The Hyper Text Transfer Protocol (HTTP)

Christian Grothoff

8.6.2018

Agenda

The Hyper Text Transfer Protocol (HTTP)

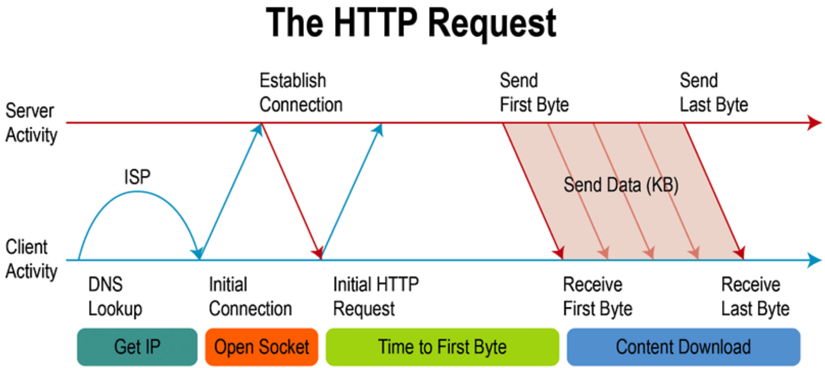
- Initially standardized in RFC 2616
- HTTP/0.9 (1990), HTTP/1.0 (1996), HTTP/1.1 (1999), HTTP/2 (2016)
- Runs over TCP (port 80) or as HTTPS over TLS (port 443)

Uniform Resource Locators (URLs)

http://www.example.com:80/path?key=value#anchor

PROTOCOL://HOST:PORT/PATH?QUERY#FRAGMENT

Anatomy of an HTTP request



HTTP 1.x Request Format

```
GET / HTTP/1.0
Key1: value1
Key2: value2
Key3: value3
  value3 may be continued here
Key4: value4
```

- Each line SHOULD be terminated by CRLF, but MAY be terminated only by CR or LF.
- The header ends with an empty line by itself.
- HTTP does not specify a maximum header length

HTTP Headers

HTTP headers are used in many ways:

- control the connection (Keep-alive)
- control caching
- provide meta data (content-length, content-type, content-encoding)
- request and provide authentication

HTTP knows four types of headers:

- General header: can be used in both request and response
- Request header: only applicable to request messages
- Response header: only applicable to response messages
- Entity header: define meta-information about the body

Exercise 1: HTTP/1.0, GET

```
$ telnet grothoff.org 80
GET / HTTP/1.0
```

HTTP Methods (or `verbs`)

`GET` is just one HTTP method. Other common HTTP/1.0 methods include:

- HEAD
- PUT
- POST
- OPTIONS
- PUT
- DELETE
- TRACE
- CONNECT

HTTP Methods: Safety and Idempotence

Method	Description	Idempotent	Safe
GET	Fetch resource	✓	✓
HEAD	Fetch header only	✓	✓
PUT	Store entity	✓	✗
POST	Accept entity as subordinate	✗	✗
OPTIONS	Return supported HTTP methods	✓	✓
DELETE	Delete resource	✓	✗
PATCH	Change resource	✗	✗
TRACE	Echo request back to client	✓	✓
CONNECT	Convert connection to tunnel		

Exercise 2: HTTP/1.0 HEAD

```
$ telnet grothoff.org 80
HEAD / HTTP/1.0
```

- What happens if you use "HTTP/1.1" instead of "HTTP/1.0"?

HTTP Responses

A HTTP response generally consists of three parts:

1. HTTP Status code line (version, numeric status code, human readable status code)
2. HTTP (response) headers, followed by empty line
3. HTTP response body

HTTP 1.x Response Format

```
HTTP/1.1 200 OK
Server: some advertisement
Date: Sun, 31 Aug 1999 24:00:00 GMT
Content-Type: text/html
Content-Length: 11
Connection: close

Hello World
```

All of the above headers are technically optional.

HTTP Status Codes

The numeric range of the HTTP status code is already meaningful:

1. Informational 1xx: Indicate a provisional response
2. Successful 2xx: Indicate that the client request was successful
3. Redirection 3xx: Indicates that further action is needed
4. Client Error 4xx Indicates when the client seems to have erred
5. Internal Server Error 5xx: Indicates cases in which the server is aware that it has erred

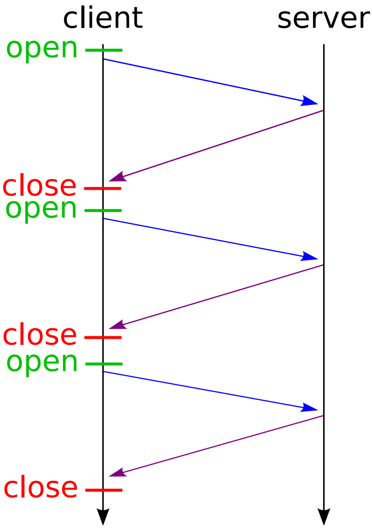
Common HTTP Status Codes

100	Continue
200	Ok
301	Moved Permanently
304	Not Modified
400	Bad Request
401	Authentication Required
402	Payment Required
403	Forbidden
404	Not Found
500	Internal Server Error

Exercise 3: HTTP/1.1

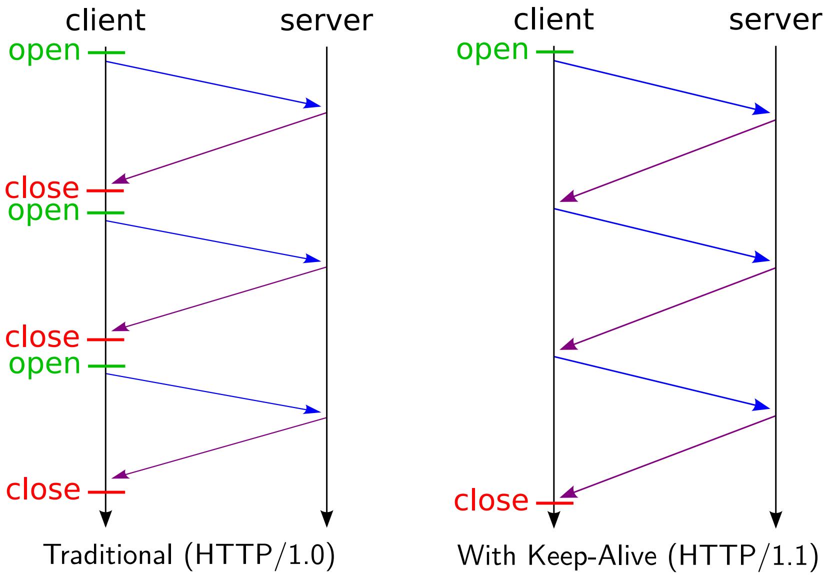
```
$ telnet grothoff.org 80
GET / HTTP/1.1
Host: grothoff.org
```

Multiple HTTP requests



Traditional (HTTP/1.0)

Multiple HTTP requests



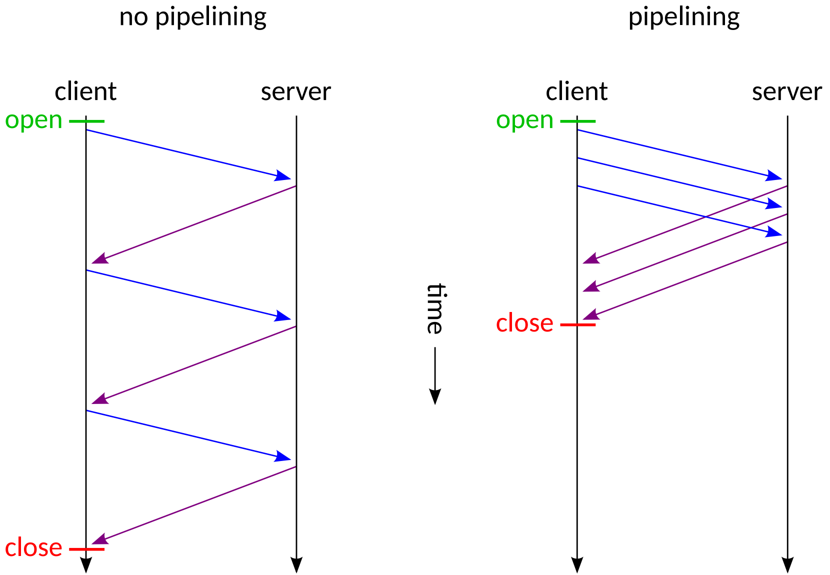
Exercise 4: HTTP/1.1, Connection: close

```
GET / HTTP/1.1
Host: grothoff.org
Connection: close
```

Exercise 5: HTTP/1.0, Connection: Keap-alive

```
GET / HTTP/1.0
Connection: Keep-alive
```

HTTP/1.1 pipelining



HTTP/1.1 Response length

- `Content-Length` header defines body length
- `Content-encoding: chunked` provides alternative if length not known
- Otherwise, no keep-alive possible (`Connection: close` header implied)

Content-encoding: chunked

RFC 2616, section 3.6.1 defines chunked encoding:

```
Chunked-Body     = *chunk
                   last-chunk
                   trailer
                   CRLF
chunk            = chunk-size [ chunk-extension ] CRLF
                   chunk-data CRLF
chunk-size       = 1*HEX
last-chunk       = 1*("0") [ chunk-extension ] CRLF
chunk-extension= *( ";" chunk-ext-name [ "=" chunk-ext-val ]
chunk-ext-name = token
chunk-ext-val    = token | quoted-string
chunk-data       = chunk-size(OCTET)
trailer          = *(entity-header CRLF)
```

Long polling

HTTP may generate a response incrementally,

With or without chunked encoding

Request $\rightarrow$ Response $\Rightarrow$ Request $\rightarrow$ Response, [wait, Response]*, fin.

HTTP Benchmarking

Web performance is complex:

- number of requests required per Web page in total
- parallel TCP connections used by browser
- static content vs. dynamic content generation
- impact of caching, proxies, network speed
- HTTP vs. HTTPS
- Use of “Connection: Keep-alive”
- Browser HTML parsing and rendering

We will focus on a few simple tools for the <u>server</u>.

Exercise 6: siege

`apt-get install siege`

```
$ siege -t5S http://grothoff.org/
```

```
Transactions:          876 hits
Availability:       100.00 %
Elapsed time:         4.64 secs
Data transferred:         0.17 MB
Response time:          0.01 secs
Transaction rate:       188.79 trans/sec
Throughput:         0.04 MB/sec
Concurrency:          2.22
Successful transactions:          878
Failed transactions:             0
Longest transaction:          0.05
Shortest transaction:          0.00
```

Exercise 7: Apache Benchmark (ab)

`apt-get install apache2-utils`

```
$ ab -c 25 -t5 http://grothoff.org/
```

Exercise 7: Apache Benchmark (ab)

```
apt-get install apache2-utils
```

```
$ ab -c 25 -t5 http://grothoff.org/
```

```
Time taken for tests:   5.000 seconds
Complete requests:      14096
Failed requests:        0
Non-2xx responses:      14096
Total transferred:      5300096 bytes
HTML transferred:       2607760 bytes
Requests per second:    2819.09 [#/sec] (mean)
Time per request:       8.868 [ms] (mean)
Time per request:       0.355 [ms] (mean, across all concurrent requests)
Transfer rate:          1035.14 [Kbytes/sec] received
```

Benchmarking

Lesson learned:

- HTTP servers are very fast
- You may be benchmarking the client
- You may be benchmarking the bandwidth
- You may be benchmarking the network latency

HTTP Caching

HTTP response headers control how long a resource is valid:

- `Cache-control: max-age=3600`
- `Expires: Mon, 31 Aug 2020 00:00:00 GMT`
- `ETag: "727285929572e8a"` — assign unique ID to resource

HTTP request headers can be used to inquire if a resource changed:

- `If-Modified-Since: Mon, 31 Aug 2000 00:00:00 GMT`
- `If-None-Match: "727285929572e8a"`

HTTP Methods & Caching

Method	Description	Cacheable
GET	Fetch resource	✓
HEAD	Fetch header only	✓
PUT	Store entity	✗
POST	Accept entity as subordinate	✓(*)
DELETE	Delete resource	✗
PATCH	Change resource	✗
TRACE	Echo request back to client	✗
CONNECT	Convert connection to tunnel	✗

(*) Only if HTTP response includes explicit freshness information.

Cookies

HTTP is a “stateless” protocol. Cookies are a mechanism to add state.

Server to client:

```
Set-Cookie: key=value;OPTIONS
```

Client to server:

```
Cookie: key=value
```

Cookie options

- Expires=DATE — if not set, cookies expire at the end of the session
- Domain=DOMAIN — for which (sub)domain does the cookie apply
- Path=PATH — for which URL paths should the cookie be sent
- Secure — only send cookie over HTTPS
- HttpOnly — only send cookie over HTTP
- SameSite=Strict — do not send along cross-site requests

Range queries

HTTP supports incremental downloads:

```
GET / HTTP/1.1
Host: grothoff.org
Content-range: 40-42/bytes

206 Partial Content
Content-length: 3
Accept-ranges: bytes
Content-range: 40-42/64
```

HTTP/1.x supports body compression

```
GET / HTTP/1.0
Accept-encoding: gzip,deflate

200 OK
Content-encoding: gzip
Content-length: 42
```

The content length is that of the compressed body.

HTTP Requests: Methods with Bodies in Request

Method	Description	Body
GET	Fetch resource	✗
HEAD	Fetch header only	✗
PUT	Store entity	✓
POST	Accept entity as subordinate	✓
DELETE	Delete resource	✗
PATCH	Change resource	✓
TRACE	Echo request back to client	✗
CONNECT	Convert connection to tunnel	✓

100 Continue

Uploading a body may be expensive! HTTP can check if the HTTP server is willing to handle it first!

```
POST / HTTP/1.1
Host: grothoff.org
Content-length: 1000
Expect: 100-continue

100 Continue

UPLOAD-BODY

200 Ok

RESPONSE-BODY
```

100 Continue

Uploading a body may be expensive! HTTP can check if the HTTP server is willing to handle it first!

```
POST / HTTP/1.1
Host: grothoff.org
Content-length: 1000
Expect: 100-continue

100 Continue

UPLOAD-BODY

200 Ok

RESPONSE-BODY
```

```
POST / HTTP/1.1
Host: grothoff.org
Content-length: 1000
Expect: 100-continue

417 Expectation Failed

ERROR-BODY
```

HTTP Upgrade

- HTTP includes a mechanism to “upgrade” or switch to another protocol
- The client requests the upgrade using the `Connection` header
- The client offers one or more protocols to upgrade to
- The server replies with which protocol it wants to use
- Afterwards, the underlying TCP stream is used bi-directionally for the new protocol

HTTP Upgrade: Web Sockets

```
GET / HTTP/1.0
Host: example.com
Connection: Upgrade
Upgrade: WebSocket
Sec-WebSocket-Key: HEXCODE==
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 13

HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: HEXCODE=
Sec-WebSocket-Protocol: chat

WEBSOCKET V13.
```

HTTP Upgrade: HTTP/2

```
GET / HTTP/1.0
Connection: Upgrade
Upgrade: h2c

HTTP/1.1 101 Switching Protocols
Upgrade: h2c

HTTP2 IN CLEARTEXT.
```

Virtual hosting

- There are only $\approx$ 4 billion IPv4 addresses
- We may not have one for every Web server
- We also may not have a physical machine for every domain

$\Rightarrow$ Goal: allow one IP to host many HTTP domains

Problem: HTTP server needs to know which domain is requested!

Virtual hosting

- There are only $\approx$ 4 billion IPv4 addresses
- We may not have one for every Web server
- We also may not have a physical machine for every domain

$\Rightarrow$ Goal: allow one IP to host many HTTP domains

Problem: HTTP server needs to know which domain is requested!

Solution: HTTP/1.1 mandates `Host:` header to indicate domain.

Sample Apache configuration (sites-enabled/)

```
<VirtualHost my-domain.com:80>
        ServerAdmin webmaster@my-comain.com
        ServerName "my-comain.com"
        DocumentRoot /var/www/my-domain/
        <Directory />
                Options FollowSymLinks
                AllowOverride None
        </Directory>
        <Directory "/var/www/my-domain">
                AllowOverride None
                Order allow,deny
                Allow from all
        </Directory>
</VirtualHost>
```

HTTP servers can act as proxies

This is called a reverse proxy:

```
<VirtualHost my-domain.com:80>
        ProxyPass /foo/ http://localhost:58080/
        ProxyPass /bar/ https://localhost:58081/
        ProxyPass /bfh/ https://bfh.ch/
        ProxyPass /ws/ ws://localhost:4242/
</VirtualHost>
```

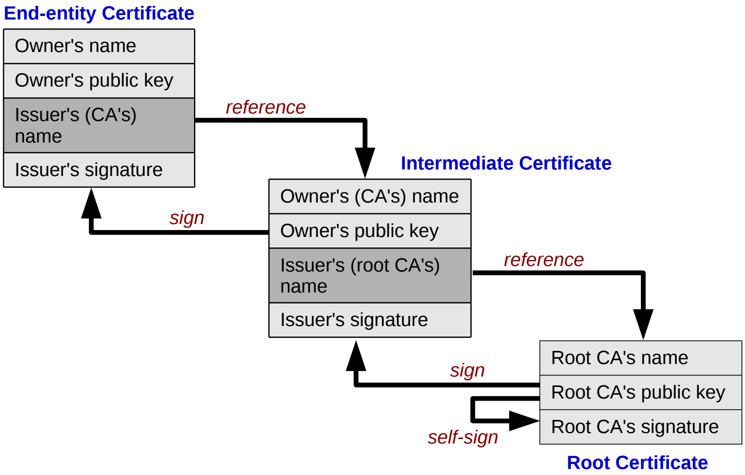
This is in contrast to an HTTP client using a proxy (such as Squid, Tor or WWWOFFLE).

Exercise 8: Reverse proxy

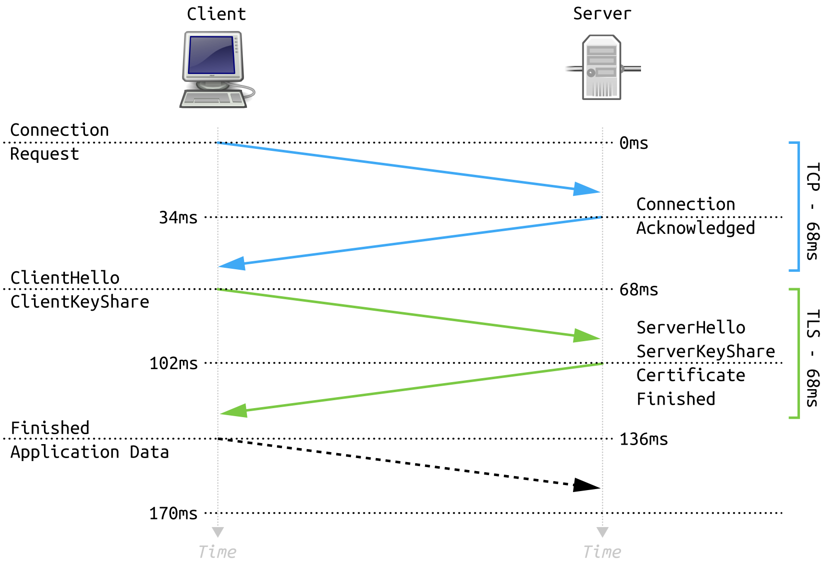
- Configure an Apache server for your site
- Redirect a particular path to another HTTP server
- Redirect another (virtual) domain to your another HTTP server

Hint: use `/etc/hosts` to map the IP address(es) if you do not have sufficient control over DNS!

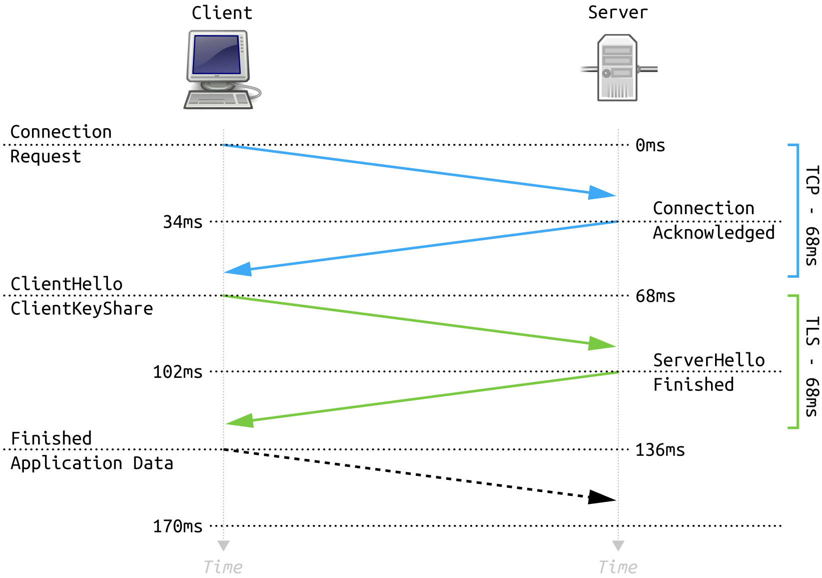
X.509 Trust Chains



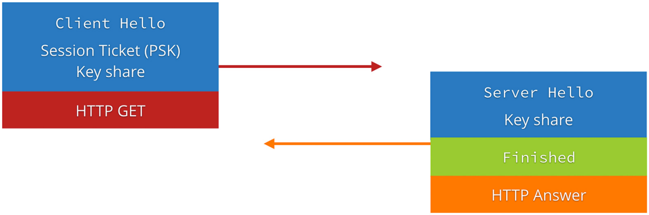
TLS 1.3: Full Handshake



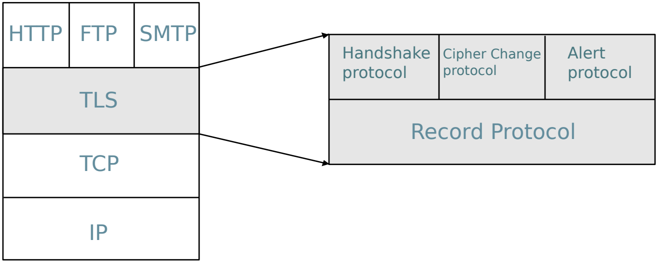
TLS 1.3: Abbreviated Handshake



TLS 1.3: 0.5 RTT Handshake



TLS Protocol Stack



Exercise 9: Enable TLS

- Obtain a TLS certificate via the "Let's encrypt" CA (you need a global DNS name!):

```
# letsencrypt -D DOMAIN.TLD --standalone certonly  # or
# letsencrypt -D DOMAIN.TLD --standalone run # may work
```

- Configure your Apache server to use it:

```
SSLEngine on
SSLProtocol -ALL +TLSv1.2 +TLSv1.1 +TLSv1
SSLCertificateKeyFile /etc/letsencrypt/live/example.com/privkey.pem
SSLCertificateChainFile /etc/letsencrypt/live/example.com/fullchain.pem
SSLCertificateFile /etc/letsencrypt/live/example.com/cert.pem
```

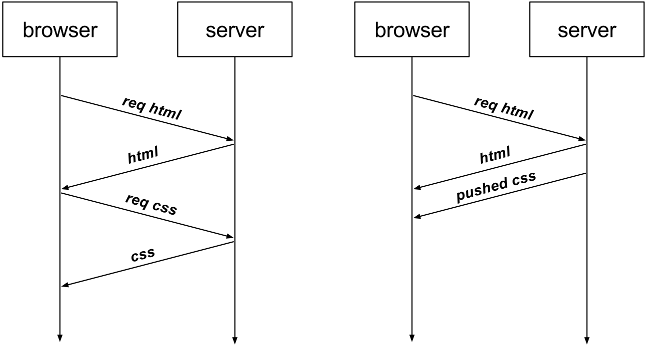
- Verify your configuration using `https://www.ssllabs.com/ssltest/` and `https://observatory.mozilla.org/`

HTTP/2

Key changes:

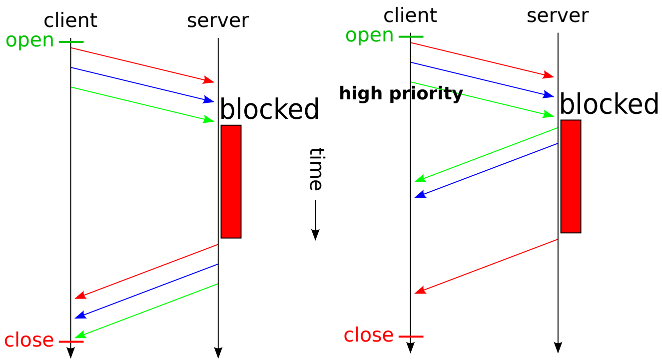
- HTTP/1 is stateless. HTTP/2 is stateful.
- HTTP/1 is human readable. HTTP/2 is binary.
- HTTP/1 is in cleartext. HTTP/2 browsers today require TLS.
- HTTP/1 is reactive. HTTP/2 servers can be proactive.
- HTTP/1 handled requests in order. HTTP/2 allows out of order.
- HTTP/1 is mature. HTTP/2 was rushed to avoid fragmentation.

HTTP/2 Push



Without Push **With Push**

HOL blocking and prioritization



Exercise 10: Enable HTTP/2 for Apache

First, enable the HTTP/2 module:

```
# a2enmod http2
```

Then, enable HTTP/2 for your site:

```
<VirtualHost *:443>
  Protocols h2 http/1.1
  ServerAdmin admin@example.com
  ServerName examp.e.com
  ...
</VirtualHost>
```

Exercise 11: Putting it all together

- Configure your site for HTTPS
- Enable HTTP/2
- Reverse proxy to another HTTP server
- Add `Link:` headers to add PUSH support:

```
Link: </assets/styles.css>;rel=preload
```

RTFL