

# Internet Technology

A summary? [Rush Release]

## Table of Contents

|                                                                    |    |
|--------------------------------------------------------------------|----|
| Preamble.....                                                      | 2  |
| W1: Intro.....                                                     | 3  |
| Internet.....                                                      | 3  |
| The edge.....                                                      | 4  |
| The core.....                                                      | 4  |
| Packets.....                                                       | 5  |
| Circuit switching.....                                             | 5  |
| Store and Forward model.....                                       | 6  |
| W2: Multimedia and Applications.....                               | 8  |
| What protocol should I use for my app?.....                        | 8  |
| Client-Server or Peer to Peer?.....                                | 9  |
| A simple network app recipe.....                                   | 9  |
| URL's.....                                                         | 11 |
| HTTP.....                                                          | 11 |
| HTTP how it works.....                                             | 12 |
| Cookies.....                                                       | 12 |
| W3: Naming and addressing.....                                     | 14 |
| Ipv4 and ports.....                                                | 14 |
| DHCP.....                                                          | 15 |
| Link Layer.....                                                    | 16 |
| ARP (Address Resolution Protocol).....                             | 17 |
| DNS.....                                                           | 17 |
| W4: Wireless communication.....                                    | 19 |
| Broadcasting.....                                                  | 19 |
| Wireless.....                                                      | 19 |
| WiFi: Connecting and Transmitting.....                             | 21 |
| WiFi: Addressing.....                                              | 22 |
| W5: "Cybersecurity".....                                           | 23 |
| DDoS basics.....                                                   | 23 |
| Where do I get some of them compromised systems?.....              | 23 |
| Amplification and Reflection.....                                  | 24 |
| DDoS attack types.....                                             | 25 |
| Mitigation.....                                                    | 25 |
| W6: IOT Guest lecture.....                                         | 26 |
| IDK, It was basically making fun of IOT and more abt DDoSsing..... | 26 |
| TLDR.....                                                          | 26 |

## Preamble

This summary is definitely better than what is out there now. But I am a bit sleep deprived and I am writing this summary on my 3 hour train commute to uni, so some things might not be mathing.

When it comes to my IntTech experience, I passively listened to the lectures since it looked like something I saw already.

I got all my network engineering experience mostly from youtube, setting up Wifi on linux (The memes are true when it comes to that) and hacking into the family router to lift my internet curfew (terminally online behaviour)

But anyway, take this with a grain of salt, since I might not stick to the material 100%. That is because there is some context that might people understand the abstract material more.

Enjoy this wreck!

# W1: Intro

## Internet

The internet is a network of billions of devices that **in which any device can communicate with any other device** (if not impeded in any way like a firewall or something)

All the devices use the internet **for networked applications** (application that use the network, think of browsers etc.)

They communicate **using protocols**, which are a common format and procedures on how to communicate with another device. (Think of how you ask a stranger for time)

There are two types of devices on the internet:

- **Edge** devices/hosts/servers, like computers, phones and servers. They are like leaves on a big internet tree.
- **Core** devices, like routers, switches, access points and cell towers. They are the branches which hold everything together.

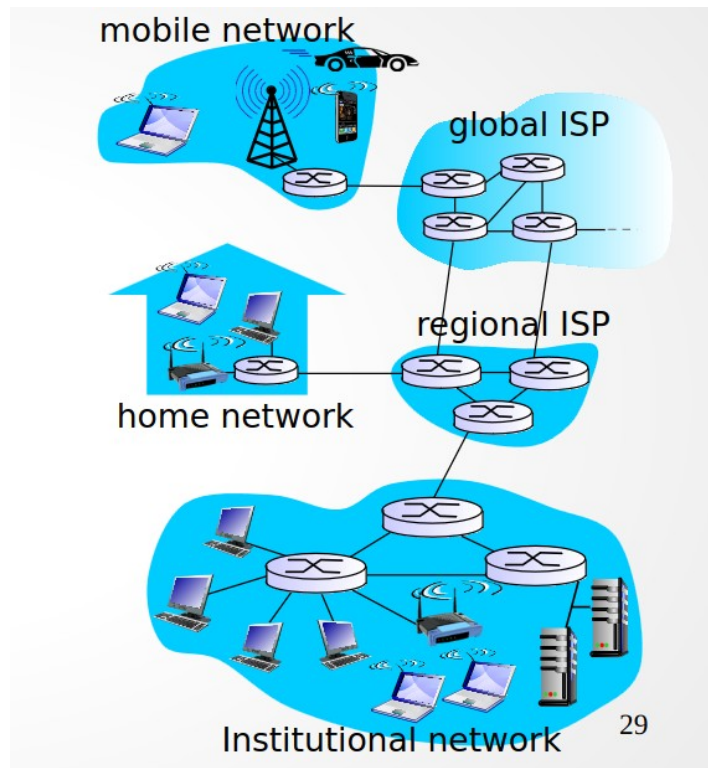
Those devices are linked using physical links with a certain bandwidth, here are some examples:

- **Wires** like Phone wire, Ethernet cable,
- **Light:** Fibre-optic cables
- **Radio(Wireless):** 4G, Wifi, Bluetooth, Starlink ,Zigbee, Lora and more

The internet is divided into networks that are managed by Internet Service Providers (ISP's) so most of the time when you want to communicate with a device, this is the path the communication takes:

Starting device → Wifi router → Modem → {A bunch of ISP infrastructure} → {Global ISP's infrastructure} → {A bunch of targets ISP's infrastructure} → Target's modem → Targets router → Target

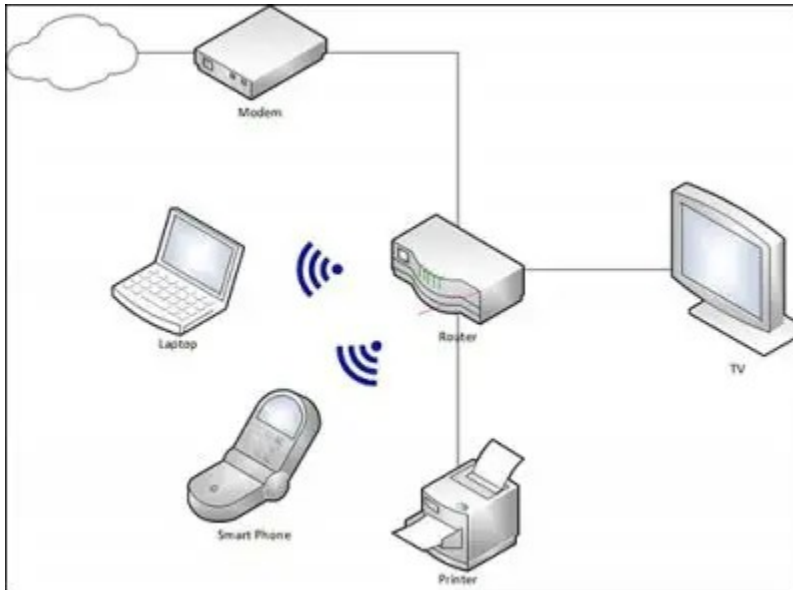
With all the arrows being physical links that have their own bandwidth.



## The edge

The edge is comprised of smaller **access networks** like home networks and networks in institutions, that serve the purpose of connecting the hosts and servers to the bigger core networks. They are more local and do not have the comical amount of bandwidth core networks have.

### Home network example:



## The core

The core networks are here to connect access networks to the rest of the internet. They are the railways, highways and airlines of the internet. They are owned by ISP's which collect all of their client access networks and connect them to other ISP's. This network of ISP's is then what comprises the whole internet. Basically it is a "Mesh of interconnected routers" that passes stuff from one edge network to any other network.

The core has two important tasks to accomplish its goals:

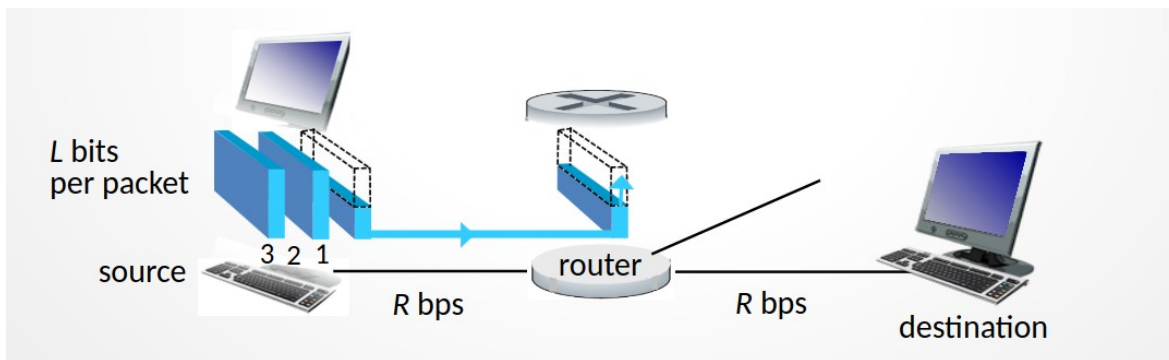
- **Routing:** Create a path from one point on the edge to another using smart algorithms and stuff
- **Forwarding:** Get the input of the router to the appropriate output, in order to make the packet travel on the path specified in the last step.

## Packets

.Just like you probably don't want a completely assembled closet dumped on you front door, Google drive upload servers do not want to have a 1TB zip file of "Homework material" thrown at them in one go either. This is why **communication on the internet is done using packets**, which are neatly **divided and labled parts of the message** that get sent separately through the network, just like a Ikea closet would come in a bunch of boxes and a manual on how to assemle the closet. This is called packet switchih.

The technique used to pass those packets through the network to allow multiple hosts on a network is called **Store and Forward**. Packets sent this way are **pushed in their entirety through the link at full link capacity** from one device to the other along the route to finally reach the destination.

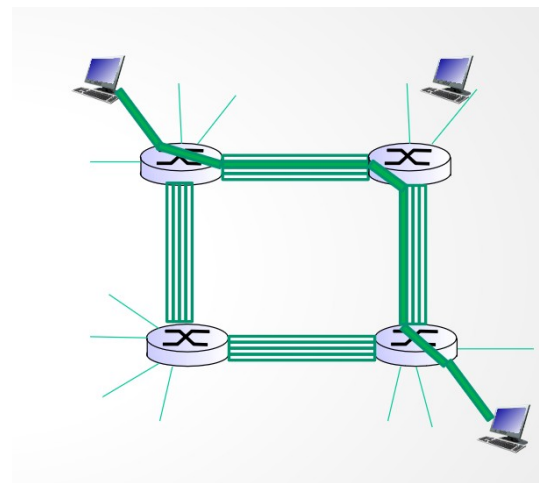
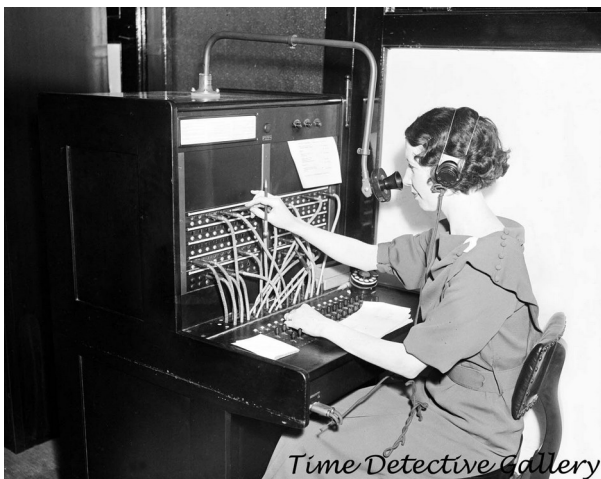
The packet needs to be received in its entirety before it can be sent further.



## Circuit switching

Instead of switching using packets, you could also have multiple dedicated circuits per link that are then connected together between circuit switches to create a route. This ensures maximum performance per built link, but also limits the amount of devices to the amount of links. This approach also means that you loose out on performance when there is less active hosts than circuits since you cannot share bandwidth across those links.

This is the way old telephone lines worked.



## Store and Forward model

A packet going through a router using store and forward model can be compared to a person shopping around the mall, where the routers are stores:

| Router                                                             | Store                                                                               | Name               |
|--------------------------------------------------------------------|-------------------------------------------------------------------------------------|--------------------|
| Kinds of delay                                                     |                                                                                     |                    |
| Processing incoming packet                                         | Putting the products on the conveyor                                                | Processing delay   |
| Waiting in the packet queue                                        | Waiting for the clients in front of you                                             | Queueing delay     |
| Packet being transmitted                                           | Having the cashier scan your products                                               | Transmission delay |
| Traveling across the link to the next router or to the target host | Walking to the next store or going home                                             | Propagation delay  |
| Things that can go wrong                                           |                                                                                     |                    |
| Queue is full and the packet is dropped.                           | The queue in the supermarket reaches all the way to the entrance 🧟, so you give up. | Packet loss        |

What happens under the hood for each of these things, and what are the formulas 🤖?

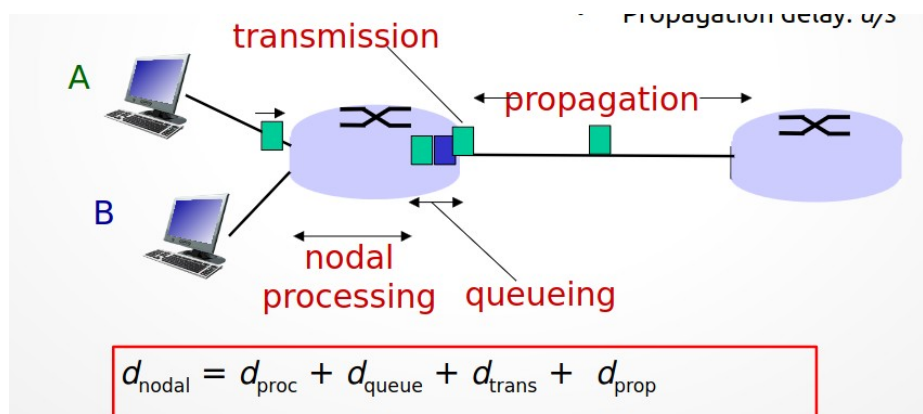
**The processing delay** is where the router checks if there are bit errors (AKA if the packet did not get corrupted) and determines where to send the packet next. It has no formula and is mostly negligible.

During the **Queueing delay** there isn't much happening for the packet besides waiting. There is no formula specified in the slides.

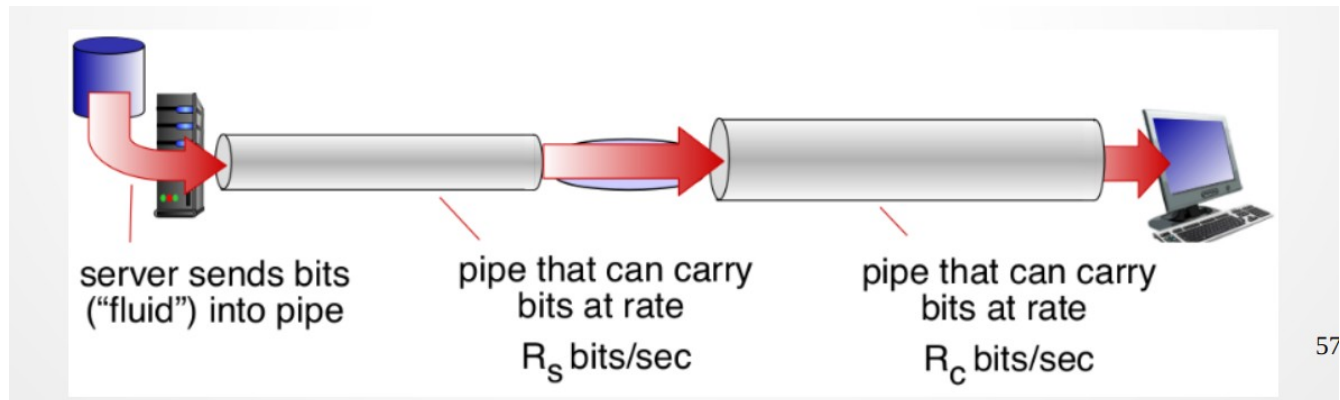
The **transmission delay** is when the bits of the packet are transmitted across the link. The formula is:  $L/R$  where  $L$  is the size of the packet in bits and  $R$  is the bandwidth in bits/second.

**Propagation delay** is the time it takes to travel across the link. The formula is  $d/s$  where  $d$  is the length of the link in meters and  $s$  is the propagation speed. The propagation speed is often a constant from nature, like the speed of light or the speed of sound.

A lil picture to visualize this shiz



The transfer speed on the route in this model is limited by the slowest link. This means there is a possibility of a bottleneck. In this image the throughput of the first pipe is less of the next one limiting the second pipe from using full capacity.



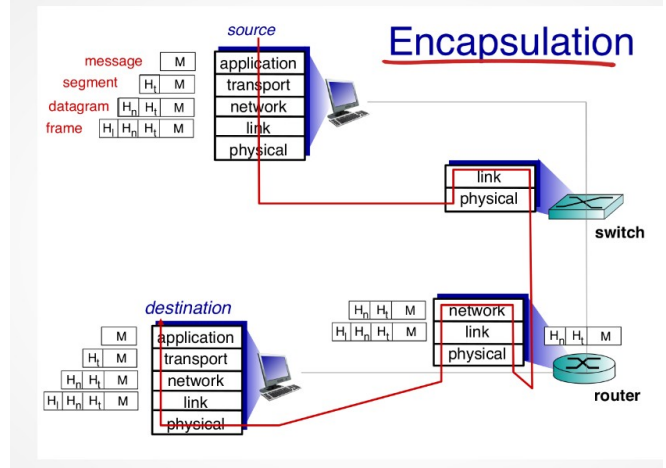
When measuring the amount of data that crosses from the source to the destination (with routers in the middle) we speak of **throughput**, which is the bits per second on received at the destination.

- Instantaneous: Throughput right now
- Average: Throughput over a given period of time.

As said before, the packet also is labeled in some way. This is done using a **layered architecture** that also abstracts some of the complexity of the whole network away. The labeling has **multiple layers** to Each **layer does not care what happens at a previous layer for the most part.**

| L# | Layer name        | Explanation                                  | Examples                       |
|----|-------------------|----------------------------------------------|--------------------------------|
| 5  | Application layer | The networked application                    | HTTP (Browsers), SMTP, (Email) |
| 4  | Transport layer   | Communication protocol between apps          | UDP, TCP                       |
| 3  | Networking layer  | Routing between the source and destination   | Ipv4, Ipv6                     |
| 2  | Data-Link layer   | Communication between two neighbouring nodes | Ethernet, Wifi, Bluetooth      |
| 1  | Physical layer    | Putting the data over the link               | Radio, Wires, Fibre            |

The data from each layer is stacked on top of the message in form of headers:



Disambiguation of terms and data unit content:

| L# | Layer       | How to call a unit of data here? | What does it contain                    |
|----|-------------|----------------------------------|-----------------------------------------|
| 5  | Application | Data                             | Not specified (Body + Headers for HTTP) |
| 4  | Transport   | Segment                          | Data + Headers                          |
| 3  | Network     | Packet                           | Segment + Headers                       |
| 2  | Data-Link   | Frame                            | Packet + Headers                        |
| 1  | Physical    | Bit/Symbol                       | Part of the Frame                       |

## W2: Multimedia and Applications

*Because im in a state of delirium while traveling between Enschede and the Hague, I decided to have a little bit of fun in this chapter.*

### What protocol should I use for my app?

So you decided that you want to make a new shiny network app. What are the steps? What do I need? Let's start with choosing the Data-Link layer protocol, since that is the only thing from the stacked architecture we need to care about.

There are two big protocols that are on the test, **TCP and UDP**.

You also have QUIC(experimental TCP killer) and UNIX sockets (Connections between two processes on the same host) among others, but they are out of scope of the test.

Let's start with explaining **TCP**.

TCP stands for **Transfer Control Protocol**, which is a mid name. What makes TCP unique though are the following things.

- **Reliability:**
  - It makes sure that **all sent data is received**. It **resends** anything that is lost.
  - It makes sure that the **data is received in order**.
- **Congestion control:** It throttles itself when the network is congested.
- **Flow control:** The sender won't overwhelm the reciever with messages.
- **Connection oriented:** It needs setup.

This makes **TCP good for things that need to be sent 100% reliably and in correct order**, like text, code, files, etc.

But for some things like **video and audio, you don't care that much about lost packets** and care a lot more about the **speed**, since both video and audio are quite sizable and require really short delays when streaming. This is where **UDP** comes in. And yes, **User Datagram Protocol** is another non-descriptive and mid name. Let's see **what makes UDP fit for the job:**



- **Does not have the same overhead TCP** has since it **just yeets the packets**.
- It does not care **if the packets arrive and in what order**.
- It does **not maintain a connection** so it is stateless

This makes **UDP way faster than TCP** as long as you don't mind some data not having a smooth landing or just ending up MIA.

You should have some understanding about how they work under the hood from the Observation Labs.

But here are some videos that explains them:

<https://youtu.be/uwoD5YsGACg> UDP vs TCP

[Mebe some more vids here later ]

## Client-Server or Peer to Peer?

The hardest part is organizing how the hosts connect to eachother, since that is where the training wheels of the layered architecture fall off, you are on your own at this point. Luckily this is not that hard. There are only two types of architectures at this point. Client-Server and Peer to Peer.

The first one is the Client-Server architecture.

Client-Server is the simplest architecture to maintain, since there in one side that listens for connections and one side that connects.

It has a little problem though, this architecture has a **bottleneck** and a **single point of failiure, the server**. Everyone has to connect to the server, so when the server goes down or experiences too much traffic, everyone feels the effects of the disruption.

Another way is the Peer to Peer architecture. Most known from things like bittorrent, it is a more distributed method. It **does not involve a server**, so every host using the app is a "peer"(as in equal to other hosts). This makes a **peer both a server and a client** in some sense.

This **eliminates the bottleneck** of the C-S architecture, and makes the system **self-scaling**, but also **takes out the simplicity**. Now every peer on the network both has to connect to other peers and also listen for incoming connections.

Additionally such a complex systems of **hosts that are not there at all times** and **change addresses constantly** is **difficult to manage**.

## A simple network app recipe

To create a network application we need to first define what we want from our app.

The basics of a network app are:

- **Running on multiple hosts**
- **Making the hosts communicate**

This seems pretty easy and it actually is that easy. And that is all thanks to the **layered architecture**. The layered architecture makes it so you don't need to care about anything between the two hosts, you don't even have to care about what the two hosts are doing beyond some surface understanding about the **transport layer protocols** your damned soul chose to use. We will use **TCP** in the recipe.

To communicate between two hosts, you need a **process** on both of them which is a way to represent a running program by your Operating System. Luckily **running any kind of code gives you a process**. Next, you need to open a **socket** on both of the processes. A **socket is basically a portal to another socket** as far as we are concerned, it is owned by one process (as long as you are not doing any shenanigans). With **open and connected sockets you can communicate between the two processes** and in this case two hosts.

You need to determine how you want to organise those socket connections. Most of your favourite network apps, like a minecraft server work on a client-server architecture, so let's use that.

To conclude, this are **how to create a simple networked app**:

- Take **two networked devices, one will be the client and the other a server**.
- Write some code on the **server** that:
  1. Opens a TCP socket and **listens for incoming connections**.
  2. **Accept the connection** from the client
  3. Sends some data, and processes the data the client sends to it.
- Write some code on the **client** that:
  1. **Connects to the server** by opening a TCP socket with the server as the target.
  2. Sends some data, and processes the data the server sends to it.
- Run both of the programs (tip: run the server first)
- Enjoy you network app.

*Note: If you choose a different kind of architecture, this will differ. If you choose a different protocol, you probably just need to open a different kind of socket on both sides.*

# URL's

You seen them, you know them:

|                 |          |           |                       |                    |                   |                                 |
|-----------------|----------|-----------|-----------------------|--------------------|-------------------|---------------------------------|
| URL:            | https:// | www.      | wilkuu.               | xyz                | /projects/inttech | ?this_does=nothing              |
| What is it for? | Protocol | subdomain | Domain (second level) | Domain (Top level) | Path              | Query (DDA thingies right here) |
| Used by         | Browser  | DNS       |                       |                    | HTTP              |                                 |

# HTTP

*This chapter was written 3 times because Libreoffice kept fighting over RAM with Firefox.*

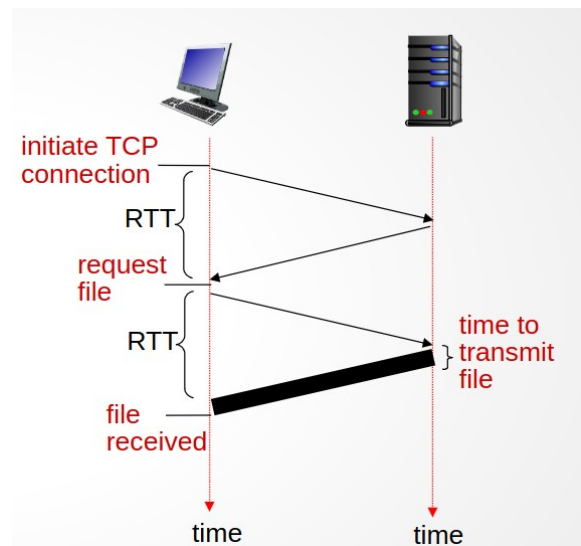
Now that you know how networked apps work. Let's go into the best example of a **Application Layer protocol** (Basically a networked app), HTTP.

**HTTP is the protocol that powers the web.** It is so good they made HTTP/2 and even HTTP/3. And some people use HTTP outside just serving websites, since it is so easy to use.

HTTP is a Client-Server protocol based on TCP, (Or QUIC when talking abt. HTTP/3), that is made to send text, images and other resources needed to deploy a website.

The working principles of HTTP are simple:

1. **Establish a TCP connection** with the server.
  - o The server **accepts** the connection
2. **Request** a resource/object using a HTTP request.
  - o The server **sends** you the resource
- At this point the client **receives** the resource and server **closes** the connection.



**Each of the steps needs to travel in the network to be completed**, so each of the numbered things is a **round trip**, round trip time is the time it takes the client to send something to the server and the server to respond. In the list above we can see **2 round trips**. In addition to that the **response time is dependent on the size of the object** so the final formula for the response time is:

$2 * RTT + F/R$ , where F(bits) is the file size and R is the transmission rate (bits/s)

In this case you only receive **one object per TCP connection**, this is not good for **response times**. That is because you **waste one round trip just the TCP connection per object**. And because you have to request a lot of objects to get the whole website, like website code(html, js, css), images, videos, fonts, etc., this adds up! In addition, making **TCP connections has overhead from the Operating System**. The only redeeming quality of this is that the browsers can request multiple connections making the process **parallel**.

This is where persistent HTTP comes in. It keeps the TCP socket alive for a bit longer after each request. With persistent HTTP you can maintain the TCP connection and thus avoid having it needing to be reestablished all the time, that is it. There is nothing more to it.

## HTTP how it works

Forgor 🧠, not enough time 🕒🛑, L + Ratio

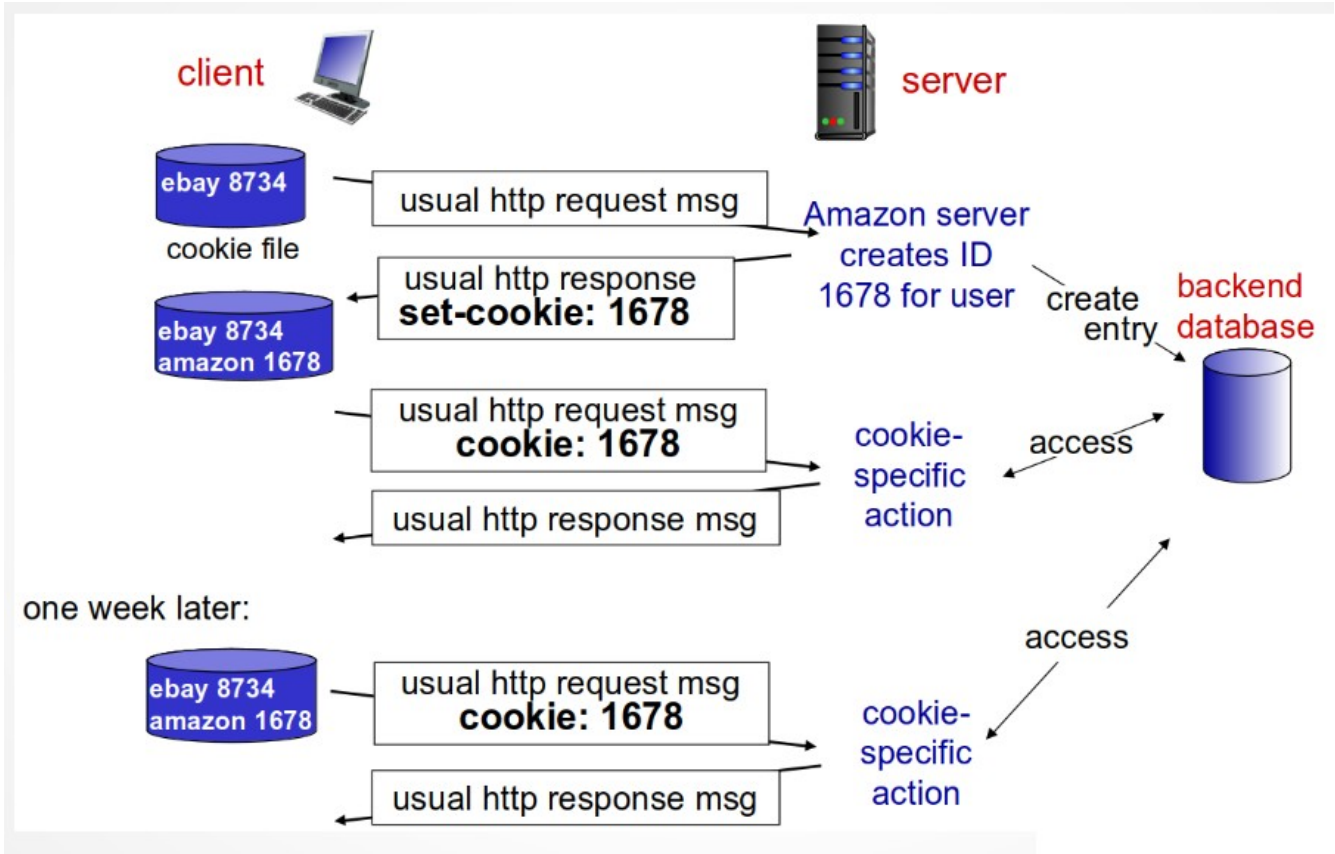
Watch this video on 2x while listening to nightcore, trust me bro.

<https://youtu.be/wW2A5SZ3GkI?t=321>

## Cookies

Because HTTP is stateless, you need some way to maintain sessions, so users can stay logged in when going to a different page for example.

This is what cookies are for. They are basically a piece of data that is unique per user and is used to fetch session data from a database. They can be used to track you, so they need your consent to use them.



## W3: Naming and addressing

### Ipv4 and ports

When you consider that **sending data over the network is like delivering a letter to another room in another building** it you might wonder, **how do I know which room in which building the destination?**

This is where **addressing** comes in. In most cases your **building has an address**, and an **apartment in the building as a letter/room number for it**. So, an example set of adresses would be:

From: Deez Street 10**b**

To: Joe's Avenue 69**a**

**Ipv4** is the most often used **Network Layer Protocol** and it defines a format for the **first part of this address(black)**:

From: 167.255.210.0

To: 42.0 .69.100

And because the **building symbolises a host in this allegory**. The first part of the address is the only thing that Ipv4 cares about, since it's **goal is to show what host the packet is for**. But there is also **the question of which room** you need to bring the mail. The packet, now technically a segment, has another address **in the Transport Layer Header**, the **port**, which shows **which socket(aka the room)** segment needs to go to:

From: 167.255.210.0:**42000**

To: 42.0 .69.100:**80**

This is how Ipv4 and adressing work on surface level.

You can also decode the ip into hex or bits like this:

167.255.210.0 → a7.ff.d2.00 → 10100111 11111111 11010010 00000000

**A building address has a street name and a building number**. It is also the case in Ipv4. The **subnet** is the **street name** and the **rest of the address** is the **street number**. To split the address into a subnet and the host portion of the address we can either use **classes** or **CIDR**.

## Classes:

| Class | Subnet mask size (Street name size) | Host part size (House number size) | Split: Subnet vs. Host |
|-------|-------------------------------------|------------------------------------|------------------------|
| A     | 8 bits                              | 24 bits                            | 100.1.2.3              |
| B     | 16 bits                             | 16 bits                            | 200.100.1.2            |
| C     | 24 bits                             | 8 bits                             | 255.200.100.1          |
| D     | n/a                                 | n/a                                | Multicasting           |
| E     | n/a                                 | n/a                                | Research purposes      |

## CIDR:

167.255.210.0 /24 → first **24** bits for the subnet (subnet mask) and the rest for the host

Also, the **things connecting different subnets are the routers**, they have **multiple interfaces** connecting to different **subnets**, so the **routers are like intersections** in our allegory.

## DHCP

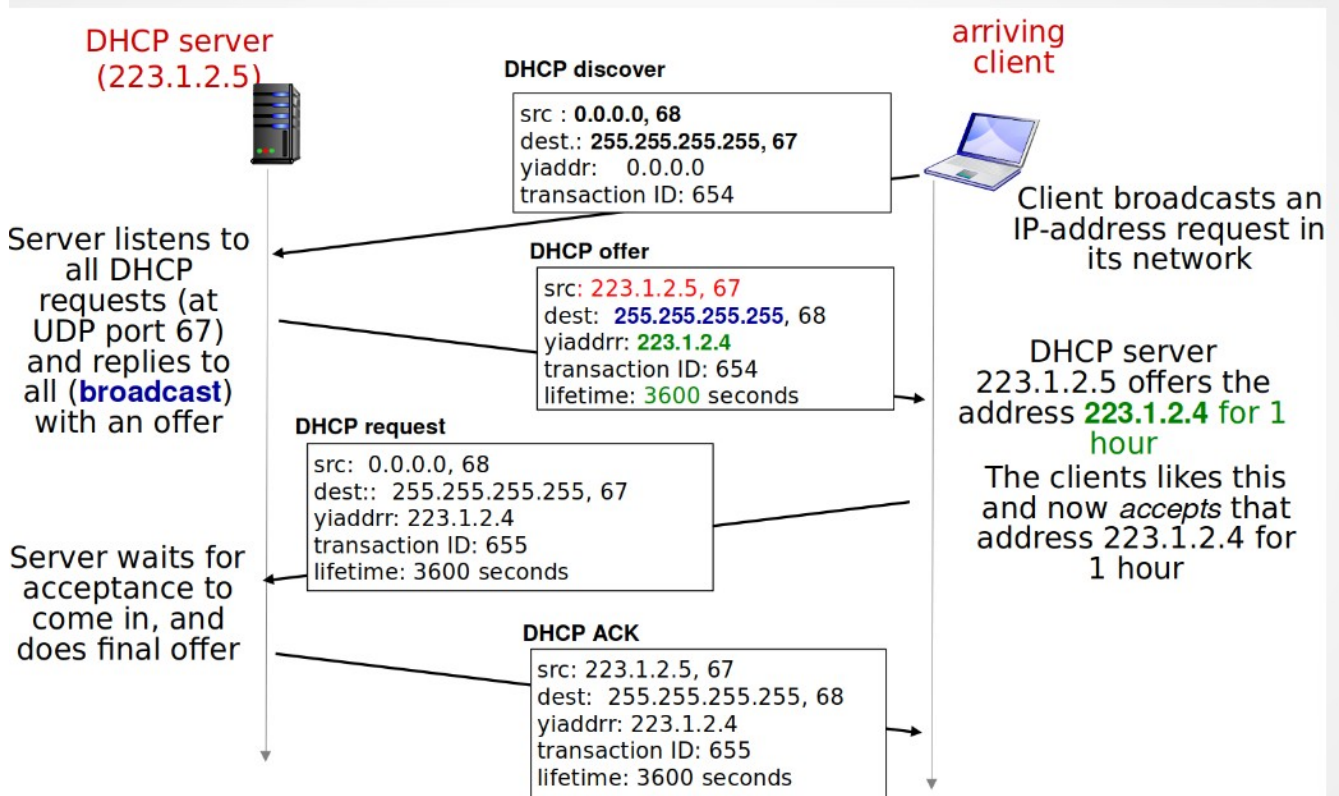
*Im stealing too much stuff, I know, but honestly DHCP is not that interesting.*

DHCP is a way to give a newly connected host a new IP address. It is good if your host disconnect and reconnect as in case of any mobile device. It also allows reuse of addresses, since hosts only have a IP address when they are on.

HCP overview(stolen from the powerpoint):

- Host broadcasts DHCP discover message (optional)
- DHCP server responds with DHCP offer (optional)
- Host requests IP address through DHCP request message

- DHCP server sends IP address in DHCP ack message

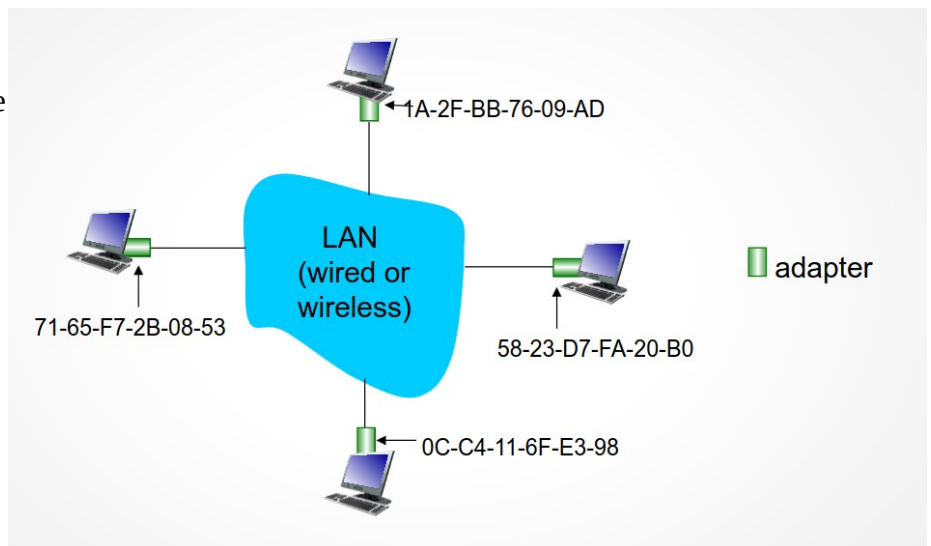


## Link Layer

In the **link layer**, the both **hosts and routers are called nodes**. And because some nodes can connect, disconnect and **reconnect somewhere else**. Just like **people can move to a different street**, so can a **node reconnect at a different place on the network**. And just like there is a need for people to have a **ID number(BSN, SSN etc)**, there is a need for a unique identifier per device. And that identifier is the **MAC (Media Access Control) address**:

1A-2F-BB-76-09-AD (48 bits)

The MAC address is used just to move a the frame **between two directly connected (neighbouring) nodes**. And each network interface has its own MAC.





## ARP (Address Resolution Protocol)

But because a Media Access Control (MAC) address is needed to communicate between two hosts, how do you communicate if you only know the IP address?

For this we can use ARP tables:

| IP           | MAC               | TTL (Time to Live) |
|--------------|-------------------|--------------------|
| 192.168.1.69 | DE-AD-BE-EF-3A-78 | 600                |
| 192.168.1.10 | 42-0B-B0-69-10-AC | 1000               |
| 192.168.1.1  | 0A-C4-B1-D7-CD-EF | 1200               |

TTL is the time for which the record is valid. After this time you need to look for the MAC address again. And **how do you look for the MAC address?** You **ask everyone!**

What you do is you send a ARP frame that has the **destination MAC address of FF-FF-FF-FF-FF-FF**, which means **everyone**. That frame **basically says**:

*“Who has the IP address of X and what is their MAC address?”*

Then **the node with that IP address responds** with a frame basically saying:

*“I have the IP address of X and my MAC address is Y”*

Note that this **only works if the node you want to send the frame to is directly connected to you**.

Otherwise, the correct step is to **send the packet to the local router** (Using ARP to get it’s MAC address) and make the router send it further.

The router then looks if it’s directly connected to the target node and if so, uses ARP (table) to find the targets MAC address and send it to that MAC address.

## DNS

But when you downloaded this summary **you did not type in the IP address of my web server in your browser**, instead you typed in `wilkuu.xyz/projects/inttech` (or you clicked on the link). How does that work?

Introducing **DNS** (Domain Name System).

**DNS is like Googling what address of the closest Albert Heijn is**. Or if you are a boomer, **looking up the company in a phone book**.

It **translates domain names to IP addresses** and is a **application layer protocol**, it has a **port of 53** and uses **UDP** in most cases.

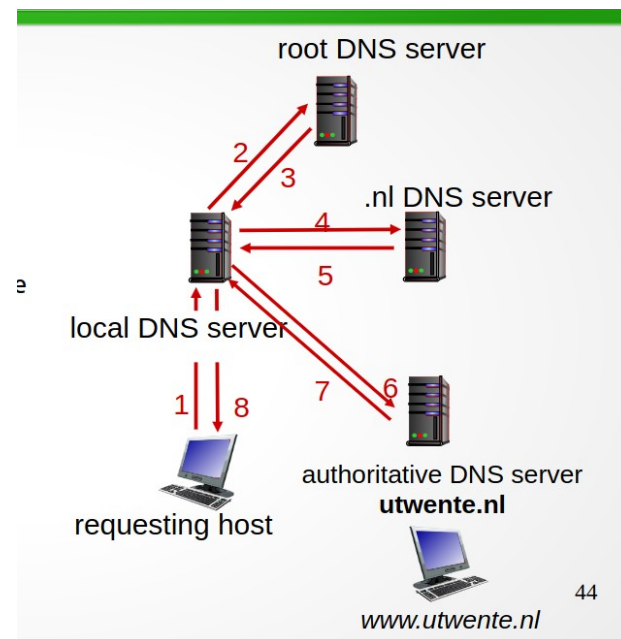
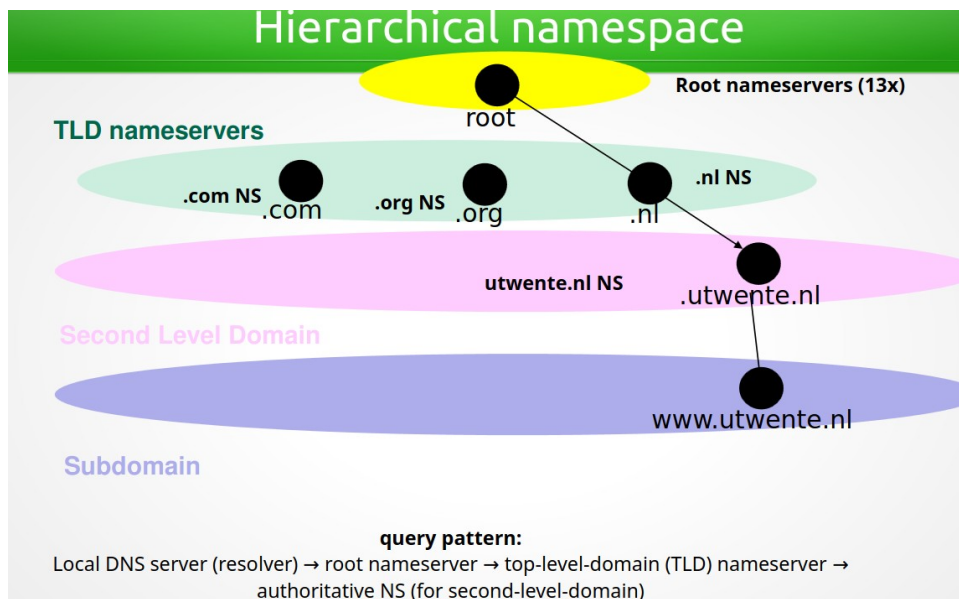
It is **distributed and hierarchical**.

A centralized server would be a **single point of failure**. Also the **link delay would make a DNS lookup too slow on the other side of the world** and the **traffic** to that one server would be massive.

And because it is distributed, some **names resolve to different addresses depending on which DNS server you ask**. (most likely a close one) This can be useful for load balancing (CDN's).

The way DNS is hierarchical, is that **there is a root that redirects to the top level domain server**, depending on the **top-level domain**, which in turn redirects to the **authoritative nameserver**, depending on the **second level domain**.

There is also the **local nameserver** which is outside of this hierarchy and acts as a **cache**(just like a ARP table) and a proxy (be the man in the middle) for the requests to all the servers in the hierarchy.



## W4: Wireless communication

*Speedrun time!!! Kinda incomplete, be warned!*

### Broadcasting

Two kinds of connections:

- Point to Point → Host to Switch → Ethernet nowadays
- Broadcast → Shared medium → WiFi, Ethernet in the past

Broadcast connections have a problem of **multiple hosts sending information at the same time: Interference.**

This is **solved using Multiple Access Control (MAC)**. *Really confusing, I know.*

Here is a good article about this, I don't have time:

<https://www.geeksforgeeks.org/multiple-access-protocols-in-computer-network/>

-- Skipping slide 1-21 of week 4 --

### Wireless

The wireless networks works in two modes:

- **Infrastructure:** You connect to a **Base Station**, which then are connected to the rest of the network.
- **Ad-Hoc:** You connect **directly between nodes**.

Transmitting wirelessly has some unique challenges:

*Stolen from the powerpoint*

Signal strength decreases as radio signal „**attenuates**“ as it propagates through matter

**Path loss:**  $1/r^2 \sim 1/r^5$  – indoor:  $1/r^{3.5}$

This is why local transmission overwhelms received signal strength

**Interference from other radio sources:**

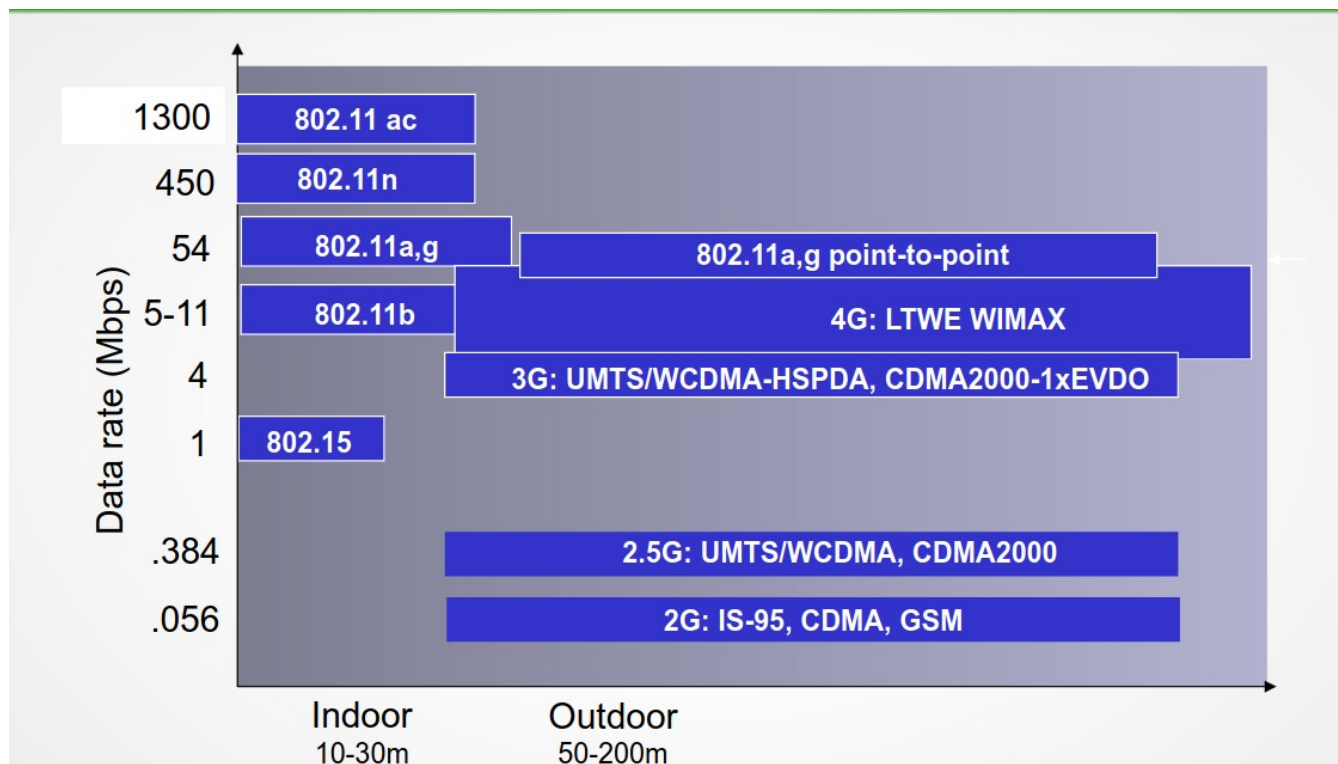
Standardized **frequencies shared by other devices communicating**

Devices such as **motors may interfere** as well

**Multipath propagation:**

Radio signal **reflects off objects/ ground arrives at receiver at slightly different offset /times**

Some wireless protocols with their speed and range: (802.11 is WiFi)



## WiFi: Connecting and Transmitting

Everyone asks for the WiFi, but nobody asks how is WiFi. WiFi is a wireless protocol that most mobile devices use for connectivity.

It can run in **both Infrastructure mode and Ad Hoc mode**, but **most often Infrastructure mode** is used.

To connect to a WiFi network a **host must associate with an Access Point**:

1. **Listen** for the Access Points to broadcast a beacon frame, which indicates the AP's presence.
2. **Select** one of the AP's
3. **Send a Association Request Frame**, which **tells the AP that you want to connect**
4. **Recieve a Association Response Frame** from the AP.
5. You are connected and probably want to **claim a IP address through DHCP**

*Note that you might have to authenticate before association.*

This is called **passive association**.

There is also **active association**, where you replace step 1 with:

1. **Broadcast a Probe Request**
2. **Listen for a Probe Responses** from AP you can associate with.
3. Continue with step 2...

Wifi uses either **CSMA/CA** protocol or **CTS/RTS** protocol for **Multiple Access Control** see powerpoint slides: 38-43 and <https://www.geeksforgeeks.org/multiple-access-protocols-in-computer-network/>

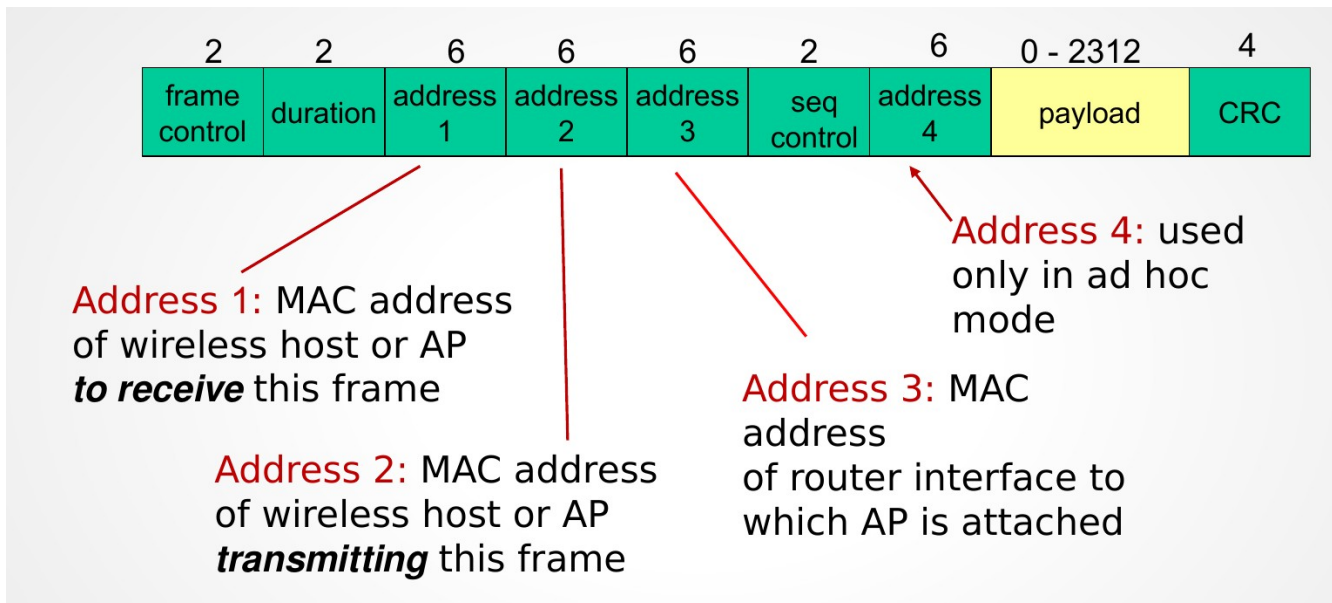
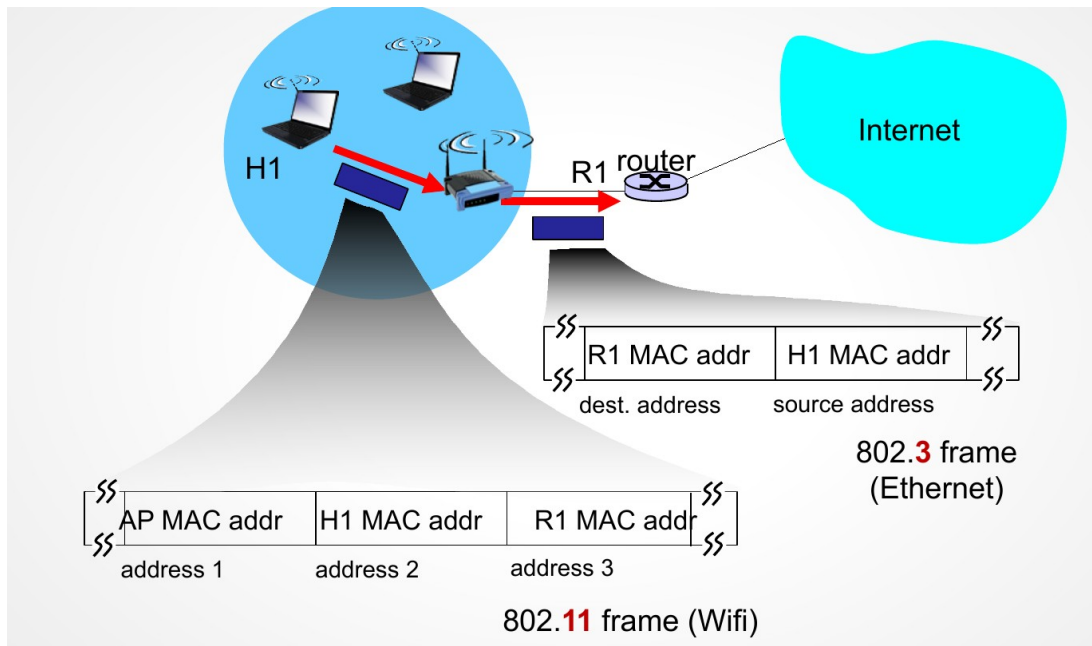
A quick summary of both is:

- **CSMA/CA**: Avoids collisions, but does not detect them. It sees if the channel is busy and if it isn't for some time, then it sends the data, and then waits for an ACKnowledgement.
- **CTS/RTS**: **Asks the AP** if it does not expect a transmission from any other node by sending a **Request To Send** frame. The AP responds with **Clear To Send** frame that also specifies the node that can start transmitting.
- **CTS/RTS** → **is better**, since you **avoid large frames colliding** and wasting transmission time.

# WiFi: Addressing

WiFi Addressing is a bit more complicated than Ethernet addressing, but it is doable:

| Header                   | Address 1                     | Address 2                           | Address 3                                         |
|--------------------------|-------------------------------|-------------------------------------|---------------------------------------------------|
| What is the address for: | The WiFi Frame goes here (AP) | The frame was sent from here (Host) | Where should the frame be forwarded to (A router) |



## W5: “Cybersecurity”

In this week on Internet Technology, time to explain the least Cybersecurity topic of Cybersecurity, the DDoS attack.

### DDoS basics

A Distributed Denial of Service attack is based on flooding a certain server or service with trash packets/requests, using multiple compromised(hacked) systems in order to disrupt the normal activities of the server/service.

This is like forting 10 people to prank call a pizza place, by ordering 100 pizzas to the north pole each.

DDoS attacks are often performed against big targets, like governments, big companies, financial institutions, ISP's etc.

In addition to that, smaller DDoS attacks are used as a signature weapon of the most oppressed group of people on earth, gamers, against other gamers.

### Where do I get some of them compromised systems?

Internet Of Things (IOT) devices of course. IOT devices are **small, cheap and often securityless hosts** that are connected **to the internet**, just **because it sounds cool to the consumer for them to be connected to a phone app**.

Those are perfect, since you **don't have to be a epic hackerman to get into a lot of them** and there **are millions of them** just waiting for you to compromise them. (*Most of them have the same default admin username and password.*) When you have so **many compromised systems** at your disposal, you can call them a **botnet**.

This is how Mirai, one of the biggest Botnets, was formed and then used to DDoS Dyn, which shut down it's operations shortly afterwards due to the sheer scale of the attack.

Botnet malware is in essence a networked application and it for controlling it it might use either a Client-Server or a Peer to Peer protocol.

Mirai used a Client-Server protocol by having a central Command&Control server.

Hajime, another botnet used Peer to Peer instead, controlling it means connecting to one of the compromised hosts, which then spreads the word to the rest of the botnet.

## Amplification and Reflection

To reduce the traffic the target returns to your botnet, you might consider reflecting the traffic of other hosts, this works by faking the source IP to that of the target on the Network layer header and sending this fake packet to the reflector server. The reflector thinks that the target sent this packet and responds by sending a packet to it. This way your botnet only needs a high sending bandwidth.

In order to make the most of your botnet you want to amplify the amount of traffic it generates. To do that you might use a amplification method along reflection. This means that the reflector generates a way bigger response than the fake packet it was sent.

An example of this is a fake DNS request:

```
$ dig any utwente.nl @y.y.y.y +notcp
```

```
00:00:00.937437 IP (... , proto UDP (17), length 79)  
x.x.x.x.40669 > y.y.y.y.53: ... ANY? Utwente.nl.
```

```
00:00:00.965843 IP (... , proto UDP (17), length 1468)  
y.y.y.y.53 > x.x.x.x.40669: ...
```

```
00:00:00.965843 IP (... , proto UDP (17), length 1468)  
y.y.y.y > x.x.x.x: ip-proto-17
```

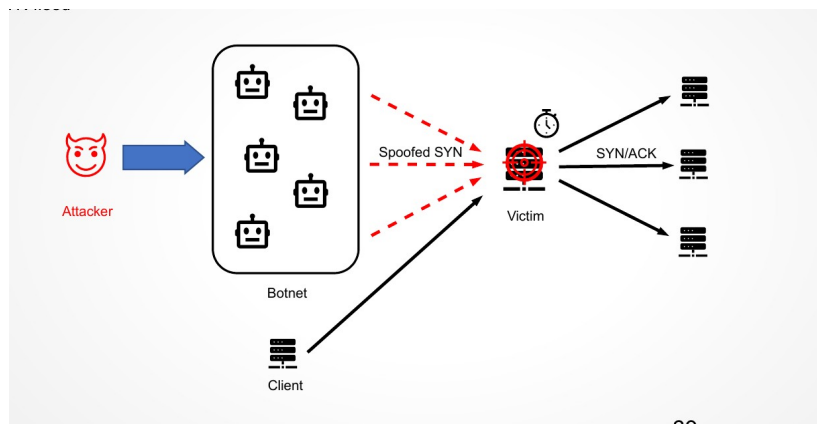
```
00:00:00.965843 IP (... , proto UDP (17), length 624)  
y.y.y.y > x.x.x.x: ip-proto-17
```

```
Amplification Factor = (received bytes) / (sent bytes) = (1468+1468+624) / (79) ≈ 45
```



## DDoS attack types

- **Volumetric attacks (Layer 3 and below):** Throw as many bits of data at the target as possible. Measured in **bits per second (bps)**
  - Sending a bunch of UDP packets. (Amplified using DNS)
- **Protocol (Layer 4) attacks: Exhaust as much of the target's resources** by for examples maintaining as many connections to the target as possible. Measured in **packets per second (pps)**
  - TCP SYN Flood: Fake as many TCP connections as possible by initiating handshakes for other hosts.



- **Application attacks (Layer 5/7):** Make as many request to the networked application as possible. Measured in **requests per second (rps)**
  - HTTP Flood: Send a bunch of GET requests.

## Mitigation

- **Anycast:** use multiple servers with the same IP
- **Blackholing:** redirect malicious clients to a fake server
- **Rate limiting:** limit the amount of requests/connections a single hosts can do.
- **Detect and discard spoofed packets** (eliminates reflection)
- **Firewall:** Block certain hosts from using the service

## **W6: IOT Guest lecture**

### **IDK, It was basically making fun of IOT and more abt DDoSing**

I put most of the things from here into Week 5 where they fit better.

#### **TLDR**

IOT devices are made to be novel and cheap and not secure. The manufacturers do not have incentives to make stuff secure, since security does not sell them products. This means that IOT devices are a security and privacy risk, unless regulated correctly.