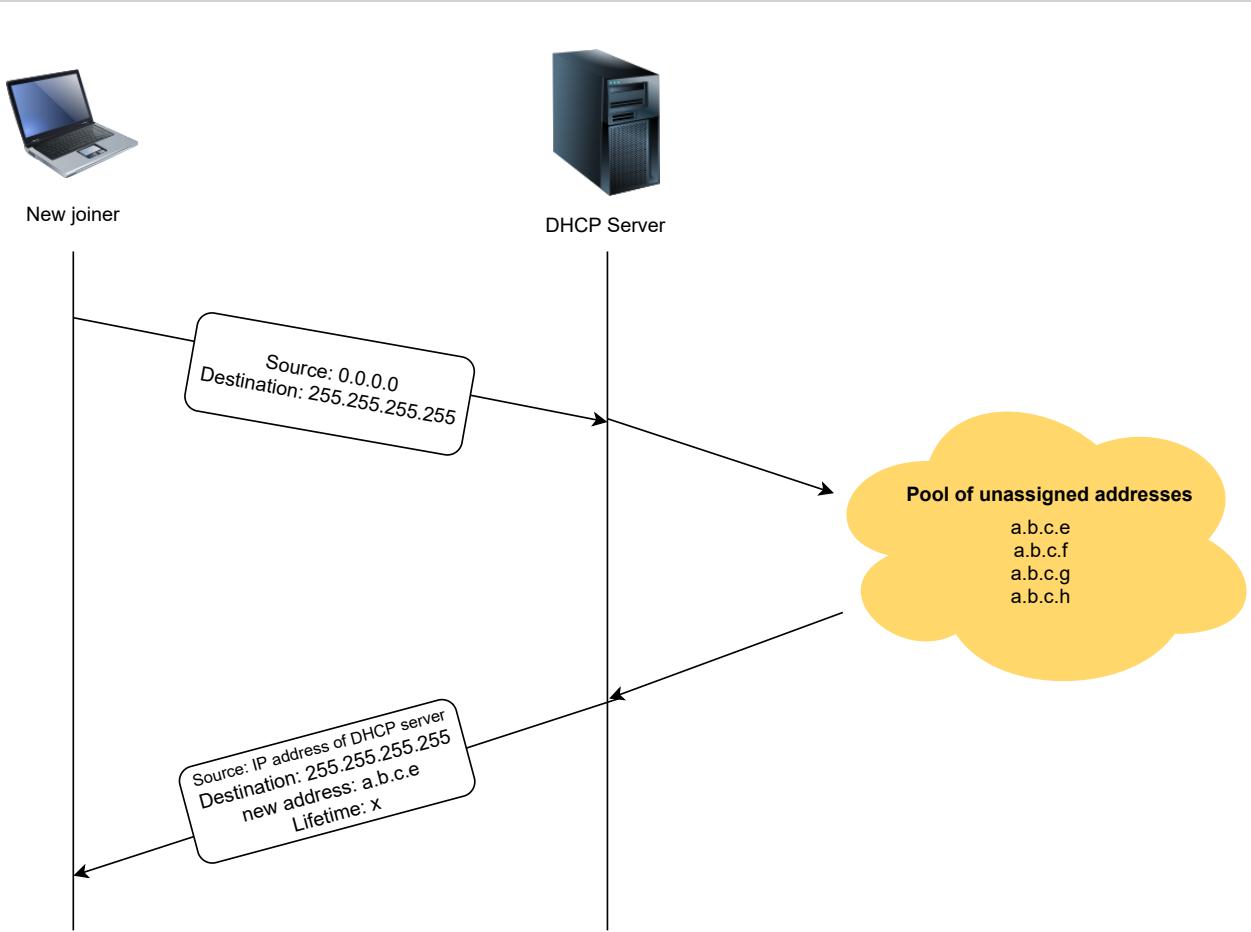
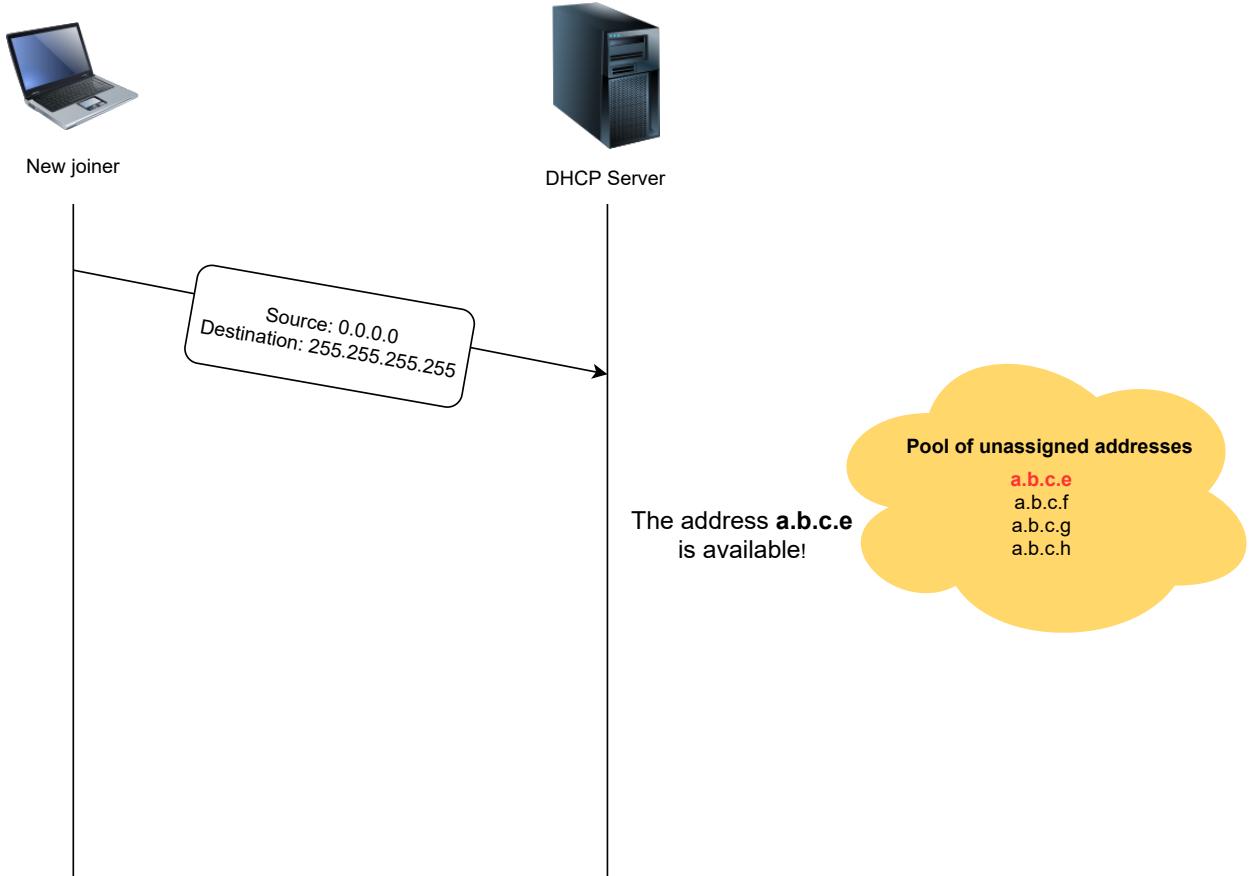
## How It Works #

- A DHCP server is associated with the subnet to which it is connected. Routers do not forward DHCP traffic from one subnet to another.
- Each DHCP server manages a pool of IPv4 addresses assigned to the subnet.
- When a host is first attached to the subnet, it sends a DHCP request message in a UDP segment to the DHCP server (the DHCP server listens on port 67).

- As the host knows neither its own IPv4 address nor the IPv4 address of the DHCP server, this UDP segment is sent inside an IPv4 packet whose **source and destination addresses are 0.0.0.0 and 255.255.255.255 respectively.**
  - The DHCP request **may contain options such as the data link layer address of the host.**
- The server captures the DHCP request and selects an unassigned address in its address pool.
- It then sends the assigned IPv4 address in a DHCP reply message which contains:
  - The data link layer address of the host and additional information such as
    - The subnet mask of the IPv4 address
    - The address of the default router or the address of the DNS resolver.
  - The DHCP reply also specifies the lifetime of the address allocation. This forces the host to renew its address allocation once it expires.
- This DHCP reply message is sent in an IPv4 packet whose source and destination addresses are respectively the IPv4 address of the DHCP server and the 255.255.255.255 broadcast address.
- Thanks to the limited lease time, IP addresses are automatically returned to the pool of addresses when hosts are powered off. This reduces the waste of IPv4 addresses. Furthermore, the IP has to be renewed with the server every so often.

Have a look at the following slides to see how an IP address is retrieved from a DHCP server. Note however, that the DHCP request response is an abstraction that we have created here. The DHCP protocol specifies several messages and their formats to do its job. For example, a DHCP server makes an offer for address assignment to a client, which the client may or may not accept. So, there are "Offers," "Acks" and "Nacks" etc.







## Quick Quiz! #

1

What are the responsibilities of the DHCP server?

COMPLETED 0%

1 of 3



In the next lesson, we'll put everything we've learned together to see how an IPv4 packet travels over the Internet.

# IPv4 in Practice: The Life of a Packet

In this lesson, we'll consolidate everything we have learned about the network layer so far by tracing the journey of a packet.

## WE'LL COVER THE FOLLOWING



- Sending a Packet
- Receiving A Packet
  - If ICMP is Received
  - How Routers Handle Packets
- Quick Quiz!

At this point in the description of IPv4, it is useful to have a detailed look at how an IPv4 implementation sends, receives and forwards IPv4 packets.

The simplest case is when a host needs to send a transport layer segment in an IPv4 packet. In order to do so, it performs two operations.

1. First, it must **decide on which interface the packet will be sent**.
2. Second, it must **create the corresponding IP packet(s)**.

To simplify the discussion in this section, we ignore the utilization of IPv4 options. This is not a bad idea as most of the traffic today consists of IP packets that don't make any use of IP options. Furthermore, we also assume that only point-to-point links are used.

An IPv4 host with  $n$  data link layer interfaces manage  $n + 1$  IPv4 addresses:

- The 127.0.0.1/32 IPv4 address assigned by convention to its loopback address.
- One  $A.B.C.D/p$  IPv4 address assigned to *each* of its  $n$  data link layer interfaces.

- The host maintains a forwarding table that contains one entry for its loopback address and one entry for each subnet identifier assigned to its interfaces.
- Furthermore, the host usually uses one of its interfaces as the default interface when sending packets that are not addressed to a directly connected destination. This is represented by the default route: 0.0.0.0/0 that is associated with one interface.

## Sending a Packet #

- When a transport protocol running on the host requests the transmission of a segment, it usually provides the IPv4 destination address to the IPv4 layer in addition to the segment.
- The IPv4 implementation first performs a longest prefix match with the destination address in its forwarding table. The lookup returns the identification of the interface that must be used to send the packet.
- The host can then create the IPv4 packet that contains the segment! The source IPv4 address of the packet is the IPv4 address of the host on the interface returned by the longest prefix match.
- The **Protocol** field of the packet is set to the identification of the local transport protocol which created the segment.
- The **TTL** field of the packet is set to the default TTL used by the host.
- The host must now choose the packet's **Identification**. This Identification is important if the packet becomes fragmented in the network, as it ensures that the destination is able to reassemble the received fragments.
  - Ideally, a sending host should never send a packet twice with the same identification to the same destination host, in order to ensure that all fragments are correctly reassembled by the destination.
  - Unfortunately, a 16-bit Identification field and an expected MSL of 2 minutes, and maximum packet size of 65535 implies that the maximum bandwidth to a given destination is limited to roughly 286 Gbps.

Mbps. Here's the derivation:

- The identification field is 16 bits so there can be 65536 ( $2^{16}$ ) different unique packets between a specific client-server pair.
- Hence, at most 65536 unique packets may be sent in one maximum segment lifetime. Then, another 65536 packets in the next MSL and so on.
- Suppose you are sending 65535 byte packets (the maximum size), then you are transmitting
$$65535 \times 8 \times 65536 = 34,359,214,080 \text{ bits per MSL.}$$
- Assuming that the MSL is 120 seconds long, that implies a rate of  $34,359,214,080/120$  bits per second which is  $\approx 286$  Mbps.

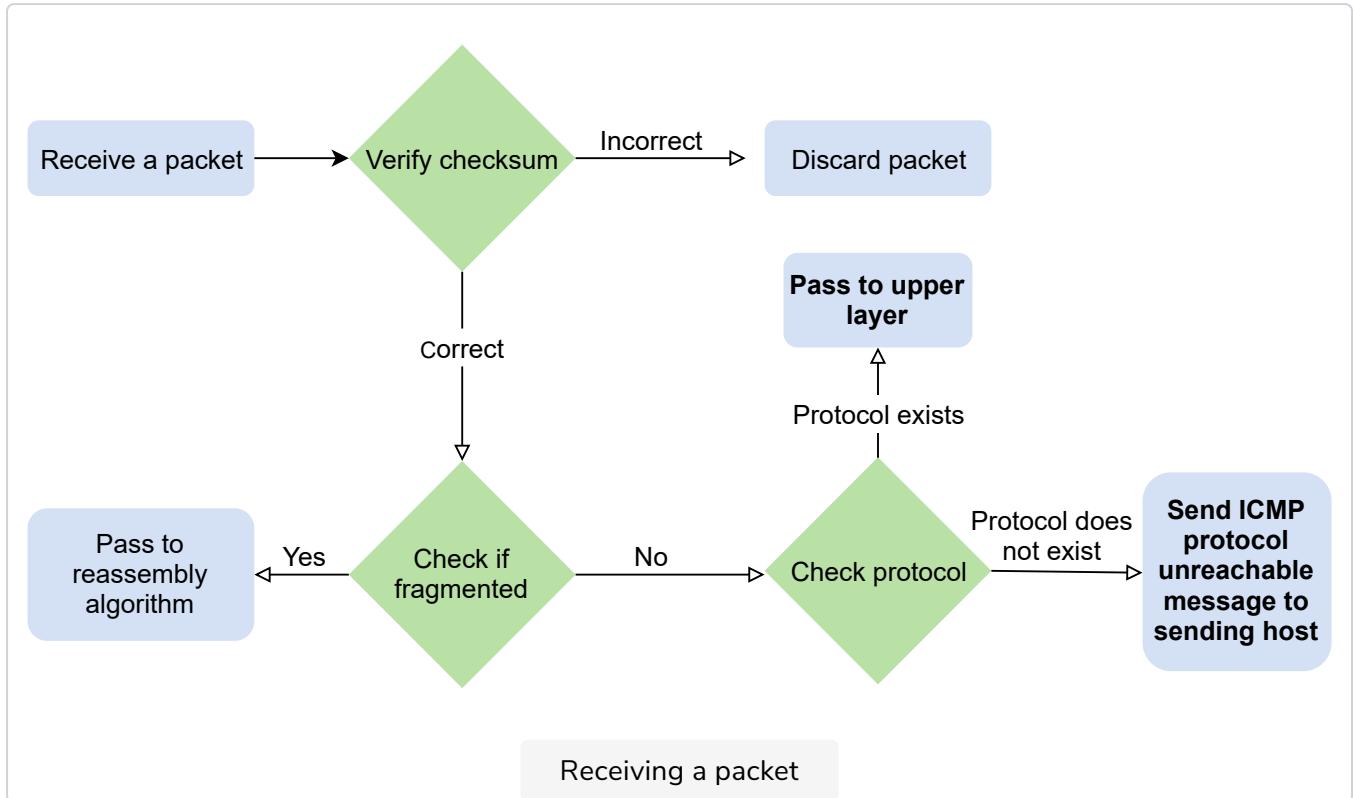
However, with a more realistic 1500 bytes MTU, that bandwidth drops to 6.4 Mbps to make fragmentation possible. This is very low and is another reason why hosts are highly encouraged to avoid fragmentation. If despite all of this, the MTU of the outgoing interface is smaller than the packet's length, the packet is fragmented.

- Finally, the packet's checksum is computed before transmission.

## Receiving A Packet #

When a host receives an IPv4 packet destined to itself, there are several operations that it must perform.

- First, it must check the packet's **checksum**. If the checksum is incorrect, the packet is discarded.
- Then, it must check whether the packet has been fragmented. If yes, the packet is passed to the reassembly algorithm described earlier. Otherwise, the packet must be passed to the upper layer. This is done by looking at the Protocol field (6 for TCP and 17 for UDP).
- If the host doesn't implement the transport layer protocol corresponding to the received Protocol field, it sends a **Protocol unreachable ICMP message** to the sending host.



## If ICMP is Received #

If the received packet contains an ICMP message (with the protocol field set to 1), the processing is more complex.

- An **Echo-request ICMP message** triggers the transmission of an ICMP Echo-reply message.
- The other types of ICMP messages, except for ICMP Echo Response, indicate an error that was caused by a previously transmitted packet. These ICMP messages are usually forwarded to the transport protocol that sent the erroneous packet. This can be done by inspecting the contents of the ICMP message that includes the header and the first 64 bits of the erroneous packet.
- If the IP packet did not contain options, which is the case for most IPv4 packets, the transport protocol can find in the first 32 bits of the transport header the source and destination ports to determine the affected transport flow. This is important for Path MTU discovery for example.

## How Routers Handle Packets #

When a router receives an IPv4 packet, it must:

- First check the packet's checksum. If the checksum is invalid, it's discarded.
- Otherwise, the router must check whether the destination address is one of the IPv4 addresses assigned to the router. If so, the router must behave as a host and process the packet as described above. Although routers mainly forward IPv4 packets, they sometimes need to be accessed as hosts by network operators or network management software.
- If the packet is not addressed to the router, it must be forwarded on an outgoing interface according to the router's forwarding table.
- The router first decrements the packet's TTL.
  - If the TTL reaches 0, a TTL Exceeded ICMP message is sent back to the source.
  - As the packet header has been modified, the checksum must be recomputed. Fortunately, as IPv4 uses an arithmetic checksum, a router can incrementally update the packet's checksum.
- Then, the router performs a longest prefix match for the packet's destination address in its forwarding table.
  - If no match is found, the router must return a **Destination unreachable ICMP** message to the source.
  - Otherwise, the lookup returns the interface over which the packet must be forwarded.
- Before forwarding the packet over this interface, the router must first compare the length of the packet with the MTU of the outgoing interface.
  - If the packet is smaller than the MTU, it is forwarded.
  - Otherwise, a **Fragmentation needed ICMP message** is sent if the DF flag was sent or the packet is fragmented if the DF was not set.

## Quick Quiz! #

1

When is an ICMP protocol unreachable message sent?

COMPLETED 0%

1 of 3



---

In the next lesson, we'll study IPv6!

# Why IPv6?

In this lesson, we'll get an introduction to IP version 6.

## WE'LL COVER THE FOLLOWING



- Why IPV6?
- IPv6 Features
  - Pros
  - Cons
- Textual representation of IPv6 addresses
- Quick Quiz!

## Why IPV6? #

- IPv4 was initially designed for a research network that would interconnect *some* research labs and universities. For this purpose, 32 bit addresses, i.e.,  $2^{32} = 4,294,967,296 \approx 4.3$  billion addresses seemed sufficient. Also, 32 bits was an incredibly convenient address size for software-based routers.

However, the popularity of the Internet, i.e., the number of smartphones and Internet of Things devices, was not anticipated. We've made do with 4.3 billion addresses so far by reusing them and with [NAT boxes](#). Nonetheless, we are running out of addresses. Hence, **IPv6 was designed to tackle these limitations of IPv4.**

## IPv6 Features #

IPv6 has some distinguishing pros and cons.

### Pros #

- **Simplified Header:** All IPv4 options are moved to the end of the IPv6

header. IPv6 header is twice as large as IPv4 headers but only because IPv6 addresses are four times longer.

- **Larger Address Space:** IPv6 addresses face **4 times as many bits** as IPv4. So all IPv6 addresses are 128 bits wide. This implies that there are 340, 282, 366, 920, 938, 463, 463, 374, 607, 431, 768, 211, 456  $2^{32 \times 4} = 2^{128} \approx 3.4 \times 10^{38} \approx 340 \text{ undecillion addresses}$  different IPv6 addresses. As the surface of the Earth is about  $510,072,000 \text{ km}^2$ , this implies that there are about  $6.67 \times 10^{23}$  IPv6 addresses per square meter on Earth. Compared to IPv4, which offers only 8 addresses per square kilometer, this is a significant improvement on paper.

## Cons #

- IPv6 is a complete redesign over IPv4 and hence is **not backward compatible**. This means that devices configured over IPv4 can NOT access websites on servers configured with IPv6!
- Upgrading to IPv6 enabled hardware is an expensive shift for ISPs and is not directly translatable in terms of profit. This is part of the reason why the world has not shifted entirely to IPv6.

## Textual representation of IPv6 addresses #

It's sometimes necessary to write IPv6 addresses in text format, e.g., when manually configuring addresses or for documentation purposes.

The preferred format for writing IPv6 addresses is **x:x:x:x:x:x:x:x**, where the x's are hexadecimal digits representing the eight 16-bit parts of the address. Here are a few examples of IPv6 addresses:

- ABCD:EF01:2345:6789:ABCD:EF01:2345:6789
- 2001:DB8:0:0:8:800:200C:417A
- FE80:0:0:0:219:E3FF:FED7:1204

IPv6 addresses often contain a long sequence of bits set to 0. In this case, a compact notation has been defined. With this notation, :: is used to indicate one or more groups of 16 bit blocks containing only bits set to 0. For example:

- 2001:DB8:0:0:8:800:200C:417A is represented as  
2001:DB8::8:800:200C:417A

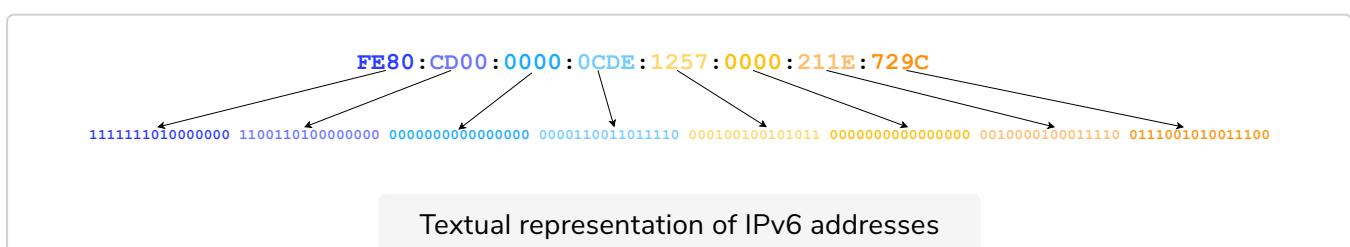
- FF01:0:0:0:0:0:101 is represented as FF01::101

- 0:0:0:0:0:0:1 is represented as ::1
- 0:0:0:0:0:0:0 is represented as ::

An IPv6 prefix can be represented as **address/length**, where length is the length of the prefix in bits. For example, the three notations below correspond to the same IPv6 prefix:

- 2001:0DB8:0000:CD30:0000:0000:0000:0000/60
- 2001:0DB8::CD30:0:0:0:0/60
- 2001:0DB8:0:CD30::/60

Here's a drawing that represents how an IPv6 address is written in text form.



## Quick Quiz! #

1

What's a compact way of representing the IPv6 address:  
**FE00:0:0:0:219:A34F:F3D7:1204?**

COMPLETED 0%

1 of 3



In the next lesson, we'll study IPv6 address types!

# IPv6 Features

In this lesson, we'll study IPv6 address types.

## WE'LL COVER THE FOLLOWING



- IPv6 Address Format & Types
- Unicast
  - Provider Independent Addresses
  - Disadvantages of Provider Independent Addresses
  - Unique Local Unicast Addresses
  - Link-Local Unicast Addresses
- Anycast Addresses
- Multicast Addresses
- Quick Quiz!

## IPv6 Address Format & Types #

The experience of IPv4 revealed that the **scalability** of a network layer protocol **heavily depends on its addressing architecture**. The designers of IPv6 therefore spent a lot of effort defining its addressing architecture [RFC 3513](#). IPv6 supports **unicast, multicast and anycast** addresses.

## Unicast #

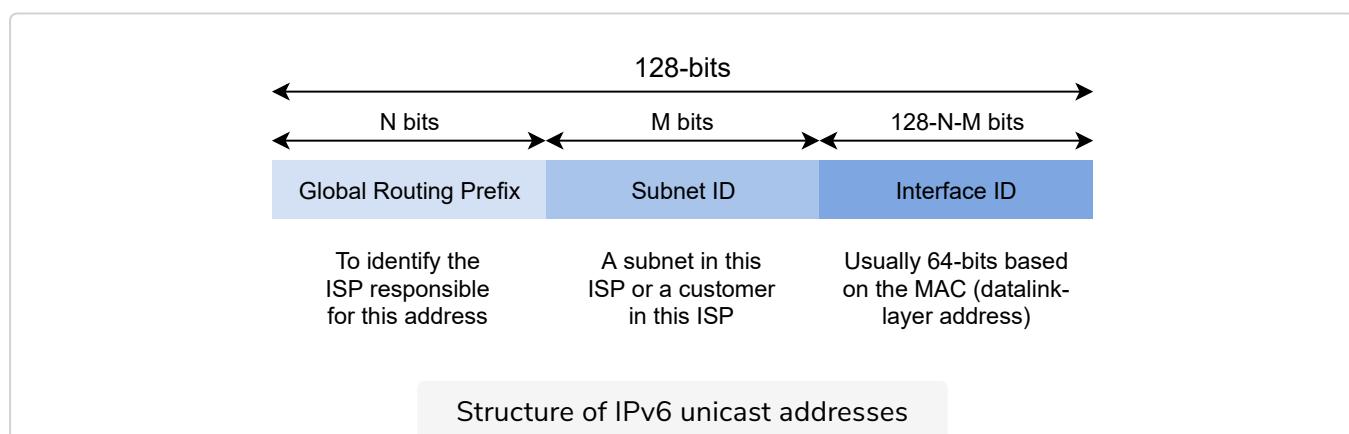
As with IPv4, an IPv6 unicast address is used to identify one data link layer interface on a host. If a host has several data link layer interfaces (such as an Ethernet interface and a WiFi interface), then it needs several IPv6 addresses. An IPv6 unicast address is composed of three parts:

1. A **global routing prefix** that is assigned to the Internet Service Provider that owns this block of addresses
2. A **subnet identifier** that identifies a customer of the ISP

### 3. An **interface identifier** that identifies a particular interface on an end-system

As described in the previous lesson, IPv6 addresses are divided into **eight 16-bit groups separated by colons**. Each group is represented by four hexadecimal digits.

In general, an IPv6 unicast address is structured as shown in the figure below:



In today's deployments, interface identifiers are always **64 bits**. This implies that while there are  $2^{128}$  different IPv6 addresses, they must be grouped in  $2^{64}$  subnets. This could appear as a waste of resources, however using 64 bits for the host identifier allows IPv6 addresses to be auto-configured and also provides some benefits from a security point of view.

In practice, there are **several types of IPv6 unicast address**.

## Provider Independent Addresses #

Most of the **IPv6 unicast addresses** are allocated in blocks under the responsibility of **IANA**. The current IPv6 allocations are part of the **2000::/3** address block. Regional Internet Registries (RIR) such as RIPE in Europe, ARIN in North-America or AfriNIC in Africa have each received a block of IPv6 addresses that they sub-allocate to Internet Service Providers in their region. The ISPs then sub-allocate addresses to their customers as usual. When considering the allocation of IPv6 addresses, two types of address allocations are often distinguished. The RIRs allocate **provider-independent (PI) addresses**.

- PI addresses are usually allocated to Internet Service Providers and large companies that are connected to at least two different ISPs.
- Once a PI address block has been allocated to a company, this company can use its address block with the provider of its choice and change its provider at will.
- Internet Service Providers allocate provider-aggregatable (PA) address blocks from their own PI address block to their customers.
- A company that is connected to only one ISP should only use PA addresses.

## Disadvantages of Provider Independent Addresses #

- The drawback of PA addresses is that when a company using a PA address block changes its provider, it needs to change all the addresses that it uses. This can be a nightmare from an operational perspective, and many companies are lobbying to obtain PI address blocks even if they are small and connected to a single provider.

The typical size of the IPv6 address blocks are:

- /32 for an Internet Service Provider
- /48 for a single company
- /64 for a single user (e.g. a home user connected via ADSL)
- /128 in the rare case when it is known that no more than one end host will be attached.

## Unique Local Unicast Addresses #

For the companies that want to use IPv6 without being connected to the IPv6 Internet, [RFC 4193](#) defines the Unique Local Unicast (ULA) addresses (FC00::/7).

These ULA addresses play a similar role as the private IPv4 addresses defined in [RFC 1918](#). However, the size of the FC00::/7 address block allows ULA to be much more flexible than private IPv4 addresses. Furthermore, the IETF has reserved some IPv6 addresses for a special usage. The two most important ones are:

- **0:0:0:0:0:1** (::1 in compact form) is the IPv6 **loopback address**. This is the address of a logical interface that is always up and running on IPv6

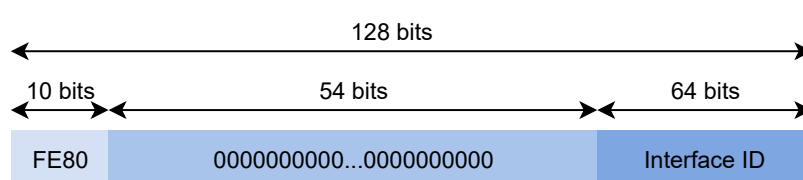
enabled hosts. This is the equivalent of **127.0.0.1** in IPv4.

- **0:0:0:0:0:0:0:0** (:: in compact form) is the **unspecified IPv6 address**. This is the IPv6 address that a host can use as source address when trying to acquire an official address.

## Link-Local Unicast Addresses #

The last type of unicast IPv6 addresses is the **Link-Local Unicast addresses**. These addresses are part of the **FE80::/10** address block and are defined in [RFC 4291](#).

- Each host can compute its own link-local address by concatenating the **FE80::/64** prefix with the 64 bits identifier of its interface.
- Link-local addresses can be used when hosts that are attached to the same link (or local area network) need to exchange packets.
- They are used notably for address discovery and auto-configuration purposes.
- Their usage is restricted to each link, and a router cannot forward a packet whose source or destination address is a link-local address.
- Link-local addresses have also been defined for IPv4 [RFC 3927](#). However, the IPv4 link-local addresses are only used when a host cannot obtain a regular IPv4 address, e.g., on an isolated LAN.



Pv6 link local address structure

An important consequence of the IPv6 unicast addressing architecture and the utilization of link-local addresses is that **an IPv6 host has several IPv6 addresses**. This implies that an IPv6 stack must be able to **handle multiple IPv6 addresses**. This was not always the case with IPv4.

## Anycast Addresses #

[RFC 4291](#) defines a special type of IPv6 **anycast address**. On a subnetwork,

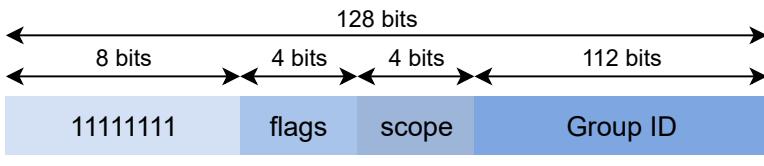
[RFC 4291](#) defines a special type of IPv6 **anycast address**. On a subnet-work having prefix  $p/n$ , the IPv6 address whose 128-n low-order bits are set to 0 is the anycast address that corresponds to all routers inside this subnet-work. This anycast address can be used by hosts to quickly send a packet to any of the routers inside their own subnetwork.

## Multicast Addresses #

Finally, [RFC 4291](#) defines the structure of the IPv6 multicast addresses. This structure is depicted in the figure below.

The lower order 112 bits of an IPv6 multicast address are the group's identifier. The higher-order bits are used as a marker to distinguish multicast addresses from unicast addresses.

- Notably, the 4 bits flag field indicates whether the address is temporary or permanent.
- Finally, the scope field indicates the boundaries of the forwarding of packets destined to a particular address.
  - A link-local scope indicates that a router should not forward a packet destined to such a multicast address.
  - An organization local-scope indicates that a packet sent to such a multicast destination address should not leave the organization.
  - Finally, the global scope is intended for multicast groups spanning the global Internet.



IPv6 multicast address structure

Among these addresses, some are well known. For example, all end-system automatically belong to the **FF02::1** multicast group while all routers automatically belong to the **FF02::2** multicast group.

## Quick Quiz! #

1

In an IPv6 unicast address, if 12 bits are used for the global routing prefix and 64 bits are used for the interface ID, how many are used for the subnet ID?

COMPLETED 0%

1 of 3



In the next lesson, we'll study middleboxes!

# Middleboxes: Firewalls

In this lesson, we'll study middleboxes!

## WE'LL COVER THE FOLLOWING



- Introduction
  - Firewall Interfaces
  - Firewall Filters
  - Stateless Vs. Stateful Firewalls
  - Host-based Vs. Network-based Firewalls
- Quick Quiz!

## Introduction #

When the TCP/IP architecture and the IP protocol were defined, two types of devices were considered in the network layer:

1. **End hosts** which are the sources and destinations of IP packets
2. **Routers** that forward packets. When a router forwards an IP packet, it consults its forwarding table, updates the packet's TTL, recomputes its checksum and forwards it to the next hop. A router **does not need to read or change the contents of the packet's payload.**

However, in today's Internet, there exist devices called **middleboxes** that are **not strictly routers** but which **process, sometimes modify, and forward IP packets** ([RFC 3234](#)). Some middleboxes only operate in the network layer, but most middleboxes are able to analyze the payload of the received packets and extract the transport header, and in some cases the application layer headers.

Over the next couple of lessons, we'll briefly describe **two types of middleboxes: firewalls and network address translation (NAT) devices.**

# Firewalls

## Why Firewalls?

When the Internet was only a research network interconnecting research labs, security was not a concern. However, as the Internet grew in popularity, security concerns grew.



This was exacerbated by several security issues at the end of the 1980s such as [the first Internet worm](#) and some other widely publicized security breaches.



**Did You Know?** The term **firewall** originates from a special wall used to confine the spread of fire in a building. It was also used to refer to a metallic wall between the engine compartment and the passenger area in a car. The purpose of this metallic wall is to prevent the spread of a fire in the engine compartment into the passenger area.

## Firewall Interfaces #

These security problems convinced the industry that their networks should be protected by special devices the way security guards and fences are used to protect buildings. These special devices came to be called **firewalls**. A typical firewall has **two interfaces**:

1. An external interface connected to the global Internet.
2. An internal interface connected to a trusted network.

## Firewall Filters #

The first firewalls included configurable **packet filters**. A packet filter is a set of rules defining the security policy of a network. In practice, these rules are based on the values of fields in the IP or transport layer headers. Any field of the IP or transport header can be used in a firewall rule, but the most common ones are:

- Filter on the **source address**. For example, a company may decide to discard all packets received from one of its competitors in certain portions of the network while maintaining access to public resources. Another example of source based filtering is **black lists**. Any packets from an IP on the black list will be discarded. IPs known for their use by spammers, for instance, are blacklisted by many networks.
- Filter on the **destination address**. For example, the hosts of the research lab of a company may receive packets from the global Internet, but not the hosts of the financial department.
- Filter on the **Protocol number** found in the IP header. For example, a company may only allow its hosts to use TCP or UDP, but not other, more experimental, transport protocols.
- Filter on the TCP or UDP **port numbers**. For example, only the DNS server of a company should receive UDP segments whose destination port is set to 53, or only the official SMTP servers of the company can send TCP segments whose source ports are set to 25.
- Filter on the **TCP flags**. For example, a simple solution to prohibit external hosts from opening TCP connections with hosts inside the company is to discard all TCP segments received from the external interface with only the SYN flag set.

## Stateless Vs. Stateful Firewalls #

A firewall that does not maintain the state of flows passing through it is known as a **stateless firewall**.

However, a **stateful firewall**, on the other hand, sees the first packet in a flow that is allowed by the configured security rules it creates a **session state** for it.

All subsequent packets belonging to that flow are allowed to go through. This filtering is more efficient compared to stateless firewalls that have to apply their rules to each and every packet. The flip side is the maintenance of state, which needs to be controlled.

## Host-based Vs. Network-based Firewalls #

Network-based firewalls are hardware based and generally deployed on the edge of a network. They are easy to scale and simple to maintain.

A host based firewalls, however, are software based and are deployed on end-systems. They are generally not easy to scale and require maintenance.

### Quick Quiz! #

1

What is the purpose of firewalls?

COMPLETED 0%

1 of 2



In the next lesson, we'll have a look at another middlebox, NAT!

# Middleboxes: NATs

In this lesson, we'll study Network Address Translation!

## WE'LL COVER THE FOLLOWING



- Introduction
  - Broadband Access Routers
  - Enterprise Networks
  - Sending a Message over a NAT
    - Sending a Message
    - Receiving a Message
  - Disadvantages of NATs
  - Quick Quiz!

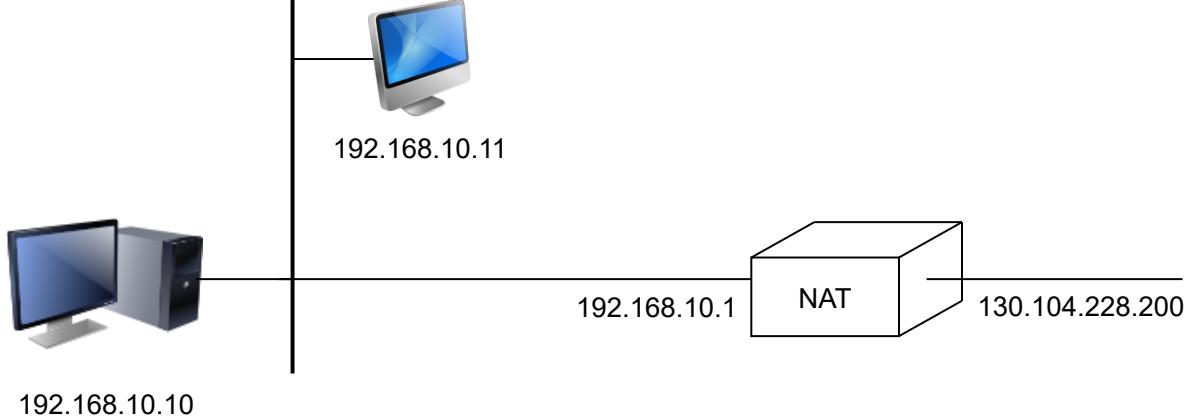
## Introduction #

Network Address Translation (NAT) was proposed as a short term solution to deal with the expected shortage of IPv4 addresses in the late 1980s to early 1990s. Combined with CIDR, NAT helped to significantly slow down the consumption of IPv4 addresses. A NAT is a middlebox that interconnects two networks that are using IPv4 addresses from different addressing spaces. Usually, one of these addressing spaces is the public Internet while the other is using a private IPv4 address. Unlike a router, when a NAT box forwards traffic, it modifies the IP addresses in the IP header, as will be described shortly.

## Broadband Access Routers #

A very common deployment of NAT is in broadband access routers as shown in the figure below. The broadband access router interconnects a home network, either WiFi or Ethernet-based, and the global Internet via one ISP.

A single IPv4 address is allocated to the broadband access router and network address translation allows **all of the hosts** attached to the home network **to share a single public IPv4 address.**



A simple NAT with one public IPv4 address

## Enterprise Networks #

The second type of deployment is in enterprise networks. In this case, the NAT functionality is installed on a border router of the enterprise. A private IPv4 address is assigned to each enterprise host while the border router manages a **pool containing several public IPv4 addresses**.



**Note:** In typical home usage scenarios, only one public IP address is available to the NAT. However, in enterprise settings, a public IP address pool may also be configured on the NAT box.

## Sending a Message over a NAT #

The simplest NAT is a middlebox a mapping between a private IP address and a public IP address. To understand its operation, let's assume that a NAT has just booted.

### Sending a Message #

- When the NAT receives the first packet from source **S** in the internal network which is destined to the public Internet, it creates a mapping between internal address **S** and the first address of its pool of public addresses (**P<sub>1</sub>**).

- Then, it translates the received packet so that it can be sent to the public

Internet. This translation is performed as followed:

- i. The source address of the packet (**S**) is replaced by the mapped public address (**P1**)
- ii. The checksum of the IP header is incrementally updated as its content has changed
- iii. If the packet carried a TCP or UDP segment, the transport layer checksum found in the included segment must also be updated as it is computed over the segment and a pseudo-header that includes the source and destination addresses.

## Receiving a Message #

- When a packet destined to **P1** is received from the public Internet, the NAT consults its mapping table to find **S**.
- The received packet is translated and forwarded in the internal network.

This works **as long as the pool of public IP addresses of the NAT does not become empty**. In this case, a mapping must be removed from the mapping table to allow a packet from a new host to be translated. This **garbage collection** can be implemented by adding to each entry in the mapping table a timestamp that contains the last utilization time of a mapping entry. This timestamp is updated each time the corresponding entry is used. Then, the garbage collection algorithm can remove the oldest mapping entry in the table.

## Disadvantages of NATs #

NAT allows many hosts to share one or a few public IPv4 addresses. However, using NAT has two important drawbacks.

1. First, it's not easily possible for external hosts to open TCP connections with hosts that are behind a NAT. Some consider this to be a benefit from a security perspective. However, a NAT should not be confused with a firewall, as there are some techniques to traverse NATs.
2. Second, NAT breaks the end-to-end transparency of the network and transport layers.

# Quick Quiz! #

1

Suppose several end systems including a web server is behind a NAT.  
Will external clients be able to initiate connections with it?

COMPLETED 0%

1 of 3



---

In the next lesson, we'll have an introduction to routing in IP networks!

# Introduction to Routing in IP: Intradomain & Interdomain

In this lesson, we'll look at an introduction to intradomain and interdomain routing algorithms.

## WE'LL COVER THE FOLLOWING



- Introduction
  - Intradomain Vs. Interdomain Routing
- Quick Quiz!

## Introduction #

If every router on the Internet had to manage routing entries for the entire Internet, then we would need very high-end and high performing routers. Also, the scale of exchanging routing information would be humongous. Instead, the Internet consists of **separate administrative domains**. Each domain is run and managed by an independent authority.

The Internet is hence comprised of **domains**. A domain can be a small enterprise that manages a few routers in a single building, a larger enterprise with a hundred routers at multiple locations, or a large Internet Service Provider managing thousands of routers. For example, Verizon is responsible for managing all devices in its networks.

As of this writing, the Internet is composed of more than 30,000 such different domains and this number is still growing.

## Intradomain Vs. Interdomain Routing #

The two main classes of routing protocols that are used to allow these domains to efficiently exchange routing information are: **intradomain routing protocols** and **interdomain routing protocols**.

Routers *within* a domain exchange routing information with each other - this is called **intradomain routing**.

Routers at the edges of the domains that connect to routers in other domains are called **border routers**. These routers run **interdomain routing** protocols. These protocols import routing information from other domains and export summarized information about their own domain to other domains.

Due to the differences in goals and priorities for interdomain and intradomain routing, the routing algorithms and protocols used are significantly different. We'll look at key protocols from each class.

## Quick Quiz! #

1

Why are domains needed?

COMPLETED 0%

1 of 3



In the next lesson, we'll look at an intradomain routing algorithm called OSPF.

# Intradomain Routing: OSPF

In this lesson we'll study the OSPF protocol.

## WE'LL COVER THE FOLLOWING ^

- Intradomain Routing
- OSPF
  - Hierarchical Routing
  - Areas
  - The Backbone Area

## Intradomain Routing #

Intradomain routing protocols have **two objectives**:

1. They distribute routing information that corresponds to the **shortest path between two routers in the domain**.
2. They should allow the routers to quickly **recover from link and router failures**.

## OSPF #

Open Shortest Path First (OSPF), defined in [RFC 2328](#), is one of the link-state routing protocols that has been standardized by the IETF.

## Hierarchical Routing #

A domain can consist of thousands of routers. In such a large network, exchanging link-state information, building a global view of the network, and efficiently handling changes in the network becomes **unscalable**. Instead, a large domain is divided hierarchically into separate zones, or in OSPF terms, **areas**. Link state information flooding is restricted to an area. Routers run Dijkstra's algorithm based on this information. All the routers inside an area have detailed information about the routes of the other areas.

have detailed information about the topology of the area but only learn

aggregated information about the topology of the other areas and their interconnections.

## Areas #

**OSPF supports a restricted variant of hierarchical routing.** In OSPF's terminology, **a region is called an area**. OSPF imposes restrictions on how a network can be divided into areas. An area is a set of routers and links that are grouped together. Usually, the topology of an area is chosen so that a packet sent by one router inside the area can reach any other router in the area without leaving the area. An OSPF area contains two types of routers:

- **Internal router:** A router is called an internal router if all of its interfaces are connected to other routers in the same area.
- **Area border routers:** A router that is connected to other routers in more than one areas.

For example, the network shown in the figure below has been divided into **three areas**:

1. **Area 0** that contains RA, RB, RC, and RD.
2. **Area 1** that contains routers R1, R3, R4, R5 and RA.
3. **Area 2** that contains R7, R8, R9, R10, RB and RC.

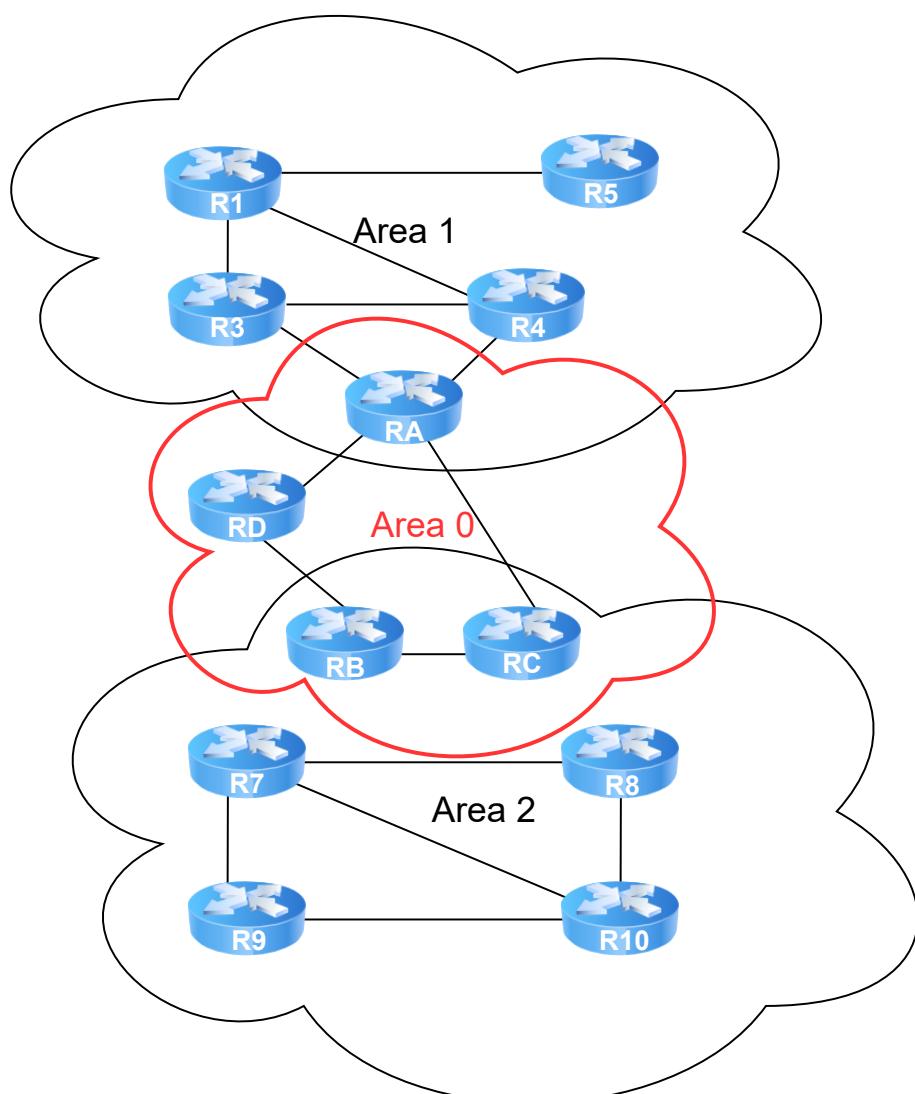
## The Backbone Area #

OSPF areas are identified by a 32 bit integer, which is sometimes represented as an IP address. Among the OSPF areas, area 0, also called **the backbone area**, has a special role.

- Some routers in the backbone area are connected to area border routers from a different area. On the other hand, some routers in the backbone area may not be connected to routers in any other area. In the following figure, RA, RB and RC fall in the former category while RD falls in the latter.
- All area border routers not belonging to the backbone area must be physically connected to an area border router in the backbone area!



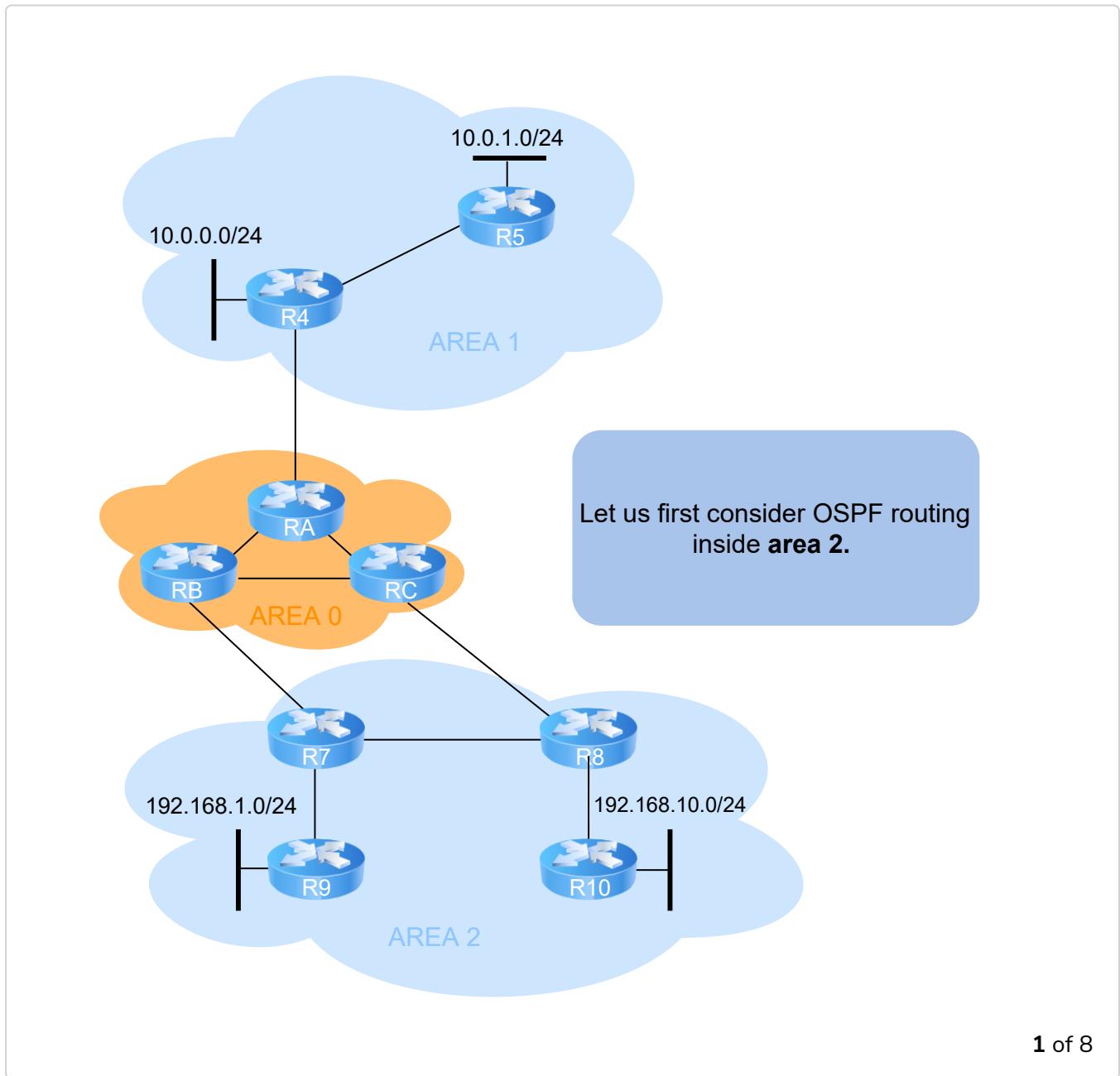
**Note:** It's possible to establish adjacency between area border routers that are not physically connected, but this is not preferred. Sometimes it might not be possible to connect two routers physically. A certain area border router could be miles and miles away from anything in the backbone area and not connected physically. In that case, its adjacency with the backbone area router can be established by configuration.

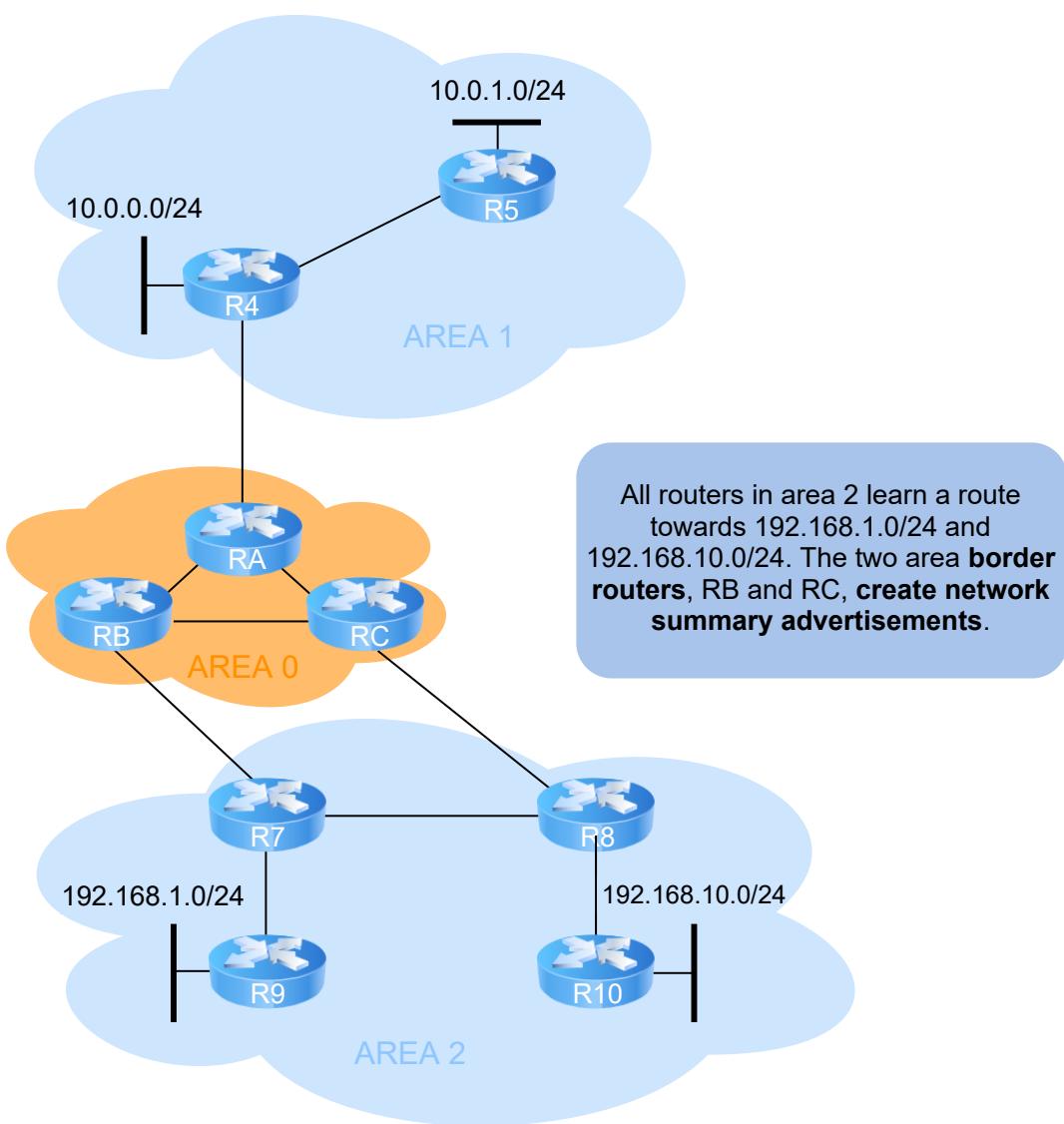


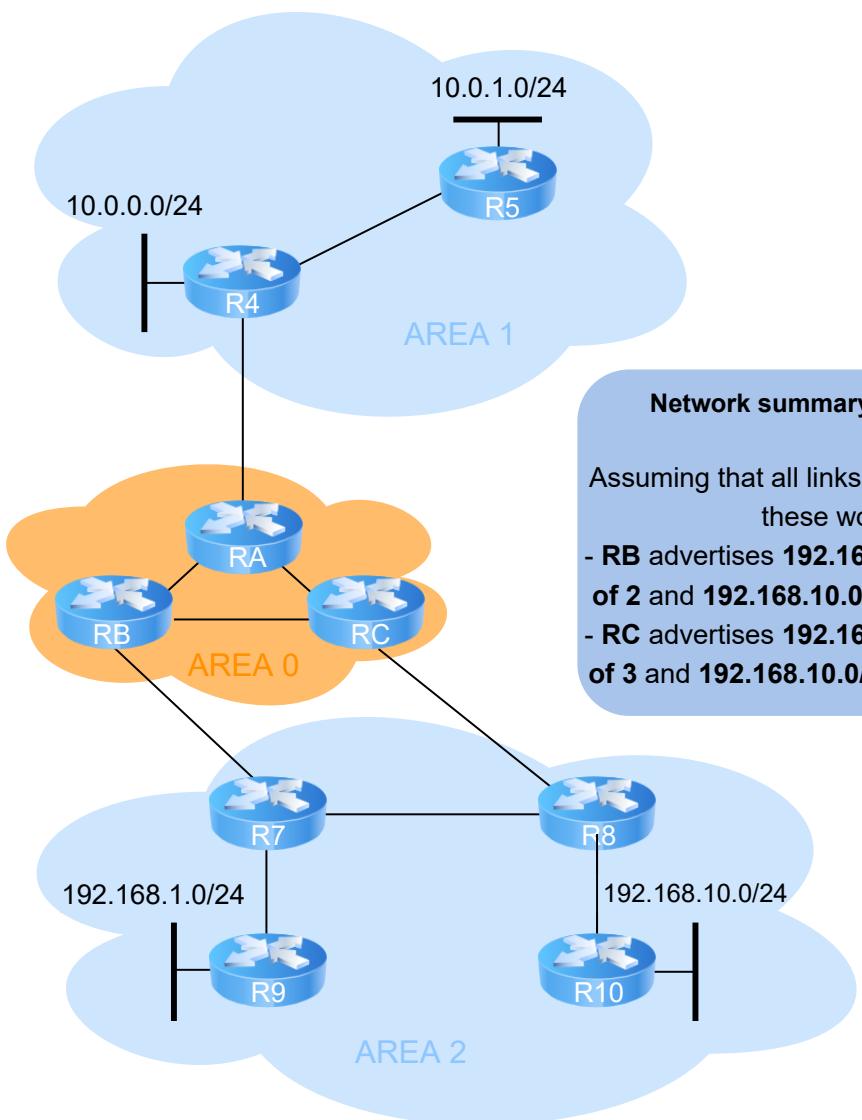
OSPF areas

- Inside each non-backbone area, routers distribute the topology of the area by exchanging link-state packets with only the other routers reachable inside that area.

- The internal routers don't know the topology of other areas, but each router knows how to reach the backbone area.
- Non-backbone area border routers inject their summarized intra-area routes as distance vectors into the backbone area. The backbone area border router propagates this information to others of its kind, and to different areas through their respective border routers.



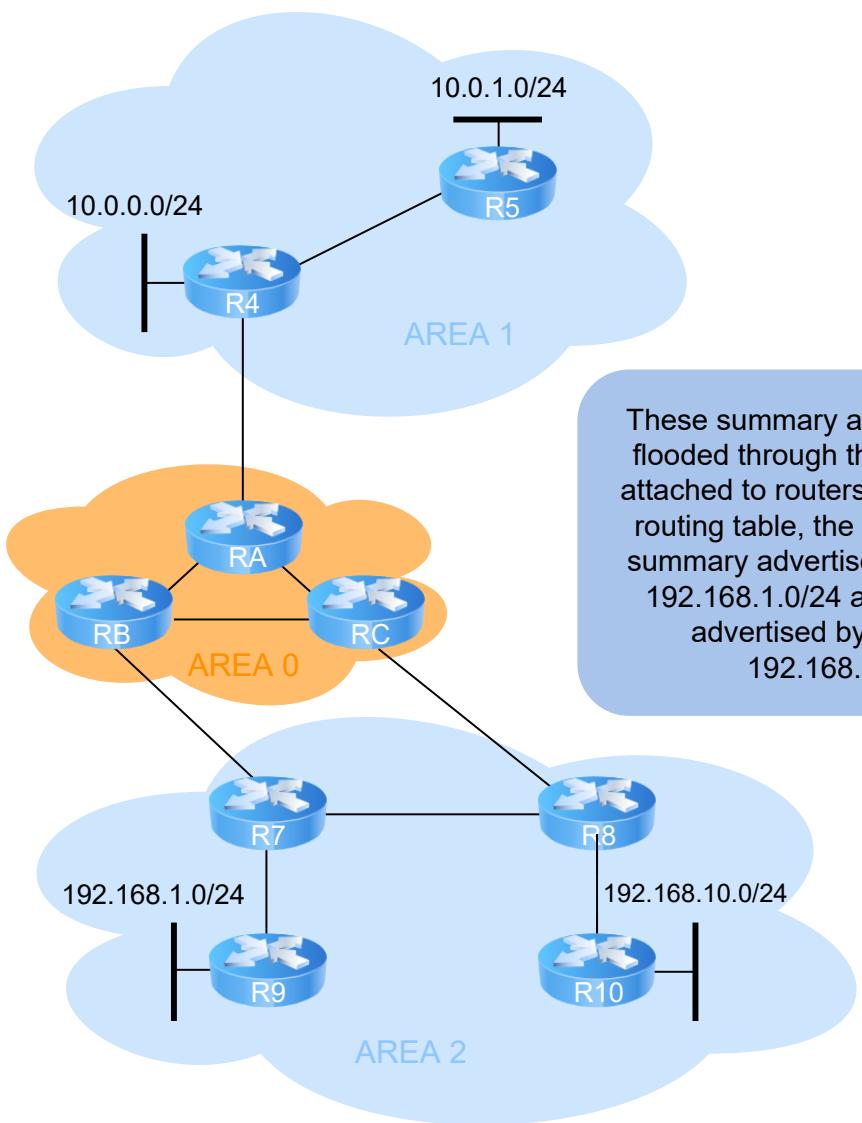


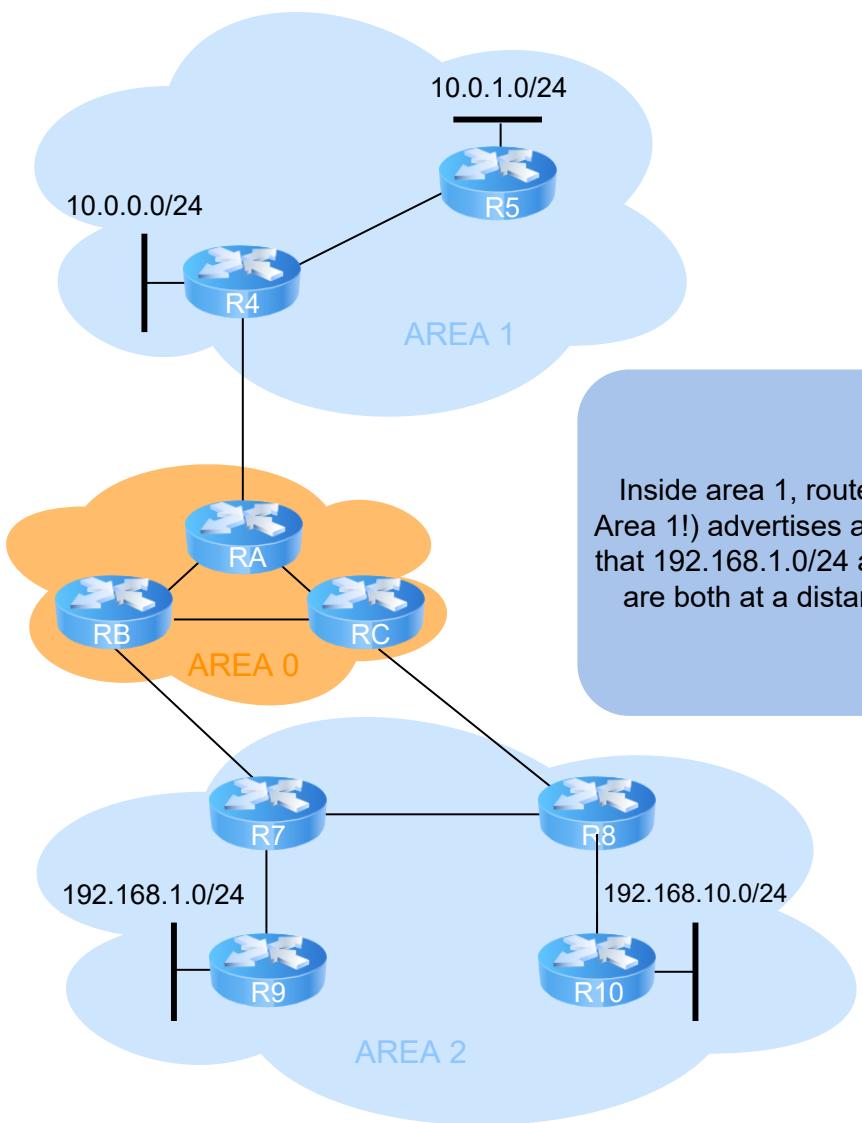


**Network summary advertisements:**

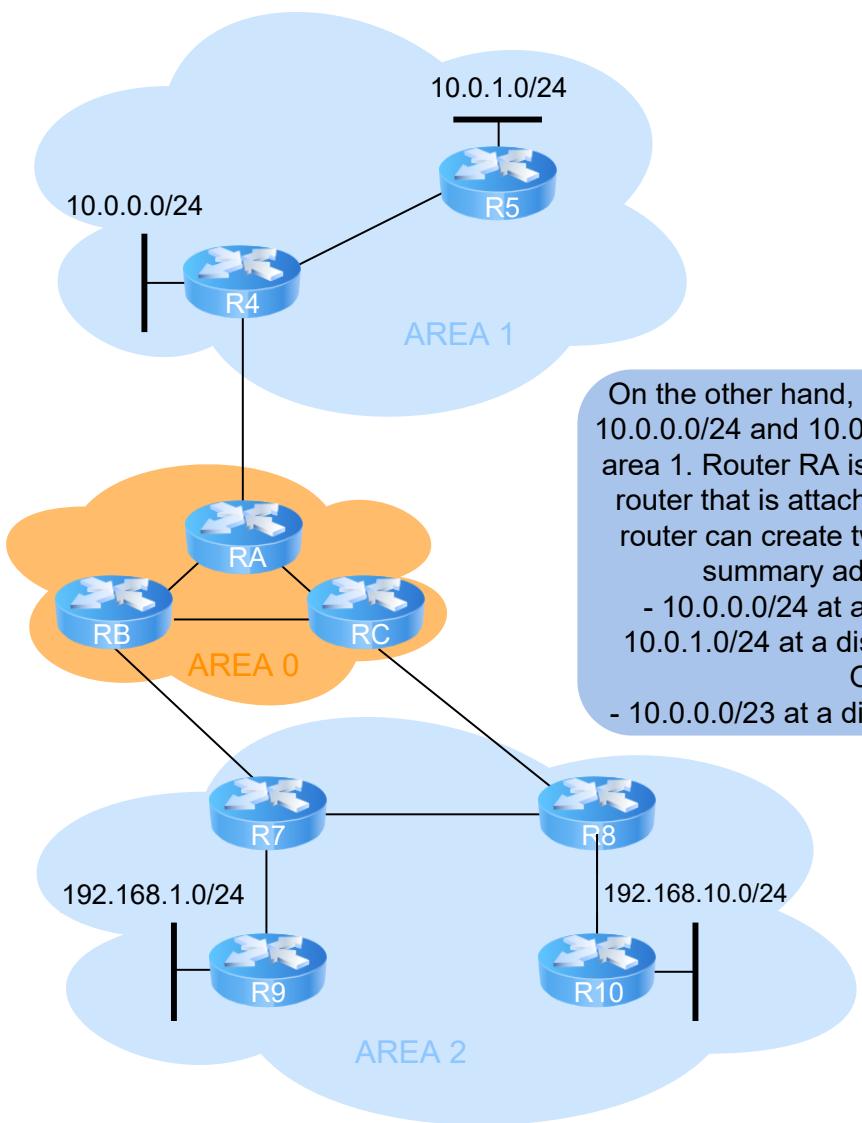
Assuming that all links have a unit link metric,  
these would be:

- RB advertises **192.168.1.0/24** at a **distance of 2** and **192.168.10.0/24** at a **distance of 3**
- RC advertises **192.168.1.0/24** at a **distance of 3** and **192.168.10.0/24** at a **distance of 2**.



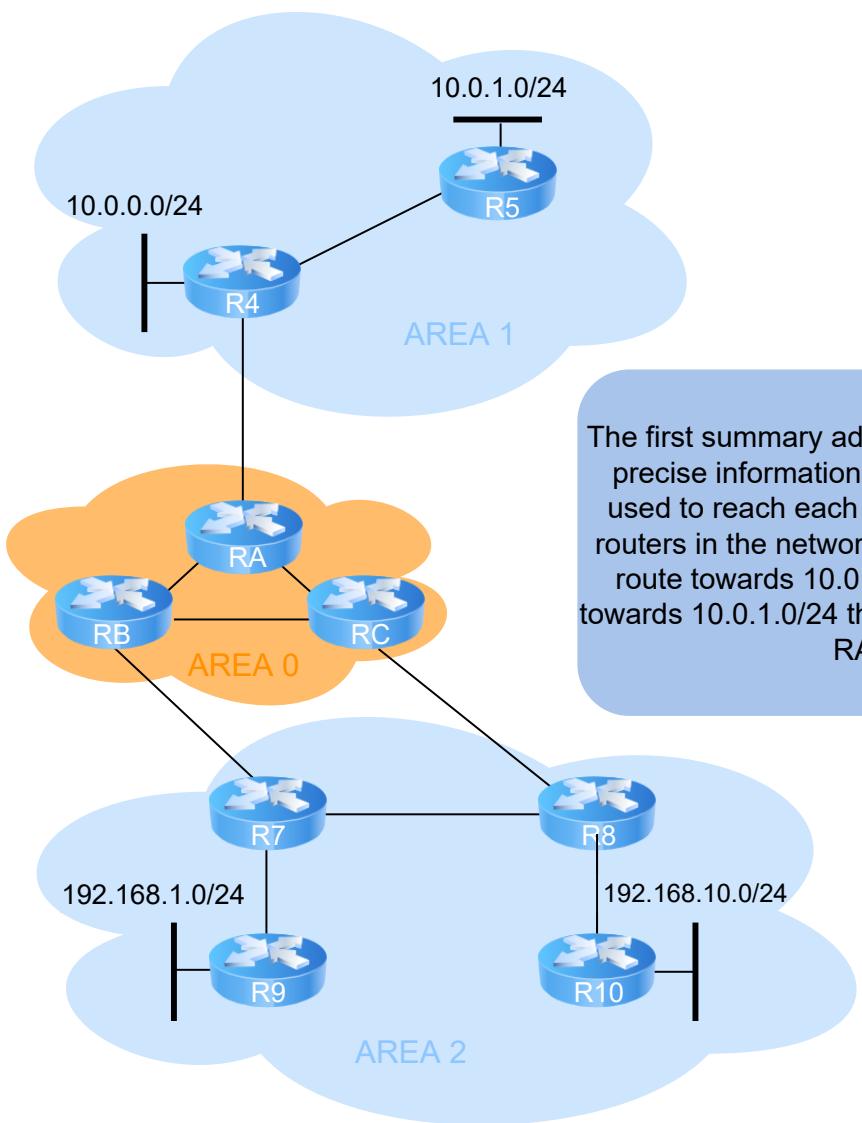


Inside area 1, router RA (RA is part of Area 1!) advertises a summary indicating that 192.168.1.0/24 and 192.168.10.0/24 are both at a distance of 3 from itself

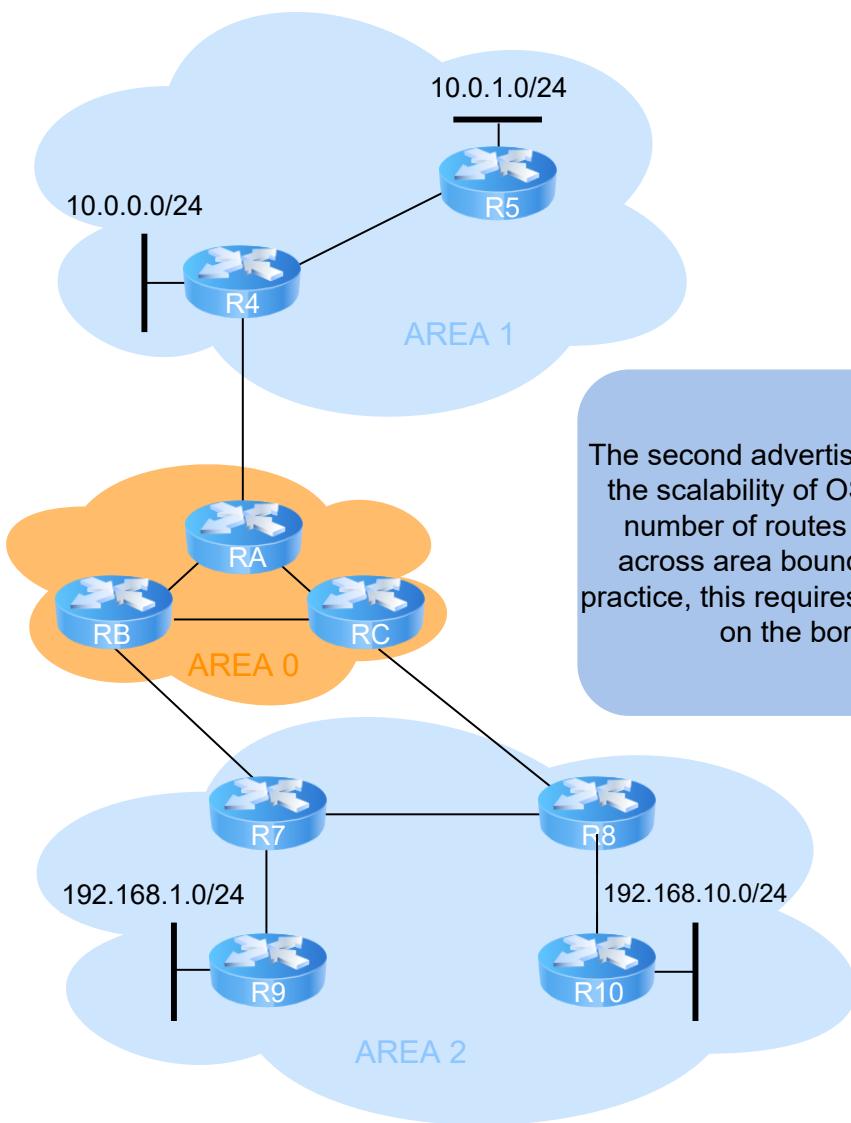


On the other hand, consider the prefixes 10.0.0.0/24 and 10.0.1.0/24 that are inside area 1. Router RA is the only area border router that is attached to this area. This router can create two different network summary advertisements:

- 10.0.0.0/24 at a distance of 1 and 10.0.1.0/24 at a distance of 2 from RA  
OR
- 10.0.0.0/23 at a distance of 1 from RA



The first summary advertisement provides precise information about the distance used to reach each prefix. However, all routers in the network have to maintain a route towards 10.0.0.0/24 and a route towards 10.0.1.0/24 that are both via router RA.



The second advertisement would improve the scalability of OSPF by reducing the number of routes that are advertised across area boundaries. However, in practice, this requires manual configuration on the border routers.



Coming up next, we'll get an introduction to interdomain routing and the only interdomain routing protocol in use today: BGP

# Interdomain Routing: Border Gateway Protocol

In this lesson, we'll study the border gateway protocol.

## WE'LL COVER THE FOLLOWING



- Introduction
- The Role of BGP
- Advertising BGP Route Information
  - Internal Routers & Gateway Routers
  - Propagating Information
- Quick Quiz!

## Introduction #

Just the way packets need to move around *within* a domain or autonomous system, packets need to move across them too. To enable all domains to communicate with each other, they need to be talking in the same language, in other words, they need to be using the same protocol. Hence, there exists just one interdomain routing protocol and it's called the **Border Gateway Protocol (BGP)**. We'll spend a bit of time on this one as it is essentially the glue that holds the Internet together!

## The Role of BGP #

BGP routers have two key responsibilities:

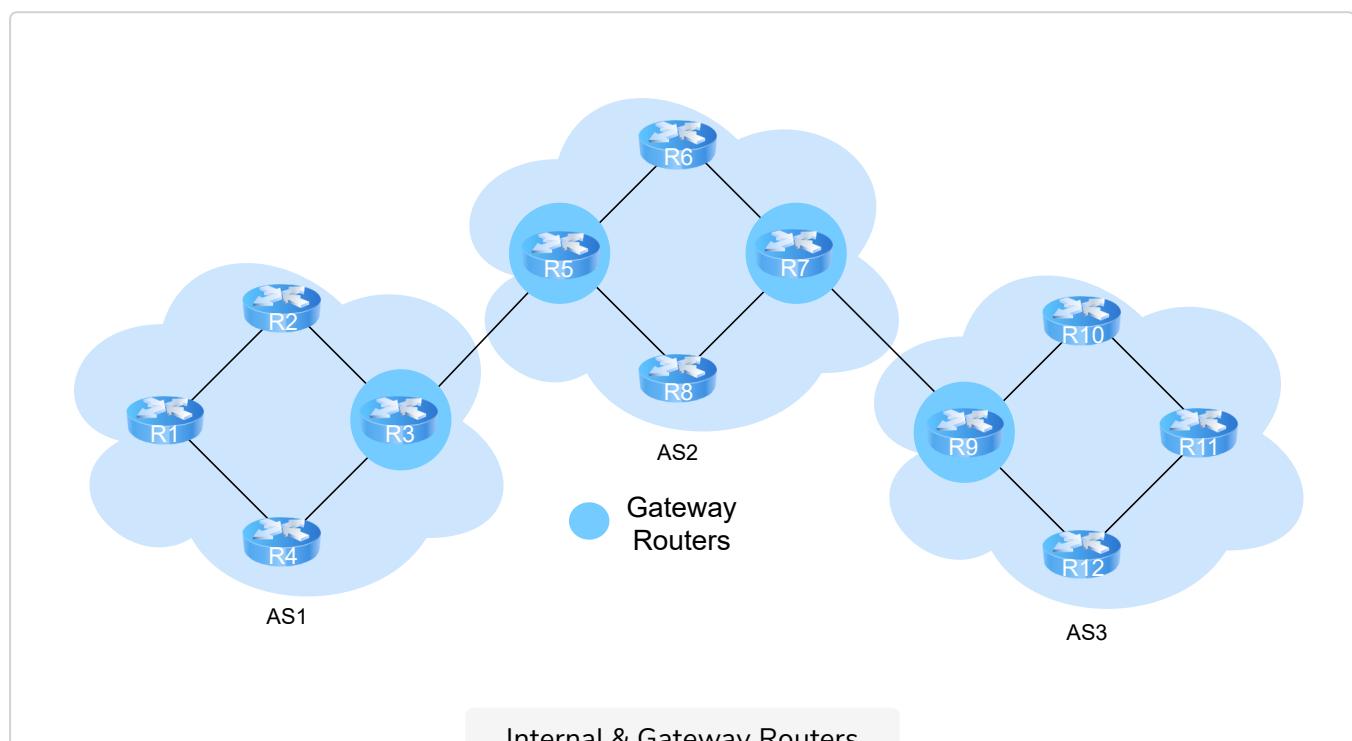
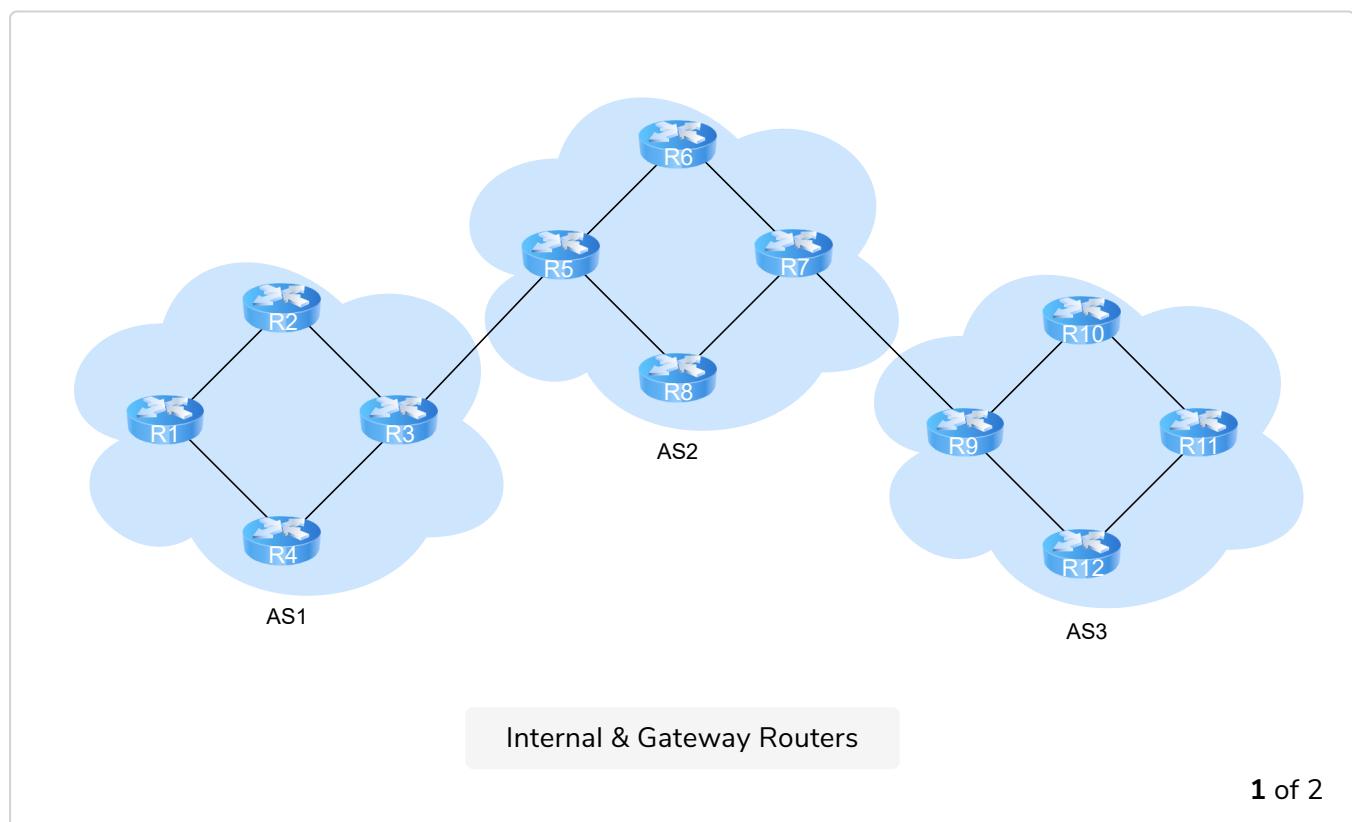
1. To make **each domain announce its presence to other domains**. If it weren't for this, no domain would know about the existence of other domains, and each domain would be an isolated island. This is done by each domain obtaining prefix reachability information from neighboring domains.
2. To determine the **best route to each prefix**.

Let's study how each objective is achieved.

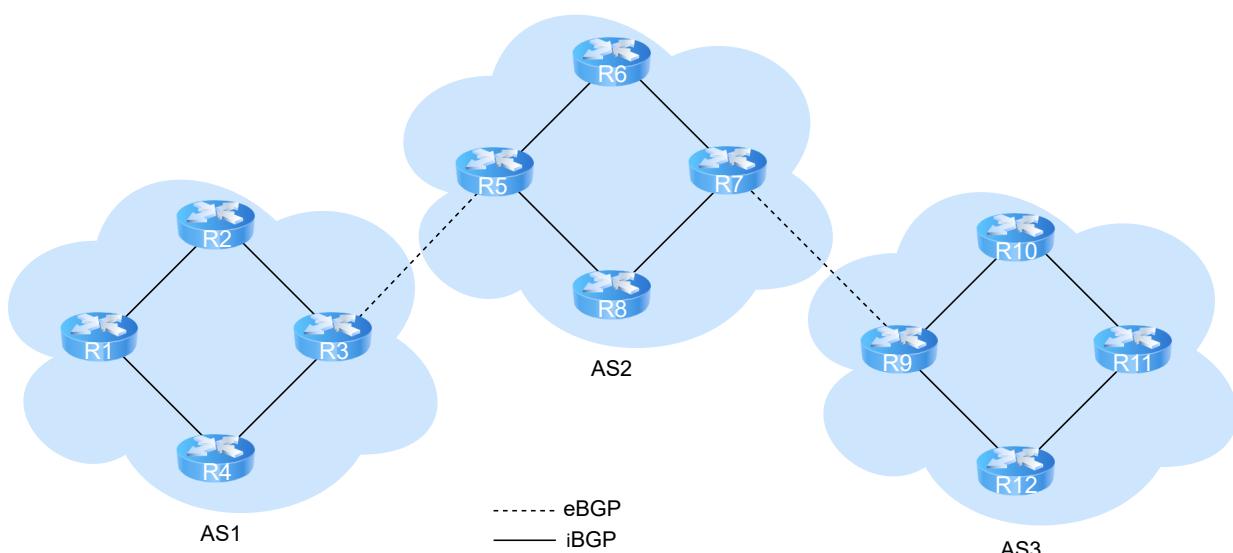
## Advertising BGP Route Information #

### Internal Routers & Gateway Routers #

- An **internal router** is **only** connected to routers within the domain it resides in.
- A **gateway router** is connected directly to one or many routers in one or many domains.



- For a domain to advertise its existence with BGP, gateway routers establish a TCP connection on port 179.
- In BGP, each domain is identified by a unique Autonomous System (AS) number.
- The BGP routers exchange routing information with each other over this TCP connection. The connection and the information exchange is known as a **BGP session**.
- A BGP session **across two domains** is called an **eBGP session**.
- A BGP session **within a domain** is called an **iBGP session**.



external & internal border gateway protocol sessions

## Propagating Information #

In order to propagate BGP information, both iBGP and eBGP links are used. In the example above, consider router *R12* advertising a prefix *P12* to *AS1* and *AS2*.

- The gateway router *R9* will first send an eBGP message like "*AS3 P12*" to the gateway router *R7*. This message is in the form "**via AS.**" So it is essentially saying "*P12* is reachable via *AS3*."

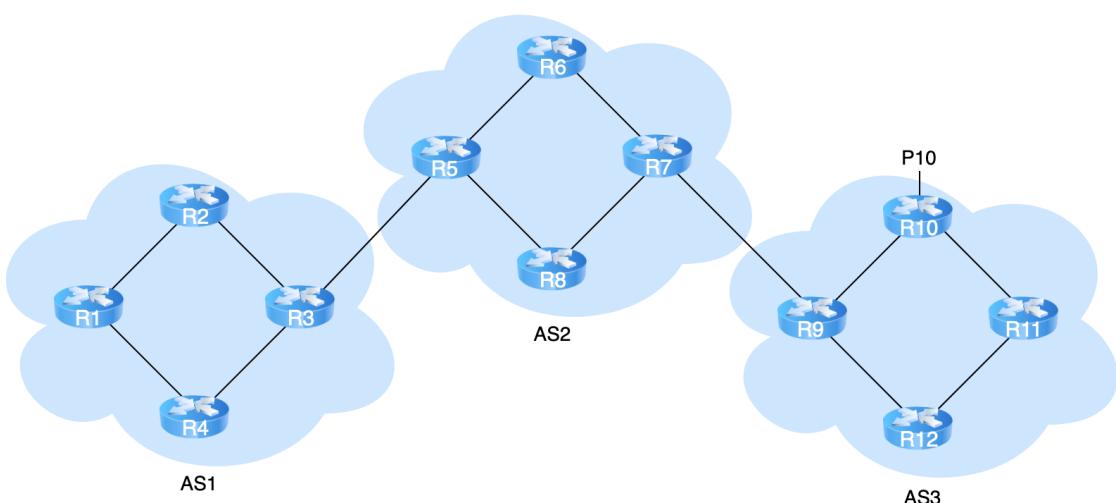
- The gateway router  $R7$  then will send the iBGP message "AS3  $P12$ " to all of the other routers in AS2. This includes the gateway router  $R5$ .
- The gateway router  $R5$  then sends the eBGP message "AS2 AS3  $P12$ " to the gateway router  $R3$ . This message says " $P12$  is reachable via AS2 then through AS3".
- Finally, the gateway router  $R3$  uses an iBGP session to send the message "AS2 AS3  $P12$ " to all the routers in AS1.
- Each router in the network now knows about the presence of  $P12$  and how to reach it.

Lastly, **only a few routers in a network might speak BGP**. Others can be dedicated to intra-domain routing. Routes learned through the intradomain routing protocol such as OSPF are injected into the BGP process on a BGP speaking router, which passes it on. Also unlike OSPF border routers, BGP neighbors don't need to be physically connected directly.

## Quick Quiz! #

1

What would be the AS path to the prefix  $P10$  in AS1?



COMPLETED 0%

1 of 2



---

In real life, however, many different paths exist from each router to all other routers. How do we pick the best one? We'll look at how BGP does that in the next lesson!

# Border Gateway Protocol: Determining the Best Routes

We left the last lesson off at the question: how do routers pick a path to reach a specific destination out of a given number of paths?

## WE'LL COVER THE FOLLOWING ^

- Terminology
- Hot Potato Routing
- Route Selection Algorithm
- Quick Quiz!

## Terminology #

Before addressing this question, we need to familiarize ourselves with some more BGP jargon.

1. **BGP Attributes:** certain attributes are sent along with path advertisement. Two of the more important ones are as follows:
  - i. **AS-PATH:** When a BGP router advertises a route towards a prefix, it announces the IP prefix and the **path used to reach this prefix**. In BGP, each domain is identified by a unique Autonomous System (AS) number. The AS-Path is just a list of AS numbers that you need to go through to get to a prefix. AS-PATHs are also used to avoid loops by domains rejecting advertisements that already contain the domain's AS number.
  - ii. **NEXT-HOP:** the next hop is the IP address of the relevant interface of the router that starts a certain AS-PATH.

BGP routes are written as a combination of many things out of which we will consider three:

1. NEXT-HOP

2. AS-PATH

3. Destination Prefix

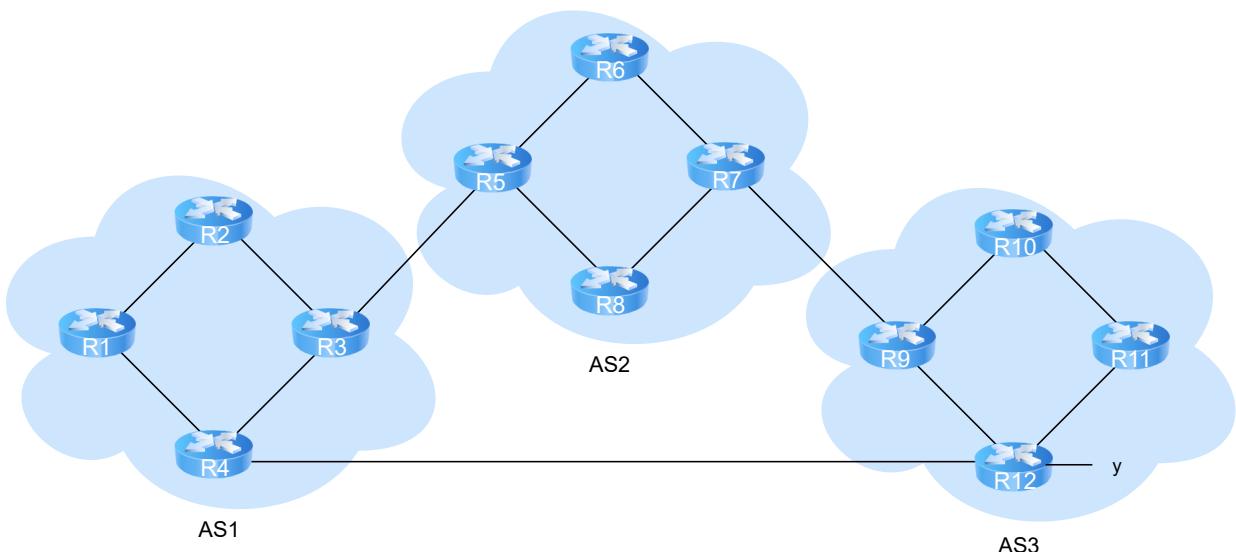
## Hot Potato Routing #

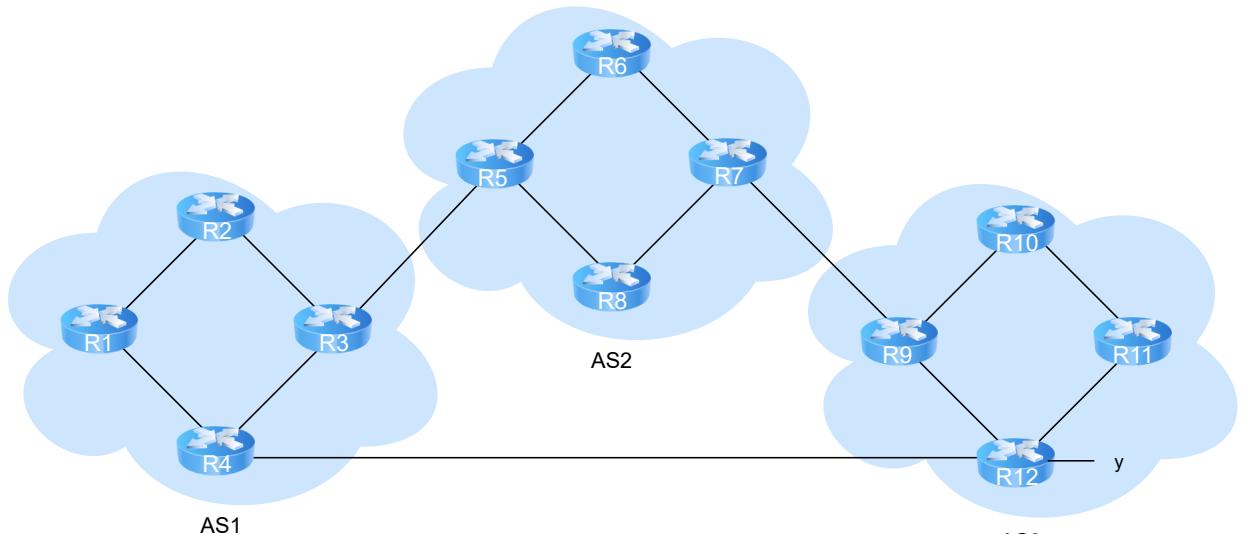
Assuming that a **router has two paths to send a packet over to another domain**, **Hot Potato Routing** would calculate the best path like so:

1. The router will calculate the cost to reach the NEXT-HOP router of each given path. The cost can be calculated using intradomain routing protocols like OSPF.
2. The path with the NEXT-HOP router that is least costly to send a packet to is chosen.

Essentially, a router has learned routes to a prefix from multiple BGP border routers. It consults its intra domain routing information to determine the border router that is reachable with the least cost. This is akin to getting rid of the packet as quickly as possible, without paying any attention to the fact that some other border router might offer a lower overall distance to the destination.

Let's take a modification of our previous example to illustrate how this works in practice.

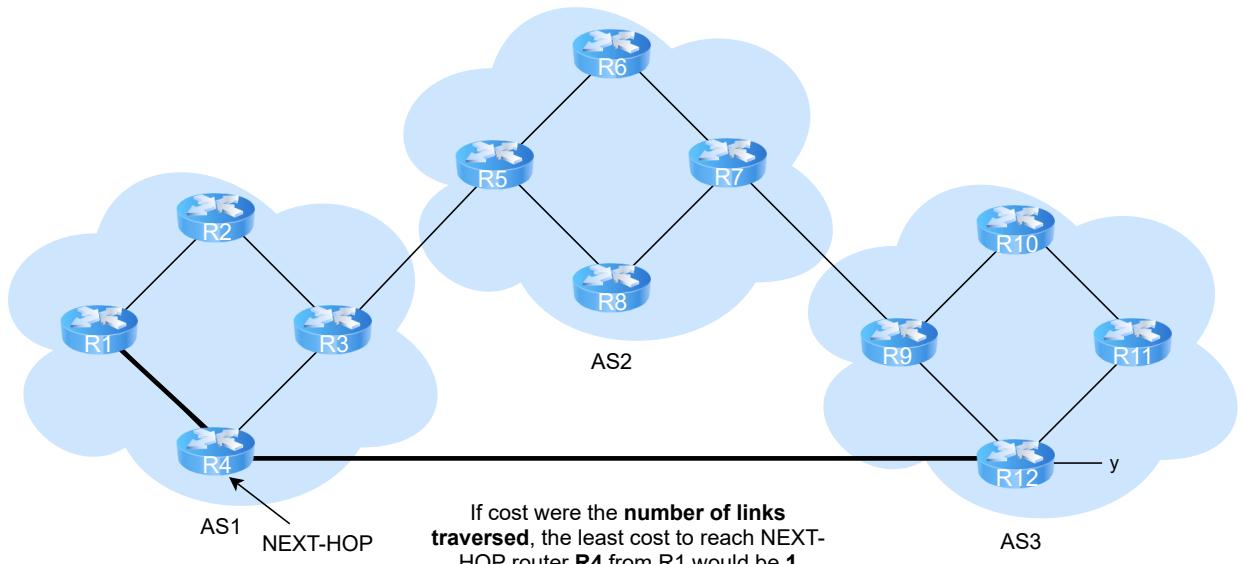




Suppose we are trying to reach prefix y from router R1 in the given topology

### Hot Potato Routing

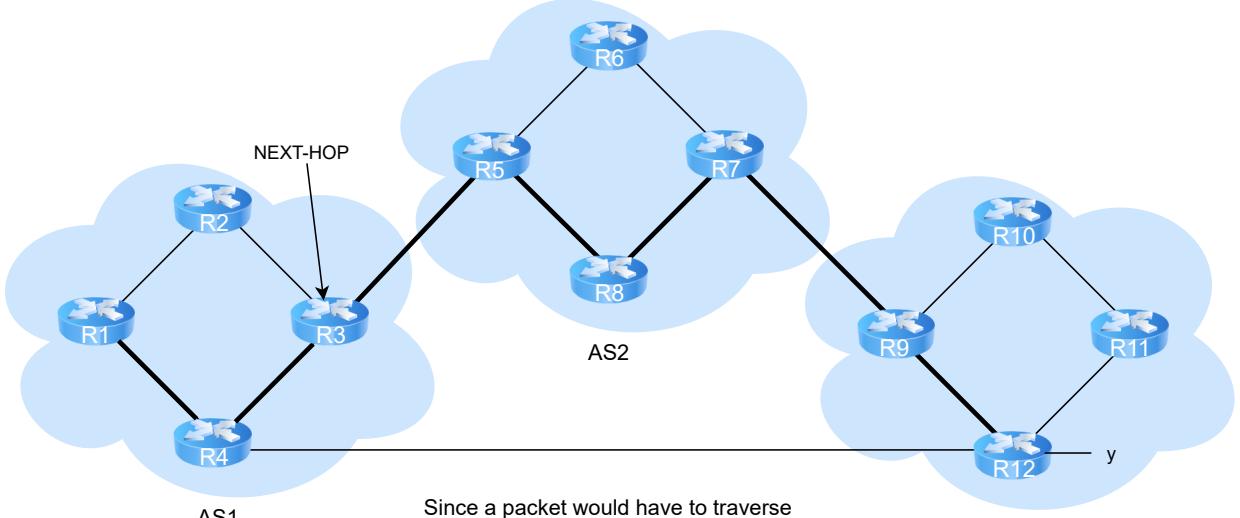
2 of 5



If cost were the **number of links traversed**, the least cost to reach NEXT-HOP router R4 from R1 would be 1

### Hot Potato Routing

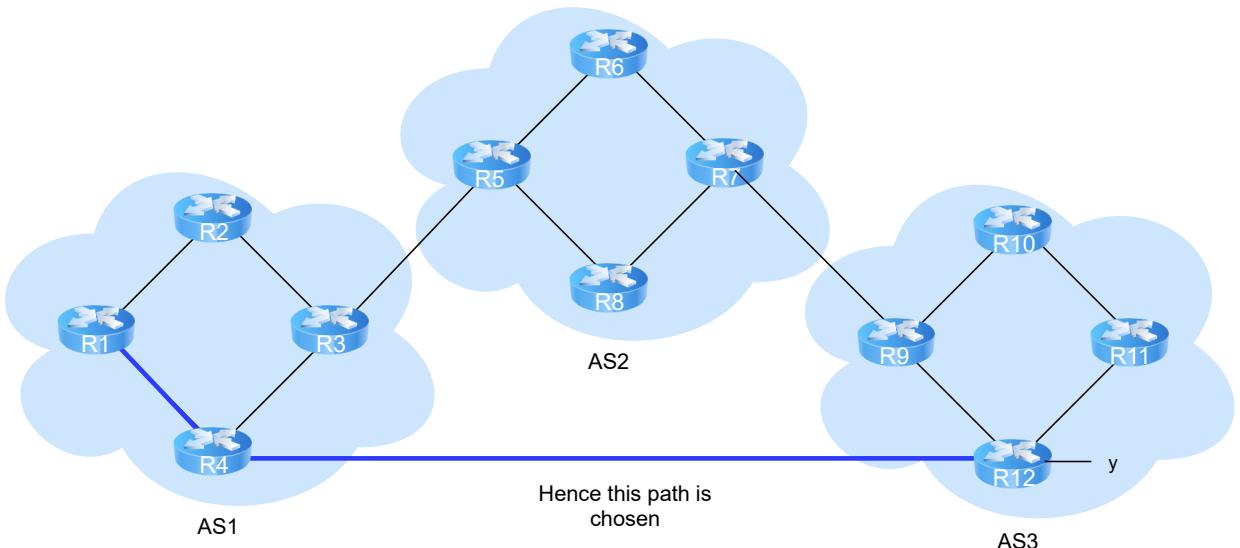
3 of 5



Since a packet would have to traverse 2 links to reach the NEXT-HOP router, R3, the cost for this path is 2 which is greater than the first one we looked at

#### Hot Potato Routing

4 of 5



Hence this path is chosen

#### Hot Potato Routing

5 of 5



Note that this is a selfish algorithm, as it only cares about reducing the immediate cost and does not consider the cost of the path outside of its domain.

Furthermore, two routers within the same domain may select two different

Furthermore, two routers within the same domain may select two different paths to reach the same destination. An example would be  $R2$  that would select the path via AS2 to the prefix  $y$ , and  $R1$  that would undeniably bypass AS2 and reach  $y$  via AS3.

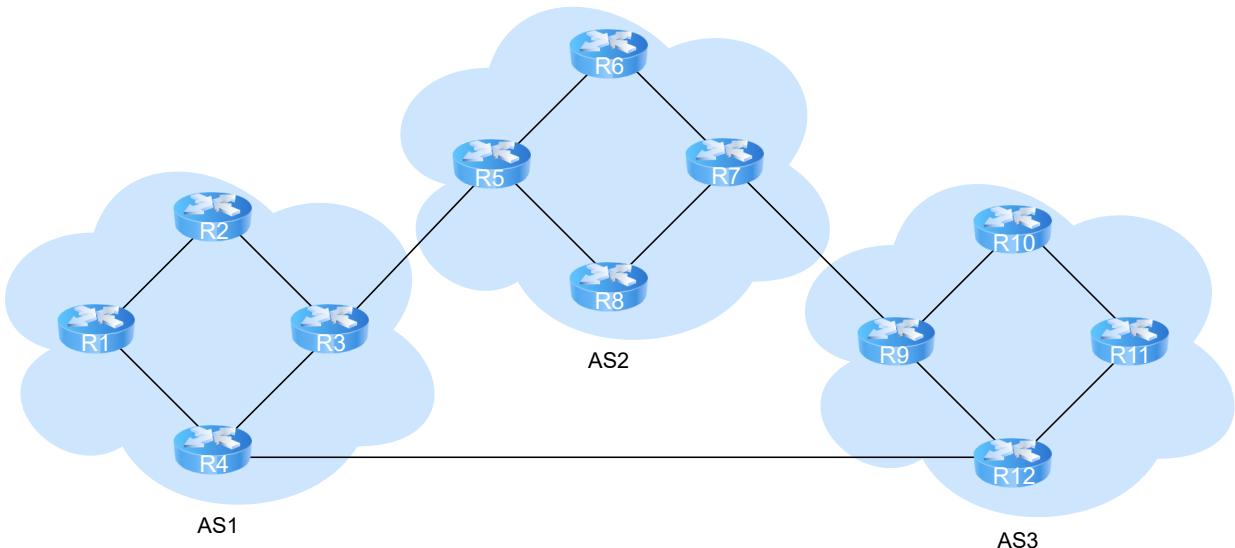
BGP does use hot potato routing in some cases, but that is based on a more sophisticated algorithm described as follows.

# Route Selection Algorithm #

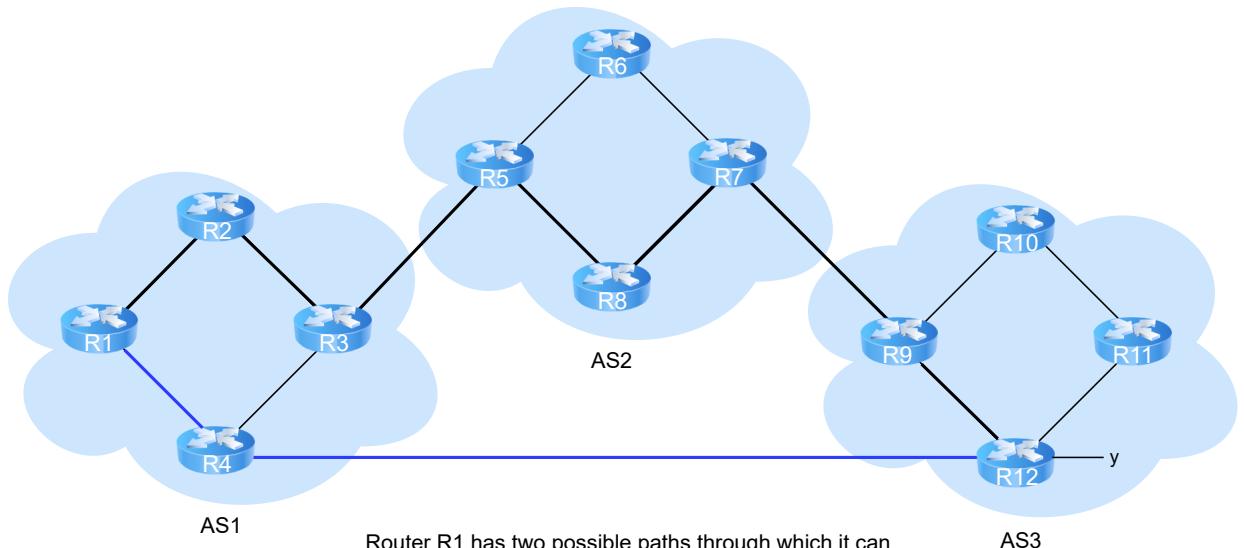
BGP uses a slightly more complicated algorithm in the real world.

To pick the best path out of two or more paths to the same prefix, BGP uses the following rules to eliminate each path until the best one is left:

1. Each route is assigned a **local preference** value which is determined entirely by how the policy of each individual domain is set up by the network administrator. The route with the highest local preference value is chosen.
    - Example:



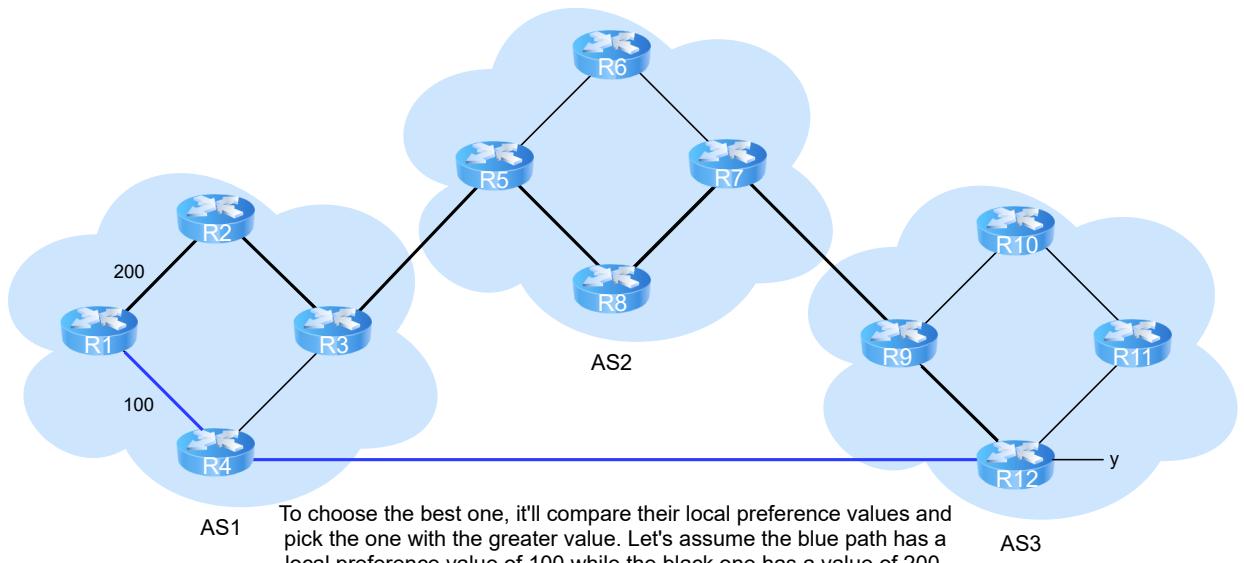
## BGP route selection



Router R1 has two possible paths through which it can reach prefix y. One through AS2 and the other through AS3.

#### BGP route selection

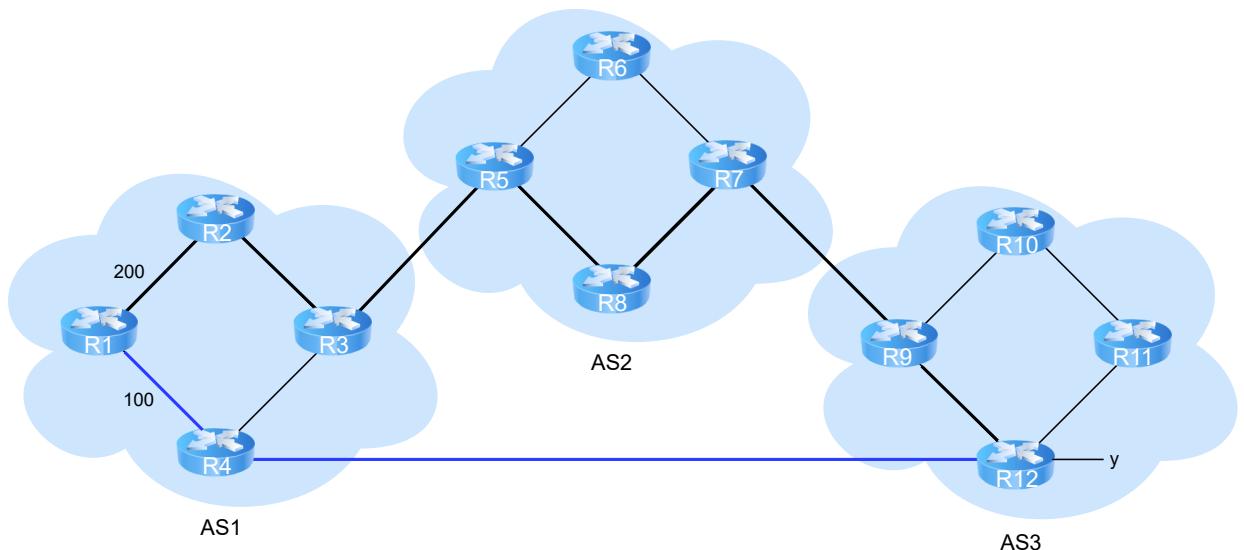
2 of 4



To choose the best one, it'll compare their local preference values and pick the one with the greater value. Let's assume the blue path has a local preference value of 100 while the black one has a value of 200.

#### BGP route selection

3 of 4



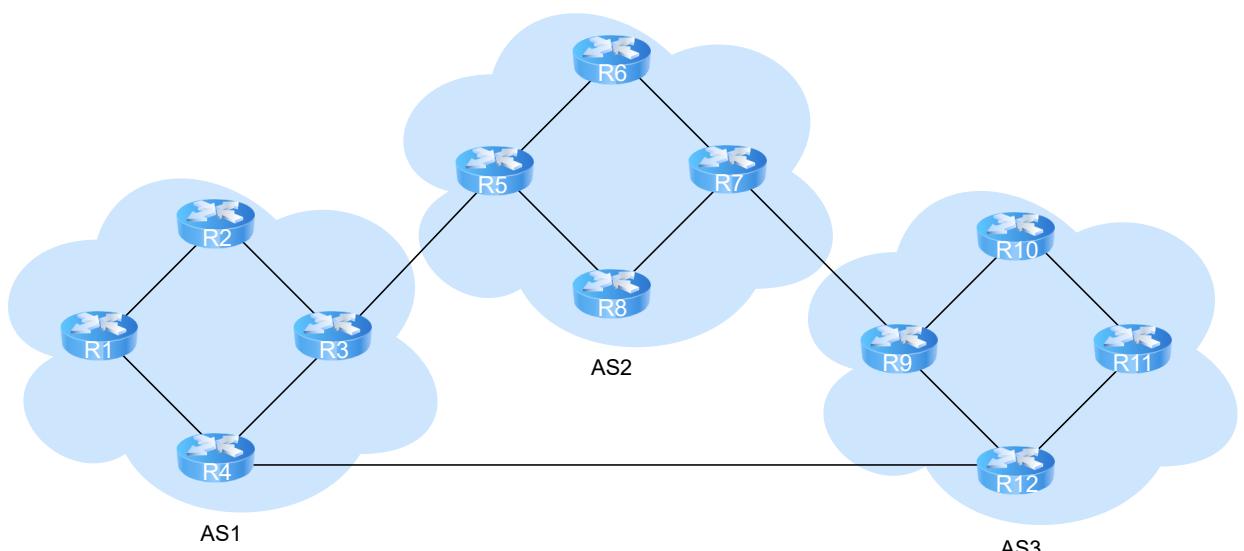
BGP route selection

4 of 4



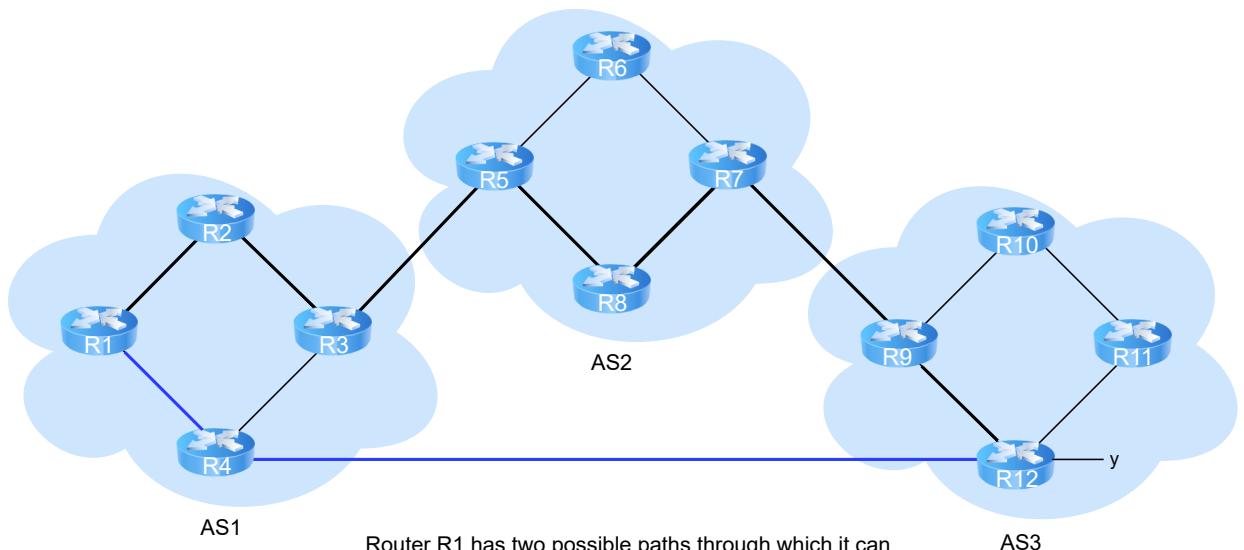
2. If several routes have the same highest value, then the one with **the shortest AS-PATH is selected.**

- Example:



BGP route selection

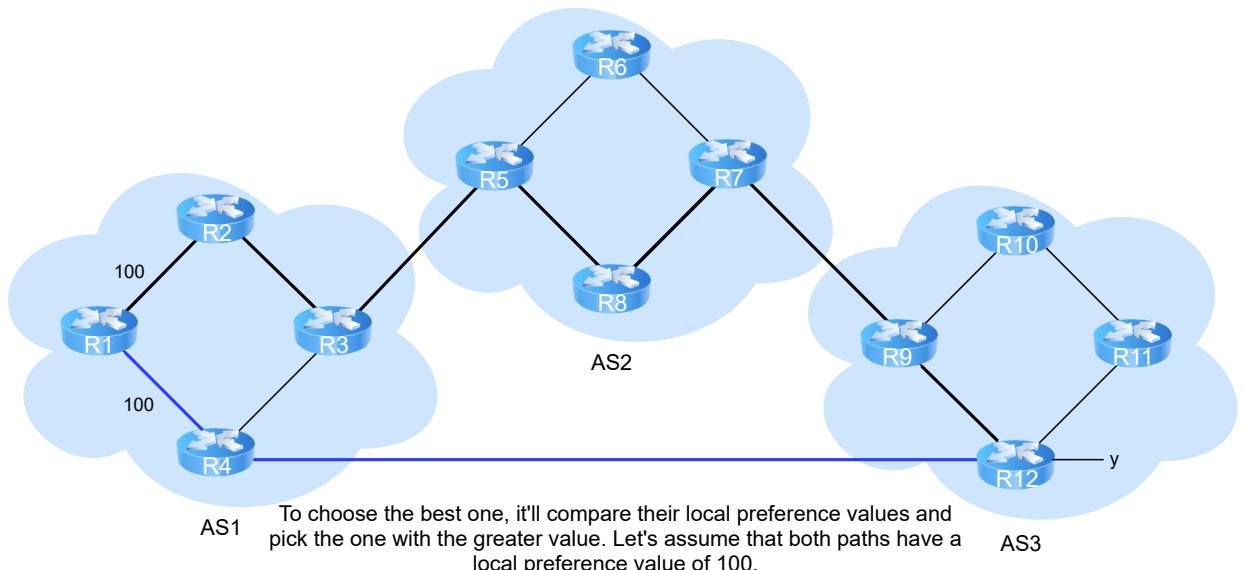
1 of 4



Router R1 has two possible paths through which it can reach prefix y. One through AS2 and the other through AS3.

#### BGP route selection

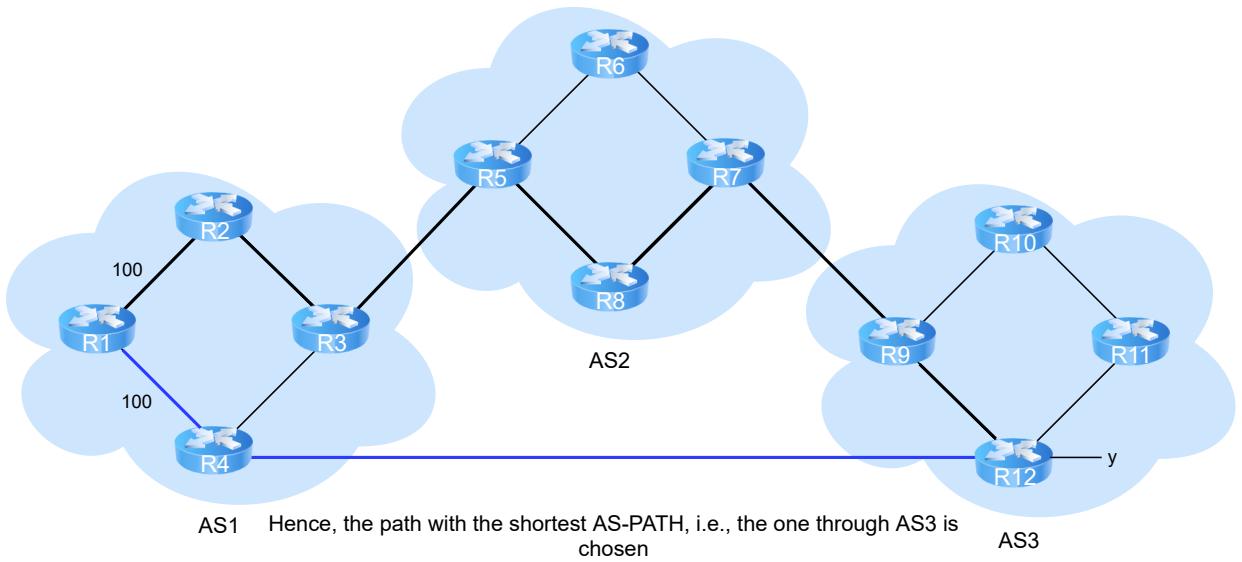
2 of 4



To choose the best one, it'll compare their local preference values and pick the one with the greater value. Let's assume that both paths have a local preference value of 100.

#### BGP route selection

3 of 4

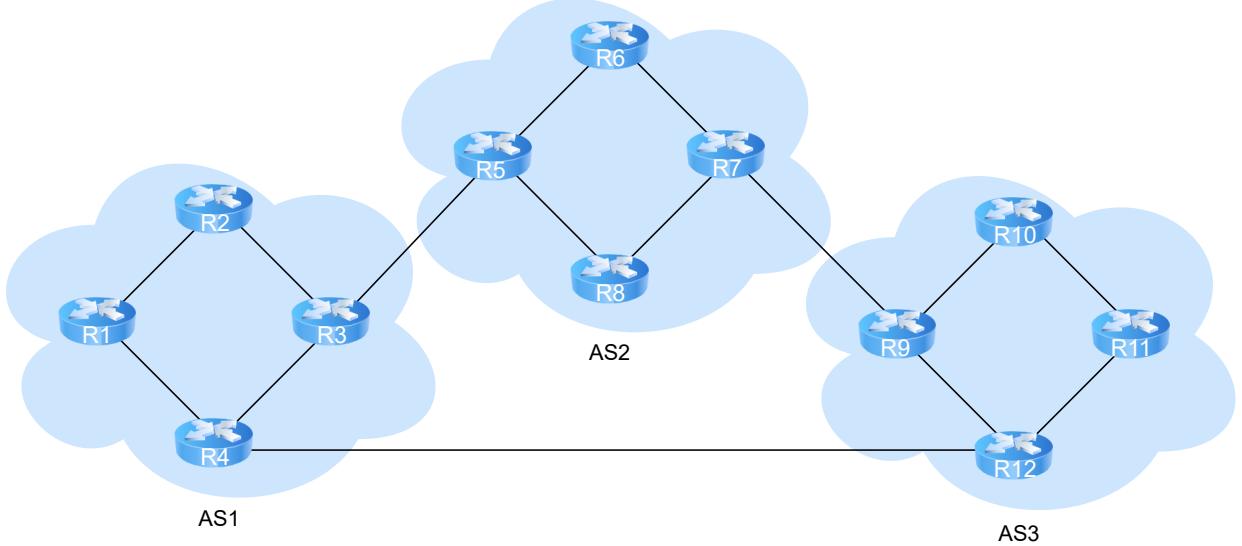


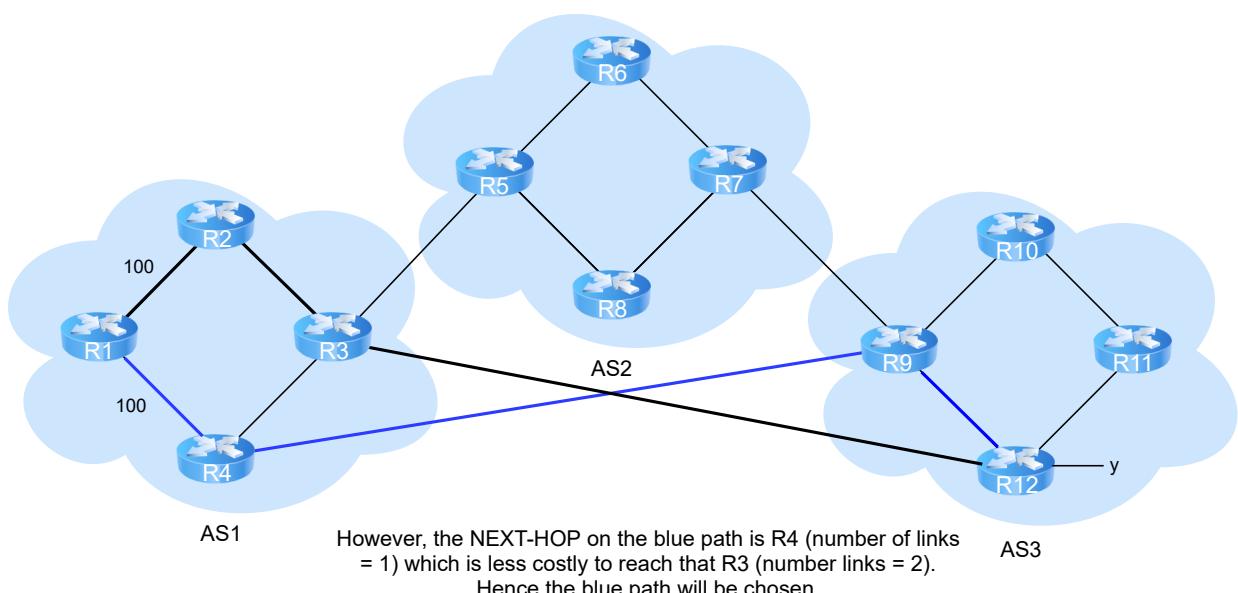
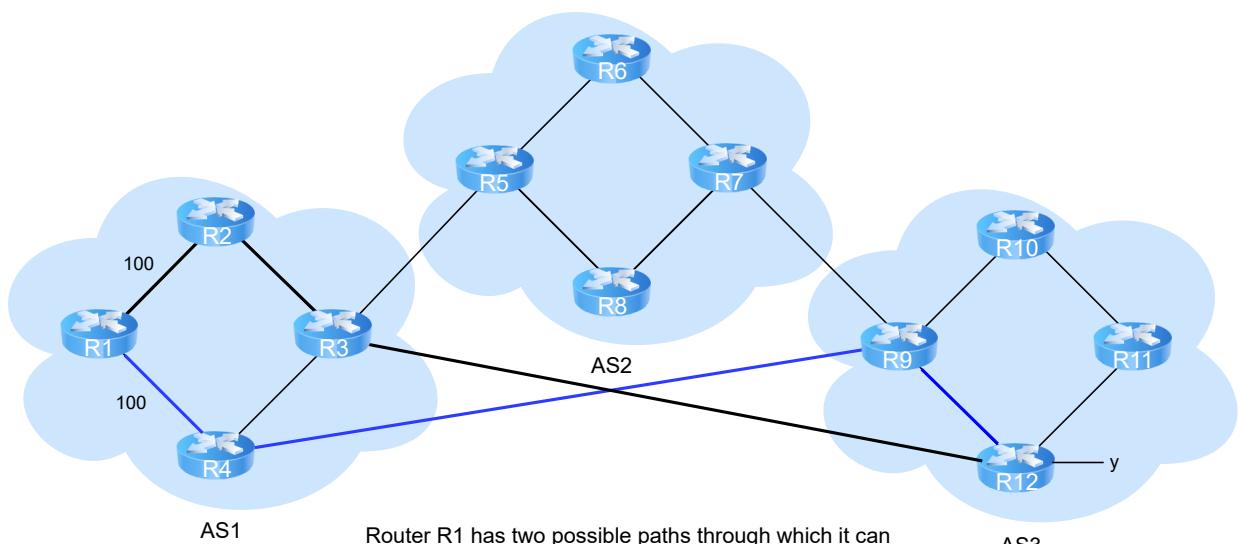
#### BGP route selection

4 of 4



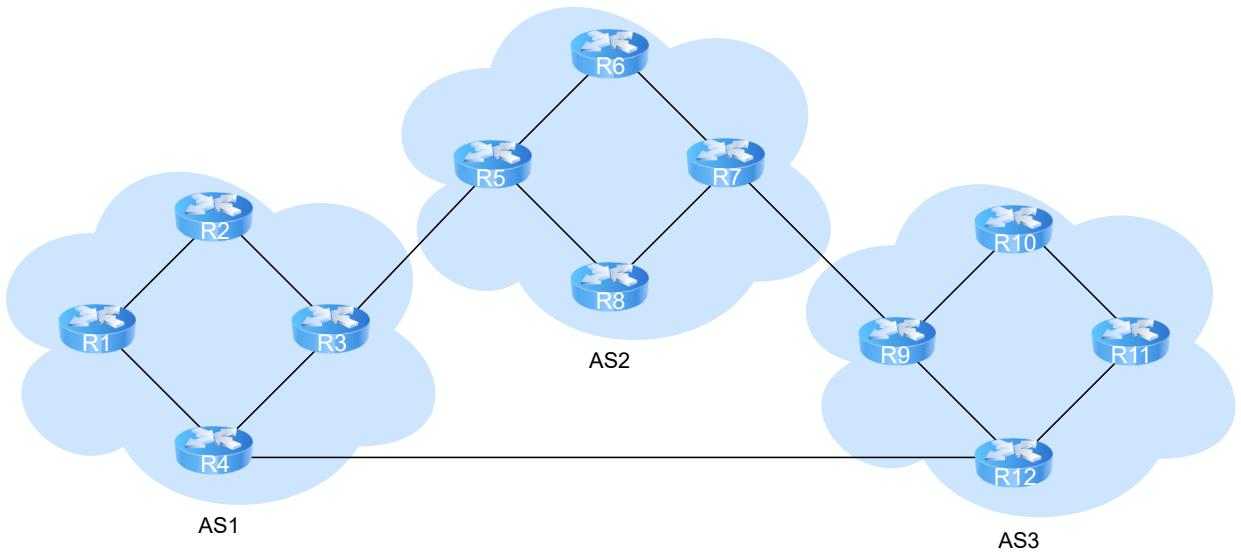
3. Prefer routes with the smallest MED.
4. Prefer routes learned via eBGP sessions over routes learned via iBGP sessions.
5. If we still have a tie, then **hot potato routing is applied** and the one with the least costly NEXT-HOP is chosen.





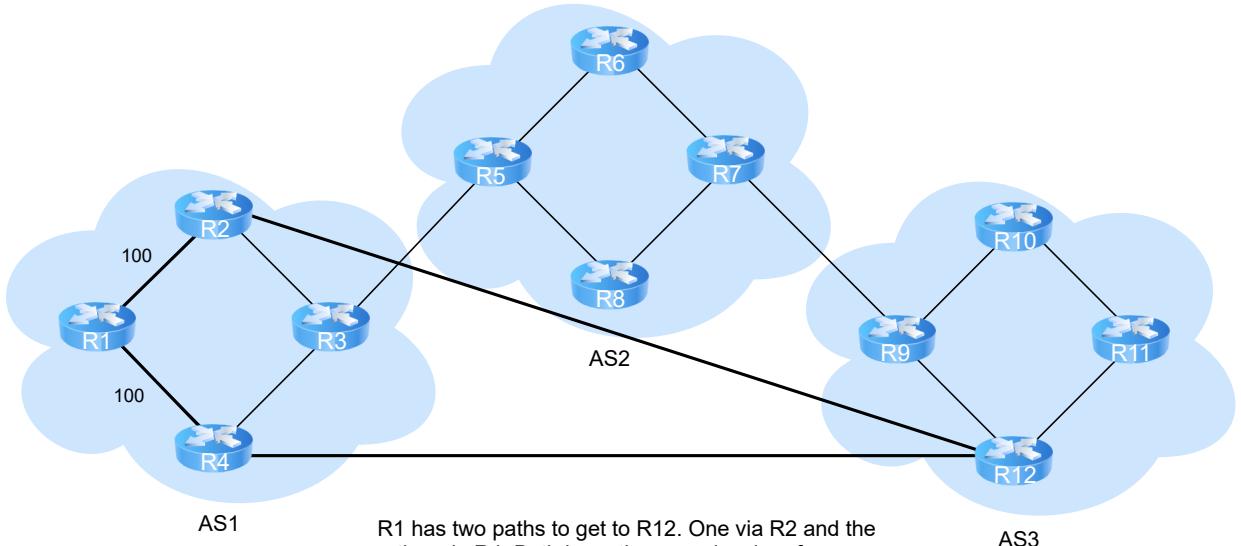
- If more than one route still remains, routes learned from the router with the lowest router ID is preferred.

- Example



BGP route selection

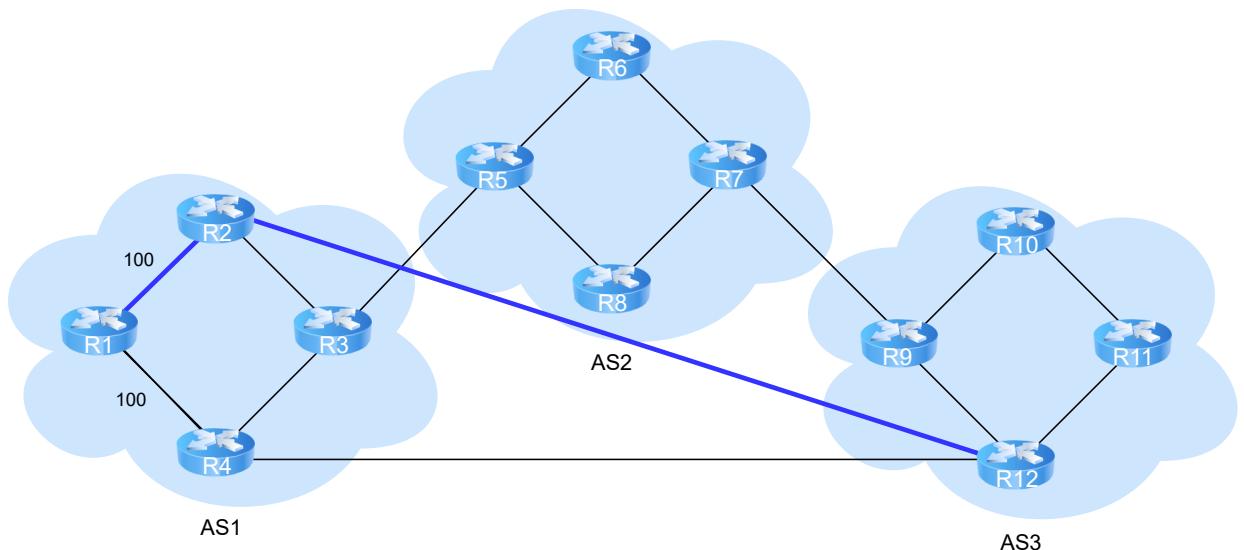
1 of 3



R1 has two paths to get to R12. One via R2 and the other via R4. Both have the same local preference number, the same number of AS-HOPS, and the same cost to reach the border router.

BGP route selection

2 of 3



Since R2 has a lower ID, the path through it is chosen.

BGP route selection

3 of 3



## Quick Quiz! #

1

Hot potato routing always results in the overall least cost path being chosen.

COMPLETED 0%

1 of 2



This concludes our discussion of the network layer! We'll get into the link layer next.



# What is The Data Link Layer?

This lesson will give us a quick introduction to the data link layer!

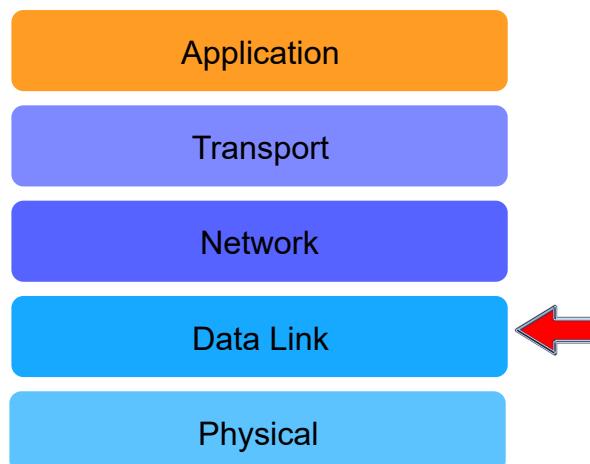
## WE'LL COVER THE FOLLOWING



- You Are Here!
- Responsibilities of The Data Link Layer
- Types of Data Link Layers
- What To Expect
- Quick Quiz!

## You Are Here! #

Let's zoom out and have a look at the big picture.



you  
are  
here

## Responsibilities of The Data Link Layer #

The data link layer receives packets from the network layer and deals with **providing hop to hop communication** or communication between entities that are **directly connected by a physical link**.

In other words, it makes intelligible communication possible over a physical link that just transports 0s and 1s between two directly connected hosts.

# Types of Data Link Layers #

The data link layer is the lowest layer of the reference model that we will discuss in detail. Data link layer protocols exchange **frames** that are transmitted through the physical layer. There are **three main types of data link layers**:

1. The *simplest* data link layer type is one that has only **two communicating systems connected directly through the physical layer** also known as the **point-to-point data link layer**. This type of layer can either provide an unreliable service or a reliable service. The unreliable service is frequently used above physical layers (e.g., optical fiber, twisted pairs) that have a low bit error ratio, while reliability mechanisms are often used in wireless networks to recover locally from transmission errors.
2. The second type of data link layer is the one used in Local Area Networks (LAN) called **Broadcast multi-access**. Both end-systems and routers can be connected to a LAN.
  - An important difference between point-to-point data and Broadcast multi-access is that in a Broadcast multi-access, **each communicating device is identified by a unique data link layer address**. This address is usually embedded in the hardware of the device and different types of LANs use different types of data link layer addresses. However, since there is only one party at the “other end of the wire,” in point-to-point, there is no ambiguity what entity should receive a frame that is transmitted, thus there is no need for addressing.
  - A communicating device attached to a LAN can send a data link frame to any other communication device that is attached to the same LAN.
  - Most LANs also **support special broadcast and multicast data link layer addresses**. A frame sent to the broadcast address of the LAN is delivered to all communicating devices that are attached to the LAN. The multicast addresses are used to send a frame to one specific group.

3. The third type of data link layer is used in **Non-Broadcast Multi-Access (NBMA) networks**. These networks are used to interconnect devices like a LAN. All devices attached to an NBMA network are identified by a unique data link layer address.

- The main difference between an NBMA network and a traditional LAN is that the NBMA service **only supports unicast** and **supports neither broadcast nor multicast**.
- ATM, Frame Relay and X.25 are examples of NBMA.



**Note** You may be wondering why some functions are repeated at multiple layers. The reason is that different network stacks may implement different services at each layer. So, it is possible that a data link layer implementation may only provide unreliable frame transport. Furthermore, suppose that the network implements IP. In such a case, in order to have reliable data transport, the transport layer must provide its own reliability.

## What To Expect #

This chapter is organized as follows.

1. We will first discuss the principles of the data link layer as well as the services that it uses from the physical layer.
2. We'll then describe in more detail several **Medium Access Control algorithms** that are used in Local Area Networks to regulate the access to the shared medium.
3. Finally, we'll discuss Ethernet, the ubiquitous data link layer type.

## Quick Quiz! #

COMPLETED 0%

1 of 2



---

Let's study some key principles of the link layer next!

# Principles of The Data Link Layer: The Framing Problem

We'll discuss some key principles of the data link layer in this lesson.

## WE'LL COVER THE FOLLOWING



- Limitations Imposed Upon The Data Link Layer
  - Limitations Imposed by The Data Link Layer
  - Limitations Imposed By The Physical Layer
- The Framing Problem
  - Solution #1: Idle Physical Layer
  - Solution #2: Multi-symbol Encodings
  - Solution #3: Stuffing
    - Bit Stuffing
    - Example
    - Character Stuffing
    - Examples
- Disadvantages of Stuffing
- Quick Quiz!

## Limitations Imposed Upon The Data Link Layer #

### Limitations Imposed by The Data Link Layer #

The data link layer uses the service provided by the physical layer. Although there are many different implementations of the physical layer from a technological perspective, they all provide a service that enables the data link layer to send and receive bits between directly connected devices.

Most data link layer technologies **impose limitations on the size of the frames**:

1. Some technologies only impose a maximum frame size.
2. Others enforce both minimum and maximum frame sizes.
3. Finally, some technologies only support a single frame size. In this case, the data link layer will usually include an adaptation sub-layer to allow the network layer to send and receive variable-length packets. This adaptation layer may include fragmentation and reassembly mechanisms.

## Limitations Imposed By The Physical Layer #

The physical layer service facilitates the sending and receiving of bits, but it's usually far from perfect:

- The physical layer **may change the value of a bit** being transmitted due to any reason, e.g., electromagnetic interferences.
- The Physical layer **may deliver more bits** to the receiver than the bits sent by the sender.
- The Physical layer **may deliver fewer bits** to the receiver than the bits sent by the sender.

## The Framing Problem #

The data link layer must allow end systems to exchange frames containing packets despite all of these limitations.

On point-to-point links and Local Area Networks, the first problem to be solved is **how to encode a frame as a sequence of bits** so that the receiver can easily recover the received frame **despite the limitations of the physical layer**. This is the **framing problem**. It can be defined as: "*How does a sender encode frames so that the receiver can efficiently extract them from the stream of bits that it receives from the physical layer?*" Several solutions have been proposed and are used in practice in different data link layer technologies.

## Solution #1: Idle Physical Layer #

A first solution to solve the framing problem is to **require the physical layer to remain idle for some time after the transmission of each frame**. These idle periods can be detected by the receiver and serve as a marker **to indicate frame boundaries**.

**frame boundaries.**

Unfortunately, this solution is **not sufficient for two reasons**:

1. First, some physical layer implementations **can't remain idle** and always need to transmit bits.
2. Second, inserting an idle period between frames **decreases the maximum bandwidth that can be achieved** by the data link layer.

## Solution #2: Multi-symbol Encodings #

Some physical layer implementations provide an alternative to this idle period. **All physical layer types are able to send and receive physical symbols that represent values 0 and 1.** Also, **several physical layer types are able to exchange other physical symbols as well.** Some technologies use these other special symbols as markers for the beginning or end of frames. **For example, the Manchester encoding used in several physical layers can send four different symbols.** Apart from the encodings for 0 and 1, the Manchester encoding also **supports two additional symbols: InvH and InvB.**

## Solution #3: Stuffing #

Unfortunately, multi-symbol encodings cannot be used by all physical layer implementations and a generic solution with which any physical layer that is able to transmit and receive only 0s and 1s works is required. This **generic solution is called stuffing** and two variants exist:

1. Bit stuffing
2. Character stuffing.

To enable a receiver to easily delineate the frame boundaries, these two techniques **reserve special bit strings as frame boundary markers** and encode the frames such that these special bit strings do not appear inside the frames.

### Bit Stuffing #

Bit stuffing **reserves a special bit pattern**, for example, the 01111110 bit string as the frame boundary marker. However, if the same bit pattern occurs in the data link layer payload, it must be modified before being sent,

otherwise, the receiving data link layer entity will detect it as a start or end of frame.

Assuming that the 0111110 pattern is used as the frame delimiter, a frame is sent as follows:

1. First, the sender transmits the marker, i.e. 0111110.
2. Then, it sends all the bits of the frame and inserts an additional bit set to 0 after each sequence of five consecutive 1 bits. This ensures that the sent frame never contains a sequence of six consecutive bits set to 1. As a consequence, the marker pattern cannot appear inside the frame sent.
3. The marker is also sent to mark the end of the frame.
4. The receiver performs the opposite to decode a received frame.
  - It first detects the beginning of the frame thanks to the 0111110 marker.
  - Then, it processes the received bits and counts the number of consecutive bits set to 1.
  - If a 0 follows five consecutive bits set to 1, this bit is removed since it was inserted by the sender.
  - If a 1 follows five consecutive bits set to 1, it indicates a marker if it is followed by a bit set to 0.

The table below illustrates the application of bit stuffing to some frames.

Original frame	Transmitted frame
0001001001001001001000011	01111100001001001001001000 0110111110
01111110	0111111001111101001111110

#### Example #

For example, consider the transmission of 011011111111111111110010.

1. The sender will first send the 0111110 marker followed by 01101111.

2. After these five consecutive bits set to 1, it inserts a bit set to 0 followed by 11111.
3. A new 0 is inserted, followed by 11111.
4. A new 0 is inserted followed by the end of the frame 110010 and the 01111110 marker.

Have a look at the slides for a visual demonstration of bit stuffing.

011011111111111111110010

Example of Bit Stuffing

1 of 7

01111110+011011111111111111110010

Starting segment added

Example of Bit Stuffing

2 of 7

01111110011011111+0+1111111111110010

0 inserted because 5 1s  
are encountered

Example of Bit Stuffing

3 of 7

01111110011011111011111+0+11111110010

0 inserted because 5 1s  
are encountered

Example of Bit Stuffing

4 of 7

0111110011011111011111011111+0+110010

0 inserted because 5  
1s are encountered

Example of Bit Stuffing

5 of 7

01111100110111110111110111110110010+01111110

Ending sequence  
appended

Example of Bit Stuffing

6 of 7

0111110011011111011111011111011001001111110

Final sequence

Example of Bit Stuffing

7 of 7



You might be wondering **what happens if the sequence 011111010 appears in the data?** Bit stuffing inserts a 0 bit after every consecutive five 1s in the payload. What does that give us? Well, the payload could be one of the following:

- 0111 11 00
- 0111 11 01
- 0111 11 10
- 0111 11 11

These will be encoded to, respectively:

- 0111 11 0 00

- 0111 11 0 01

- 0111 11 0 10

- 0111 11 0 11 So, in any case, **the receiver can only expect 0111110 at the beginning and end of frame.** If it receives five consecutive 1s, followed by a 0, it removes the 0 as redundancy. If it receives six consecutive 1s, there must've been an error.

## Character Stuffing #

This technique operates on frames that contain an integer number of characters of a fixed size, such as 8-bit characters. Some characters are used as markers to delineate the frame boundaries. Many character stuffing techniques use the **DLE**, **STX** and **ETX** characters of the ASCII character set. **DLE STX** is used to mark the beginning of a frame, and **DLE ETX** is used to mark the end of a frame.

If the character DLE appears in the payload, the data link layer entity prepends DLE as an escape character before the transmitted DLE character from the payload. This ensures that none of the markers can appear inside the transmitted frame. The receiver detects the frame boundaries and removes the second DLE when it receives two consecutive DLE characters.



**Note:** Software implementations prefer to process characters than bits, hence software-based data link layers usually use **character stuffing**.

## Examples #

For example, to transmit frame **1 2 3 DLE STX 4**:

1. A sender will first send **DLE STX** as a marker
2. Followed by **1 2 3 DLE**
3. Then, the sender transmits an additional **DLE** character
4. Followed by **STX 4** and the **DLE ETX** marker
5. The final string is: **DLE STX 1 2 3 DLE DLE STX 4 DLE ETX**

Have a look at the following table for more details:

Original frame	Transmitted frame
1 2 3 4	DLE STX 1 2 3 4 DLE ETX
1 2 3 DLE STX 4	DLE STX 1 2 3 DLE DLE STX 4 DLE ETX
DLE STX DLE ETX	DLE STX DLE DLE STX DLE DLE ETX DLE ETX

 **Did You Know?** DLE is the bit pattern 00010000, STX is 00000010 and ETX is 00000011.

1 2 3 DLE STX 4

Example of Character Stuffing

1 of 6

DLE STX 1 2 3 DLE STX 4

DLE STX added as the starting sequence

Example of Character Stuffing

2 of 6

DLE STX 1 2 3 DLE STX 4

The sequence DLE is encountered in the payload

Example of Character Stuffing

3 of 6

## DLE STX 1 2 3 + DLE + DLE STX 4

Another 'DLE' is added as an escape sequence

Example of Character Stuffing

4 of 6

## DLE STX 1 2 3 DLE DLE STX 4 DLE ETX

DLE ETX is added as the ending sequence

Example of Character Stuffing

5 of 6

## DLE STX 1 2 3 DLE DLE STX 4 DLE ETX

Final sequence

Example of Character Stuffing

6 of 6



## Disadvantages of Stuffing #

1. In character stuffing and in bit stuffing, the length of the transmitted frames is increased. The worst case redundant frame in case of bit stuffing is one that has a long sequence of all 1s, whereas in the case of character stuffing, it's a frame consisting entirely of DLE characters.
2. When transmission errors occur, the receiver may incorrectly decode one or two frames (e.g., if the errors occur in the markers). However, it'll be able to resynchronize itself with the next correctly received markers.
3. Bit stuffing can be easily implemented in hardware. However, implementing it in software is difficult given the higher overhead of bit manipulations in software.

## Quick Quiz! #

1

Consider that **bit stuffing** is used and the delimiting sequence is 01111110. The following bit pattern is received. Where does the frame start and where does it end?

11100111110010001010100011111010

COMPLETED 0%

1 of 4



In the next lesson, we'll study error detection in the data link layer.

# Principles of The Data Link Layer: Error Detection

This lesson will go into depth with how the data link layer does error detection

## WE'LL COVER THE FOLLOWING

- Error Detection Codes
- Parity Bit
  - Example
- Error Correction Mechanisms
  - Triple Modular Redundancy (TMR)
  - Other Techniques
- Quick Quiz!



## Error Detection Codes #

Besides framing, the data link layer also includes mechanisms to detect and sometimes even recover from transmission errors.

To allow a receiver to detect transmission errors:

1. A sender must add some redundant information (some  $r$  bits) as an **error detection code** to the frame sent. This error detection code is computed by the sender on the frame that it transmits.
2. When the receiver receives a frame with an error detection code, it recomputes it and verifies whether the received error detection code matches the computed error detection code.
3. If they match, the frame is considered to be valid.

Many error detection schemes exist and entire books have been written on the subject. A detailed discussion of these techniques is outside the scope of this course, and we will only discuss some examples to illustrate the key

principles.

To understand error detection codes, let us consider two devices that exchange bit strings containing  $N$  bits. To allow the receiver to detect a transmission error:

1. The sender converts each string of  $N$  bits into a string of  $N + r$  bits.
2. Usually, the  $r$  redundant bits are added at the beginning or the end of the transmitted bit string, but some techniques interleave redundant bits with the original bits.
3. An error detection code can be defined as a function that computes the  $r$  redundant bits corresponding to each string of  $N$  bits.

## Parity Bit #

The simplest error detection code is the parity bit. In this case, the number of redundant bits is 1. There are **two types of parity schemes**:

1. **Even parity:** With the even parity scheme, the redundant bit is chosen so that an even number of bits are set to 1 in the transmitted bit string of  $N + 1$  bits.
2. **Odd parity:** With the odd parity scheme, the redundant bit is chosen so that an odd number of bits are set to 1 in the transmitted bit string of  $N + 1$  bits.

The receiver can easily recompute the parity of each received bit string and discard the strings with an invalid parity. The parity scheme is often used when 7-bit characters are exchanged. In this case, the eighth bit is often a parity bit.

## Example #

The table below shows the parity bits that are computed for bit strings containing three bits.

3 bits string	Odd parity	Even parity
000	1	0

001	0	1
010	0	1
100	0	1
111	0	1
110	1	0
101	1	0
011	1	0

The parity bit allows a receiver to detect transmission errors that have affected a single bit among the transmitted  $N + 1$  bits. If there are an even number of bits in error, the errors wouldn't be detected. An odd number of errors will still be detected.

## Error Correction Mechanisms #

It is also possible to design a code that allows the receiver to **correct transmission errors**.

### Triple Modular Redundancy (TMR) #

The simplest error correction code is the **triple modular redundancy (TMR)**.

- To transmit a bit set to 1, the sender transmits 111 and to transmit a bit set to 0, the sender transmits 000.
- When there are no transmission errors, the receiver can decode 111 as 1.
- If transmission errors have affected a single bit, the **receiver performs majority voting** as shown in the table below. This scheme allows the receiver to correct all transmission errors that affect a single bit.

Received bits	Decoded bit
---------------	-------------

000	0
001	0
010	0
100	0
111	1
110	1
101	1
011	1

## Other Techniques #

Other more powerful error correction codes have been proposed and are used in some applications. **The Hamming Code** is a clever combination of parity bits that provides error detection and correction capabilities.

In practice, data link layer protocols combine bit stuffing or character stuffing with a length indication in the frame header and a checksum. The checksum is computed by the sender and placed in the frame before applying bit/character stuffing.

## Quick Quiz! #

1

What would be the parity bit for the following string in an odd parity scheme?

100101001

COMPLETED 0%

1 of 3



In the next lesson, we'll start with data link layer medium access control!

# Medium Access Control: Static Allocation

In this lesson, we introduce medium access control protocols.

## WE'LL COVER THE FOLLOWING ^

- Introduction
- LAN Organizations
- Collisions
- Medium Access Control Algorithms
- Static allocation Algorithms
  - Frequency Division Multiplexing
  - TDM
  - Dynamic TDM
  - Disadvantages
- Quick Quiz!

## Introduction #

Point-to-point data link layer types need to select one of the framing techniques described previously and optionally add retransmission algorithms, such as those explained for the transport layer to provide a reliable service.

## LAN Organizations #

A LAN is composed of several hosts that are attached to the same shared physical medium. And LAN can be organized in a [few different ways](#). We'll focus on **four main ones**:

1. A **bus-shaped network** where all hosts are attached to the same physical cable.

2. A **ring-shaped network** where all hosts are attached to an upstream and a downstream node so that the entire network forms a ring.
3. A **star-shaped network** where all hosts are attached to the same device.
4. A **wireless network** where all hosts can send and receive frames using radio signals.

## Collisions #

The common problem among all of these network organizations is how to efficiently share access to the local area network. If two devices send a frame at the same time, the two electrical, wireless, or optical signals that correspond to these frames will appear at the same time on the transmission medium, and a receiver will not be able to decode either frame. Such simultaneous transmissions are called **collisions**. A **collision** may involve frames transmitted by two or more devices attached to the Local Area Network. Collisions are the main cause of errors in wired Local Area Networks. They also reduce the throughput of the network, which is problematic.

## Medium Access Control Algorithms #

All Local Area Network technologies hence rely on **Medium Access Control algorithms** to regulate the transmissions **to either minimize or avoid collisions**.

There are **two broad families** of Medium Access Control algorithms: **deterministic and optimistic**.

## Static allocation Algorithms #

**Deterministic or pessimistic MAC algorithms** assume that collisions are a very severe problem and that they must be completely avoided. These algorithms ensure that at any time, **at most one device is allowed to send a frame** on the LAN. This is usually achieved by using either a distributed protocol which elects one device that is allowed to transmit at each time, or a central protocol such that there is a central entity that assigns transmit time to hosts. A deterministic MAC algorithm **ensures that no collision will happen**, but there is some **overhead in regulating the transmission of all the devices** attached to the LAN.

A first solution is to define, *a priori*, the distribution of the transmission resources among the different devices that want to share the resources. These methods are referred to as **Channel Partitioning Protocols**. If  $N$  devices need to share the transmission capacities of a LAN operating at  $b$  Mbps, each device could be allocated a bandwidth of  $\frac{b}{N}$  Mbps (although, note that the practically achievable throughput is less than this limit).

## Frequency Division Multiplexing #

**Frequency Division Multiplexing (FDM)** is a static allocation scheme in which a frequency is allocated to each device attached to the shared medium. As each device uses a different transmission frequency, collisions cannot occur.

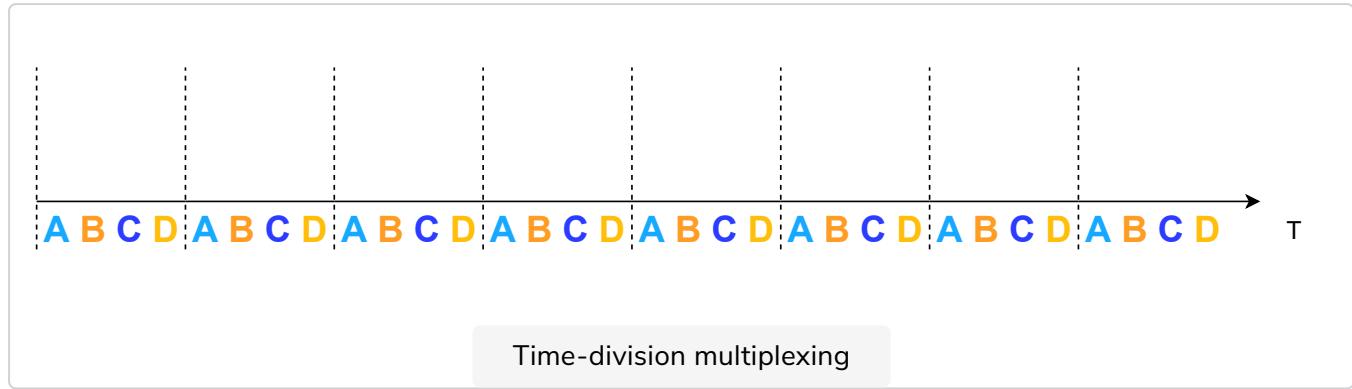
FDM applied to optical fiber is typically referred to as **Wavelength Division Multiplexing (WDM)**. In optical networks, WDM can be used which uses the same principles as FDM but is applied to light. An optical fiber can transport light at different wavelengths without interference. With WDM, a different wavelength is allocated to each of the devices that share the same optical fiber.

## TDM #

**Time Division Multiplexing (TDM)** is a static bandwidth allocation method that was initially defined for the telephone network. In the fixed telephone network, a voice conversation is usually transmitted as a 64 kbps signal. Thus, *a telephone conversation generates 8 kbps, or 1 byte every 125 microseconds*. Telephone conversations often need to be multiplexed together on a single line. For example, in Europe, thirty 64 Kbps voice signals are multiplexed over a single 2 Mbps line. This is done by using Time Division Multiplexing (TDM).

**TDM divides the transmission opportunities into slots.** In the telephone network, a slot corresponds to 125 microseconds. A position inside each slot is reserved for each voice signal. The figure below illustrates TDM on a link that is used to carry four voice conversations. The vertical lines represent the slot boundaries and the letters the different voice conversations. One byte from each voice conversation is sent during each 125 microsecond slot. The byte corresponding to a given conversation is always sent at the same position in

each slot.



## Dynamic TDM #

TDM, as shown above, can be completely static, i.e., the same conversations always share the link. However, if the time slot allocation is static and some users are not currently transmitting, the resources are being wasted.

Dynamic TDM can be used in that scenario, which simply allows for the time slot allocations to be **dynamically adjusted** to make more efficient use of the transmission resources. In order to do so, the two parties part of a session must exchange messages specifying which conversation uses which byte inside each slot. Thanks to these signaling messages, it is possible to dynamically add and remove voice conversations from a given link.

## Disadvantages #

TDM and FDM are widely used in telephone networks to support fixed bandwidth conversations. **Using them in Local Area Networks that support computers would probably be inefficient though**, because computers usually don't send information at a fixed rate. Instead, they often have an on-off behavior. During the on-period, the computer tries to send at the highest possible rate, e.g., to transfer a file. During the off-period, which is often much longer than the on-period, the computer does not transmit any packet. Using a static allocation scheme for computers attached to a LAN would lead to huge inefficiencies, as they would only be able to transmit at  $\frac{1}{N}$  of the total bandwidth during their on-period. This is despite the fact that the other computers are in their off-period and therefore don't need to transmit any information.

## Quick Quiz! #

1

There is a fiber optic channel with a bandwidth of 30 MHz that needs to be equally shared amongst 30 users based on a static allocation scheme. The bandwidth allocated to each user will be \_\_\_\_.

COMPLETED 0%

1 of 2



The dynamic MAC algorithms discussed in the remainder of this chapter aim to solve this problem. In the next lesson, we'll look at some optimistic allocation protocols!

# Medium Access Control: Stochastic Methods - ALOHA

In this lesson, we'll study ALOHANet!

## WE'LL COVER THE FOLLOWING



- Additive Links On-line Hawaii Area (ALOHA)
  - ALOHANet
  - Pseudocode
  - Slotted ALOHA
- Quick Quiz!

**Stochastic or Optimistic MAC algorithms** assume that collisions are part of the normal operation of a Local Area Network. They **aim to minimize the number of collisions**, but they **don't try to avoid all collisions**.

Stochastic algorithms are usually easier to implement than deterministic ones.

## Additive Links On-line Hawaii Area (ALOHA) #

In the 1960s, computer networks consisted of mainframes with a few dozen terminals attached to them using the telephone network or physical cables. The University of Hawaii chose a different organization though. Instead of using telephone lines to connect the terminals to the mainframes, they developed the **first packet radio technology**.

## ALOHANet #

ALOHANet showed that it was **possible to use radio signals to interconnect computers**. The first version of ALOHANet, operated as follows:

- First, the terminals and the mainframe exchanged fixed-length frames composed of 704 bits. Each frame contained 80 8-bit characters, some control bits and parity information to detect transmission errors.

- Two channels in the 400 MHz range were reserved for the operation of ALOHANet.
  1. The first channel was used by the mainframe to send frames to all terminals.
  2. The second channel was shared among all terminals to send frames to the mainframe.
- As all terminals shared the same transmission channel, there was a risk of collision. To deal with this problem as well as transmission errors, the mainframe verified the parity bits of the received frame and sent an acknowledgment on its channel for each correctly received frame. The terminals, on the other hand, had to retransmit the unacknowledged frames.

## Pseudocode #

The pseudo-code below shows the **operation of an ALOHANet terminal**. We use this python syntax for all Medium Access Control algorithms described in this chapter. The algorithm is applied to each new frame that needs to be transmitted. It attempts to transmit a frame at most `max` times (while loop). Each transmission attempt is performed as follows:

1. The frame is sent. Each frame is protected by a timeout.
2. Then, the terminal waits for either a valid acknowledgment frame or the expiration of its timeout.
3. If the terminal receives an acknowledgment, the frame has been delivered correctly and the algorithm terminates. Otherwise, the terminal waits for a random time and attempts to retransmit the frame.

```
# ALOHA
N=1
while N <= max:
    send(frame)
    wait(ack_on_return_channel or timeout)
    if (ack_on_return_channel):
        break # transmission was successful
    else if(timeout):
        # timeout
        wait(random_time)
    N=N+1
```



```
else:  
    # Too many transmission attempts
```

Pseudocode: operation of an ALOHANet terminal

## Slotted ALOHA #

Many improvements to ALOHANet have been proposed, and this technique, or some of its variants, are still found in wireless networks today. **The slotted technique** proposed in Roberts' 1975 paper titled “[ALOHA packet system with and without slots and capture](#)” is important because it **shows that a simple modification can significantly improve channel utilization**.

It works like so:

- Instead of allowing all terminals to transmit at any time, divide the time into slots and allow terminals to transmit only at the beginning of each slot.
- Each slot corresponds to the time required to transmit one fixed size frame.
- In practice, these slots can be imposed by a single clock that is received by all terminals. In ALOHANet, it could have been located on the central mainframe.

The analysis in Roberts's paper reveals that this simple modification **improves the channel utilization by a factor of two**.

## Quick Quiz! #

1

Slotted ALOHA improves channel utilization

---

In the next lesson, we'll look at another stochastic MAC protocol called carrier sense multiple access.

# Medium Access Control: Stochastic Methods - CSMA

In this lesson, we'll study the carrier sense multiple access protocol.

## WE'LL COVER THE FOLLOWING ^

- Carrier Sense Multiple Access (CSMA)
  - Why CSMA?
  - How It Works
  - Pseudocode
  - Non-persistent CSMA
  - Performance of CSMA Variants
- Quick Quiz!

## Carrier Sense Multiple Access (CSMA) #

### Why CSMA? #

- **ALOHA and slotted ALOHA** can easily be implemented, but using them in anything but a lightly loaded network will be extremely inefficient. Designing a network for a very low utilization is possible, but it clearly increases the cost of the network.
- To overcome the problems of ALOHA, many Medium Access Control mechanisms have been proposed which **improve channel utilization**.
- Carrier Sense Multiple Access (CSMA) is one of these and is a significant improvement compared to ALOHA.

### How It Works #

CSMA requires all nodes to listen to the transmission channel to verify that it's free before transmitting a frame. When a node senses the channel to be busy, it defers its transmission until the channel becomes free again.

## Pseudocode #

The pseudocode below provides a more detailed description of the operation of CSMA.

```
# persistent CSMA
N=1
while N <= max:
    wait(channel_becomes_free)
    send(frame)
    wait(ack or timeout)
    if ack:
        break # transmission was successful
    else :
        # timeout
        N=N+1
# end of while loop
# Too many transmission attempts
```

Pseudocode: operation of CSMA terminal

The above pseudocode is often called **persistent CSMA** as the terminal will **continuously listen** to the channel and transmit its frame as soon as the channel becomes free.

## Non-persistent CSMA #

Another important variant of CSMA is the **non-persistent CSMA**. The main difference between persistent and non-persistent CSMA described in the pseudocode below is that a non-persistent CSMA node **does not continuously listen to the channel** to determine when it becomes free. When a non-persistent CSMA terminal **senses the transmission channel to be busy, it waits for a random time before sensing the channel again**. This improves channel utilization compared to persistent CSMA. With persistent CSMA, when two terminals sense the channel to be busy, they will both transmit (and thus cause a collision) as soon as the channel becomes free.

However, the higher channel utilization achieved by non-persistent CSMA comes at the expense of **slightly higher waiting time** in the terminals when the network is lightly loaded.

```
# Non persistent CSMA
N=1
while N <= max:
    listen(channel)
```

```

if free(channel):
    send(frame)
    wait(ack or timeout)

if received(ack):
    break # transmission was successful
else:
    # timeout
    N=N+1
else:
    wait(random_time)
# end of while loop
# Too many transmission attempts

```

## Performance of CSMA Variants #

Kleinrock and Tobagi analyzed the performance of several CSMA variants in detail. Under some assumptions about the transmission channel and the traffic, here's a table of the channel utilization of each protocol we've looked at so far.

Protocol	Channel Utilization
ALOHA	18.4%
Slotted ALOHA	36.6%
Persistent CSMA	52.9%
non-persistent CSMA	81.5%

## Quick Quiz! #

1

What's the difference between persistent and non-persistent CSMA?

COMPLETED 0%

1 of 2



---

In the next lesson, we'll look at an incredibly popular variant of the carrier sense multiple access protocol.

# Medium Access Control: Stochastic Methods - CSMA/CD

In this lesson, we'll look at the variant of CSMA that also detects collisions.

## WE'LL COVER THE FOLLOWING ^

- Carrier Sense Multiple Access with Collision Detection
  - Worst Case
- Quick Quiz!

## Carrier Sense Multiple Access with Collision Detection #

CSMA improves channel utilization compared to ALOHA. However, the performance can still be improved further, especially in wired networks.

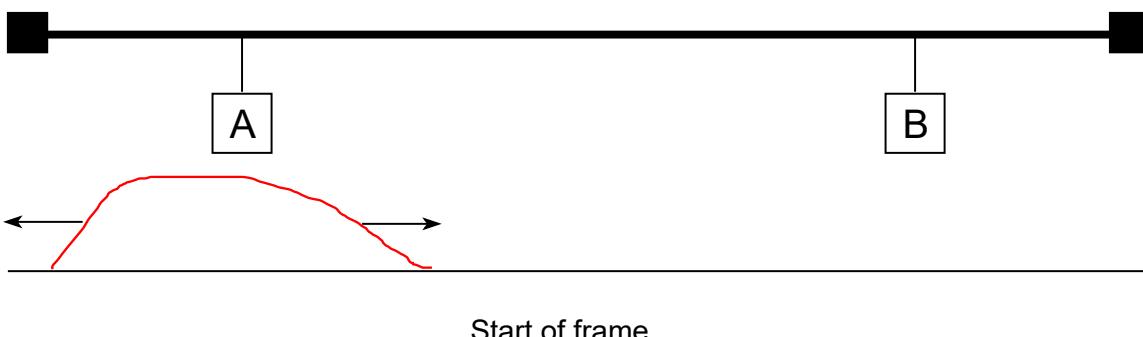
Consider the situation of two terminals that are connected to the same cable. This cable could, for example, be a coaxial cable, or it could also be built with twisted pairs. Before extending CSMA, it's useful to **understand more intuitively, how frames are transmitted in such a network** and how collisions can occur.

The slides below illustrate the physical transmission of a frame on such a cable. To transmit its frame, **host A must send an electrical signal on the shared medium.**

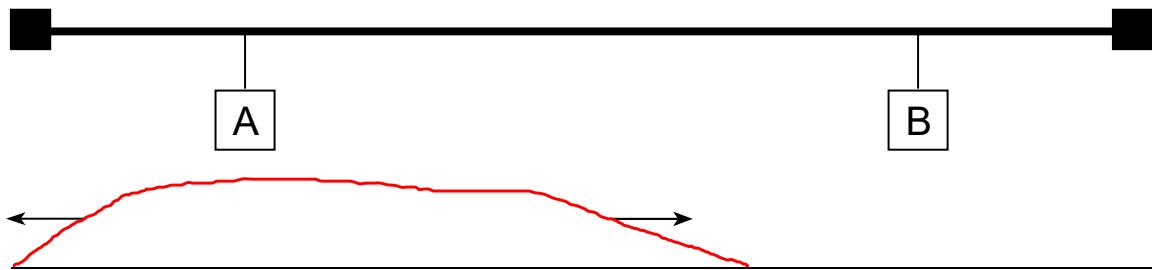
1. The first step is thus to begin the transmission of the electrical signal.  
This is depicted in the first slide below.
2. This electrical signal will travel along the cable. Although electrical signals travel fast, we know that information cannot travel faster than the speed of light (i.e. 300, 000 kilometers/second). On a coaxial cable, an electrical signal is slightly slower than the speed of light in a vacuum

electrical signal is slightly slower than the speed of light in a vacuum which is at about 200,000 kilometers/second.

- This implies that if the cable has a length of one kilometer, the electrical signal will need 5 microseconds to travel from one end of the cable to the other.
3. The ends of coaxial cables are equipped with termination points that ensure that the electrical signal is not reflected back to its source. This is illustrated in the third slide below, where the electrical signal has reached the left endpoint and host B.
4. At this point, B starts to receive the frame being transmitted by A. Notice that there is a delay between the transmission of a bit on host A and its reception by host B. If there were other hosts attached to the cable, they would receive the first bit of the frame at slightly different times. As we will see later, this timing difference is a key problem for MAC algorithms.
5. In slide 4, the electrical signal has reached both ends of the cable and occupies it completely. Host A continues to transmit the electrical signal until the end of the frame.
6. As shown in slide 5, when the sending host stops its transmission, the electrical signal corresponding to the end of the frame leaves the coaxial cable.
7. The channel becomes empty again once the entire electrical signal has been removed from the cable.



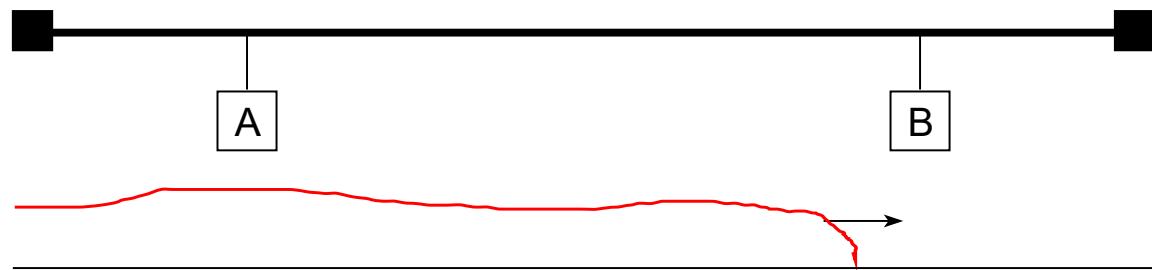
Frame transmission on a shared bus



frame is propagated on LAN (5 microseconds per kilometer)

Frame transmission on a shared bus

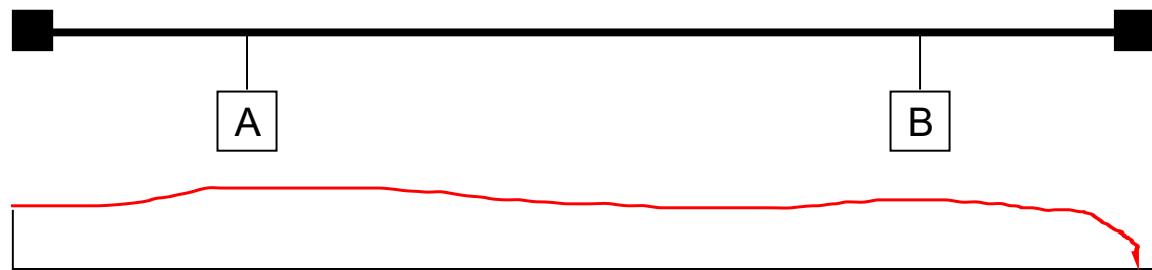
2 of 6



Frame's first bit reaches B

Frame transmission on a shared bus

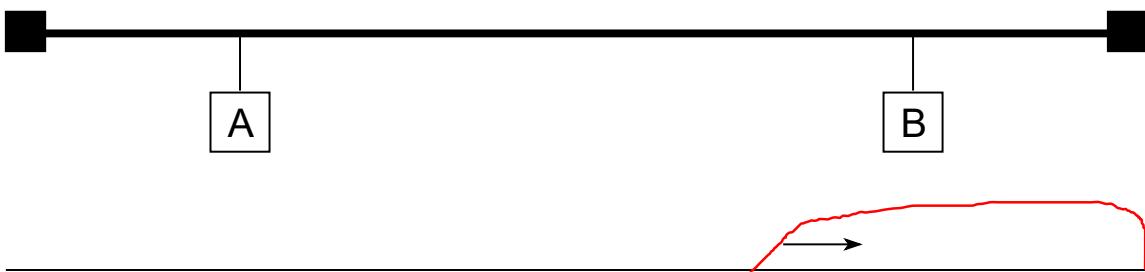
3 of 6



Frame reaches both ends of the cable

Frame transmission on a shared bus

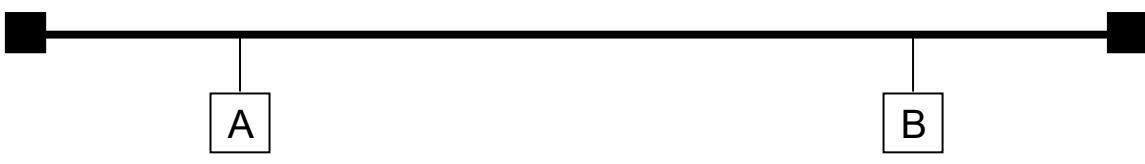
4 of 6



Sender stops transmission

Frame transmission on a shared bus

5 of 6



Channel becomes empty again

Frame transmission on a shared bus

6 of 6

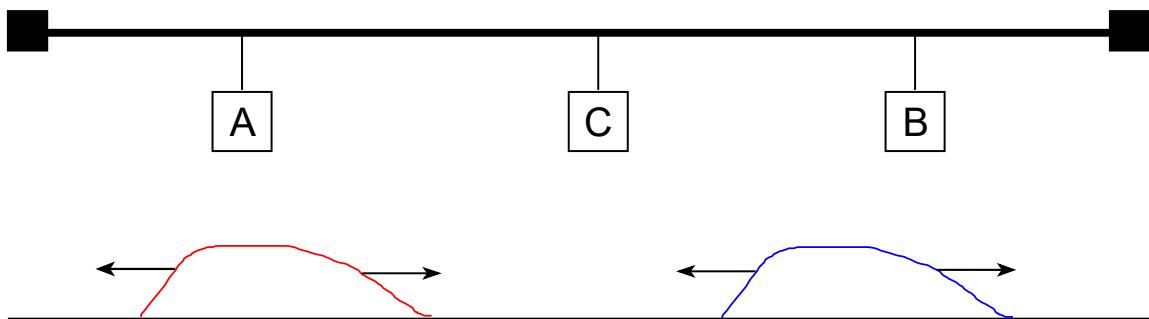


Now that we have looked at how a frame is actually transmitted as an electrical signal on a shared bus, it's interesting to look in more detail at **what happens when two hosts transmit a frame at almost the same time**. This is illustrated in the slides below:

- **Hosts A and B start their transmission at the same time** (first slide).
- At this time, if host C senses the channel, it will consider it to be free.
- This will not last a long time, and in the second slide the electrical signals from both host A and host B reach host C.
- The combined electrical signal (shown graphically as the superposition of the two curves in the figure) cannot be decoded by host C.

- Host C detects a collision, as it receives a signal that it cannot decode. Since on a wire, this could be a voltage level that corresponds to neither 0 nor 1.
- Since host C cannot decode the frames, it cannot determine which hosts are sending the colliding frames. Note that host A (and host B) will detect the collision after host C (third slide).
- In a wired network, a host is able to detect such a collision both while it's listening (e.g., like host C in the figure above) and also while it is sending its own frame. When a host transmits a frame, it can compare the electrical signal that it transmits with the electrical signal that it senses on the wire.
- In the first and second slides in the figure above, host A senses only its own signal.
- In the third slide, it senses an electrical signal that differs from its own signal and can thus detect the collision.
- At this point, its frame is corrupted and it can stop its transmission.

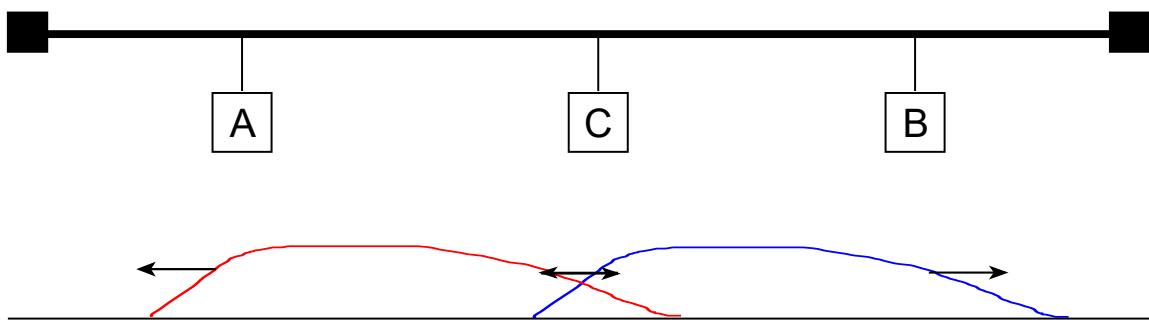
The ability to detect collisions while transmitting is the starting point for the **Carrier Sense Multiple Access with Collision Detection (CSMA/CD)** MAC algorithm which is used in Ethernet networks. When an Ethernet host detects a collision while it's transmitting, it immediately stops its transmission. Compared with pure CSMA, CSMA/CD is an important improvement since when collisions occur, they only last until colliding hosts have detected it and stopped their transmission instead of continuing to transport the rest of the frame unnecessarily. In practice, when a host detects a collision, it sends a special jamming signal on the cable to ensure that all hosts have detected the collision.



Frame starts at A and B almost at the same time

Frame collision on a shared bus

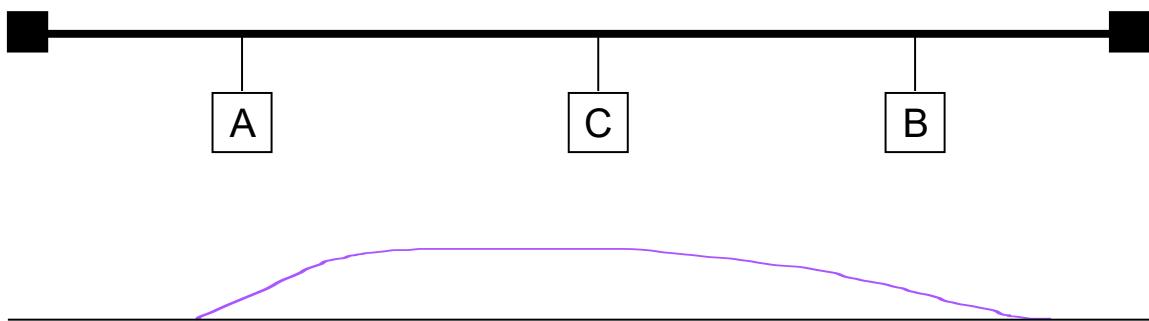
1 of 4



Collision between frames A and B

Frame collision on a shared bus

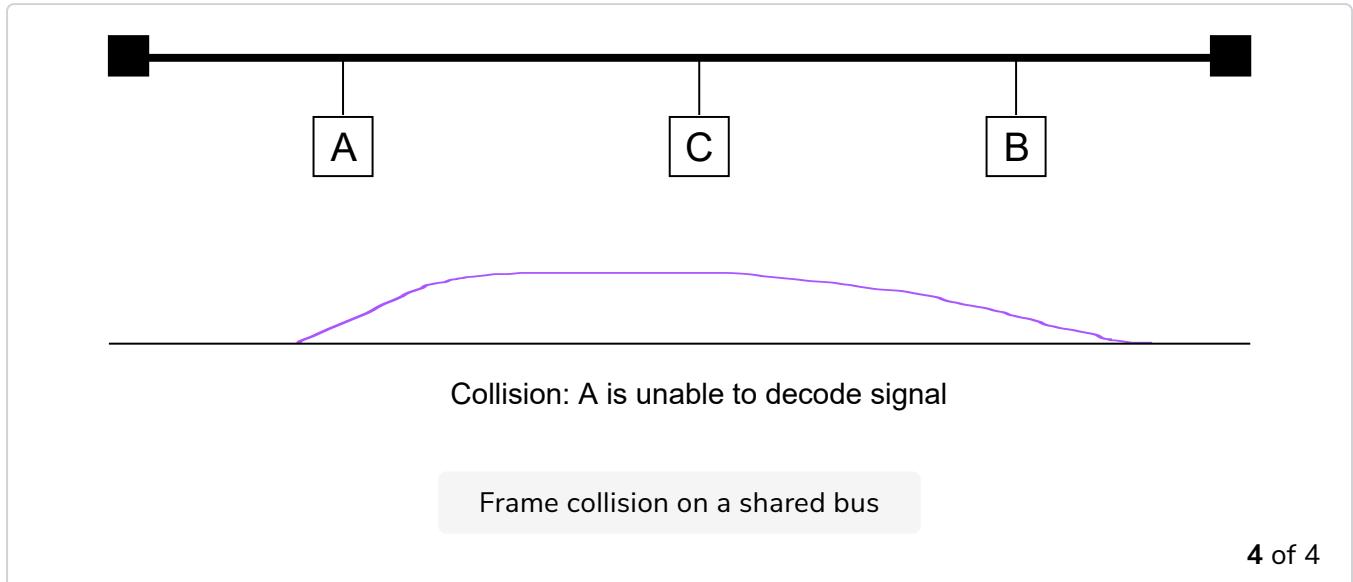
2 of 4



Collision: C is unable to decode signal

Frame collision on a shared bus

3 of 4

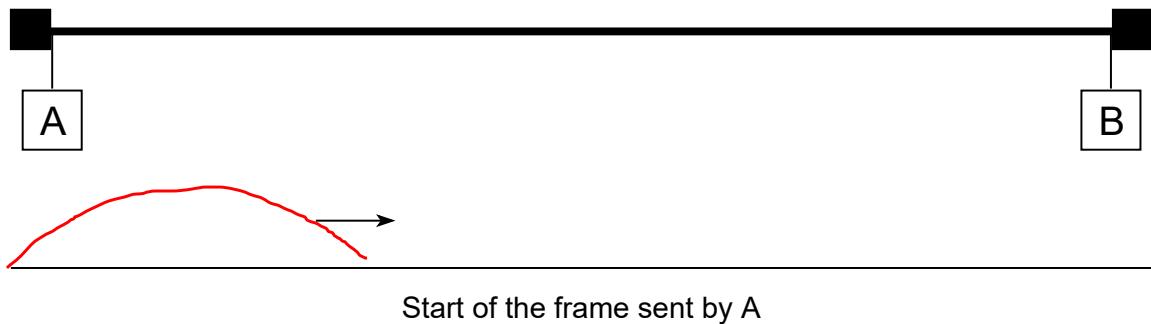


## Worst Case #

To better understand these collisions, it's useful to analyse what would be the worst collision on a shared bus network. Let's consider a wire with two hosts attached at both ends, as shown in the slides below.

- Host A starts to transmit its frame and its electrical signal is propagated on the cable. Its propagation time depends on the physical length of the cable and the speed of the electrical signal. Let us use  $\tau$  to represent this propagation delay in seconds.
- Slightly less than  $\tau$  seconds after the beginning of the transmission of A's frame, B decides to start transmitting its own frame.
- After  $\epsilon$  seconds, B senses A's frame, detects the collision and stops transmitting.
- The beginning of B's frame travels on the cable until it reaches host A.
- Host A can thus detect the collision at time  $\tau - \epsilon + \tau \approx 2 \times \tau$ .
- An important point to note is that a collision can only occur during the first  $2 \times \tau$  seconds of its transmission. If a collision did not occur during this period, it cannot occur afterward since the transmission channel is busy after  $\tau$  seconds and CSMA/CD hosts sense the transmission channel before transmitting their frame.

before transmitting their frame.



Start of the frame sent by A

The worst collision on a shared bus

1 of 3



After  $\tau$  seconds, A's frame reaches B. At time  $\tau-\varepsilon$ , B starts to transmit its own frame. B notices the collision immediately and stops transmitting

The worst collision on a shared bus

2 of 3



A detects collision at time  $\tau+\tau-\varepsilon$

The worst collision on a shared bus

3 of 3



Quick Quiz! #

1

What would happen if a collision occurs and regular CSMA was deployed?

COMPLETED 0%

1 of 3



Now that we have a basic idea of how CSMA/CD works, we'll look at a few ways that it's optimized in the next lesson.

# Medium Access Control: Stochastic Methods - Optimizing CSMA/CD

In this lesson, we'll learn about techniques to optimize CSMA/CD

## WE'LL COVER THE FOLLOWING ^

- Removing Acknowledgements
  - Edge Case: Short Frames
  - Retransmission Timeout
    - Performance
    - Pseudocode
  - Quick Quiz!

## Removing Acknowledgements #

On the wired networks where CSMA/CD is used, collisions are almost the only cause of transmission errors that affect frames. Transmission errors that only affect a few bits inside a frame seldom occur in these wired networks. For this reason, the designers of CSMA/CD chose to **completely remove the acknowledgment frames in the data link layer.**

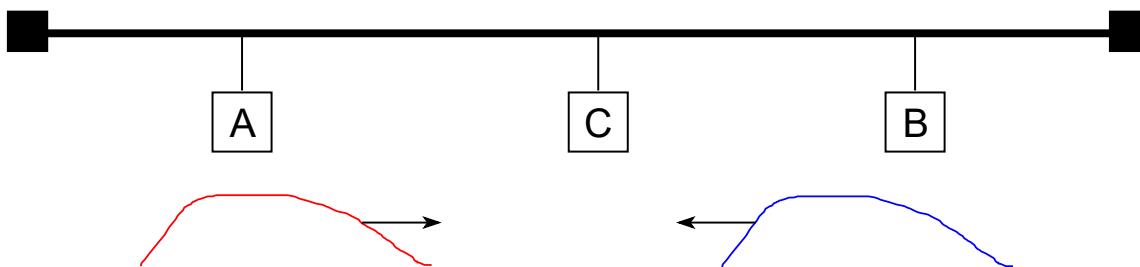
- When a host transmits a frame, it verifies whether its transmission has been affected by a collision.
- If not, given the negligible Bit Error Ratio of the underlying network, it assumes that the frame was received correctly by its destination. So, the bit errors might be detected or corrected using a checksum field. If not, we can rely on the layers above to implement retransmission.
- Otherwise, the frame is retransmitted after some delay.

## Edge Case: Short Frames #

Removing acknowledgments reduces the number of frames that are exchanged on the network and the number of frames that need to be processed by the hosts. However, to use this optimization, **we must ensure that all hosts will be able to detect all the collisions that affect their frames**. The problem is important for short frames.

Let us consider two hosts, A and B, that are sending a small frame to host C as illustrated in the slides below. If the frames sent by A and B are very short, the situation illustrated below may occur.

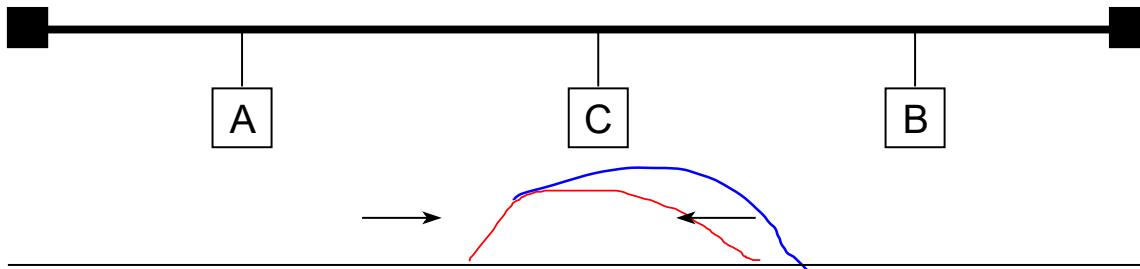
1. Hosts A and B send their frame and stop transmitting (first slide). Since until the end of the transmission, no collision was detected, hosts A and B are content that they were successfully able to use the channel to transmit the frame.
2. When the two short frames arrive at the location of host C, they collide and host C cannot decode them (second slide).
3. The two frames are absorbed by the ends of the wire. **Neither host A nor host B has detected the collision**. They both consider their frame to have been received correctly by its destination.



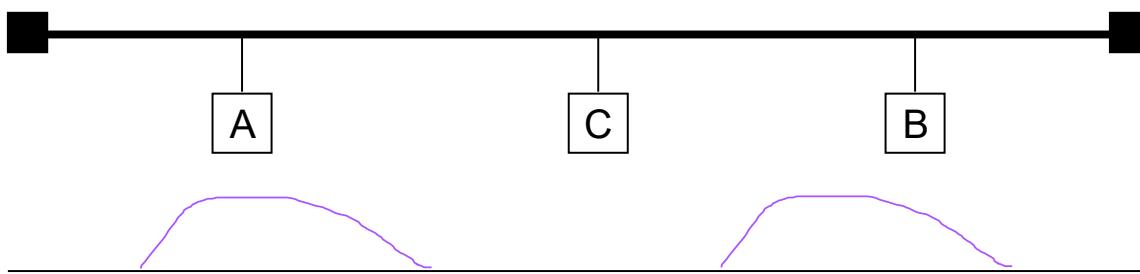
A and B send a small frame almost at the same time

The short-frame collision problem

1 of 3



Two frames collide in the middle of the medium



A and B both did not notice the collision and consider their frames to have been received correctly



To solve this problem, networks using CSMA/CD require hosts to transmit for at least  $2 \times \tau$  seconds. Since the network transmission speed is fixed for a given network technology, this implies that a technology that uses CSMA/CD enforces a minimum frame size. In the most popular CSMA/CD technology, Ethernet,  $2 \times \tau$  is called the **slot time**.

## Retransmission Timeout #

The last innovation introduced by CSMA/CD is the computation of the **retransmission timeout**.

Setting such a timeout is always a compromise between the network access delay and the number of collisions.

- A short timeout would lead to a low network access delay but with a higher risk of collisions.
- On the other hand, a long timeout would cause a long network access delay but a lower risk of collisions.

The **binary exponential back-off** algorithm was introduced in CSMA/CD networks to solve this problem.

To understand binary exponential back-off, let us consider a collision caused

by exactly two hosts.

- Once it has detected the collision, a host can either retransmit its frame immediately or defer its transmission for some time.
- If each colliding host flips a coin to decide whether to retransmit immediately or to defer its retransmission, four cases are possible:
  - Both hosts retransmit immediately and a new collision occurs.
  - The first host retransmits immediately and the second defers its retransmission.
  - The second host retransmits immediately and the first defers its retransmission.
  - Both hosts defer their retransmission and a new collision occurs.
- In the second and third cases, both hosts have flipped different coins. The delay chosen by the host that defers its retransmission should be long enough to ensure that its retransmission will not collide with the immediate retransmission of the other host.
- However, the delay should not be longer than the time necessary to avoid the collision, because if both hosts decide to defer their transmission, the network will be idle during this delay.
- The slot time is the optimal delay since it is the shortest delay that ensures that the first host will be able to retransmit its frame completely without any collision.

## Performance #

- If two hosts are competing, **the algorithm above will avoid a second collision 50% of the time.**
- However, if the network is heavily loaded, several hosts may be competing at the same time. In this case, the hosts should be able to automatically adapt their retransmission delay. The binary exponential back-off performs this adaptation based on the number of collisions that have affected a frame:

After the first collision, the host flips a coin and waits 0 or 1 slot

- After the first collision, the host flips a coin and waits 0 or 1 slot time.
- After the second collision, it generates a random number and waits 0, 1, 2 or 3 slot times, and so on.
- The duration of the waiting time is doubled after each collision.

## Pseudocode #

The complete pseudocode for the CSMA/CD algorithm is shown in the figure below.

```
N=1
while N <= max:
    wait(channel_becomes_free)
    send(frame)
    wait_until (end_of_frame) or (collision)
    if collision detected:
        stop transmitting
        end(jamming)
        k = min (10, N)
        r = random(0, 2k - 1)*slotTime
        wait(r*slotTime)
        N=N+1
    else:
        wait(inter-frame_delay)
        break
# end of while loop
# Too many transmission attempts
```



Pseudocode: CSMA/MD

The **inter-frame delay** used in this pseudocode is a short delay corresponding to the time required by a network adapter to switch from transmitting to receiving mode. It's also used to prevent a host from sending a continuous stream of frames without leaving any transmission opportunities for other hosts on the network. This contributes to the fairness of CSMA/CD.

Despite this delay, there are still conditions where CSMA/CD is not completely fair. Consider for example a network with two hosts: a server sending long frames and a client sending acknowledgments. Measurements reported have shown that there are situations where the client could suffer from repeated collisions that lead it to wait for long periods of time due to the exponential back-off algorithm.

## Quick Quiz! #

1

Why is a minimum frame size necessary?

COMPLETED 0%

1 of 2



Now that we're done with stochastic algorithms, we'll study some key data link layer technologies.

# Introduction to Ethernet

In this lesson, we give a quick introduction to Ethernet.

## WE'LL COVER THE FOLLOWING ^

- Introduction
- First Official Ethernet Specification
  - Important Parameters
  - Changes Recommended by First Official Specification
- MAC Addresses
  - Checking Your MAC Address
- Quick Quiz!

## Introduction #

Ethernet was designed in the 1970s at the Palo Alto Research Center. The first prototype used a coaxial cable as the shared medium and 3 Mbps of bandwidth.

## First Official Ethernet Specification #

Ethernet was improved during the late 1970s and in the 1980s, Digital Equipment, Intel and Xerox published the first official Ethernet specification.

## Important Parameters #

This specification defines several important parameters for Ethernet networks.

1. The first decision was to **standardize the commercial Ethernet at 10 Mbps.**
2. The second decision was the **duration of the slot time**. In Ethernet, a transmission could not last longer than a certain amount of time.

long slot time enables networks to span a long distance but forces the host to use a larger minimum frame size. The compromise was a **slot time of 51.2 microseconds**, which corresponds to a minimum frame size of 64 bytes.

3. The third decision was the **frame format**. The experimental 3 Mbps Ethernet network built at Xerox used short frames containing 8 bit source and destination address fields. Up to 554 bytes of payload using 8 bit addresses was suitable for an experimental network, but it was clearly too small for commercial deployments. Hence, they came up with 48 bit source and destination address fields and up to 1500 bytes of payload.

## Changes Recommended by First Official Specification #

The initial Ethernet specification recommended **three important changes** compared to the networking technologies that were available at that time.

1. The first change was to require each host attached to an Ethernet network to have a globally unique data link layer address. Until then, data link layer addresses were manually configured on each host.
2. The second change introduced by Ethernet was to encode each address as a 48 bit field. 48 bit addresses were huge compared to the networking technologies available in the 1980s, but the huge address space had several advantages, including the ability to allocate large blocks of addresses to manufacturers. Eventually, other LAN technologies opted for 48 bit addresses as well.
3. The third change introduced by Ethernet was the definition of broadcast and multicast addresses. The need for multicast Ethernet was foreseen. Thanks to the size of the addressing space, it was possible to reserve a large block of multicast addresses for each manufacturer.



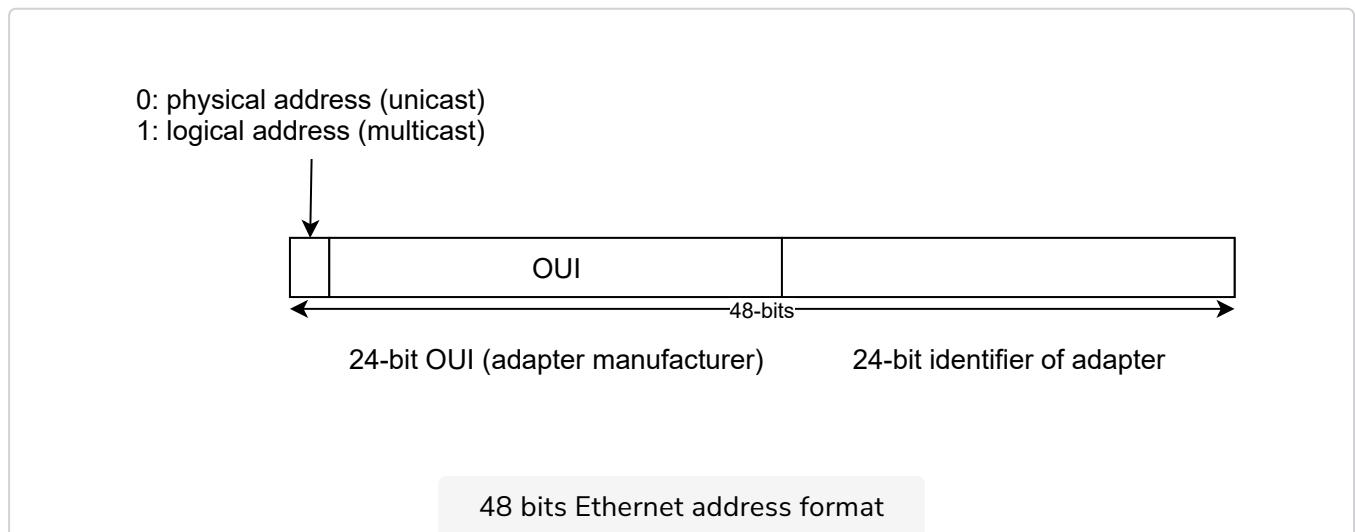
### Note: Unicast, Multicast, & Broadcast:

- **Unicast** messages are sent to **one host** on the network.
- **Multicast** messages are sent to **a group of hosts** on the network
- **Broadcast** messages are sent to **all hosts on the network**.

# MAC Addresses #

The **data link layer addresses used in Ethernet networks are often called MAC addresses**. They are structured as shown in the figure below.

- The first bit of the address indicates whether the address identifies a network adapter or a multicast group.
- The upper 24 bits are used to encode an **Organization Unique Identifier (OUI)**. This OUI identifies a block of addresses that has been allocated by the secretariat who is responsible for the uniqueness of Ethernet addresses to a manufacturer. For instance, **00000C** belongs to **Cisco Systems Inc.**. Once a manufacturer has received an OUI, it can build and sell products with any of the ~16 million addresses in this block. A manufacturer may obtain more than one OUIs.



Mac addresses are generally represented as 6 hex numbers separated by colons.

## Checking Your MAC Address #

You can run the following command on Unix based systems like Linux and Mac OS to get a list of interfaces available on your system.

```
ifconfig
```

Run the command ifconfig to list available interfaces. The interface name ens4 is highlighted in red.

Navigation icons: back, forward, search, etc.

Pick the ethernet interface. Our's is called **ens4**. You can run the following

command to get only your ethernet interface's details.

```
ifconfig ens4
```



The MAC Address is printed after `HWaddr`. So the output may be as follows:

```
ens4      Link encap:Ethernet HWaddr 42:01:0a:80:00:31
          inet addr:10.128.0.49 Bcast:10.128.0.49 Mask:255.255.255.255
          inet6 addr: fe80::4001:aff:fe80:31%1/64 Scope:Link
                     UP BROADCAST RUNNING MULTICAST MTU:1460 Metric:1
                     RX packets:484039 errors:0 dropped:0 overruns:0 frame:2374
                     TX packets:442202 errors:0 dropped:0 overruns:0 carrier:0
                     collisions:0 txqueuelen:1000
                     RX bytes:2637134894 (2.6 GB) TX bytes:73969976 (73.9 MB)
```

So, the MAC Address is `42:01:0a:80:00:31`.

Next, we can use a tool called `macchanger` to change our MAC address.

```
ifconfig ens4
ifconfig ens4 down # Turn the interface off
macchanger -r ens4 # Change Mac Address
ifconfig eth0 up   # Turn it back on
ifconfig ens4      # Check new MAC address
```



Try changing your MAC Address!

## Quick Quiz! #

1

Which of the following is NOT a valid unicast MAC address?

COMPLETED 0%

1 of 3



Now that we have a basic understanding of ethernet, let's look at what an ethernet frame looks like in the next lesson!

# Ethernet Frame Format

In this lesson, we'll study the ethernet frame format.

## WE'LL COVER THE FOLLOWING ^

- Ethernet Frames
  - Problem: Sending Short Frames
  - Solution: Add Length Field
- Quick Quiz!

## Ethernet Frames #

The original 10 Mbps Ethernet specification defined a simple frame format where each frame is composed of five fields.

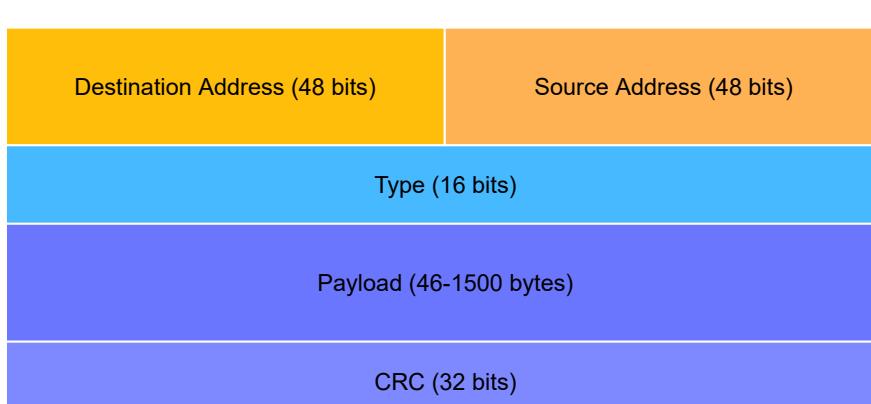
0. The Ethernet frame starts with a **preamble** (not shown in the figure below) that's used by the physical layer of the receiver to synchronise its clock with the sender's clock.
1. The first field of the frame is the **destination address**. As this address is placed at the beginning of the frame, an Ethernet interface can quickly verify whether it's the frame recipient and if not, cancel the processing of the arriving frame.
2. The second field is the **source address**. While the destination address can be either a unicast or a multicast/broadcast address, the source address must always be a unicast address.
3. The third field is a 16 bit integer that indicates which type of network layer packet is carried inside the frame. This field is often called the **Ether Type**. Frequently used EtherType values include: 0x0800 for IPv4, 0x86DD for IPv6, and 0x806 for the Address Resolution Protocol (ARP).
4. The fourth part of the Ethernet frame is the payload. This is the data being transferred between hosts.

4. The fourth part of the Ethernet frame is the **payload**. The minimum length of the payload is 46 bytes to ensure a minimum frame size,

including the header of 64 bytes. The Ethernet payload cannot be longer than 1500 bytes. This size was found reasonable when the first Ethernet specification was written. 1500 bytes was large enough without forcing the network adapters to contain overly large memories.

5. The last field of the Ethernet frame is a 32 bit **Cyclical Redundancy Check (CRC)**. This CRC is able to catch a much larger number of transmission errors than the Internet checksum used by IP, UDP and TCP.

The format of the Ethernet frame is shown below:



The Ethernet frame format shown above is highlighted in DIX, i.e., the first ethernet specification. This is the format used to send both IPv4 and IPv6 packets.

## Problem: Sending Short Frames #

After the publication of DIX, the Institute of Electrical and Electronic Engineers (IEEE) began to standardize several Local Area Network technologies starting with Ethernet.

- While developing its Ethernet standard, the IEEE 802.3 working group was confronted with a problem: Ethernet mandated a minimum payload size of 46 bytes, while some companies were looking for a LAN technology that could transparently transport short frames containing only a few bytes of payload.
- Such a frame can be sent by an Ethernet host by padding it to ensure that

the payload is at least 46 bytes long. However since the original Ethernet

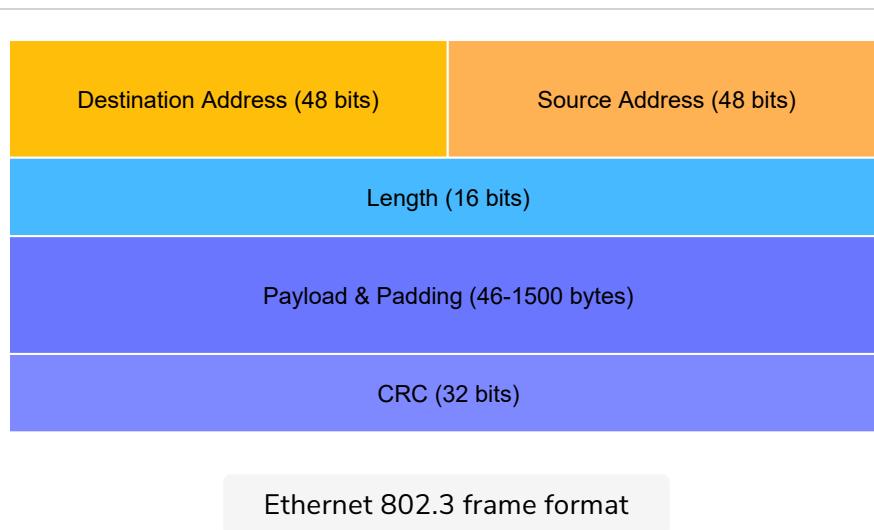
header does not contain a length field, it's impossible for the receiver to determine how many useful bytes were placed inside the payload field.

## Solution: Add Length Field #

To solve this problem, the IEEE decided to **replace the Type field with a length field**. This **Length field contains the number of useful bytes in the frame payload**. The payload must still contain at least 46 bytes, but padding bytes are added by the sender and removed by the receiver.

Without the type field, however, it's impossible for a receiving host to identify the type of network layer packet inside a received frame. To solve this new problem, IEEE developed a completely new sublayer called the **Logical Link Control**. Several protocols were defined in this sublayer. One of them provided a slightly different version of the Type field of the original Ethernet frame format. Another contained acknowledgments and retransmissions to provide a reliable service.

The figure below shows the official 802.3 frame format.



## Quick Quiz! #

1

What would be the total size of a minimal sized TCP packet encapsulated inside an IP packet, encapsulated inside an Ethernet frame?

COMPLETED 0%

1 of 3



---

In the next lesson, we'll look at physical layers that have been defined for Ethernet networks.

# Physical Layers for Ethernet

In this lesson, we'll look at various types of physical layers and their limitations and benefits to Ethernet.

## WE'LL COVER THE FOLLOWING



- 10Base5
- 10Base2
- 10BaseF
- 10BaseT
  - Twisted Pairs
  - Changes to Ethernet
    - Change in Topology
    - Introduction of Ethernet Hubs
- Fast Ethernet
- Quick Quiz!

Several physical layers have been defined for Ethernet networks.

## 10Base5 #

The first type of physical layer, usually called **10Base5**, provided 10 Mbps over a thick coaxial cable. The characteristics of the cable and transceivers that were used then enabled the utilization of 500 meter long segments. A 10Base5 network can also include repeaters between segments.

## 10Base2 #

The second type of physical layer was **10Base2**. 10Base2 used a thin coaxial cable that was easier to install than the 10Base5 cable but could not be longer than 185 meters.

## 10BaseF #

A 10BaseF type of physical layer was also defined to transport Ethernet over point-to-point optical links.

## 10BaseT #

### Twisted Pairs #

The major change to the physical layer was the support of twisted pairs in the 10BaseT specification. Twisted pair cables are traditionally used to support the telephone service in office buildings. Most office buildings today are equipped with structured cabling. Several twisted pair cables are installed between any room and a central telecom closet per building or per floor in large buildings. These telecom closets act not only as concentration points for the telephone service but also for LANs.

### Changes to Ethernet #

The introduction of the twisted pairs led to two major changes to Ethernet. Let's discuss each.

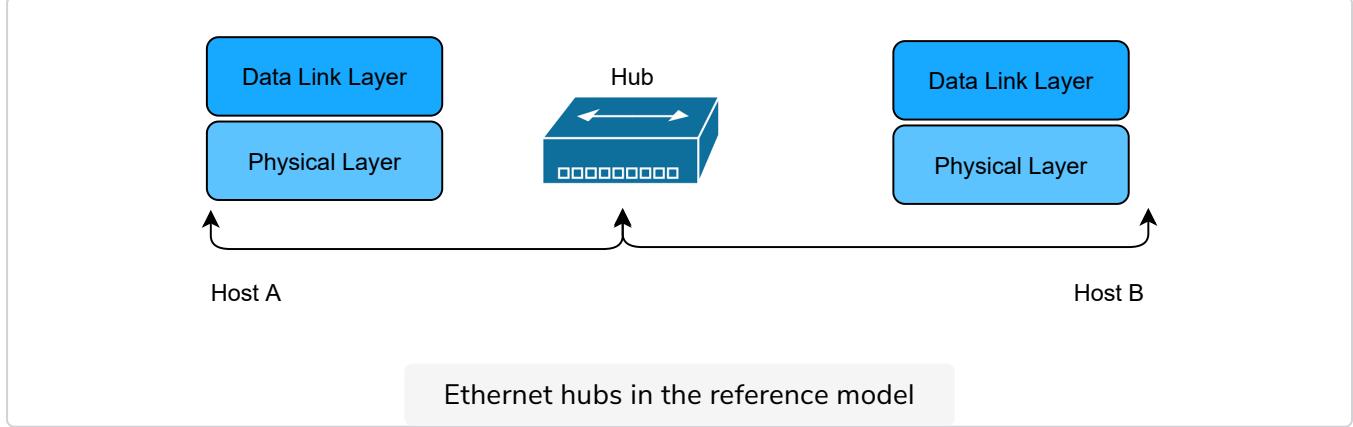
#### Change in Topology #

The first change concerns the physical topology of the network. 10Base2 and 10Base5 networks are shared buses, the coaxial cable typically passes through each room that contains a connected computer. A 10BaseT network, however, is a **star-shaped network**. All the devices connected to the network are attached to a twisted pair cable that ends in the telecom closet. From a maintenance perspective, this is a major improvement. The cable is a weak point in 10Base2 and 10Base5 networks. Any physical damage on the cable broke the entire network and when such a failure occurred, the network administrator had to manually check the entire cable to detect where it was damaged. With 10BaseT, when one twisted pair is damaged, only the device connected to this twisted pair is affected and this does not affect the other devices.

#### Introduction of Ethernet Hubs #

The second major change introduced by 10BaseT was that it was impossible to build a 10BaseT network by simply connecting all the twisted pairs together. All devices in the network must be connected to a central hub or switch.

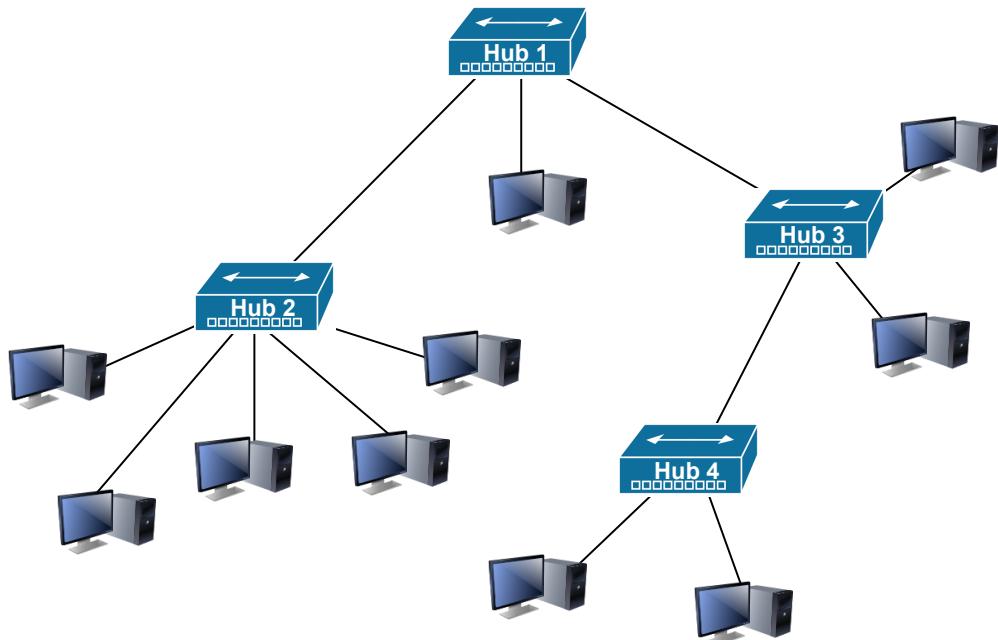
All the twisted pairs must be connected to a relay that operates in the physical layer. This relay is called an **Ethernet hub**. A hub is thus a physical layer relay that receives an electrical signal on one of its interfaces, regenerates the signal and transmits it over all its other interfaces. Some hubs are also able to convert the electrical signal from one physical layer to another, such as a 10BaseT to 10Base2 conversion.



Computers can directly be attached to Ethernet hubs. Ethernet hubs themselves can be attached to other Ethernet hubs to build a larger network.

However, some important guidelines must be followed when building a complex network with hubs.

1. First, **the network topology must be a tree**. As hubs are relays in the physical layer, adding a link between Hub 2 and Hub 3 in the network below would create an electrical shortcut that would completely disrupt the network. This implies that there cannot be any redundancy in a hub-based network. A failure of a hub or a link between two hubs would partition the network into two isolated networks.
2. Second, as hubs are relays in the physical layer, collisions can happen and must be handled by CSMA/CD as in a 10Base5 network. This implies that the maximum delay between any pair of devices in the network can't be longer than the 51.2 microseconds, slot time. If the delay is longer, collisions between short frames may not be correctly detected. This **constraint limits the geographical spread of 10BaseT networks containing hubs**.



A hierarchical Ethernet network composed of hubs

## Fast Ethernet #

In the late 1980s, 10 Mbps became too slow for some applications and network manufacturers developed several LAN technologies that offered higher bandwidth, such as the 100 Mbps FDDI LAN that used optical fibers.

As the development of 10Base5, 10Base2 and 10BaseT had shown that Ethernet could be adapted to different types of physical layers, several manufacturers started to work on 100 Mbps Ethernet and convinced IEEE to standardize this new technology that was initially called **Fast Ethernet**. Fast Ethernet was designed under two constraints:

1. First, Fast Ethernet had to support twisted pairs. Although it was easier from a physical layer perspective to support higher bandwidth on coaxial cables than on twisted pairs, coaxial cables were a nightmare from deployment and maintenance perspectives.
2. Second, Fast Ethernet had to be perfectly compatible with the existing 10 Mbps Ethernets to allow Fast Ethernet technology to be used initially as a backbone technology to interconnect 10 Mbps Ethernet networks. This forced fast Ethernet to use exactly the same frame format as 10 Mbps Ethernet. This implied that the minimum Fast Ethernet frame size remained at 512 bits. To preserve CSMA/CD with this minimum frame

remained at 512 bits. To preserve CSMA/CD with this minimum frame size and 100 Mbps instead of 10 Mbps, the duration of the slot time was

decreased to 5.12 microseconds, which implies that the maximum distance between two end stations was also reduced.

The evolution of Ethernet did not stop. In 1998, the IEEE published the first standard to provide Gigabit Ethernet over optical fibers. Several other types of physical layers were added afterward. The 10 Gigabit Ethernet standard appeared in 2002.

Work is ongoing to create faster ethernet standards. The table below lists the main Ethernet standards. A more detailed list may be found [here](#).

Standard	Comments
10Base2	Thick coaxial cable, 500m
10Base5	Thin coaxial cable, 185m
10BaseT	Two pairs of category 3+ UTP
10Base-F	10 Mb/s over optical fiber
100Base-Tx	Category 5 UTP or STP, 100 m maximum
1000Base-CX	Two multimode optical fiber, 2 km maximum
100Base-FX	Two multimode or single mode optical fibers with lasers
1000Base-SX	Two pairs shielded twisted pair, 25m maximum
40-100 Gbps	Optical fiber but also Category 6 UTP

# Quick Quiz! #

1

Which of the following is an advantage of 10BaseT?

COMPLETED 0%

1 of 3



In the next lesson, we'll study Ethernet switches.

# Ethernet Switches

In this lesson, we'll gain a brief introduction to Ethernet switches.

## WE'LL COVER THE FOLLOWING



- Introduction
  - MAC Address Tables
  - Retaining Plug & Play with Switches
    - MAC address learning algorithm
- Pseudocode
  - Timestamp
- Unicast, Broadcast, & Multicast Frames
- Handling Failures
  - Spanning Tree Protocol
- Quick Quiz!

## Introduction #

Increasing the physical layer bandwidth as in Fast Ethernet was only one of the solutions to improve the performance of Ethernet LANs.

A second solution was to replace the hubs with more intelligent devices. As Ethernet hubs operate in the physical layer, they can only regenerate the electrical signal to extend the geographical reach of the network. From a performance perspective, it would be more interesting to have **devices that operate in the data link layer** and can analyze the destination address of each frame and forward the frames selectively on the link that leads to the destination. This would allow two hosts to communicate on one pair of interfaces while other pairs of interfaces can be simultaneously used for other communication, thereby improving communication efficiency. Such devices are usually called **Ethernet switches**. An **Ethernet switch is a relay that**

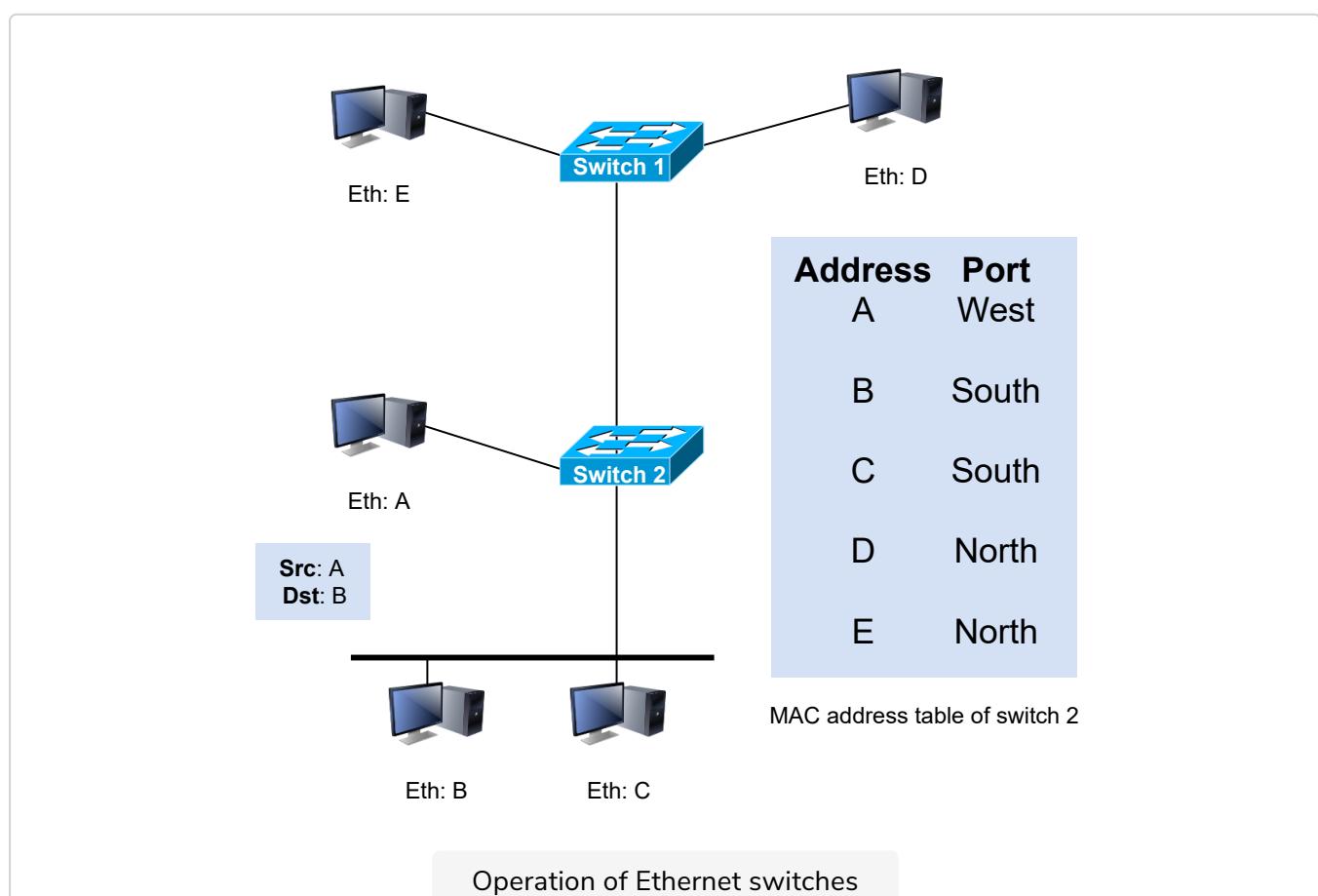
operates in the data link layer.

## MAC Address Tables #

An Ethernet switch understands the format of the Ethernet frames and can *selectively* forward frames over each interface. For this, each Ethernet switch maintains a **MAC address table**. This table contains, for each MAC address known by the switch, the identifier of the switch's port over which a frame sent towards this address must be forwarded to reach its destination.

This is illustrated below with the MAC address table of **switch 2**.

- When the switch receives a frame destined to address B, it forwards the frame on its South port.
- If it receives a frame destined to address D, it forwards it only on its North port.



## Retaining Plug & Play with Switches #

One of the selling points of Ethernet networks is that thanks to the utilization of 48 bit MAC addresses, an Ethernet LAN is plug and play at the data link layer. When two hosts are attached to the same Ethernet segment or hub, they can immediately exchange Ethernet frames without requiring any configuration.

can immediately start learning destinations without requiring any configuration.

## MAC address learning algorithm #

It is important to retain this plug and play capability for Ethernet switches as well. This implies that **Ethernet switches must be able to build their MAC address table automatically without requiring any manual configuration**. This automatic configuration is performed by the **MAC address learning algorithm** that runs on each Ethernet switch.

1. This algorithm extracts the source address of the received frames and remembers the port over which a frame from each source Ethernet address has been received.
2. This information is inserted into the MAC address table that the switch uses to forward frames.
3. This allows the switch to automatically learn the ports that it can use to reach each destination address, provided that this host has previously sent at least one frame. This is not a problem since most upper-layer protocols use acknowledgments at some layer and thus even an Ethernet printer sends Ethernet frames as well.

## Pseudocode #

The pseudocode below details how an Ethernet switch forwards Ethernet frames. It first updates its MAC address table with the source address of the frame.

## Timestamp #

The MAC address table used by some switches also contains a timestamp that is updated each time a frame is received from each known source address. This timestamp is used to remove from the MAC address table entries that have not been active during the last  $n$  minutes. This limits the growth of the MAC address table, but also allows hosts to move from one port to another.

```
# Arrival of frame F on port P
# Table : MAC address table dictionary : addr->port
# Ports : list of all ports on the switch
src=F.SourceAddress
dst=F.DestinationAddress
Table[src]=P # src heard on port P
if isUnicast(dst):
    if dst in Table:
```



```

if dst in Table:
    ForwardFrame(F,Table[dst])
else:
    for o in Ports:
        if o!= P: ForwardFrame(F,o)
else:
    # broadcast destination
    for o in Ports:
        if o!=P: ForwardFrame(F,o)

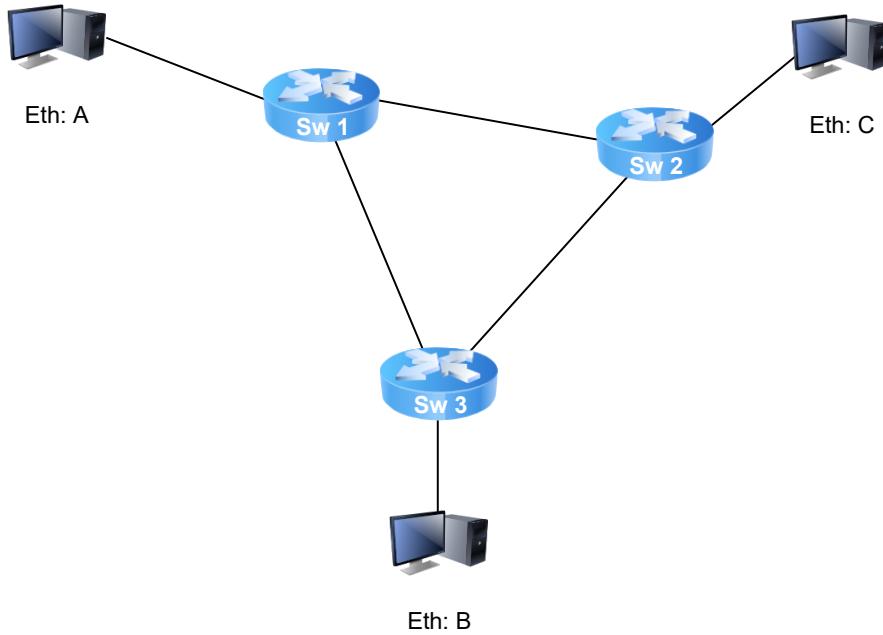
```

## Unicast, Broadcast, & Multicast Frames #

The switch uses its MAC address table to forward the received unicast frame. If there is an entry for the frame's destination address in the MAC address table, the frame is forwarded selectively on the port listed in this entry. Otherwise, the switch does not know how to reach the destination address and it must forward the frame on all its ports except the port from which the frame has been received. This is called **flooding** and it ensures that the frame will reach its destination, at the expense of some unnecessary transmissions. These unnecessary transmissions will only last until the destination has sent its first frame. Multicast and Broadcast frames are also forwarded in a similar way.

## Handling Failures #

The MAC address learning algorithm combined with the forwarding algorithm work well in a tree-shaped network such as the one shown above. However, to deal with link and switch failures, network administrators often add redundant links to ensure that their network remains connected even after a failure. Let us consider what happens in the Ethernet network shown in the figure below.



Ethernet switches in a loop

When all switches boot, their MAC address table is empty. Assume that host A sends a frame towards host C.

- Upon reception of this frame, switch 1 updates its MAC address table to remember that address A is reachable via its West port.
- As there is no entry for address C in switch1's MAC address table, the frame is forwarded to both switch 2 and switch 3.
- When switch 2 receives the frame, it updates its MAC address table for address A and forwards the frame to host C as well as to switch 3. switch 3 has thus received two copies of the same frame.
- As switch 3 does not know how to reach the destination address, it forwards the frame received from switch1 to switch 2 and the frame received from switch2 to switch1 and so on...
- The single frame sent by host A will be continuously duplicated by the switches until their MAC address table contains an entry for address C.
- Quickly, all the available link bandwidth will be exhausted to forward all the copies of this frame.

As Ethernet does not contain any TTL or Hop Limit, this loop will never

- As Ethernet does not contain any TTL or Hop Limit, this loop will never stop.

The MAC address learning algorithm allows switches to be plug-and-play. Unfortunately, the loops that arise when the network topology is not a tree are a severe problem. Forcing the switches to only be used in tree-shaped networks as hubs would be a severe limitation.

## Spanning Tree Protocol #

To solve this problem, the inventors of Ethernet switches have developed the **Spanning Tree Protocol**. The spanning tree protocol enables switches to exchange control messages by means of which a root switch is first elected in the topology. Then, all switches designate one of their ports that reaches the root switch with the minimum number of hops to be part of the spanning tree. All other ports that lead to the root switch, and hence create a loop are disabled as far as frame forwarding is concerned. Eventually, the frames are only forwarded on ports that are part of the spanning tree. The ports that are not part of the spanning tree continue to send and receive control frames. This helps to recover from switch or link failures.

## Quick Quiz! #

1

Consider the first network given in this lesson. Suppose host D joins it later than the rest and immediately sends a frame to host C. Will host A then be able to send it a frame right after?

COMPLETED 0%

1 of 2



In the next lesson, we'll study virtual LANs!



# Programming Challenge: Spanning Tree Protocol

WE'LL COVER THE FOLLOWING



- Problem Statement
- Spanning Tree Protocol
  - Configuration Bridge Protocol Data Units
  - Comparing Bridge Protocol Data Units
  - How It Works
- Coding Challenge

## Problem Statement #

In this challenge, you will implement a simplified version of the spanning tree protocol. For this challenge, you should assume that a switch and a bridge are the same things. The only practical difference is that bridges have few ports, whereas switches have many many ports.

## Spanning Tree Protocol #

We saw a gist of this protocol in the last lesson. Let's build upon that.

## Configuration Bridge Protocol Data Units #

The spanning tree is created by the exchange of messages between the bridges. These messages are called **Configuration Bridge Protocol Data Units**, or **Configuration BPDUs**. We'll refer to them as **BPDUs**. The format of a BPDU for this challenge is a list of integers as follows: [Root ID, Cost, Transmitting Bridge ID, Transmitting Port ID]

The following table explains what each field of a BPDU means.

Field	Explanation of field
Root ID	ID of the bridge currently believed/known to be root
Transmitting Bridge ID	ID of the bridge transmitting this BPDU
Cost	Cost of the least cost path from the transmitting bridge to the currently known root
Transmitting Port ID	ID of the port out of which BPDU is transmitted

## Comparing Bridge Protocol Data Units #

Some BPDUs are better than others. Here's how to determine which is better.

- A BPDU with a **lower root bridge ID** is **better** than a BPDU with a higher root bridge ID.
- If the root bridge ID of both BPDUs is the same, then a BPDU with a **lower cost** is **better** than a BPDU with a higher cost.
- If the cost and bridge ID of both BPDUs is the same, then a BPDU with a **lower transmitting bridge ID** is **better** than a BPDU with a higher transmitting bridge ID.
- If the cost, the bridge ID, transmitting bridge ID of both BPDUs is the same, then a BPDU with a **lower transmitting port ID** is **better** than a BPDU with a higher transmitting port ID.

## How It Works #

As described above, the protocol aims to build a tree in which one bridge is determined to be the root. Bridges can only forward data frames **towards** the root bridge or away from the root bridge. In this way, cycles are avoided.

Each bridge is assigned a unique bridge ID and the root bridge is the one with the smallest bridge ID.

- When a bridge first boots up, it believes itself to be the root, and hence its BPDU looks like [its bridge ID, 0, its bridge ID, Port ID].
- It multicasts this BPDU to all of its neighbors. The neighbors of this bridge are all the bridges on the LAN segment to which it is connected via any of its ports.
- It also receives BPDUs from all of its neighbors.
- It then processes these BPDUs to determine a couple of things:
  - The root bridge. It declares a port to be root if the root bridge is accessible from this port.
  - Which ports it should declare blocking. It declares a port blocking if it receives a better BPDU than the one it would have sent on that port. The ports are forwarding by default.
- Finally, the bridge sends all of this newly learned information to all bridges accessible via its forwarding or root ports.

## Coding Challenge #

We've given you some starter code. Some helper methods have been created that you might find useful and others are declared but left empty. The main task is to fill in the functions `send_BPDUs()` and `receive_BPDUs()`. Good luck!

main.py topology_reader.py ports.py simulator.py <b>bridge.py</b>	
---	--

```
from ports import ports

def is_better(BPDU1, BPDU2):
    # If root is greater than BPDU1 is better
    if(BPDU1[0] < BPDU2[0]):
        return 1
    elif(BPDU1[0] == BPDU2[0] and BPDU1[1] < BPDU2[1]):
        return 1
    elif(BPDU1[0] == BPDU2[0] and BPDU1[1] == BPDU2[1] and BPDU1[2] < BPDU2[2]):
        return 1
```

```

        elif(BPDU1[0] == BPDU2[0] and BPDU1[1] == BPDU2[1] and BPDU1[2] == BPDU2[2] and BPDU1[3] <
        return 1
    else:
        return 0

class bridge:
    def __init__(self, bridge_ID, port_list):
        self.bridge_ID = bridge_ID
        self.port_list = port_list # port_list[0] is the port with port number 0
        self.config_BPDU = [bridge_ID, 0, bridge_ID, None] # Root ID, Cost, Transmitting Bridge
        self.receive_queue = {}

    def initialize_recv_queue(self, bridges_dict):
        for b in bridges_dict:
            self.receive_queue[b] = []

    def set_bridge(self, bridge_ID, num_ports, mac_addresses):
        self.bridge_ID = bridge_ID
        self.num_ports = num_ports
        self.port_list = port_list

    def get_port(self, bridge_number, bridges_dict):
        for i in range(len(self.port_list)):
            if(bridge_number in self.port_list[i].get_reachable_bridge_ID(bridges_dict, self.bridge_ID)):
                return i

    def find_best_BPDUs_received(self, bridges_dict):
        # Select best BPDU at each bridge
        best_BPDUs_recvd = {} # Bridge Number : BPDU
        # Write your code here
        return best_BPDUs_recvd

    def update_ports(self, bridges_dict, best_BPDUs_recvd):
        # Write your code here
        pass

    def elect_root(self, bridges_dict, best_BPDUs_recvd):
        # Write your code here
        pass

    def send_BPDUs(self, bridges_dict):
        # Write your code here
        return(bridges_dict)

    def receive_BPDUs(self, bridges_dict):
        # These functions are here to give you some structure
        # Feel free to ignore them and implement your own
        # Find best BPDU received from each bridge
        best_BPDUs_recvd = self.find_best_BPDUs_received(bridges_dict)

        # Compare BPDUs with those received at non-root ports
        self.update_ports(bridges_dict, best_BPDUs_recvd)

        # Find root bridge
        self.elect_root(bridges_dict, best_BPDUs_recvd)

        # Update dictionary and return
        bridges_dict[self.bridge_ID] = self
        return bridges_dict

    def get_root_port_id(self):
        for p in range(len(self.port_list)):
```

```
        if self.port_list[p].port_type == 2:  
            return p  
        return None  
  
def get_config_BPDU_at_port(self, port_number):  
    BPDU_at_port = self.config_BPDU  
    BPDU_at_port[3] = self.port_list[port_number].mac_address  
    return(BPDU_at_port)  
  
def print_bridge(self):  
    print("~~~~~Bridge ID: " + str(self.bridge_ID) + " Root ID: " + str(self.config_BPDU[0]))  
    print("BPDU:")  
    print(self.config_BPDU)  
  
    print("MAC address | Port Type | Segment Number")  
    for port in self.port_list:  
        port.print_port()
```



---

Coming up next, we'll look at the solution to the spanning tree protocol programming challenge!

# Solution Review: Spanning Tree Protocol

In this lesson, we'll look at a solution to the spanning tree protocol programming assignment.

## WE'LL COVER THE FOLLOWING ^

- Solution
- Explanation
  - `send_BPDUs()`
  - `receive_BPDUs()`

## Solution #

main.py



topology\_reader.py

ports.py

simulator.py

bridge.py

```
from ports import ports

def is_better(BPDU1, BPDU2):
    # If root is greater than BPDU1 is better
    if(BPDU1[0] < BPDU2[0]):
        return 1
    elif(BPDU1[0] == BPDU2[0] and BPDU1[1] < BPDU2[1]):
        return 1
    elif(BPDU1[0] == BPDU2[0] and BPDU1[1] == BPDU2[1] and BPDU1[2] < BPDU2[2]):
        return 1
    elif(BPDU1[0] == BPDU2[0] and BPDU1[1] == BPDU2[1] and BPDU1[2] == BPDU2[2] and BPDU1[3] <
         return 1
    else:
        return 0

class bridge:
    def __init__(self, bridge_ID, port_list):
        self.bridge_ID = bridge_ID
        self.port_list = port_list # port_list[0] is the port with port number 0
```

```

self.config_BPDU[0] = bridge_ID # port number
self.config_BPDU[1] = 0 # Cost
self.config_BPDU[2] = bridge_ID # Transmitting Bridge
self.config_BPDU[3] = None # Root ID

self.receive_queue = {}

def initialize_recv_queue(self, bridges_dict):
    for b in bridges_dict:
        self.receive_queue[b] = []

def set_bridge(self, bridge_ID, num_ports, mac_addresses):
    self.bridge_ID = bridge_ID
    self.num_ports = num_ports
    self.port_list = port_list

def get_port(self, bridge_number, bridges_dict):
    for i in range(len(self.port_list)):
        if(bridge_number in self.port_list[i].get_reachable_bridge_ID(bridges_dict, self.bridge_ID)):
            return i

def find_best_BPDUs_received(self, bridges_dict):
    # Select best BPDU at each bridge
    best_BPDUs_rcvd = {} # Bridge Number : BPDU
    best_bpdu = 0
    for sending_brg_id in self.receive_queue:
        if(len(self.receive_queue[sending_brg_id]) > 0):
            best_bpdu = self.receive_queue[sending_brg_id][0]
            for bpdu in self.receive_queue[sending_brg_id]:
                if is_better(bpdu, best_bpdu):
                    best_bpdu = bpdu
            best_BPDUs_rcvd[sending_brg_id] = best_bpdu
    self.initialize_recv_queue(bridges_dict)
    return best_BPDUs_rcvd

def update_ports(self, bridges_dict, best_BPDUs_rcvd):
    for sending_brg_id in best_BPDUs_rcvd:
        if sending_brg_id is not self.config_BPDU[0]:
            if self.port_list[self.get_port(sending_brg_id, bridges_dict)].port_type != 2:
                pn = self.get_port(sending_brg_id, bridges_dict)
                if(is_better(best_BPDUs_rcvd[sending_brg_id], self.get_config_BPDU_at_port(pn))):
                    self.port_list[self.get_port(sending_brg_id, bridges_dict)].port_type = 0
                else:
                    self.port_list[self.get_port(sending_brg_id, bridges_dict)].port_type = 1

def elect_root(self, bridges_dict, best_BPDUs_rcvd):
    for sending_brg_id in best_BPDUs_rcvd:
        if(best_BPDUs_rcvd[sending_brg_id][0] < self.config_BPDU[0]):
            # New root bridge
            self.config_BPDU[0] = best_BPDUs_rcvd[sending_brg_id][0]
            self.config_BPDU[1] = best_BPDUs_rcvd[sending_brg_id][1] + 1
            self.config_BPDU[2] = self.bridge_ID
            self.config_BPDU[3] = self.get_port(sending_brg_id, bridges_dict)
            if(self.get_root_port_id() is not None):
                self.port_list[self.get_root_port_id()].port_type = 1
            self.port_list[self.get_port(sending_brg_id, bridges_dict)].port_type = 2

def send_BPDUs(self, bridges_dict):
    # Find all neighbors
    for p in self.port_list:
        if(p.get_port_state() > 0): # If sending port
            # Send to all bridges on that segment
            for b in p.get_reachable_bridge_ID(bridges_dict, self.bridge_ID):
                pn = self.get_port(b, bridges_dict)
                bridges_dict[b].receive_queue[self.bridge_ID].append(self.get_config_BPDU_at_port(pn))
    return(bridges_dict)

```

```

def receive_BPDUs(self, bridges_dict):
    # Update the bridge's state according to the best BPDUs received
    # Find best BPDU received from each bridge
    best_BPDUs_recv = self.find_best_BPDUs_received(bridges_dict)

    # Compare BPDUs with those received at non-root ports
    self.update_ports(bridges_dict, best_BPDUs_recv)

    # Find root bridge
    self.elect_root(bridges_dict, best_BPDUs_recv)

    # Update dictionary and return
    bridges_dict[self.bridge_ID] = self
    return bridges_dict

def get_root_port_id(self):
    for p in range(len(self.port_list)):
        if self.port_list[p].port_type == 2:
            return p
    return None

def get_config_BPDU_at_port(self, port_number):
    BPDU_at_port = self.config_BPDU
    BPDU_at_port[3] = self.port_list[port_number].mac_address
    return(BPDU_at_port)

def print_bridge(self):
    print("~~~~~Bridge ID: " + str(self.bridge_ID) + " Root ID: " + str(self.config_BPDU[0]))
    print("BPDU:")
    print(self.config_BPDU)

    print("MAC address | Port Type | Segment Number")
    for port in self.port_list:
        port.print_port()

```



## Explanation #

### `send_BPDUs()` #

This function called on every bridge in the topology in a round-robin fashion. This function takes the bridges dictionary as input, makes some updates, and returns it.

It first iterates over all of **non-blocking** ports to find its neighbors. The neighbors are found by calling the function `get_reachable_bridge_ID()` on each port. This function returns a list of bridge IDs that are reachable from that port. Each of those bridges is sent the bridge's current BPDU by appending them to that bridge's `receive_queue`.

## `receive_BPDUs()` #

This function called on every bridge in the topology in a round-robin fashion right after the `send_BPDUs()` function is called. It consists of a number of function calls. Let's discuss each.

1. `find_best_BPDUs_received(bridges_dict)` : this function returns the best BPDUs received on each port. Since each port of a bridge could have received multiple BPDUs, the best ones out of those need to be retrieved first. A dictionary where the key is the bridge number and the received BPDU is the value is generated and returned.
2. `update_ports()` : this function iterates over the best BPDUs and updates the ports to be blocking if the one received is better than the one it would have sent and to forwarding otherwise. It uses the helper function `is_better()` to do this. `is_better()` is fairly straightforward: it takes two BPDUs as input and returns 1 if the first one is better and 0 if it's not.
3. `elect_root` : lastly, the root is elected. The best BPDUs are iterated over and if any one of them's root ID is smaller than the one that the bridge currently believes to be the root, it's updated to the new values and the port from which this BPDU was received is set to be the new root. Furthermore, if an old root port existed, it is changed to `forwarding`.

Lastly, the dictionary of bridges is updated with `self` and returned.

---

In the next lesson, we'll study virtual LANs!

# Networks Career Paths

In this lesson, we'll look at some possible career paths that you can pursue after you finish this course!

- **Network technicians** are staff that deploy cabling, equipment, troubleshoot networks.
- **Network administrators** supervise network operations and maintenance.
- Various vendor-specific certifications are popular for demonstrating mastery of networking concepts. For instance, **CISCO** has a few [certifications](#) which have credibility in the networking world.
- For **software developers**, knowing how networks operate is very useful.
- **Network specialists** set up and maintain networks and networking systems such as local area networks and wide area networks. They also install routers, switches, firewalls, and other networks-related software programs.

## Final Thoughts

Thanks for reading!

Thank you for taking this course! We hope that you have gained an in-depth and working knowledge of all things networks.

Feel free to drop us an email or to comment on the community forum about any suggestions or comments.

— Team Educative