__/__/__

# ~~DBC~~

# UNDERSTANDING HTTP

1. Stateless

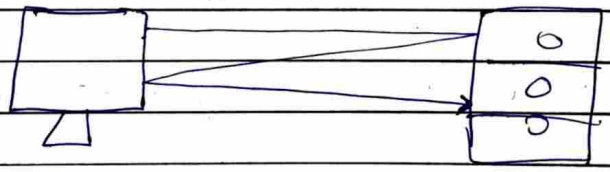
↳ (no memory of past interaction)

↳ each HTTP requests have all the necessary info for the server to process

→ every request is treated as brand new

↳ Stateless system allow each request to be handled independently, which makes them easier to scale horizontally, more fault-tolerant, and better suited for modern cloud and microservices architecture.

2. Client - server model




→ HTTP protocol states that communication is always init by client to get some kind of response by server.

→ HTTP uses uses TCP for connection.

↓

more reliable (connection based)

- HTTP/0.9 (1991) Purpose: Just fetch documents
  - Only support GET
  - No headers
  - No status code
  - Only plain text

Why? The web was just text document.

__/__/____

- ~~The 7 layers of OSI~~ *

→ HTTP/1.0 → each request open a new connection
(1996) ↳ inefficient

→ HTTP/1.1 → purpose : improve performance
(1997)
- Persistant connections (keep alive)
- multiple requests over one connection
- Added caching
- Added Host header

- HTTP/2.0 → purpose : speed & efficiency
(2015)
- Binary protocol (faster then text)
- Multiplexing (multiple requests at once)
- Header compression
- One connection handles everything.

Solved : Head-of-Line blocking in HTTP/1.1

- HTTP/3 purpose : Faster & more reliable
(2022)
- Runs on QUIC instead of TCP
- Uses UDP
- faster connection setup
- better performance on poor networks (mobile)

Why it exists : TCP was bottleneck.

1. HTTP is stateless
2. HTTP/1.1 is still everywhere
3. HTTP/2 improve performance
4. HTTP/3 improve reliability on bad networks.
5. HTTPS = HTTP + encryption

__/__/____

# Types of HTTP headers #

- Request Headers
  - User - Agent
  - Authorisation
  - Cookie
  - Accept

→ Request headers helps server understand the clients environment, preferences and its capabalities.

- General Header
  - Date
  - Cache - control
  - Connection

- Representation Headers
  - Content - Type
  - content - Length
  - content - Encoding
  - Etag

- Security Header
  - Strict - Transport - Security (HSTS)
  - Content - Security - Policy (CSP)
  - X - Frame - Options
  - X - Content - Type - Options
  - Set - Cookie

__/__/__

- **Extensibility**

→ HTTP is highly extensible because headers can be easily ~~add~~ added or customise without altering the underline protocol.

- **Remote control**

→ HTTP headers acts as kind of RC on the server side. they allow the client to send instructions or preferences to the server influencing how the server responds or processs requests.
e.g, content-type negotiations.

# HTTP methods

GET POST PUT (replace the data.) PATCH (use patch unless you have specific use of PUT) DELETE

→ HTTP methods exists to represent different kinds of actions that a client (browser/en) can request on a server. Instead of every request doing the same thing method define the intent of the interaction.

# Idempotent vs Non-idempotent

1. **Idempotent**

An operation is idempotent if making the same request multiple times produces the same result on the server.

One request = many requests → same effect

e.g, GET , PUT , HEAD , DELETE , OPTIONS

__/__/____

2. Non-Idempotent

An operation is non-idempotent if repeating the same request changes the result.

One request ≠ many requests

E.g, POST PATCH

# OPTIONS method # used in CORS → same origin policy.

→ used to ask the server what operation are allowed on a resource.

What OPTIONS DOES:

- Returns supported HTTP methods for a URL
- Helps client check capabilities before making real requests.
- Used heavily by browsers for CORS preflight

# CORS # (cross-Origin Resource sharing)

↳ is a browser security mechanism that controls which website are allowed to acess resources from another origin.

[CORS decids wheather your browser is allowed to talk to another backend]

• Why CORS exits:

(Browser follow the same-Origin Policy)

- frontend and backend must have the same orgin (scheme + domain + port)
- Otherwise, the browser blocks the response

→ Only browser enforces this

→ Postman / curl ignore CORS

→ Preflight exist for security.

__/__/__

- What is Origin?

  An origin = protocol + domain + port

  e.g.,

  https://example.com ✓

  https://example.com:3000 ✗ (different port)

  http://example.com ✗ (different protocol)

- in a cross origin request there two type flows

  ① Simple request ② Preflighted request.

① Simple request

Cross origin request that the browser allows without sending an OPTIONS request first

Conditions:

- Allowed http methods
  - GET
  - POST
  - HEAD

- Allowed headers (only these)
  - Accept
  - Accept Language
  - Content-Language
  - Content-type (with restrictions)

- Allowed (content-type)
  - text/plain
  - application/x-www-form-urlencoded
  - multipart/form-data

__/__/____

② Preflighted Request :

is when a browser first sends an OPTIONS request to ask permission before sending actual request.

- When preflighted happens :

If any these are true

- Method is not GET, POST, HEAD
  e.g, (PUT, DELETE, PATCH)
- Custom headers are used
  (e.g Authorisation)
- Content-type : application/json

- Option request :

An Option request is used to ask a server what is allowed before doing anything risky.

→ The request has a content-type other than application/x-www-form-urlencoded, multipart/form-data or text/pain.

# Response header

{ :
Access-control-Max-Age : 86400 ⇒ don't make any preflighted request to me.

→ This will be same for atleast next 24 hours.

- TCP → Transmission control protocol → acknowledge based mechanism. (feedback)
  (Transport layer protocol)

① What does TCP does?

① send data (appropriate transmission rate)

② segment data

③ Congestion control
↳ a technique to prevent the network from becoming overloaded with too much data at once.

④ Identify and retransmit message.

* Application

① FTP → port 21 or 22

② SSH →

③ Email →

④ web browsing → HTTP | HTTPS

- Key features *
  - connection oriented
  - full duplex
  - Point to point transmission (should have 2 endpoints)
  - Error Control
  - Congesion control

- segment Header

• Headers •

↳ small blocks of metadata attached to data that tells the network how to handle, route and deliver the data.

• TCP

- Byte streaming
- connection oriented
- Full Duplex
- Piggybacking
- Error control
- Flow control
- Congestion control

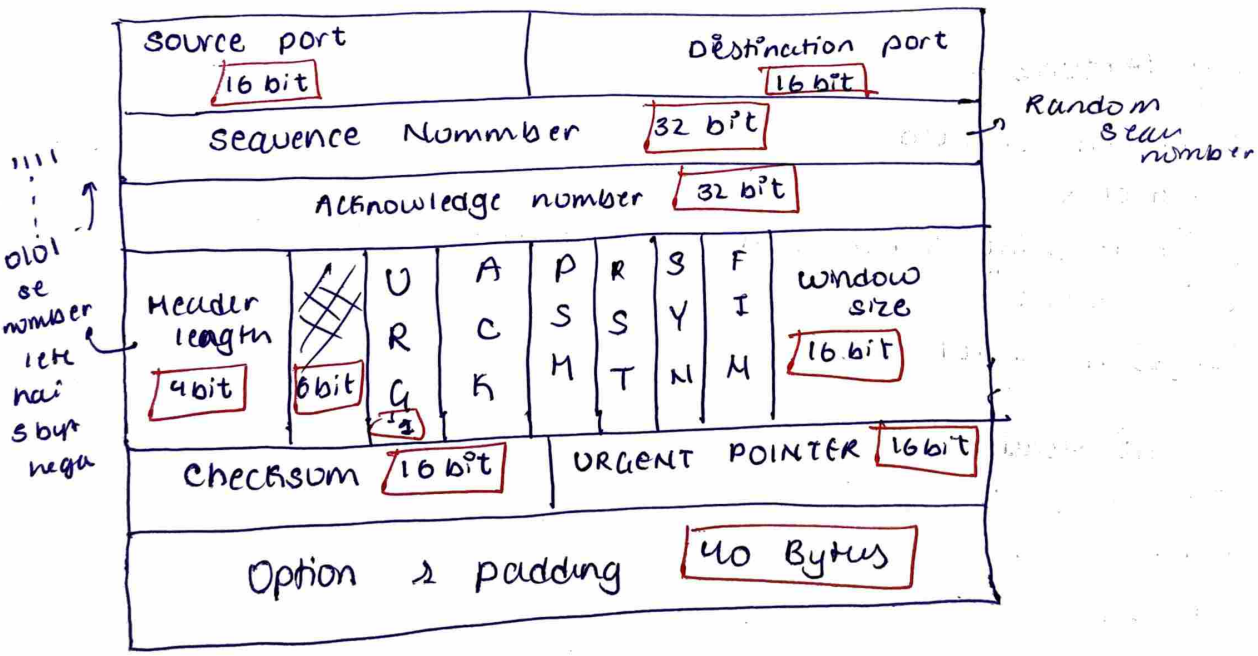
0-1023 → well known port

16 bit ⇒ $2^{16}$
↓
0-65535

o

collection of byte → segment

• * TCP Header (20 - 60B) (min → max.)



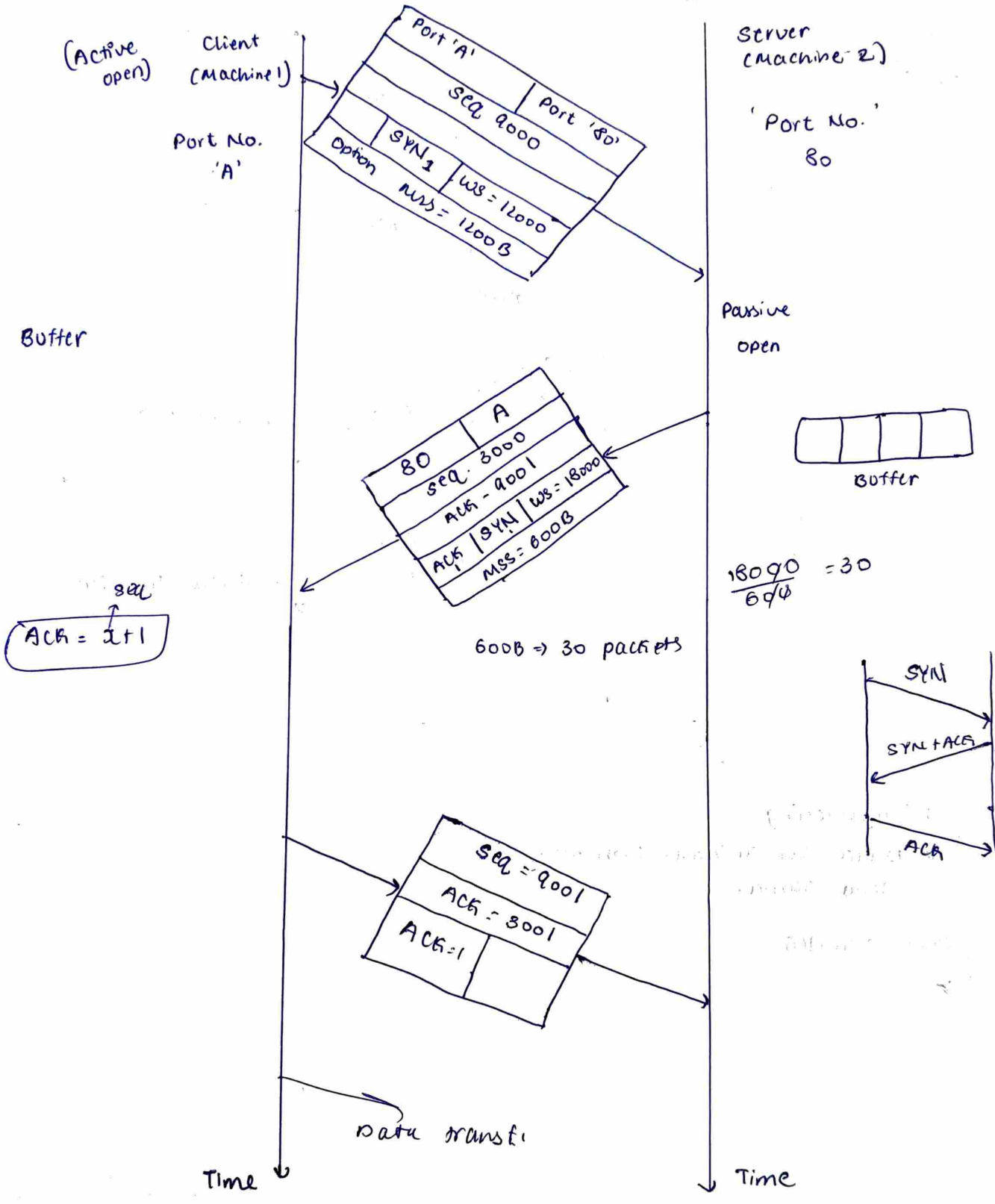

Checksum → used in error control

↳ urgent pointer → isma data urgent hai aur baaki normal hai.

Ⓟ → O & P → Maximum segment size (MSS)

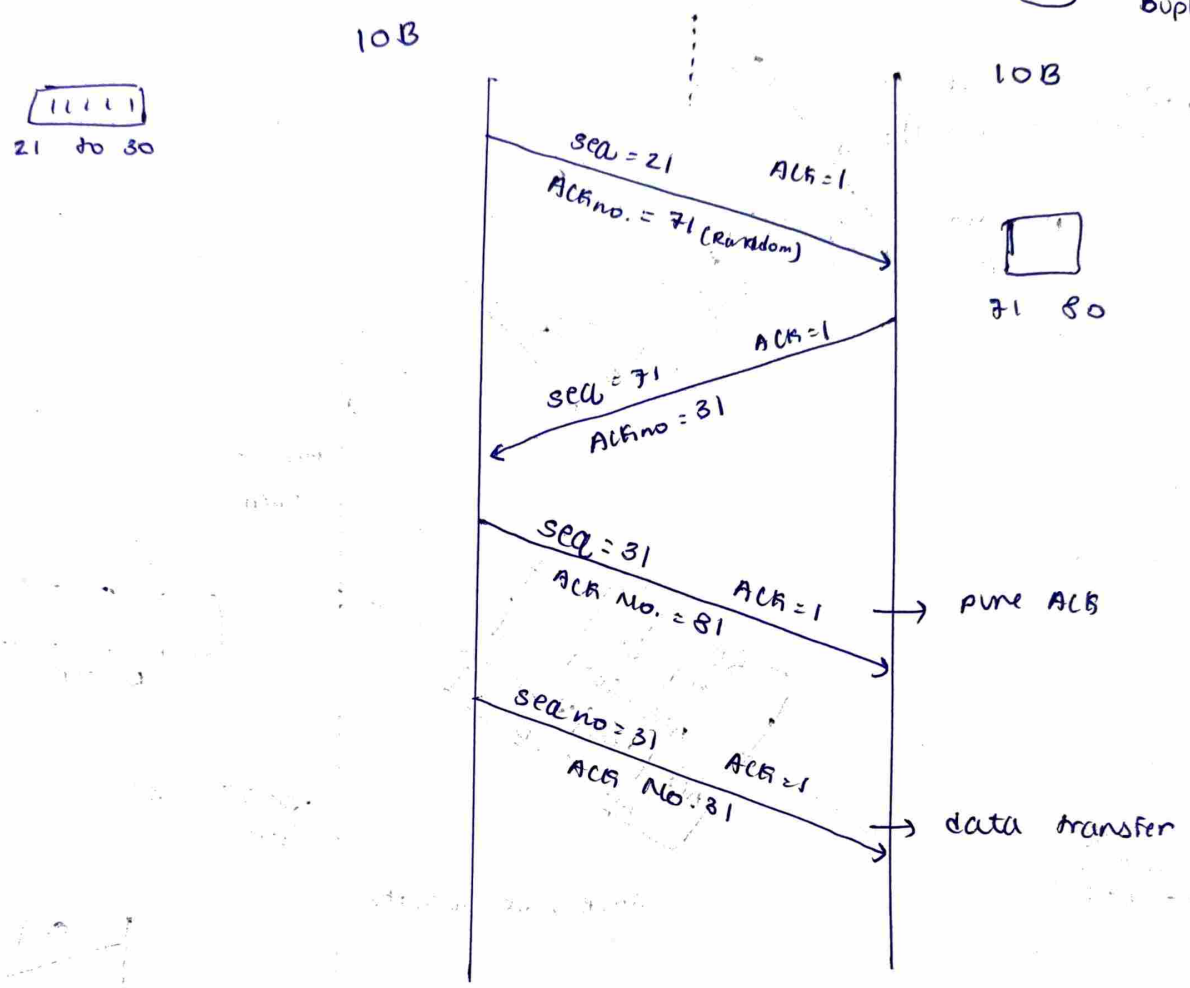
→ TCP 3 way connection Establishment :

Seq → 0 → $2^{32}-1$



# TCP Data Transfer #

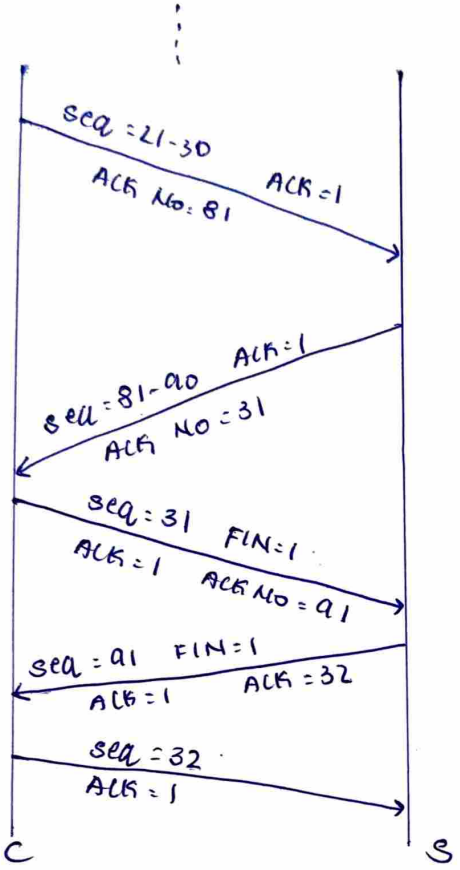
C ⇄ S Full Duplex

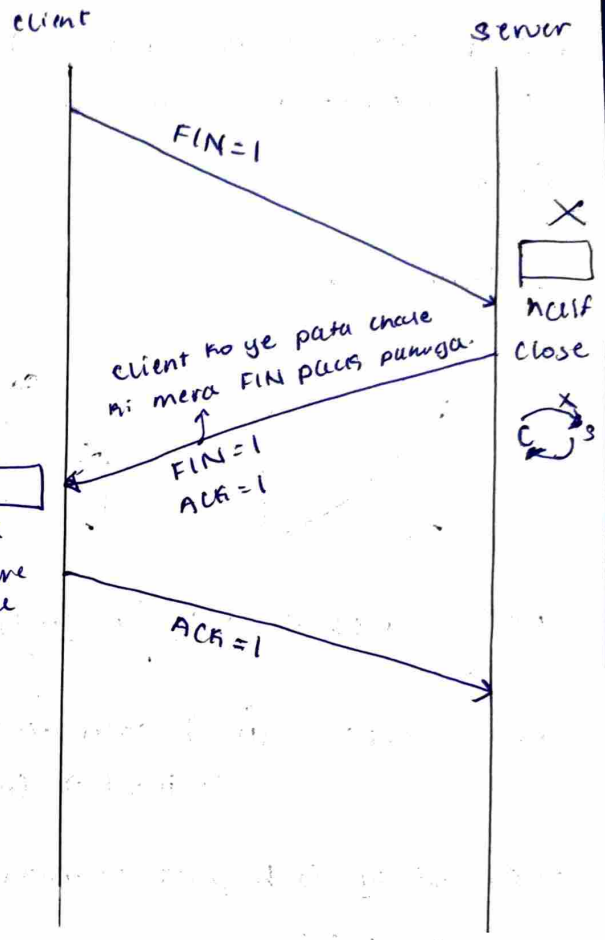



Piggybacking
↳ Data aur ack ko ek sath mai send karna.

Pure ~~ACK~~ ACK
↳

TCP connection termination ; 3 step or 4 step



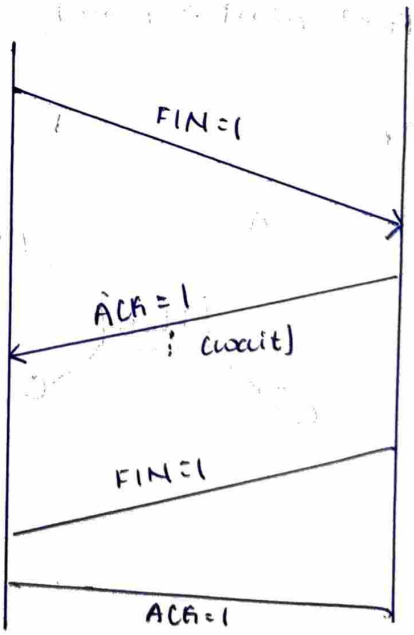




ye seaverce number consume karta hai.

FIN :) FINISH

□ → Jo bhi resources server reserve kiya hai unko release kar dega.
X

seq = 31 → ye consume nhi hoga.




Agar server ka man nhi kar rha hai toh vo bas ACK = 1 aur data bhej sakta hai. Bad mai jab FIN = 1 hoga connection band hoga.

# Application Layer Protocol :

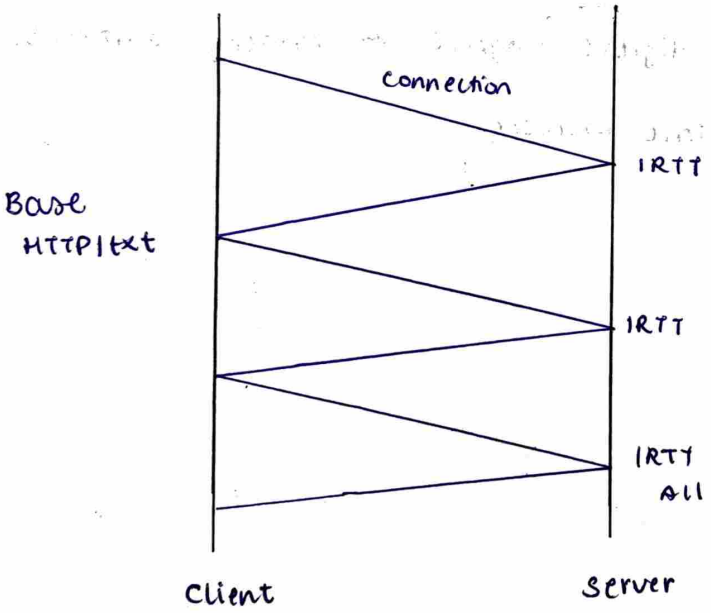
## HTTP (Hyper text transfer protocol)

- Port 80
- Itself not reliable
- Inband Protocol
- Stateless
- HTTP 1.0 Non-Persistent
- HTTP 1.1 Persistent
- Commands (Head, Get, Post, Put, Delete, Connect)

- **Persistant HTTP connection (HTTP/1.1)**

① server leaves connection open after sending response 1 RTT for all referenced object

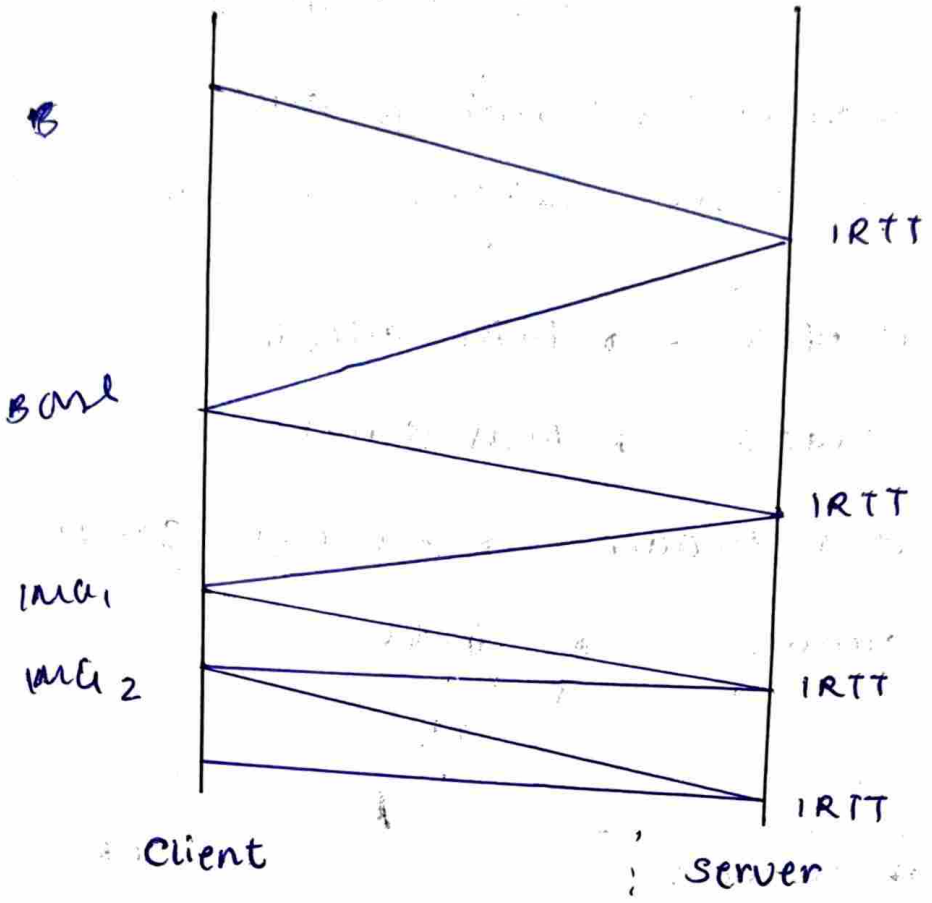
② Less overhead



# Non-Persistant HTTP (HTTP/1.0)

① It require 2 RTTs per object

② More overhead



__/__/__

- What happens when we hit URL in browser?

- What constitutes a URL?

Human-readable way to know what we are interested in domain?

Optional KV pairs we can pass to additional optional info in the request

https://www.google.com/api/search?q=home

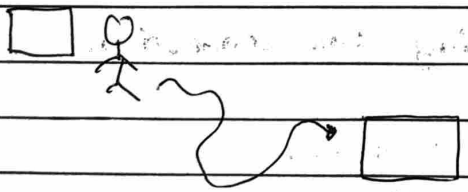
scheme

path

Tells browser which protocol to use while connecting other scheme: http, ws, etc

on the product we are accessing, I am interested in this path / resource.

- The DNS ~~to~~ resolution.



google.com

17.53.21.253

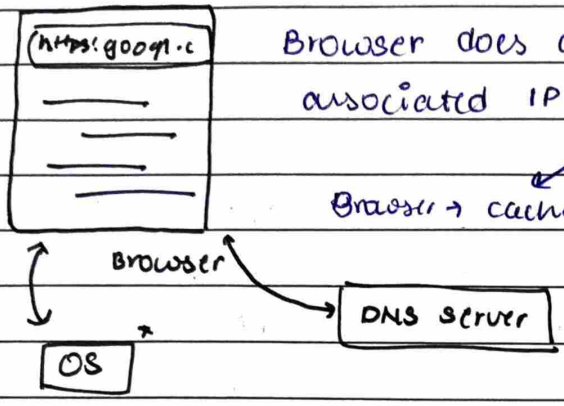
Every machine on the internet has an "address" enabling us to reach it over the network,

↳ this is IP address

* easier to remember "google.com" than the IP address. Hence, we need a way that converts

google.com ⟶ 17.53.21.253

__/__/____




Browser does a DNS lookup to get the associated IP address.

IP of google → not going to change

Browser → cached. DNS information is heavily cached

in the browser ~~in the~~

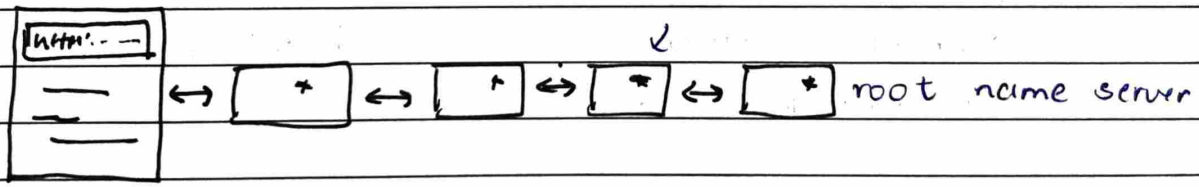
in the operating system

across all machines of DNS resolution.

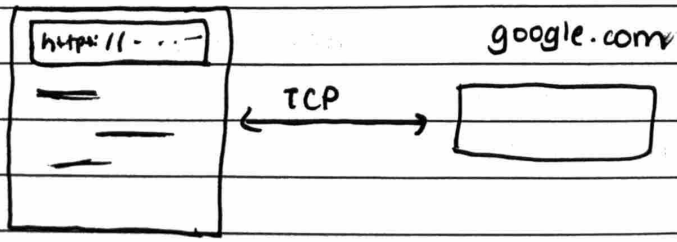
What looks like a simple call, actually involves lots of machine

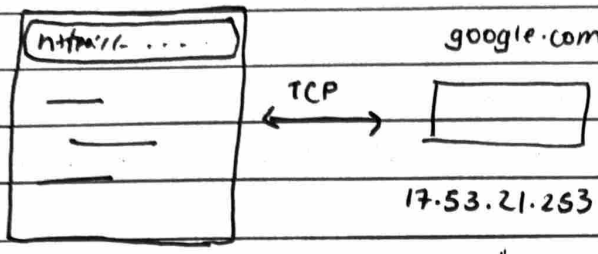
iterative / recursive

cached

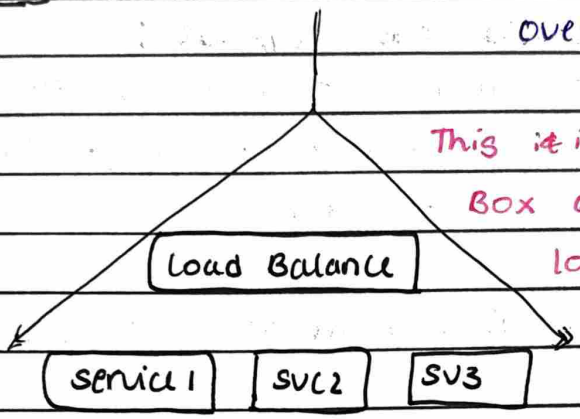



After this resolution process, browser has the IP address to connect to establishing the connection.






Browser now establishes a TCP connection with the machine [server] and can now talk to it over the network.




This it in itself is a Pandora's Box and constitutes of 1000's of machines

## Sending the request :-

other protocol possible

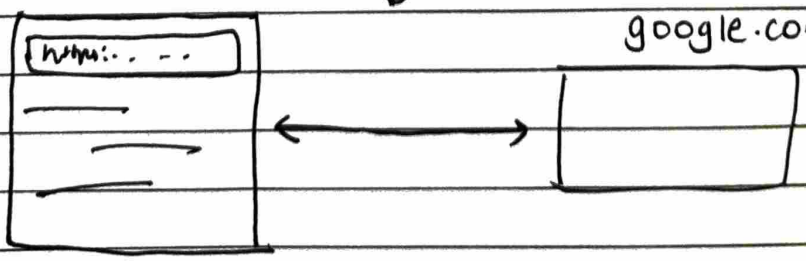
Browser now compliers the request into HTTP specification and sends it across to the server.

```
GET /api/search?q=home HTTP/1.1
Host : www.google.com
connection : Keep-alive
```

gives we are just hitting URL in browser it fires an HTTP GET to the server

→ HEADERS

↳ instructions to server + meta data



• why protocol exist "

__/__/____

* HTTP is protocol that specifies :

1. how to pack the data
2. what to do before, during & after the request

• Server procuses the request :

Once the server ~~reci~~ receives the HTTP request, it parses the above message and understands what needs to be done.

* Server may just load the file from local disk & server
it may make call to database to get responses
it may throw error if malformed.

it complies a proper HTTP responses
and respond back over same of TCP connection
(html, text, JSON, etc)

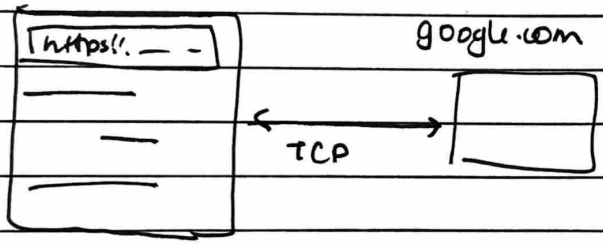
```
HTTP/1.1  200  OK
Content-Type : text/html
Content-Length : 2042
```

- Browser upon receiving the response :

| | |
|---|---|
| HTTP/1.1 200K OK ___ status code | Browser upon recieving |
| content-Type : text/html _ type | the response parses |
| content-Length : 2042 _ length | the message, extracts |
| <html> <head> ... _ body | the info and |
| | "renders" |

↳ http message

↳ showing html as html
text as text, etc




* if browser does not support the response type then it downloads the file locally.

- When HTML is rendered, browser may come across

1. linked CSS file
2. img tags to render an image
3. inline Javascript code

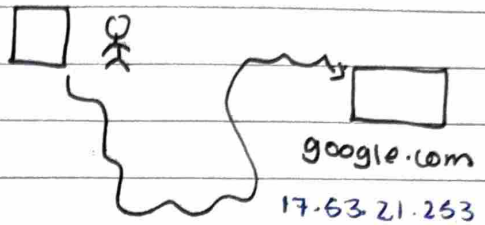
↳ it fetches the ~~additionally~~ additional files by going through the exact same process.

↳ starts executing it
[may involve making more HTTP requests]
↳ API calls

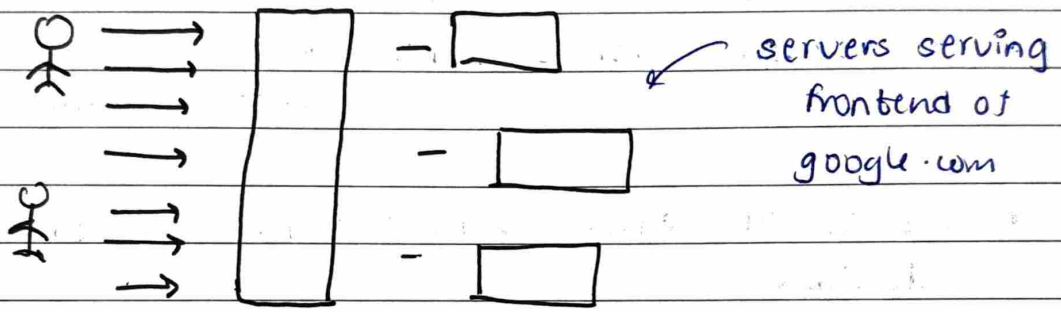
__/__/____

## * How DNS works *

To connect to a machine, you need its IP address
But how do we discover




google.com ⟶ 17.53.21.253




So, there would be a place where the mapping is stored.

www.google.com ⟶ 17.53.21.253
⋮

along with other records like gemini.gv → gemini.es

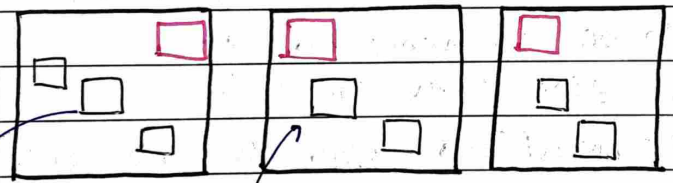
__/__/____

| | | |
|---|---|---|
| www.google.com → 17.53.21.253<br>bard.google.com → _ _ _<br>⋮<br>MX, Txt, etc | zone<br>google.com | All DNS records of a domain are part of zone (logical entry) |

→ what we configure on Route53, GoDaddy etc.

* **Authoritative Name servers** (Zones are served via)

An authoritative name server hosts multiples zones, and it answer DNS questions for the zones it is occurs.




Zones are served from authroritative NS

ns1.gns.com or aws.dns.net

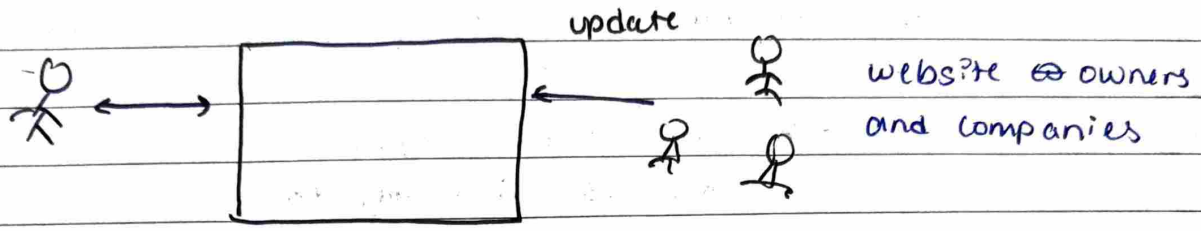
ns2.gns.com duggup.com

ns3.gns.com

Typically offered by cloud providers or ~~god~~ godaddy, etc

Now, how would we reach to these servers?

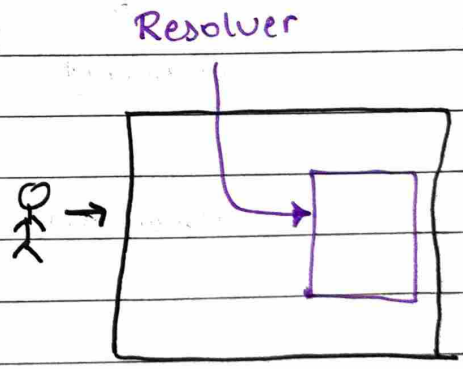
Option 1: let there be a single massive "system" that everybody reaches out to when they want this information.




This centralised way is not scalable, fault tolerant, or even

concers: volume of data, number of requests and updates + any change in any info needs to be communicated to (would has gone with option 2)

Option 2: Decentralized approch - no one machine ~~knows~~ knows it all.

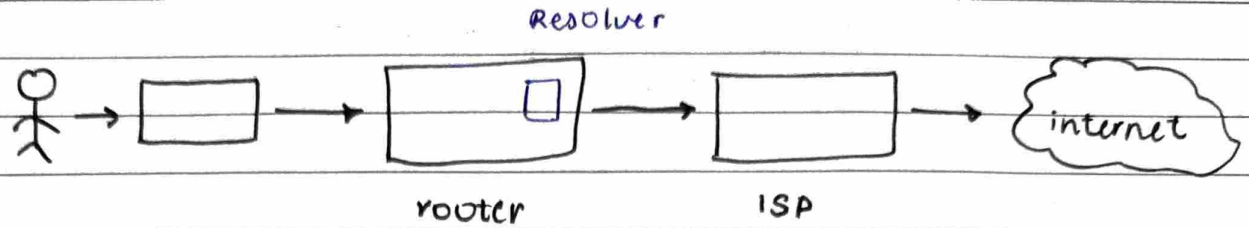



- DNS Resolver.

DNS resolver is a server that carrier out the resolution of Domain Name to IP address

* typically runs at isp. but you can run your own locally.
This can be your internet router

* most home routers are real DNS resolvers



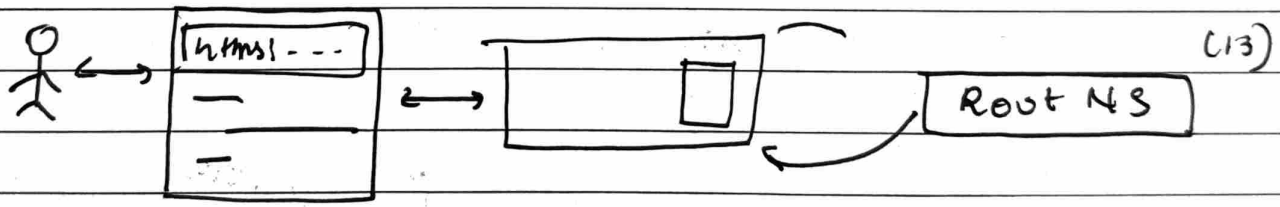

$ ipconfig/all ← from your windows CMD

Two popular DNS resolvers are
Google : 8.8.8.8 Cloudflare : 1.1.1.1

* SO, what exactly happens during resolution?

* How DNS resolution works?

Say, we are looking for www.google.com
→ we need to reach it authouritative Name server
But we do not know where it is !!




(13)

## Root Nameservers :

Total 13 root NS in the world, fixed IP address

verisign ← a.root-servr.net
USC ← b.root-servers.net
⋮
m.root-su.net

* this data does not mean there are 13 servers
Root servers from single operator are distributed across the world and leverage anycast (ex Cadvertising same IP)

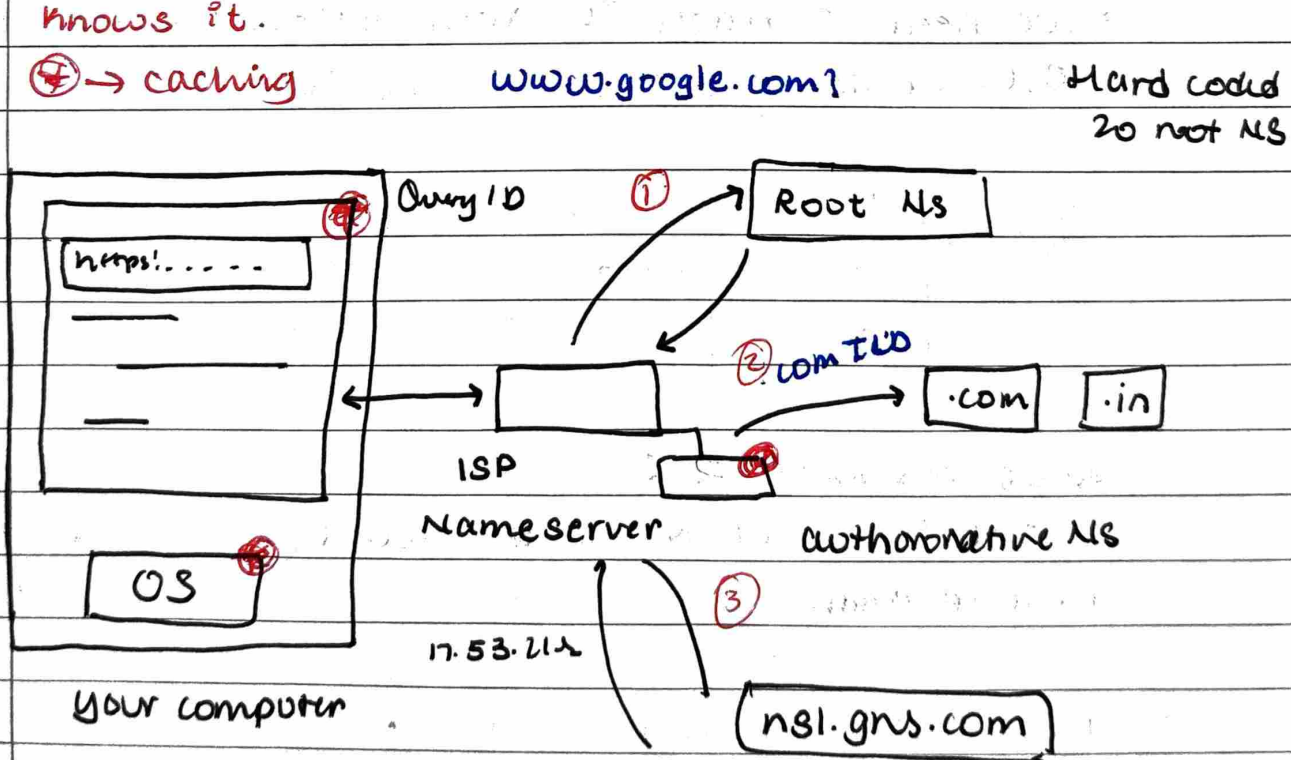
This way, incoming request connect to nearest Root Name server

when queried for domain name, it responds with IP address of some TDL (e.g .com, .in, .edu, .ch)

* calls to Root NS are infrequent, because even IP of TLD server does not changes often, so is heavily cached.
(you might be thinking everytime we do this google.com this all process happens any, no!)

* each machine takes us closer to machine that knows it.

→ caching




* cached for a few hours

__/__/____

This recursion continues from Google Name server to Authoritative Name server of domain zone google.com } where zone are served

then it checks the record against 'www', which may point to IB DNS and it then resolves to IP of load balancer.

Then the browser connects to the machine & finds the request.