



# Advanced Computer Networking

Ian Batten and Matthew Leeke  
School of Computer Science  
University of Birmingham

Topic 5 - HTTP and Friends

# 91254005



# FTP(s)

- Wide range of protocols available to “send a filename, receive a file”.
  - FTP, which we mentioned in NAT lecture, probably oldest and certainly ugliest
  - kermit and uucp different ways to use serial lines (complete with own “transport” layers)
    - uucp mostly Unix-only (although there were versions for DOS), still shipped on MacOS
    - kermit multi-platform
    - both can be coerced into running over a TCP connection
  - Also xmodem, ymodem, an endless parade.

# pre-HTTP

- Range of protocols for finding resources to then fetch (probably with FTP)
  - gopher
  - archie
  - There were others
- All completely killed by rise of HTTP and (later) Search Engines

# Why HTTP?

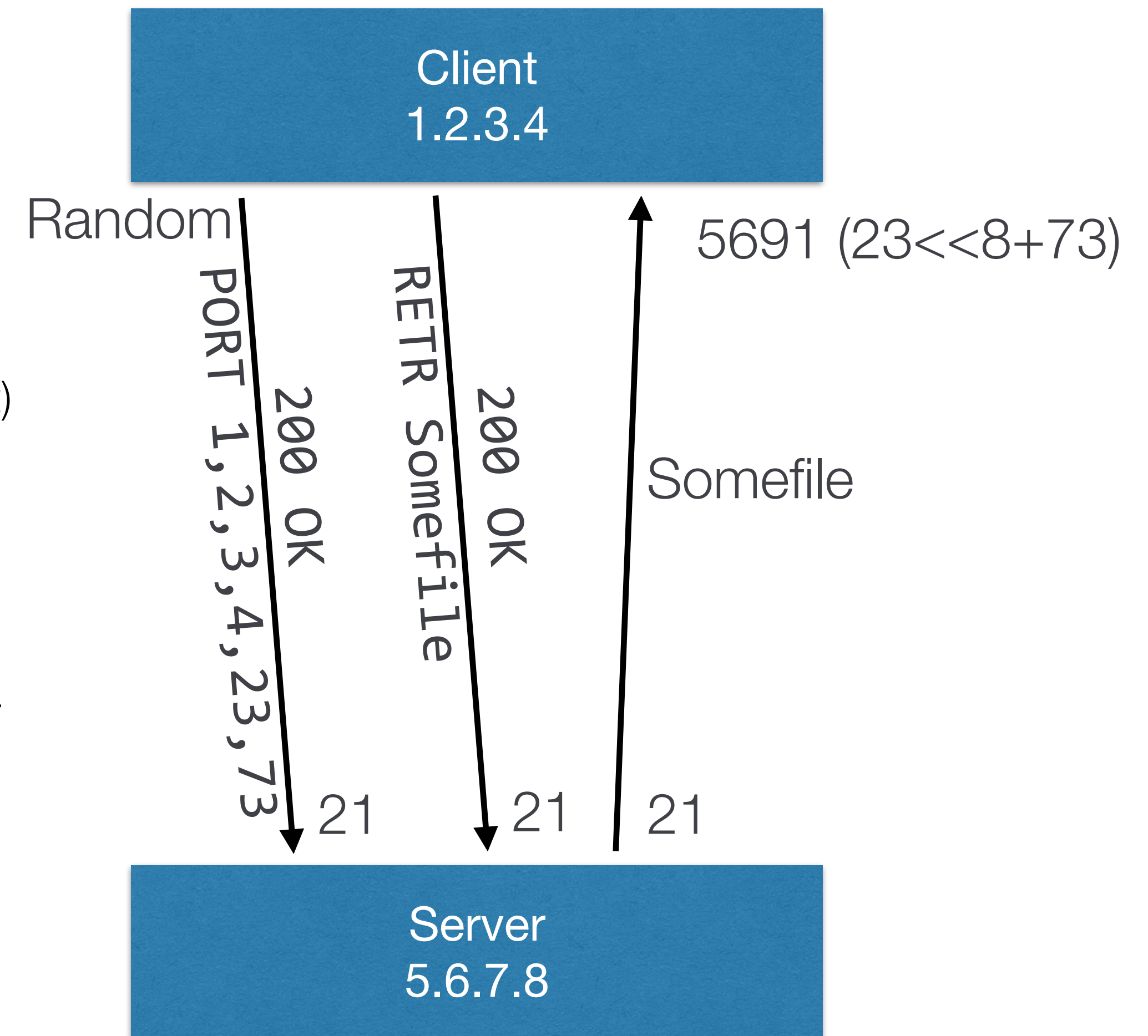
- Hypertext Transport Protocol.
- Originally designed to deal with downloading HTML with support for hyperlinks (“HTTP GET”).
- Extensible to a wide range of other tasks with other commands (“HTTP POST”, “HTTP PUT”).
- Flexible concept of URL means that it can do many other tasks apart from shipping files.

# Why HTTP?

- Protocol is relatively easy to implement, but very flexible.
  - “Universal firewall bypass protocol”
- Protocol is not a screaming nightmare for networking engineers, unlike FTP.
- Protocol decouples names from things (URLs look like Unix paths, but don't have to be).

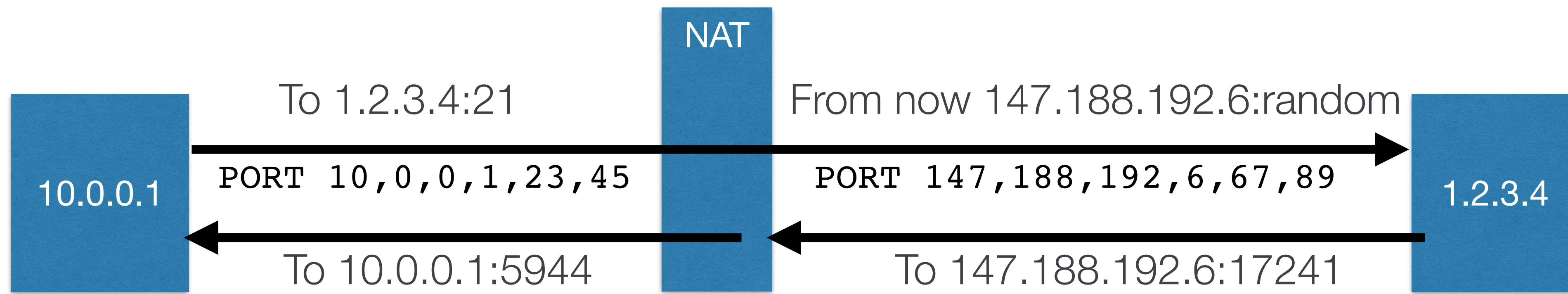
# Contrast with FTP (RFC 959)

- FTP uses a lot of connections
  - Open a control connection from client to server
  - When transferring data, open a random LISTENING socket on the CLIENT, send the IP/Port pair over the control connection from the server to the client (originally port 20, but that's not usable by not-root)
  - Client blocks waiting for a call back from server
  - Server uses information that arrived over the control connection to make the connection to client (doesn't work via firewalls without help, doesn't work via NAT without a lot of help) and then data passes over that connection until it closes.
  - Legacy of simplex NCP connections, but also need to send data whose length in bytes may not be easy to compute in advance and in which it is difficult to put EOF markers (record images, for example).



# What is the NAT “helper” doing?

- **We will never speak of these details again, especially not in an exam.**
- NAT point looks at an FTP control connection, identified by being port 21 at one end and having done “FTP-ish” things, hunting for the sequence “PORT” followed by six comma-separated numbers. Note: we are inside the payload here.
- When it sees it, do NAT (ie, change the first four numbers to our IP address, and allocate a port we encode in the last two) and then set up inbound NAT from the newly allocated port back to the client’s expected port.
- MODIFY THE PAYLOAD so that it’s correct now. This is the horrible thing.
- Client 10.0.0.1 is listening on 5933. NAT sets up inbound mapping for 147.188.192.6 port 17241 to 10.0.0.1 port 5944.
  - Mapping may be limited to things coming from the destination,
- Changes payload as below



# Passive Mode does not really help

- In Active (traditional, legacy) mode, data passes over a connection built by the server to a random port on the client
- In Passive mode, data passes over a connection built by the client to a random port on the server
  - Server opens random port, informs the client, client calls to that random port
  - Solves the NAT problem, usually
  - Still difficult for firewalls



# Why the complexity?

- Firstly, because NCP did not have full-duplex channels
- Also...

Data is transferred from a storage device in the sending host to a storage device in the receiving host. Often it is necessary to perform certain transformations on the data because data storage representations in the two systems are different. For example, NVT-ASCII has different data storage representations in different systems. DEC TOPS-20s's generally store NVT-ASCII as five 7-bit ASCII characters, left-justified in a 36-bit word. IBM Mainframe's store NVT-ASCII as 8-bit EBCDIC codes. Multics stores NVT-ASCII as four 9-bit characters in a 36-bit word. It is desirable to convert characters into the standard NVT-ASCII representation when transmitting text between dissimilar systems. The sending and receiving sites would have to perform the necessary transformations between the standard representation and their internal representations. [...]

The "natural" structure of a file will depend on which host stores the file. A source-code file will usually be stored on an IBM Mainframe in fixed length records but on a DEC TOPS-20 as a stream of characters partitioned into lines, for example by <CRLF>. If the transfer of files between such disparate sites is to be useful, there must be some way for one site to recognize the other's assumptions about the file.

# HTTP Basic structure

- Client sends command
- Client sends some (optional) options
- Client sends a blank line
- Server sends a status
- Server sends some commentary and information
- Server sends a blank line
- Server sends some data
- **This all happens over one single TCP connection, build from client to server.**

# Lines

- This being the Internet, lines are terminated with `\r\n` (carriage-return line feed, aka control-M control-J, aka 0x0d, 0x0a).
- “man ascii”
  - Just to be annoying, Unix convention is `\n`, DOS convention is `\r`.



# Example

- We are examining the behaviour of the command:
  - `curl --http1.1 --no-alpn -6 -v https://tcpdynamics.uk`
- The extra options are in order to force the (simpler, easier to follow by eye) HTTP 1.1. HTTP 2 is negotiated to provide support for multiple streams and a variety of other optimisations.

# HTTPS vs HTTP

- HTTP (port 80) talks HTTP (amazingly).
- HTTPS(port 443) talks HTTP over an immediately-negotiated TLS session.
  - TLS is arguably outside scope of this course, but in 2023 it is vital: I will try to find time to talk about it

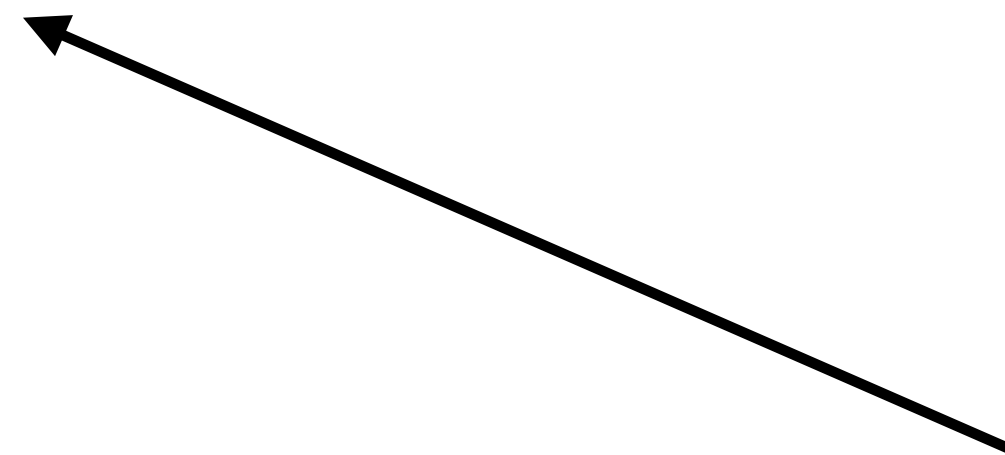
# First, TLS is set up

```
igb@Downstairs-Imac:~$ curl -6 -v --http1.1 --no-alpn https://tcpdynamics.uk
* Trying [2001:19f0:5001:32f3:5400:3ff:fe07:2a1c]:443...
* Connected to tcpdynamics.uk (2001:19f0:5001:32f3:5400:3ff:fe07:2a1c) port 443 (#0)
* (304) (OUT), TLS handshake, Client hello (1):
* CAfile: /etc/ssl/cert.pem
* CApath: none
* (304) (IN), TLS handshake, Server hello (2):
* (304) (IN), TLS handshake, Unknown (8):
* (304) (IN), TLS handshake, Certificate (11):
* (304) (IN), TLS handshake, CERT verify (15):
* (304) (IN), TLS handshake, Finished (20):
* (304) (OUT), TLS handshake, Finished (20):
* SSL connection using TLSv1.3 / AEAD-AES256-GCM-SHA384
* Server certificate:
* subject: CN=tcpdynamics.uk
* start date: Sep 14 19:25:55 2023 GMT
* expire date: Dec 13 19:25:54 2023 GMT
* subjectAltName: host "tcpdynamics.uk" matched cert's "tcpdynamics.uk"
* issuer: C=US; O=Let's Encrypt; CN=R3
* SSL certificate verify ok.
```



# What is sent

```
> GET / HTTP/1.1  
> Host: tcpdynamics.uk  
> User-Agent: curl/8.1.2  
> Accept: */*  
>
```



Note there is a blank line here

# Line formats

- The **is no colon** in the GET/POST/etc command line (and in the status response to it).
- There **is a colon** in the remaining option lines.
  - Format taken from email headers
  - Key: Value
  - Lines starting with whitespace are continuations of the previous line. Yes, this is a pain to code.

# GET /index.html HTTP/1.1

- Operation (others are POST, the less used PUT, the far less used DELETE, the sometimes useful HEAD)
- Name of resource to be fetched. Note, does **not** include hostname: dates back to era when there was a 1:1 relationship between IP numbers and name-spaces
- Note that this isn't the (obsolete) HTTP 1 protocol.
- There is also a HTTP/2 which is more complex in timing, but essentially the same syntax and concepts.



# There's More

- Optional, but useful: sometimes a server will send different content based on its knowledge of the capabilities of the client.
- Remember, people can lie (most browsers have a debug option to allow you to set pretty well any User-Agent field).

User-Agent: curl/8.1.2

# Host: `tcpdynamics.uk`

- Indicates the namespace, and allows multiple virtual servers to live on one IP number
- Historically, HTTP required one IP number per namespace, which clearly isn't going to end well for (eg) Wordpress — even if they had enough IP numbers, there are limits to how many you can stick on one interface
- By passing the hostname in the header, you can use the same IP address for an arbitrary number of names and use the name to switch the content
  - And as we will see, you can also use the hostname to do additional magic in proxies

# Accept: \*/\*

- I will accept any content type (cf. `text/html`).



# What comes back?

```
< HTTP/1.1 200 OK
< Server: nginx/1.18.0
< Date: Thu, 16 Nov 2023 11:12:36 GMT
< Content-Type: text/html
< Content-Length: 288
< Last-Modified: Fri, 29 Oct 2021 16:19:48 GMT
< Connection: keep-alive
< ETag: "617c1f24-120"
< Strict-Transport-Security: max-age=31536000; includeSubDomains; preload
< Accept-Ranges: bytes
<
```

# HTTP/1.1 200 OK

- Only compulsory element
- List of error codes taken from (essentially) SMTP and previously FTP
- See also 404 Not found, 403 Forbidden, 451 censored, etc
- 1xx informational response – the request was received, continuing process
- 2xx successful – the request was successfully received, understood, and accepted
- 3xx redirection – further action needs to be taken in order to complete the request
- 4xx client error – the request contains bad syntax or cannot be fulfilled
- 5xx server error – the server failed to fulfil an apparently valid request

# Useful Commentary

- Case can be made that both are revealing information of possible use to attacker, although (a) it's trivial to fingerprint servers and (b) everyone synch's their clock these days.

< Server: nginx/1.18.0

< Date: Thu, 16 Nov 2023 11:12:36 GMT



# Content Types and Length

- Type allows client to deal with content appropriately (in this case, hint to render it as HTML)
- Content-Length can optimise subsequent `read()`
  - Might be missing if length unknown prior to sending

Content-Type: text/html  
Content-Length: 288

# End of File: Nightmare

- FTP opens a separate TCP connection for each file (using a “control connection” to orchestrate them)
- File can then be sent unmodified (“BINARY MODE”) followed by closing the connection to mark its end.
  - Inefficient, complex, hard on the network
- SMTP, POP3, others use a lone “.” on a line on to mean “end of file”, with an extra dot put onto lines starting with a dot.
  - Means sender and receiver have to scan whole file, and inserting a single character breaks the flow of “read block, send block”.
  - Prevents protocol from being “8 bit clean” as it has to avoid sequence “\r\n.\r\n” at all costs.
  - One of the most broken things about existing email standards.
- Byte count (IMAP, HTTP) allows sender to stat() a file and then read it as one operation and send it as one operation, and allows receiver to just route next *n* bytes to the client software. HTTP doesn’t mandate \r\n for data, and high performance IMAP servers store messages with \r\n so they can do the same “stat(), open(), read()” trick.

# Cache Control, Ranges

- Last-Modified and ETag helpful for caching (storing copies of someone else's content)
  - Etag is opaque fingerprint of exact contents, which is a stronger check than comparing the last-modified string.
- Accept-Ranges allows a client to ask for sections of the file, by byte

```
< Last-Modified: Fri, 29 Oct 2021 16:19:48 GMT  
< Connection: keep-alive  
< ETag: "617c1f24-120"  
< Accept-Ranges: bytes
```

# Keep Alive / Re-use Control

- Indicates to client that the connection can be re-used for more requests once this one has completed
- Saves the building of a fresh connection and has performance benefits we will talk about later
- Early browsers built a fresh TCP connection for every item on a webpage: catastrophic performance

```
< Last-Modified: Fri, 29 Oct 2021 16:19:48 GMT  
< Connection: keep-alive  
< ETag: "617c1f24-120"
```

# Other stuff

- HSTS extension: says that this namespace is only available from HTTPS encrypted HTTP, and all requests for the unencrypted form should be rewritten to the encrypted version
- Ignored unless sent over HTTPS, cached for max-age seconds (here 365 days) by client
- Can be “pre-loaded”
- Use this whenever possible: all content should be encrypted to ensure authenticity

**Strict-Transport-Security: max-age=31536000; includeSubDomains; preload**



# Then the content

```
< Accept-Ranges: bytes      Note blank line
<
<html>
<head><title>You need to choose a file</title></head>
<body>
You need to specify a file to download.  Please consult the exercise description.
<p>
There will
be more help available in the Wednesday face to face, the Friday Zoom or the Friday
face to face at 3pm.
</body>
</html>
```

# Flexibility

- Trivially, the requested name can be a file in a directory rooted somewhere in the filestore.
- But can just as easily be a key into a database, a parameter to a program, whatever.
- The server converts URNs into content however it wants (and, presumably, the user expects).

# By convention

- You can pass multiple parameters with
  - `/some/path?var1=value1&var=value2`
- But note this is just a convention (“CGI”, Common Gateway Interface) and nothing stops you turning URL `/foo/bar/baz` into a call to `foo` with arguments `bar` and `baz`.
- Passing arguments via GET makes them trivially available in logs

# POST

```
curl -d foo=bar -v -v -6 https://www.batten.eu.org/index.html
```



```
> POST /index.html HTTP/1.1  
> User-Agent: curl/7.77.0  
> Host: www.batten.eu.org  
> Accept: */*  
> Content-Length: 7  
> Content-Type: application/x-www-form-urlencoded
```

# HEAD

```
HEAD / HTTP/1.1
Host: tcpdynamics.uk
User-Agent: curl/8.1.2
Accept: */*
```

Request sent,  
note blank line



```
HTTP/1.1 200 OK
Server: nginx/1.18.0
Date: Thu, 16 Nov 2023 11:26:44 GMT
Content-Type: text/html
Content-Length: 288
Last-Modified: Fri, 29 Oct 2021 16:19:48 GMT
Connection: keep-alive
ETag: "617c1f24-120"
Strict-Transport-Security: max-age=31536000; includeSubDomains; preload
Accept-Ranges: bytes
```

← Response Received



# Accept-Encoding

- Client can indicate that it can handle encodings (distinguish from encryption) of the content.
- `Accept-Encoding: deflate, gzip`
- Server can (at its discretion) respond with...
  - `Content-Encoding: gzip`
- One common trick to to store gzip'd versions of files and offer them direct from disk: saves I/O, CPU and network resources on server, client can probably decompress at effective wire speed.
- Compressing “on the fly” rarely worth it in 2023

# Languages

- Similarly for `Accept-Language` and `Content-Language`.
- And `Accept-Charset` and `Content-Charset`.
- More generally, client sends requests and capabilities, server indicates formats and content.

# Username and Passwords


- HTTP Basic Auth is a cesspit of security problems
- But, if we must...
- If there is no authentication in the request, it is rejected as follows:

`HTTP 401 Unauthorized`

`WWW-Authenticate: Basic realm="Some Tag"`

# Username and Passwords

- If you have a login and password for the domain, you send
  - `Authorization: Basic aWdiOmlnYjEyMw==`
  - Where does the gibberish come from?
  - Base64 encoding of username:password



```
igb@pi-one:~$ echo -n 'igb:igb123' | base64
aWdiOmlnYjEyMw==
igb@pi-one:~$
```

# HTTPS Only

- Obviously, don't do this over unencrypted channels!
- There is a challenge/response mechanism ("Digest", RFC 2069) using hashes, but...
  - It involves the server storing plaintext passwords.
  - Almost no-one implements it anyway.



# Cookies

- Another security cesspit.
- And wildly abused for a wide range of things they aren't suitable for.

# Cookies

- Any operation can return a “Set-Cookie” operation, which logs a key=value pair against (usually) the current domain and host.
- Whenever a request is made for that domain and host, all relevant cookies are sent with “Cookie” header.
- Cookies can have lifetimes, various security properties.

# Cookies

- One common use is/was to use HTTPS to protect username and password, returning a cookie for successful authentication, and then rely on that being safe to pass unencrypted to prove authentication has happened
  - HTTPS no longer expensive, and should be used for all operations, particularly...
  - ...those involving cookies
- More generally, anything which involves storing client-side state: navigation, shopping carts, etc
- Best limited to a random session-id and nothing else, with everything else stored server-side.

# Caching

- Rises and falls in popularity
- Idea is that you can keep a copy of “all the internet that matters” at the edge of a **consuming** network
  - Ian F brought the [cs.bham.ac.uk](http://cs.bham.ac.uk) caching strategy to [ftel.co.uk](http://ftel.co.uk); when I came back, I brought its end
- Alternatively, a more usefully, you keep a copy of “all our content from lots of slow machines” at the edge of a **producing** network
  - Or in the cloud, which is what Cloudflare do
- Caches and proxies tend to be the same boxes: we talked about proxying when we talked about NAT.

# Caching

- Usually done with special software (“squid” is the most common) or with special appliances.
- Trick is to cache stuff which is static and used repeatedly, while not interfering with content that changes rapidly or is rarely accessed.
- My gut feel is that 2023-stylee, all outbound caching is out of favour — the internet is too big, and bandwidth is cheap
- Inbound caches...let's talk.



# Simple Graphing

[Click here for standard URL](#) [Click here for uncached URL](#)

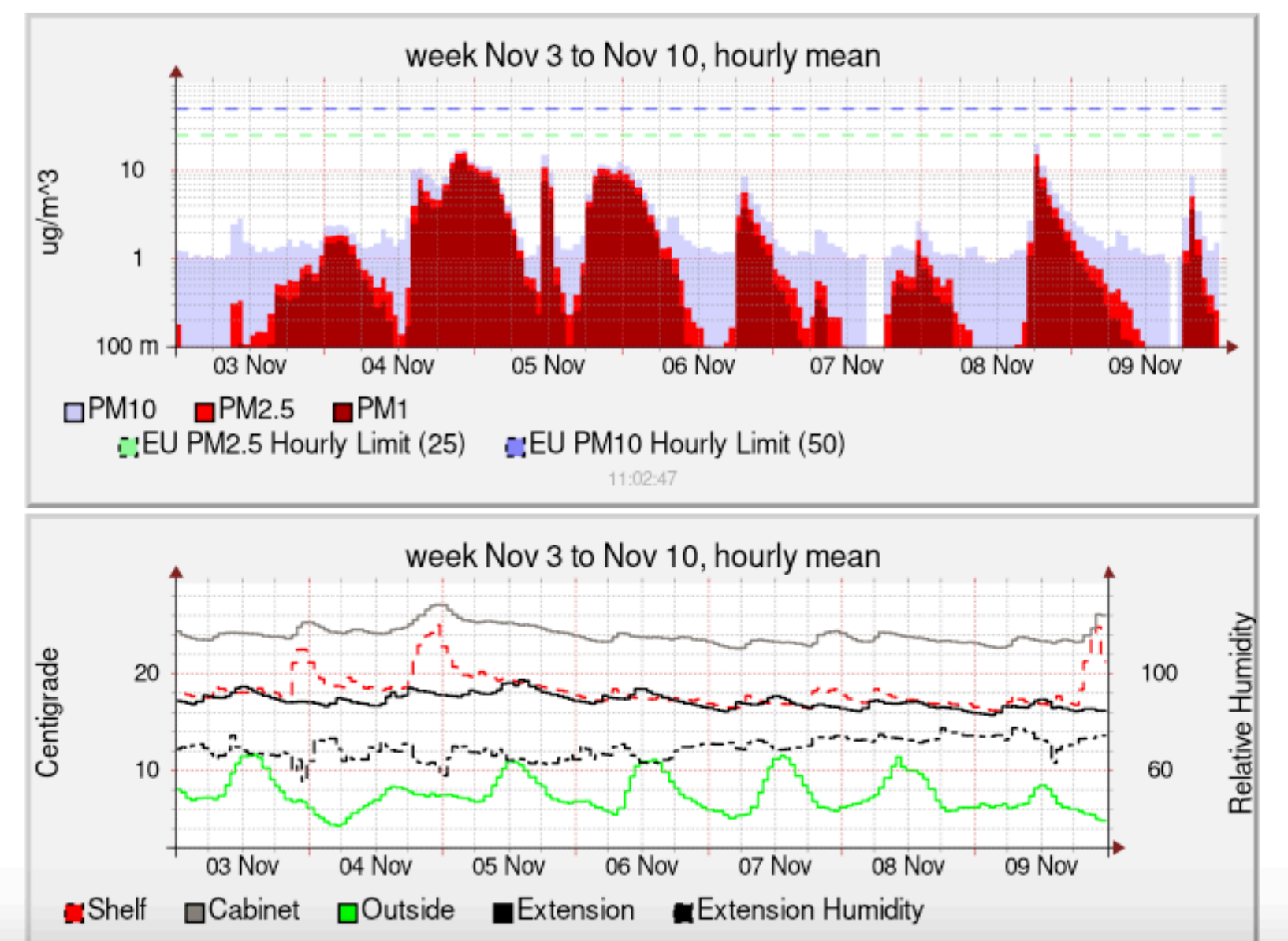
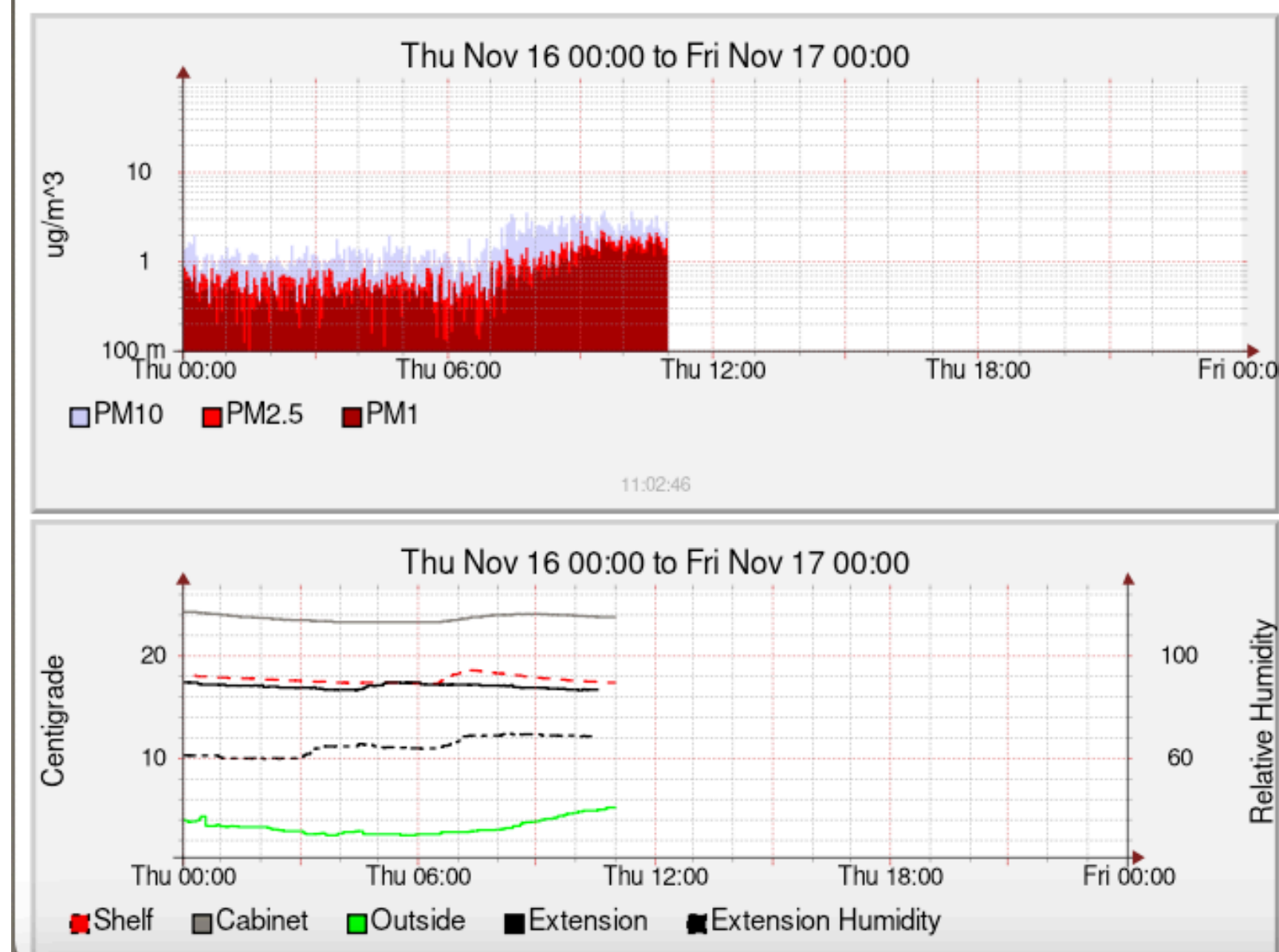
Last Update: Thu Nov 16 11:02:46 2023

PM1	PM2.5	PM10
2	2	6

Inside	Humidity
18.4C	53.6%

Outside
5.2C

Shelf	Cabinet	Fireplace	Extension
17.4C	23.8C	17.6C	16.7C



# Simple Graphing

```
38 <div class="column">
39
40 <br/>
41 
42 <br/>
43 
44 <br/>
45
46 <br/>
47 
48 <br/>
49 
50 <br/>
51
52 <br/>
53 
54 <br/>
55 
56 <br/>
57
58 <br/>
59 
60 <br/>
61 
62 <br/>
63
64 </div>
65
```



# Simple Graphing

```
def do_GET(self):
    """handle an incoming request"""
    self.close_connection = False # pylit
    p = urlparse(self.path)
    status, errortext = 500, "unknown internal error"

    try:
        if p.path in ("/", "", "/index.html"):
            status, body, bodytype, cache = self.__make_index()
        elif p.path.startswith("/graph/"):
            elems = p.path.split("/")
            status, body, bodytype, cache = self.__make_graph(
                graphtype=elems[2], endtime=elems[3]
            )
        else:
            raise FileNotFoundError(p.path)
    except:
```

```
<br/>

<br/>

<br/>

<br/>

<br/>

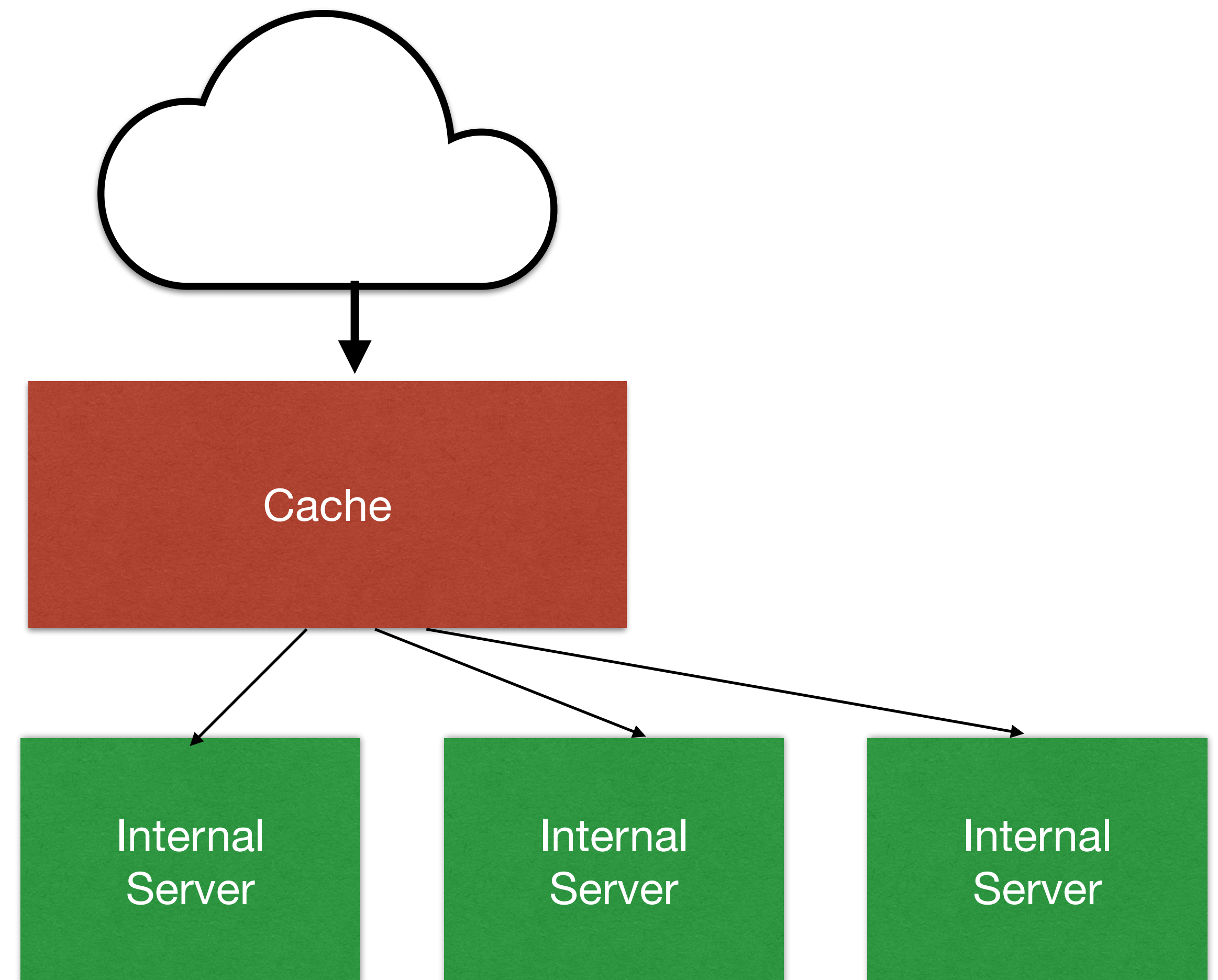
<br/>

<br/>

<br/>
```

# Inbound caches, aka reverse proxies

- Inbound caches are operated at the edge of your network to handle **incoming** requests
- Cache might be useful, but computers are fast
- Main attraction is single point of contact, single request parser, single checker, possibly single application of certificates (wildcards should not be widely distributed)
- I have things like hand-written servers, non-https capable devices and devices with self-signed certificates sat behind mine.
- Heavily used in businesses to organise multiple systems into one namespace (you can route by presented hostname or by presented URL)



# Things you can do with inbound caches and proxies

```
server {
    listen 443 ssl http2 ;
    listen [::]:443 ssl http2 ;
    server_name ~^(gs1900-24e-[0-9]+|gs1900-10hp-[0-9]+).(batten.eu.org|batten-family.uk|batten-family.org.uk)$ ;

    location / {
        proxy_pass https://$1.$2;
    }
}
```

```
server {
    listen 443 ssl http2 ;
    listen [::]:443 ssl http2 ;
    server_name pmgraph.batten.eu.org pmgraph.batten-family.uk pmgraph.batten-family.org.uk ;

    location / {

        # force resolution with built-in resolver at run time
        resolver 10.92.213.1 ipv6=off;
        set $pisix "pi-six.batten.eu.org";
        proxy_pass http://$pisix:8080 ;
        expires off;

        proxy_http_version 1.1;
        proxy_buffering on;
        proxy_cache STATIC;
        proxy_cache_valid 200 10m;
    }
}
```

