HTTPd-1 — Subject

version **#84fae3c4944d6c55849e55a61bd313ee04600cde**

Assistants C/Unix 2024 `<assistants@tickets.assistants.epita.fr>`

Copyright

Contents

*https://intra.forge.epita.fr

Obligations

*Obligations are **fundamental** rules shared by all subjects. They are non-negotiable and to not apply them means to face sanctions. Therefore, do not hesitate to ask for explanations if you do not understand one of these rules.*

Obligation #0: **Cheating**, as well as sharing source code, tests, test tools or coding-style correction tools is **strictly forbidden** and penalized by not being graded, being flagged as a cheater and reported to the academic staff.

Obligation #1: If you do not submit your work before the deadline, it will not be graded.

Obligation #2: Your submission repository must be **clean**. Except for special cases, which (if any) are **explicitely** mentioned in this document, an *unclean* repository may contain:

- binary files;[1]
- files with inappropriate privileges;
- forbidden files: `*~`, `*.swp`, `*.o`, `*.a`, `*.so`, `*.class`, `*.log`, `*.core`, etc.;
- a file tree that does not follow our specifications.

Obligation #3: All your files must be encoded in ASCII or UTF-8 without BOM.

Obligation #4: When examples demonstrate the use of an output format, you must follow it scrupulously.

Obligation #5: The coding-style needs to be respected at all times.

Obligation #6: **Global variables** are forbidden, unless they are **explicitly** authorized.

Obligation #7: Anything that is not **explicitly** allowed is **disallowed**.

Advice

- ▷ Read the *whole* subject.
- ▷ If the slightest project-related problem arise, you can get in touch with the assistants.

  Post to the dedicated **Discourse category** (with the appropriate **tag**) for questions about this document, or send a **ticket** to **<assistants@tickets.assistants.epita.fr>** otherwise.
- ▷ In examples, `42sh$` is our prompt: use it as a reference point.
- ▷ Do **not** wait for the last minute to start your project!

[1]If an executable file is required, please provide its sources **only**. We will compile it ourselves.

1 Introduction

1.1 The project

The goal of this project is to write your own HTTP[1] server, following the HTTP/1.1 specification.

The implementation of an HTTP server follows a defined protocol. This protocol is defined in the RFCs and you **should** read them carefully to write a correct HTTP/1.1 server.

Your HTTP server must be able to run as a daemon.

You must first follow the subject specifications, and then the RFCs.

1.2 How to read this subject?

HTTPd is probably the biggest project you had to do so far. This means that you will have to be very organized and methodical in your work. You should read the whole subject before starting to code anything. You need to understand the project, its goals, and the implementation choices before starting to code. If you do not understand a concept or some parts of the subject feel free to ask an assistant to clarify it for you.

The subject will link multiple external resources. You **must** read them carefully and make sure you understand them. Some of these resources (mainly RFCs and man pages) are essential for the project and you **must** read them multiple times.

Other resources will include tutorials, guides, or concept explanations that will help you understand some basic implementation details in network programming. You are **not** expected to read them completely, but you **should** at least take a look at them.

The project has been divided into different modules to help you organize your work. Make sure to not only understand the required modules individually but also how they should interact with each other to form a basic HTTP server. Focusing on one module at a time **should** help you to go through the project smoothly.

1.3 RFC

As you may have guessed, HTTP is not a trivial protocol and this document only provides a quick overview to help you get started. To find every piece of information you will need to create your own HTTP/1.1 server, you **should** read and follow the RFCs on this topic.

RFC (Requests For Comments) are informational documents. They describe how most web technologies work and some are used as internet standards. You should follow them to create your HTTPd. The most important ones for you are listed in the annex section of the subject.

[1] https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol

2 HTTPd

You **must** implement the core features for your daemon. The bonus features are optional. You should have a minimum functional HTTPd project before trying to implement bonus features.

The core features are necessary for any HTTP server daemon and will help you learn the basics of network programming along with some system concepts such as daemons, asynchronous programming, and advanced Linux system calls. You will also need to use your parsing knowledge to resolve the different user requests.

The bonus features will mostly showcase performance and security features that will require system programming skills and UNIX proficiency.

HTTPd stands for HTTP **daemon**, as such, you will also have to handle the necessary system operations for it to run as a Linux service.

Be careful!

All the core features are **mandatory** for the assessment of the rest of the project.

2.1 Architecture

2.1.1 Exercise

File Tree

```
httpd/
├─ Makefile  (to submit)
├─ src/
│  ├─ Makefile  (to submit)
│  ├─ config/
│  │  ├─ *  (to submit)
│  │  ├─ Makefile  (to submit)
│  │  └─ tests/
│  │     ├─ *  (to submit)
│  │     └─ config_test.c  (to submit)
│  ├─ flags.mk  (to submit)
│  ├─ libs.mk  (to submit)
│  ├─ main.c  (to submit)
│  ├─ utils/
│  │  ├─ *  (to submit)
│  │  ├─ Makefile  (to submit)
│  │  └─ tests/
│  │     └─ *  (to submit)
│  └─ {module_name}/
│     ├─ *  (to submit)
│     ├─ Makefile  (to submit)
│     └─ tests/
│        └─ *  (to submit)
└─ tests/
   └─ *  (to submit)
```

Compilation : Your code must compile with the following flags

- -std=c99 -Werror -Wall -Wextra -Wvla

Authorized functions : You are only allowed to use the following functions

- getopt
- getaddrinfo
- freeaddrinfo

- gai_strerror
- fcntl
- open
- close
- dup
- dup2
- fork
- getpid
- waitpid
- _exit
- read
- write
- lseek
- sendfile
- isatty
- inet_ntop

Authorized headers : You are only allowed to use the functions defined in the following headers

- assert.h
- ctype.h
- errno.h
- fnmatch.h
- limits.h
- signal.h
- stdarg.h
- stdbool.h
- stddef.h
- stdio.h
- stdlib.h
- string.h
- time.h
- pthread.h
- err.h
- sys/epoll.h

- sys/eventfd.h
- sys/socket.h
- sys/types.h
- sys/stat.h

2.1.2 Project Structure

Building HTTPd

HTTPd is the largest project you have worked on this year until now. It is a project that comprises various smaller projects, and while code modularization is essential, connecting everything can become quite intricate. As a project expands in size, it brings forth new challenges, including writing maintainable code and establishing a sustainable architecture.

Tips

Maintainable means that you do not want to pull your hair out when opening the project directory or reading the code...

Building the project also becomes more complex, which is why we have **build systems**.

Modules

When a project is set to be used in another project (e.g: malloc), then a library is preferred to a binary since they are binary object files that have yet to be linked. Since you can see HTTPd as a composition of smaller projects, each sub-project is considered to be a **module**.

Tips

For HTTPd, we will build modules as static libraries.

A **module** is a part of the program that separates different functionalities of a project. You might want to compile this module differently according to its specific needs: flags, macros, or non-standard extensions. Sometimes, you have to make trade-offs or concessions. By constructing a project using modules, these trade-offs can be contained within a specific part of the project and not affect the entire project as a whole.

In HTTPd, modules will be enforced. It is an entire problem on its own that you will discover in due time and we do not want you to focus on that (yet).

Since the target will be a binary, your root Makefile must be able to build every module in your project and link them to the resulting binary. This implies that each module **must** be built using its own `Makefile` that is called by the root `Makefile` **recursively**.

Going further...

Modern projects are built with build systems such as CMake, Autotools , or Meson. They offer lots of abstractions when compared to simpler build tools like GNU Make. They allow recursive configuration files to specify how each source file is compiled.

Bear in mind that build systems do not by default build the project, but they rather generate the build files using the target build tool.

For now, you will (sadly) not be asked to use a build system. To better understand the challenges behind this complex architecture, we would rather have you build your project with `Makefiles`.

Recursive Makefiles

Tips

In this section, we assume that you are already familiar with `make` and simple Makefiles. For a refresher, you may look into the various Piscine tutorials on Makefiles as well as the recorded conference.

When using `make(1)`, the option `-C` can be used to specify the build directory. Make will then move into the directory and run as usual using the Makefile in it.

```
42sh$ ls Test./
Makefile votai.c

42sh$ cat Test./Makefile
votai: votai.o

42sh$ make -C Test./
make: Entering directory '/home/lucasTheSpider/Test.'
cc    -c -o votai.o votai.c
cc   votai.o   -o votai
make: Leaving directory '/home/lucasTheSpider/Test.'

42sh$ Test./votai
Votai Test.
```

This is equivalent to `cd Test./ && make && cd -` but with additional information about which directory you are calling and which commands are called.

As you may recall, a Makefile is composed of rules, and rules are composed of three things :

- targets (The goal to produce)
- dependencies (What the target needs prior)
- a recipe (Composed of various commands to run)

Since recipes are composed of any command, it is possible to declare a rule that can call another `Makefile` if needed.

```
42sh$ tree
.
|-- Makefile
`-- Test.
    |-- Makefile
    `-- votai.c
```

(continues on next page)

(continued from previous page)

```
2 directories, 3 files

42sh$ cat Makefile
all:
        make -C Test./

42sh$ make
make -C Test.
make[1]: Entering directory '/home/lucasTheSpider/Test.'
cc    -c -o votai.o votai.c
cc   votai.o   -o votai
make[1]: Leaving directory '/home/lucasTheSpider/Test.'

42sh$ Test./votai
Votai Test.
```

Now, the root Makefile is considered the top-level Makefile, and every Makefile called by it is considered as a sub-Makefile. Each Makefile handles its own environment, so changing the flags of a sub-Makefile does not affect the top-level one but modifying the top level will change the environment for the sub-Makefiles.

Tips

Think of it as 'scoping'

Since each module will contain its own Makefile, building HTTPd comes down to calling make on each module folder and linking everything at the end. This mechanic is quite complex and can be quite daunting at first. Luckily for you, we decided to give you the Makefile (with a catch ...)!

Given Makefile

In HTTPd, three Makefiles are given :

- The root Makefile containing the target to build the `httpd` binary.
- The Makefile in `src/` that will handle most of the work
- A Makefile in config that builds `libconfig.a` to be linked.

Additionally, two `.mk` files are provided :

- `flags.mk` which contains common flags for each module.
- `libs.mk` that should contain every target that builds a module.

Since modules are dependencies for the `httpd` binary, the targets that build modules must be added to the list of dependencies and each module needs to have a defined target that will be used to build the dependency. `libs.mk` makes it easy to build any dependency needed by the binary. For example :

```
include flags.mk

OBJ = main.o
LIBS=config/libconfig.a
```

(continues on next page)

(continued from previous page)

```
all: httpd

httpd: $(OBJ) $(LIBS)
	$(CC) $(LDFLAGS) -o $@ $(OBJ) -Wl,--start-group $(LIBS) -Wl,--end-group

include libs.mk

.PHONY: all
```

```
SRC_DIR := $(dir $(lastword $(MAKEFILE_LIST)))

config/libconfig.a:
	$(MAKE) -C $(SRC_DIR)config
```

To build the binary, the `main.o` object file is required but we also need to link with the config library. Using the `include` directive allows us to copy the contents of `libs.mk` and paste it to our Makefile. When we ask for `config/libconfig.a`, we will call the target that was declared in `libs.mk` which calls the Makefile in config.

Tips

Do not worry about `SRC_DIR`, you only need to understand that it computes the value once and it corresponds to the path to the *src* directory. This guarantees that no matter where you call the target, the correct path will be used.

As always, you should take a look at the given files and ask an assistant for help if needed.

Be careful!

Libraries are glorified objects that can contain multiple other compressed object files. Some symbols can still be marked undefined as they are not needed when building. This means that you may need to link with other libraries or object files to resolve undefined symbols.

Expected Makefile

You *Makefiles* **MUST** respect the following rules:

Path	Target	Produces	Description
./httpd/Makefile	httpd	httpd	Produces the httpd binary
./httpd/Makefile	check	X	Runs your testsuite(s)
./httpd/src/config/Makefile	config	libconfig.a	Produces the libconfig.a archive
./httpd/src/utils/Makefile	utils	libutils.a	Produces the libutils.a archive
./httpd/src/daemon/Makefile	daemon	libdaemon.a	Produces the libdaemon.a archive
./httpd/src/logger/Makefile	logger	liblogger.a	Produces the liblogger.a archive
./httpd/src/server/Makefile	server	libserver.a	Produces the libserver.a archive
./httpd/src/http/Makefile	http	libhttp.a	Produces the libhttp.a archive

Be careful!

The path of the produced files is **relative** to the associated *Makefile* location.

2.2 HTTPd modules

To simplify the development of this project, we provide you with a mandatory architecture to follow. We will define several modules that will have to implement specific entry points.

Your code will have to generate static libraries for each module that will implement entry points defined in the given header files.

2.2.1 Config

This module will be used to load and handle configuration files.

- output: `libconfig.a`

```
struct config *parse_configuration(const char *path);
void config_destroy(struct config *config);
```

Tips

You will find more information about this module in the *Core Features* section of this subject.

2.2.2 Utils

This module will be used to handle utilities that include:

- output: `libutils.a`
  - string

```
struct string *string_create(const char *str, size_t size);
int string_compare_n_str(const struct string *str1, const char *str2, size_t n);
void string_concat_str(struct string *str, const char *to_concat, size_t size);
void string_destroy(struct string *str);
```

Be careful!

Most functions in the libc `string.h` header do not work with strings that are not null-terminated. You will have to add your functions to this module to correctly handle strings.

2.3 Other Modules

Having separated modules means you will be able to easily and cleanly add features and test them individually to make sure each part of your project works before putting them together.

You **must** create the following modules.

2.3.1 HTTP

- output: `libhttp.a`

A module that will contain all the HTTP-related functions.

It will have to implement the following functionalities:

- parse HTTP requests
- create a response from an HTTP request
- convert an HTTP response to a string

Tips

You will have to implement some structures to represent responses and requests. We strongly advise reading the RFCs to understand how to represent them correctly.

2.3.2 Daemon

- output: `libdaemon.a`

A module that will help you handle the daemonization of your program.

It will have to handle forking the program and handling the different signals to respect the required behavior.

Be careful!

The implementation of each module is up to you, but we strongly suggest to at least implement the suggested behaviors described above.

Be careful!

All modules **will have to be able to compile independently**. We will test running each of your makefiles to make sure rules exist and produce the expected output.

2.3.3 Server

- output: `libserver.a`

A module that will handle the creation and execution of the server.

It will prepare a valid socket for the server and run the accept loop.

Be careful!

You should isolate the server handling and the HTTP parsing. This module must use your other HTTP module and only handle reading from a client.

2.3.4 Logger

- output: `liblogger.a`

A module that will be the logger for your server. It will be useful when debugging network interactions.

2.4 Usage

```
42sh$ ./httpd [--dry-run] [-a (start | stop | reload | restart)] server.conf
```

If the arguments do not follow this usage or if any argument is invalid your binary must return 1 and output an error on `stderr`.

If the option `-a` is not given, the binary must run the server without daemonizing it (i.e.: on the parent process).

2.5 Technical constraints

- You may request authorization to use another library if you explain why you need it using the newsgroup.
- Memory leaks and open file descriptors are a threat to a good HTTP server and will be checked.
- You can choose to go a bit further in this project. You are authorized to implement additional features as long as it doesn't break the subject's rules.
- We advise you to test the output of every system call that you will have to use during this project.
- You are not authorized to use external libraries unless specified in the subject.

Tips

When you are handling syscalls, you **must** check for `errno` in order to spot exceptions that could arise (especially for threads or called syscalls).

3 Core Features

3.1 Configuration

Since the purpose of this project is not to implement parsing algorithms, you need an easy and efficient way to provide parameters to your server. The first step in building your HTTP Server is to be able to read the configuration file.

If the --dry-run option is passed to the program, it is expected to perform a dry-run: parse the configuration file, check that it is valid, and then exit with 0. If the configuration file is invalid, your server must instead return 2 and output an error on stderr.

The format is very simple as it consists of tag sections containing pairs of keys and values.

Here is an example of a configuration file:

```
[global]
log_file = server.log
log = true
pid_file = /tmp/HTTPd.pid

[[vhosts]]
server_name = images
port = 1312
ip = 127.0.0.1
root_dir = votai/test.
```

Tips

The first line of the file will always be the tag `[global]`.

Here are the keys to the global tag section:

Name	Description	Mandatory
pid_file	**Absolute** path to the file containing the PID of the daemon.	Yes
log_file	**Relative** path to the file where your logs must be written.	No
log	Whether or not you should write logs.	No

Tips

A tag section will always end with an empty line.

Be careful!

Make sure you understand the difference between relative and absolute paths.

The second tag you will have to parse is the tag `[[vhosts]]`.

Tips

The configuration **must** contain at least one `[[vhosts]]` tag section. For now, consider a single `[[vhosts]]` tag section to simplify your parser.

It will contain the following keys:

Name	Description	Mandatory
server_name	Hostname of the server.	Yes
port	Port of the server.	Yes
ip	IP of the VHost.	Yes
root_dir	Location on the host machine where the server can serve files.	Yes
default_file	Default file to serve when requesting a directory.	No

The configuration file will always be valid. The only error case you will encounter is a missing mandatory key. If that is the case your server **must** stop and return the value of 2.

Also, note that values cannot contain spaces, you will never encounter the following: "key = value1 value2 value3"

A complete configuration example can be found in the annex section.

Tips

You will understand the purpose of each key while reading the subject.

Tips

You could take a look at the `strchr(3)` function (and more generally the `string.h` header), it should prove quite useful and should save you some time.

Now that you can read the parameters passed to your program, the real work can begin!

3.2 Basic Server

The goal of this section is to concentrate on having a functioning server. This means that before daemonizing your server, you must make sure that the binary can run on the parent process if the `-a` option is not specified.

3.2.1 Network programming

The Beej's guide is an excellent introduction to network programming, we advise you to read it carefully.

As seen in Beej's Guide, a network layer[1] protocol known as IP is used to communicate over the Internet. There are two widely used versions of this protocol: IPv4 and IPv6. **In HTTPd, you only have to handle IPv4. Do not implement anything related to IPv6.**

IP is not the only protocol HTTP relies on, HTTP/1.1 uses TCP as its transport layer protocol[2]. It allows the server and client to create a reliable connection between the two with error detection, and to guarantee the order of the received packets.

[1] https://en.wikipedia.org/wiki/Network_layer
[2] https://en.wikipedia.org/wiki/Transport_layer

Be careful!

Whenever a client closes a socket while a process is still writing on it, the process will receive the signal SIGPIPE, which by default will stop the process.

However, a server **must not** consider this signal as fatal since it may happen often, and should never shut down.

You **must** take a look at `sigaction(2)`.

3.2.2 Echo Server

The first step to understanding network programming is usually to implement an echo server. It is a server that echoes whatever it receives from a client.

Tcpbin provides a simple echo server that you can play around with to visualize how it works.

Here is an example of a session with the echo server:

```
42sh$ nc tcpbin.com 4242
Hello World! // written on stdin
Hello World! // sent back by the server
```

`nc` stands for Netcat, it is a very useful tool that is explained in the `Test` section.

Tips

It is **strongly** advised to first implement a simple echo server before. Most of the code will be reused for the HTTP server. It will allow you to better understand the concepts of sockets.

3.3 Daemon control

Servers are usually running in the background as daemons. We need an elegant way to ask our server to start, stop, restart, and reload. UNIX signals perfectly fit our needs.

To create a daemon:

1. fork
2. close the parent's process's file descriptors to avoid undefined behaviors like having the logs written twice
3. terminate the parent.

Your child will be assigned to `init`.

```
#include <unistd.h>
#include <stdio.h>

int main(void)
{
  pid_t val = fork();
```

(continues on next page)

(continued from previous page)

```
  /* check fork errors */

  if (!val)
    /* inside daemon */
  else
    printf("%d\n", val);

  return 0;
}
```

In this example, the parent will terminate and the child will be running in the background.

Be careful!

This is the only time you are allowed to use `fork(2)`.

3.3.1 Configuration

Here is the configuration variable that you will use:

Name	Description	Mandatory
pid_file	**Absolute path** to the file containing the PID of the daemon.	Yes

3.3.2 Argument

With this feature, a new **optional** argument can be passed to the *httpd* binary: `-a ACTION`. It will be used to specify an action to your server. ACTION values can be:

- `start`
- `stop`
- `reload`
- `restart`

If any other value is given or if multiple `-a` are present your binary must stop and return 1.

Be careful!

Note that after this level, your program must work when started with a `-a` option, but it must **still work** when started *without* a `-a` option (which will not daemonize the server).

3.3.3 Starting

When the server is launched with `-a start`, the *new* PID must be stored in the file indicated by the `PID_FILE` configuration variable. If this file does not exist, you must create it. If the file exists and contains a PID, check if the process is alive. If it is, your binary must stop and return 1. If it is not, delete the content and proceed as specified above.

This PID will be used every time you need to send a signal to the server.

```
42sh$ cat server.config
...
pid_file=/tmp/HTTPd.pid
42sh$ ./httpd -a start server.config
42sh$ pgrep httpd
5396
42sh$ cat -e /tmp/HTTPd.pid
5396$
```

3.3.4 Quitting

When `-a stop` is given, check if the file exists and if the process is alive. If it is, a `SIGINT` is sent to the server, which will handle this signal and stop itself. After this step (or if the process is not alive), delete the `PID_FILE` and return 0.

3.3.5 Reloading

When started with `-a reload`, a signal is sent to the server. Once the signal is received, `HTTPd` will reload: the configuration file will be read again, updating the different vhosts, but the vhosts shall not be stopped.

Since there is no predefined signal corresponding to the behavior we want to implement, we need to use our custom signals. There exist two signals left specifically for user usage, we expect you to use those.

Tips

Take a look at `signal(7)`

Note that modification of the keys `IP`, `PORT` and `PID_FILE` will not be tested as a core feature, as they require significant work on your part.

3.3.6 Restarting

When started with `-a restart`, check the content of `PID_FILE`. If the file does not exist or if the PID refers to a dead process, perform the same as if `-a start` was passed. Otherwise, a `SIGINT` is sent to the server, which will handle it and terminate itself. Then start the daemon and rewrite the content of `PID_FILE`.

This is equivalent to calling `HTTPd` with `-a stop`, then calling it again with `-a start`.

3.3.7 Output

When started as a daemon, your server **must** not print on stdout or stderr as we do not want to flood the terminal. If no log file is given as a daemon, write to `HTTPd.log` in the current directory.

3.4 HTTP 1.1

3.4.1 Conventions

- **Uppercase words are protocol keywords.** Exceptions to this convention are `CR` ("`\r`", *carriage return*), `LF` ("`\n`", *line feed*) and `SP` (" ", *space*).
- **Printed spaces are meaningless.** Only `CR`, `LF` or `SP` are meaningful blanks.
- **The end of the headers section is denoted by two ``CRLF``.** You **must** keep this in mind while reading the rest of this section.

In order to handle an HTTP request, you must be able to read the request and send the correct file with an HTTP response. Keep in mind that you must first follow the subject's protocol, and then the RFC 9110 and RFC 9112.

Tips

For the grammar definitions, the links in the RFC 9112 are broken, you should look in the RFC 9110.

3.4.2 HTTP-message

In the RFC 9112 (section 2.1) an HTTP message is defined as follows:

```
HTTP-message   = start-line CRLF
*( field-line CRLF )
CRLF
[ message-body ]
```

The only difference between a request and a response is the start-line.

- The start line is either a request-line or a status-line, which will be explained just after.
- The field-lines, also known as headers, are additional information about the HTTP-message and how it should be handled.
- The message body, if present, is used to carry content for the request or response.

3.4.3 Request

A request is an HTTP-message sent from a client to a server used to access a resource on the targeted server.

Request-Line

RFC 9112 (section 3) says that the **Request-Line** begins with a **method** token, followed by the **Request-target** and the protocol version, and ends with **CRLF**:

```
Request-Line = Method SP Request-target SP HTTP-Version CRLF
```

- The **Method** token indicates the method to be performed on the the resource identified by the **Request-target**.
- The **Request-target** is a *Uniform Resource Identifier* that identifies the resource upon which to apply the request.
- The **HTTP-Version** indicates the protocol version used to form this request.

In the context of HTTPd the following headers should be checked at each request received regardless of its relevance:

- Content-Length: Indicates the length of the request body. This header is not mandatory but it must always be valid if present in request header.
- Host: Should describe the host that corresponds to at least one vhost in the config file

In the case of HTTPd, `Host` header is considered valid if it matches one of the following cases:

- The `server_name` of a vhost (e.g. `www.example.com`)
- The IP address of a vhost (e.g. `127.0.0.1`)
- The IP port combination of a vhost (e.g. `127.0.0.1:4242`)

Be careful!

Since the end of the headers section is indicated by two `CRLF`, make sure that you have read until the former before starting to parse the message. Data can (and will) be sent in multiple sends. Crucial information such as the content length is necessary before attempting to read the potential payload.

3.4.4 Response

A response is an HTTP-message from a server to a client used to respond to a previously received request.

In HTTP/1.1, the Date header is mandatory in every response.

Even though the Content-Length header is not mandatory in the RFC, your server must include this header in every response since it is strongly suggested.

Be careful!

Do not use the `asctime(3)` function, because according to the Date/Time Formats, its format is different. Furthermore, time must be in GMT.

Tips

You might want to take a look at `strftime(3)`.

You **must** use the **root_dir** configuration variable to find your resources. For example:

A request with target “`/lucas_the_spider.png`” on a server with root_dir “`~/best`” will get the file whose full path is “`~/best/lucas_the_spider.png`”.

Tips

If all is well, you should send a response with the status code 200 OK

Tips

You should use `sendfile(2)` when possible. It is very useful to optimize data transfer between file descriptors.

Status-Line

The first line of a response is the Status-Line, consisting of the protocol version followed by a numeric status code and its associated textual phrase. Each element is separated by **SP** characters.

```
Status-Line = HTTP-Version SP Status-Code SP Reason-Phrase CRLF
```

- The **HTTP-Version** indicates the protocol version used to form this request.
- The **Status-Code** element is the 3-digit integer result code of the attempt to understand and satisfy the request.
- The **Reason-Phrase** is intended to give a short textual description of the **Status-Code**; this is not defined by the RFC and it is intended for humans only.

Be careful!

You will have to respect the *Status-Codes* and *Reason-Phrases* defined in the section 15 of the RFC 9110.

3.4.5 Persistence

By default, the protocol HTTP/1.1 specifies that a connection between a client and a server is persistent.

It means that when a client connects to a server, it can send as many requests as it wants before closing the connection with the server.

Your server must **not** respect this behavior. The connection between the clients and the server must not be persistent.

This means that your server must handle a single request from a client, then close the connection with it.

Be careful!

Due to this behavior, each response must have the connection header set to “close”.

3.4.6 Method

You will only have to handle two methods:

- GET: **Used to get the targeted resource. It is the most commonly used** method. Almost all requests you emit are GET requests.

As an example:

```
42sh$ curl -i https://www.vim.org/404
HTTP/1.1 200 OK
Date: Mon, 19 Sep 2022 15:52:45 GMT
Server: Apache/2.4.10 (Debian)
Content-Location: 404.php
Vary: negotiate
TCN: choice
Content-Length: 447
Content-Type: text/html
Content-Language: ja


<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Vim online - Page not found</TITLE>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
</HEAD>
<BODY BGCOLOR="#FFFFFF">
<h2>404 document not found</h2>
<BLOCKQUOTE><TT>/404</TT></BLOCKQUOTE>
does not exist.
<P>
Try vim online at <a href="http://www.vim.org/">http://www.vim.org/</a>.
</BODY>
</HTML>
<!-- vim: set ts=3 sts=3 sw=3: -->
```

- **HEAD: Same as GET, but it will not send the body. It is used to get some** information about the resource without the resource itself, as it is way faster.

As an example:

```
42sh$ curl -i --head https://www.vim.org/404
HTTP/1.1 200 OK
Date: Mon, 19 Sep 2022 15:52:45 GMT
Server: Apache/2.4.10 (Debian)
Content-Location: 404.php
Vary: negotiate
TCN: choice
Content-Length: 447
Content-Type: text/html
Content-Language: ja
```

3.5 HTTP Error Handling

3.5.1 Status-line Related Errors

- 403 Forbidden: Raised when a client tries to access a resource it does not have permission to.
- 404 Not Found: Raised when a client tries to access a resource that does not exist.
- 405 Method Not Allowed: Raised when a method not supported/allowed by the server is used in a request.
- 505 HTTP Version Not Supported: Raised when an HTTP version not supported by the server is used in a request. Anything other than 1.1 in our case.

3.5.2 Malformed Request Error

This part will focus on security and handling complex requests. Web servers are inherently complex programs and severe vulnerabilities like Heartbleed or Shellshock can have a huge impact. Once publicly disclosed, vulnerabilities are given an identifier and referenced in the Common Vulnerabilities and Exposure databases, or CVE. In this part, we ask you to make sure your web server is protected against some common vulnerabilities that can leak sensitive data or crash your server using malicious requests:

- Invalid syntax in the client's request.
- Malformed requests using `CRLF` at random positions.

Your server must not crash in case of a malformed request. Catching `SIGSEGV` is forbidden and will be considered cheating. In case of an error, you must be able to throw 400 Bad Request if the request is malformed.

3.6 Null Bytes

Since you are receiving bytes by reading from the socket fd, you must bear in mind that there is no standard way to know if you have finished reading everything. This means that you can no longer use the null byte as the end of a string since the notion of string is no longer applicable.

This implies that should a user send you null bytes (`\0`), there could still be some data left to be read. Your server **must** continue reading beyond the null byte if you still need to read anything. Meaning you can have a header with the value "Host: serv\0name"

Tips

If you were not before, you should be using the `string` structure that was provided to help you.

3.7 Default File

You will often find that the target of a request is a directory. This kind of request is perfectly valid and should be handled. To do so, you will have to use what we call a `default_file`. It corresponds to the file to serve when requesting a directory.

Here is a reminder of the variable in the configuration:

Name	Decription	Mandatory
default_file	Default file to serve when requesting a directory.	No

If no default file is specified in the configuration, it will have a default value of "`index.html`"

3.8 Logging

Most moderns application log what they do, as it is very useful for debugging purposes. Thus we also ask you to do some logging, but only the most basic one. Of course, if you want to log more than what is asked, you can. But you **must** log the necessary lines, as requested.

Here are the two configuration variables you will use in this section:

Name	Description	Mandatory
log_file	Relative path to the file where your logs must be written.	No
log	Whether or not you should write logs.	No

By default, `log` should be considered true. If you are asked to log, when no `log_file` is provided, you should log on stdout.

Tips

We advise you to implement this feature as soon as possible if you do not want to end up in a debugging hell.

Be careful!

If the `log` variable is set to false, you **must not** create a log file.

3.8.1 Requests

Every request received should be logged using the following format:

```
DATE [SERV_NAME] received REQUEST_TYPE on 'TARGET' from CLIENT_IP
```

Be careful!

As for the Date header, you should follow the GMT format.

Which will give you something like this:

```
Sun, 06 Nov 1994 08:49:37 GMT [localhost] received GET on '/index.html' from 218.20.4.2
```

For `Bad Request`, since the request will be ill-formed you should replace the target with `Bad Request`:

```
Sun, 06 Nov 1994 08:49:37 GMT [localhost] received Bad Request from 218.20.4.2
```

3.8.2 Responses

Every response sent should be logged using the following format:

```
DATE [SERV_NAME] responding with STATUS_CODE to CLIENT_IP for REQUEST_TYPE on 'TARGET'
```

Be careful!

As for the Date header, you should follow the GMT format.

Which will give you something like this:

```
Mon, 07 Nov 1994 20:42:27 GMT [localhost] responding with 200 to 218.20.4.2 for GET on '/
↪index.html'
```

For `Bad Request`, since the request will be ill-formed you should remove the `TARGET` and `REQUEST_TYPE` parts:

```
Mon, 07 Nov 1994 20:42:27 GMT [localhost] responding with 400 to 218.20.4.2
```

For `Method Not Allowed` you must put `UNKNOWN` instead of `REQUEST_TYPE`:

```
Mon, 07 Nov 1994 20:42:27 GMT [localhost] responding with 405 to 218.20.4.2 for UNKNOWN on '/
↪index.html'
```

3.9 Graceful shutdown

Once your server has started, you need a way to stop it. Killing the process by sending a non-handled signal is not the correct way to stop it. Some signals can be sent to your program using combinations. For example:

- `CTRL+C` sends `SIGINT`
- `CTRL+Z` sends `SIGTSTP`

We must be able to stop your running server by using `CTRL+C`. See `sigaction(2)` to catch `SIGINT` and perform a clean shutdown of your server (sockets, threads, children ...). The return value must be 0.

Be careful!

Functions like `on_exit(3)` or `at_exit(3)` that would allow us to clean before exit are not called when a signal is sent.

Be careful!

Signal handlers are run asynchronously. This means you will have to use signal-safe functions in your handler. See `signal-safety(7)` for more information. Note that `exit(3)` **is not** signal-safe.

4 Testing

4.1 cURL

You are highly advised to use `curl(1)` to test your server. It is a command line tool that you can use to send HTTP requests to your server. cURL will be very useful when developing your server as it allows you to see the HTTP request headers. Feel free to read more on the cURL man page.

Here's what curl should look like on your HTTPd.

```
42sh$ curl -v http://127.0.0.1:4242
*   Trying 127.0.0.1:4242...
* Connected to 127.0.0.1 (127.0.0.1) port 4242
> GET / HTTP/1.1
> Host: 127.0.0.1:4242
> User-Agent: curl/8.4.0
> Accept: */*
>
< HTTP/1.1 200 OK
< Date: Sat, 01 Nov 2022 23:42:00 GMT
< Content-Length: 22
< Connection: close
<
<h1>Hello World !</h1>
* Closing connection
```

Every line starting with '*' is some meta information cURL will display.

Every line starting with '>' is the request that will be sent.

Every line starting with '<' is the response that HTTPd will send.

On the first line of the request, you will find the request-line:

- `GET` is the method
- `/` is the request target
- `HTTP/1.1` is the HTTP-version

the rest are Headers generated by curl.

On the first line of the response, you will find the status line:

- `HTTP/1.1` is the HTTP-version
- `200` is the status code
- `OK` is the reason phrase

the rest are Headers generated by HTTPd followed by the `body`.

To help you out, here is a non-exhaustive list of options you might find useful:

- `-i` **includes the HTTP headers and status-line of the response in the cURL** output.
- `-I` **performs a `HEAD` request and displays the headers in the cURL** output.
- `-v` enables the cURL verbose mode.
- `-H` includes extra headers or overrides existing ones in your request.
- `-X` chooses the request method to use (`GET`, `POST`...).
- `-d` **sends the specified data in a `POST` or `PUT` request to the HTTP** server.

4.2 netcat

netcat (binary called `nc(1)`) is a networking utility that is used to open connections and operate on them. It can be used, for instance, to check if ports are open, or to send irregular packets to a host. Once the connection opens, netcat will forward data from standard input into the socket.

Multiple versions of Netcat exist. We are using the OpenBSD one.

```
42sh$ echo -ne 'GET /index.html HTTP/1.1\r\nHost: localhost\r\n\r\n' | nc localhost 4242
HTTP/1.1 200 OK
Date: Sat, 01 Nov 2022 23:42:00 GMT
Content-Length: 22
Connection: close

<h1>Hello World !</h1>
```

Tips

You can use `nc -C <IP> <PORT>` to have a better hands-on way of testing more complicated nonconforming requests and tricky IO situations.

```
42sh$ nc -C 127.0.0.1 8080
```

4.3 Unit Tests

The advantage of using independent modules is that you can test them independently. You **must** write unit tests for each module you write using criterion. The tests **must** be able to be launched with the `make check` rule inside the module directory. The `make check` rule at the root of the project should launch all the tests of all the modules.

Be careful!

Writing unit tests is **MANDATORY**. You will have to write enough tests to cover a big portion of your code. **This will be tested**.

4.4 Creating a testsuite

During this project, you will need a functional testsuite to ensure your HTTP Daemon is working properly and follows every requirement. It will also help you avoid regressions if used correctly.

Use Python or any other scripting language of your choice. However, we will not be able to help you with your testsuite if it is not written in *Python*. *Python* is a safe choice as it can be learned quickly, and has all the modules you may need to test your HTTP server.

4.4.1 Useful modules

The requests module is one of the best HTTP client APIs in Python. It is easy to use, well-documented, and complete.

However, this module won't be enough to test all the features we ask you to implement, especially the network problems. Take a look at the socket API to test some corner cases.

The subprocess module will be useful to launch your daemon before running your tests. You can also use it to verify the return value of your daemon and make sure it has properly exited.

Try to make things generic, to add new tests easily as your project grows.

4.4.2 Virtual Environment

You **should** use a `virtualenv` and `pip` to install the needed modules. Have a look at `pylint`, `logging` or `subprocess`. We strongly advise you to **not** use Python's *os.run* to run your daemon.

Be careful, during a permanence we **MUST** be able to launch your testsuite with `make check`. It **MUST** build your environment, install all the requirements, and then launch the tests.

```
42sh$ python -m venv env
42sh$ source env/bin/activate
42sh$ python -m pip install -r requirements.txt
42sh$ ./<your testsuite>
```

Tips

You can use `pip freeze` to generate a `requirements.txt` file containing all the modules installed in your environment.

```
42sh$ python -m pip freeze > requirements.txt
```

Be careful!

Environment files are considered trash files. You should only provide your Python scripts.

4.4.3 Testing framework

To make your testsuite more readable and easier to maintain, you should use a test framework. You will have to use pytest.

It is a very powerful framework that will help you write your tests very quickly.

Pytests has some very easy conventions to follow to start writing tests. Your test files must be named `test_*.py` or `*_test.py`. Your test functions must be named `test_*`. You can read more about it in the pytest documentation.

You can also add very useful plugins such as pytest-timeout to make sure your test get interrupted after a certain amount of time.

Going further...

Code coverage measures the amount of code covered by your tests. It is an important metric to know how well your tests are and which tests you will need to add.

You can use pytest-cov to get a coverage report of your tests.

Going further...

Your testsuite will most likely have to set up an environment and launch your binary before running the tests. We advise you to look into pytest fixtures to do so.

5 Bonus Features

5.1 Path Traversal Attacks

This part is a security feature.

Usually, your server serves files contained in its 'root_dir'. But what if someone tried to GET files or directories outside of this root directory, for example, putting "../../passwords/bank.txt" as the target of a request.

This could become an enormous problem, and that is what *path traversal* attacks are about: they try to get arbitrary files and directories stored on the filesystem[1].

Thus, unless you want your server to leak every single file you have, you have to handle this type of request.

5.2 Daemon VHost reloading

As we mentioned before, the modification of the keys `IP`, `PORT` and `PID_FILE` during a reload will not be tested as a core feature. The modification of the `IP` and `PORT` keys is, however, tested as a bonus feature.

To implement this feature, you must now handle changes in your VHosts configurations when a reload is performed.

5.3 VHosts

The term Virtual Host, often called vhost, refers to the practice of running multiple websites on a single server. These virtual hosts can be **IP-based** (each vhost has its IP address) or **name-based** (each vhost has its name but they all run on the same IP address). Each vhost will have its own `root_dir`, containing its files. For example, `cri.epita.fr` and `intra.forge.epita.fr` are named-based vhosts.

In our case, we will have an **IP-based** implementation since it is simpler.

Be careful!

You should take a look at `setsockopt(2)` and the flag `SO_REUSEADDR`

5.3.1 Configuration

Each vhost will be defined by a name, the IP address, and the Port it is bound to. As such, each vhost **must** have a different IP/Port combination. The root directory, the server name, and the default file may be different for each vhost.

Tips

You do not have to handle multiple `[[vhosts]]` tags as long as you do not support this feature.

[1] https://www.owasp.org/index.php/Path_Traversal

Here is an example of a configuration file with multiple vhosts:

```
[global]
log_file = server.log
log = true
pid_file = /tmp/HTTPd.pid

[[vhosts]]
server_name = images
port = 4243
ip = 127.0.0.1
default_file = club_des_frais.html
root_dir = tests/fake_images/

[[vhosts]]
server_name = files
port = 8080
ip = 127.0.0.2
default_file = password.html
root_dir = tests/precious_files/
```

6 Annex

6.1 Network programming core guidelines

- Beej's guide to network programming
- Socket programming in C

6.2 RFC

- RFC 9110
- RFC 9112

6.3 Useful functions

6.3.1 Networking

- `getaddrinfo(3)`
- `socket(2)`
- `bind(2)`
- `listen(2)`
- `accept(2)`
- `send(2)` and `recv(2)`

- `close(2)`

Tips

We strongly advise you to read the read the manpages of these functions along with the beej guide to fully understand how they work.

6.3.2 Daemon

- `fork(2)`
- `signal(2)`
- `sigaction(2)`

6.4 TCP

- https://www.khanacademy.org/computing/computers-and-internet/xcae6f4a7ff015e7d:the-internet/xcae6f4a7ff015e7d:transporting-packets/a/transmission-control-protocol--tcp
- https://users.cs.northwestern.edu/~kch670/eecs340/proj2-TCP_IP_State_Transition_Diagram.pdf

6.5 Complete Configuration Example

```
[global]
log_file = server.log
log = true
pid_file = /tmp/HTTPd.pid

[[vhosts]]
server_name = images
port = 4243
ip = 127.0.0.1
default_file = club_des_frais.html
root_dir = tests/fake_images

[[vhosts]]
server_name = files
port = 6666
ip = 127.0.0.3
root_dir = tests/fake_files
```

I must not fear. Fear is the mind-killer.