A Tutorial for GNU libmicrohttpd

Version 0.9.48
2 April 2016

Sebastian Gerhardt (sebgerhardt@gmx.net)
Christian Grothoff (christian@grothoff.org)
Matthieu Speder (mspeder@users.sourceforge.net)

This tutorial documents GNU libmicrohttpd version 0.9.48, last updated 2 April 2016.

Copyright (c) 2008 Sebastian Gerhardt.

Copyright (c) 2010, 2011, 2012, 2013, 2016, 2021 Christian Grothoff.

> Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

Table of Contents

1 Introduction

This tutorial is for developers who want to learn how they can add HTTP serving capabilities to their applications with the *GNU libmicrohttpd* library, abbreviated *MHD*. The reader will learn how to implement basic HTTP functions from simple executable sample programs that implement various features.

The text is supposed to be a supplement to the API reference manual of *GNU libmicrohttpd* and for that reason does not explain many of the parameters. Therefore, the reader should always consult the manual to find the exact meaning of the functions used in the tutorial. Furthermore, the reader is encouraged to study the relevant *RFCs*, which document the HTTP standard.

GNU libmicrohttpd is assumed to be already installed. This tutorial is written for version 0.9.48. At the time being, this tutorial has only been tested on *GNU/Linux* machines even though efforts were made not to rely on anything that would prevent the samples from being built on similar systems.

1.1 History

This tutorial was originally written by Sebastian Gerhardt for MHD 0.4.0. It was slightly polished and updated to MHD 0.9.0 by Christian Grothoff.

2 Hello browser example

The most basic task for a HTTP server is to deliver a static text message to any client connecting to it. Given that this is also easy to implement, it is an excellent problem to start with.

For now, the particular URI the client asks for shall have no effect on the message that will be returned. In addition, the server shall end the connection after the message has been sent so that the client will know there is nothing more to expect.

The C program `hellobrowser.c`, which is to be found in the examples section, does just that. If you are very eager, you can compile and start it right away but it is advisable to type the lines in by yourself as they will be discussed and explained in detail.

After the necessary includes and the definition of the port which our server should listen on

```
#include <sys/types.h>
#include <sys/select.h>
#include <sys/socket.h>
#include <microhttpd.h>

#define PORT 8888
```

the desired behaviour of our server when HTTP request arrive has to be implemented. We already have agreed that it should not care about the particular details of the request, such as who is requesting what. The server will respond merely with the same small HTML page to every request.

The function we are going to write now will be called by *GNU libmicrohttpd* every time an appropriate request comes in. While the name of this callback function is arbitrary, its parameter list has to follow a certain layout. So please, ignore the lot of parameters for now, they will be explained at the point they are needed. We have to use only one of them, `struct MHD_Connection *connection`, for the minimalistic functionality we want to achieve at the moment.

This parameter is set by the *libmicrohttpd* daemon and holds the necessary information to relate the call with a certain connection. Keep in mind that a server might have to satisfy hundreds of concurrent connections and we have to make sure that the correct data is sent to the destined client. Therefore, this variable is a means to refer to a particular connection if we ask the daemon to sent the reply.

Talking about the reply, it is defined as a string right after the function header

```
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 **req_cls)
{
  const char *page  = "<html><body>Hello, browser!</body></html>";
```

HTTP is a rather strict protocol and the client would certainly consider it "inappropriate" if we just sent the answer string "as is". Instead, it has to be wrapped with additional information stored in so-called headers and footers. Most of the work in this area is done by the library for us—we just have to ask. Our reply string packed in the necessary layers will be called a "response". To obtain such a response we hand our data (the reply–string) and its size over to the `MHD_create_response_from_buffer` function. The last two parameters basically tell *MHD* that we do not want it to dispose the message data for us when it has been sent and there also needs no internal copy to be done because the *constant* string won't change anyway.

```
  struct MHD_Response *response;
  int ret;

  response = MHD_create_response_from_buffer (strlen (page),
                                           (void*) page, MHD_RESPMEM_PERSISTENT);
```

Now that the the response has been laced up, it is ready for delivery and can be queued for sending. This is done by passing it to another *GNU libmicrohttpd* function. As all our work was done in the scope of one function, the recipient is without doubt the one associated with the local variable `connection` and consequently this variable is given to the queue function. Every HTTP response is accompanied by a status code, here "OK", so that the client knows this response is the intended result of his request and not due to some error or malfunction.

Finally, the packet is destroyed and the return value from the queue returned, already being set at this point to either MHD_YES or MHD_NO in case of success or failure.

```
  ret = MHD_queue_response (connection, MHD_HTTP_OK, response);
  MHD_destroy_response (response);

  return ret;
}
```

With the primary task of our server implemented, we can start the actual server daemon which will listen on `PORT` for connections. This is done in the main function.

```
int main ()
{
  struct MHD_Daemon *daemon;

  daemon = MHD_start_daemon (MHD_USE_INTERNAL_POLLING_THREAD, PORT, NULL, NULL,
                             &answer_to_connection, NULL, MHD_OPTION_END);
  if (NULL == daemon) return 1;
```

The first parameter is one of three possible modes of operation. Here we want the daemon to run in a separate thread and to manage all incoming connections in the same thread. This means that while producing the response for one connection, the other connections will be put on hold. In this example, where the reply is already known and therefore the request is served quickly, this poses no problem.

We will allow all clients to connect regardless of their name or location, therefore we do not check them on connection and set the third and fourth parameter to NULL.

Parameter five is the address of the function we want to be called whenever a new connection has been established. Our `answer_to_connection` knows best what the client wants and needs no additional information (which could be passed via the next parameter) so the next (sixth) parameter is NULL. Likewise, we do not need to pass extra options to the daemon so we just write the MHD_OPTION_END as the last parameter.

As the server daemon runs in the background in its own thread, the execution flow in our main function will continue right after the call. Because of this, we must delay the execution flow in the main thread or else the program will terminate prematurely. We let it pause in a processing-time friendly manner by waiting for the enter key to be pressed. In the end, we stop the daemon so it can do its cleanup tasks.

```
  getchar ();

  MHD_stop_daemon (daemon);
  return 0;
}
```

The first example is now complete.

Compile it with

```
cc hellobrowser.c -o hellobrowser -I$PATH_TO_LIBMHD_INCLUDES
  -L$PATH_TO_LIBMHD_LIBS -lmicrohttpd
```

with the two paths set accordingly and run it.

Now open your favorite Internet browser and go to the address `http://localhost:8888/`, provided that 8888 is the port you chose. If everything works as expected, the browser will present the message of the static HTML page it got from our minimal server.

Remarks

To keep this first example as small as possible, some drastic shortcuts were taken and are to be discussed now.

Firstly, there is no distinction made between the kinds of requests a client could send. We implied that the client sends a GET request, that means, that he actually asked for some data. Even when it is not intended to accept POST requests, a good server should at least recognize that this request does not constitute a legal request and answer with an error code. This can be easily implemented by checking if the parameter `method` equals the string "GET" and returning a `MHD_NO` if not so.

Secondly, the above practice of queuing a response upon the first call of the callback function brings with it some limitations. This is because the content of the message body will not be received if a response is queued in the first iteration. Furthermore, the connection will be closed right after the response has been transferred then. This is typically not what you want as it disables HTTP pipelining. The correct approach is to simply not queue a message on the first callback unless there is an error. The `void**` argument to the callback provides a location for storing information about the history of the connection; for the first call, the pointer will point to NULL. A simplistic way to differentiate the first call from others is to check if the pointer is NULL and set it to a non-NULL value during the first call.

Both of these issues you will find addressed in the official `minimal_example.c` residing in the `src/examples` directory of the *MHD* package. The source code of this program should look very familiar to you by now and easy to understand.

For our example, we create the response from a static (persistent) buffer in memory and thus pass `MHD_RESPMEM_PERSISTENT` to the response construction function. In the usual case, responses are not transmitted immediately after being queued. For example, there might be other data on the system that needs to be sent with a higher priority. Nevertheless, the queue function will return successfully—raising the problem that the data we have pointed to may be invalid by the time it is about being sent. This is not an issue here because we can expect the `page` string, which is a constant *string literal* here, to be static. That means it will be present and unchanged for as long as the program runs. For dynamic data, one could choose to either have *MHD* free the memory `page` points to itself when it is not longer needed (by passing `MHD_RESPMEM_MUST_FREE`) or, alternatively, have the library to make and manage its own copy of it (by passing `MHD_RESPMEM_MUST_COPY`). Naturally, this last option is the most expensive.

Exercises

- While the server is running, use a program like `telnet` or `netcat` to connect to it. Try to form a valid HTTP 1.1 request yourself like

  ```
  GET /dontcare HTTP/1.1
  Host: itsme
  <enter>
  ```

  and see what the server returns to you.
- Also, try other requests, like POST, and see how our server does not mind and why. How far in malforming a request can you go before the builtin functionality of *MHD* intervenes and an altered response is sent? Make sure you read about the status codes in the *RFC*.
- Add the option `MHD_USE_PEDANTIC_CHECKS` to the start function of the daemon in `main`. Mind the special format of the parameter list here which is described in the manual. How indulgent is the server now to your input?
- Let the main function take a string as the first command line argument and pass `argv[1]` to the `MHD_start_daemon` function as the sixth parameter. The address of this string will be passed to the callback function via the `cls` variable. Decorate the text given at the command line when the server is started with proper HTML tags and send it as the response instead of the former static string.
- *Demanding:* Write a separate function returning a string containing some useful information, for example, the time. Pass the function's address as the sixth parameter and evaluate this function on every request anew in `answer_to_connection`. Remember to free the memory of the string every time after satisfying the request.

3 Exploring requests

This chapter will deal with the information which the client sends to the server at every request. We are going to examine the most useful fields of such an request and print them out in a readable manner. This could be useful for logging facilities.

The starting point is the *hellobrowser* program with the former response removed.

This time, we just want to collect information in the callback function, thus we will just return MHD_NO after we have probed the request. This way, the connection is closed without much ado by the server.

```
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 **req_cls)
{
  ...
  return MHD_NO;
}
```

The ellipsis marks the position where the following instructions shall be inserted.

We begin with the most obvious information available to the server, the request line. You should already have noted that a request consists of a command (or "HTTP method") and a URI (e.g. a filename). It also contains a string for the version of the protocol which can be found in `version`. To call it a "new request" is justified because we return only `MHD_NO`, thus ensuring the function will not be called again for this connection.

```
printf ("New %s request for %s using version %s\n", method, url, version);
```

The rest of the information is a bit more hidden. Nevertheless, there is lot of it sent from common Internet browsers. It is stored in "key-value" pairs and we want to list what we find in the header. As there is no mandatory set of keys a client has to send, each key-value pair is printed out one by one until there are no more left. We do this by writing a separate function which will be called for each pair just like the above function is called for each HTTP request. It can then print out the content of this pair.

```
int print_out_key (void *cls, enum MHD_ValueKind kind,
                   const char *key, const char *value)
{
  printf ("%s: %s\n", key, value);
  return MHD_YES;
}
```

To start the iteration process that calls our new function for every key, the line

```
MHD_get_connection_values (connection, MHD_HEADER_KIND, &print_out_key, NULL);
```

needs to be inserted in the connection callback function too. The second parameter tells the function that we are only interested in keys from the general HTTP header of the request. Our iterating function `print_out_key` does not rely on any additional information to fulfill its duties so the last parameter can be NULL.

All in all, this constitutes the complete `logging.c` program for this chapter which can be found in the `examples` section.

Connecting with any modern Internet browser should yield a handful of keys. You should try to interpret them with the aid of *RFC 2616*. Especially worth mentioning is the "Host" key which is often used to serve several different websites hosted under one single IP address but reachable by different domain names (this is called virtual hosting).

Conclusion

The introduced capabilities to itemize the content of a simple GET request—especially the URI—should already allow the server to satisfy clients' requests for small specific resources (e.g. files) or even induce alteration of server state. However, the latter is not recommended as the GET method (including its header data) is by convention considered a "safe" operation, which should not change the server's state in a significant way. By convention, GET operations can thus be performed by crawlers and other automatic software. Naturally actions like searching for a passed string are fine.

Of course, no transmission can occur while the return value is still set to `MHD_NO` in the callback function.

Exercises

- By parsing the `url` string and delivering responses accordingly, implement a small server for "virtual" files. When asked for `/index.htm{l}`, let the response consist of a HTML page containing a link to `/another.html` page which is also to be created "on the fly" in case of being requested. If neither of these two pages are requested, `MHD_HTTP_NOT_FOUND` shall be returned accompanied by an informative message.
- A very interesting information has still been ignored by our logger—the client's IP address. Implement a callback function

  ```
  static int on_client_connect (void *cls,
                                const struct sockaddr *addr,
                                socklen_t addrlen)
  ```

  that prints out the IP address in an appropriate format. You might want to use the POSIX function `inet_ntoa` but bear in mind that `addr` is actually just a structure containing other substructures and is *not* the variable this function expects. Make sure to return `MHD_YES` so that the library knows the client is allowed to connect (and to then process the request). If one wanted to limit access basing on IP addresses, this would be the place to do it. The address of your `on_client_connect` function must be passed as the third parameter to the `MHD_start_daemon` call.

4 Response headers

Now that we are able to inspect the incoming request in great detail, this chapter discusses the means to enrich the outgoing responses likewise.

As you have learned in the *Hello, Browser* chapter, some obligatory header fields are added and set automatically for simple responses by the library itself but if more advanced features are desired, additional fields have to be created. One of the possible fields is the content type field and an example will be developed around it. This will lead to an application capable of correctly serving different types of files.

When we responded with HTML page packed in the static string previously, the client had no choice but guessing about how to handle the response, because the server had not told him. What if we had sent a picture or a sound file? Would the message have been understood or merely been displayed as an endless stream of random characters in the browser? This is what the mime content types are for. The header of the response is extended by certain information about how the data is to be interpreted.

To introduce the concept, a picture of the format *PNG* will be sent to the client and labeled accordingly with `image/png`. Once again, we can base the new example on the `hellobrowser` program.

```
#define FILENAME "picture.png"
#define MIMETYPE "image/png"

static enum MHD_Result
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 **req_cls)
{
  unsigned char *buffer = NULL;
  struct MHD_Response *response;
```

We want the program to open the file for reading and determine its size:

```
  int fd;
  enum MHD_Result ret;
  struct stat sbuf;

  if (0 != strcmp (method, "GET"))
    return MHD_NO;
  if ( (-1 == (fd = open (FILENAME, O_RDONLY))) ||
       (0 != fstat (fd, &sbuf)) )
    {
     /* error accessing file */
      /* ... (see below) */
    }
 /* ... (see below) */
```

When dealing with files, there is a lot that could go wrong on the server side and if so, the client should be informed with `MHD_HTTP_INTERNAL_SERVER_ERROR`.

```
      /* error accessing file */
     if (fd != -1) close (fd);
      const char *errorstr =
        "<html><body>An internal server error has occurred!\
                              </body></html>";
      response =
        MHD_create_response_from_buffer (strlen (errorstr),
                                         (void *) errorstr,
                                         MHD_RESPMEM_PERSISTENT);
      if (response)
        {
          ret =
            MHD_queue_response (connection, MHD_HTTP_INTERNAL_SERVER_ERROR,
                                response);
          MHD_destroy_response (response);

          return MHD_YES;
        }
      else
        return MHD_NO;
  if (!ret)
    {
      const char *errorstr = "<html><body>An internal server error has occurred!\
                              </body></html>";

      if (buffer) free(buffer);

      response = MHD_create_response_from_buffer (strlen(errorstr), (void*) errorstr,
                                                  MHD_RESPMEM_PERSISTENT);

      if (response)
        {
          ret = MHD_queue_response (connection,
                                    MHD_HTTP_INTERNAL_SERVER_ERROR,
                                    response);
          MHD_destroy_response (response);

          return MHD_YES;
        }
      else return MHD_NO;
    }
```

Note that we nevertheless have to create a response object even for sending a simple error code. Otherwise, the connection would just be closed without comment, leaving the client curious about what has happened.

But in the case of success a response will be constructed directly from the file descriptor:

```
    /* error accessing file */
    /* ... (see above) */
   }

  response =
    MHD_create_response_from_fd_at_offset (sbuf.st_size, fd, 0);
  MHD_add_response_header (response, "Content-Type", MIMETYPE);
  ret = MHD_queue_response (connection, MHD_HTTP_OK, response);
  MHD_destroy_response (response);
```

Note that the response object will take care of closing the file descriptor for us.

Up to this point, there was little new. The actual novelty is that we enhance the header with the meta data about the content. Aware of the field's name we want to add, it is as easy as that:

```
MHD_add_response_header(response, "Content-Type", MIMETYPE);
```

We do not have to append a colon expected by the protocol behind the first field—*GNU libhttpdmicro* will take care of this.

The function finishes with the well-known lines

```
  ret = MHD_queue_response (connection, MHD_HTTP_OK, response);
  MHD_destroy_response (response);
  return ret;
}
```

The complete program `responseheaders.c` is in the `examples` section as usual. Find a *PNG* file you like and save it to the directory the example is run from under the name `picture.png`. You should find the image displayed on your browser if everything worked well.

Remarks

The include file of the *MHD* library comes with the header types mentioned in *RFC 2616* already defined as macros. Thus, we could have written `MHD_HTTP_HEADER_CONTENT_TYPE` instead of `"Content-Type"` as well. However, one is not limited to these standard headers and could add custom response headers without violating the protocol. Whether, and how, the client would react to these custom header is up to the receiver. Likewise, the client is allowed to send custom request headers to the server as well, opening up yet more possibilities how client and server could communicate with each other.

The method of creating the response from a file on disk only works for static content. Serving dynamically created responses will be a topic of a future chapter.

Exercises

- Remember that the original program was written under a few assumptions—a static response using a local file being one of them. In order to simulate a very large or hard to reach file that cannot be provided instantly, postpone the queuing in the callback with the `sleep` function for 30 seconds *if* the file `/big.png` is requested (but deliver the same as above). A request for `/picture.png` should provide just the same but without any artificial delays.

Now start two instances of your browser (or even use two machines) and see how the second client is put on hold while the first waits for his request on the slow file to be fulfilled.

Finally, change the sourcecode to use `MHD_USE_THREAD_PER_CONNECTION` when the daemon is started and try again.

- Did you succeed in implementing the clock exercise yet? This time, let the server save the program's start time `t` and implement a response simulating a countdown that reaches 0 at `t+60`. Returning a message saying on which point the countdown is, the response should ultimately be to reply "Done" if the program has been running long enough,

  An unofficial, but widely understood, response header line is `Refresh: DELAY; url=URL` with the uppercase words substituted to tell the client it should request the given resource after the given delay again. Improve your program in that the browser (any modern browser should work) automatically reconnects and asks for the status again every 5 seconds or so. The URL would have to be composed so that it begins with "http://", followed by the *URI* the server is reachable from the client's point of view.

  Maybe you want also to visualize the countdown as a status bar by creating a `<table>` consisting of one row and `n` columns whose fields contain small images of either a red or a green light.

5 Supporting basic authentication

With the small exception of IP address based access control, requests from all connecting clients where served equally until now. This chapter discusses a first method of client's authentication and its limits.

A very simple approach feasible with the means already discussed would be to expect the password in the *URI* string before granting access to the secured areas. The password could be separated from the actual resource identifier by a certain character, thus the request line might look like

```
GET /picture.png?mypassword
```

In the rare situation where the client is customized enough and the connection occurs through secured lines (e.g., a embedded device directly attached to another via wire) and where the ability to embed a password in the URI or to pass on a URI with a password are desired, this can be a reasonable choice.

But when it is assumed that the user connecting does so with an ordinary Internet browser, this implementation brings some problems about. For example, the URI including the password stays in the address field or at least in the history of the browser for anybody near enough to see. It will also be inconvenient to add the password manually to any new URI when the browser does not know how to compose this automatically.

At least the convenience issue can be addressed by employing the simplest built-in password facilities of HTTP compliant browsers, hence we want to start there. It will, however, turn out to have still severe weaknesses in terms of security which need consideration.

Before we will start implementing *Basic Authentication* as described in *RFC 2617*, we will also abandon the simplistic and generally problematic practice of responding every request the first time our callback is called for a given connection. Queuing a response upon the first request is akin to generating an error response (even if it is a "200 OK" reply!). The reason is that MHD usually calls the callback in three phases:

1. First, to initially tell the application about the connection and inquire whether it is OK to proceed. This call typically happens before the client could upload the request body, and can be used to tell the client to not proceed with the upload (if the client requested "Expect: 100 Continue"). Applications may queue a reply at this point, but it will force the connection to be closed and thus prevent keep-alive / pipelining, which is generally a bad idea. Applications wanting to proceed with the request throughout the other phases should just return "MHD_YES" and not queue any response. Note that when an application suspends a connection in this callback, the phase does not advance and the application will be called again in this first phase.
2. Next, to tell the application about upload data provided by the client. In this phase, the application may not queue replies, and trying to do so will result in MHD returning an error code from `MHD_queue_response`. If there is no upload data, this phase is skipped.
3. Finally, to obtain a regular response from the application. This can be almost any type of response, including ones indicating failures. The one exception is a "100 Continue" response, which applications must never generate: MHD generates that response automatically when necessary in the first phase. If the application does not queue a response, MHD may call the callback repeatedly (depending a bit on the threading model, the application should suspend the connection).

But how can we tell whether the callback has been called before for the particular request? Initially, the pointer this parameter references is set by *MHD* in the callback. But it will also be "remembered" on the next call (for the same request). Thus, we can use the `req_cls` location to keep track of the request state. For now, we will simply generate no response until the parameter is non-null—implying the callback was called before at least once. We do not need to share information between different calls of the callback, so we can set the parameter to any address that is assured to be not null. The pointer to the `connection` structure will be pointing to a legal address, so we take this.

The first time `answer_to_connection` is called, we will not even look at the headers.

```
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 **req_cls)
{
  if (0 != strcmp(method, "GET")) return MHD_NO;
  if (NULL == *req_cls) {*req_cls = connection; return MHD_YES;}

  ...
  /* else respond accordingly */
  ...
}
```

Note how we lop off the connection on the first condition (no "GET" request), but return asking for more on the other one with `MHD_YES`. With this minor change, we can proceed to implement the actual authentication process.

Request for authentication

Let us assume we had only files not intended to be handed out without the correct username/password, so every "GET" request will be challenged. *RFC 7617* describes how the server shall ask for authentication by adding a *WWW-Authenticate* response header with the name of the *realm* protected. MHD can generate and queue such a failure response for you using the `MHD_queue_basic_auth_fail_response` API. The only thing you need to do is construct a response with the error page to be shown to the user if he aborts basic authentication. But first, you should check if the proper credentials were already supplied using the `MHD_basic_auth_get_username_password` call.

Your code would then look like this:

```
static enum MHD_Result
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 **req_cls)
{
  struct MHD_BasicAuthInfo *auth_info;
  enum MHD_Result ret;
  struct MHD_Response *response;
```

```
if (0 != strcmp (method, "GET"))
  return MHD_NO;
if (NULL == *req_cls)
{
  *req_cls = connection;
  return MHD_YES;
}
auth_info = MHD_basic_auth_get_username_password3 (connection);
if (NULL == auth_info)
{
  static const char *page =
    "<html><body>Authorization required</body></html>";
  response = MHD_create_response_from_buffer_static (strlen (page), page);
  ret = MHD_queue_basic_auth_fail_response3 (connection,
                                             "admins",
                                             MHD_YES,
                                             response);
}
else if ((strlen ("root") != auth_info->username_len) ||
         (0 != memcmp (auth_info->username, "root",
                       auth_info->username_len)) ||
         /* The next check against NULL is optional,
          * if 'password' is NULL then 'password_len' is always zero. */
         (NULL == auth_info->password) ||
         (strlen ("pa$$w0rd") != auth_info->password_len) ||
         (0 != memcmp (auth_info->password, "pa$$w0rd",
                       auth_info->password_len)))
{
  static const char *page =
    "<html><body>Wrong username or password</body></html>";
  response = MHD_create_response_from_buffer_static (strlen (page), page);
  ret = MHD_queue_basic_auth_fail_response3 (connection,
                                             "admins",
                                             MHD_YES,
                                             response);
}
else
{
  static const char *page = "<html><body>A secret.</body></html>";
  response = MHD_create_response_from_buffer_static (strlen (page), page);
  ret = MHD_queue_response (connection, MHD_HTTP_OK, response);
}
if (NULL != auth_info)
  MHD_free (auth_info);
MHD_destroy_response (response);
return ret;
```

```
}
```

See the `examples` directory for the complete example file.

Remarks

For a proper server, the conditional statements leading to a return of `MHD_NO` should yield a response with a more precise status code instead of silently closing the connection. For example, failures of memory allocation are best reported as *internal server error* and unexpected authentication methods as *400 bad request*.

Exercises

- Make the server respond to wrong credentials (but otherwise well-formed requests) with the recommended *401 unauthorized* status code. If the client still does not authenticate correctly within the same connection, close it and store the client's IP address for a certain time. (It is OK to check for expiration not until the main thread wakes up again on the next connection.) If the client fails authenticating three times during this period, add it to another list for which the `AcceptPolicyCallback` function denies connection (temporally).
- With the network utility `netcat` connect and log the response of a `"GET"` request as you did in the exercise of the first example, this time to a file. Now stop the server and let *netcat* listen on the same port the server used to listen on and have it fake being the proper server by giving the file's content as the response (e.g. `cat log | nc -l -p 8888`). Pretending to think your were connecting to the actual server, browse to the eavesdropper and give the correct credentials.

  Copy and paste the encoded string you see in `netcat`'s output to some of the Base64 decode tools available online and see how both the user's name and password could be completely restored.

6 Processing POST data

The previous chapters already have demonstrated a variety of possibilities to send information to the HTTP server, but it is not recommended that the *GET* method is used to alter the way the server operates. To induce changes on the server, the *POST* method is preferred over and is much more powerful than *GET* and will be introduced in this chapter.

We are going to write an application that asks for the visitor's name and, after the user has posted it, composes an individual response text. Even though it was not mandatory to use the *POST* method here, as there is no permanent change caused by the POST, it is an illustrative example on how to share data between different functions for the same connection. Furthermore, the reader should be able to extend it easily.

GET request

When the first *GET* request arrives, the server shall respond with a HTML page containing an edit field for the name.

```
const char* askpage = "<html><body>\
                       What's your name, Sir?<br>\
                       <form action=\"/namepost\" method=\"post\">\
                       <input name=\"name\" type=\"text\"\
                       <input type=\"submit\" value=\" Send \"></form>\
                       </body></html>";
```

The `action` entry is the *URI* to be called by the browser when posting, and the `name` will be used later to be sure it is the editbox's content that has been posted.

We also prepare the answer page, where the name is to be filled in later, and an error page as the response for anything but proper *GET* and *POST* requests:

```
const char* greatingpage="<html><body><h1>Welcome, %s!</center></h1></body></html>";

const char* errorpage="<html><body>This doesn't seem to be right.</body></html>";
```

Whenever we need to send a page, we use an extra function `int send_page(struct MHD_Connection *connection, const char* page)` for this, which does not contain anything new and whose implementation is therefore not discussed further in the tutorial.

POST request

Posted data can be of arbitrary and considerable size; for example, if a user uploads a big image to the server. Similar to the case of the header fields, there may also be different streams of posted data, such as one containing the text of an editbox and another the state of a button. Likewise, we will have to register an iterator function that is going to be called maybe several times not only if there are different POSTs but also if one POST has only been received partly yet and needs processing before another chunk can be received.

Such an iterator function is called by a *postprocessor*, which must be created upon arriving of the post request. We want the iterator function to read the first post data which is tagged `name` and to create an individual greeting string based on the template and the name. But in order to pass this string to other functions and still be able to differentiate different connections, we must first define a structure to share the information, holding the most import entries.

```
struct connection_info_struct
{
  int connectiontype;
  char *answerstring;
  struct MHD_PostProcessor *postprocessor;
};
```

With these information available to the iterator function, it is able to fulfill its task. Once it has composed the greeting string, it returns `MHD_NO` to inform the post processor that it does not need to be called again. Note that this function does not handle processing of data for the same `key`. If we were to expect that the name will be posted in several chunks, we had to expand the namestring dynamically as additional parts of it with the same `key` came in. But in this example, the name is assumed to fit entirely inside one single packet.

```
static enum MHD_Result
iterate_post (void *coninfo_cls, enum MHD_ValueKind kind, const char *key,
              const char *filename, const char *content_type,
              const char *transfer_encoding, const char *data,
              uint64_t off, size_t size)
{
  struct connection_info_struct *con_info = coninfo_cls;

  if (0 == strcmp (key, "name"))
    {
      if ((size > 0) && (size <= MAXNAMESIZE))
        {
          char *answerstring;
          answerstring = malloc (MAXANSWERSIZE);
          if (!answerstring) return MHD_NO;

          snprintf (answerstring, MAXANSWERSIZE, greatingpage, data);
          con_info->answerstring = answerstring;
        }
      else con_info->answerstring = NULL;

      return MHD_NO;
    }

  return MHD_YES;
}
```

Once a connection has been established, it can be terminated for many reasons. As these reasons include unexpected events, we have to register another function that cleans up any resources that might have been allocated for that connection by us, namely the post processor and the greetings string. This cleanup function must take into account that it will also be called for finished requests other than *POST* requests.

```
void
request_completed (void *cls, struct MHD_Connection *connection,
                        void **req_cls,
                        enum MHD_RequestTerminationCode toe)
{
  struct connection_info_struct *con_info = *req_cls;

  if (NULL == con_info)
    return;
  if (con_info->connectiontype == POST)
    {
      MHD_destroy_post_processor (con_info->postprocessor);
      if (con_info->answerstring) free (con_info->answerstring);
    }

  free (con_info);
  *req_cls = NULL;
}
```

GNU libmicrohttpd is informed that it shall call the above function when the daemon is started in the main function.

```
...
daemon = MHD_start_daemon (MHD_USE_INTERNAL_POLLING_THREAD, PORT, NULL, NULL,
                           &answer_to_connection, NULL,
                           MHD_OPTION_NOTIFY_COMPLETED, &request_completed, NULL,
                           MHD_OPTION_END);
...
```

Request handling

With all other functions prepared, we can now discuss the actual request handling.

On the first iteration for a new request, we start by allocating a new instance of a `struct connection_info_struct` structure, which will store all necessary information for later iterations and other functions.

```
static enum MHD_Result
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 **req_cls)
{
  if(NULL == *req_cls)
    {
      struct connection_info_struct *con_info;

      con_info = malloc (sizeof (struct connection_info_struct));
      if (NULL == con_info) return MHD_NO;
      con_info->answerstring = NULL;
```

If the new request is a *POST*, the postprocessor must be created now. In addition, the type of the request is stored for convenience.

```
      if (0 == strcmp (method, "POST"))
        {
          con_info->postprocessor
            = MHD_create_post_processor (connection, POSTBUFFERSIZE,
                                         iterate_post, (void*) con_info);

          if (NULL == con_info->postprocessor)
            {
              free (con_info);
              return MHD_NO;
            }
          con_info->connectiontype = POST;
        }
      else con_info->connectiontype = GET;
```

The address of our structure will both serve as the indicator for successive iterations and to remember the particular details about the connection.

```
      *req_cls = (void*) con_info;
      return MHD_YES;
    }
```

The rest of the function will not be executed on the first iteration. A *GET* request is easily satisfied by sending the question form.

```
  if (0 == strcmp (method, "GET"))
    {
      return send_page (connection, askpage);
    }
```

In case of *POST*, we invoke the post processor for as long as data keeps incoming, setting `*upload_data_size` to zero in order to indicate that we have processed—or at least have considered—all of it.

```
  if (0 == strcmp (method, "POST"))
    {
      struct connection_info_struct *con_info = *req_cls;

      if (*upload_data_size != 0)
        {
          MHD_post_process (con_info->postprocessor, upload_data,
                            *upload_data_size);
          *upload_data_size = 0;

          return MHD_YES;
        }
      else if (NULL != con_info->answerstring)
```

```
        return send_page (connection, con_info->answerstring);
    }
```

Finally, if they are neither *GET* nor *POST* requests, the error page is returned.

```
  return send_page(connection, errorpage);
}
```

These were the important parts of the program `simplepost.c`.

7 Improved processing of POST data

The previous chapter introduced a way to upload data to the server, but the developed example program has some shortcomings, such as not being able to handle larger chunks of data. In this chapter, we are going to discuss a more advanced server program that allows clients to upload a file in order to have it stored on the server's filesystem. The server shall also watch and limit the number of clients concurrently uploading, responding with a proper busy message if necessary.

Prepared answers

We choose to operate the server with the `SELECT_INTERNALLY` method. This makes it easier to synchronize the global states at the cost of possible delays for other connections if the processing of a request is too slow. One of these variables that needs to be shared for all connections is the total number of clients that are uploading.

```
#define MAXCLIENTS      2
static unsigned int    nr_of_uploading_clients = 0;
```

If there are too many clients uploading, we want the server to respond to all requests with a busy message.

```
const char* busypage =
  "<html><body>This server is busy, please try again later.</body></html>";
```

Otherwise, the server will send a *form* that informs the user of the current number of uploading clients, and ask her to pick a file on her local filesystem which is to be uploaded.

```
const char* askpage = "<html><body>\n\
                       Upload a file, please!<br>\n\
                       There are %u clients uploading at the moment.<br>\n\
                       <form action=\"/filepost\" method=\"post\" \
                         enctype=\"multipart/form-data\">\n\
                       <input name=\"file\" type=\"file\">\n\
                       <input type=\"submit\" value=\" Send \"></form>\n\
                       </body></html>";
```

If the upload has succeeded, the server will respond with a message saying so.

```
const char* completepage = "<html><body>The upload has been completed.</body></html>";
```

We want the server to report internal errors, such as memory shortage or file access problems, adequately.

```
const char* servererrorpage
  = "<html><body>An internal server error has occurred.</body></html>";
const char* fileexistspage
  = "<html><body>This file already exists.</body></html>";
```

It would be tolerable to send all these responses undifferentiated with a `200 HTTP_OK` status code but in order to improve the `HTTP` conformance of our server a bit, we extend the `send_page` function so that it accepts individual status codes.

```
static enum MHD_Result
send_page (struct MHD_Connection *connection,
           const char* page, int status_code)
{
  enum MHD_Result ret;
  struct MHD_Response *response;

  response = MHD_create_response_from_buffer (strlen (page), (void*) page,
                                              MHD_RESPMEM_MUST_COPY);
  if (!response) return MHD_NO;

  ret = MHD_queue_response (connection, status_code, response);
  MHD_destroy_response (response);

  return ret;
}
```

Note how we ask *MHD* to make its own copy of the message data. The reason behind this will become clear later.

Connection cycle

The decision whether the server is busy or not is made right at the beginning of the connection. To do that at this stage is especially important for *POST* requests because if no response is queued at this point, and `MHD_YES` returned, *MHD* will not sent any queued messages until a postprocessor has been created and the post iterator is called at least once.

```
static enum MHD_Result
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 **req_cls)
{
  if (NULL == *req_cls)
    {
      struct connection_info_struct *con_info;

      if (nr_of_uploading_clients >= MAXCLIENTS)
        return send_page(connection, busypage, MHD_HTTP_SERVICE_UNAVAILABLE);
```

If the server is not busy, the `connection_info` structure is initialized as usual, with the addition of a filepointer for each connection.

```
      con_info = malloc (sizeof (struct connection_info_struct));
      if (NULL == con_info) return MHD_NO;
      con_info->fp = 0;

      if (0 == strcmp (method, "POST"))
        {
          ...
        }
```

```
    else con_info->connectiontype = GET;

    *req_cls = (void*) con_info;

    return MHD_YES;
  }
```

For *POST* requests, the postprocessor is created and we register a new uploading client. From this point on, there are many possible places for errors to occur that make it necessary to interrupt the uploading process. We need a means of having the proper response message ready at all times. Therefore, the `connection_info` structure is extended to hold the most current response message so that whenever a response is sent, the client will get the most informative message. Here, the structure is initialized to "no error".

```
    if (0 == strcmp (method, "POST"))
      {
        con_info->postprocessor
          = MHD_create_post_processor (connection, POSTBUFFERSIZE,
                                       iterate_post, (void*) con_info);

        if (NULL == con_info->postprocessor)
          {
            free (con_info);
            return MHD_NO;
          }

        nr_of_uploading_clients++;

        con_info->connectiontype = POST;
        con_info->answercode = MHD_HTTP_OK;
        con_info->answerstring = completepage;
      }
    else con_info->connectiontype = GET;
```

If the connection handler is called for the second time, *GET* requests will be answered with the *form*. We can keep the buffer under function scope, because we asked *MHD* to make its own copy of it for as long as it is needed.

```
if (0 == strcmp (method, "GET"))
  {
    char buffer[1024];

    sprintf (buffer, askpage, nr_of_uploading_clients);
    return send_page (connection, buffer, MHD_HTTP_OK);
  }
```

The rest of the `answer_to_connection` function is very similar to the `simplepost.c` example, except the more flexible content of the responses. The *POST* data is processed until there is none left and the execution falls through to return an error page if the connection constituted no expected request method.

```
  if (0 == strcmp (method, "POST"))
    {
      struct connection_info_struct *con_info = *req_cls;

      if (0 != *upload_data_size)
        {
          MHD_post_process (con_info->postprocessor,
                            upload_data, *upload_data_size);
          *upload_data_size = 0;

          return MHD_YES;
        }
      else
        return send_page (connection, con_info->answerstring,
                          con_info->answercode);
    }

  return send_page(connection, errorpage, MHD_HTTP_BAD_REQUEST);
}
```

Storing to data

Unlike the `simplepost.c` example, here it is to be expected that post iterator will be called several times now. This means that for any given connection (there might be several concurrent of them) the posted data has to be written to the correct file. That is why we store a file handle in every `connection_info`, so that the it is preserved between successive iterations.

```
static enum MHD_Result
iterate_post (void *coninfo_cls, enum MHD_ValueKind kind,
              const char *key,
              const char *filename, const char *content_type,
              const char *transfer_encoding, const char *data,
              uint64_t off, size_t size)
{
  struct connection_info_struct *con_info = coninfo_cls;
```

Because the following actions depend heavily on correct file processing, which might be error prone, we default to reporting internal errors in case anything will go wrong.

```
con_info->answerstring = servererrorpage;
con_info->answercode = MHD_HTTP_INTERNAL_SERVER_ERROR;
```

In the "askpage" *form*, we told the client to label its post data with the "file" key. Anything else would be an error.

```
  if (0 != strcmp (key, "file")) return MHD_NO;
```

If the iterator is called for the first time, no file will have been opened yet. The `filename` string contains the name of the file (without any paths) the user selected on his system. We want to take this as the name the file will be stored on the server and make sure no file of that name exists (or is being uploaded) before we create one (note that the code below technically contains a race between the two "fopen" calls, but we will overlook this for portability sake).

```
  if (!con_info->fp)
    {
      if (NULL != (fp = fopen (filename, "rb")) )
        {
          fclose (fp);
          con_info->answerstring = fileexistspage;
          con_info->answercode = MHD_HTTP_FORBIDDEN;
          return MHD_NO;
        }

      con_info->fp = fopen (filename, "ab");
      if (!con_info->fp) return MHD_NO;
    }
```

Occasionally, the iterator function will be called even when there are 0 new bytes to process. The server only needs to write data to the file if there is some.

```
if (size > 0)
    {
      if (!fwrite (data, size, sizeof(char), con_info->fp))
        return MHD_NO;
    }
```

If this point has been reached, everything worked well for this iteration and the response can be set to success again. If the upload has finished, this iterator function will not be called again.

```
  con_info->answerstring = completepage;
  con_info->answercode = MHD_HTTP_OK;

  return MHD_YES;
}
```

The new client was registered when the postprocessor was created. Likewise, we unregister the client on destroying the postprocessor when the request is completed.

```
void
request_completed (void *cls, struct MHD_Connection *connection,
                        void **req_cls,
                        enum MHD_RequestTerminationCode toe)
{
  struct connection_info_struct *con_info = *req_cls;

  if (NULL == con_info) return;

  if (con_info->connectiontype == POST)
    {
      if (NULL != con_info->postprocessor)
```

```
        {
          MHD_destroy_post_processor (con_info->postprocessor);
          nr_of_uploading_clients--;
        }

      if (con_info->fp) fclose (con_info->fp);
    }

  free (con_info);
  *req_cls = NULL;
}
```

This is essentially the whole example `largepost.c`.

Remarks

Now that the clients are able to create files on the server, security aspects are becoming even more important than before. Aside from proper client authentication, the server should always make sure explicitly that no files will be created outside of a dedicated upload directory. In particular, filenames must be checked to not contain strings like "../".

8 Session management

This chapter discusses how one should manage sessions, that is, share state between multiple HTTP requests from the same user. We use a simple example where the user submits multiple forms and the server is supposed to accumulate state from all of these forms. Naturally, as this is a network protocol, our session mechanism must support having many users with many concurrent sessions at the same time.

In order to track users, we use a simple session cookie. A session cookie expires when the user closes the browser. Changing from session cookies to persistent cookies only requires adding an expiration time to the cookie. The server creates a fresh session cookie whenever a request without a cookie is received, or if the supplied session cookie is not known to the server.

Looking up the cookie

Since MHD parses the HTTP cookie header for us, looking up an existing cookie is straightforward:

```
const char *value;

value = MHD_lookup_connection_value (connection,
                                     MHD_COOKIE_KIND,
                                     "KEY");
```

Here, "KEY" is the name we chose for our session cookie.

Setting the cookie header

MHD requires the user to provide the full cookie format string in order to set cookies. In order to generate a unique cookie, our example creates a random 64-character text string to be used as the value of the cookie:

```
char value[128];
char raw_value[65];

for (unsigned int i=0;i<sizeof (raw_value);i++)
  raw_value = 'A' + (rand () % 26); /* bad PRNG! */
raw_value[64] = '\0';
snprintf (value, sizeof (value),
          "%s=%s",
          "KEY",
          raw_value);
```

Given this cookie value, we can then set the cookie header in our HTTP response as follows:

```
assert (MHD_YES ==
        MHD_set_connection_value (connection,
                                  MHD_HEADER_KIND,
                                  MHD_HTTP_HEADER_SET_COOKIE,
                                  value));
```

Remark: Session expiration

It is of course possible that clients stop their interaction with the server at any time. In order to avoid using too much storage, the server must thus discard inactive sessions at some point. Our example implements this by discarding inactive sessions after a certain amount of time. Alternatively, the implementation may limit the total number of active sessions. Which bounds are used for idle sessions or the total number of sessions obviously depends largely on the type of the application and available server resources.

Example code

A sample application implementing a website with multiple forms (which are dynamically created using values from previous POST requests from the same session) is available as the example `sessions.c`.

Note that the example uses a simple, $O(n)$ linked list traversal to look up sessions and to expire old sessions. Using a hash table and a heap would be more appropriate if a large number of concurrent sessions is expected.

Remarks

Naturally, it is quite conceivable to store session data in a database instead of in memory. Still, having mechanisms to expire data associated with long-time idle sessions (where the business process has still not finished) is likely a good idea.

9 Adding a layer of security

We left the basic authentication chapter with the unsatisfactory conclusion that any traffic, including the credentials, could be intercepted by anyone between the browser client and the server. Protecting the data while it is sent over unsecured lines will be the goal of this chapter.

Since version 0.4, the *MHD* library includes support for encrypting the traffic by employing SSL/TSL. If *GNU libmicrohttpd* has been configured to support these, encryption and decryption can be applied transparently on the data being sent, with only minimal changes to the actual source code of the example.

Preparation

First, a private key for the server will be generated. With this key, the server will later be able to authenticate itself to the client—preventing anyone else from stealing the password by faking its identity. The *OpenSSL* suite, which is available on many operating systems, can generate such a key. For the scope of this tutorial, we will be content with a 1024 bit key:

```
> openssl genrsa -out server.key 1024
```

In addition to the key, a certificate describing the server in human readable tokens is also needed. This certificate will be attested with our aforementioned key. In this way, we obtain a self-signed certificate, valid for one year.

```
> openssl req -days 365 -out server.pem -new -x509 -key server.key
```

To avoid unnecessary error messages in the browser, the certificate needs to have a name that matches the *URI*, for example, "localhost" or the domain. If you plan to have a publicly reachable server, you will need to ask a trusted third party, called *Certificate Authority*, or *CA*, to attest the certificate for you. This way, any visitor can make sure the server's identity is real.

Whether the server's certificate is signed by us or a third party, once it has been accepted by the client, both sides will be communicating over encrypted channels. From this point on, it is the client's turn to authenticate itself. But this has already been implemented in the basic authentication scheme.

Changing the source code

We merely have to extend the server program so that it loads the two files into memory,

```
int
main ()
{
  struct MHD_Daemon *daemon;
  char *key_pem;
  char *cert_pem;

  key_pem = load_file (SERVERKEYFILE);
  cert_pem = load_file (SERVERCERTFILE);
  if ((key_pem == NULL) || (cert_pem == NULL))
  {
    printf ("The key/certificate files could not be read.\n");
    return 1;
  }
```

and then we point the *MHD* daemon to it upon initialization.

```
  daemon = MHD_start_daemon (MHD_USE_INTERNAL_POLLING_THREAD | MHD_USE_SSL,
                             PORT, NULL, NULL,
                             &answer_to_connection, NULL,
                             MHD_OPTION_HTTPS_MEM_KEY, key_pem,
                             MHD_OPTION_HTTPS_MEM_CERT, cert_pem,
                             MHD_OPTION_END);

  if (NULL == daemon)
    {
      printf ("%s\n", cert_pem);

      free (key_pem);
      free (cert_pem);

      return 1;
    }
```

The rest consists of little new besides some additional memory cleanups.

```
  getchar ();

  MHD_stop_daemon (daemon);
  free (key_pem);
  free (cert_pem);

  return 0;
}
```

The rather unexciting file loader can be found in the complete example `tlsauthentication.c`.

Remarks

- While the standard *HTTP* port is 80, it is 443 for *HTTPS*. The common internet browsers assume standard *HTTP* if they are asked to access other ports than these. Therefore, you will have to type `https://localhost:8888` explicitly when you test the example, or the browser will not know how to handle the answer properly.
- The remaining weak point is the question how the server will be trusted initially. Either a *CA* signs the certificate or the client obtains the key over secure means. Anyway,

the clients have to be aware (or configured) that they should not accept certificates of unknown origin.

- The introduced method of certificates makes it mandatory to set an expiration date—making it less feasible to hardcode certificates in embedded devices.
- The cryptographic facilities consume memory space and computing time. For this reason, websites usually consists both of uncritically *HTTP* parts and secured *HTTPS*.

Client authentication

You can also use MHD to authenticate the client via SSL/TLS certificates (as an alternative to using the password-based Basic or Digest authentication). To do this, you will need to link your application against *gnutls*. Next, when you start the MHD daemon, you must specify the root CA that you're willing to trust:

```
  daemon = MHD_start_daemon (MHD_USE_INTERNAL_POLLING_THREAD | MHD_USE_SSL,
                             PORT, NULL, NULL,
                             &answer_to_connection, NULL,
                             MHD_OPTION_HTTPS_MEM_KEY, key_pem,
                             MHD_OPTION_HTTPS_MEM_CERT, cert_pem,
                             MHD_OPTION_HTTPS_MEM_TRUST, root_ca_pem,
                             MHD_OPTION_END);
```

With this, you can then obtain client certificates for each session. In order to obtain the identity of the client, you first need to obtain the raw GnuTLS session handle from *MHD* using `MHD_get_connection_info`.

```
#include <gnutls/gnutls.h>
#include <gnutls/x509.h>

gnutls_session_t tls_session;
union MHD_ConnectionInfo *ci;

ci = MHD_get_connection_info (connection,
                              MHD_CONNECTION_INFO_GNUTLS_SESSION);
tls_session = (gnutls_session_t) ci->tls_session;
```

You can then extract the client certificate:

```
/**
 * Get the client's certificate
 *
 * @param tls_session the TLS session
 * @return NULL if no valid client certificate could be found, a pointer
 *      to the certificate if found
 */
static gnutls_x509_crt_t
get_client_certificate (gnutls_session_t tls_session)
{
  unsigned int listsize;
  const gnutls_datum_t * pcert;
  gnutls_certificate_status_t client_cert_status;
  gnutls_x509_crt_t client_cert;

  if (tls_session == NULL)
    return NULL;
  if (gnutls_certificate_verify_peers2(tls_session,
                                       &client_cert_status))
    return NULL;
  if (0 != client_cert_status)
  {
    fprintf (stderr,
            "Failed client certificate invalid: %d\n",
            client_cert_status);
    return NULL;
  }
  pcert = gnutls_certificate_get_peers(tls_session,
                                       &listsize);
  if ( (pcert == NULL) ||
       (listsize == 0))
    {
      fprintf (stderr,
              "Failed to retrieve client certificate chain\n");
      return NULL;
    }
  if (gnutls_x509_crt_init(&client_cert))
    {
      fprintf (stderr,
              "Failed to initialize client certificate\n");
      return NULL;
    }
  /* Note that by passing values between 0 and listsize here, you
     can get access to the CA's certs */
  if (gnutls_x509_crt_import(client_cert,
                             &pcert[0],
                             GNUTLS_X509_FMT_DER))
    {
      fprintf (stderr,
              "Failed to import client certificate\n");
      gnutls_x509_crt_deinit(client_cert);
      return NULL;
    }
  return client_cert;
}
```

Using the client certificate, you can then get the client's distinguished name and alternative names:

```
/**
 * Get the distinguished name from the client's certificate
 *
 * @param client_cert the client certificate
 * @return NULL if no dn or certificate could be found, a pointer
 *                      to the dn if found
 */
char *
cert_auth_get_dn(gnutls_x509_crt_t client_cert)
{
  char* buf;
  size_t lbuf;

  lbuf = 0;
  gnutls_x509_crt_get_dn(client_cert, NULL, &lbuf);
  buf = malloc(lbuf);
  if (buf == NULL)
    {
      fprintf (stderr,
               "Failed to allocate memory for certificate dn\n");
      return NULL;
    }
  gnutls_x509_crt_get_dn(client_cert, buf, &lbuf);
  return buf;
}


/**
 * Get the alternative name of specified type from the client's certificate
 *
 * @param client_cert the client certificate
 * @param nametype The requested name type
 * @param index The position of the alternative name if multiple names are
 *                      matching the requested type, 0 for the first matching name
 * @return NULL if no matching alternative name could be found, a pointer
 *                      to the alternative name if found
 */
char *
MHD_cert_auth_get_alt_name(gnutls_x509_crt_t client_cert,
                           int nametype,
                           unsigned int index)
{
  char* buf;
  size_t lbuf;
  unsigned int seq;
  unsigned int subseq;
  unsigned int type;
  int result;
  subseq = 0;
  for (seq=0;;seq++)
    {
      lbuf = 0;
      result = gnutls_x509_crt_get_subject_alt_name2(client_cert, seq, NULL, &lbuf,
                                                     &type, NULL);
      if (result == GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE)
        return NULL;
      if (nametype != (int) type)
        continue;
      if (subseq == index)
        break;
      subseq++;
    }
  buf = malloc(lbuf);
  if (buf == NULL)
    {
      fprintf (stderr,
               "Failed to allocate memory for certificate alt name\n");
      return NULL;
    }
  result = gnutls_x509_crt_get_subject_alt_name2(client_cert,
                                                 seq,
                                                 buf,
                                                 &lbuf,
                                                 NULL, NULL);
  if (result != nametype)
    {
      fprintf (stderr,
               "Unexpected return value from gnutls: %d\n",
               result);
      free (buf);
      return NULL;
    }
  return buf;
}
```

Finally, you should release the memory associated with the client certificate:

```
gnutls_x509_crt_deinit (client_cert);
```

Using TLS Server Name Indication (SNI)

SNI enables hosting multiple domains under one IP address with TLS. So SNI is the TLS-equivalent of virtual hosting. To use SNI with MHD, you need at least GnuTLS 3.0. The main change compared to the simple hosting of one domain is that you need to provide a callback instead of the key and certificate. For example, when you start the MHD daemon, you could do this:

```
  daemon = MHD_start_daemon (MHD_USE_INTERNAL_POLLING_THREAD | MHD_USE_SSL,
                             PORT, NULL, NULL,
                             &answer_to_connection, NULL,
                             MHD_OPTION_HTTPS_CERT_CALLBACK, &sni_callback,
                             MHD_OPTION_END);
```

Here, `sni_callback` is the name of a function that you will have to implement to retrieve the X.509 certificate for an incoming connection. The callback has type `gnutls_certificate_retrieve_function2` and is documented in the GnuTLS API for the `gnutls_certificate_set_retrieve_function2` as follows:

`int *gnutls_certificate_retrieve_function2` [Function Pointer]
`(gnutls_session_t, const gnutls_datum_t* req_ca_dn, int nreqs, const gnutls_pk_algorithm_t* pk_algos, int pk_algos_length, gnutls_pcert_st** pcert, unsigned int *pcert_length, gnutls_privkey_t * pkey)`

req_ca_cert
: is only used in X.509 certificates. Contains a list with the CA names that the server considers trusted. Normally we should send a certificate that is signed by one of these CAs. These names are DER encoded. To get a more meaningful value use the function `gnutls_x509_rdn_get()`.

pk_algos
: contains a list with server's acceptable signature algorithms. The certificate returned should support the server's given algorithms.

pcert
: should contain a single certificate and public or a list of them.

pcert_length
: is the size of the previous list.

pkey
: is the private key.

A possible implementation of this callback would look like this:

```
struct Hosts
{
  struct Hosts *next;
  const char *hostname;
  gnutls_pcert_st pcrt;
  gnutls_privkey_t key;
};

static struct Hosts *hosts;

int
sni_callback (gnutls_session_t session,
              const gnutls_datum_t* req_ca_dn,
              int nreqs,
              const gnutls_pk_algorithm_t* pk_algos,
              int pk_algos_length,
              gnutls_pcert_st** pcert,
              unsigned int *pcert_length,
              gnutls_privkey_t * pkey)
{
  char name[256];
  size_t name_len;
  struct Hosts *host;
  unsigned int type;

  name_len = sizeof (name);
  if (GNUTLS_E_SUCCESS !=
      gnutls_server_name_get (session,
                              name,
                              &name_len,
                              &type,
                              0 /* index */))
    return -1;
  for (host = hosts; NULL != host; host = host->next)
    if (0 == strncmp (name, host->hostname, name_len))
      break;
  if (NULL == host)
    {
      fprintf (stderr,
               "Need certificate for %.*s\n",
               (int) name_len,
               name);
      return -1;
    }
  fprintf (stderr,
           "Returning certificate for %.*s\n",
           (int) name_len,
           name);
  *pkey = host->key;
  *pcert_length = 1;
  *pcert = &host->pcrt;
  return 0;
}
```

Note that MHD cannot offer passing a closure or any other additional information to this callback, as the GnuTLS API unfortunately does not permit this at this point.

The `hosts` list can be initialized by loading the private keys and X.509 certificates from disk as follows:

```
static void
load_keys(const char *hostname,
          const char *CERT_FILE,
          const char *KEY_FILE)
{
  int ret;
  gnutls_datum_t data;
```

```
struct Hosts *host;

host = malloc (sizeof (struct Hosts));
host->hostname = hostname;
host->next = hosts;
hosts = host;

ret = gnutls_load_file (CERT_FILE, &data);
if (ret < 0)
{
  fprintf (stderr,
           "*** Error loading certificate file %s.\n",
           CERT_FILE);
  exit(1);
}
ret =
  gnutls_pcert_import_x509_raw (&host->pcrt, &data, GNUTLS_X509_FMT_PEM,
                                0);
if (ret < 0)
{
  fprintf(stderr,
          "*** Error loading certificate file: %s\n",
          gnutls_strerror (ret));
  exit(1);
}
gnutls_free (data.data);

ret = gnutls_load_file (KEY_FILE, &data);
if (ret < 0)
{
  fprintf (stderr,
           "*** Error loading key file %s.\n",
           KEY_FILE);
  exit(1);
}

gnutls_privkey_init (&host->key);
ret =
  gnutls_privkey_import_x509_raw (host->key,
                                  &data, GNUTLS_X509_FMT_PEM,
                                  NULL, 0);
if (ret < 0)
{
  fprintf (stderr,
           "*** Error loading key file: %s\n",
           gnutls_strerror (ret));
  exit(1);
  }
  gnutls_free (data.data);
}
```

The code above was largely lifted from GnuTLS. You can find other methods for initializing certificates and keys in the GnuTLS manual and source code.

10 Websockets

Websockets are a genuine way to implement push notifications, where the server initiates the communication while the client can be idle. Usually a HTTP communication is half-duplex and always requested by the client, but websockets are full-duplex and only initialized by the client. In the further communication both sites can use the websocket at any time to send data to the other site.

To initialize a websocket connection the client sends a special HTTP request to the server and initializes a handshake between client and server which switches from the HTTP protocol to the websocket protocol. Thus both the server as well as the client must support websockets. If proxys are used, they must support websockets too. In this chapter we take a look on server and client, but with a focus on the server with *libmicrohttpd*.

Since version 0.9.52 *libmicrohttpd* supports upgrading requests, which is required for switching from the HTTP protocol. Since version 0.9.74 the library *libmicrohttpd_ws* has been added to support the websocket protocol.

Upgrading connections with libmicrohttpd

To support websockets we need to enable upgrading of HTTP connections first. This is done by passing the flag `MHD_ALLOW_UPGRADE` to `MHD_start_daemon()`.

```
daemon = MHD_start_daemon (MHD_USE_INTERNAL_POLLING_THREAD |
                           MHD_USE_THREAD_PER_CONNECTION |
                           MHD_ALLOW_UPGRADE |
                           MHD_USE_ERROR_LOG,
                           PORT, NULL, NULL,
                           &access_handler, NULL,
                           MHD_OPTION_END);
```

The next step is to turn a specific request into an upgraded connection. This done in our `access_handler` by calling `MHD_create_response_for_upgrade()`. An `upgrade_handler` will be passed to perform the low-level actions on the socket.

Please note that the socket here is just a regular socket as provided by the operating system. To use it as a websocket, some more steps from the following chapters are required.

```
static enum MHD_Result
access_handler (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 **ptr)
{
  /* ... */
  /* some code to decide whether to upgrade or not */
  /* ... */
  /* create the response for upgrade */
  response = MHD_create_response_for_upgrade (&upgrade_handler,
                                              NULL);

  /* ... */
  /* additional headers, etc. */
  /* ... */

  ret = MHD_queue_response (connection,
                            MHD_HTTP_SWITCHING_PROTOCOLS,
                            response);
  MHD_destroy_response (response);

  return ret;
}
```

In the `upgrade_handler` we receive the low-level socket, which is used for the communication with the specific client. In addition to the low-level socket we get:

- Some data, which has been read too much while *libmicrohttpd* was switching the protocols. This value is usually empty, because it would mean that the client has sent data before the handshake was complete.
- A `struct MHD_UpgradeResponseHandle` which is used to perform special actions like closing, corking or uncorking the socket. These commands are executed by passing the handle to `MHD_upgrade_action()`.

Depending of the flags specified while calling `MHD_start_deamon()` our `upgrade_handler` is either executed in the same thread as our daemon or in a thread specific for each connection. If it is executed in the same thread then `upgrade_handler` is a blocking call for our webserver and we should finish it as fast as possible (i. e. by creating a thread and passing the information there). If `MHD_USE_THREAD_PER_CONNECTION` was passed to `MHD_start_daemon()` then a separate thread is used and thus our `upgrade_handler` needs not to start a separate thread.

An `upgrade_handler`, which is called with a separate thread per connection, could look like this:

```
static void
upgrade_handler (void *cls,
                 struct MHD_Connection *connection,
                 void *req_cls,
                 const char *extra_in,
                 size_t extra_in_size,
                 MHD_socket fd,
                 struct MHD_UpgradeResponseHandle *urh)
{
  /* ... */
  /* do something with the socket `fd` like `recv()` or `send()` */
  /* ... */
  /* close the socket when it is not needed anymore */
  MHD_upgrade_action (urh,
                      MHD_UPGRADE_ACTION_CLOSE);
}
```

This is all you need to know for upgrading connections with *libmicrohttpd*. The next chapters focus on using the websocket protocol with *libmicrohttpd_ws*.

Websocket handshake with libmicrohttpd_ws

To request a websocket connection the client must send the following information with the HTTP request:

- A `GET` request must be sent.
- The version of the HTTP protocol must be 1.1 or higher.
- A `Host` header field must be sent
- A `Upgrade` header field containing the keyword "websocket" (case-insensitive). Please note that the client could pass multiple protocols separated by comma.
- A `Connection` header field that includes the token "Upgrade" (case-insensitive). Please note that the client could pass multiple tokens separated by comma.
- A `Sec-WebSocket-Key` header field with a base64-encoded value. The decoded the value is 16 bytes long and has been generated randomly by the client.
- A `Sec-WebSocket-Version` header field with the value "13".

Optionally the client can also send the following information:

- A `Origin` header field can be used to determine the source of the client (i. e. the website).
- A `Sec-WebSocket-Protocol` header field can contain a list of supported protocols by the client, which can be sent over the websocket.
- A `Sec-WebSocket-Extensions` header field which may contain extensions to the websocket protocol. The extensions must be registered by IANA.

A valid example request from the client could look like this:

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Sec-WebSocket-Version: 13
```

To complete the handshake the server must respond with some specific response headers:

- The HTTP response code `101 Switching Protocols` must be answered.
- An `Upgrade` header field containing the value "websocket" must be sent.
- A `Connection` header field containing the value "Upgrade" must be sent.
- A `Sec-WebSocket-Accept` header field containing a value, which has been calculated from the `Sec-WebSocket-Key` request header field, must be sent.

Optionally the server may send following headers:

- A `Sec-WebSocket-Protocol` header field containing a protocol of the list specified in the corresponding request header field.
- A `Sec-WebSocket-Extension` header field containing all used extensions of the list specified in the corresponding request header field.

A valid websocket HTTP response could look like this:

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
```

To upgrade a connection to a websocket the *libmicrohttpd_ws* provides some helper functions for the `access_handler` callback function:

- `MHD_websocket_check_http_version()` checks whether the HTTP version is 1.1 or above.
- `MHD_websocket_check_connection_header()` checks whether the value of the `Connection` request header field contains an "Upgrade" token (case-insensitive).
- `MHD_websocket_check_upgrade_header()` checks whether the value of the `Upgrade` request header field contains the "websocket" keyword (case-insensitive).
- `MHD_websocket_check_version_header()` checks whether the value of the `Sec-WebSocket-Version` request header field is "13".
- `MHD_websocket_create_accept_header()` takes the value from the `Sec-WebSocket-Key` request header and calculates the value for the `Sec-WebSocket-Accept` response header field.

The `access_handler` example of the previous chapter can now be extended with these helper functions to perform the websocket handshake:

```
static enum MHD_Result
access_handler (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 **ptr)
{
  static int aptr;
  struct MHD_Response *response;
  enum MHD_Result ret;

  (void) cls;               /* Unused. Silent compiler warning. */
  (void) upload_data;       /* Unused. Silent compiler warning. */
  (void) upload_data_size;  /* Unused. Silent compiler warning. */
```

```
if (0 != strcmp (method, "GET"))
  return MHD_NO;              /* unexpected method */
if (&aptr != *ptr)
{
  /* do never respond on first call */
  *ptr = &aptr;
  return MHD_YES;
}
*ptr = NULL;                  /* reset when done */

if (0 == strcmp (url, "/"))
{
  /* Default page for visiting the server */
  struct MHD_Response *response = MHD_create_response_from_buffer (
                                   strlen (PAGE),
                                   PAGE,
                                   MHD_RESPMEM_PERSISTENT);
  ret = MHD_queue_response (connection,
                            MHD_HTTP_OK,
                            response);
  MHD_destroy_response (response);
}
else if (0 == strcmp (url, "/chat"))
{
  char is_valid = 1;
  const char* value = NULL;
  char sec_websocket_accept[29];

  if (0 != MHD_websocket_check_http_version (version))
  {
    is_valid = 0;
  }
  value = MHD_lookup_connection_value (connection,
                                       MHD_HEADER_KIND,
                                       MHD_HTTP_HEADER_CONNECTION);
  if (0 != MHD_websocket_check_connection_header (value))
  {
    is_valid = 0;
  }
  value = MHD_lookup_connection_value (connection,
                                       MHD_HEADER_KIND,
                                       MHD_HTTP_HEADER_UPGRADE);
  if (0 != MHD_websocket_check_upgrade_header (value))
  {
    is_valid = 0;
  }
  value = MHD_lookup_connection_value (connection,
                                       MHD_HEADER_KIND,
                                       MHD_HTTP_HEADER_SEC_WEBSOCKET_VERSION);
  if (0 != MHD_websocket_check_version_header (value))
  {
    is_valid = 0;
  }
  value = MHD_lookup_connection_value (connection,
                                       MHD_HEADER_KIND,
                                       MHD_HTTP_HEADER_SEC_WEBSOCKET_KEY);
  if (0 != MHD_websocket_create_accept_header (value, sec_websocket_accept))
  {
    is_valid = 0;
  }

  if (1 == is_valid)
  {
    /* upgrade the connection */
    response = MHD_create_response_for_upgrade (&upgrade_handler,
                                                NULL);
    MHD_add_response_header (response,
                             MHD_HTTP_HEADER_CONNECTION,
                             "Upgrade");
    MHD_add_response_header (response,
                             MHD_HTTP_HEADER_UPGRADE,
                             "websocket");
    MHD_add_response_header (response,
                             MHD_HTTP_HEADER_SEC_WEBSOCKET_ACCEPT,
                             sec_websocket_accept);
    ret = MHD_queue_response (connection,
                              MHD_HTTP_SWITCHING_PROTOCOLS,
                              response);
    MHD_destroy_response (response);
  }
  else
  {
    /* return error page */
    struct MHD_Response*response = MHD_create_response_from_buffer (
                                     strlen (PAGE_INVALID_WEBSOCKET_REQUEST),
                                     PAGE_INVALID_WEBSOCKET_REQUEST,
                                     MHD_RESPMEM_PERSISTENT);
    ret = MHD_queue_response (connection,
                              MHD_HTTP_BAD_REQUEST,
                              response);
    MHD_destroy_response (response);
  }
}
else
{
  struct MHD_Response*response = MHD_create_response_from_buffer (
                                   strlen (PAGE_NOT_FOUND),
                                   PAGE_NOT_FOUND,
                                   MHD_RESPMEM_PERSISTENT);
  ret = MHD_queue_response (connection,
                            MHD_HTTP_NOT_FOUND,
                            response);
  MHD_destroy_response (response);
}

return ret;
}
```

Please note that we skipped the check of the Host header field here, because we don't know the host for this example.

Decoding/encoding the websocket protocol with libmicrohttpd_ws

Once the websocket connection is established you can receive/send frame data with the low-level socket functions `recv()` and `send()`. The frame data which goes over the low-level socket is encoded according to the websocket protocol. To use received payload data, you need to decode the frame data first. To send payload data, you need to encode it into frame data first.

libmicrohttpd_ws provides several functions for encoding of payload data and decoding of frame data:

- `MHD_websocket_decode()` decodes received frame data. The payload data may be of any kind, depending upon what the client has sent. So this decode function is used for all kind of frames and returns the frame type along with the payload data.
- `MHD_websocket_encode_text()` encodes text. The text must be encoded with UTF-8.
- `MHD_websocket_encode_binary()` encodes binary data.
- `MHD_websocket_encode_ping()` encodes a ping request to check whether the websocket is still valid and to test latency.
- `MHD_websocket_encode_ping()` encodes a pong response to answer a received ping request.
- `MHD_websocket_encode_close()` encodes a close request.
- `MHD_websocket_free()` frees data returned by the encode/decode functions.

Since you could receive or send fragmented data (i. e. due to a too small buffer passed to `recv`) all of these encode/decode functions require a pointer to a `struct MHD_WebSocketStream` passed as argument. In this structure *libmicrohttpd_ws* stores information about encoding/decoding of the particular websocket. For each websocket you need a unique `struct MHD_WebSocketStream` to encode/decode with this library.

To create or destroy `struct MHD_WebSocketStream` we have additional functions:

- `MHD_websocket_stream_init()` allocates and initializes a new `struct MHD_WebSocketStream`. You can specify some options here to alter the behavior of the websocket stream.
- `MHD_websocket_stream_free()` frees a previously allocated `struct MHD_WebSocketStream`.

With these encode/decode functions we can improve our `upgrade_handler` callback function from an earlier example to a working websocket:

```
static void
upgrade_handler (void *cls,
                 struct MHD_Connection *connection,
                 void *req_cls,
                 const char *extra_in,
                 size_t extra_in_size,
                 MHD_socket fd,
                 struct MHD_UpgradeResponseHandle *urh)
{
  /* make the socket blocking (operating-system-dependent code) */
  make_blocking (fd);

  /* create a websocket stream for this connection */
  struct MHD_WebSocketStream* ws;
  int result = MHD_websocket_stream_init (&ws,
                                          0,
                                          0);
  if (0 != result)
  {
    /* Couldn't create the websocket stream.
     * So we close the socket and leave
     */
    MHD_upgrade_action (urh,
                        MHD_UPGRADE_ACTION_CLOSE);
    return;
  }

  /* Let's wait for incoming data */
  const size_t buf_len = 256;
  char buf[buf_len];
  ssize_t got;
  while (MHD_WEBSOCKET_VALIDITY_VALID == MHD_websocket_stream_is_valid (ws))
  {
    got = recv (fd,
                buf,
                buf_len,
                0);
    if (0 >= got)
    {
      /* the TCP/IP socket has been closed */
      break;
    }

    /* parse the entire received data */
    size_t buf_offset = 0;
    while (buf_offset < (size_t) got)
    {
      size_t new_offset = 0;
      char *frame_data = NULL;
      size_t frame_len  = 0;
      int status = MHD_websocket_decode (ws,
                                         buf + buf_offset,
                                         ((size_t) got) - buf_offset,
                                         &new_offset,
                                         &frame_data,
                                         &frame_len);
      if (0 > status)
      {
        /* an error occurred and the connection must be closed */
        if (NULL != frame_data)
        {
          MHD_websocket_free (ws, frame_data);
        }
        break;
      }
      else
      {
        buf_offset += new_offset;
        if (0 < status)
        {
          /* the frame is complete */
          switch (status)
          {
          case MHD_WEBSOCKET_STATUS_TEXT_FRAME:
            /* The client has sent some text.
             * We will display it and answer with a text frame.
             */
            if (NULL != frame_data)
            {
              printf ("Received message: %s\n", frame_data);
              MHD_websocket_free (ws, frame_data);
              frame_data = NULL;
            }
            result = MHD_websocket_encode_text (ws,
                                                "Hello",
                                                5,  /* length of "Hello" */
```

```
                                           0,
                                           &frame_data,
                                           &frame_len,
                                           NULL);
            if (0 == result)
            {
              send_all (fd,
                        frame_data,
                        frame_len);
            }
            break;

          case MHD_WEBSOCKET_STATUS_CLOSE_FRAME:
            /* if we receive a close frame, we will respond with one */
            MHD_websocket_free (ws,
                                frame_data);
            frame_data = NULL;

            result = MHD_websocket_encode_close (ws,
                                                 0,
                                                 NULL,
                                                 0,
                                                 &frame_data,
                                                 &frame_len);
            if (0 == result)
            {
              send_all (fd,
                        frame_data,
                        frame_len);
            }
            break;

          case MHD_WEBSOCKET_STATUS_PING_FRAME:
            /* if we receive a ping frame, we will respond */
            /* with the corresponding pong frame */
            {
              char *pong = NULL;
              size_t pong_len = 0;
              result = MHD_websocket_encode_pong (ws,
                                                  frame_data,
                                                  frame_len,
                                                  &pong,
                                                  &pong_len);
              if (0 == result)
              {
                send_all (fd,
                          pong,
                          pong_len);
              }
              MHD_websocket_free (ws,
                                  pong);
            }
            break;

          default:
            /* Other frame types are ignored
             * in this minimal example.
             * This is valid, because they become
             * automatically skipped if we receive them unexpectedly
             */
            break;
          }
        }
        if (NULL != frame_data)
        {
          MHD_websocket_free (ws, frame_data);
        }
      }
    }
  }

  /* free the websocket stream */
  MHD_websocket_stream_free (ws);

  /* close the socket when it is not needed anymore */
  MHD_upgrade_action (urh,
                      MHD_UPGRADE_ACTION_CLOSE);
}

/* This helper function is used for the case that
 * we need to resend some data
 */
static void
send_all (MHD_socket fd,
          const char *buf,
          size_t len)
{
  ssize_t ret;
  size_t off;

  for (off = 0; off < len; off += ret)
  {
    ret = send (fd,
                &buf[off],
                (int) (len - off),
                0);
    if (0 > ret)
    {
      if (EAGAIN == errno)
      {
        ret = 0;
        continue;
      }
      break;
    }
    if (0 == ret)
      break;
  }
}

/* This helper function contains operating-system-dependent code and
 * is used to make a socket blocking.
 */
static void
make_blocking (MHD_socket fd)
{
#if defined(MHD_POSIX_SOCKETS)
  int flags;

  flags = fcntl (fd, F_GETFL);
  if (-1 == flags)
    return;
  if ((flags & ~O_NONBLOCK) != flags)
    if (-1 == fcntl (fd, F_SETFL, flags & ~O_NONBLOCK))
      abort ();
#elif defined(MHD_WINSOCK_SOCKETS)
  unsigned long flags = 0;

  ioctlsocket (fd, FIONBIO, &flags);
#endif /* MHD_WINSOCK_SOCKETS */
}
```

Please note that the websocket in this example is only half-duplex. It waits until the blocking `recv()` call returns and only does then something. In this example all frame types are decoded by *libmicrohttpd_ws*, but we only do something when a text, ping or close frame is received. Binary and pong frames are ignored in our code. This is legit, because the server is only required to implement at least support for ping frame or close frame (the other frame types could be skipped in theory, because they don't require an answer). The pong frame doesn't require an answer and whether text frames or binary frames get an answer simply belongs to your server application. So this is a valid minimal example.

Until this point you've learned everything you need to basically use websockets with *libmicrohttpd* and *libmicrohttpd_ws*. These libraries offer much more functions for some specific cases.

The further chapters of this tutorial focus on some specific problems and the client site programming.

Using full-duplex websockets

To use full-duplex websockets you can simply create two threads per websocket connection. One of these threads is used for receiving data with a blocking `recv()` call and the other thread is triggered by the application internal codes and sends the data.

A full-duplex websocket example is implemented in the example file `websocket_chatserver_example.c`.

Error handling

The most functions of *libmicrohttpd_ws* return a value of `enum MHD_WEBSOCKET_STATUS`. The values of this enumeration can be converted into an integer and have an easy interpretation:

- If the value is less than zero an error occurred and the call has failed. Check the enumeration values for more specific information.
- If the value is equal to zero, the call succeeded.
- If the value is greater than zero, the call succeeded and the value specifies the decoded frame type. Currently positive values are only returned by `MHD_websocket_decode()` (of the functions with this return enumeration type).

A websocket stream can also get broken when invalid frame data is received. Also the other site could send a close frame which puts the stream into a state where it may not be used for regular communication. Whether a stream has become broken, can be checked with `MHD_websocket_stream_is_valid()`.

Fragmentation

In addition to the regular TCP/IP fragmentation the websocket protocol also supports fragmentation. Fragmentation could be used for continuous payload data such as video data from a webcam. Whether or not you want to receive fragmentation is specified upon initialization of the websocket stream. If you pass `MHD_WEBSOCKET_FLAG_WANT_FRAGMENTS` in the flags parameter of `MHD_websocket_stream_init()` then you can receive fragments. If you don't pass this flag (in the most cases you just pass zero as flags) then you don't want to handle fragments on your own. *libmicrohttpd_ws* removes then the fragmentation for you in the background. You only get the completely assembled frames.

Upon encoding you specify whether or not you want to create a fragmented frame by passing a flag to the corresponding encode function. Only `MHD_websocket_encode_text()` and `MHD_websocket_encode_binary()` can be used for fragmentation, because the other frame types may not be fragmented. Encoding fragmented frames is independent of the `MHD_WEBSOCKET_FLAG_WANT_FRAGMENTS` flag upon initialization.

Quick guide to websockets in JavaScript

Websockets are supported in all modern web browsers. You initialize a websocket connection by creating an instance of the `WebSocket` class provided by the web browser.

There are some simple rules for using websockets in the browser:

- When you initialize the instance of the websocket class you must pass an URL. The URL must either start with `ws://` (for not encrypted websocket protocol) or `wss://` (for TLS-encrypted websocket protocol).

  IMPORTANT: If your website is accessed via `https://` then you are in a security context, which means that you are only allowed to access other secure protocols. So you can only use `wss://` for websocket connections then. If you try to `ws://` instead then your websocket connection will automatically fail.
- The WebSocket class uses events to handle the receiving of data. JavaScript is per definition a single-threaded language so the receiving events will never overlap. Sending is done directly by calling a method of the instance of the WebSocket class.

Here is a short example for receiving/sending data to the same host as the website is running on:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Websocket Demo</title>
<script>

let url = 'ws' + (window.location.protocol === 'https:' ? 's' : '') + '://' +
          window.location.host + '/chat';
let socket = null;

window.onload = function(event) {
  socket = new WebSocket(url);
  socket.onopen = function(event) {
    document.write('The websocket connection has been established.<br>');

    // Send some text
    socket.send('Hello from JavaScript!');
  }

  socket.onclose = function(event) {
    document.write('The websocket connection has been closed.<br>');
  }

  socket.onerror = function(event) {
    document.write('An error occurred during the websocket communication.<br>');
  }

  socket.onmessage = function(event) {
    document.write('Websocket message received: ' + event.data + '<br>');
  }
}

</script>
</head>
<body>
</body>
</html>
```

Appendix A Bibliography

API reference

- The *GNU libmicrohttpd* manual by Marco Maggi and Christian Grothoff 2008 `http://gnunet.org/libmicrohttpd/microhttpd.html`
- All referenced RFCs can be found on the website of *The Internet Engineering Task Force* `http://www.ietf.org/`
- *RFC 2616*: Fielding, R., Gettys, J., Mogul, J., Frystyk, H., and T. Berners-Lee, "Hypertext Transfer Protocol – HTTP/1.1", RFC 2016, January 1997.
- *RFC 2617*: Franks, J., Hallam-Baker, P., Hostetler, J., Lawrence, S., Leach, P., Luotonen, A., and L. Stewart, "HTTP Authentication: Basic and Digest Access Authentication", RFC 2617, June 1999.
- *RFC 6455*: Fette, I., Melnikov, A., "The WebSocket Protocol", RFC 6455, December 2011.
- A well–structured *HTML* reference can be found on `http://www.echoecho.com/html.htm`

  For those readers understanding German or French, there is an excellent document both for learning *HTML* and for reference, whose English version unfortunately has been discontinued. `http://de.selfhtml.org/` and `http://fr.selfhtml.org/`

Appendix B GNU Free Documentation License

Version 1.3, 3 November 2008

Copyright © 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc.
`http://fsf.org/`

Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

0. PREAMBLE

   The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

   This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

   We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

   This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

   A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

   A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

   The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

   If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

   If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

   If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

   It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

   You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

   A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any,

be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their

titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

   You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

   The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

   In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an “aggregate” if the copyright resulting from the compilation is not used to limit the legal rights of the compilation’s users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

   If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document’s Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

   If a section in the Document is Entitled “Acknowledgements”, “Dedications”, or “History”, the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

   However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

   Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

   Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `http://www.gnu.org/copyleft/`.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

    "Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

    "CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

    "Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

    An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

    The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.3
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
Texts.  A copy of the license is included in the section entitled ``GNU
Free Documentation License''.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with... Texts." line with this:

```
with the Invariant Sections being list their titles, with
the Front-Cover Texts being list, and with the Back-Cover Texts
being list.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Appendix C Example programs

C.1 hellobrowser.c

```
/* Feel free to use this example code in any way
   you see fit (Public Domain) */

#include <sys/types.h>
#ifndef _WIN32
#include <sys/select.h>
#include <sys/socket.h>
#else
#include <winsock2.h>
#endif
#include <string.h>
#include <microhttpd.h>
#include <stdio.h>

#define PORT 8888

static enum MHD_Result
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 **req_cls)
{
  const char *page = "<html><body>Hello, browser!</body></html>";
  struct MHD_Response *response;
  enum MHD_Result ret;
  (void) cls;               /* Unused. Silent compiler warning. */
  (void) url;               /* Unused. Silent compiler warning. */
  (void) method;            /* Unused. Silent compiler warning. */
  (void) version;           /* Unused. Silent compiler warning. */
  (void) upload_data;       /* Unused. Silent compiler warning. */
  (void) upload_data_size;  /* Unused. Silent compiler warning. */
  (void) req_cls;           /* Unused. Silent compiler warning. */

  response = MHD_create_response_from_buffer_static (strlen (page), page);
  ret = MHD_queue_response (connection, MHD_HTTP_OK, response);
  MHD_destroy_response (response);

  return ret;
}


int
main (void)
{
  struct MHD_Daemon *daemon;

  daemon = MHD_start_daemon (MHD_USE_AUTO | MHD_USE_INTERNAL_POLLING_THREAD,
                             PORT, NULL, NULL,
                             &answer_to_connection, NULL, MHD_OPTION_END);
  if (NULL == daemon)
    return 1;

  (void) getchar ();
  MHD_stop_daemon (daemon);
  return 0;
}
```

C.2 logging.c

```
/* Feel free to use this example code in any way
   you see fit (Public Domain) */

#include <sys/types.h>
#ifndef _WIN32
#include <sys/select.h>
#include <sys/socket.h>
#else
#include <winsock2.h>
#endif
#include <microhttpd.h>
#include <stdio.h>

#define PORT 8888


static enum MHD_Result
print_out_key (void *cls, enum MHD_ValueKind kind, const char *key,
               const char *value)
{
  (void) cls;    /* Unused. Silent compiler warning. */
  (void) kind;   /* Unused. Silent compiler warning. */
  printf ("%s: %s\n", key, value);
  return MHD_YES;
}


static enum MHD_Result
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 **req_cls)
{
  (void) cls;               /* Unused. Silent compiler warning. */
  (void) version;           /* Unused. Silent compiler warning. */
  (void) upload_data;       /* Unused. Silent compiler warning. */
  (void) upload_data_size;  /* Unused. Silent compiler warning. */
  (void) req_cls;           /* Unused. Silent compiler warning. */
  printf ("New %s request for %s using version %s\n", method, url, version);

  MHD_get_connection_values (connection, MHD_HEADER_KIND, print_out_key,
                             NULL);

  return MHD_NO;
}


int
main (void)
{
  struct MHD_Daemon *daemon;
  daemon = MHD_start_daemon (MHD_USE_INTERNAL_POLLING_THREAD, PORT, NULL, NULL,
                             &answer_to_connection, NULL, MHD_OPTION_END);
  if (NULL == daemon)
    return 1;

  (void) getchar ();

  MHD_stop_daemon (daemon);
  return 0;
}
```

C.3 responseheaders.c

```
/* Feel free to use this example code in any way
   you see fit (Public Domain) */

#include <sys/types.h>
#ifndef _WIN32
#include <sys/select.h>
#include <sys/socket.h>
#else
#include <winsock2.h>
#endif
#include <microhttpd.h>
#include <time.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <stdio.h>

#define PORT 8888
#define FILENAME "picture.png"
#define MIMETYPE "image/png"

static enum MHD_Result
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 **req_cls)
{
  struct MHD_Response *response;
  int fd;
  enum MHD_Result ret;
  struct stat sbuf;
  (void) cls;               /* Unused. Silent compiler warning. */
  (void) url;               /* Unused. Silent compiler warning. */
  (void) version;           /* Unused. Silent compiler warning. */
  (void) upload_data;       /* Unused. Silent compiler warning. */
  (void) upload_data_size;  /* Unused. Silent compiler warning. */
  (void) req_cls;           /* Unused. Silent compiler warning. */

  if (0 != strcmp (method, "GET"))
    return MHD_NO;

  if ( (-1 == (fd = open (FILENAME, O_RDONLY))) ||
       (0 != fstat (fd, &sbuf)) )
  {
    const char *errorstr =
      "<html><body>An internal server error has occurred!\
                              </body></html>";
    /* error accessing file */
    if (fd != -1)
      (void) close (fd);
    response =
      MHD_create_response_from_buffer_static (strlen (errorstr), errorstr);
    if (NULL != response)
    {
      ret =
        MHD_queue_response (connection, MHD_HTTP_INTERNAL_SERVER_ERROR,
                            response);
      MHD_destroy_response (response);

      return ret;
    }
    else
      return MHD_NO;
  }
  response =
    MHD_create_response_from_fd_at_offset64 ((size_t) sbuf.st_size,
                                              fd,
                                              0);
  if (MHD_YES !=
      MHD_add_response_header (response,
                               MHD_HTTP_HEADER_CONTENT_TYPE,
                               MIMETYPE))
  {
    fprintf (stderr,
             "Failed to set content type header!\n");
    /* return response without content encoding anyway ... */
  }
  ret = MHD_queue_response (connection,
                            MHD_HTTP_OK,
                            response);
  MHD_destroy_response (response);

  return ret;
}


int
main (void)
{
  struct MHD_Daemon *daemon;

  daemon = MHD_start_daemon (MHD_USE_INTERNAL_POLLING_THREAD, PORT, NULL, NULL,
                             &answer_to_connection, NULL, MHD_OPTION_END);
  if (NULL == daemon)
    return 1;

  (void) getchar ();

  MHD_stop_daemon (daemon);

  return 0;
}
```

C.4 basicauthentication.c

```
/* Feel free to use this example code in any way
   you see fit (Public Domain) */

#include <sys/types.h>
#ifndef _WIN32
#include <sys/select.h>
#include <sys/socket.h>
#else
#include <winsock2.h>
#endif
#include <microhttpd.h>
#include <time.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

#define PORT 8888


static enum MHD_Result
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 **req_cls)
{
  struct MHD_BasicAuthInfo *auth_info;
  enum MHD_Result ret;
  struct MHD_Response *response;
  (void) cls;               /* Unused. Silent compiler warning. */
  (void) url;               /* Unused. Silent compiler warning. */
  (void) version;           /* Unused. Silent compiler warning. */
  (void) upload_data;       /* Unused. Silent compiler warning. */
  (void) upload_data_size;  /* Unused. Silent compiler warning. */

  if (0 != strcmp (method, "GET"))
    return MHD_NO;
  if (NULL == *req_cls)
  {
    *req_cls = connection;
    return MHD_YES;
  }
  auth_info = MHD_basic_auth_get_username_password3 (connection);
  if (NULL == auth_info)
  {
    static const char *page =
      "<html><body>Authorization required</body></html>";
    response = MHD_create_response_from_buffer_static (strlen (page), page);
    ret = MHD_queue_basic_auth_required_response3 (connection,
                                                   "admins",
                                                   MHD_YES,
                                                   response);
  }
  else if ((strlen ("root") != auth_info->username_len) ||
           (0 != memcmp (auth_info->username, "root",
                         auth_info->username_len)) ||
           /* The next check against NULL is optional,
            * if 'password' is NULL then 'password_len' is always zero. */
           (NULL == auth_info->password) ||
           (strlen ("pa$$w0rd") != auth_info->password_len) ||
           (0 != memcmp (auth_info->password, "pa$$w0rd",
                         auth_info->password_len)))
  {
    static const char *page =
      "<html><body>Wrong username or password</body></html>";
    response = MHD_create_response_from_buffer_static (strlen (page), page);
    ret = MHD_queue_basic_auth_required_response3 (connection,
                                                   "admins",
                                                   MHD_YES,
                                                   response);
  }
  else
  {
    static const char *page = "<html><body>A secret.</body></html>";
    response = MHD_create_response_from_buffer_static (strlen (page), page);
    ret = MHD_queue_response (connection, MHD_HTTP_OK, response);
  }
  if (NULL != auth_info)
    MHD_free (auth_info);
  MHD_destroy_response (response);
  return ret;
}


int
main (void)
{
  struct MHD_Daemon *daemon;

  daemon = MHD_start_daemon (MHD_USE_INTERNAL_POLLING_THREAD, PORT, NULL, NULL,
                             &answer_to_connection, NULL, MHD_OPTION_END);
  if (NULL == daemon)
    return 1;

  (void) getchar ();

  MHD_stop_daemon (daemon);
  return 0;
}
```

C.5 simplepost.c

```
/* Feel free to use this example code in any way
   you see fit (Public Domain) */

#include <sys/types.h>
#ifndef _WIN32
#include <sys/select.h>
#include <sys/socket.h>
#else
#include <winsock2.h>
#endif
#include <microhttpd.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
```

```
#if defined(_MSC_VER) && _MSC_VER + 0 <= 1800
/* Substitution is OK while return value is not used */
#define snprintf _snprintf
#endif

#define PORT            8888
#define POSTBUFFERSIZE  512
#define MAXNAMESIZE     20
#define MAXANSWERSIZE   512

#define GET             0
#define POST            1

struct connection_info_struct
{
  int connectiontype;
  char *answerstring;
  struct MHD_PostProcessor *postprocessor;
};

static const char *askpage =
  "<html><body>\n"
  "What's your name, Sir?<br>\n"
  "<form action=\"/namepost\" method=\"post\">\n"
  "<input name=\"name\" type=\"text\">\n"
  "<input type=\"submit\" value=\" Send \"></form>\n"
  "</body></html>";

#define GREETINGPAGE \
  "<html><body><h1>Welcome, %s!</center></h1></body></html>"

static const char *errorpage =
  "<html><body>This doesn't seem to be right.</body></html>";


static enum MHD_Result
send_page (struct MHD_Connection *connection, const char *page)
{
  enum MHD_Result ret;
  struct MHD_Response *response;


  response = MHD_create_response_from_buffer_static (strlen (page), page);
  if (! response)
    return MHD_NO;

  ret = MHD_queue_response (connection, MHD_HTTP_OK, response);
  MHD_destroy_response (response);

  return ret;
}


static enum MHD_Result
iterate_post (void *coninfo_cls, enum MHD_ValueKind kind, const char *key,
              const char *filename, const char *content_type,
              const char *transfer_encoding, const char *data, uint64_t off,
              size_t size)
{
  struct connection_info_struct *con_info = coninfo_cls;
  (void) kind;               /* Unused. Silent compiler warning. */
  (void) filename;           /* Unused. Silent compiler warning. */
  (void) content_type;       /* Unused. Silent compiler warning. */
  (void) transfer_encoding;  /* Unused. Silent compiler warning. */
  (void) off;                /* Unused. Silent compiler warning. */

  if (0 == strcmp (key, "name"))
  {
    if ((size > 0) && (size <= MAXNAMESIZE))
    {
      char *answerstring;
      answerstring = malloc (MAXANSWERSIZE);
      if (! answerstring)
        return MHD_NO;

      snprintf (answerstring, MAXANSWERSIZE, GREETINGPAGE, data);
      con_info->answerstring = answerstring;
    }
    else
      con_info->answerstring = NULL;

    return MHD_NO;
  }

  return MHD_YES;
}


static void
request_completed (void *cls, struct MHD_Connection *connection,
                   void **req_cls, enum MHD_RequestTerminationCode toe)
{
  struct connection_info_struct *con_info = *req_cls;
  (void) cls;         /* Unused. Silent compiler warning. */
  (void) connection;  /* Unused. Silent compiler warning. */
  (void) toe;         /* Unused. Silent compiler warning. */

  if (NULL == con_info)
    return;

  if (con_info->connectiontype == POST)
  {
    MHD_destroy_post_processor (con_info->postprocessor);
    if (con_info->answerstring)
      free (con_info->answerstring);
  }

  free (con_info);
  *req_cls = NULL;
}


static enum MHD_Result
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 **req_cls)
{
  (void) cls;               /* Unused. Silent compiler warning. */
  (void) url;               /* Unused. Silent compiler warning. */
  (void) version;           /* Unused. Silent compiler warning. */

  if (NULL == *req_cls)
  {
    struct connection_info_struct *con_info;

    con_info = malloc (sizeof (struct connection_info_struct));
    if (NULL == con_info)
      return MHD_NO;
    con_info->answerstring = NULL;

    if (0 == strcmp (method, "POST"))
    {
      con_info->postprocessor =
        MHD_create_post_processor (connection, POSTBUFFERSIZE,
                                   iterate_post, (void *) con_info);

      if (NULL == con_info->postprocessor)
      {
        free (con_info);
        return MHD_NO;
      }

      con_info->connectiontype = POST;
    }
    else
      con_info->connectiontype = GET;

    *req_cls = (void *) con_info;

    return MHD_YES;
  }

  if (0 == strcmp (method, "GET"))
  {
    return send_page (connection, askpage);
  }

  if (0 == strcmp (method, "POST"))
  {
    struct connection_info_struct *con_info = *req_cls;

    if (*upload_data_size != 0)
    {
      if (MHD_YES !=
          MHD_post_process (con_info->postprocessor,
                            upload_data,
                            *upload_data_size))
        return MHD_NO;
      *upload_data_size = 0;

      return MHD_YES;
    }
    else if (NULL != con_info->answerstring)
      return send_page (connection, con_info->answerstring);
  }

  return send_page (connection, errorpage);
}


int
main (void)
{
  struct MHD_Daemon *daemon;

  daemon = MHD_start_daemon (MHD_USE_AUTO | MHD_USE_INTERNAL_POLLING_THREAD,
                             PORT, NULL, NULL,
                             &answer_to_connection, NULL,
                             MHD_OPTION_NOTIFY_COMPLETED, request_completed,
                             NULL, MHD_OPTION_END);
  if (NULL == daemon)
    return 1;

  (void) getchar ();

  MHD_stop_daemon (daemon);

  return 0;
}
```

C.6 largepost.c

```
/* Feel free to use this example code in any way
   you see fit (Public Domain) */

#include <sys/types.h>
#ifndef _WIN32
#include <sys/select.h>
#include <sys/socket.h>
#else
#include <winsock2.h>
#endif
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <microhttpd.h>

#if defined(_MSC_VER) && _MSC_VER + 0 <= 1800
/* Substitution is OK while return value is not used */
#define snprintf _snprintf
#endif

#define PORT            8888
#define POSTBUFFERSIZE  512
#define MAXCLIENTS      2

enum ConnectionType
{
  GET = 0,
  POST = 1
};

static unsigned int nr_of_uploading_clients = 0;


/**
 * Information we keep per connection.
 */
struct connection_info_struct
{
  enum ConnectionType connectiontype;

  /**
   * Handle to the POST processing state.
   */
  struct MHD_PostProcessor *postprocessor;

  /**
   * File handle where we write uploaded data.
   */
  FILE *fp;

  /**
   * HTTP response body we will return, NULL if not yet known.
   */
  const char *answerstring;

  /**
   * HTTP status code we will return, 0 for undecided.
   */
  unsigned int answercode;
};


#define ASKPAGE \
  "<html><body>\n" \
  "Upload a file, please!<br>\n" \
  "There are %u clients uploading at the moment.<br>\n" \
  "<form action=\"/filepost\" method=\"post\" enctype=\"multipart/form-data\">\n" \
  "<input name=\"file\" type=\"file\">\n" \
  "<input type=\"submit\" value=\" Send \"></form>\n" \
  "</body></html>"
static const char *busypage =
  "<html><body>This server is busy, please try again later.</body></html>";
static const char *completepage =
  "<html><body>The upload has been completed.</body></html>";
static const char *errorpage =
  "<html><body>This doesn't seem to be right.</body></html>";
static const char *servererrorpage =
  "<html><body>Invalid request.</body></html>";
static const char *fileexistspage =
  "<html><body>This file already exists.</body></html>";
static const char *fileioerror =
  "<html><body>IO error writing to disk.</body></html>";
static const char *const postprocerror =
  "<html><head><title>Error</title></head><body>Error processing POST data</body></html>";
```

```
static enum MHD_Result
send_page (struct MHD_Connection *connection,
           const char *page,
           unsigned int status_code)
{
  enum MHD_Result ret;
  struct MHD_Response *response;

  response = MHD_create_response_from_buffer_static (strlen (page), page);
  if (! response)
    return MHD_NO;
  if (MHD_YES !=
      MHD_add_response_header (response,
                               MHD_HTTP_HEADER_CONTENT_TYPE,
                               "text/html"))
  {
    fprintf (stderr,
             "Failed to set content type header!\n");
  }
  ret = MHD_queue_response (connection,
                            status_code,
                            response);
  MHD_destroy_response (response);

  return ret;
}


static enum MHD_Result
iterate_post (void *coninfo_cls,
              enum MHD_ValueKind kind,
              const char *key,
              const char *filename,
              const char *content_type,
              const char *transfer_encoding,
              const char *data,
              uint64_t off,
              size_t size)
{
  struct connection_info_struct *con_info = coninfo_cls;
  FILE *fp;
  (void) kind;               /* Unused. Silent compiler warning. */
  (void) content_type;       /* Unused. Silent compiler warning. */
  (void) transfer_encoding;  /* Unused. Silent compiler warning. */
  (void) off;                /* Unused. Silent compiler warning. */

  if (0 != strcmp (key, "file"))
  {
    con_info->answerstring = servererrorpage;
    con_info->answercode = MHD_HTTP_BAD_REQUEST;
    return MHD_YES;
  }

  if (! con_info->fp)
  {
    if (0 != con_info->answercode)   /* something went wrong */
      return MHD_YES;
    if (NULL != (fp = fopen (filename, "rb")))
    {
      fclose (fp);
      con_info->answerstring = fileexistspage;
      con_info->answercode = MHD_HTTP_FORBIDDEN;
      return MHD_YES;
    }
    /* NOTE: This is technically a race with the 'fopen()' above,
       but there is no easy fix, short of moving to open(O_EXCL)
       instead of using fopen(). For the example, we do not care. */
    con_info->fp = fopen (filename, "ab");
    if (! con_info->fp)
    {
      con_info->answerstring = fileioerror;
      con_info->answercode = MHD_HTTP_INTERNAL_SERVER_ERROR;
      return MHD_YES;
    }
  }

  if (size > 0)
  {
    if (! fwrite (data, sizeof (char), size, con_info->fp))
    {
      con_info->answerstring = fileioerror;
      con_info->answercode = MHD_HTTP_INTERNAL_SERVER_ERROR;
      return MHD_YES;
    }
  }

  return MHD_YES;
}


static void
request_completed (void *cls,
                   struct MHD_Connection *connection,
                   void **req_cls,
                   enum MHD_RequestTerminationCode toe)
{
  struct connection_info_struct *con_info = *req_cls;
  (void) cls;         /* Unused. Silent compiler warning. */
  (void) connection;  /* Unused. Silent compiler warning. */
  (void) toe;         /* Unused. Silent compiler warning. */

  if (NULL == con_info)
    return;

  if (con_info->connectiontype == POST)
  {
    if (NULL != con_info->postprocessor)
    {
      MHD_destroy_post_processor (con_info->postprocessor);
      nr_of_uploading_clients--;
    }

    if (con_info->fp)
      fclose (con_info->fp);
  }
  free (con_info);
  *req_cls = NULL;
}


static enum MHD_Result
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 **req_cls)
{
  (void) cls;               /* Unused. Silent compiler warning. */
  (void) url;               /* Unused. Silent compiler warning. */
  (void) version;           /* Unused. Silent compiler warning. */

  if (NULL == *req_cls)
  {
    /* First call, setup data structures */
    struct connection_info_struct *con_info;

    if (nr_of_uploading_clients >= MAXCLIENTS)
      return send_page (connection,
                        busypage,
                        MHD_HTTP_SERVICE_UNAVAILABLE);

    con_info = malloc (sizeof (struct connection_info_struct));
    if (NULL == con_info)
      return MHD_NO;
    con_info->answercode = 0;   /* none yet */
    con_info->fp = NULL;

    if (0 == strcmp (method, MHD_HTTP_METHOD_POST))
    {
      con_info->postprocessor =
        MHD_create_post_processor (connection,
                                   POSTBUFFERSIZE,
                                   &iterate_post,
                                   (void *) con_info);

      if (NULL == con_info->postprocessor)
      {
        free (con_info);
        return MHD_NO;
      }

      nr_of_uploading_clients++;

      con_info->connectiontype = POST;
    }
    else
    {
      con_info->connectiontype = GET;
    }
    *req_cls = (void *) con_info;

    return MHD_YES;
  }

  if (0 == strcmp (method, MHD_HTTP_METHOD_GET))
  {
    /* We just return the standard form for uploads on all GET requests */
    char buffer[1024];

    snprintf (buffer,
              sizeof (buffer),
              ASKPAGE,
              nr_of_uploading_clients);
    return send_page (connection,
                      buffer,
                      MHD_HTTP_OK);
  }

  if (0 == strcmp (method, MHD_HTTP_METHOD_POST))
  {
    struct connection_info_struct *con_info = *req_cls;

    if (0 != *upload_data_size)
    {
      /* Upload not yet done */
      if (0 != con_info->answercode)
      {
        /* we already know the answer, skip rest of upload */
        *upload_data_size = 0;
        return MHD_YES;
      }
      if (MHD_YES !=
          MHD_post_process (con_info->postprocessor,
                            upload_data,
                            *upload_data_size))
      {
        con_info->answerstring = postprocerror;
        con_info->answercode = MHD_HTTP_INTERNAL_SERVER_ERROR;
      }
      *upload_data_size = 0;

      return MHD_YES;
    }
    /* Upload finished */
    if (NULL != con_info->fp)
    {
      fclose (con_info->fp);
      con_info->fp = NULL;
    }
    if (0 == con_info->answercode)
    {
      /* No errors encountered, declare success */
      con_info->answerstring = completepage;
      con_info->answercode = MHD_HTTP_OK;
    }
    return send_page (connection,
                      con_info->answerstring,
                      con_info->answercode);
  }

  /* Note a GET or a POST, generate error */
  return send_page (connection,
                    errorpage,
                    MHD_HTTP_BAD_REQUEST);
}


int
main (void)
{
  struct MHD_Daemon *daemon;

  daemon = MHD_start_daemon (MHD_USE_INTERNAL_POLLING_THREAD,
                             PORT, NULL, NULL,
                             &answer_to_connection, NULL,
                             MHD_OPTION_NOTIFY_COMPLETED, &request_completed,
                             NULL,
                             MHD_OPTION_END);
  if (NULL == daemon)
  {
    fprintf (stderr,
             "Failed to start daemon.\n");
    return 1;
  }
  (void) getchar ();
  MHD_stop_daemon (daemon);
  return 0;
}
```

C.7 sessions.c

```
/* Feel free to use this example code in any way
   you see fit (Public Domain) */

#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <errno.h>
#include <time.h>
#include <microhttpd.h>

/**
 * Invalid method page.
 */
#define METHOD_ERROR \
  "<html><head><title>Illegal request</title></head><body>Go away.</body></html>"

/**
 * Invalid URL page.
 */
#define NOT_FOUND_ERROR \
  "<html><head><title>Not found</title></head><body>Go away.</body></html>"

/**
 * Front page. (/)
 */
#define MAIN_PAGE \
  "<html><head><title>Welcome</title></head><body><form action=\"/2\" method=\"post\">What is your name?

#define FORM_V1 MAIN_PAGE

/**
 * Second page. (/2)
 */
#define SECOND_PAGE \
  "<html><head><title>Tell me more</title></head><body><a href=\"/\">previous</a> <form action=\"/S\" met

#define FORM_V1_V2 SECOND_PAGE

/**
 * Second page (/S)
 */
#define SUBMIT_PAGE \
  "<html><head><title>Ready to submit?</title></head><body><form action=\"/F\" method=\"post\"><a href=\"

/**
 * Last page.
 */
#define LAST_PAGE \
  "<html><head><title>Thank you</title></head><body>Thank you.</body></html>"

/**
 * Name of our cookie.
 */
#define COOKIE_NAME "session"


/**
 * State we keep for each user/session/browser.
 */
struct Session
{
  /**
   * We keep all sessions in a linked list.
   */
  struct Session *next;

  /**
   * Unique ID for this session.
   */
  char sid[33];

  /**
   * Reference counter giving the number of connections
   * currently using this session.
   */
  unsigned int rc;

  /**
   * Time when this session was last active.
   */
  time_t start;
  /**
   * String submitted via form.
   */
  char value_1[64];

  /**
   * Another value submitted via form.
   */
  char value_2[64];

};


/**
 * Data kept per request.
 */
struct Request
{

  /**
   * Associated session.
   */
  struct Session *session;

  /**
   * Post processor handling form data (IF this is
   * a POST request).
   */
  struct MHD_PostProcessor *pp;

  /**
   * URL to serve in response to this POST (if this request
   * was a 'POST')
   */
  const char *post_url;

};


/**
 * Linked list of all active sessions.  Yes, O(n) but a
 * hash table would be overkill for a simple example...
 */
static struct Session *sessions;


/**
 * Return the session handle for this connection, or
 * create one if this is a new user.
 */
static struct Session *
get_session (struct MHD_Connection *connection)
{
  struct Session *ret;
  const char *cookie;

  cookie = MHD_lookup_connection_value (connection,
                                        MHD_COOKIE_KIND,
                                       COOKIE_NAME);
  if (cookie != NULL)
  {
    /* find existing session */
    ret = sessions;
    while (NULL != ret)
    {
      if (0 == strcmp (cookie, ret->sid))
        break;
      ret = ret->next;
    }
    if (NULL != ret)
    {
      ret->rc++;
      return ret;
    }
  }
  /* create fresh session */
  ret = calloc (1, sizeof (struct Session));
  if (NULL == ret)
  {
    fprintf (stderr, "calloc error: %s\n", strerror (errno));
    return NULL;
  }
  /* not a super-secure way to generate a random session ID,
     but should do for a simple example... */
  snprintf (ret->sid,
            sizeof (ret->sid),
            "%X%X%X%X",
            (unsigned int) rand (),
            (unsigned int) rand (),
            (unsigned int) rand (),
            (unsigned int) rand ());
  ret->rc++;
  ret->start = time (NULL);
  ret->next = sessions;
  sessions = ret;
  return ret;
}


/**
 * Type of handler that generates a reply.
 *
 * @param cls content for the page (handler-specific)
 * @param mime mime type to use
 * @param session session information
 * @param connection connection to process
 * @param #MHD_YES on success, #MHD_NO on failure
 */
typedef enum MHD_Result (*PageHandler)(const void *cls,
                                       const char *mime,
                                       struct Session *session,
                                       struct MHD_Connection *connection);


/**
 * Entry we generate for each page served.
 */
struct Page
{
  /**
   * Acceptable URL for this page.
   */
  const char *url;

  /**
   * Mime type to set for the page.
   */
  const char *mime;

  /**
   * Handler to call to generate response.
   */
  PageHandler handler;

  /**
   * Extra argument to handler.
   */
  const void *handler_cls;
};


/**
 * Add header to response to set a session cookie.
 *
 * @param session session to use
 * @param response response to modify
 */
static void
add_session_cookie (struct Session *session,
                    struct MHD_Response *response)
{
  char cstr[256];
  snprintf (cstr,
            sizeof (cstr),
            "%s=%s",
            COOKIE_NAME,
            session->sid);
  if (MHD_NO ==
      MHD_add_response_header (response,
                               MHD_HTTP_HEADER_SET_COOKIE,
                               cstr))
  {
    fprintf (stderr,
             "Failed to set session cookie header!\n");
  }
}


/**
 * Handler that returns a simple static HTTP page that
 * is passed in via 'cls'.
 *
 * @param cls a 'const char *' with the HTML webpage to return
 * @param mime mime type to use
 * @param session session handle
 * @param connection connection to use
 */
static enum MHD_Result
serve_simple_form (const void *cls,
                   const char *mime,
                   struct Session *session,
                   struct MHD_Connection *connection)
{
  enum MHD_Result ret;
  const char *form = cls;
  struct MHD_Response *response;

  /* return static form */
  response = MHD_create_response_from_buffer_static (strlen (form), form);
  add_session_cookie (session, response);
  if (MHD_YES !=
      MHD_add_response_header (response,
                               MHD_HTTP_HEADER_CONTENT_TYPE,
                               mime))
  {
    fprintf (stderr,
             "Failed to set content type header!\n");
    /* return response without content type anyway ... */
  }
  ret = MHD_queue_response (connection,
                            MHD_HTTP_OK,
                            response);
  MHD_destroy_response (response);
  return ret;
}


/**
 * Handler that adds the 'v1' value to the given HTML code.
 *
 * @param cls a 'const char *' with the HTML webpage to return
 * @param mime mime type to use
 * @param session session handle
 * @param connection connection to use
 */
static enum MHD_Result
fill_v1_form (const void *cls,
              const char *mime,
              struct Session *session,
              struct MHD_Connection *connection)
{
  enum MHD_Result ret;
  char *reply;
  struct MHD_Response *response;
  int reply_len;
  (void) cls; /* Unused */

  /* Emulate 'asprintf' */
  reply_len = snprintf (NULL, 0, FORM_V1, session->value_1);
  if (0 > reply_len)
    return MHD_NO; /* Internal error */
  reply = (char *) malloc ((size_t) ((size_t) reply_len + 1));
  if (NULL == reply)
    return MHD_NO; /* Out-of-memory error */

  if (reply_len != snprintf (reply,
                             (size_t) (((size_t) reply_len) + 1),
                             FORM_V1,
                             session->value_1))
  {
    free (reply);
    return MHD_NO; /* printf error */
  }

  /* return static form */
  response =
    MHD_create_response_from_buffer_with_free_callback ((size_t) reply_len,
                                                        (void *) reply,
                                                        &free);
  if (NULL != response)
  {
    add_session_cookie (session, response);
    if (MHD_YES !=
        MHD_add_response_header (response,
                                 MHD_HTTP_HEADER_CONTENT_TYPE,
                                 mime))
    {
      fprintf (stderr,
               "Failed to set content type header!\n");
      /* return response without content type anyway ... */
    }
    ret = MHD_queue_response (connection,
                              MHD_HTTP_OK,
                              response);
    MHD_destroy_response (response);
  }
  else
  {
    free (reply);
    ret = MHD_NO;
  }
  return ret;
}


/**
 * Handler that adds the 'v1' and 'v2' values to the given HTML code.
 *
 * @param cls a 'const char *' with the HTML webpage to return
 * @param mime mime type to use
 * @param session session handle
 * @param connection connection to use
 */
static enum MHD_Result
fill_v1_v2_form (const void *cls,
                 const char *mime,
                 struct Session *session,
                 struct MHD_Connection *connection)
{
  enum MHD_Result ret;
  char *reply;
  struct MHD_Response *response;
  int reply_len;
  (void) cls; /* Unused */

  /* Emulate 'asprintf' */
  reply_len = snprintf (NULL, 0, FORM_V1_V2, session->value_1,
                        session->value_2);
  if (0 > reply_len)
    return MHD_NO; /* Internal error */

  reply = (char *) malloc ((size_t) ((size_t) reply_len + 1));
  if (NULL == reply)
    return MHD_NO; /* Out-of-memory error */

  if (reply_len != snprintf (reply,
                             (size_t) ((size_t) reply_len + 1),
                             FORM_V1_V2,
                             session->value_1,
                             session->value_2))
  {
    free (reply);
    return MHD_NO; /* printf error */
  }

  /* return static form */
  response =
    MHD_create_response_from_buffer_with_free_callback ((size_t) reply_len,
                                                        (void *) reply,
                                                        &free);
  if (NULL != response)
  {
    add_session_cookie (session, response);
    if (MHD_YES !=
        MHD_add_response_header (response,
                                 MHD_HTTP_HEADER_CONTENT_TYPE,
                                 mime))
    {
      fprintf (stderr,
               "Failed to set content type header!\n");
      /* return response without content type anyway ... */
    }
    ret = MHD_queue_response (connection,
                              MHD_HTTP_OK,
                              response);
    MHD_destroy_response (response);
  }
  else
  {
    free (reply);
    ret = MHD_NO;
  }
  return ret;
}


/**
```

```
 * Handler used to generate a 404 reply.
 *
 * @param cls a 'const char *' with the HTML webpage to return
 * @param mime mime type to use
 * @param session session handle
 * @param connection connection to use
 */
static enum MHD_Result
not_found_page (const void *cls,
                const char *mime,
                struct Session *session,
                struct MHD_Connection *connection)
{
  enum MHD_Result ret;
  struct MHD_Response *response;
  (void) cls;     /* Unused. Silent compiler warning. */
  (void) session; /* Unused. Silent compiler warning. */

  /* unsupported HTTP method */
  response = MHD_create_response_from_buffer_static (strlen (NOT_FOUND_ERROR),
                                                     NOT_FOUND_ERROR);
  ret = MHD_queue_response (connection,
                            MHD_HTTP_NOT_FOUND,
                            response);
  if (MHD_YES !=
      MHD_add_response_header (response,
                               MHD_HTTP_HEADER_CONTENT_TYPE,
                               mime))
  {
    fprintf (stderr,
             "Failed to set content type header!\n");
    /* return response without content type anyway ... */
  }
  MHD_destroy_response (response);
  return ret;
}


/**
 * List of all pages served by this HTTP server.
 */
static const struct Page pages[] = {
  { "/", "text/html",  &fill_v1_form, NULL },
  { "/2", "text/html", &fill_v1_v2_form, NULL },
  { "/S", "text/html", &serve_simple_form, SUBMIT_PAGE },
  { "/F", "text/html", &serve_simple_form, LAST_PAGE },
  { NULL, NULL, &not_found_page, NULL }   /* 404 */
};


/**
 * Iterator over key-value pairs where the value
 * maybe made available in increments and/or may
 * not be zero-terminated.  Used for processing
 * POST data.
 *
 * @param cls user-specified closure
 * @param kind type of the value
 * @param key 0-terminated key for the value
 * @param filename name of the uploaded file, NULL if not known
 * @param content_type mime-type of the data, NULL if not known
 * @param transfer_encoding encoding of the data, NULL if not known
 * @param data pointer to size bytes of data at the
 *             specified offset
 * @param off offset of data in the overall value
 * @param size number of bytes in data available
 * @return #MHD_YES to continue iterating,
 *         #MHD_NO to abort the iteration
 */
static enum MHD_Result
post_iterator (void *cls,
               enum MHD_ValueKind kind,
               const char *key,
               const char *filename,
               const char *content_type,
               const char *transfer_encoding,
               const char *data, uint64_t off, size_t size)
{
  struct Request *request = cls;
  struct Session *session = request->session;
  (void) kind;               /* Unused. Silent compiler warning. */
  (void) filename;           /* Unused. Silent compiler warning. */
  (void) content_type;       /* Unused. Silent compiler warning. */
  (void) transfer_encoding;  /* Unused. Silent compiler warning. */

  if (0 == strcmp ("DONE", key))
  {
    fprintf (stdout,
             "Session `%s' submitted `%s', `%s'\n",
             session->sid,
             session->value_1,
             session->value_2);
    return MHD_YES;
  }
  if (0 == strcmp ("v1", key))
  {
    if (off >= sizeof(session->value_1) - 1)
      return MHD_YES; /* Discard extra data */
    if (size + off >= sizeof(session->value_1))
      size = (size_t) (sizeof (session->value_1) - off - 1); /* crop extra data */
    memcpy (&session->value_1[off],
            data,
            size);
    if (size + off < sizeof (session->value_1))
      session->value_1[size + off] = '\0';
    return MHD_YES;
  }
  if (0 == strcmp ("v2", key))
  {
    if (off >= sizeof(session->value_2) - 1)
      return MHD_YES; /* Discard extra data */
    if (size + off >= sizeof(session->value_2))
      size = (size_t) (sizeof (session->value_2) - off - 1); /* crop extra data */
    memcpy (&session->value_2[off],
            data,
            size);
    if (size + off < sizeof (session->value_2))
      session->value_2[size + off] = '\0';
    return MHD_YES;
  }
  fprintf (stderr, "Unsupported form value `%s'\n", key);
  return MHD_YES;
}


/**
 * Main MHD callback for handling requests.
 *
 *
 * @param cls argument given together with the function
 *        pointer when the handler was registered with MHD
 * @param connection handle to connection which is being processed
 * @param url the requested url
 * @param method the HTTP method used ("GET", "PUT", etc.)
 * @param version the HTTP version string (i.e. "HTTP/1.1")
 * @param upload_data the data being uploaded (excluding HEADERS,
 *        for a POST that fits into memory and that is encoded
 *        with a supported encoding, the POST data will NOT be
 *        given in upload_data and is instead available as
 *        part of MHD_get_connection_values; very large POST
 *        data *will* be made available incrementally in
 *        upload_data)
 * @param upload_data_size set initially to the size of the
 *        upload_data provided; the method must update this
 *        value to the number of bytes NOT processed;
 * @param req_cls pointer that the callback can set to some
 *        address and that will be preserved by MHD for future
 *        calls for this request; since the access handler may
 *        be called many times (i.e., for a PUT/POST operation
 *        with plenty of upload data) this allows the application
 *        to easily associate some request-specific state.
 *        If necessary, this state can be cleaned up in the
 *        global "MHD_RequestCompleted" callback (which
 *        can be set with the MHD_OPTION_NOTIFY_COMPLETED).
 *        Initially, <tt>*req_cls</tt> will be NULL.
 * @return MHS_YES if the connection was handled successfully,
 *         MHS_NO if the socket must be closed due to a serious
 *         error while handling the request
 */
static enum MHD_Result
create_response (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 **req_cls)
{
  struct MHD_Response *response;
  struct Request *request;
  struct Session *session;
  enum MHD_Result ret;
  unsigned int i;
  (void) cls;               /* Unused. Silent compiler warning. */
  (void) version;           /* Unused. Silent compiler warning. */

  request = *req_cls;
  if (NULL == request)
  {
    request = calloc (1, sizeof (struct Request));
    if (NULL == request)
    {
      fprintf (stderr, "calloc error: %s\n", strerror (errno));
      return MHD_NO;
    }
    *req_cls = request;
    if (0 == strcmp (method, MHD_HTTP_METHOD_POST))
    {
      request->pp = MHD_create_post_processor (connection, 1024,
                                               &post_iterator, request);
      if (NULL == request->pp)
      {
        fprintf (stderr, "Failed to setup post processor for `%s'\n",
                 url);
        return MHD_NO; /* internal error */
      }
    }
    return MHD_YES;
  }
  if (NULL == request->session)
  {
    request->session = get_session (connection);
    if (NULL == request->session)
    {
      fprintf (stderr, "Failed to setup session for `%s'\n",
               url);
      return MHD_NO; /* internal error */
    }
  }
  session = request->session;
  session->start = time (NULL);
  if (0 == strcmp (method, MHD_HTTP_METHOD_POST))
  {
    /* evaluate POST data */
    if (MHD_YES !=
        MHD_post_process (request->pp,
                          upload_data,
                          *upload_data_size))
      return MHD_NO; /* internal error */
    if (0 != *upload_data_size)
    {
      *upload_data_size = 0;
      return MHD_YES;
    }
    /* done with POST data, serve response */
    MHD_destroy_post_processor (request->pp);
    request->pp = NULL;
    method = MHD_HTTP_METHOD_GET;   /* fake 'GET' */
    if (NULL != request->post_url)
      url = request->post_url;
  }
  if ( (0 == strcmp (method, MHD_HTTP_METHOD_GET)) ||
       (0 == strcmp (method, MHD_HTTP_METHOD_HEAD)) )
  {
    /* find out which page to serve */
    i = 0;
    while ( (pages[i].url != NULL) &&
            (0 != strcmp (pages[i].url, url)) )
      i++;
    ret = pages[i].handler (pages[i].handler_cls,
                            pages[i].mime,
                            session, connection);
    if (ret != MHD_YES)
      fprintf (stderr, "Failed to create page for `%s'\n",
               url);
    return ret;
  }
  /* unsupported HTTP method */
  response = MHD_create_response_from_buffer_static (strlen (METHOD_ERROR),
                                                     METHOD_ERROR);
  ret = MHD_queue_response (connection,
                            MHD_HTTP_NOT_ACCEPTABLE,
                            response);
  MHD_destroy_response (response);
  return ret;
}


/**
 * Callback called upon completion of a request.
 * Decrements session reference counter.
 *
 * @param cls not used
 * @param connection connection that completed
 * @param req_cls session handle
 * @param toe status code
 */
static void
request_completed_callback (void *cls,
                            struct MHD_Connection *connection,
                            void **req_cls,
                            enum MHD_RequestTerminationCode toe)
{
  struct Request *request = *req_cls;
  (void) cls;         /* Unused. Silent compiler warning. */
  (void) connection;  /* Unused. Silent compiler warning. */
  (void) toe;         /* Unused. Silent compiler warning. */

  if (NULL == request)
    return;
  if (NULL != request->session)
    request->session->rc--;
  if (NULL != request->pp)
    MHD_destroy_post_processor (request->pp);
  free (request);
}
```

```
/**
 * Clean up handles of sessions that have been idle for
 * too long.
 */
static void
expire_sessions (void)
{
  struct Session *pos;
  struct Session *prev;
  struct Session *next;
  time_t now;

  now = time (NULL);
  prev = NULL;
  pos = sessions;
  while (NULL != pos)
  {
    next = pos->next;
    if (now - pos->start > 60 * 60)
    {
      /* expire sessions after 1h */
      if (NULL == prev)
        sessions = pos->next;
      else
        prev->next = next;
      free (pos);
    }
    else
      prev = pos;
    pos = next;
  }
}


/**
 * Call with the port number as the only argument.
 * Never terminates (other than by signals, such as CTRL-C).
 */
int
main (int argc, char *const *argv)
{
  struct MHD_Daemon *d;
  struct timeval tv;
  struct timeval *tvp;
  fd_set rs;
  fd_set ws;
  fd_set es;
  MHD_socket max;
  uint64_t mhd_timeout;
  unsigned int port;

  if (argc != 2)
  {
    printf ("%s PORT\n", argv[0]);
    return 1;
  }
  if ( (1 != sscanf (argv[1], "%u", &port)) ||
       (0 == port) || (65535 < port) )
  {
    fprintf (stderr,
             "Port must be a number between 1 and 65535.\n");
    return 1;
  }

  /* initialize PRNG */
  srand ((unsigned int) time (NULL));
  d = MHD_start_daemon (MHD_USE_ERROR_LOG,
                        (uint16_t) port,
                        NULL, NULL,
                        &create_response, NULL,
                        MHD_OPTION_CONNECTION_TIMEOUT, (unsigned int) 15,
                        MHD_OPTION_NOTIFY_COMPLETED,
                        &request_completed_callback, NULL,
                        MHD_OPTION_APP_FD_SETSIZE, (int) FD_SETSIZE,
                        MHD_OPTION_END);
  if (NULL == d)
    return 1;
  while (1)
  {
    expire_sessions ();
    max = 0;
    FD_ZERO (&rs);
    FD_ZERO (&ws);
    FD_ZERO (&es);
    if (MHD_YES != MHD_get_fdset (d, &rs, &ws, &es, &max))
      break; /* fatal internal error */
    if (MHD_get_timeout64 (d, &mhd_timeout) == MHD_YES)
    {
#if ! defined(_WIN32) || defined(__CYGWIN__)
      tv.tv_sec = (time_t) (mhd_timeout / 1000);
#else  /* Native W32 */
      tv.tv_sec = (long) (mhd_timeout / 1000);
#endif /* Native W32 */
      tv.tv_usec = ((long) (mhd_timeout % 1000)) * 1000;
      tvp = &tv;
    }
    else
      tvp = NULL;
    if (-1 == select ((int) max + 1, &rs, &ws, &es, tvp))
    {
      if (EINTR != errno)
        fprintf (stderr,
                 "Aborting due to error during select: %s\n",
                 strerror (errno));
      break;
    }
    MHD_run (d);
  }
  MHD_stop_daemon (d);
  return 0;
}
```

C.8 tlsauthentication.c

```
/* Feel free to use this example code in any way
   you see fit (Public Domain) */
```

```
#include <sys/types.h>
#ifndef _WIN32
#include <sys/select.h>
#include <sys/socket.h>
#else
#include <winsock2.h>
#endif
#include <microhttpd.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

#define PORT 8888

#define REALM     "Maintenance"
#define USER      "a legitimate user"
#define PASSWORD  "and his password"

#define SERVERKEYFILE "server.key"
#define SERVERCERTFILE "server.pem"


static size_t
get_file_size (const char *filename)
{
  FILE *fp;

  fp = fopen (filename, "rb");
  if (fp)
  {
    long size;

    if ((0 != fseek (fp, 0, SEEK_END)) || (-1 == (size = ftell (fp))))
      size = 0;

    fclose (fp);

    return (size_t) size;
  }
  else
    return 0;
}


static char *
load_file (const char *filename)
{
  FILE *fp;
  char *buffer;
  size_t size;

  size = get_file_size (filename);
  if (0 == size)
    return NULL;

  fp = fopen (filename, "rb");
  if (! fp)
```

```
    return NULL;

  buffer = malloc (size + 1);
  if (! buffer)
  {
    fclose (fp);
    return NULL;
  }
  buffer[size] = '\0';

  if (size != fread (buffer, 1, size, fp))
  {
    free (buffer);
    buffer = NULL;
  }

  fclose (fp);
  return buffer;
}


static enum MHD_Result
ask_for_authentication (struct MHD_Connection *connection, const char *realm)
{
  enum MHD_Result ret;
  struct MHD_Response *response;

  response = MHD_create_response_empty (MHD_RF_NONE);
  if (! response)
    return MHD_NO;

  ret = MHD_queue_basic_auth_required_response3 (connection,
                                                 realm,
                                                 MHD_YES,
                                                 response);
  MHD_destroy_response (response);
  return ret;
}


static int
is_authenticated (struct MHD_Connection *connection,
                  const char *username,
                  const char *password)
{
  struct MHD_BasicAuthInfo *auth_info;
  int authenticated;

  auth_info = MHD_basic_auth_get_username_password3 (connection);
  if (NULL == auth_info)
    return 0;
  authenticated =
    ( (strlen (username) == auth_info->username_len) &&
      (0 == memcmp (auth_info->username, username, auth_info->username_len)) &&
      /* The next check against NULL is optional,
       * if 'password' is NULL then 'password_len' is always zero. */
      (NULL != auth_info->password) &&
      (strlen (password) == auth_info->password_len) &&
      (0 == memcmp (auth_info->password, password, auth_info->password_len)) );

  MHD_free (auth_info);

  return authenticated;
}


static enum MHD_Result
secret_page (struct MHD_Connection *connection)
{
  enum MHD_Result ret;
  struct MHD_Response *response;
  const char *page = "<html><body>A secret.</body></html>";

  response = MHD_create_response_from_buffer_static (strlen (page), page);
  if (! response)
    return MHD_NO;

  ret = MHD_queue_response (connection, MHD_HTTP_OK, response);
  MHD_destroy_response (response);

  return ret;
}


static enum MHD_Result
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 **req_cls)
{
  (void) cls;               /* Unused. Silent compiler warning. */
  (void) url;               /* Unused. Silent compiler warning. */
  (void) version;           /* Unused. Silent compiler warning. */
  (void) upload_data;       /* Unused. Silent compiler warning. */
  (void) upload_data_size;  /* Unused. Silent compiler warning. */

  if (0 != strcmp (method, "GET"))
    return MHD_NO;
  if (NULL == *req_cls)
  {
    *req_cls = connection;
    return MHD_YES;
  }

  if (! is_authenticated (connection, USER, PASSWORD))
    return ask_for_authentication (connection, REALM);

  return secret_page (connection);
}


int
main (void)
{
  struct MHD_Daemon *daemon;
  char *key_pem;
  char *cert_pem;

  key_pem = load_file (SERVERKEYFILE);
  cert_pem = load_file (SERVERCERTFILE);

  if ((key_pem == NULL) || (cert_pem == NULL))
  {
    printf ("The key/certificate files could not be read.\n");
    if (NULL != key_pem)
      free (key_pem);
    if (NULL != cert_pem)
      free (cert_pem);
    return 1;
  }

  daemon =
    MHD_start_daemon (MHD_USE_INTERNAL_POLLING_THREAD | MHD_USE_TLS, PORT, NULL,
                      NULL, &answer_to_connection, NULL,
                      MHD_OPTION_HTTPS_MEM_KEY, key_pem,
                      MHD_OPTION_HTTPS_MEM_CERT, cert_pem, MHD_OPTION_END);
  if (NULL == daemon)
  {
    printf ("%s\n", cert_pem);

    free (key_pem);
    free (cert_pem);

    return 1;
  }

  (void) getchar ();

  MHD_stop_daemon (daemon);
  free (key_pem);
  free (cert_pem);

  return 0;
}
```

C.9 websocket.c

```
/* Feel free to use this example code in any way
   you see fit (Public Domain) */

#include <sys/types.h>
#ifndef _WIN32
#include <sys/select.h>
#include <sys/socket.h>
#include <fcntl.h>
#else
#include <winsock2.h>
#endif
#include <microhttpd.h>
#include <microhttpd_ws.h>
#include <time.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
```

```
#include <errno.h>

#define PORT 80

#define PAGE \
  "<!DOCTYPE html>\n" \
  "<html>\n" \
  "<head>\n" \
  "<meta charset=\"UTF-8\">\n" \
  "<title>Websocket Demo</title>\n" \
  "<script>\n" \
  "\n" \
  "let url = 'ws' + (window.location.protocol === 'https:' ? 's' : '')" \
    "  + ':/" "/' +\n" \
  "          window.location.host + '/chat';\n" \
  "let socket = null;\n" \
  "\n" \
  "window.onload = function(event) {\n" \
  "  socket = new WebSocket(url);\n" \
  "  socket.onopen = function(event) {\n" \
  "    document.write('The websocket connection has been " \
    "established.<br>');\n" \
  "\n" \
  "    /" "/ Send some text\n" \
  "    socket.send('Hello from JavaScript!');\n" \
  "  }\n" \
  "\n" \
  "  socket.onclose = function(event) {\n" \
  "    document.write('The websocket connection has been closed.<br>');\n" \
  "  }\n" \
  "\n" \
  "  socket.onerror = function(event) {\n" \
  "    document.write('An error occurred during the websocket " \
    "communication.<br>');\n" \
  "  }\n" \
  "\n" \
  "  socket.onmessage = function(event) {\n" \
  "    document.write('Websocket message received: ' + " \
    "event.data + '<br>');\n" \
  "  }\n" \
  "}\n" \
  "\n" \
  "</script>\n" \
  "</head>\n" \
  "<body>\n" \
  "</body>\n" \
  "</html>"

#define PAGE_NOT_FOUND \
  "404 Not Found"

#define PAGE_INVALID_WEBSOCKET_REQUEST \
  "Invalid WebSocket request!"

static void
send_all (MHD_socket fd,
          const char *buf,
          size_t len);
```

```
static void
make_blocking (MHD_socket fd);

static void
upgrade_handler (void *cls,
                 struct MHD_Connection *connection,
                 void *req_cls,
                 const char *extra_in,
                 size_t extra_in_size,
                 MHD_socket fd,
                 struct MHD_UpgradeResponseHandle *urh)
{
  /* make the socket blocking (operating-system-dependent code) */
  make_blocking (fd);

  /* create a websocket stream for this connection */
  struct MHD_WebSocketStream *ws;
  int result = MHD_websocket_stream_init (&ws,
                                          0,
                                          0);
  if (0 != result)
  {
    /* Couldn't create the websocket stream.
     * So we close the socket and leave
     */
    MHD_upgrade_action (urh,
                        MHD_UPGRADE_ACTION_CLOSE);
    return;
  }

  /* Let's wait for incoming data */
  const size_t buf_len = 256;
  char buf[buf_len];
  ssize_t got;
  while (MHD_WEBSOCKET_VALIDITY_VALID == MHD_websocket_stream_is_valid (ws))
  {
    got = recv (fd,
                buf,
                buf_len,
                0);
    if (0 >= got)
    {
      /* the TCP/IP socket has been closed */
      break;
    }

    /* parse the entire received data */
    size_t buf_offset = 0;
    while (buf_offset < (size_t) got)
    {
      size_t new_offset = 0;
      char *frame_data = NULL;
      size_t frame_len  = 0;
      int status = MHD_websocket_decode (ws,
                                         buf + buf_offset,
                                         ((size_t) got) - buf_offset,
                                         &new_offset,
                                         &frame_data,
                                         &frame_len);
      if (0 > status)
      {
        /* an error occurred and the connection must be closed */
        if (NULL != frame_data)
        {
          MHD_websocket_free (ws, frame_data);
        }
        break;
      }
      else
      {
        buf_offset += new_offset;
        if (0 < status)
        {
          /* the frame is complete */
          switch (status)
          {
          case MHD_WEBSOCKET_STATUS_TEXT_FRAME:
            /* The client has sent some text.
             * We will display it and answer with a text frame.
             */
            if (NULL != frame_data)
            {
              printf ("Received message: %s\n", frame_data);
              MHD_websocket_free (ws, frame_data);
              frame_data = NULL;
            }
            result = MHD_websocket_encode_text (ws,
                                                "Hello",
                                                5,  /* length of "Hello" */
                                                0,
                                                &frame_data,
                                                &frame_len,
                                                NULL);
            if (0 == result)
            {
              send_all (fd,
                        frame_data,
                        frame_len);
            }
            break;

          case MHD_WEBSOCKET_STATUS_CLOSE_FRAME:
            /* if we receive a close frame, we will respond with one */
            MHD_websocket_free (ws,
                                frame_data);
            frame_data = NULL;

            result = MHD_websocket_encode_close (ws,
                                                 0,
                                                 NULL,
                                                 0,
                                                 &frame_data,
                                                 &frame_len);
            if (0 == result)
            {
              send_all (fd,
                        frame_data,
                        frame_len);
            }
            break;

          case MHD_WEBSOCKET_STATUS_PING_FRAME:
            /* if we receive a ping frame, we will respond */
            /* with the corresponding pong frame */
            {
              char *pong = NULL;
              size_t pong_len = 0;
              result = MHD_websocket_encode_pong (ws,
                                                  frame_data,
                                                  frame_len,
                                                  &pong,
                                                  &pong_len);
              if (0 == result)
              {
                send_all (fd,
                          pong,
                          pong_len);
              }
              MHD_websocket_free (ws,
                                  pong);
            }
            break;

          default:
            /* Other frame types are ignored
             * in this minimal example.
             * This is valid, because they become
             * automatically skipped if we receive them unexpectedly
             */
            break;
          }
        }
        if (NULL != frame_data)
        {
          MHD_websocket_free (ws, frame_data);
        }
      }
    }
  }

  /* free the websocket stream */
  MHD_websocket_stream_free (ws);

  /* close the socket when it is not needed anymore */
  MHD_upgrade_action (urh,
                      MHD_UPGRADE_ACTION_CLOSE);
}


/* This helper function is used for the case that
 * we need to resend some data
 */
static void
```

```
send_all (MHD_socket fd,
          const char *buf,
          size_t len)
{
  ssize_t ret;
  size_t off;

  for (off = 0; off < len; off += ret)
  {
    ret = send (fd,
                &buf[off],
                (int) (len - off),
                0);
    if (0 > ret)
    {
      if (EAGAIN == errno)
      {
        ret = 0;
        continue;
      }
      break;
    }
    if (0 == ret)
      break;
  }
}


/* This helper function contains operating-system-dependent code and
 * is used to make a socket blocking.
 */
static void
make_blocking (MHD_socket fd)
{
#ifndef _WIN32
  int flags;

  flags = fcntl (fd, F_GETFL);
  if (-1 == flags)
    abort ();
  if ((flags & ~O_NONBLOCK) != flags)
    if (-1 == fcntl (fd, F_SETFL, flags & ~O_NONBLOCK))
      abort ();
#else  /* _WIN32 */
  unsigned long flags = 0;

  if (0 != ioctlsocket (fd, (int) FIONBIO, &flags))
    abort ();
#endif /* _WIN32 */
}


static enum MHD_Result
access_handler (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 **req_cls)
{
  static int aptr;
  struct MHD_Response *response;
  int ret;

  (void) cls;               /* Unused. Silent compiler warning. */
  (void) upload_data;       /* Unused. Silent compiler warning. */
  (void) upload_data_size;  /* Unused. Silent compiler warning. */

  if (0 != strcmp (method, "GET"))
    return MHD_NO;              /* unexpected method */
  if (&aptr != *req_cls)
  {
    /* do never respond on first call */
    *req_cls = &aptr;
    return MHD_YES;
  }
  *req_cls = NULL;                  /* reset when done */

  if (0 == strcmp (url, "/"))
  {
    /* Default page for visiting the server */
    struct MHD_Response *response;
    response = MHD_create_response_from_buffer_static (strlen (PAGE),
                                                       PAGE);
    ret = MHD_queue_response (connection,
                              MHD_HTTP_OK,
                              response);
    MHD_destroy_response (response);
  }
  else if (0 == strcmp (url, "/chat"))
  {
    char is_valid = 1;
    const char *value = NULL;
    char sec_websocket_accept[29];

    if (0 != MHD_websocket_check_http_version (version))
    {
      is_valid = 0;
    }
    value = MHD_lookup_connection_value (connection,
                                         MHD_HEADER_KIND,
                                         MHD_HTTP_HEADER_CONNECTION);
    if (0 != MHD_websocket_check_connection_header (value))
    {
      is_valid = 0;
    }
    value = MHD_lookup_connection_value (connection,
                                         MHD_HEADER_KIND,
                                         MHD_HTTP_HEADER_UPGRADE);
    if (0 != MHD_websocket_check_upgrade_header (value))
    {
      is_valid = 0;
    }
    value = MHD_lookup_connection_value (connection,
                                         MHD_HEADER_KIND,
                                         MHD_HTTP_HEADER_SEC_WEBSOCKET_VERSION);
    if (0 != MHD_websocket_check_version_header (value))
    {
      is_valid = 0;
    }
    value = MHD_lookup_connection_value (connection,
                                         MHD_HEADER_KIND,
                                         MHD_HTTP_HEADER_SEC_WEBSOCKET_KEY);
    if (0 != MHD_websocket_create_accept_header (value, sec_websocket_accept))
    {
      is_valid = 0;
    }

    if (1 == is_valid)
    {
      /* upgrade the connection */
      response = MHD_create_response_for_upgrade (&upgrade_handler,
                                                  NULL);
      MHD_add_response_header (response,
                               MHD_HTTP_HEADER_UPGRADE,
                               "websocket");
      MHD_add_response_header (response,
                               MHD_HTTP_HEADER_SEC_WEBSOCKET_ACCEPT,
                               sec_websocket_accept);
      ret = MHD_queue_response (connection,
                                MHD_HTTP_SWITCHING_PROTOCOLS,
                                response);
      MHD_destroy_response (response);
    }
    else
    {
      /* return error page */
      struct MHD_Response *response;
      response =
        MHD_create_response_from_buffer_static (strlen (
                                                  PAGE_INVALID_WEBSOCKET_REQUEST),
                                                PAGE_INVALID_WEBSOCKET_REQUEST);
      ret = MHD_queue_response (connection,
                                MHD_HTTP_BAD_REQUEST,
                                response);
      MHD_destroy_response (response);
    }
  }
  else
  {
    struct MHD_Response *response;
    response =
      MHD_create_response_from_buffer_static (strlen (PAGE_NOT_FOUND),
                                              PAGE_NOT_FOUND);
    ret = MHD_queue_response (connection,
                              MHD_HTTP_NOT_FOUND,
                              response);
    MHD_destroy_response (response);
  }

  return ret;
}
```

```
int
main (int argc,
      char *const *argv)
{
  (void) argc;               /* Unused. Silent compiler warning. */
  (void) argv;               /* Unused. Silent compiler warning. */
  struct MHD_Daemon *daemon;

  daemon = MHD_start_daemon (MHD_USE_INTERNAL_POLLING_THREAD
                             | MHD_USE_THREAD_PER_CONNECTION
                             | MHD_ALLOW_UPGRADE
                             | MHD_USE_ERROR_LOG,
                             PORT, NULL, NULL,
                             &access_handler, NULL,
                             MHD_OPTION_END);

  if (NULL == daemon)
    return 1;
  (void) getc (stdin);

  MHD_stop_daemon (daemon);

  return 0;
}
```