The Hyper Text Transfer Protocol (HTTP)

Christian Grothoff

Part I

The Hyper Text Transfer Protocol (HTTP)

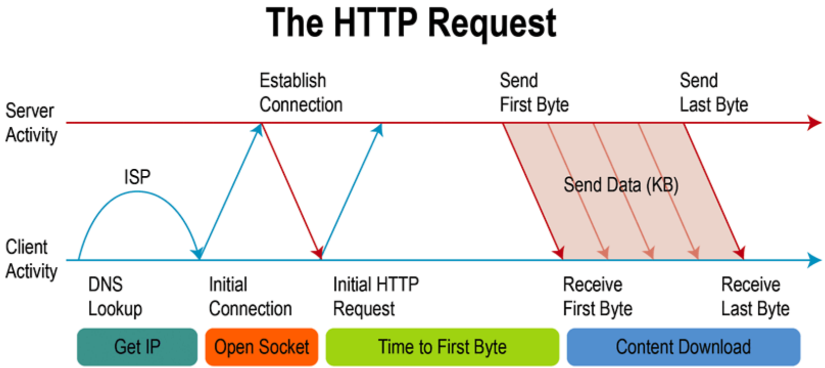
- Initially standardized in RFC 2616
- HTTP/0.9 (1990), HTTP/1.0 (1996), HTTP/1.1 (1999), HTTP/2 (2016)
- Runs over TCP (port 80) or as HTTPS over TLS (port 443)

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

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 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

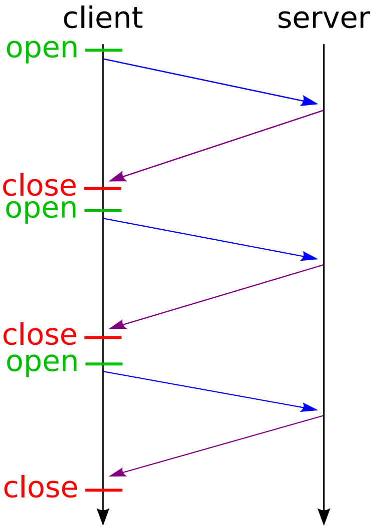
Common HTTP Status Codes

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

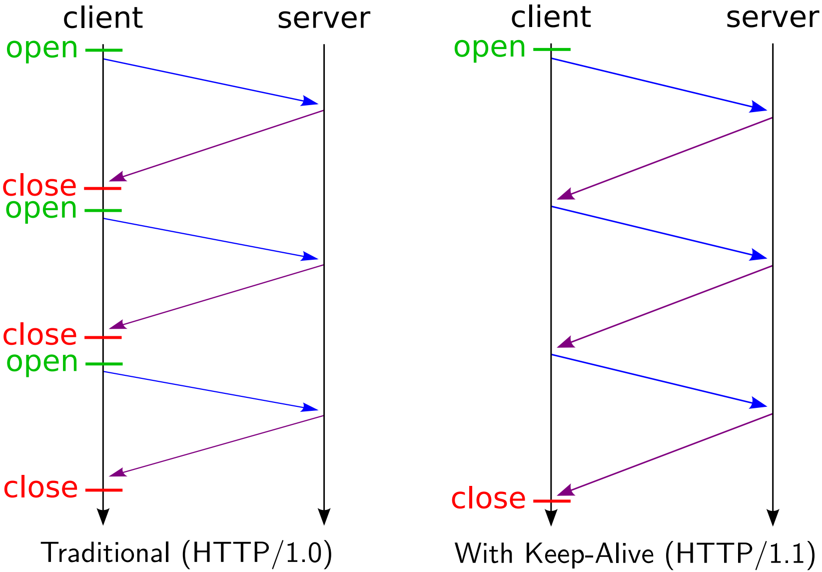
```
$ telnet grothoff.org 80
GET / HTTP/1.1
Host: grothoff.org
```

Multiple HTTP requests



Traditional (HTTP/1.0)

Multiple HTTP requests



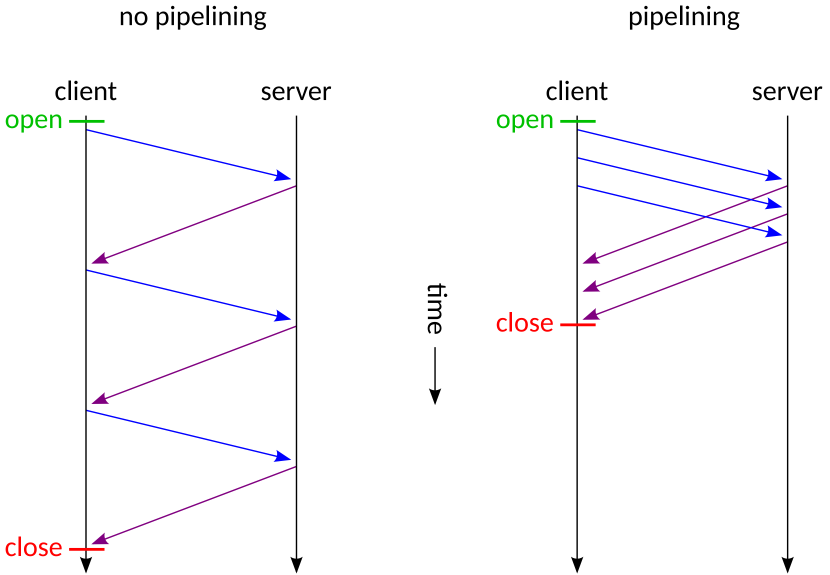
Exercise 4: HTTP/1.1, Connection: close

```
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 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

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 and Mobile Systems?

- HTTP(S) is the new IP — tunnel traffic over HTTP
- HTTP + HTML are a fast and portable way to create a GUI
- Full-blown HTTP Servers (Apache, etc.) are often overkill
- Extending Apache only natural with P-languages

Need a lightweight way to create HTTP servers!

GNU libmicrohttpd

- Free software (GNU LGPL or GPL + eCoS)
- Fully HTTP/1.0 and HTTP/1.1 compliant
- Supports all common HTTP features
- Just HTTP(S) server, small footprint
- Makes limited assumptions about event handling:
  - External select/poll loop
  - Internal select/poll loop
  - One thread per connection
  - Thread pool

MHD: Security

- Optional support for HTTPS, full X.509 support
- HTTP basic and digest authentication
- Access to client certificates
- Ability to selectively `bind` sockets
- Limiting # connections (overall, per IP), custom timeouts
- Limit memory consumption per connection
- Did very well in three independent external security audits

MHD: Performance

- No busy waiting, ever
- Zero copy, wherever possible
- Stream processing (GET, POST, PUT)
- Minimize `malloc`, handle **all** errors
- No re-inventing `strchr`, `strcmp`, etc.
- Clean C code, no code duplication

MHD: Scales up and down!

- Library binary can be as small as 32k
- We reportedly have users on systems with 50 Mhz processors with HTTPS
- We have users working with MHD on systems with 64 kb RAM

“I also ran oprofile on the system while streaming about 7gbps to (simulated) ipads and while ramping up 1000s of streams (which causes high rate of HTTP requests to read the Apple HLS playlists). libmicrohttpd barely registers as cpu usage.” – MHD user

Applications using MHD

- GNUnet, P4P Portal
- Gnome Music Player Client, Kiwix, XMBC, OpenVAS
- Psensor, Disk Nukem, Flat8, Fawkes, Conky, CallHome
- OpenDIAS, Techne, Cables communication project
- Open Lightning Architecture, OpenZWave, libhttpserver
- Plus many non-free applications (such as TVs, surveilance cameras, network appliances, etc.)

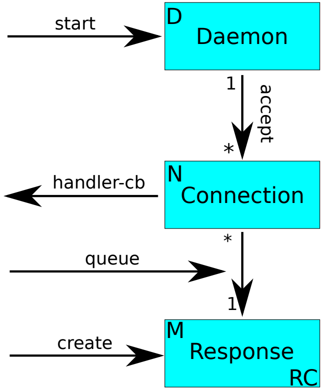
Exercise 6: Install MHD

```
$ wget https://ftp.gnu.org/gnu/libmicrohttpd/\
    libmicrohttpd-0.9.55.tar.gz
$ tar xvf libmicrohttpd-0.9.55.tar.gz
$ cd libmicrohttpd-0.9.55
$ ./configure --prefix=$HOME
$ make install
```

Exercise 7: Start MHD HTTPD

```
$ cd doc/examples/
$ gcc -I$HOME/include -L$HOME/lib \
    hellobrowser.c -lmicrohttpd -o hellobrowser
$ export LD_LIBRARY_PATH=$HOME/lib
$ ./hellobrowser  # in another shell
$ wget -q -O - http://localhost:8888/
```

The MHD API


start
D
Daemon
1
accept
*
handler-cb
N
Connection
*
queue
1
create
M
Response
RC

Launching MHD: The code

```
#include <microhttpd.h>

int main ()
{
  struct MHD_Daemon *daemon =
    = MHD_start_daemon (MHD_USE_AUTO | \
                        MHD_USE_INTERNAL_POLLING_THREAD,
                        8888,
                        NULL, NULL,
                        &answer_to_connection, NULL,
                        MHD_OPTION_END);
  if (NULL == daemon)
    return 1;
  (void) getchar ();
  MHD_stop_daemon (daemon);
  return 0;
}
```

Responding to requests: The code

```
static int
answer_to_connection (void *cls,
    struct MHD_Connection *connection,
    const char *url, const char *method,
    const char *version,
    const char *upload_data, size_t *upload_data_size,
    void **con_cls)
{
  const char *page
   = "<html><body>Hello, browser!</body></html>";
  int ret;
  struct MHD_Response *response
   = MHD_create_response_from_buffer (strlen (page),
       (void *) page, MHD_RESPMEM_PERSISTENT);
  ret = MHD_queue_response (connection, MHD_HTTP_OK,
                            response);
  MHD_destroy_response (response);
  return ret;
}
```

Exercise 8: Setting Response Headers

```
response = MHD_create_response (...);
MHD_add_response_header (response,
                         MHD_HTTP_HEADER_CONTENT_TYPE,
                         "text/html");
ret = MHD_queue_response (connection,
                          MHD_HTTP_OK,
                          response);
MHD_destroy_response (response);
```

Test it with `telnet`! Which headers does the response include?

MHD Response Generation

- Static buffer in memory
- Data stream (known or unknown size)
- Data stream with long polling
- From file at offset
- From file with `sendfile()`
- With custom HTTP headers — and trailers

Exercise 9: sendfile()

```
int fd;
struct stat sbuf;

if (0 != strcmp (method, "GET")) return MHD_NO;
if ( (-1 == (fd = open ("picture.png", O_RDONLY))) ||
     (0 != fstat (fd, &sbuf)) ) {
  if (fd != -1) close (fd);
  return report_error (connection);
}
struct MHD_Response *response =
  MHD_create_response_from_fd_at_offset (sbuf.st_size,
                                         fd, 0);
MHD_add_response_header (response,
                         "Content-Type", "image/png");
ret = MHD_queue_response (connection, MHD_HTTP_OK,
                          response);
```

Interlude: 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 server.

Excercise 10: 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
```

Excercise 11: Apache Benchmark (ab)

`apt-get install apache2-utils`

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

Excercise 11: 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

Exercise 12: Benchmark your server

- `top`
- `time` BINARY
- `strace -c` BINARY
- `iotop` (requires `root`)

MHD performance tuning

- MHD's fastest mode is a `thread pool` with `epoll()`
- You can re-use `struct MHD_Response` objects
- You can disable the "Date:" header (`MHD_USE_SUPPRESS_DATE_NO_CLOCK`)
- You could run HTTP over a UNIX domain socket
- You can enable TCP FASTOPEN (`MHD_USE_TCP_FASTOPEN`)
- You can enable crazy mode (`MHD_USE_TURBO`)
- You can disable logging (`--disable-messages`)

... but, in 99.99% of all cases, your bottleneck will be elsewhere without these!

HTTP/1.1 Responses

- `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)
```

Incremental replies with MHD

```
#include <stdlib.h>
static ssize_t crc (void *cls, uint64_t pos,
      char *buf, size_t size_max) {
  if (0 == size_max) return 0;
  if (0 == rand() % 1024 * 1024)
    return MHD_CONTENT_READER_END_OF_STREAM;
  *buf = 'b';
  return 1;
}

  struct MHD_Response *response
     = MHD_create_response_from_callback
        (MHD_SIZE_UNKNOWN,
         1024,
         &crc, NULL, NULL);
```

Exercise 13: Generating incremental replies

Using `telnet`:

- What happens if you use the code above with a HTTP/1.0-style request?
- What happens if you use the code above with a HTTP/1.1-style request?

Using `wget`:

- What is the output if you use the code above?
- What happens on the wire? Use `wireshark`!

Long polling

HTTP may generate a response incrementally:

- With or without chunked encoding
- `MHD_OPTION_CONNECTION_TIMEOUT` and `MHD_set_connection_option (connection, MHD_CONNECTION_OPTION_TIMEOUT)` can control timeout.
- `MHD_ContentReaderCallback` can return 0 to indicate "more available later"
- `MHD_suspend_connection()` can suspend handling of network data for a connection.

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

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.

```
const char *value
  = MHD_lookup_connection_value (connection,
                                 MHD_COOKIE_KIND,
                                 "key");
response = ...;
MHD_add_response_header (response,
                         MHD_HTTP_HEADER_SET_COOKIE,
                         "key=value;OPTIONS");
```

Modify your code to set cookies and print values of received cookies.

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
```

Range queries with MHD

MHD does not (yet) have build-in support, so you need to process range queries manually:

```
const char *range
  = MHD_lookup_connection_value (connection,
                                 MHD_HEADER_KIND,
                                 MHD_HTTP_HEADER_CONTENT_RANGE
response = ...;
MHD_add_response_header (response,
                         MHD_HTTP_HEADER_ACCEPT_RANGES,
                         "bytes");
```

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.

Compression in C

```c
#include <zlib.h>
/**
 * Try to compress a response body.  Updates @a buf and @a buf_size.
 *
 * @param[in,out] buf pointer to body to compress
 * @param[in,out] buf_size pointer to initial size of @a buf
 * @return true if buf was compressed
 */
int body_compress (void **buf, size_t *buf_size) {
  uLongf cbuf_size = compressBound (*buf_size);
  Bytef *cbuf = malloc (cbuf_size);
  int ret = compress (cbuf, &cbuf_size,
     (const Bytef *) *buf, *buf_size);
  if ((Z_OK != ret) || (cbuf_size >= *buf_size)) {
    free (cbuf); return false;    }
  free (*buf);
  *buf = (void *) cbuf;
  *buf_size = (size_t) cbuf_size;
  return true;
}
```

Exercise 14: Add compression support

- Add support for compression to your MHD server.
- Make sure to check the client supports compression.
- You need to link against `libz`

Part II

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
```

MHD and uploads

```
int
answer_to_connection (void *cls,
    struct MHD_Connection *connection,
    const char *url,
    const char *method,
    const char *version,
    const char *upload_data,
    size_t *upload_data_size,
    void **con_cls)
{
}
```

`answer_to_connection` will be called repeatedly!

Exercise 15: Handle uploads

- Write an HTTP server with an upload function
- Reject uploads larger than 8 MB

Why do many HTTP servers include such a limitation?

HTTP Basic authentication

```
#define DENIED "<html><body>Go away.</body></html>";

 char *pass = NULL;
 char *user
   = MHD_basic_auth_get_username_password (connection,
                                          &pass);
 if (! authentication_ok (user, pass)) {
   response =
     MHD_create_response_from_buffer (strlen (DENIED),
                                      (void *) DENIED,
                                      MHD_RESPMEM_PERSISTENT);
   return MHD_queue_basic_auth_fail_response (connection,
                                              "my realm",
                                              response);
 }
```

HTTP Digest authentication (Part I)

```
#define MY_OPAQUE_STR "11733b200778ce33060f"

  char *username
    = MHD_digest_auth_get_username(connection);
  if (NULL == username) {
    response
      = MHD_create_response_from_buffer (strlen (DENIED),
            DENIED, MHD_RESPMEM_PERSISTENT);
    return MHD_queue_auth_fail_response (connection,
                                         "my realm",
                                         MY_OPAQUE_STR,
                                         response,
                                         MHD_NO);
  }
```

HTTP Digest authentication (Part II)

```
int ret = MHD_digest_auth_check (connection, "my realm",
                                 username,
                                 password,
                                 300);
 if ( (ret == MHD_INVALID_NONCE) ||
      (ret == MHD_NO) )
 {
   response
     = MHD_create_response_from_buffer (strlen (DENIED),
          DENIED, MHD_RESPMEM_PERSISTENT);
   return MHD_queue_auth_fail_response (connection,
        "my realm", MY_OPAQUE_STR, response,
        (ret == MHD_INVALID_NONCE) ? MHD_YES : MHD_NO);
 }
```

Exercise 16: Digest Authentication

- Add support for digest authentication to your MHD server
- Observe the traffic with `wireshark`
- Attempt a replay attack using `telnet` or `nc` or `netcat`

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

First, enable the HTTP reverse proxy module:

```
# a2enmod mod_proxy
```

Then you can configure the 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>
```

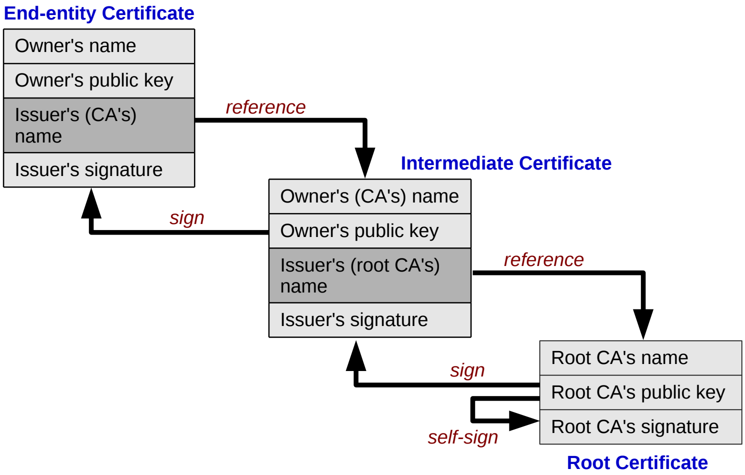
This is in contrast to an HTTP client using a proxy (such as Squid, Tor or WWWOFFLE).

Exercise 17: Reverse proxy to MHD

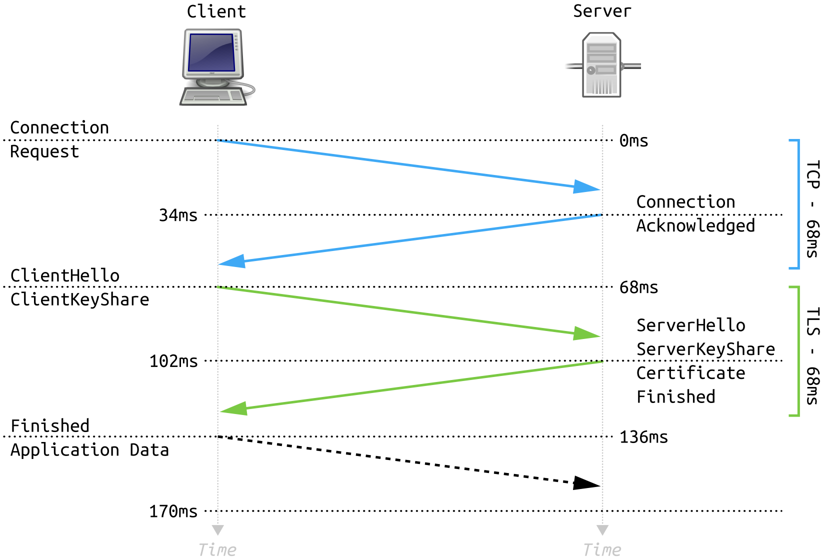
- Configure an Apache server for your site
- Redirect a particular path to your MHD instance
- Redirect another (virtual) domain to your MHD instance

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

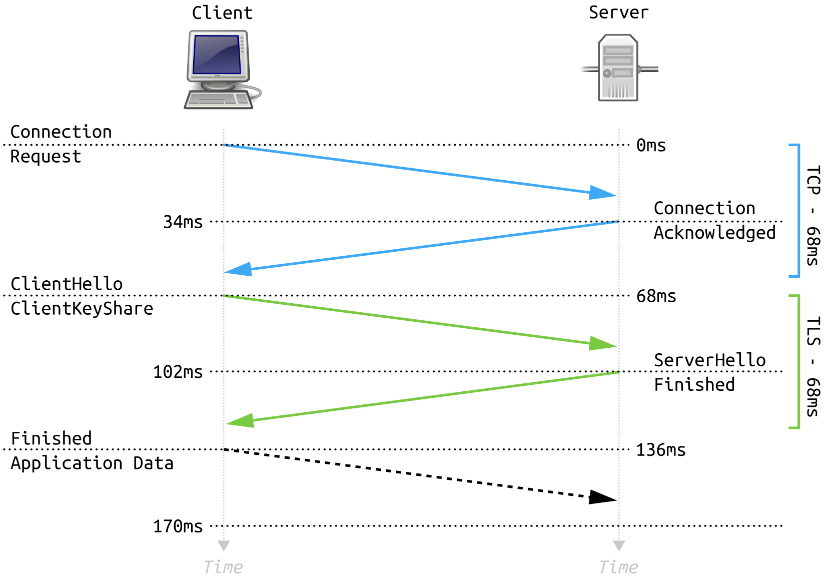
X.509 Trust Chains



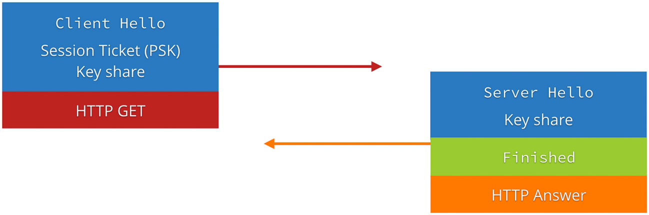
TLS 1.3: Full Handshake



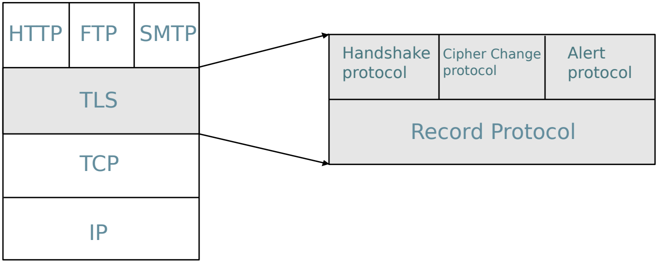
TLS 1.3: Abbreviated Handshake



TLS 1.3: 0.5 RTT Handshake



TLS Protocol Stack



Exercise 18: 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
```

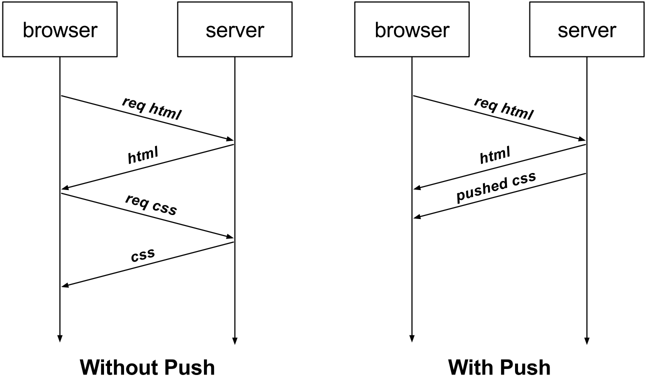
- Verify your configuration using `https://www.ssllabs.com/ssltest/` and `https://observatory.mozilla.org/`

HTTP/2

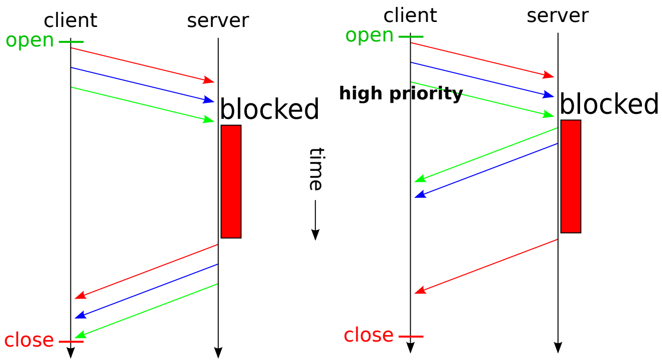
Key changes:

- 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


browser
server
req html
html
req css
css
Without Push
browser
server
req html
html
pushed css
With Push

HOL blocking and prioritization



Exercise 19: 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 20: Putting it all together

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

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

Exercise 21: Homework

- Use TLS with MHD directly
- Cross compile MHD for ARM CPUs:
  - Install `gcc-arm-linux-gnueabi` and `binutils-arm-linux-gnueabi` or `gnueabihf` for ARM systems implementing "hardfloat"
  - Configure using `./configure --host=arm-linux-gnueabi(hf)`
- Implement a RESTful IoT sensor using MHD
- Minimize the MHD binary size by setting `configure` and `gcc` options to minimize code size and omit features you do not require

Part III

Continuation-Passing Style (CPS)

Continuation-passing style (CPS) is a style of programming in which control is passed explicitly in the form of a continuation.

- In CPS, functions never return!
- Instead, functions takes an extra argument: an explicit "continuation", i.e. a function of one argument.

When the CPS function has computed its result value, it jumps to the continuation function passing its result value as the argument.

CPS transformation

Programs can always be (automatically) converted to CPS:

- Compilers for functional languages often translate to CPS
- Care must be taken to not use a call stack for the continuations! (See also: tail call)
- Event loops (libev, libevent, glib, GNUnet-SCHEDULER) use a style resembling CPS when code waits for inputs

Event loop: minimal example

```
static void task (void *cls) {
  const char *text = cls;
  printf ("%s\n", text);
}
static void start (void *cls) {
  GNUNET_SCHEDULER_add_now (&task, "Hello world");
  GNUNET_SCHEDULER_add_delayed (TIMEOUT,
                                &task, "later");
  GNUNET_SCHEDULER_add_read_net (TIMEOUT, FD,
                                 &task, "FD ready");
}
int main () {
  GNUNET_SCHEDULER_run (&start, NULL);
}
```

select()

- Traditional API for monitoring multiple file descriptors
- Read `man 2 select`

Be aware:

- Most portable API, even available on W32
- Always respect `FD_SETSIZE` when manipulating the sets!
- `FD_SETSIZE` can be very small on W32 systems

select() and MHD: minimal example

```
d = MHD_start_daemon (MHD_USE_ERROR_LOG, 8080,
      NULL, NULL, &handle_req, NULL, MHD_OPTION_END);
while (1) {
  fd_set rs, ws, es;
  int maxposixs = -1;

  FD_ZERO (&rs); FD_ZERO (&ws); FD_ZERO (&es);
  MHD_get_fdset (d, &rs, &ws, &es, &maxposixs);
  MHD_get_timeout (d, &timeout);
  tv.tv_sec = timeout / 1000; /* ms -> s */
  tv.tv_usec = (timeout % 1000) * 1000000ULL; /* ms -> us */
  select (maxposixs + 1, &rs, &ws, &es, &tv);
  MHD_run (d); // more efficient: MHD_run_from_select ();
}
```

poll()

- Instead of using 3 bit-sets, uses an array
- Mostly eliminates restriction on `FD_SETSIZE` of select()
- Allows more fine-grained control over what events we care about

poll()

```
#include <poll.h>

struct pollfd fds[2];
fds[0].fd = STDION_FILENO;
fds[0].events = POLLIN;
fds[1].fd = STDOUT_FILENO;
fds[1].events = POLLOUT;
while (1) {
  ppoll (fds, 2, NULL /* timeout */, NULL /* sigmask */);
  if (0 != (fds[0].revents & POLLIN)) do_read ();
  if (0 != (fds[1].revents & POLLOUT)) do_write ();
}
```

MHD and poll()

- MHD can use ppoll() internally:
  `MHD_USE_POLL_INTERNAL_THREAD`

Exercise 22: going beyond the `FD_SETSIZE` limit

Create an MHD service that sends a slow, infinite stream of "A"s to each HTTP client, with a delay of 1s between characters:

- Create a linked list with data about all active connections, store a flag and the connection handle in the list items.
- Add incoming connections to list, use `MHD_create_response_from_callback()` with a size of `MHD_SIZE_UNKNOWN` and a pointer to your linked list entry.
- When called, if flag set, return "A" once and clear flag. If flag unset, call `MHD_suspend_connection()`.
- In your main() function, add a loop that once per second goes over the list, sets all flags and calls `MHD_resume_connection()` on each entry. Make sure to use a mutex to access the list!

Exercise 23: going beyond the `FD_SETSIZE` limit

Create 2k concurrent clients using `wget` to test your implementations with `select()` and `ppoll()`-based event loops.

- What happens if you go beyond 1021 connections with select()?
- What happens if you go beyond $\approx$ 1600 connections with select()?
- What happens if you go beyond 1021 connections with ppoll()?
- What happens if you go beyond $\approx$ 1600 connections with ppoll()?

Hint: Read up on `listen()` and `ulimit` to explain your findings!

High-performance networking: `epoll()`

- Uses a special file descriptor to represent the `FD_SETs`
- The epoll-FD is readable when anything is ready in these sets
- Sets are manipulated using system call `epoll_ctl()`
- Read `man 3 epoll`

Be aware:

- Least portable (Linux/FreeBSD-only)
- Best performing for large event sets: $O(1)$
- Requires care when using `EPOLLET` (edge-triggered) flag

epoll() – minimal example

```c
#include <sys/epoll.h>

struct epoll_event evt;
struct epoll_event events_list[LEN];

epfd = epoll_create1 (EPOLL_CLOEXEC);
evt.events = EPOLLIN;
evt.data.ptr = NULL;
epoll_ctl (epfd, EPOLL_CTL_ADD, some_fd, &evt))

while (1)
{
  ec = epoll_wait (epfd, events_list, LEN, TIMEOUT);
  for (unsigned int i=0;i<ec;i++)
  handle_event (&events_list[i]);
}
```

MHD and epoll()

- MHD can use epoll() internally: `MHD_USE_EPOLL_INTERNAL_THREAD`
- MHD can give you its epoll() FD: `MHD_USE_EPOLL`, then ask `MHD_get_daemon_info()` for `MHD_DAEMON_INFO_EPOLL_FD`.

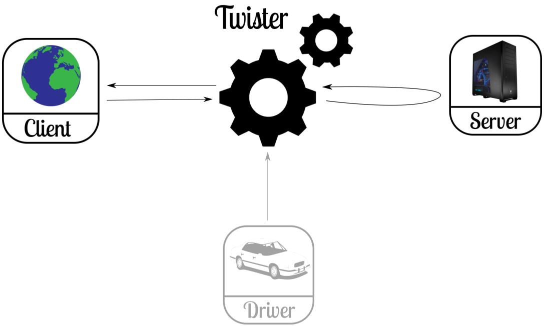
Exercise 24: combine select() and epoll()

- Rewrite your external select-based event loop to use MHD's epoll() FD

Twister Installation

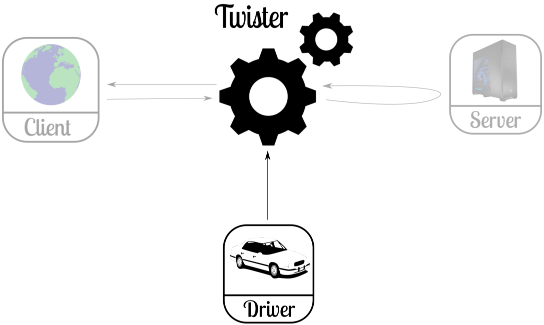
```
# apt install libtool libltdl-dev autoconf automake libunistring-dev libidn11-dev
$ git clone git://git.gnunet.org/gnunet
# cd gnunet; ./bootstrap
$ ./configure --prefix=$HOME/mc --with-microhttpd=$HOME/mc
$ git clone git://git.taler.net/exchange
$ cd exchange ; ./bootstrap
$ ./configure --prefix=$HOME/mc --with-gnunet=$HOME/mc --with-microhttpd=$HOME/mc
# make install ; cd ..

$ git clone git://git.taler.net/twister
$ cd twister; ./bootstrap
$ ./configure --prefix=$HOME --with-gnunet=$HOME/mc --with-exchange=$HOME/mc \
      --with-microhttpd=$HOME/mc
# make install ; cd ..
```

The Twister HTTP proxy[1]




```
# twister.conf
[twister]
# listens here
HTTP_PORT = 8888
# forwards there
DESTINATION_BASE_URL = http://localhost:8080
```

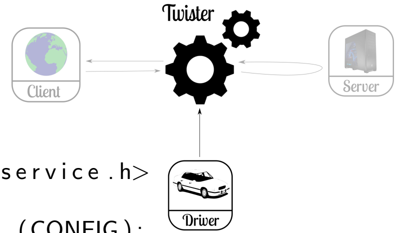
Driving it




```
# twister.conf

[twister]
# Control channel.
UNIXPATH = /tmp/taler-service-twister.sock
..
```

```
# CLI

$ taler-twister -c twister.conf \
                --flip-ul=child.one
```

Driving it (libtalertwister)




```c
#include <taler/taler_twister_service.h>

handle = TALER_TWISTER_connect (CONFIG);

/* Issue char-flipping command */
TALER_TWISTER_flip_upload (handle,
                           "child.one",
                           &callback,
                           NULL);
// ...
TALER_TWISTER_disconnect (handle);
```

Part IV

RESTful Design

Defined in Roy Fielding's dissertation "Architectural Styles and the Design of Network-based Software Architectures". Most of the six REST constraints that guide system design come from HTTP:

- Client-server architecture
- Statelessness
- Cacheability
- Layered system
- Code on demand (optional)
- Uniform interface

Uniform interface

- Resource identification in requests (use URIs)
- Resource manipulation through representations (use HTTP methods)
- Self-descriptive messages (use `Content-Type`)
- Hypermedia as the engine of application state

The last point is rarely supported by real-world REST APIs.

Resource representation

Common formats for resource representation:

- Server-side: (relational?) database (PostgreSQL)
- Client-side: IndexDB
- Network: ASN.1 formats (BER, DER), Protocol Buffers, XML, JSON

Java Script Object Notation (JSON)

Pros:

- Widely used
- Deterministic encodings available
- Schema-less

Cons:

- Schema-less
- Inefficient compared to binary encodings (solution: BSON!)

Manipulating JSON from C

Do **not** try to create JSON strings manually in C code.

- String manipulation is notoriously difficult in C
- Memory management becomes a nightmare
- Ensuring syntactic correctness will already be too hard

Solution: `libjansson`

`https://jansson.readthedocs.io/en/2.11/`

libjansson

```c
#include <jansson.h>

  // construct '{"hello":5, "world":10}'
  json_t *j = json_pack("{s:i,s:i}",
                         "hello", 5,
                         "world", 10);
  // convert to string
  char *s = json_dumps (j, 0);
  // parse string
  json_t *k = json_loads (s, 0, NULL);
  // access field
  assert (5 == json_integer_value
          (json_object_get (k, "hello")));
```

Exercise 25: Use libjansson

Modify your HTTP server to:

- Dynamically generate a JSON-formatted response with a random number
- Properly set the "Content-Type" header
- Optionally: support body compression

Parsing PUT/POST data with libgnunetjson (1/2)

```
parse_json (struct MHD_Connection *connection, void **con_cls
            const char *upload_data, size_t *upload_data_size
{
  json_t *json;
  enum GNUNET_JSON_PostResult pr
    = GNUNET_JSON_post_parser (REQUEST_BUFFER_MAX,
        con_cls, upload_data, upload_data_size, &json);
  switch (pr) {
  case GNUNET_JSON_PR_OUT_OF_MEMORY:
    /* generate "out of memory" internal server error */
  case GNUNET_JSON_PR_CONTINUE:
    /* return MHD_YES to MHD to read more from client */
  case GNUNET_JSON_PR_REQUEST_TOO_LARGE:
    /* generate "request too large" */
  case GNUNET_JSON_PR_JSON_INVALID:
  /* generate "invalid json" */
  case GNUNET_JSON_PR_SUCCESS:
    GNUNET_assert (NULL != json); /* party on! */   } }
```

Parsing PUT/POST data with `libgnunetjson` (2/2)

Clean up the state in `con_cls` on connection completion:

```
static void
handle_mhd_completion_callback (void *cls,
   struct MHD_Connection *connection, void **con_cls,
   enum MHD_RequestTerminationCode toe)
{
  GNUNET_JSON_post_parser_cleanup (*con_cls);
}

MHD_start_daemon (...,
                  MHD_OPTION_NOTIFY_COMPLETED,
                  &handle_mhd_completion_callback, NULL,
                  ...);
```

Relational databases

Common relational databases include:

- Sqlite (especially for small systems)
- PostgreSQL
- MariaDB (free software fork of MySQL)
- MySQL (dying due to Oracle acquisition)

We will use PostgreSQL.

```
postgres $ createuser -s $USER
$USER $ createdb mydb
```

Database access from C

Each database comes with a C API:

- libsqlite3
- libpq
- libmysql-client

They are all horrible. We will use the `libgnunetpq` wrapper around `libpq` which eliminates the worst atrocities.

libgnunetpq by example

```
#include <libpq-fe.h>
#include <gnunet/libgnunetpq.h>

  PGconn *conn = GNUNET_PQ_connect ("postgres:///db-name");
  // use 'conn' here, finally:
  PQfinish (conn);
```

libgnunetpq by example

```c
struct GNUNET_PQ_ExecuteStatement es[] = {
  GNUNET_PQ_make_execute
    ("DROP TABLE IF EXISTS foo CASCADE;"),
  GNUNET_PQ_make_execute
    ("CREATE TABLE IF NOT EXISTS bar (key INT4, val BYTEA);")
  GNUNET_PQ_make_try_execute
    ("CREATE INDEX bar_index ON bar (key);"),
  GNUNET_PQ_EXECUTE_STATEMENT_END
};

GNUNET_PQ_exec_statements (conn, es);
```

libgnunetpq by example

```
struct GNUNET_PQ_PreparedStatement ps[] = {
  GNUNET_PQ_make_prepare ("foo_insert",
    "INSERT INTO foo (key, val) VALUES ($1, $2);",
    2),
  GNUNET_PQ_make_prepare ("foo_select",
    "SELECT val FROM foo WHERE key = $1;",
    1),
  GNUNET_PQ_make_prepare ("foo_delete",
    "DELETE FROM foo WHERE key = $1;",
    1),
  GNUNET_PQ_PREPARED_STATEMENT_END
};

GNUNET_PQ_prepare_statements (conn, ps);
```

libgnunetpq by example

```
struct GNUNET_PQ_QueryParam params[] = {
  GNUNET_PQ_query_param_uint32 (&key),
  GNUNET_PQ_query_param_end
};

GNUNET_PQ_eval_prepared_non_select (session->conn,
                                    "foo_delete",
                                    params);
```

libgnunetpq by example

```
struct GNUNET_PQ_QueryParam params[] = {
  GNUNET_PQ_query_param_uint32 (&key),
  GNUNET_PQ_query_param_end
};
char *val; size_t val_size;
struct GNUNET_PQ_ResultSpec rs[] = {
  GNUNET_PQ_result_spec_variable_size ("val",
                                       &val, &val_size),
  GNUNET_PQ_result_spec_end
};

GNUNET_PQ_eval_prepared_singleton_select (session->conn,
                                          "foo_select",
                                          params,
                                          rs);
/* use 'val' here */
GNUNET_PQ_cleanup_result (rs);
```

libgnunetpq by example

```
struct GNUNET_PQ_QueryParam params[] = {
  GNUNET_PQ_query_param_uint32 (&key),
  GNUNET_PQ_query_param_end
};

GNUNET_PQ_eval_prepared_multi_select (session->conn,
                                      "foo_select",
                                      params,
                                      &handle_result,
                                      hr_CTX);
```

libgnunetpq by example

```
static void
handle_result (void *cls,
               PGresult *result,
               unsigned int num_results) {
  for (unsigned int i=0;i<num_results;i++) {
    char *val; size_t val_size;
    struct GNUNET_PQ_ResultSpec rs[] = {
      GNUNET_PQ_result_spec_variable_size ("val",
                                           &val, &val_size),
      GNUNET_PQ_result_spec_end
    };

    GNUNET_PQ_extract_result (result, rs, i);
    /* use 'val' here */
    GNUNET_PQ_cleanup_result (rs);
  }
}
```

Exercise 26: Implement a backup service

Modify your HTTP server to:

- Store resources in your database given PUT request
- Return resources from database given GET request
- Use HTTP features to optimize caching of GET replies
- Optionally, use `libgnunetutil` to:
  - restrict URIs to Crockford base32-encoded EdDSA public keys
  - require signatures by respective private key in separate HTTP header for PUTs
  - support transactional updates by requiring hash of last version to be supplied on PUT

Exercise 27: evaluate performance

Determine the speed of your solution:

- How many stores can you handle per second?
- How many retrievals can you handle per second?
- How much do these two depend on the size of the data?

Use Curl or libcurl to simulate clients. Make sure you measure the service and not the client!

Future Work (aka Bachelor's thesis topics!)

- Improve usability of MHD API
- Hack on GNUnet (HTTP-ng transport, HTTP/3 transport)
- Hack on GNU Taler (sync, auditor API, multi-DB support, speed)

Exam

- 15 minutes per instructor, instructors randomly assigned
- HTTP protocol, explain features:
  - Connection management
  - HTTP Methods
  - Techniques for performance optimization
  - Security features
  - HTTP/2 vs. HTTP/1.x
- Modern Internet service architecture (proxies, REST, etc.)
- Your project / implementation

RTFL