In our context, we use HTTP protocol to manage a remote file system through HTTP request and responses. Therefore, our goal is to build a file server application on top the developed HTTP server library.

Important Note:

You must use only the bare-minimum socket APIs provided by the chosen programming language. Therefore, you must not leverage any library that could abstract the socket programming.

Study and review HTTP Protocol

In this step, you are asked to review HTTP protocol. In this context, you should focus on the server side features, since you are requested to implement some of them. We urge you to consider HTTP version 1.0 due to its simplicity and easy to implement. You can find the complete specifications of HTTP protocol version 1.0 in this web link HTTP[4]. Furthermore, you can make tests by sending requests and receiving HTTP server responses using the **Telnet [3]** command line. The pervious tests could show the typical HTTP responses for your testing requests.

Develop HTTP Server Library

In this part, you are requested to develop your HTTP server library separately from application, to decouple the HTTP protocol specification from the intended end-application. Your HTTP library should be self-contained with a minimum dependency such as Socket library.

You are required to implement only a subset of the HTTP specifications. In essence, the library should include the features that can handle the requests from the **httpc** app of Assignment #1. To this end, you can test your library by implementing testing examples to check if it is working properly with client applications.

Build a File Server Application Using Your HTTP Library

In this task, you are required to make an end-application to the previous library functionalities. In other words, you should build a remote file server manager on top the library according to the following requirements:

1- `GET /` returns a list of the current files in the data directory. You can return different type format such as JSON, XML, plain text, HTML according to the `Accept` key of the

header of the request. However, this is not mandatory; you can simply ignore the header value and make your server always returns the same output.

2- `GET /foo` returns the content of the file named `foo` in the data directory. If the content does not exist, you should return an appropriate status code (e.g. HTTP ERROR 404).

3- `POST /bar` should create or overwrite the file named `bar` in the data directory with the content of the body of the request. You can implement more options for the POST such as `overwrite=true|false`, but again this is optional.

Secure Access

Your implementation may have a severe access vulnerability. The end-user could access not only the file of the default working directory of the server application but he/she could access most server files (read-write or read-only). To solve the previous problem, you should build a mechanism to prevent the clients to read/write any file outside the file server working directory.

Error Handling

In this task, you need to enhance the file manager with the appropriate error handlers. In this context, each exception on the server side should be translated to an appropriate status code and human readable messages. For example, if the requested file does not exist, the file server should send a message with this information. Similarly, if the server is unable to process the request for security reasons (the requested file is located outside the working directory), an appropriate handling must be performed.

Usage

```
httpfs is a simple file server.

usage: httpfs [-v] [-p PORT] [-d PATH-TO-DIR]


   -v   Prints debugging messages.

   -p   Specifies the port number that the server will listen and serve at.

        Default is 8080.

   -d   Specifies the directory that the server will use to read/write
requested files. Default is the current directory when launching the
application.
```

Optional Tasks

If you have successfully completed the material above, congratulations; as you now understand the implementation of an HTTP server and network protocols in general. For the rest of this lab exercise, we have included the following optional tasks. These optional tasks will help you gain a deeper understanding of the material, and if you can do so, we encourage you to complete them as well. Bonus marks will be given for that.

Multi-Requests Support

Your implementation of the file server may support only one client at given moments. Therefore, if you run multiple clients simultaneously, the server can answer only one request. To solve this problem, you are invited to develop a concurrent implementation of the file server, where the server can handle multiple requests simultaneously. The later means that you should have a dynamic data structure to scale according to the server machine capacity.

To test your concurrent version, you can write a simple script to run multiple client instances (instances of **httpc** or **curl**). The script should take the number of client instances as a parameter. If you support concurrent requests, make sure the following scenarios work correctly:

- Two clients are writing to the same file.
- One client is reading, while another is writing to the same file.
- Two clients are reading the same file.

Content-Type & Content-Disposition Support

Set appropriate values to the headers **Content-Type** and **Content-Disposition** headers for 'GET /file' requests. For more details, the you could consult

---