

HTTP(S) explained

A ~~somewhat~~ gentle introduction



Our modern world runs on the
Hypertext **T**ransfer **P**rotocol,
but how does it really work?

Why is **HTTPS** a thing?

Where do "proxies"
and "load balancers"
come into the picture?

After this presentation,
you should feel comfortable
answering these questions! :-D

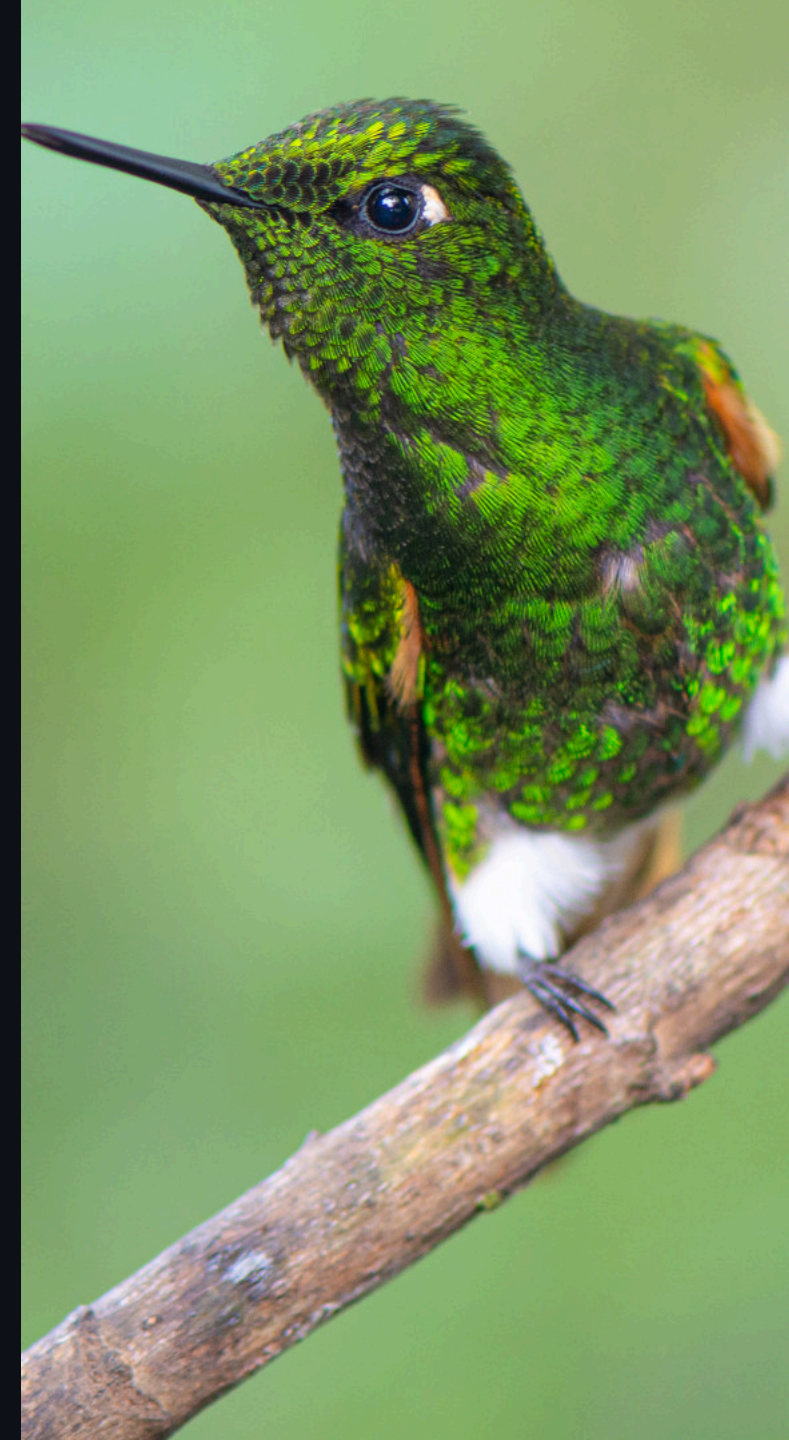


Brief history

(Relatively) simple protocol for client (*AKA "user agent"*) to server communication.

Introduced together with HTML in 1989 to serve files over the network.

Three major protocol versions exist, with the latest being release in 2022.



The early days

Basic serving of static files.

A client request for

<http://example.com/animals/horse.html>

would simply load the contents of

`/var/www/html/animals/horse.html`

from the server's file system

and transfer it to the client.



Things getting fancier

People wanted to use the web to provide interactive applications, such as online shopping malls.

Lowered the bar for adoption significantly, as users didn't have to install/update additional software on their computers.

Instead of just serving static files from disk, the server would generate dynamic responses on-the-fly.



A client request for

<http://example.com/weather.cgi?city=Gnarp>

may result in the following response being generated and returned by the server:

```
<html>
  <head>
    <title>Weather now in Gnarp</title>
  </head>
  <body>
    <p>
      The current (19:06) temperature
      in <b>Gnarp</b> is
      19 degrees celsius.<br>
      It is raining! :-(
    </p>
  </body>
</html>
```



Becoming Esperanto

These days HTTP isn't only used to serve HTML data to web browsers, but for a wide variety of client-server communication needs.

Liked by developers for its simplicity and widespread support in programming languages/toolkits.



If it's so damn simple,
can't you just get to it?!

Waow, chill - I shall!

Just one more thing...



Defining URLs

Applications are typically given **Uniform Resource Locators** to know where they should send requests.

<http://www.example.com/cocktails.txt>

tells the client to use the HTTP protocol, connect to the host address "www.example.com" and request the server path "/cocktails.txt".



Not only for HTTP

irc://chat.example.com/the_corner_bar

tells the client to use the

Internet **R**elay **C**hat protocol,

connect to the host address "chat.example.com"

and join a chat room named "the_corner_bar".

Not so complicated, right?



[http://bob:s3cret@t.example.com:1234 ↓
/about+us/faq?lan=en&s=Q%26A#q:Refund](http://bob:s3cret@t.example.com:1234 ↓ /about+us/faq?lan=en&s=Q%26A#q:Refund)



Breaking down a URL

http://bob:s3cret@t.example.com:1234 ↴
/about+us/faq?lan=en&s=Q%26A#q:Refund

Protocol, also known as "scheme".

Commonly "http" or "https".



Breaking down a URL

`http://bob:s3cret@t.example.com:1234 ↴`
`/about+us/faq?lan=en&s=Q%26A#q:Refund`

Optional username and password for authentication, separated by colon.

May be omitted and not considered best-practice.



Breaking down a URL

`http://bob:s3cret@t.example.com:1234 ↴
/about+us/faq?lan=en&s=Q%26A#q:Refund`

Target server network address.

Either host name, commonly resolved
by client using DNS, or IP address.



Breaking down a URL

`http://bob:s3cret@t.example.com:1234 ↓`
`/about+us/faq?lan=en&s=Q%26A#q:Refund`

Target port for connection to server.
If omitted, the default port is used:

HTTP version 1 and 2: 80/TCP
HTTPS: 443/TCP
HTTP version 3: 443/UDP



Breaking down a URL

`http://bob:s3cret@t.example.com:1234 ↴`
`/about+us/faq?lan=en&s=Q%26A#q:Refund`

Data path that client should request*
from the server.

The plus character is converted to space.
Other characters with special meaning in
URL path may be "percentage encoded":

`%20` = Space, `%2F` = /, `%26` = &, `%25` = %...



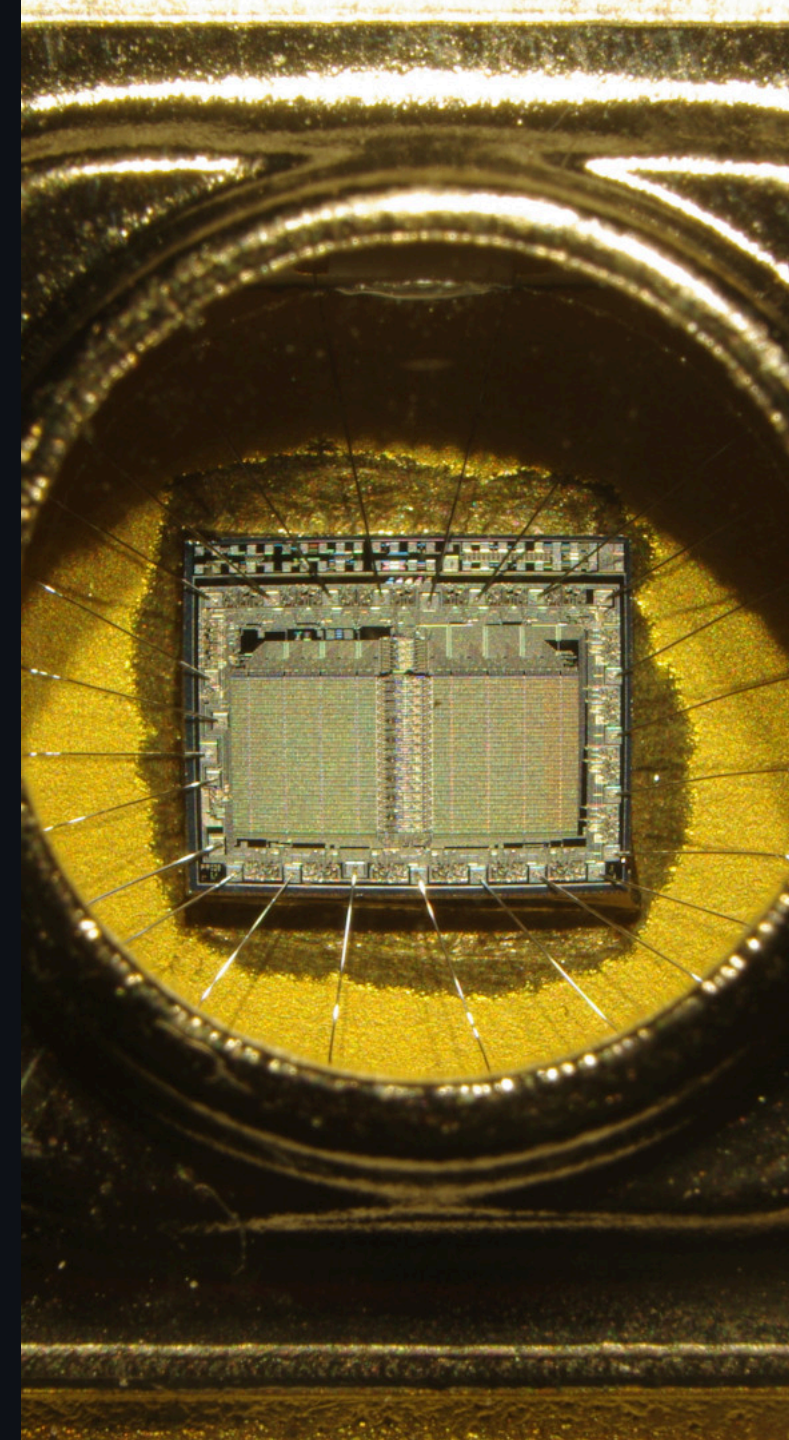
Breaking down a URL path

`http://bob:s3cret@t.example.com:1234 ↴`
`/about+us/faq?lan=en&s=Q%26A#q:Refund`

Base path.

Similar to a file system path.

Doesn't require file extension,
like ".html" or ".jpeg", other methods
exist for communicating response format.



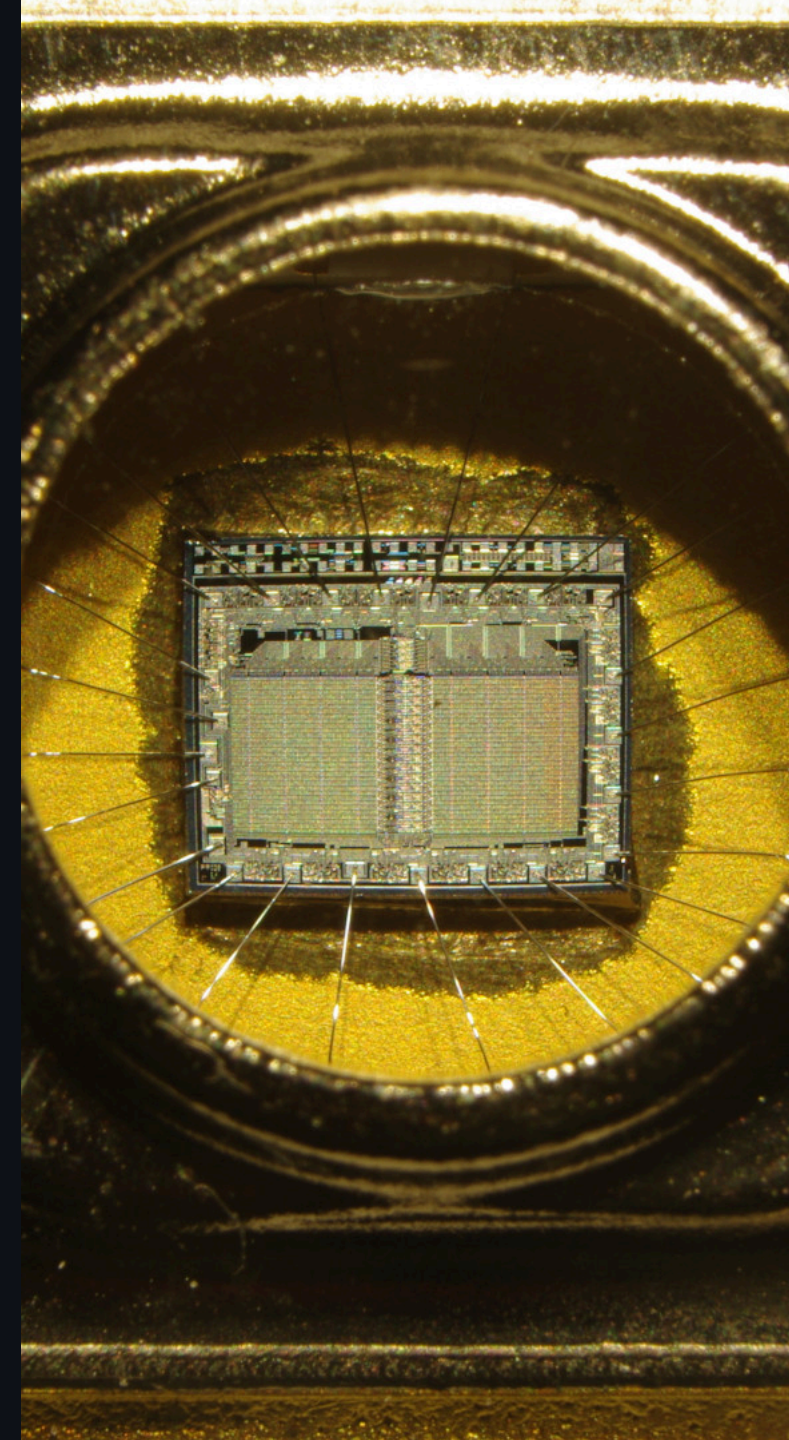
Breaking down a URL path

`http://bob:s3cret@t.example.com:1234 ↴
/about+us/faq?lan=en&s=Q%26A#q:Refund`

Optional "query string".

Key-value pairs, separated by
ampersand (or less commonly semicolon).

Commonly used to pass data to server
as input for generation of
dynamic responses.



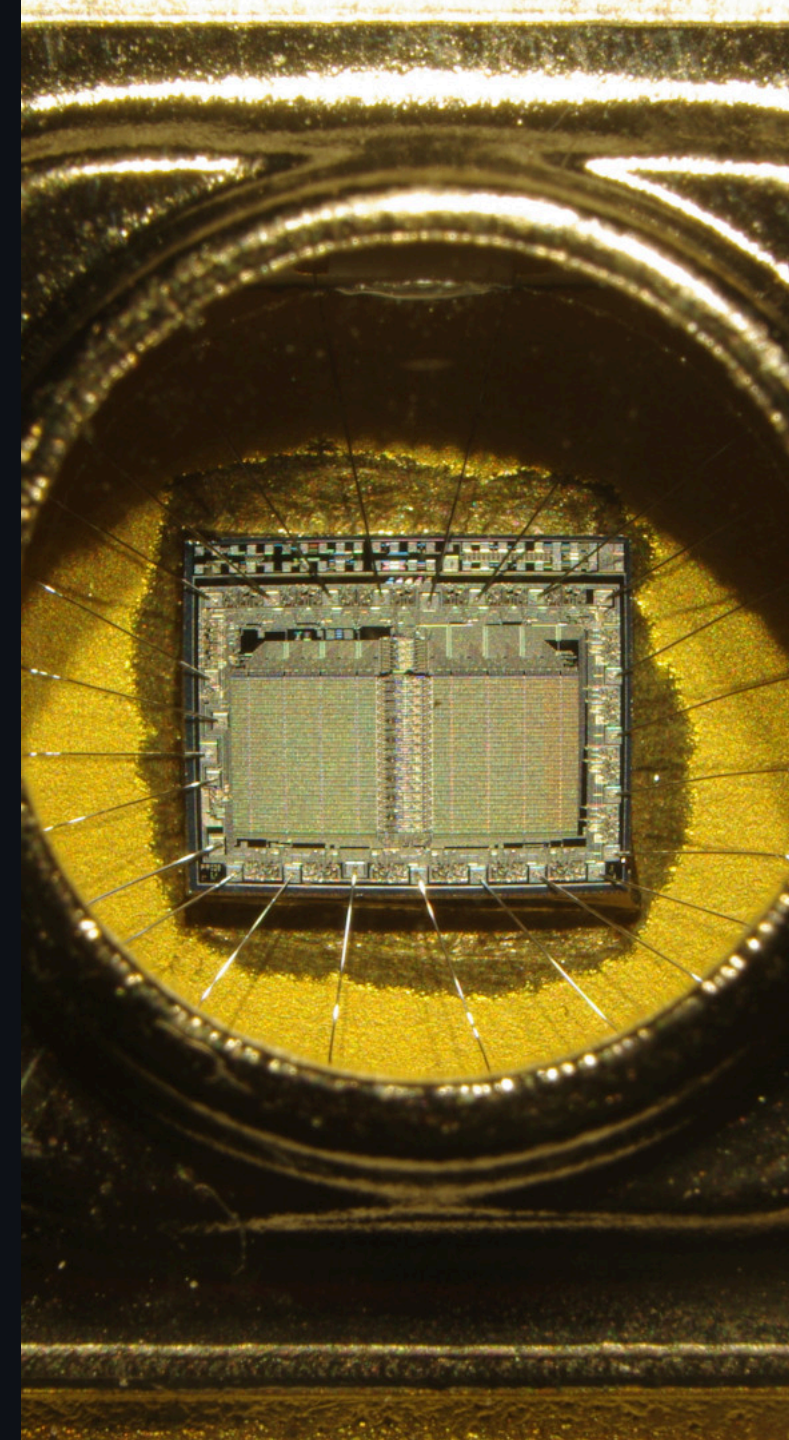
Breaking down a URL path

`http://bob:s3cret@t.example.com:1234 ↓`
`/about+us/faq?lan=en&s=Q%26A#q:Refund`

Optional "fragment".

Part of the URL that is never actually in requests to the server, but may be interpreted by the client application.

Commonly used for high-lighting text, passing client-side secrets, etc.



URL parsing woes

Properly (and consistently) grokking URLs seems tricky for both humans and computers.

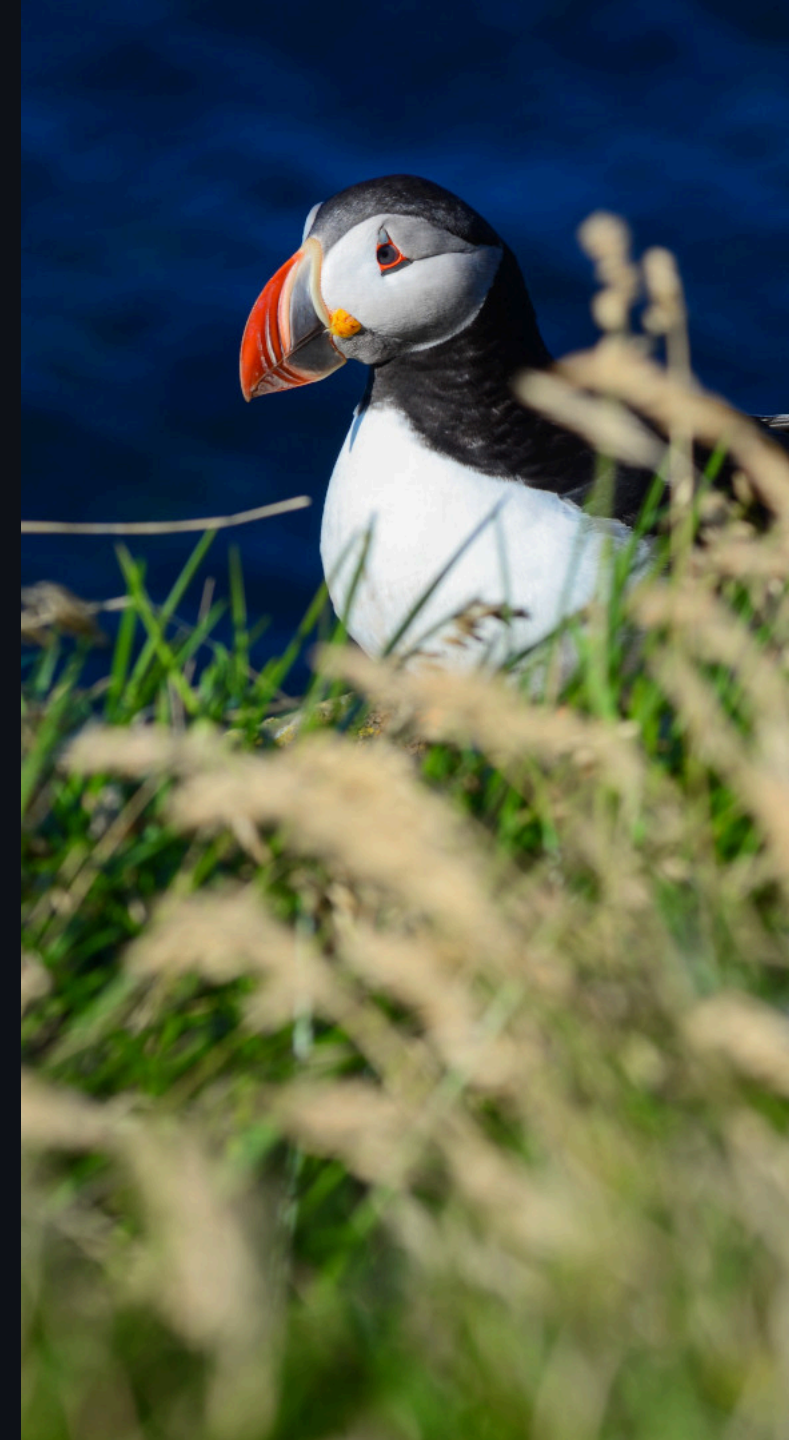
Where will we end up using

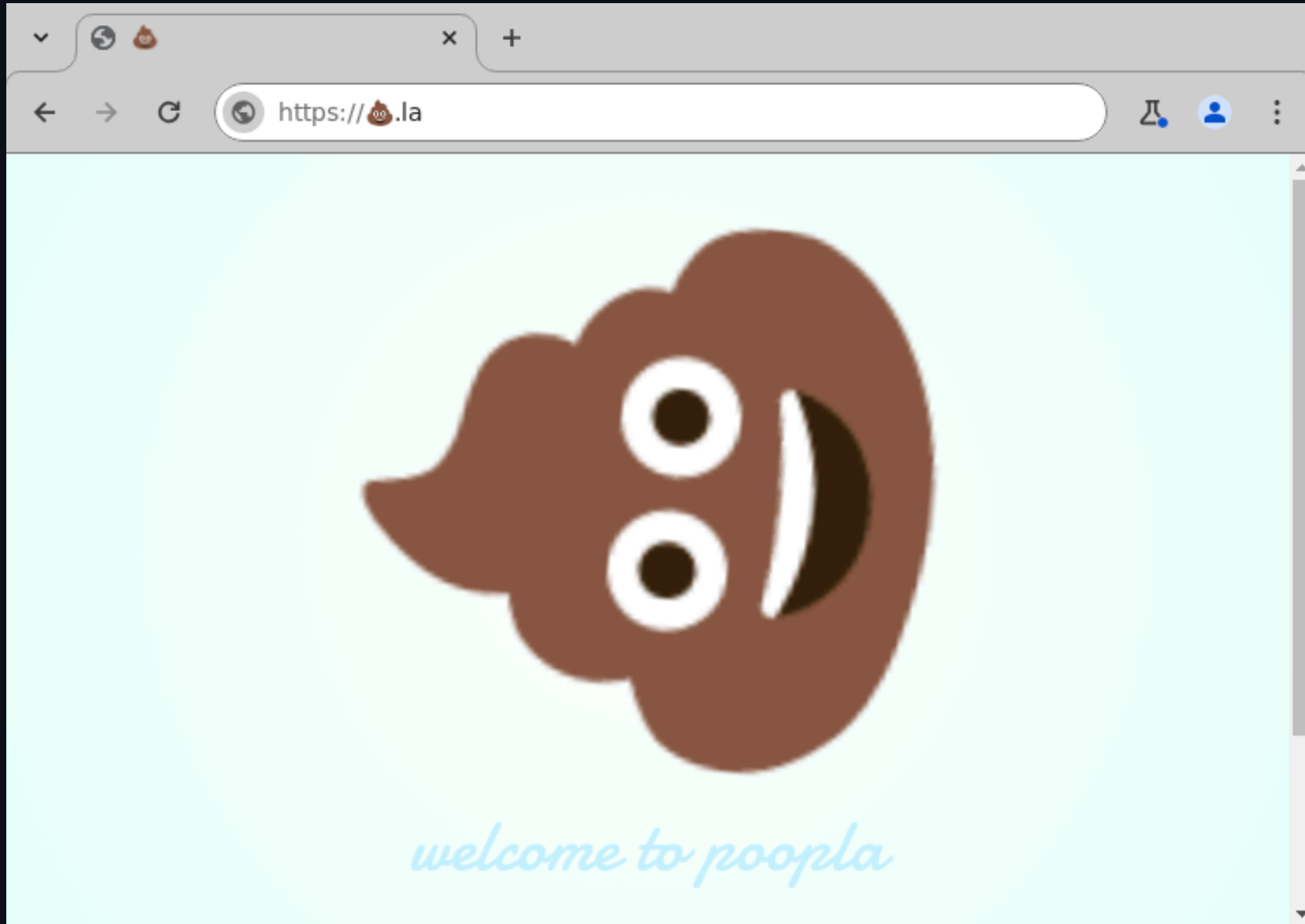
`http://chat.fb.com:1814/start.php@test.io` ,

`http://google.com` or

`http://facebook.com` ?

Loosely defined/interpreted standards have resulted in many security issues.





With that out of the way,
let's examine the
HTTP protocol!



The basics

HTTP version 1 is a text-based protocol.

Makes it simple to learn, debug and implement.

A **request** is sent by the client, resulting in a **response** being returned by the server.



HTTP v1.1 request

```
<METHOD> <PATH> HTTP/1.1  
Host: <TARGET HOST NAME OR IP ADDRESS>  
<OPTIONAL HEADER NAME>: <HEADER VALUE>  
  
<OPTIONAL BODY>
```



Very basic request

```
GET /cocktails.txt HTTP/1.1  
Host: www.example.com
```



Another simple example

```
DELETE /api/user/42 HTTP/1.1  
Host: management.example.com  
Authorization: Basic Ym9iOnMzY3JldA==
```

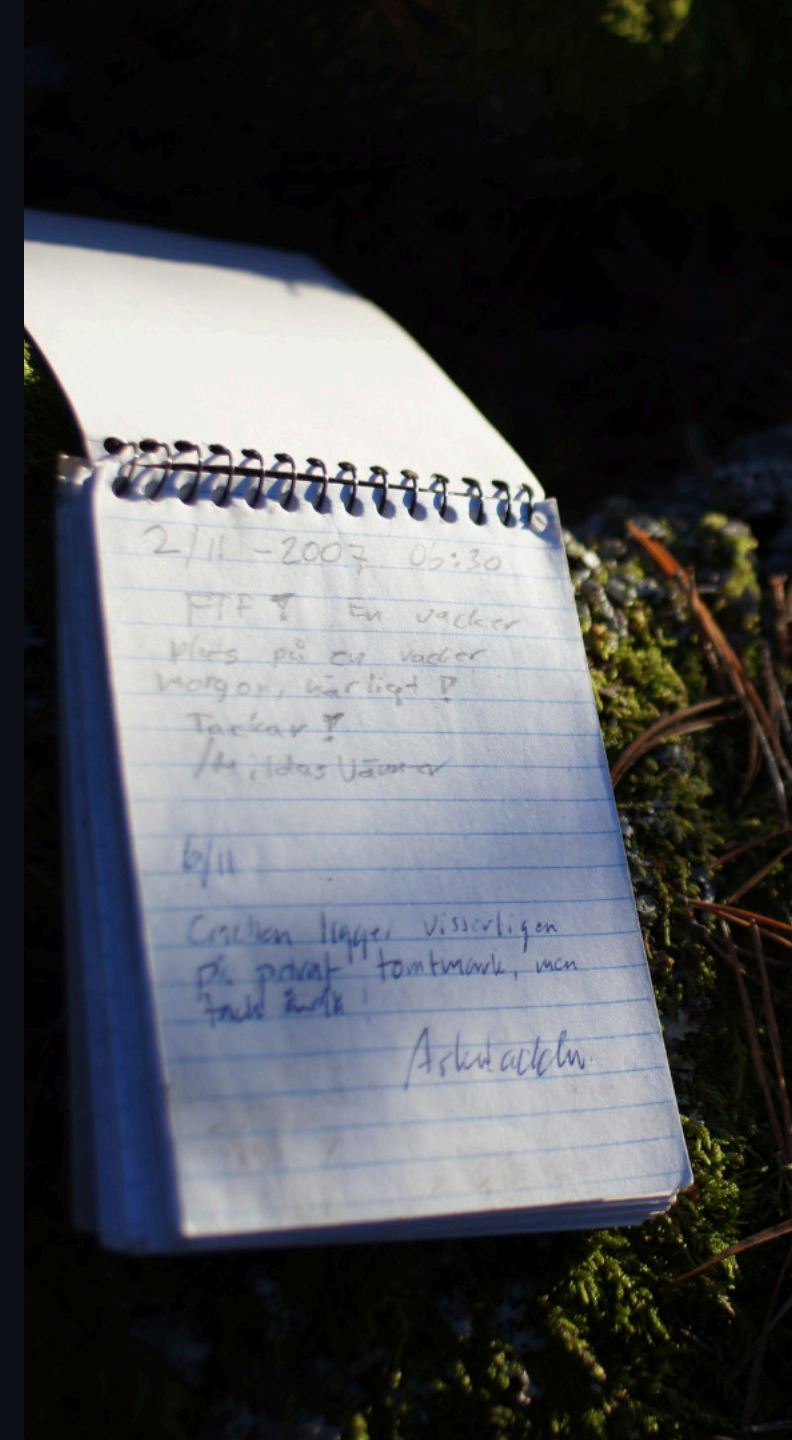
*Ym9iOnMzY3JldA== is
"bob:s3cret" encoded
using Base64.*



Including data in body

```
POST /guest_book.php HTTP/1.1
Host: social.example.com
Content-Type: application/json
Content-Length: 51
```

```
{
  "author": "adam",
  "message": "Hello Eve!!!"
}
```



HTTP v1.1 response

```
HTTP/1.1 <STATUS CODE> <STATUS MESSAGE>  
<OPTIONAL HEADER NAME>: <HEADER VALUE>  
  
<OPTIONAL BODY>
```



Very basic response

```
HTTP/1.1 204
```



Status code categories

- Informational (100 – 199)
- Successful (200 – 299)
- Redirection (300 – 399)
- Client error (400 – 499)
- Server error (500 – 599)



Common status codes

- **200**: Informational: OK
- **204**: Informational: No content
- **301**: Redirection: Moved permanently
- **400**: Client error: Bad request
- **401**: Client error: Unauthorized
- **404**: Client error: Not found
- **500**: Server error: Internal server error
- **503**: Server error: Bad gateway



...and of course **418**:

The HTTP 418 ("I'm a teapot") status response code indicates that the server refuses to brew coffee because it is, permanently, a teapot.

— *MDN web docs*



Another simple example

```
HTTP/1.1 500 Wooops  
X-Server: Example HTTPD v0.2
```

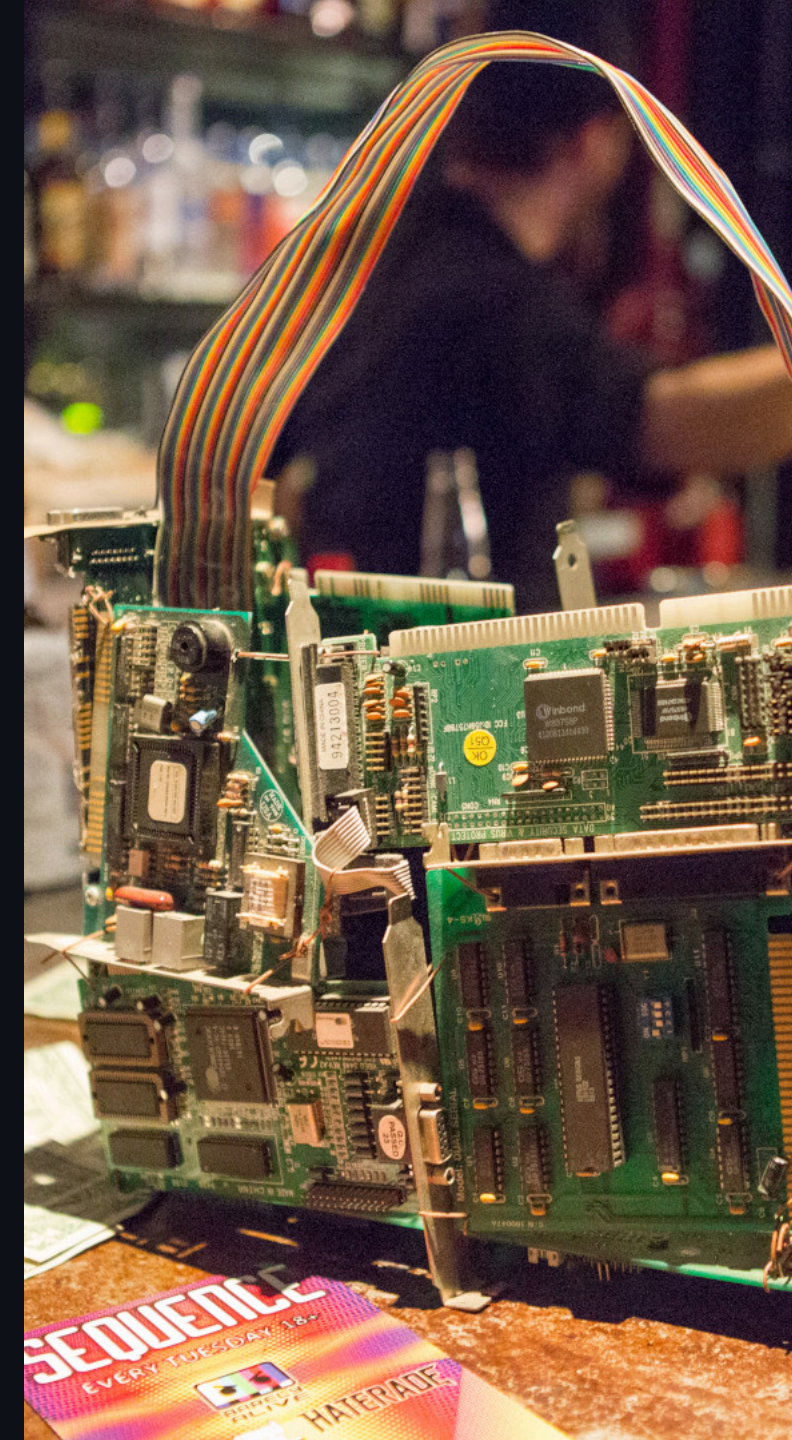


Including data in body

```
HTTP/1.1 200 OK  
Content-Type: text/plain  
Content-Length: 67
```

Top three cocktails:

1. Caipirinha
2. White Russian
3. Bloody Mary



Doesn't seem too tricky.
Let's hack together our own
client and server using
Netcat!



The S in HTTPS

HTTP is a "clear-text" protocol.

Communication can be intercepted (and modified) anywhere between the server and the client.

HTTPS was created to wrap HTTP in a layer of encryption.

Relies on both symmetric and asymmetric cryptography.

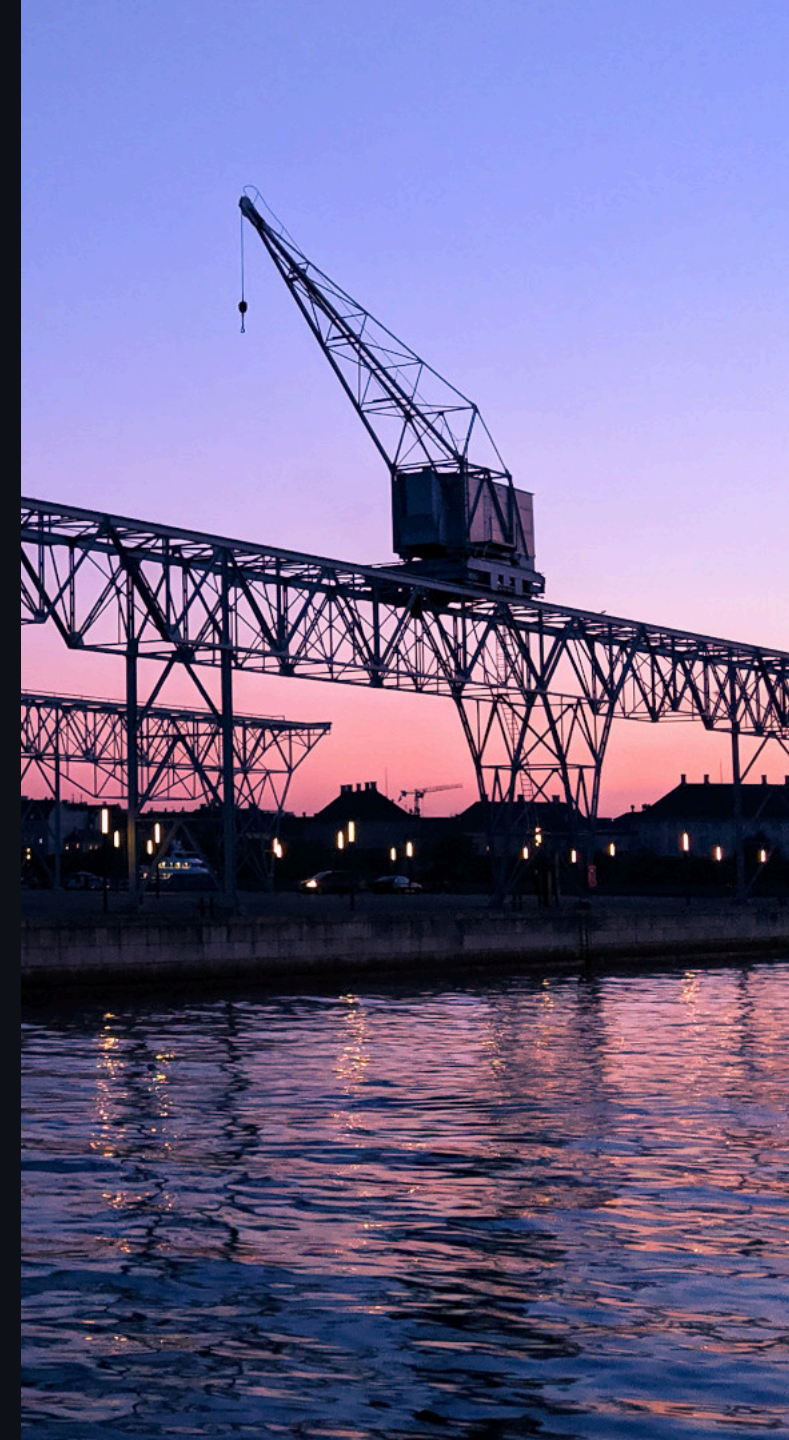
Let's jump into Menacit's
["Practical cryptography course"](#)!



HTTP proxies

An HTTP proxy is a piece of software acting as both a server and a client at the same time.

Can be used to filter, redirect and manipulate HTTP requests from clients.



Forward proxies

Commonly used to restrict egress communication on a network or provide some client anonymity.

A HTTP request reaches the forward proxy. If the host header contains "example.com", the proxy sends a HTTP request to "example.com" and returns its response to the client.

(may log requests, return status code 403 for disallowed host names and similar)



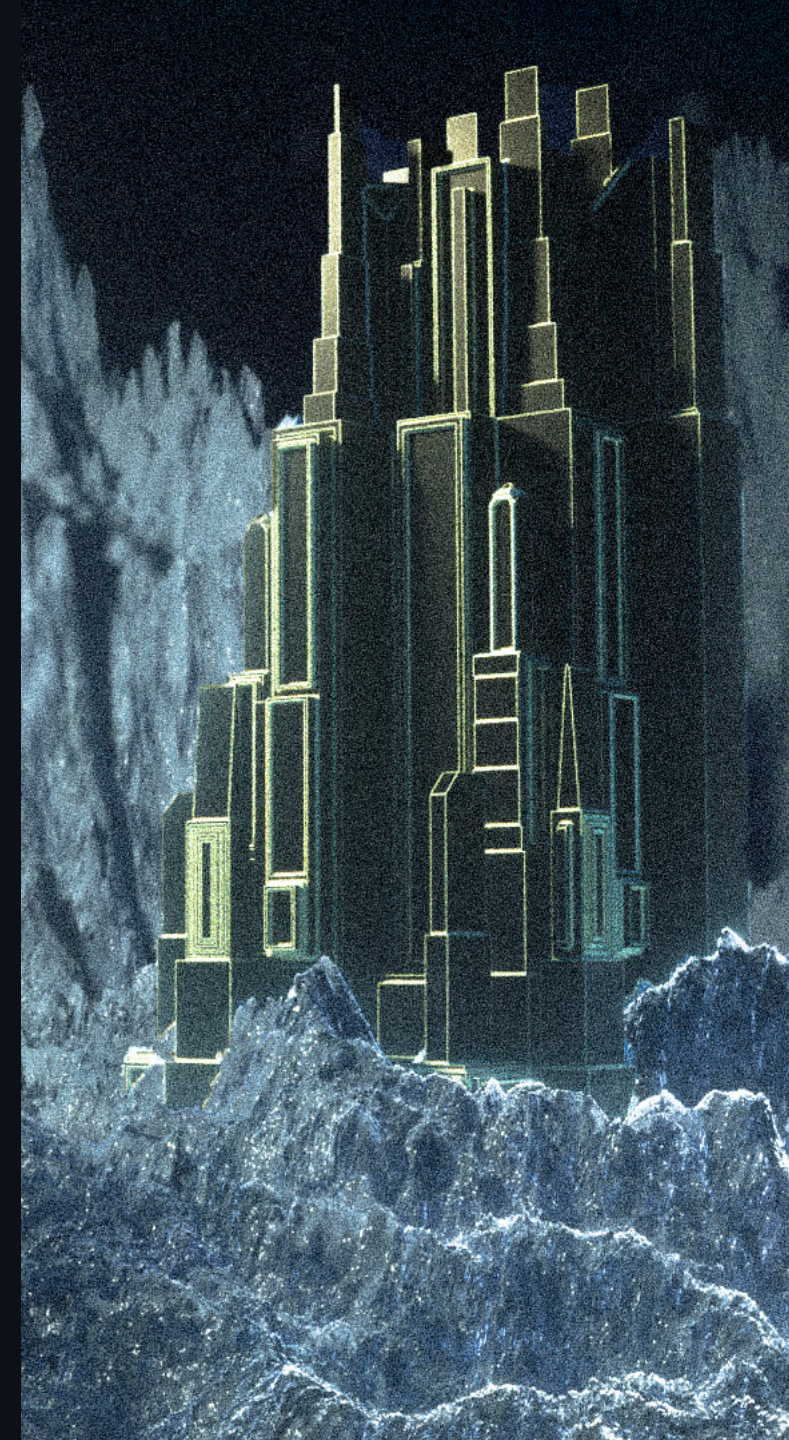
Reverse proxies

Commonly used to restrict or redirect client requests (ingress) to another HTTP server.

A HTTP request reaches the reverse proxy.

If the URL path begins with `"/contact"`, the reverse proxy sends a HTTP request to `"w1.int.example.com"` and returns its response to the client.

Otherwise, the reverse proxy sends a HTTP request to `"w2.int.example.com"` and returns its response to the client.

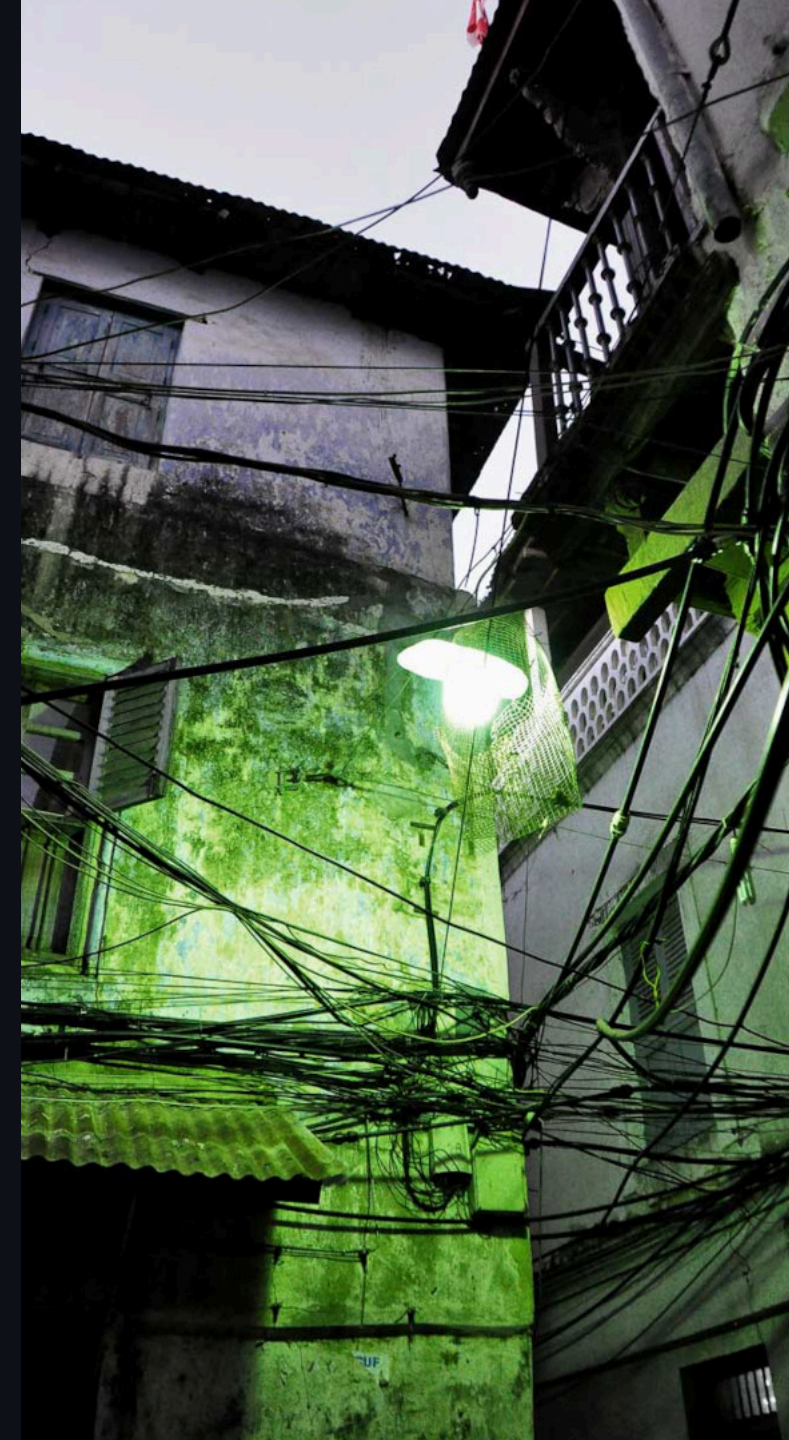


Load balancers

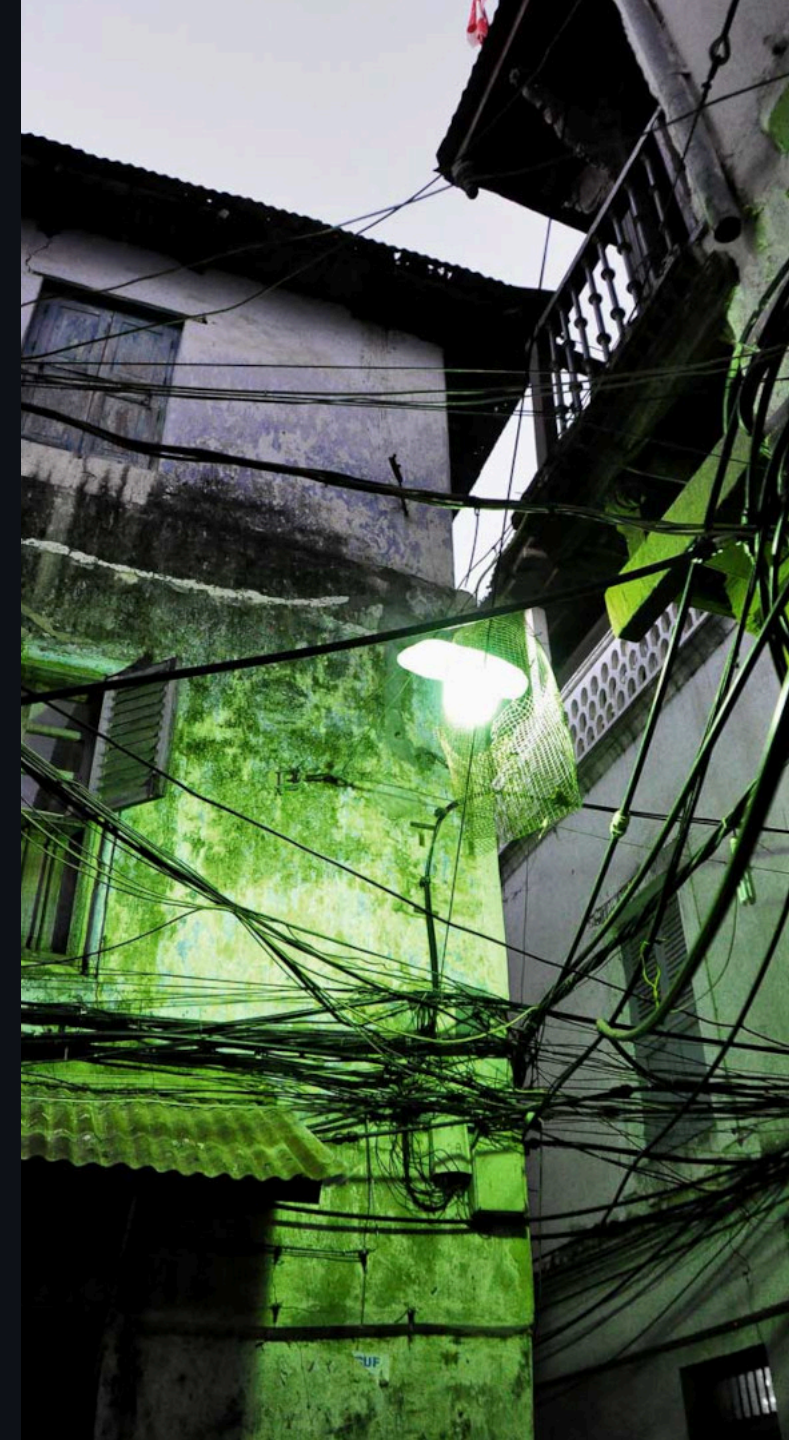
Forwards traffic to multiple servers, distributing the load.

Can monitor status of servers and exclude them as targets if they become unhealthy.

All* HTTP load balancers are reverse proxies, but not all reverse proxies are load balancers.



A HTTP request reaches the load balancer.
If the host header contains "example.com",
the load balancer sends a HTTP request to
either "w1.example.com" or
"w2.example.com"
(depending on their load/availability)
and returns its response to the client.



What happened after HTTP version 1.1?



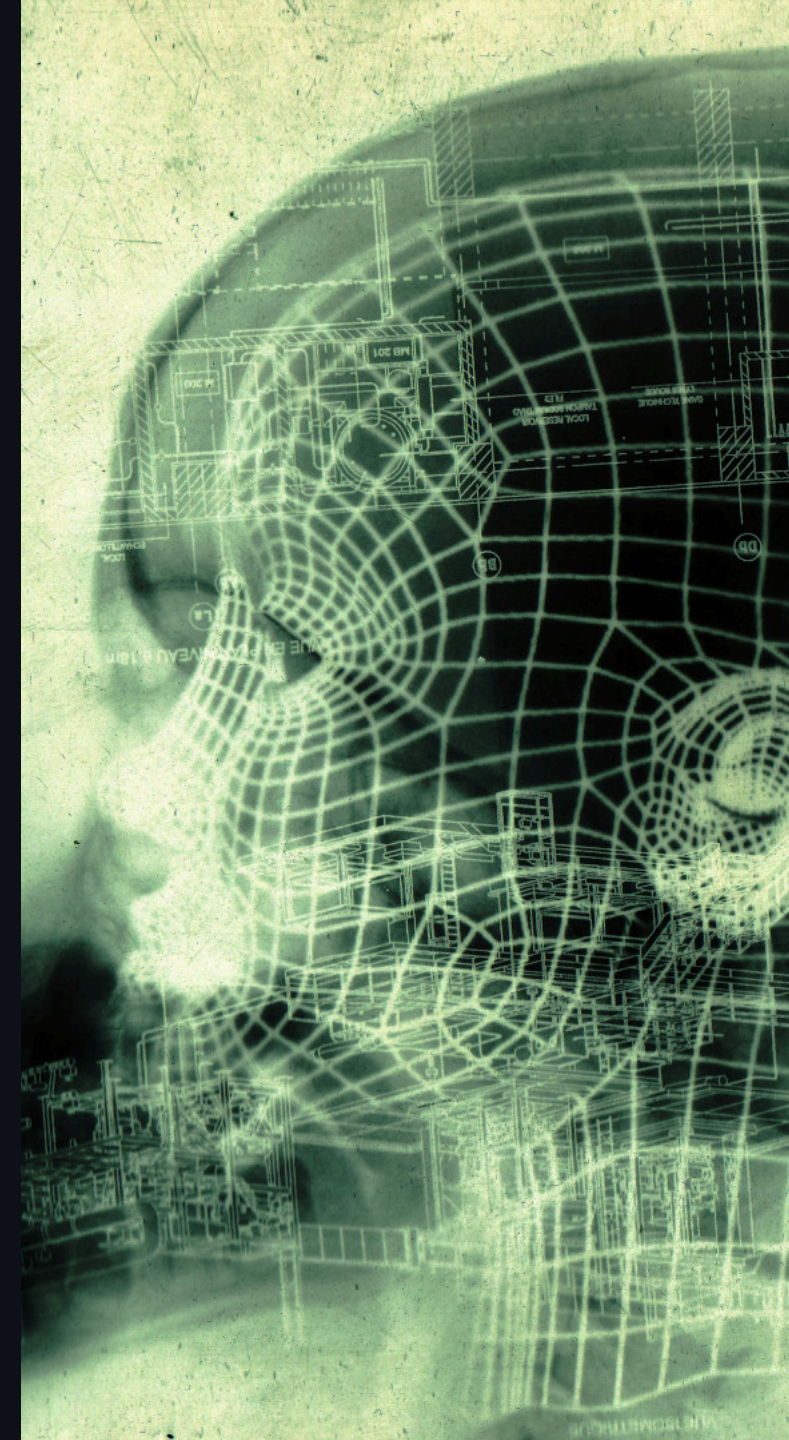
HTTP version 2

Introduced back in 2015,
first major change since 1997.

Still uses the same verbs, status codes,
header/body concepts - but no longer a
simple text based protocol.

Features like multi-plexing, server-side push
and header compression provides better
performance/lower latency.

Huge resource savings for
large web-site operators.



HTTP version 3

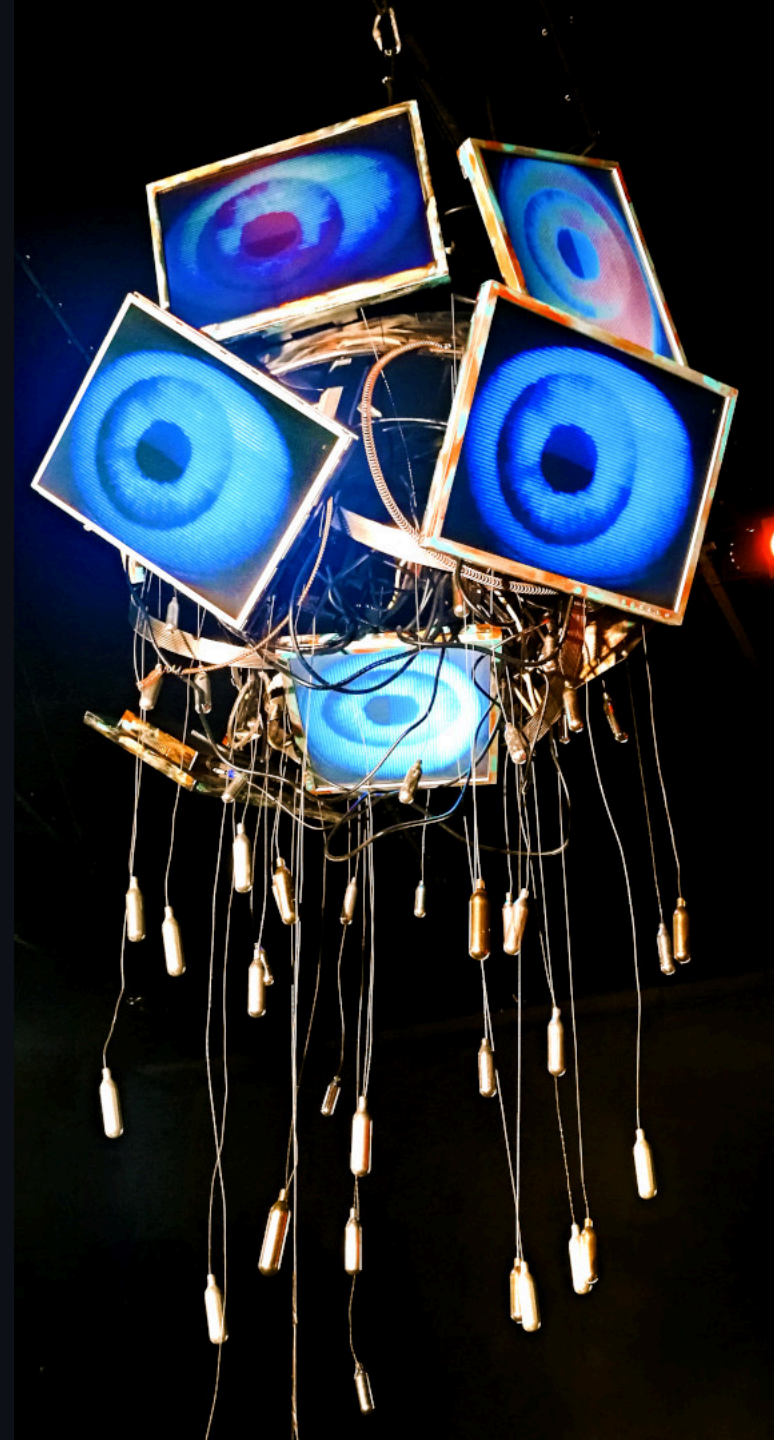
Standardized in 2022,
support still being implemented
in client/server/proxy software.

Abandons TCP in favor of the
UDP-based transport protocol "QUIC".

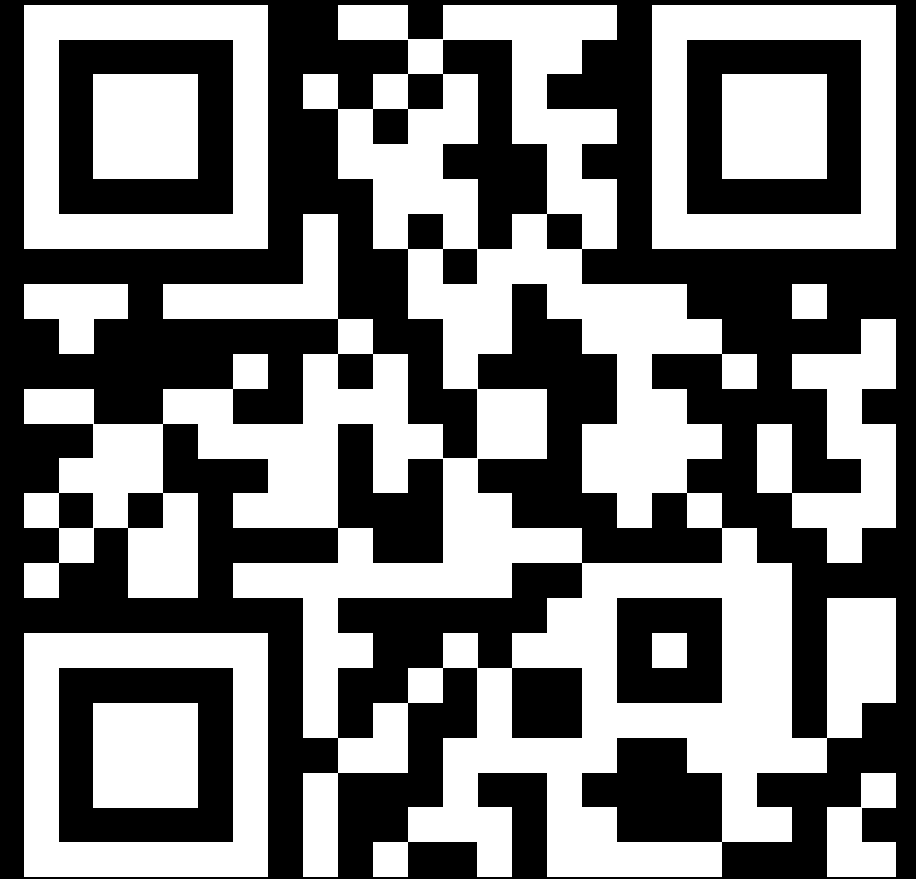
Mandatory TLS-like encryption and
further performance improvements.



We haven't yet talked about cookies,
WebSockets and other exciting things!
...but that's a story for another day.



For copy-pasteable
speaker notes, example code
and similar goodies, see:
t.menacit.se/http.zip.



Thanks for listening!

Was anything unclear?

Got ideas for improvements?

Don't fancy the animals in the slides?

Create an issue or submit a pull request to
[the repository on Github!](#)

