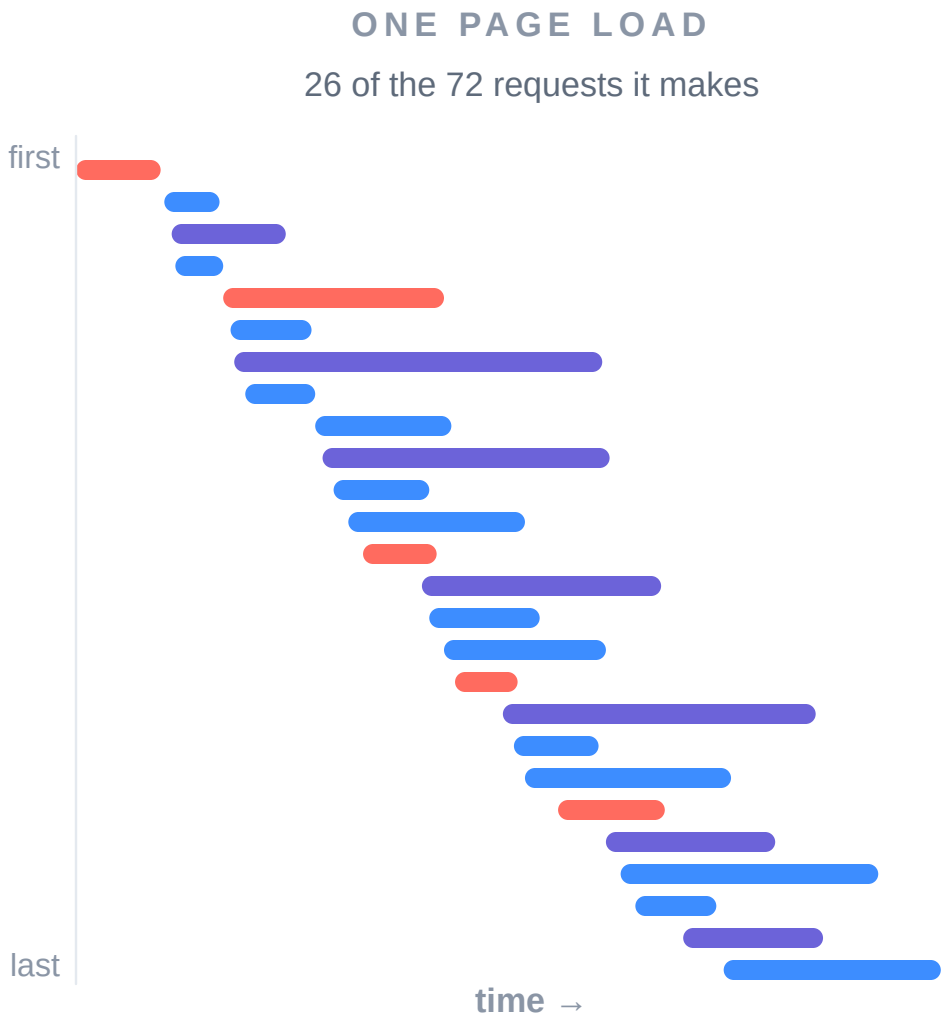
CSE589 · Applications

The Web and HTTP
Objects, requests, and the protocol that moves them

The web is where almost every application now runs, and HTTP is the protocol that carries it. This lecture covers why applications moved into the browser, what a web page is made of, and how HTTP fetches each of its files.

Yaxiong Xie



Why does the web get its own lecture?

01

Why the web

The web is where almost every application people use now runs. This part is about why that happened, and what the browser had to become for it to happen.

There are 1.49 billion websites

1,494,915,628

websites answering an HTTP request, July 2026

305,348,459	14,772,048	+5,500,000
unique domains	web-facing computers	sites added in one month

A site here is a hostname that answers an HTTP request. The number of separate machines behind them is far smaller, because one machine serves many hostnames.

Every one of these is reached with the same protocol.

About three-quarters of the world is online

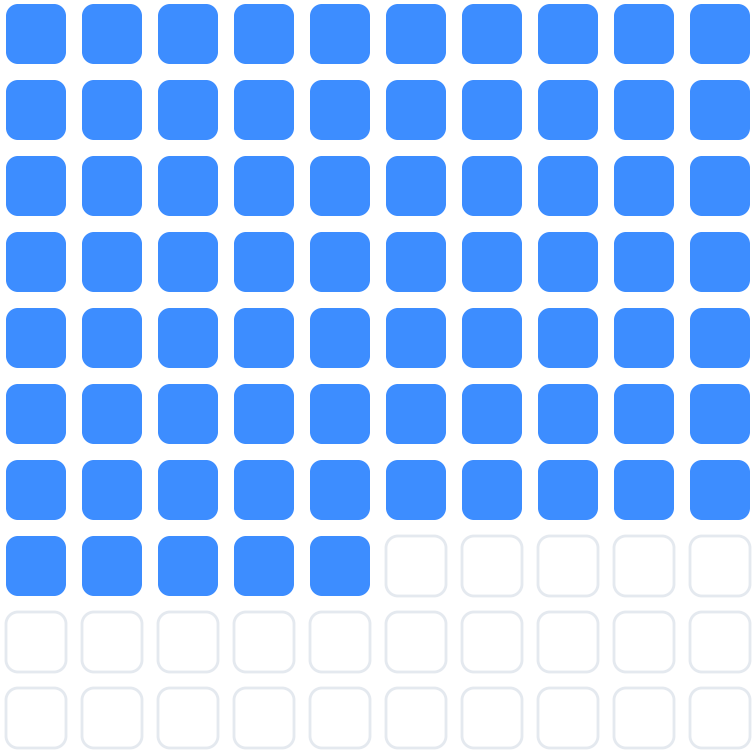
75%

of the world population is online

2.2 billion people are still offline.

ITU Facts and Figures, 2025.

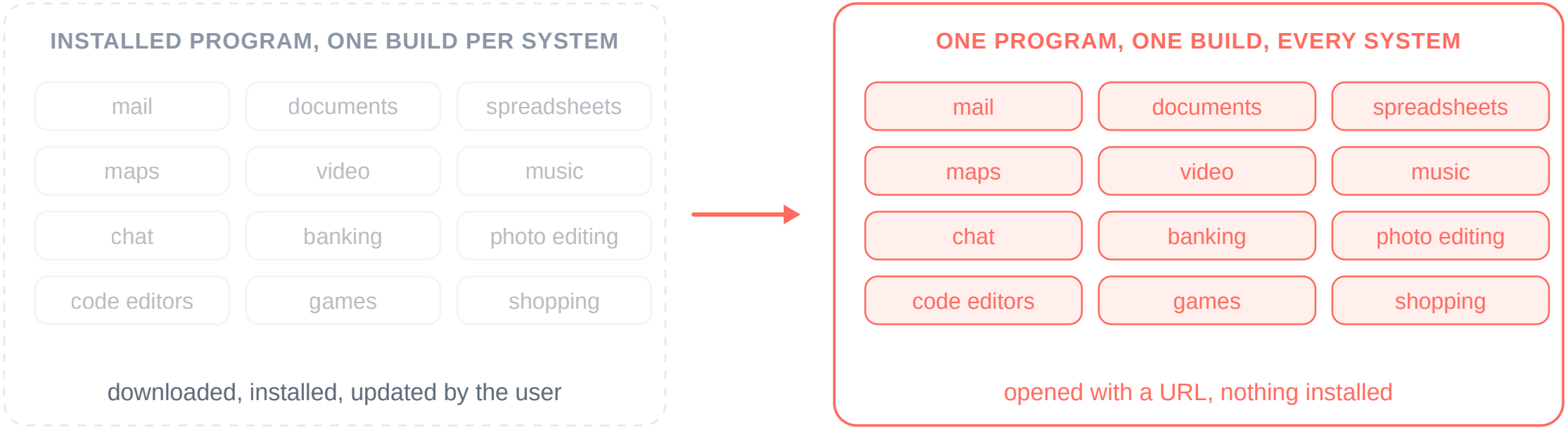
Nearly all of them reach it through a browser or an application that speaks the same protocol.



ONE SQUARE = 1% OF THE WORLD POPULATION

The web is the interface almost all of them use.

The browser runs applications that used to be installed



The browser is a runtime with a network stack, a layout engine and a JavaScript engine. An application that targets it is written once and runs on every operating system that has a browser.

The browser replaced the installer for most categories of software.

The browser is a runtime, not a document viewer

THE BROWSER

a network stack opens connections, sends and receives	**a layout engine** works out where every element goes	**a JavaScript engine** runs the code that came with the page
storage keeps files and databases on the disk	**graphics** draws through the GPU, including 3-D	**device access** camera, microphone, location, clipboard

These are the facilities an installed program gets from the operating system.

A page can use all of them, and gets them on every system at once.

This is the technical reason applications could move. A spreadsheet, a video editor or a game needs exactly these six things, and a page in a tab has all six.

Anything the browser can do, a page can do.

The program is fetched on every use, not installed once

INSTALLED PROGRAM

- build for each OS
- publish to a store
- user downloads
- user updates

old versions keep running until the user acts

WEB APPLICATION

- deploy to the server
- next request gets it

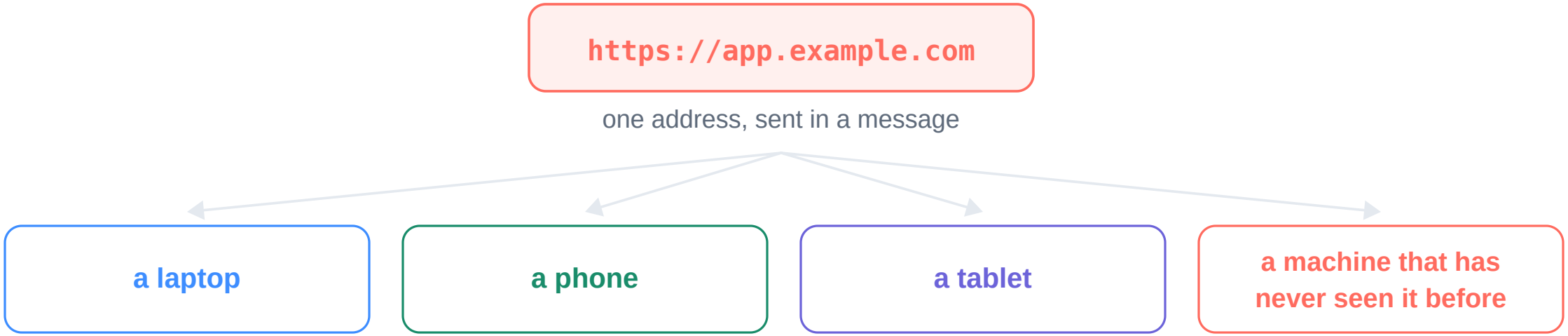
every user runs the version on the server

the code is part of the response, not a prior install

The HTML, CSS and JavaScript that make up the application are ordinary objects fetched over HTTP. Changing the copy on the server changes what every user runs on their next request.

Deployment is a file change on one machine.

Distributing a web application is sending a link



no store has to approve it · no version has to be chosen · nothing is installed first

The address is the whole distribution mechanism.

An installed program has to be packaged for each system, published, downloaded and installed before it runs once. A web application is reached by typing or clicking its address.

One address reaches every device that has a browser.

Where this goes

What the rest of this lecture answers

02 **What is a web page, and how is it built?** — HTML, objects, and how many of them

03 **How do those files reach the browser?** — HTTP requests, responses and status codes

04 **What does HTTP do beyond moving bytes?** — connections, caching, cookies

Part 2 starts with the page itself, before any protocol.

1 Why the web | 2 How a page works | 3 HTTP | 4 What HTTP adds

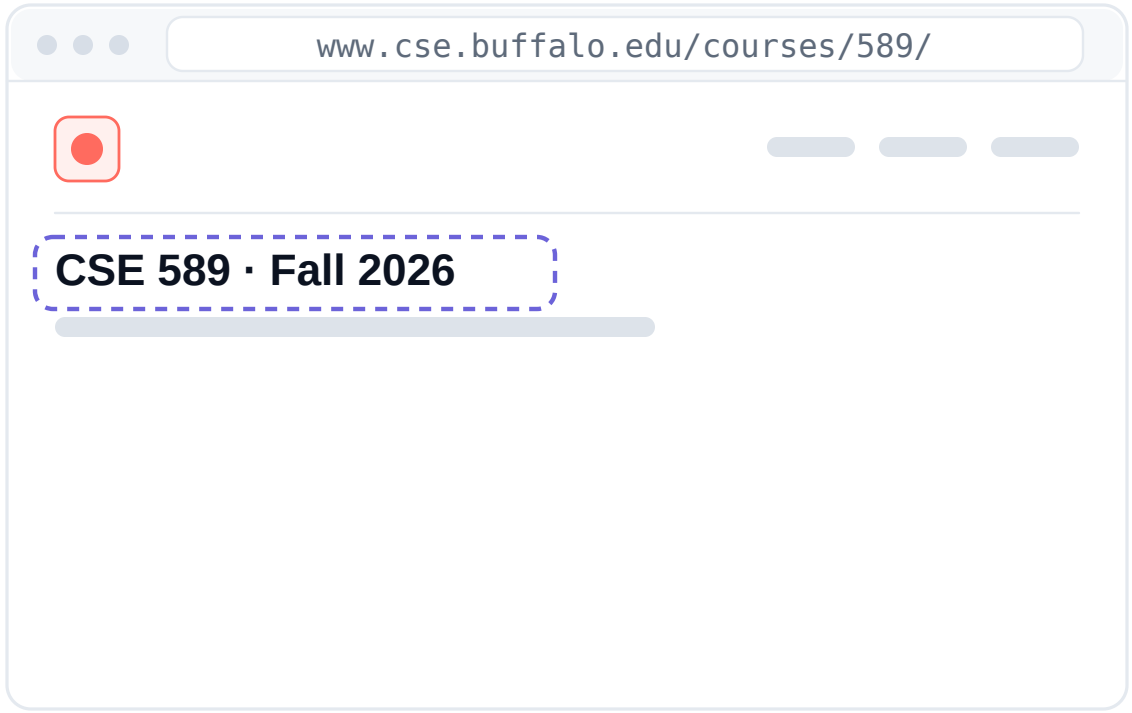
What is a web page, and how is it built?

02

How a page works

A page is a text file that the browser turns into what you see, plus a set of separate files it names. This part is the page itself, with no protocol in it yet.

The browser draws one thing for each tag it reads




```
<!doctype html>
<html><head>
<title>CSE 589</title>
</head><body>
<h1>CSE 589 · Fall 2026</h1>
<img src="/photo.jpg">
<p>Networks, from the top down.</p>
<a href="/hw1.html">Homework 1</a>
</body></html>
```

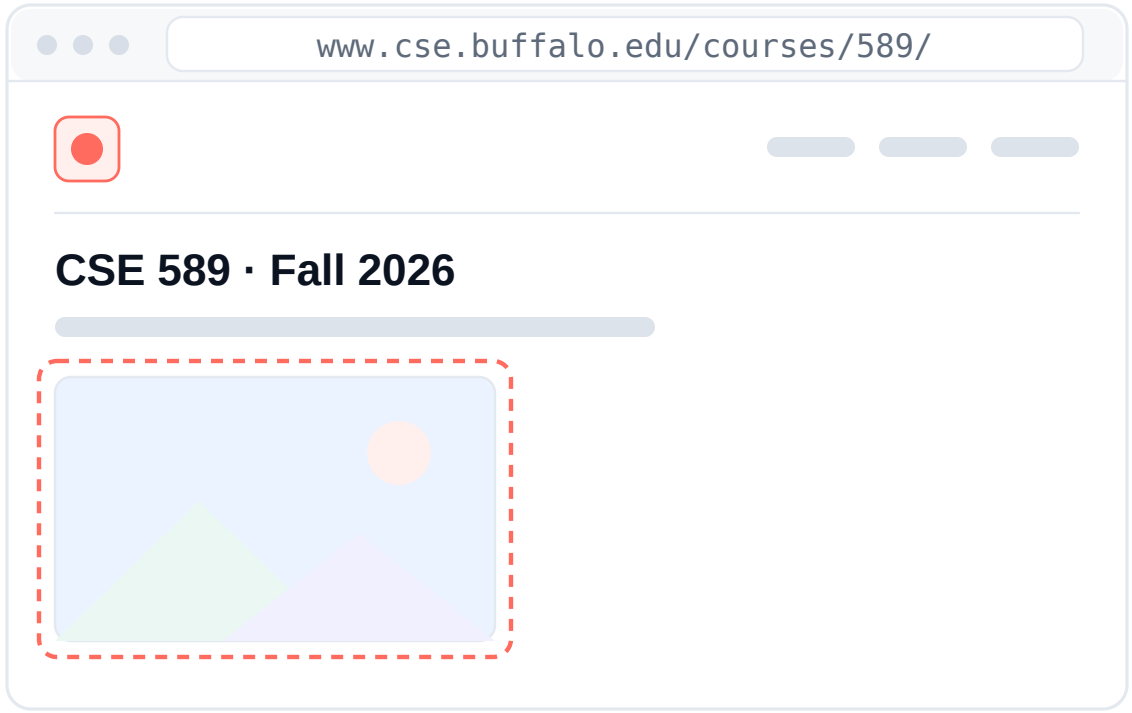
step 1 of 4

the browser has read the file down to this line

A tag marks a piece of content. **<h1>** marks a heading, so the browser draws that text large and bold, on a line of its own.

The file says what each piece is; the browser decides how it looks.

The browser draws one thing for each tag it reads




```
<!doctype html>
<html><head>
<title>CSE 589</title>
</head><body>
<h1>CSE 589 · Fall 2026</h1>
<img src="/photo.jpg">
<p>Networks, from the top down.</p>
<a href="/hw1.html">Homework 1</a>
</body></html>
```

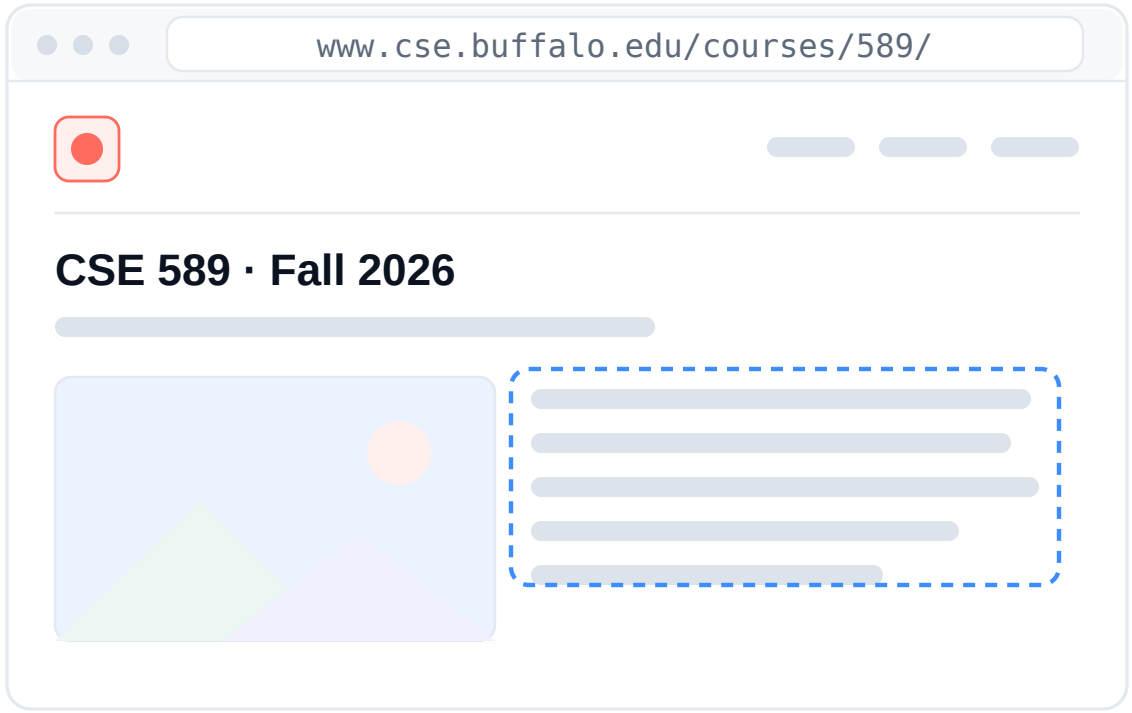
step 2 of 4

the browser has read the file down to this line

<img> does not contain a picture. It names a file. The browser fetches that file and draws it where the tag stood.

The file says what each piece is; the browser decides how it looks.

The browser draws one thing for each tag it reads




```
<!doctype html>
<html><head>
<title>CSE 589</title>
</head><body>
<h1>CSE 589 · Fall 2026</h1>
<img src="/photo.jpg">
<p>Networks, from the top down.</p>
<a href="/hw1.html">Homework 1</a>
</body></html>
```

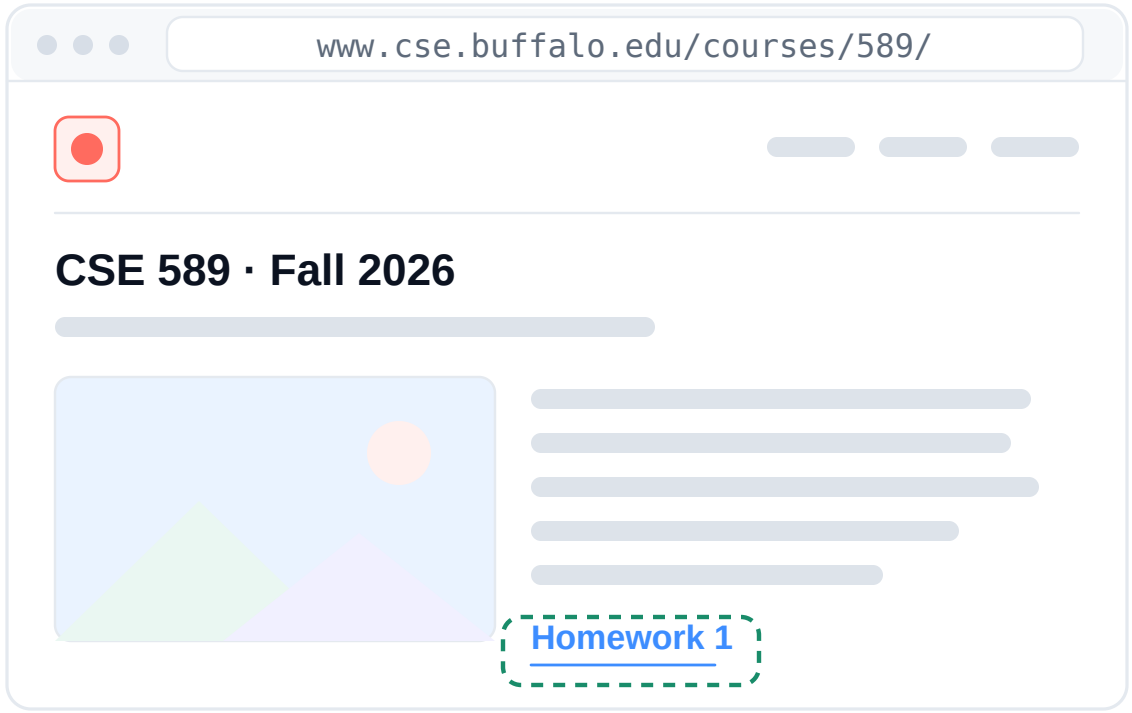
step 3 of 4

the browser has read the file down to this line

<p> marks a paragraph. The browser breaks the text into lines to fit the width it has, which is why the same file looks different in a narrow window.

The file says what each piece is; the browser decides how it looks.

The browser draws one thing for each tag it reads




```
<!doctype html>
<html><head>
<title>CSE 589</title>
</head><body>
<h1>CSE 589 · Fall 2026</h1>
<img src="/photo.jpg">
<p>Networks, from the top down.</p>
<a href="/hw1.html">Homework 1</a>
</body></html>
```

step 4 of 4

the browser has read the file down to this line

<a href="..."> marks a link. The text is drawn differently, and clicking it makes the browser load the URL in the href.

The file says what each piece is; the browser decides how it looks.

The browser builds the page in five steps

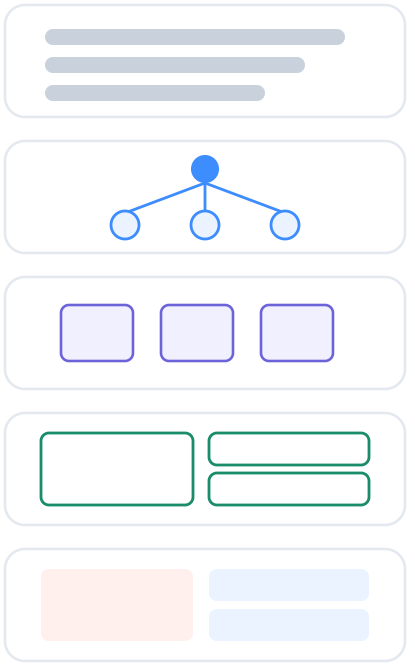
01 read the text
the file arrives as a stream of characters

02 build the tree
one node for every tag, nested as the tags are

03 fetch the files it names
the pictures, the style sheet, the script

04 compute the layout
a position and a size for every node

05 paint
turn that into pixels



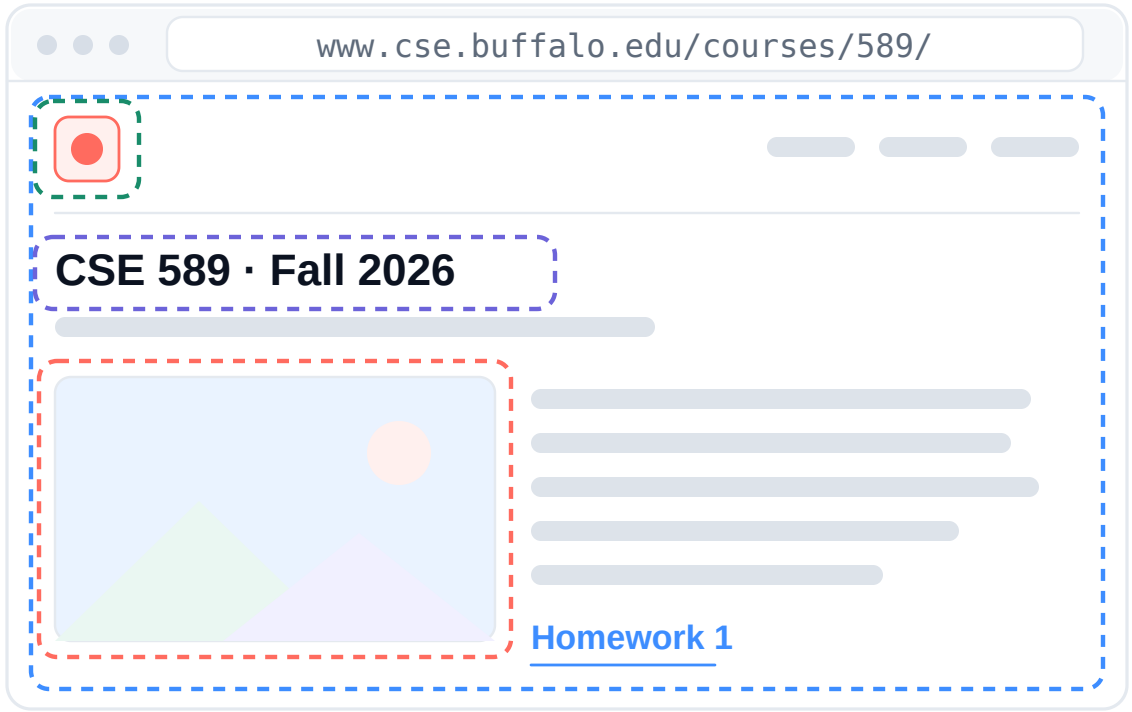
what each step produces

Steps 3 and 4 repeat as files arrive

The tree built in step 2 is the DOM. Layout, painting and any script that runs all work on that tree, not on the original text.

The page is computed by the browser, not sent as a picture.

The page on screen is six separate files



`index.html`	the text and every reference below	22 KB
`photo.jpg`	the picture	310 KB
`logo.png`	the mark at the top left	14 KB
`font.woff2`	the shapes of the letters	28 KB
`style.css`	the colours, spacing and layout	77 KB
`app.js`	the menu that opens when clicked	210 KB

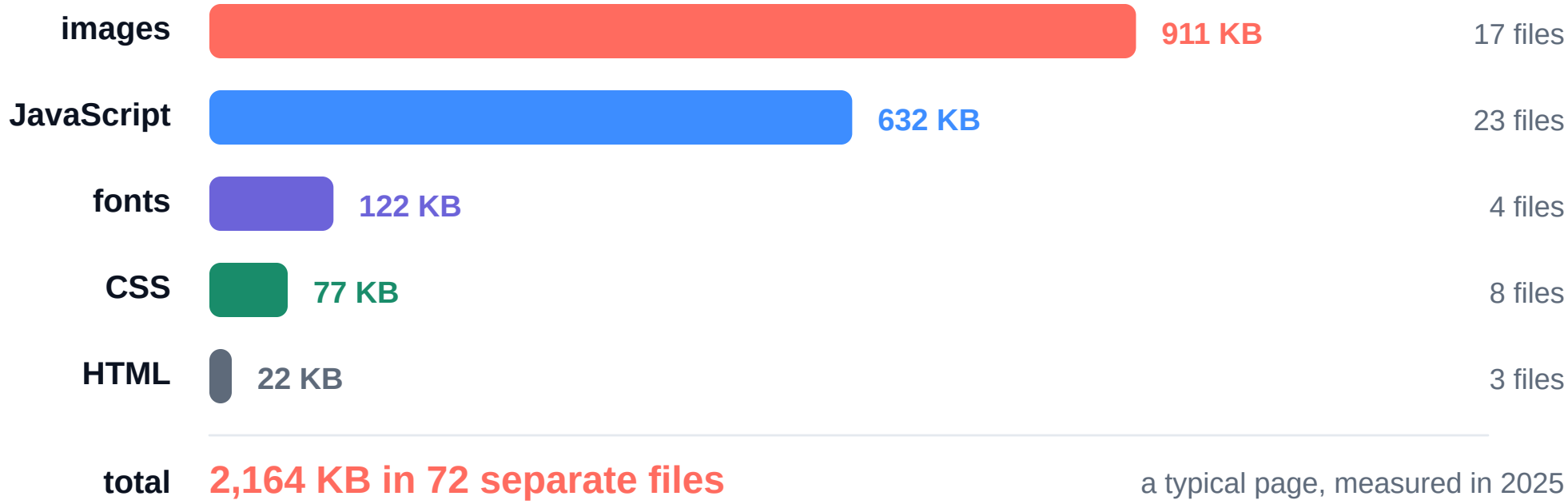
6 objects · 661 KB

and a real page carries far more than six

Only the first one is HTML. Two of the six have no region of their own: the style sheet sets how everything looks, and the script runs when the reader clicks something.

A page is not one file. It is a set of files, listed by the first one.

A typical page needs 72 files, not 6

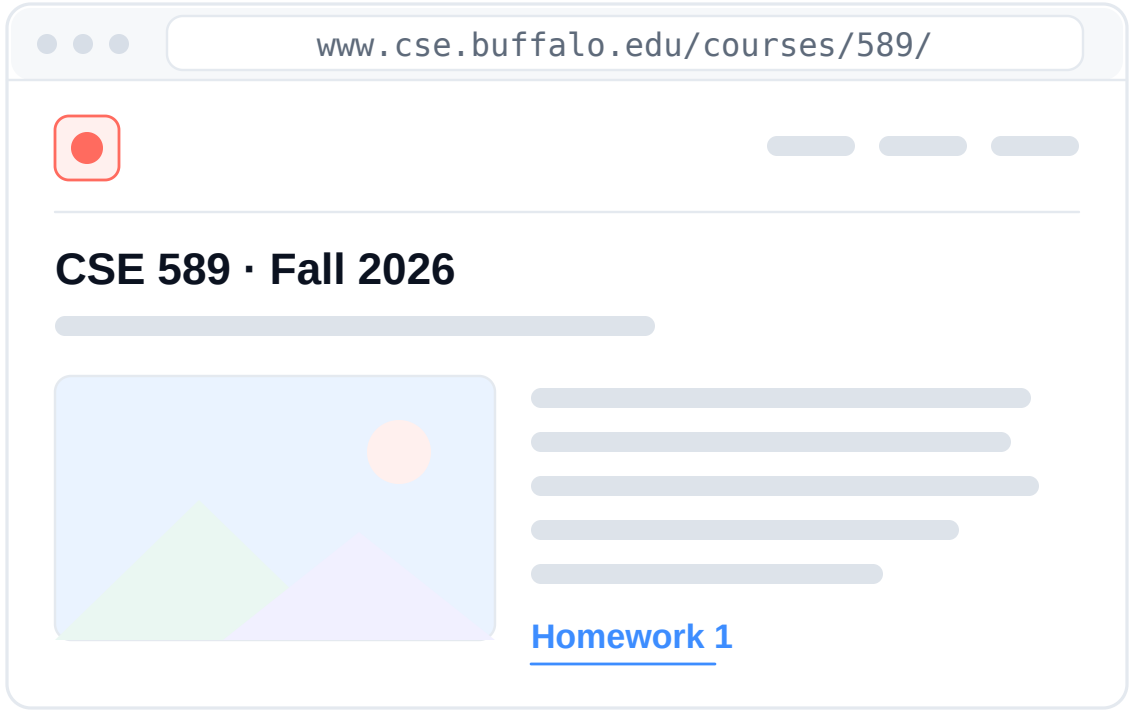


Half of all pages need more than this, and half need less.

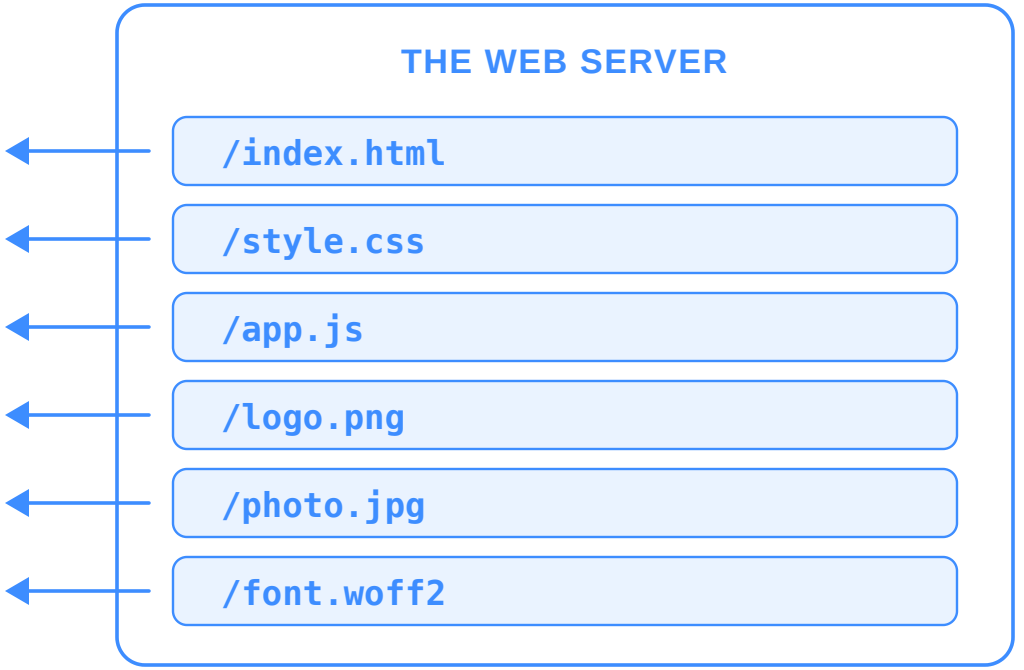
The HTTP Archive loads millions of home pages every month and records every file each one needs.

Loading one page means fetching dozens of separate files.

Every one of those files sits on a web server



The browser starts with none of them and asks for one at a time.



ordinary files, sitting on a disk

Each object has to be downloaded on its own.

A URL says which machine, and which file on it

host

which machine holds the file

DNS turns this name into an IP address

path

which file on that machine

it looks like a file path because it often is one

These two are the whole address: a machine, and a file on it.

Change either one and you are naming a different object.

To ask for one of those files you have to be able to name it, and the name has to work from anywhere. That is what a URL is.

Every object on a page has a URL of its own.

The port picks a program, the query passes it parameters

WHY A PORT

One machine runs many servers at once.

`128.205.32.52`

`:80`	the web server
`:25`	the mail server
`:22`	the ssh server

the port says which of them the connection is for

WHY A QUERY

A path names a file that exists.

`/photo.jpg`

A query asks a program to make one that does not.

`/search?q=network&page=2`

there is no file called search on the disk:
the server runs a program, reads q and page,
and builds the answer for this one request

The fragment (#syllabus) is the browser's own: it scrolls the page and is never sent.

A URL can name a file, or the input to a program.

The last piece is a way to ask for one file

the page
HTML says what the content is

✓

the objects
each one a separate file

✓

the names
a URL for every one of them

✓

missing
a way to ask for one

✗

THE RULES BOTH SIDES HAVE TO AGREE ON

how a client asks for the object named by one URL

how the server says whether it worked, and what it is sending back

how both sides know where one message ends and the next begins

how to do it dozens of times without starting over each time

Nothing said so far explains how a file gets from that server to this browser. A client and a server have to agree on the words they exchange.

That agreed set of rules is HTTP, and Part 3 is about it.

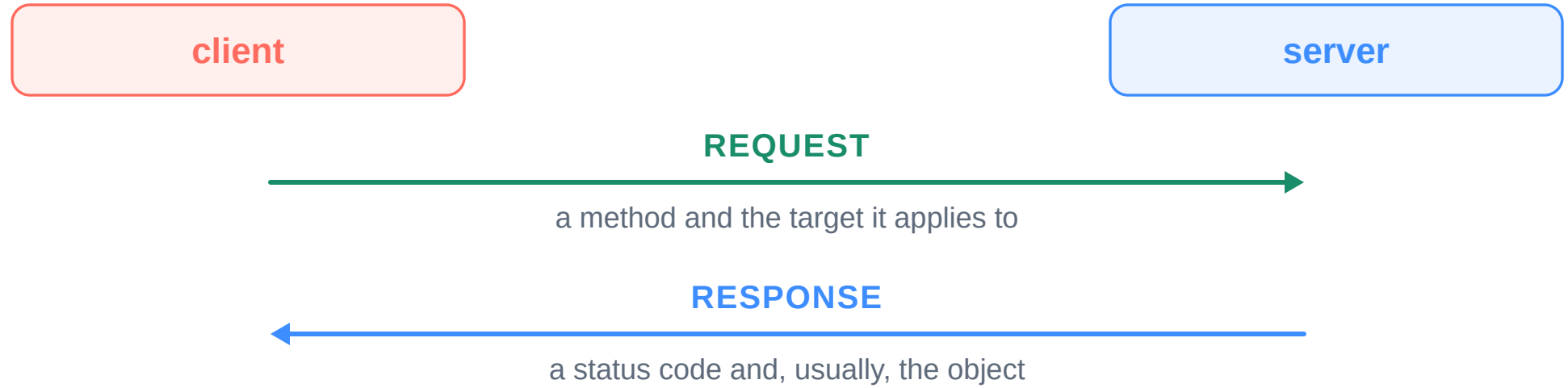
How do those files reach the browser?

03

HTTP

One request, one response, both in text. The request names a method and a target; the response carries a status code and the object itself.

HTTP is one request and one response

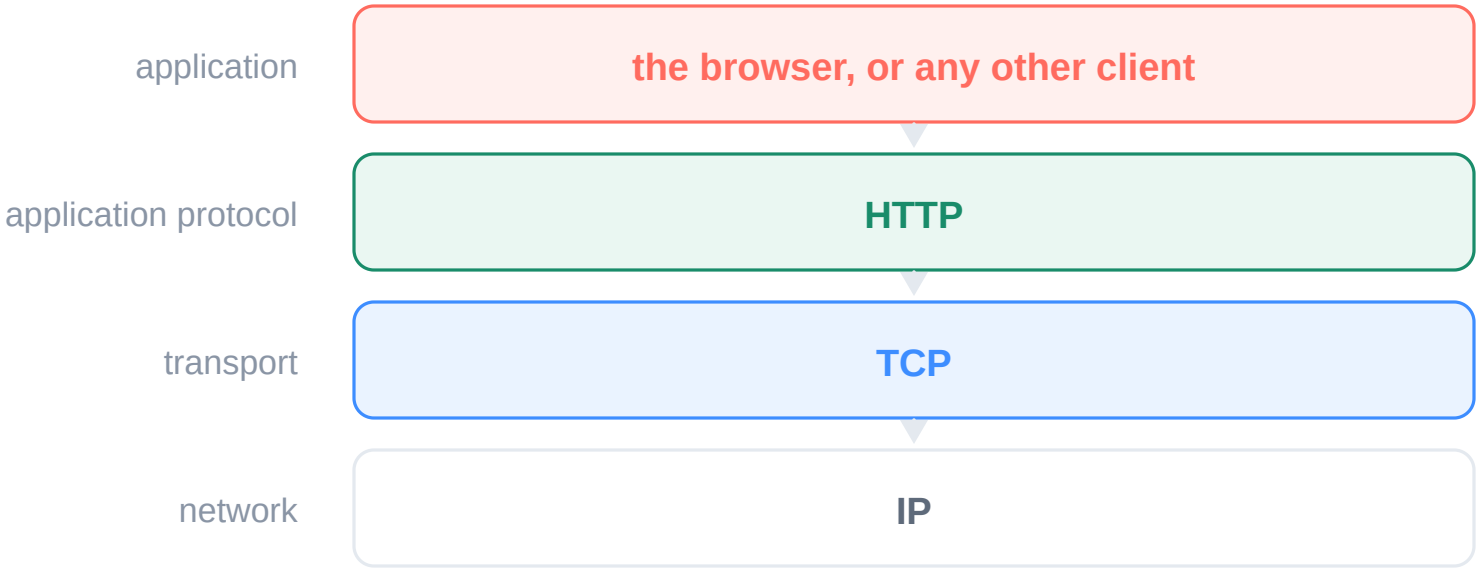


the client always speaks first | **one response per request** | **the message is text**

The server never sends anything the client did not ask for. Every object on a page is one of these exchanges, and the rules are written in RFC 9110 and RFC 9112.

The client asks for one object; the server answers once.

HTTP is an application-layer protocol



PORT

`80 for http`

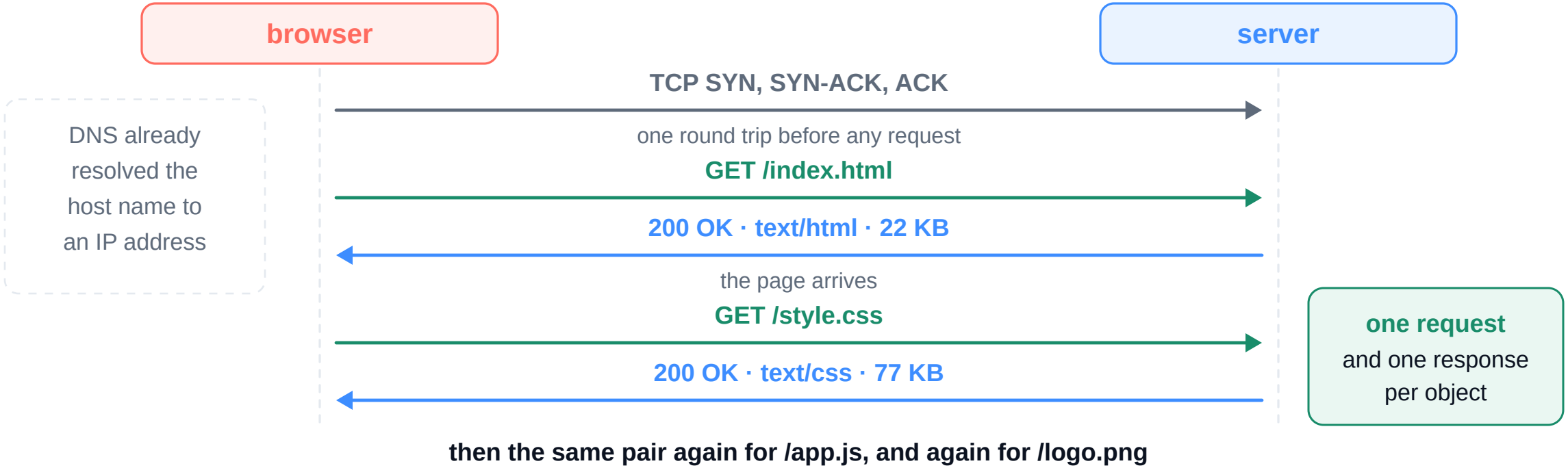
`443 for https`

HTTP defines the messages. The transport below it decides how those messages are carried.

It is a program on each side exchanging messages through a socket, like the servers written in the previous lecture. It does not move packets itself.

HTTP assumes a reliable, ordered byte stream underneath it.

What happens when you open one page



then the same pair again for /app.js, and again for /logo.png

Every arrow is a message on the same TCP connection. The objects the HTML names cannot be requested until the HTML itself has arrived and been parsed.

Each object costs a request and a response of its own.

Three round trips before the first byte of HTML

3 round trips before the first byte of HTML arrives

On a 40 ms path that is 120 ms in which nothing is transferred.

Every one of these is removed or reduced by something later in this lecture:

DNS caching · persistent connections

The DNS query is skipped when the answer is cached. The TCP handshake is paid once per connection, which is why connections are reused.

Setup cost is per connection, so connections are worth keeping.

A request is a start line, headers, and a body

THE BYTES ON THE CONNECTION

```
GET /courses/589/index.html HTTP/1.1
Host: www.cse.buffalo.edu
User-Agent: Mozilla/5.0
Accept: text/html
Accept-Encoding: gzip, br

(no body for a GET)
```

request line
one line: method, target, version

header fields
name: value, one per line

empty line
CRLF on its own — the headers end here

body
the bytes being sent, if there are any

Every line ends with CR LF. The blank line is what separates the headers from the body.

A GET request usually has no body, so it ends at the blank line. The whole message above is about 130 bytes.

Everything before the blank line is metadata; everything after is data.

The first line is a method, a target, and a version

```
GET /courses/589/index.html HTTP/1.1
```

method

what to do with the object

target

which object, on the host below

version

which rules the sender is using

The target is a path, not a full URL. The host name travels in the Host header instead.

The version tells the server which features it may use in the reply.

These three fields are separated by single spaces, and the line ends with CR LF. This is the only line every request must have.

One line states what to do and what to do it to.

RFC 9110 defines eight methods

METHOD	WHAT IT ASKS THE SERVER TO DO	SAFE	IDEMPOTENT
GET	return the object named by the target	✓	✓
HEAD	return the headers only, no body	✓	✓
POST	process the body as new data for the target	✗	✗
PUT	replace the object with the body	✗	✓
DELETE	remove the object	✗	✓
OPTIONS	report which methods the target supports	✓	✓

Safe: the request is not meant to change anything. Idempotent: sending it twice has the same effect as once.

CONNECT and TRACE are the other two; PATCH is defined by RFC 5789. In practice GET and POST carry almost all traffic, and a browser address bar can only issue GET.

The method states the intent; the server decides what it means.

GET asks the server to send back one file

THE PAGE CONTAINS <img src="/images/ub-logo.png">

THE BROWSER SENDS

```
GET /images/ub-logo.png HTTP/1.1
Host: www.buffalo.edu
```

THE SERVER SENDS BACK THE FILE

```
HTTP/1.1 200 OK
Content-Type: image/png
Content-Length: 14203

<14,203 bytes of PNG data>
```

every image on a page
is fetched with a GET of its own

every link you click
is a GET for the page behind it

the address bar
can only produce a GET

GET does not change anything on the server. Asking twice gives the same answer.

The target names the file wanted, and the response says what came back and how many bytes of it there are. The browser then draws those bytes where the tag stood.

GET means: send me the thing named by this target.

POST sends data to the server to be processed

A FORM IS FILLED IN AND SUBMITTED

user

student

password

••••••

Sign in

THE BROWSER SENDS

```
POST /login HTTP/1.1
Host: example.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 28

user=student&password=secret
```

the target says what the data is for; the body is the data

the body can be anything

a form, a file, a JSON document, an image

it is not in the address bar

so it cannot be bookmarked or re-opened

sending it twice is not the same

the browser warns before repeating a POST

POST means: here is some data, do something with it.

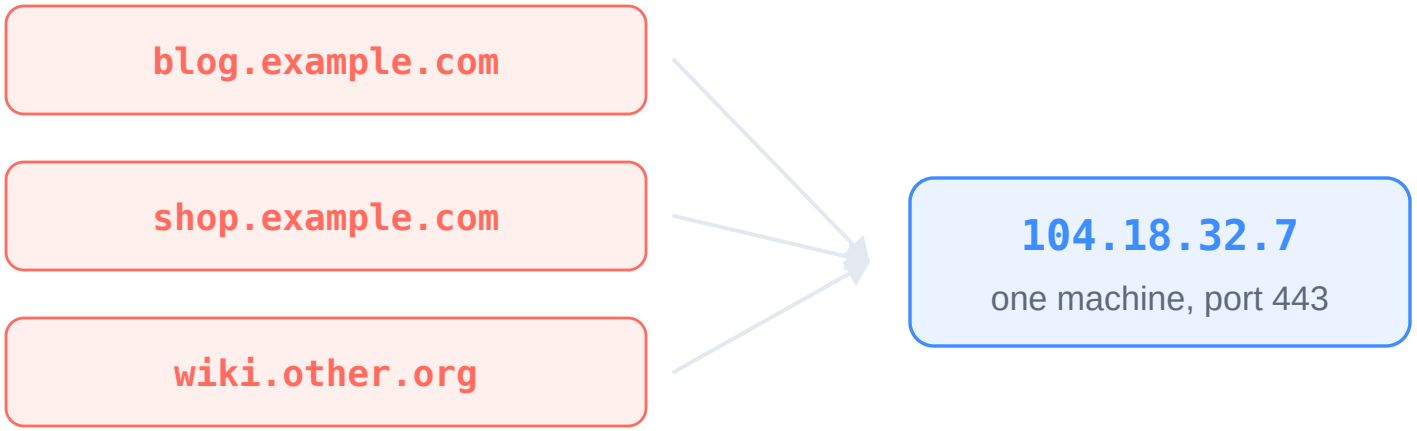
Headers describe the request and the client

`Host:`	which site on this address the request is for
`User-Agent:`	which browser and version is asking
`Accept:`	which content types the client can use
`Accept-Encoding:`	which compression formats it understands
`Accept-Language:`	which languages it prefers
`Cookie:`	the values this server stored earlier
`Referer:`	the page the link was on
`If-None-Match:`	the version already in the cache
`Range:`	which byte range of the object is wanted

Header names are case-insensitive and the order does not matter. A server ignores any header it does not recognise, which is how new headers get deployed without breaking anything.

Only Host is required; the rest are the client stating preferences.

Host is required, because one address serves many sites



DNS resolved all three names to this one address.

SENT AGAIN IN THE REQUEST

```
GET /index.html HTTP/1.1
Host: shop.example.com
```

Without this line the server cannot tell which of the three sites was requested.

The name was already translated to an address before the connection was opened, so it is gone by the time the request arrives. This header is the only place it appears again.

One IP address can host thousands of sites because of this one header.

A response is a status line, headers, and a body

THE BYTES COMING BACK

```
HTTP/1.1 200 OK
Date: Sat, 05 Sep 2026 14:02:11 GMT
Content-Type: text/html; charset=utf-8
Content-Length: 22147
Cache-Control: max-age=300

<!doctype html><html>...
```

status line
version, three-digit code, reason text

header fields
what the body is, and how to treat it

empty line
the headers end here

body
the object itself, Content-Length bytes of it

The reason text is for people. Programs read the three digits and ignore the words.

The structure is the same as the request. Only the first line differs: a status code instead of a method and a target.

The response says what happened and then sends the object.

The status code is three digits in five classes

1xx	**informational** 100 Continue · 103 Early Hints
2xx	**the request succeeded** 200 OK · 204 No Content · 206 Partial Content
3xx	**look somewhere else, or use your cache** 301 Moved Permanently · 302 Found · 304 Not Modified
4xx	**the request was wrong** 400 Bad Request · 403 Forbidden · 404 Not Found · 429 Too Many
5xx	**the server failed** 500 Internal Server Error · 502 Bad Gateway · 503 Unavailable

A client that does not know a specific code must treat it as x00 of the same class.

A response carries exactly one code. New codes keep being defined, which is why a client is required to fall back to the class the code belongs to.

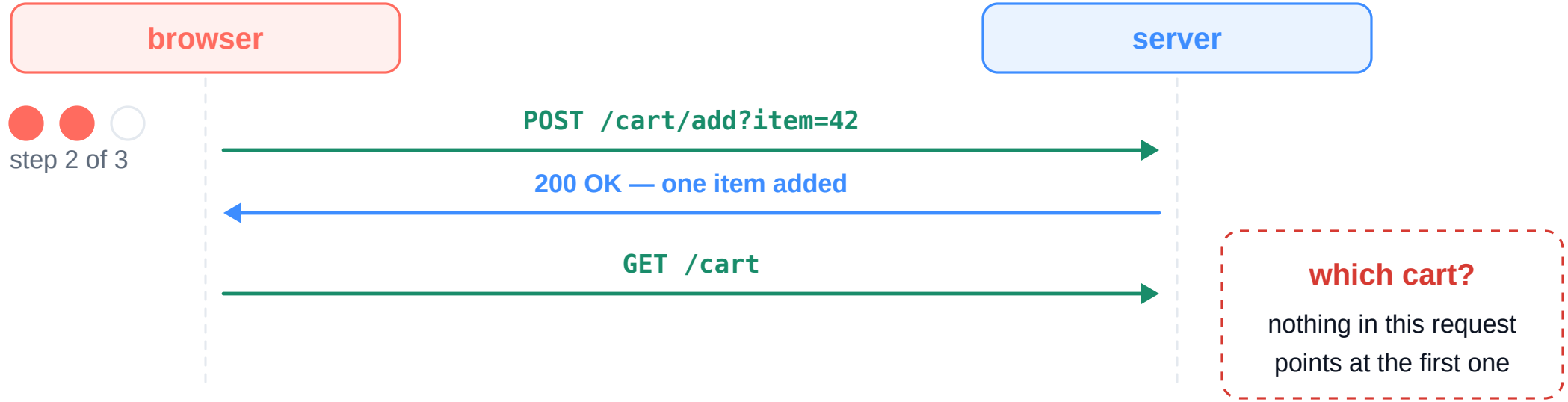
The first digit is what a client checks first.

Nothing in a request refers to the one before it

The server works out the answer and sends it. Once the response has gone it keeps nothing: no note that this browser now has a cart, and no way to be reminded of one.

One request in, one response out, and then it is over.

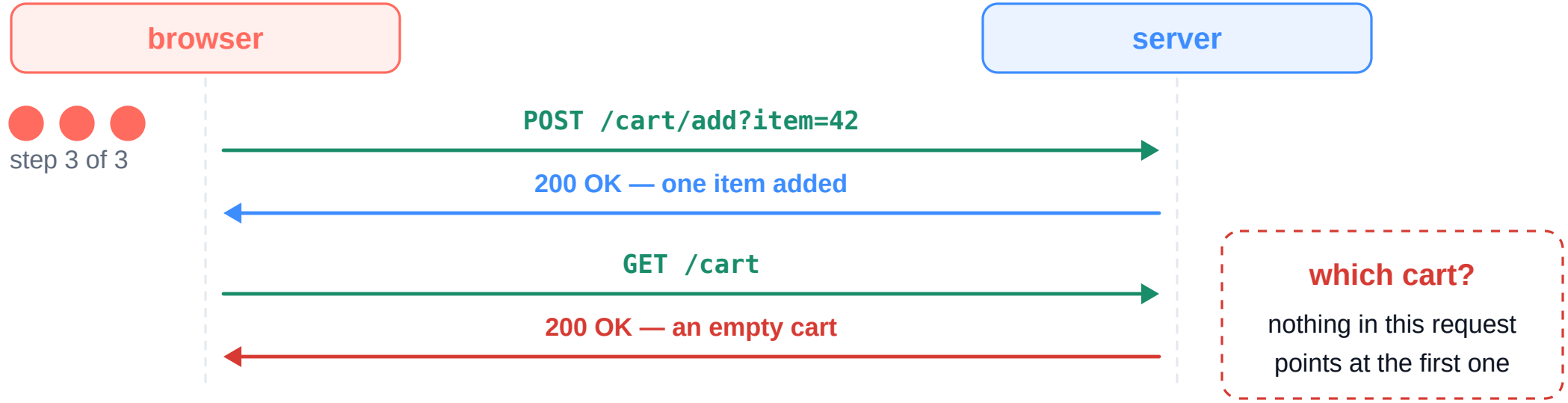
Nothing in a request refers to the one before it



This request carries a method, a target and some headers. None of them say who is asking, so the server has nothing to match it against.

The server cannot tell that this is the same browser.

Nothing in a request refers to the one before it



But a real shopping site does not lose your cart.

Something has to be added to every request to tie it to the one before it. That is Part 4.

This is a design choice, not an oversight. Because no request depends on an earlier one, any server in a group can answer any request, and one can fail without losing anything.

HTTP has no memory. Part 4 is about what gets added on top to give it one.

git clone and pip install are HTTP clients too

CLONING A GIT REPOSITORY

```
$ git clone https://github.com/git/git

GET /git/git.git/info/refs?service=git-upload-pack
Host: github.com
200 OK — the list of branches

POST /git/git.git/git-upload-pack
Content-Type: application/x-git-upload-pack-request
200 OK — every object, as one pack file
```

INSTALLING A PYTHON PACKAGE

```
$ pip install numpy

GET /simple/numpy/
Host: pypi.org
200 OK — every released file

GET /packages/.../numpy-<version>.whl
Host: files.pythonhosted.org
200 OK — the file, then pip unpacks it
```

The same methods, the same status codes, the same headers as a browser.

Neither program draws anything on a screen.

Building on HTTP is cheaper than defining a protocol: port 443 is reachable from almost every network, every language ships a client, and the caching rules in Part 4 come with it.

HTTP is the default way for one program to call another.

A loaded page keeps issuing HTTP requests

WHILE THE PAGE LOADS

the browser requests

HTML, CSS, JavaScript, images, fonts

AFTER THE PAGE IS DISPLAYED

the JavaScript on the page requests more

`fetch("/api/messages")`

search results, prices, messages, feed items

THE SAME PROTOCOL, NO PAGE INVOLVED

```
GET /api/messages?since=1830 HTTP/1.1
Host: mail.example.com
Accept: application/json

HTTP/1.1 200 OK
Content-Type: application/json

[{"id":1831,"from":"tas@buffalo.edu"}]
```

This is what a mobile application sends as well.

An application in a browser makes requests for data long after the page is displayed. The response is usually JSON rather than HTML, but nothing about the protocol changes.

HTTP carries the application data, not just the page.

What does HTTP do beyond moving bytes?

04

What HTTP adds

Reusing connections, not fetching an object twice, sending the right form of it, and recognising the same client across requests.

Four mechanisms reduce the cost of a page load

72 requests for one page, and the same page loaded again tomorrow

Reuse the connection	Do not fetch it twice	Send fewer bytes	Know who is asking
persistent connections	caching	compression and ranges	cookies

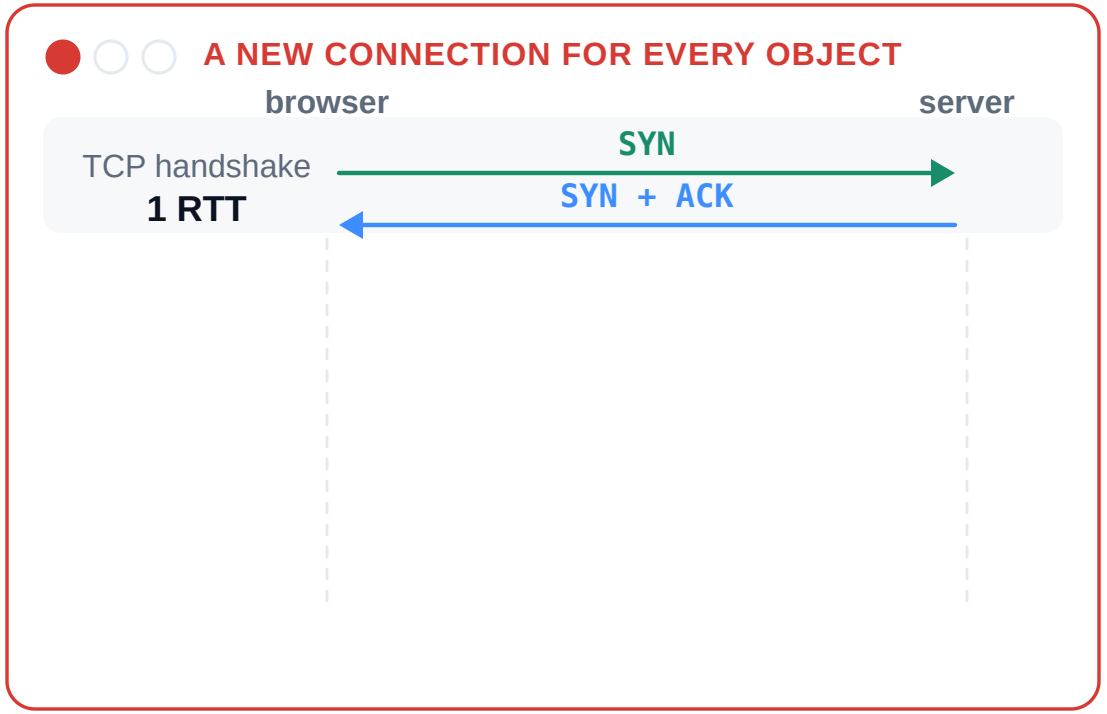
All four are in the protocol itself, not in the browser or the server application.

Each one is a set of header fields and rules for how a client and a server are allowed to behave. The rest of this part is those four, in order.

HTTP is mostly rules about what may be skipped.

1 Why the web | 2 How a page works | 3 HTTP | 4 What HTTP adds

A handshake costs one round trip, a request costs one more

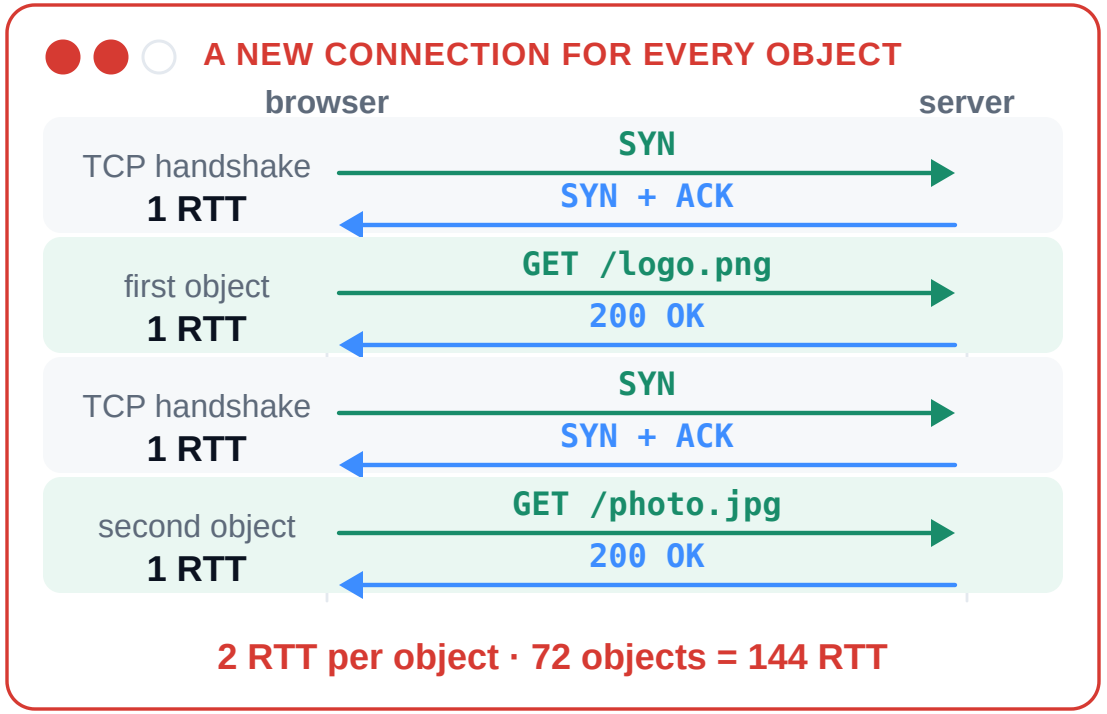


the fix comes next

Before any HTTP message can be written, TCP has to exchange SYN and SYN-ACK. That is one round trip in which nothing of the page moves.

A handshake buys nothing but the right to send.

A handshake costs one round trip, a request costs one more

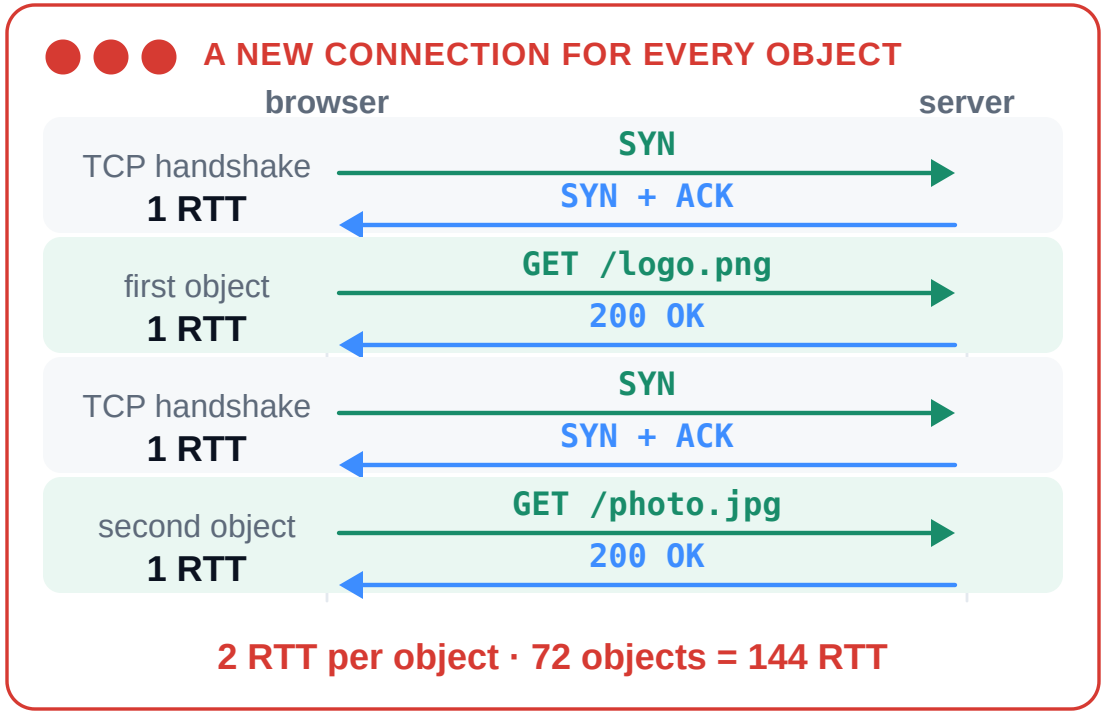


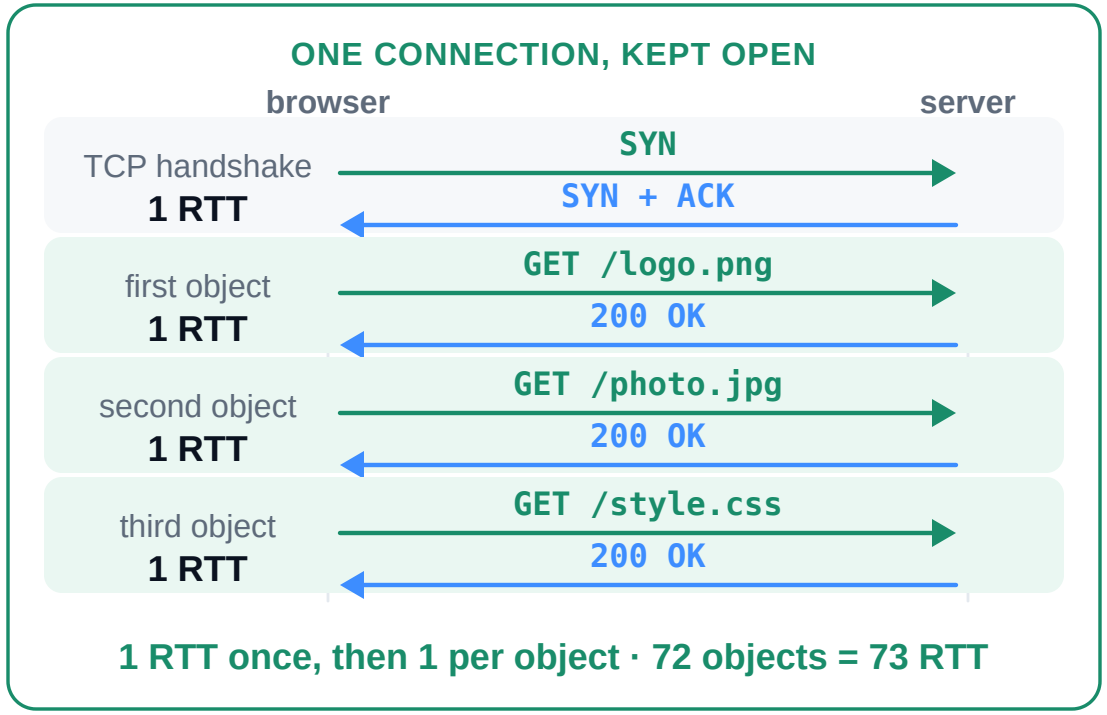
the fix comes next

Then the request goes out and the response comes back: one more round trip. Close the connection afterwards and the next object pays both again.

A connection per object pays two round trips for every file.

A handshake costs one round trip, a request costs one more

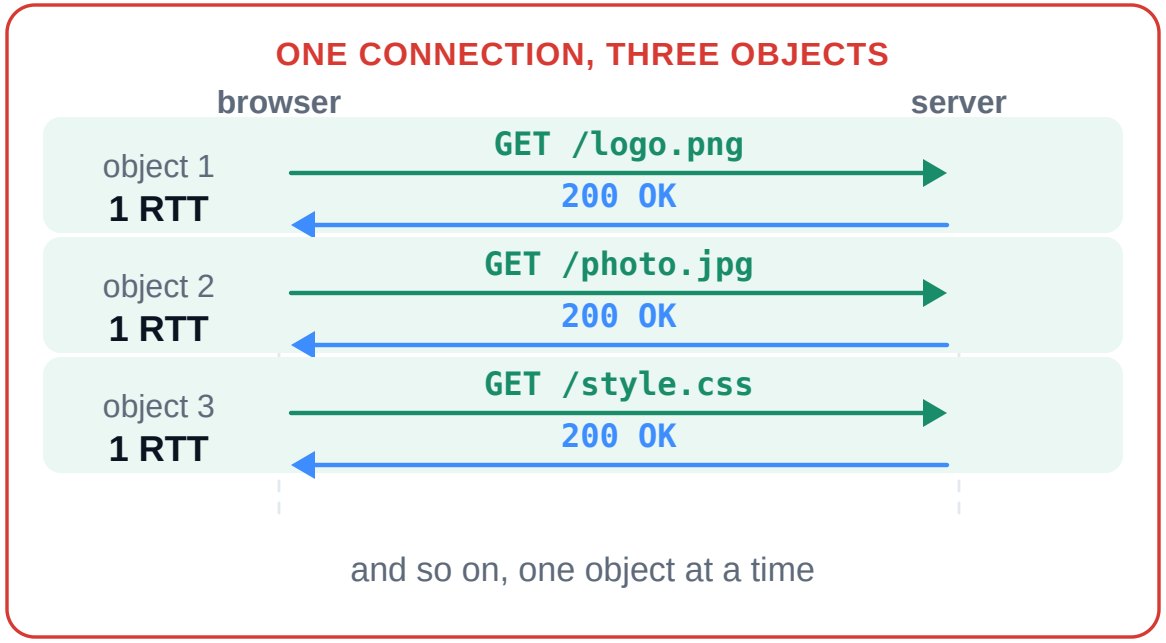




HTTP/1.1 leaves the connection open by default, so the handshake is paid once and every object after it costs one round trip.

Only the first object pays for a handshake.

One connection carries one request at a time



THE RULE IN HTTP/1.1

a request may not be sent until the whole response to the one before it has arrived

SO, FOR ONE PAGE

72 objects × 1 RTT = 72 round trips

on a 40 ms path that is 2.9 seconds of pure waiting, before a single byte

This is why no browser uses one connection.

The connection is a byte stream shared by both sides. HTTP/1.1 has no way to label which response belongs to which request, so it allows only one exchange to be in flight on it.

One connection is one object at a time.

Browsers open six connections to the same host

WHAT A BROWSER ACTUALLY DOES

connection 1	one object	one object	one object
connection 2	one object	one object	one object
connection 3	one object	one object	one object
connection 4	one object	one object	one object
connection 5	one object	one object	one object
connection 6	one object	one object	one object

six exchanges in flight at once, instead of one

This is what every browser does today.

It works, and it is still how HTTP/1.1 is used.

THE SAME PAGE, SIX CONNECTIONS

72 objects ÷ 6 = 12 rounds, not 72

about 0.5 s of waiting instead of 2.9 s

WHY NOT SIXTY?

each one pays its own handshake

they share one link, so they slow each other down

a server treats it as an attack

ROUNDS OF WAITING FOR ONE PAGE

HTTP/1.1 gets its parallelism from connections, not from the protocol.

The second visit asks for the same files again

YOUR FIRST VISIT

index.html	22 KB	downloaded
photo.jpg	310 KB	downloaded
logo.png	14 KB	downloaded
font.woff2	28 KB	downloaded
style.css	77 KB	downloaded
app.js	210 KB	downloaded

661 KB downloaded

THE SAME PAGE TOMORROW

index.html	22 KB	changed
photo.jpg	310 KB	identical
logo.png	14 KB	identical
font.woff2	28 KB	identical
style.css	77 KB	identical
app.js	210 KB	identical

22 KB changed · 639 KB identical

The same five files are also sent to every other visitor, and to you again on every page of the site.

90.4% of all responses on the web are marked as safe to store and reuse.

Most of what a page needs was already downloaded once.

1 Why the web | 2 How a page works | 3 HTTP | 4 What HTTP adds

A cache stores a response and reuses it

A CACHE

a store of responses: when a request arrives that it has already answered, it returns the stored copy instead of passing the request on.

01 **Where are the copies kept?**
in the browser, in the network, at a CDN

02 **How does anyone know a copy is still correct?**
Cache-Control, expiry, and the 304 response

03 **Who runs the ones that are not yours?**
forward proxies, reverse proxies, content delivery networks

The rest of this part answers those three questions.

1 Why the web | 2 How a page works | 3 HTTP | 4 What HTTP adds

A request passes up to three caches before the origin

THE REQUEST STOPS AT THE FIRST COPY IT FINDS

browser cache
on the same machine
`0 ms`

→

a cache at UB
on the campus network
`2 ms`

→

a CDN node
near the client
`10 ms`

→

origin server
the only full copy
`90 ms`

Typical round-trip time to each one, from a client in Buffalo.

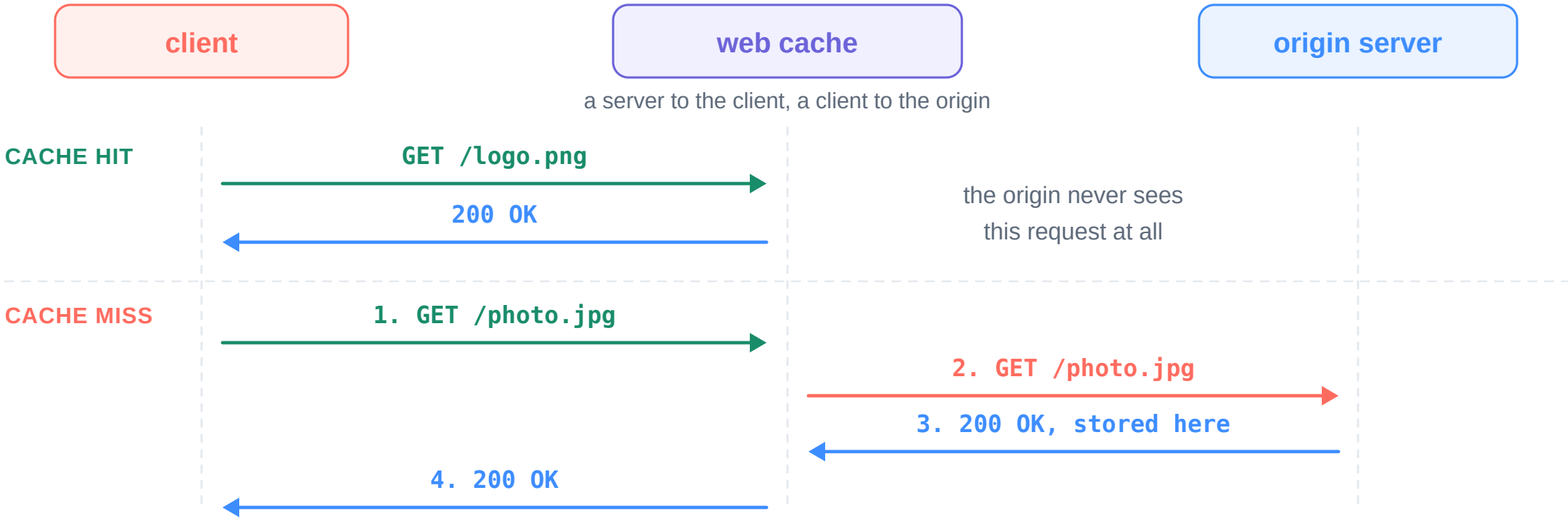
35% of HTML documents and 71% of third-party objects are served from a CDN

HTTP Archive, 2025 crawl

The browser cache is **private**: it holds one person's responses. The other two are **shared** caches, where one stored copy answers many different clients. HTTP names and controls the difference.

Caching removes the request, not just the transfer.

A web cache answers requests it can, and forwards the rest

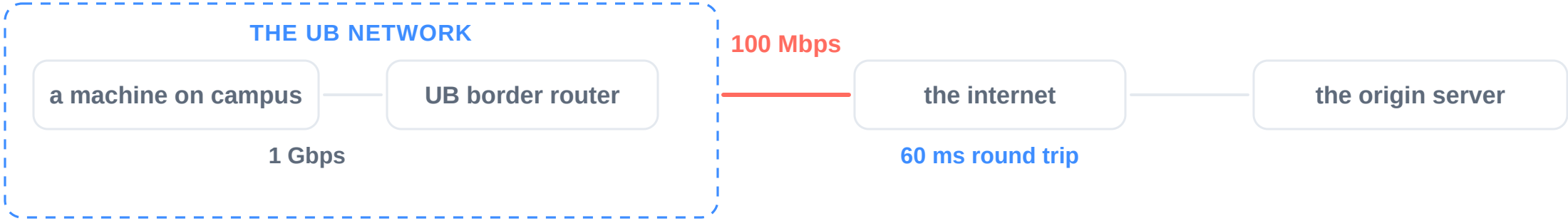


It is an ordinary HTTP server to the client and an ordinary HTTP client to the origin. Both sides use the protocol from Part 3; neither has to know a cache is there.

On a miss the cache issues its own request and stores what comes back.

A worked example

One object, from a campus machine to an origin



the average object	240 kbit	30 KB, from the page numbers in Part 2
requests leaving campus	400 per second	across everyone on the network
the link UB pays for	100 Mbps	the only slow link on the path
the internet, router to origin	60 ms	a round trip UB does not control

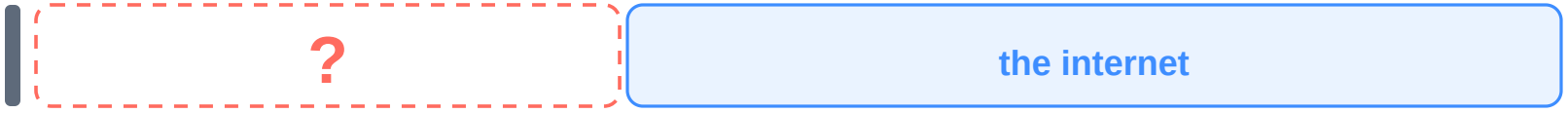
How long does one object take, from the request to its last byte?

Four numbers, and one question to answer with them.

1 Why the web | 2 How a page works | 3 HTTP | 4 What HTTP adds

The delay has three parts, and one of them moves

THE TIME FOR ONE OBJECT, ADDED UP



the campus LAN
0.24 ms

still to be worked out

60 ms

the campus LAN — **240 kbit ÷ 1 Gbps = 0.24 ms**
too small to matter, and UB could not make it smaller anyway

the internet — **fixed at about 60 ms**
the path to the origin is not UB's to change

the 100 Mbps link — **depends on how busy the link is**
this is the only part UB owns, and the only part that moves

Only the campus link is under UB's control.

Delay on a link explodes as the link fills up

THE ARITHMETIC

ONE OBJECT ON THE LINK

240 kbit ÷ 100 Mbps

= 2.4 ms

TRAFFIC OFFERED

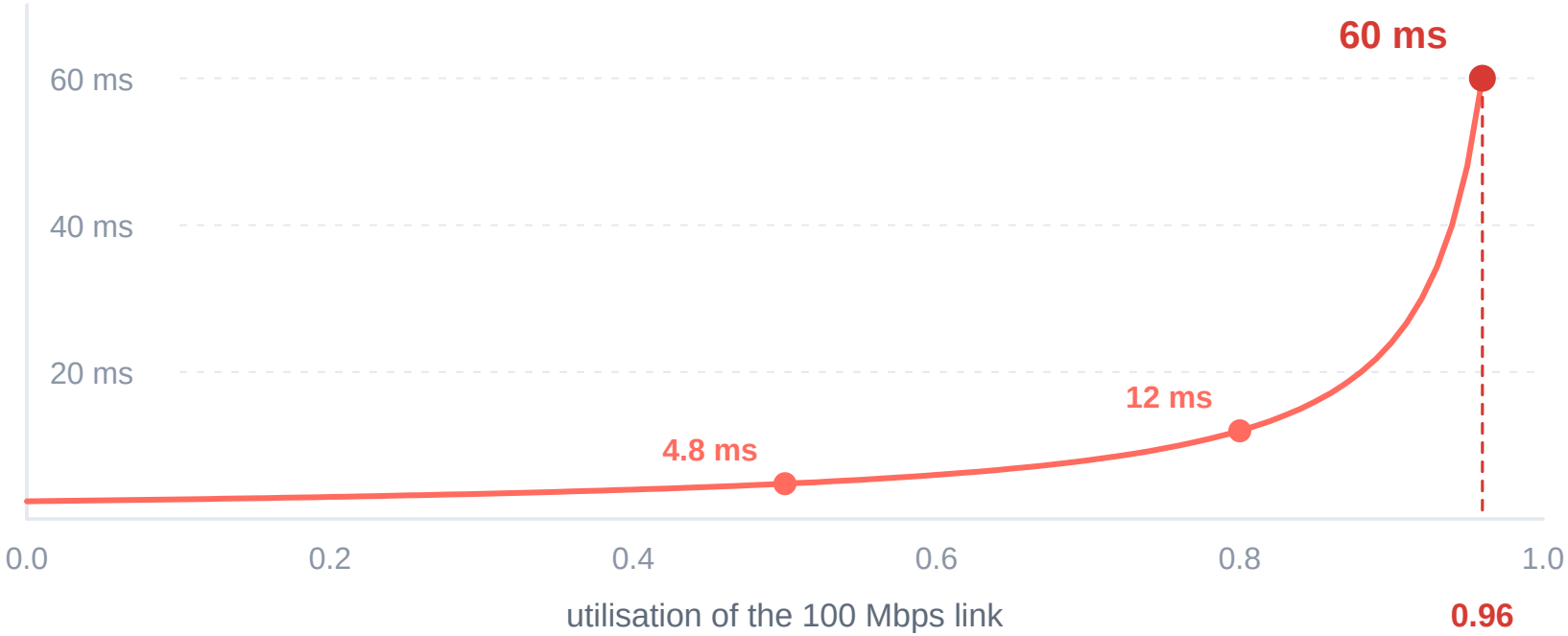
400 × 240 kbit

= 96 Mbps

UTILISATION

96 ÷ 100

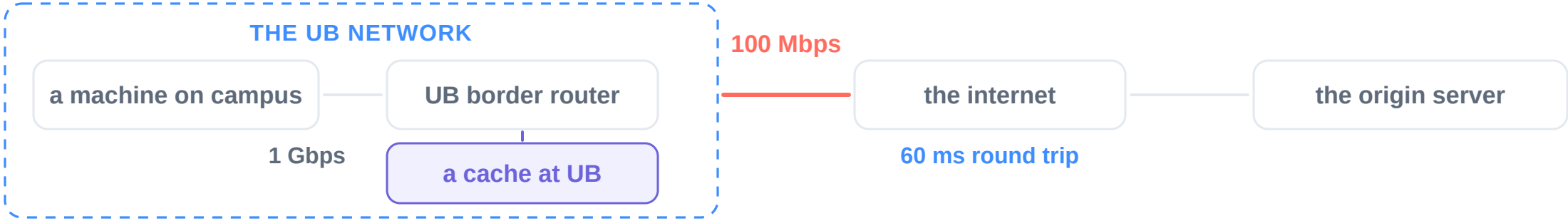
= 0.96



Average time through the link ≈ the transmission time of one object divided by one minus the utilisation. At 0.5 an object waits about twice its own transmission time; at 0.96 it waits twenty-five times.

At 96% busy, one object waits 60 ms for the link alone.

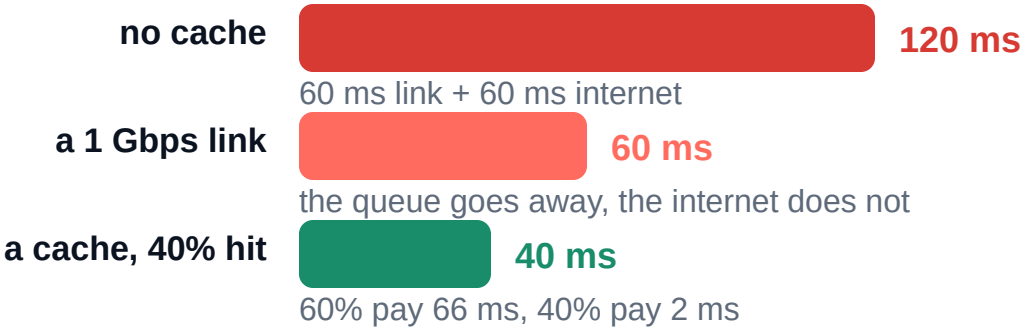
The cache beats a link ten times faster



WITH A 40% HIT RATE

```
240 × 240 kbit        = 58 Mbps
58 ÷ 100              = 0.58
2.4 ÷ (1 − 0.58)      = 5.7 ms
```

AVERAGE TIME FOR ONE OBJECT



A hit is answered inside the campus, so it pays neither the link nor the internet. That is why a cache wins against an upgrade that only removes the queue.

40% of the requests stop costing anything at all.

Cache-Control: max-age states how long a copy stays fresh

THE SERVER SETS THE LIFETIME

```
HTTP/1.1 200 OK
Content-Type: image/png
Cache-Control: max-age=2592000
ETag: "9f3c1a"
Date: Sat, 05 Sep 2026 14:02:11 GMT
```

max-age is in seconds. 2,592,000 = 30 days.

Cache-Control is on 74.8% of responses.
max-age is the most common directive, on 62.2%.
The most common value is 30 days.

within 30 days

the cache answers by itself

no request is sent

after 30 days

the response is stale

the cache must check

A fresh copy is returned without contacting the server at all. This is the only cache mechanism that removes the network entirely.

The server decides how long the client may skip asking.

A stale copy is checked, not re-downloaded

The round trip is still paid, but the object is not sent again. For a page of images that is the difference between 2 MB and a few kilobytes.

304 means: what you have is still correct.

A validator is either a version tag or a timestamp

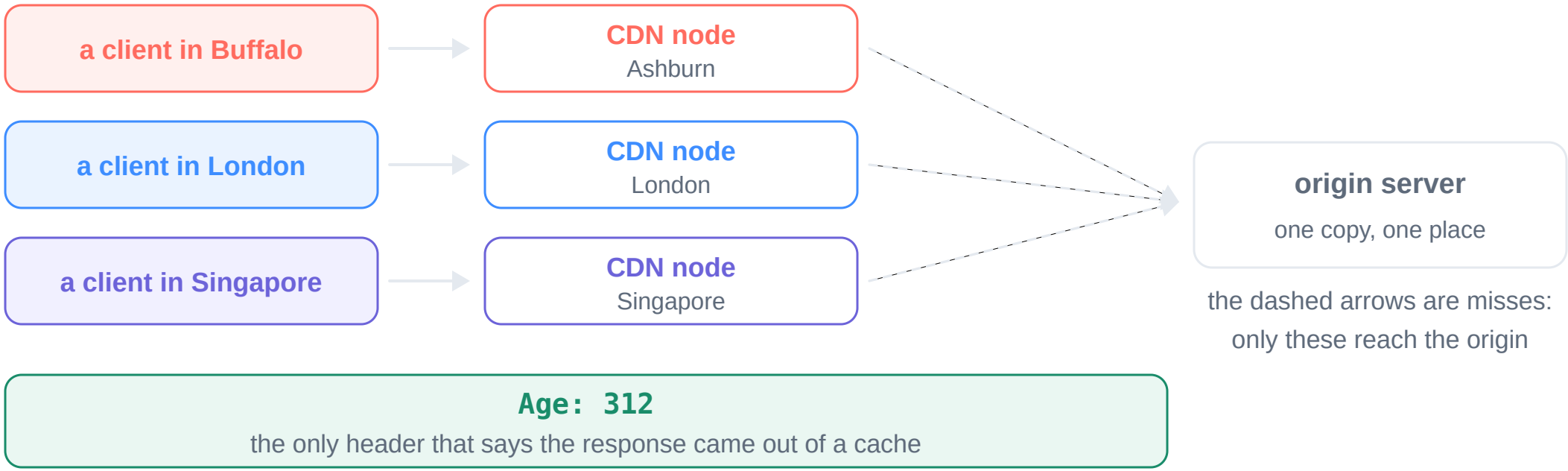
ETag / If-None-Match	Last-Modified / If-Modified-Since
a value the server made up for this version	a timestamp, one-second resolution
usually a hash of the content	two changes in the same second look identical
compares exactly: any change is a new value	a rebuilt but unchanged file looks new
on 46.5% of responses	on 70.5% of responses

A server may send both. The client sends back whichever it was given.

The client sends back whatever validator it was given and the server does the comparison, so the client never has to understand the value.

ETag compares versions exactly; Last-Modified compares seconds.

A CDN is the same caching server, in many places



`Age: 312`
the only header that says the response came out of a cache

DNS is what sends each client to a nearby node.

35% of HTML documents and 71% of third-party objects are served this way. A CDN node answers with the same status codes and header fields the origin would have sent.

Caching is what makes one origin serve the whole world.

One URL can return different forms of the same object

`Accept: text/html`	→	`Content-Type: text/html`	which format
`Accept-Language: en-US`	→	`Content-Language: en-US`	which language
`Accept-Encoding: br`	→	`Content-Encoding: br`	which compression

`Vary: Accept-Encoding, Accept-Language`

the response depends on these request headers, so a cache must key on them too
without Vary a shared cache would hand the English page to everyone

The client states what it can use and the server chooses. The URL does not change, which is why the cache has to be told what the choice depended on.

Vary tells a cache which request headers are part of the key.

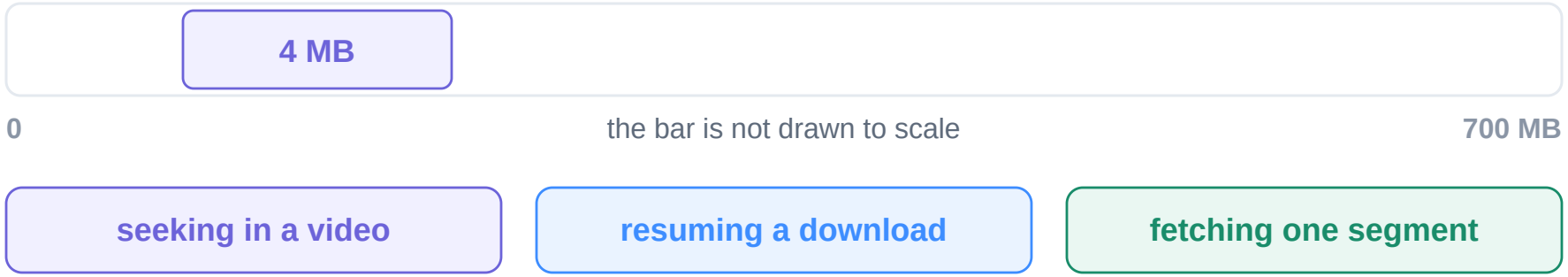
A client can ask for part of an object

ASK FOR PART OF AN OBJECT

```
GET /lecture.mp4 HTTP/1.1
Range: bytes=8388608-12582911
```

THE SERVER SENDS ONLY THAT PART

```
HTTP/1.1 206 Partial Content
Content-Range: bytes 8388608-12582911/734003200
```



seeking in a video | **resuming a download** | **fetching one segment**

The server replies 206 with the requested bytes and states which part of the whole they are. This is what lets a video player skip to the middle without downloading the beginning.

The unit of transfer does not have to be the whole object.

The application needs memory the protocol does not have

WHAT THE SERVER NEEDS

this is the same person who put
two items in the cart four requests ago
and who logged in before that

WHAT HTTP PROVIDES

nothing

each request is processed on its own

→

the client carries the state

the server sends a value once,
and the client returns it on
every later request

Defined in RFC 6265.

Statelessness is what lets any server answer any request. Cookies keep that property by moving the state into the request instead of into the server.

A cookie is state stored on the client and sent back every time.

One header sets the value, another returns it

The value means nothing to the browser: it stores what the server sent and returns it unchanged. What the value identifies is looked up on the server.

The cookie identifies the client; the data stays on the server.

The Set-Cookie header carries rules with the value

`Domain=example.com`	which hosts the cookie is sent to
`Path=/account`	which paths it is sent for
`Max-Age=31536000`	how long the browser keeps it, in seconds
`Secure`	send it only over HTTPS
`HttpOnly`	JavaScript on the page cannot read it
`SameSite=Lax`	do not send it on cross-site requests

A cookie without Max-Age or Expires is deleted when the browser closes.

The attributes are instructions to the browser, not data for the server. The browser is what enforces them.

Secure, HttpOnly and SameSite limit where the value can travel.

A typical page carries nine cookies

9
cookies on the median page

40
bytes, the median cookie size

365
days, the median lifetime

4,096
bytes, the size limit per cookie

40% first-party — set by a host other than the one in the address bar — **60%** third-party

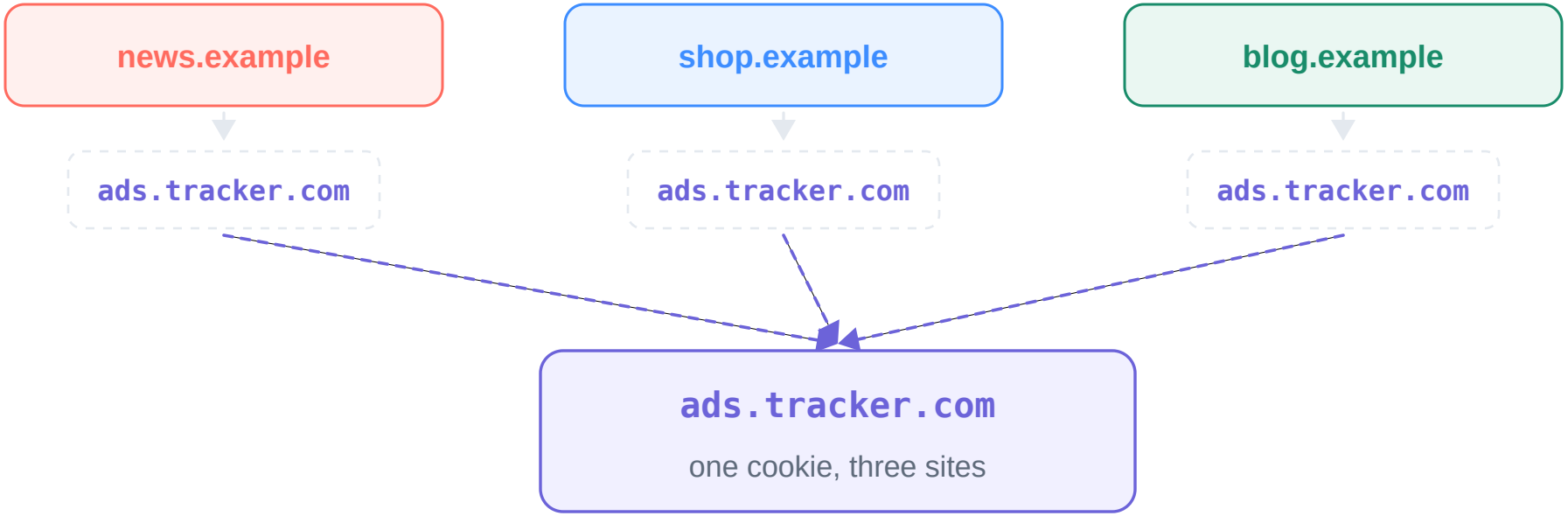
Only 12% of first-party cookies set HttpOnly, and 3% set SameSite=Strict.

Measured across millions of pages by the HTTP Archive, 2025.

Half of all pages carry more than nine. The 4,096-byte limit is set by RFC 6265 and is what stops a cookie from being used as storage.

Most cookies on a page were not set by the site being visited.

A cookie set by an embedded object follows the browser



Every page that embeds the same third-party object sends that third party the same cookie, so it learns which sites this browser visited.

The browser sends a cookie to the host that set it, and an embedded image or script is a request to that host. Nothing about this needed a new mechanism.

The cookie belongs to the host it was set by, not to the page.

Four questions to check yourself against

01 What is the unit HTTP transfers, and how is it named?

02 Which parts of a message are metadata, and which are data?

03 Which mechanism removes a request, and which only shortens it?

04 Where may a copy of a response legitimately be kept?

The same four apply to any application protocol: what is the unit, what is the metadata, what may be skipped, and who is allowed to keep a copy.

Everything in Part 4 exists to avoid sending something twice.

1 Why the web | 2 How a page works | 3 HTTP | 4 What HTTP adds