Internet Technology

A summary? [PRE RELEASE]

Table of Contents

Preamble

This summary is definetely better than what is out there now. But I am a bit sleep deprived an 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 lecures 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)

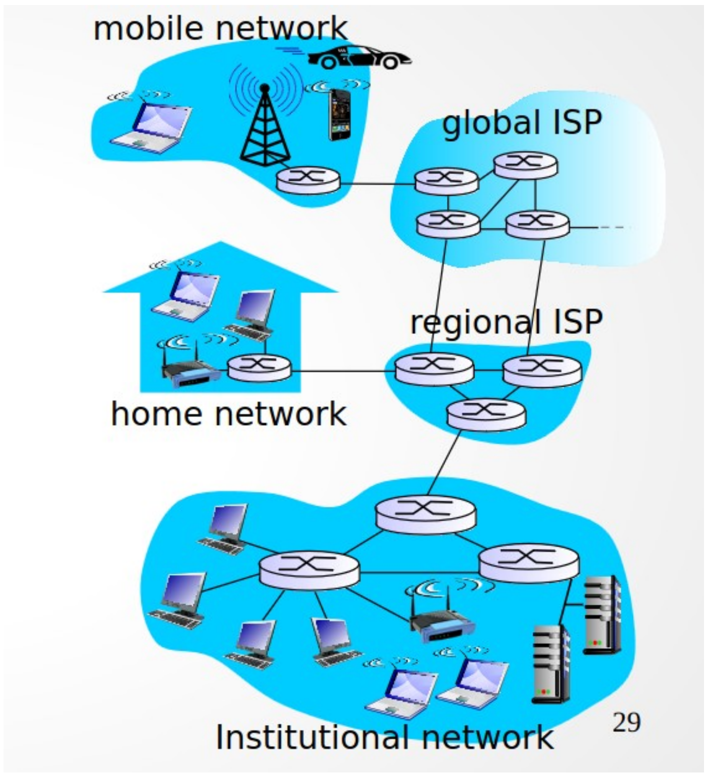
But anyway, take this with a grain of salt, since I might not stick to the material 100%. That is because there is some context that might people understand the abstract material more.

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 inpeded 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 bandwith, 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:

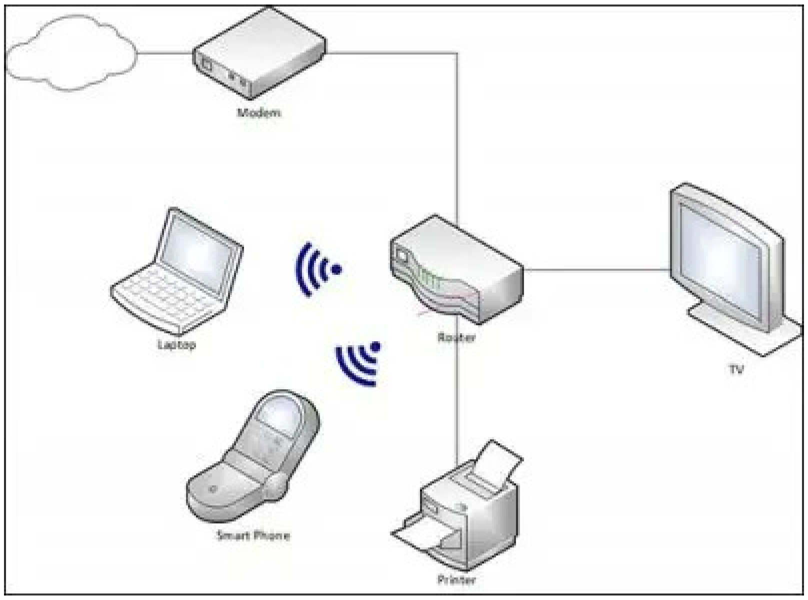
Starting device → Wifi router → Modem → {A bunch of ISP infrastracture} → {Global ISP’s infrastructure}→ {A bunch of targets ISP’s infrastructure} → Target’s modem → Targets router → Target

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

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 bandwith 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:

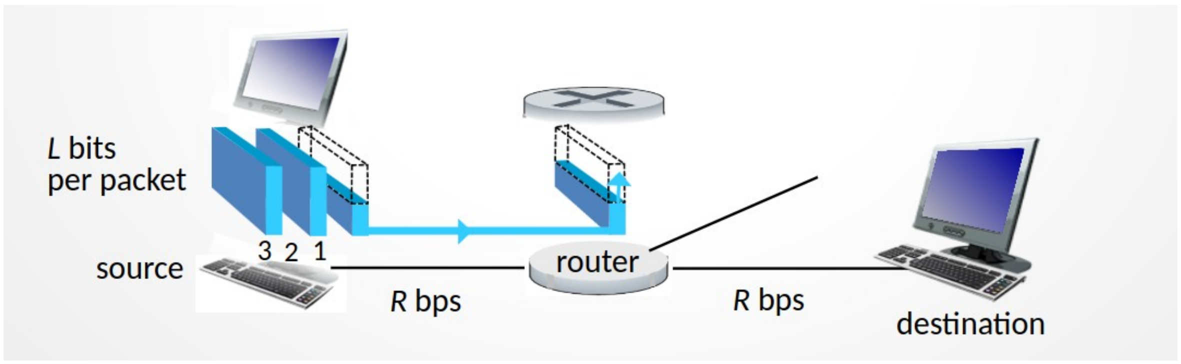
- **Routing:** Create a path from one point on the edge to another using smart algorithms and stuff
- **Forwarding:** Get the input of the router to the appropriate output, in order to make the packet travel on the path specified in the last step.

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.

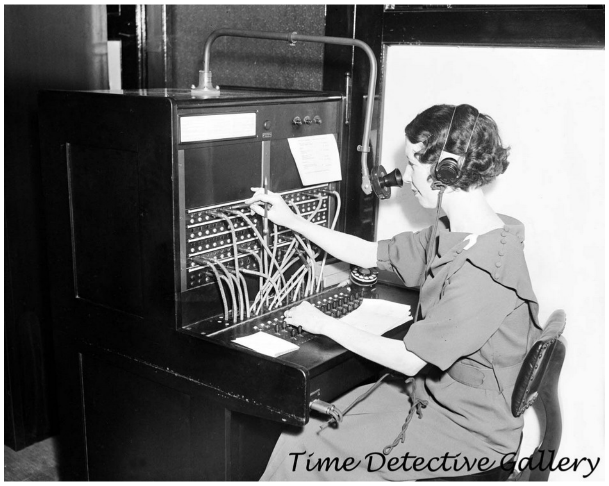
The packet needs to be received in its entirety before it can be sent further.

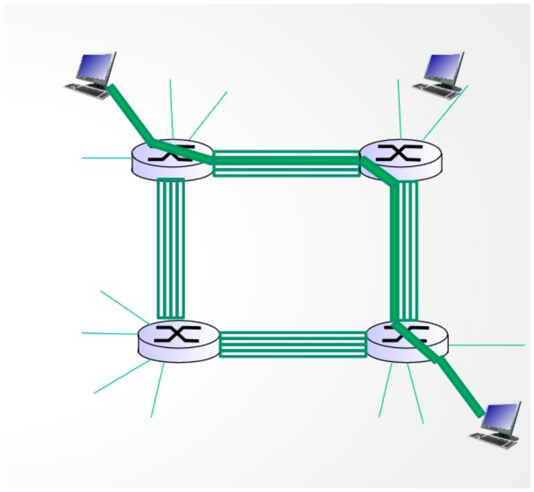


Circuit switching

Instead of switching using packets, you could also have mutltiple 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 bandwith 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🤓?

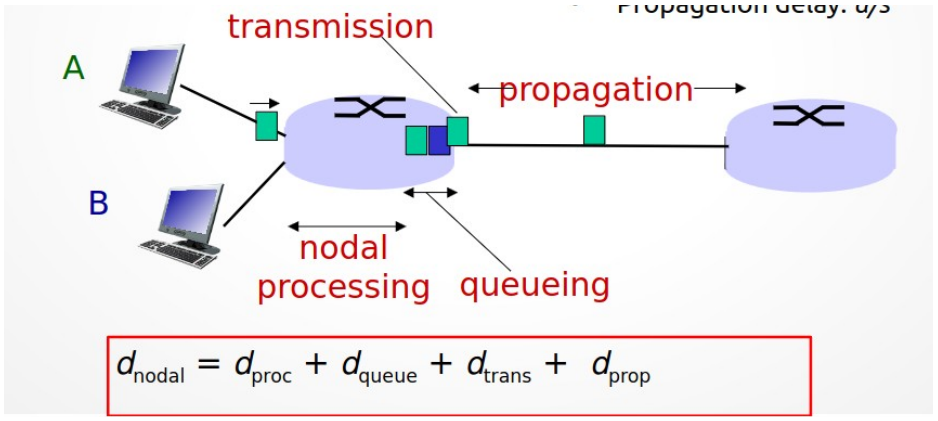
The processing delay is where the router checks if there are bit errors (AKA if the packet did not get corrupted) and determines where to send the packet next. It has no formula and is mostly negligible.

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

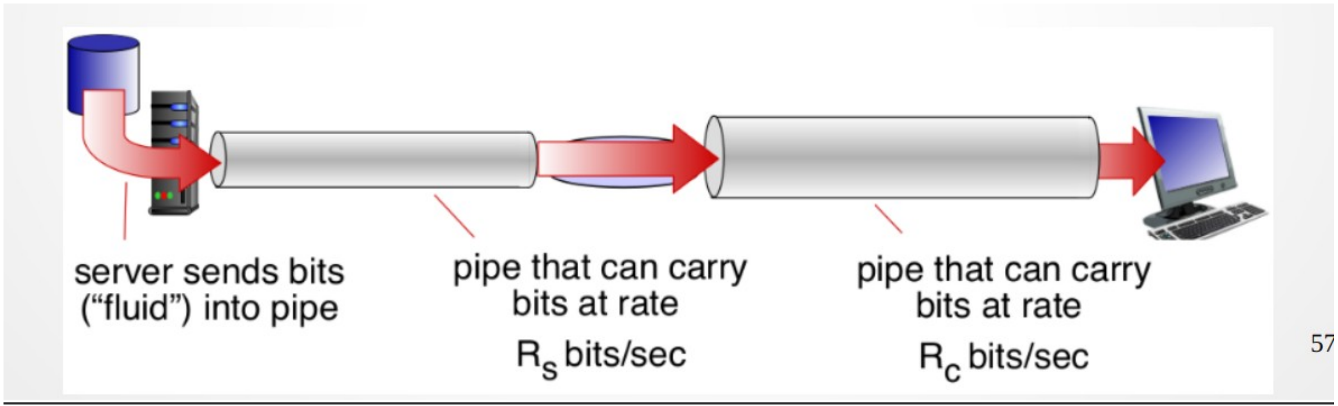
The **transmission delay** is when the bits of the packet are transmitted across the link. The formula is: L/R where L is the size of the packet in bits and R is the bandwith in bits/second.

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 possibillity of a bottleneck. In this image the througput of the first pipe is less of the next one limiting the second pipe from using full capacity.



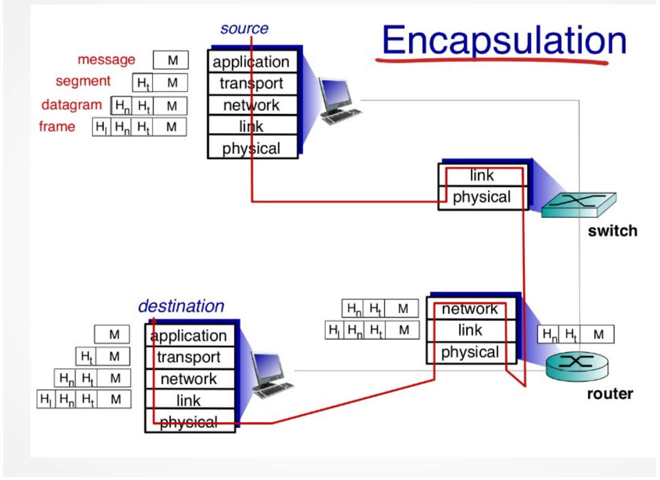
When measuring the amount of data that crosses from the source to the destination (with routers in the middle) we speak of **througput**, which is the bits per second on received at the destination.

- Instantenous: Througput right now
- Average: Througput over a given period of time.

As said before, the packet also is labled 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	Explaination	Examples
1	Application layer	The networked application	HTTP (Browsers), SMTP, (Email)
2	Transport layer	Communication protocol between apps	UDP, TCP
3	Networking layer	Routing between the source and destination	Ipv4, Ipv6
4	Data-Link layer	Communication between two neighbouring nodes	Ethernet, Wifi, Bluetooth
5	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:



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 **elliminates 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 adresses 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 **trasport 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 incomming 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.

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.

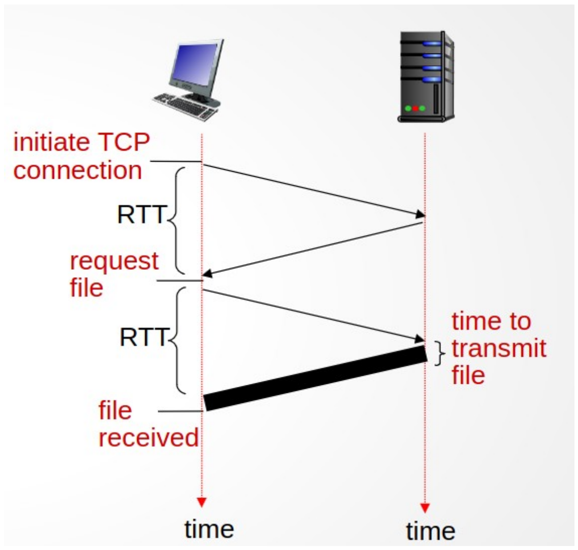
HTTP is the protocol that powers the web. It is so good they made HTTP/2 and even HTTP/3. And some people use HTTP outside just serving websites, since it is so easy to use.

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.
   - The server **accepts** the conection
2. **Request** a resource/object using a HTTP request.
   - The server **sends** you the resource

- At this point the client **recieves** 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 thingies 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)

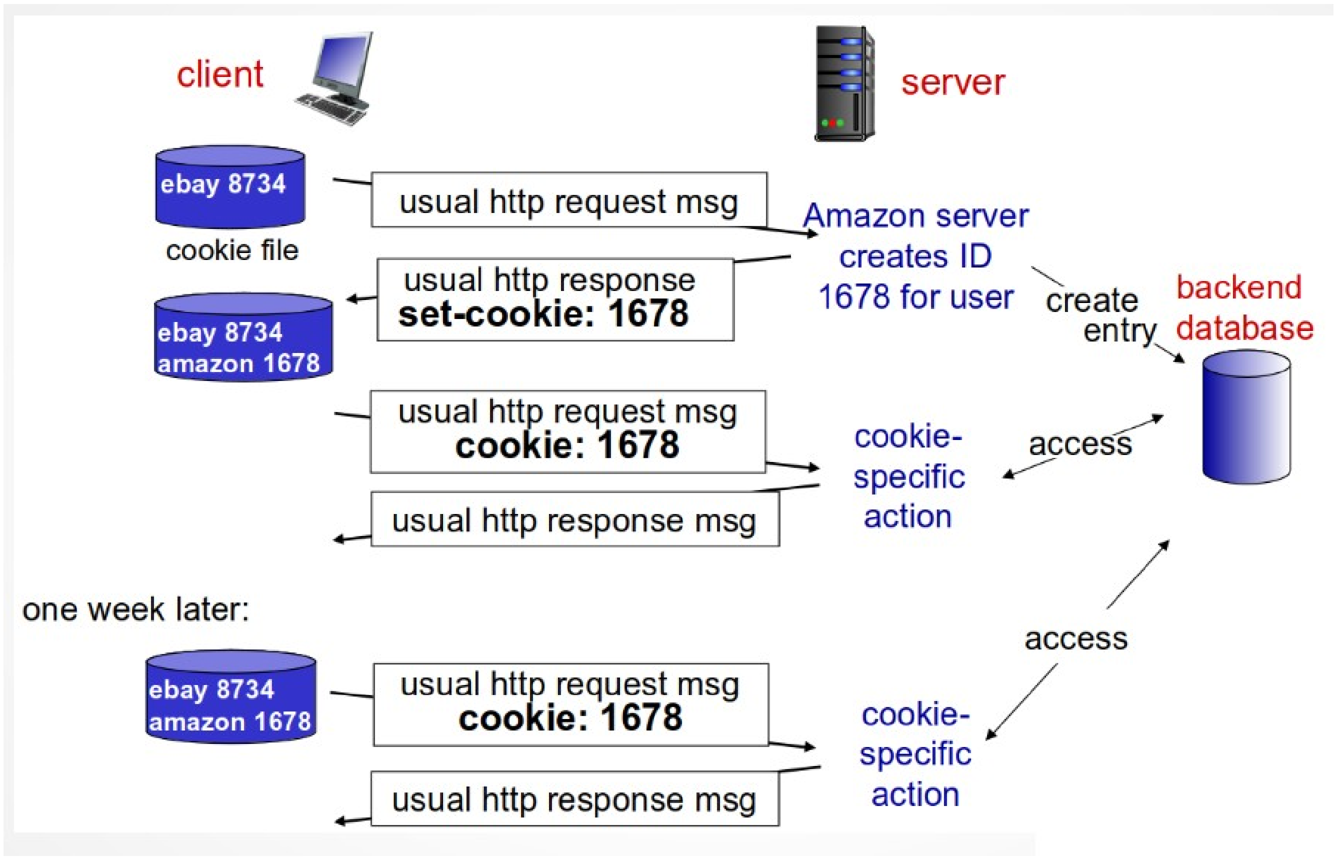
In this case you only receive **one object per TCP connection,** this is not good for **response times.** That is because you **waste one round trip just the TCP connection per object.** And because you have to request a lot of objects to get the whole website, like website code(html, js, css), images, 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.

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

TODO

W4: Wireless communication

TODO

W5: Cybersecurity

TODO

W6: IOT Guest lecure

TODO