RE2xx - Networks and Distributed Applications

Alexis Carle & Benjamin Grolleau

2024 - 2025

Enseirb-Matmeca - R&I - 2A

Table of Contents

abstract

This report presents how we have designed, implemented, and evaluated our custom HTTP proxy server, developed using the C programming language.The proxy server facilitates HTTP communication between clients and servers by providing features like request filtering based on specific domain names, configurable rules, and logging of connections. We detail the architectural decisions, discuss the functionality of core modules-including configuration management, DNS resolution with DNS cache, HTTP parsing and server handling-and explain key algorithms employed in the system. The report concludes with an analysis of challenges encountered and recommendation for future features.

1 Introduction

The exponential growth of internet usage has intensified the demand for efficient network communication and security. HTTP proxy servers play a pivotal role in managing web traffic, enhancing privacy because internet server are only requested by one client (the proxy), it also enforce organizational policies by blocking some domains that the organization considers undesirable. This is why we have choose this project and not another one. It aims to develop a robust and efficient HTTP proxy server using the C programming language.

1.1 Background

HTTP proxy servers act as intermediaries that forward client requests to web servers and relay responses back to clients. They are instrumental in tasks such as caching frequently accessed resources, filtering content based on predefined rules, and logging traffic for monitoring purposes. Implementing a proxy server in C allows for fine-grained control over system resources and network operations, leading to optimized performance.

1.2 Project Purpose

The motivation behind this project is to create a customize and efficient HTTP proxy server that can be configured to specific networking and security needs. By building the proxy server from scratch, we aim to:

- Gain a deeper understanding of how a proxy server works.
- Learn how to use socket in C and how to work with.
- Improve C level.
- Optimize performance through low-level programming techniques inherent to C.

1.3 Objectives

The specific goals of this project include:

1. Designing a Modular Architecture: Create a scalable and maintainable system structure with clearly defined modules for configuration, DNS handling, HTTP processing, logging, and server management.
2. Implementing Core Proxy Functionality: Enable the proxy to handle HTTP requests and responses effectively, including support for concurrent client connections.
3. Incorporating Configurable Features: Develop mechanisms for users to define custom rules and settings via configuration files.
4. Testing and Validation: Rigorously test the proxy server under various scenarios to validate its functionality, performance, and stability. Also create a *make test* routine to make sure that every single module of the proxy work as expected.

This report outlines the comprehensive process undertaken to achieve these objectives. It begins with a system design overview, delves into implementation details, discusses testing methodologies, and concludes with reflections on challenges faced and potential future work.

2 System Design

The design of the HTTP proxy server focuses on modularity, scalability, and maintainability. This section provides an overview of the architecture, discusses the design decisions made before and during the development, and describe the various modules that constitute the system.

2.1 Architecture Overview

The HTTP proxy server is structured using a modular architecture, separating the different part of the application. This permit an easier maintenance and scalability. The high-level architecture comprises the following primary components:

1. **Client Interface Module**: Handles incoming client connections and manages communication between clients and the proxy server.
2. **Request Processing Module**: Parses and interprets HTTP requests, applies filtering rules, and forwards valid requests to the target web server.
3. **Response Handling Module**: Receives responses from the web servers, and relays them back to the target web server.
4. **Configuration Management Module**: Manage the loading and parsing of configuration files, allowing dynamic adjustments to proxy settings.
5. **Logging Module**: Records activities, errors, and other significant events for monitoring and debugging purposes.
6. **Utility Module**: Provide common functionalities and helper functions utilized across various modules.

As shown in Figure 1 on page 14, the HTTP proxy server first connects the client with the *Client Interface Module* and then proceeds to validate the request according to predefined rules. If the request is valid, the server checks the DNS cache before forwarding the request to the target web server via the *Request Handling Module*. Finally, it forwards the response from the server to the client using the *Response Handling Module*.

2.2 Design Decisions

Several critical design decisions were made to optimize the performance and functionality of the proxy server.

- **Programming Language - C**: We did not have a choice regarding the subject, but the selection of the C programming language—known for its low-level access, efficient memory management, and performance advantages, which are essential for handling multiple concurrent connections and ensuring minimal latency—was a good option for the HTTP proxy server.
- **Modular Design**: Adopting a modular architecture allows for easier maintenance, testing, and the addition of potential future features. It permits starting with a small functional base and allows it to grow over time by adding new features step by step.
- **Configuration Flexibility**: Utilizing external configuration files (`proxy.conf` and `proxy.rules`) enables users to customize the proxy server's behavior without modifying the source code.
- **Logging Mechanism**: Incorporating a logging system aimed at monitoring server activities, diagnosing issues, and auditing traffic requests for security purposes. Logs follow a specific format (*[time] [date] [keyword] description*), as shown in Figure 2 on page 14, which allows us to easily filter and analyze them.

2.3 Scalability and Performance Considerations

To ensure that the proxy server can handle multiple concurrent connections efficiently and close them correctly after a *CTRL+C* call, several strategies were employed:

- **Signal Handling for Graceful Shutdown**: The proxy server is designed to handle interrupt signals (e.g., *CTRL+C*) through the use of the `EINTR` error code in the `poll()` system call. When a *CTRL+C* signal is received, the `errno` is set to `EINTR`, indicating that the call was interrupted. If the signal handler detects a shutdown request, the server proceeds to close all open connections and exit the main loop cleanly. This allows for an orderly shutdown, ensuring that no connections are abruptly terminated.
- **DNS Caching**: Implementing DNS caching minimizes the latency associated with repeated domain name resolutions, which is one of the most time-consuming processes for the server.
- **Efficient Memory Management**: Careful allocation and deallocation of memory resources in C prevent memory leaks and ensure optimal usage of system resources.

2.4 Security Features

Security is a critical aspect of the proxy server's design. The following measures have been integrated to safeguard against common vulnerabilities:

- **Input Validation**: All incoming data, including HTTP requests and configuration parameters, are rigorously validated to prevent malformed requests.
- **Logging**: Comprehensive logging facilitates the detection of suspicious activities and aids in forensic analysis in the event of security breaches.
- **Error Handling**: Robust error handling ensures that unexpected scenarios are managed gracefully without exposing sensitive information or compromising server stability.

3 Implemtation Details

This implementation of the HTTP proxy server is organized into several modules and sub-files, each responsible for handling a specific aspect of the system's functionality. This section describes how the code is structured and list the key functions of each module. The directory structure and the corresponding files are as show bellow:

```
.
|-- conf
|   |-- proxy.config
|   \-- proxy.rules
|-- Doxyfile
|-- Doxygen.md
|-- includes
|   \-- ...
|-- LICENSE
|-- logs
|   \-- proxy.log
|-- main.c
|-- Makefile
|-- obj
|-- README.md
|-- src
|   \-- ...
\-- test
    \-- ...
```

The `includes/` directory contains header files for each module, and the `src/` directory holds their corresponding implementations. The `test/` directory includes unit tests for each module.

Core Modules

3.1 Configuration Management (`config.c` and `config.h`)

This module is responsible for managing the server's configuration. It loads settings such as the port number, maximum clients, IP address, logger output filename and rules input filename from a configuration file (`proxy.config`), which is then used globally by other modules. The configuration filename is define in the `main.c`.

Key Functions:

- `int init_config(const char* filename)`: Loads and parses the configuration file, storing the settings in the `config_t` structure.
- `config_t config`: A global structure holding the server configuration settings such as port number, IP address, and log file paths.

Example of Configuration(`proxy.config`):

```
port=8080
address=127.0.0.1
max_client=100
logger_filename=logs/proxy.log
rules_filename=conf/proxy.rules
```

3.2 DNS Helper (`dns_helper.c` and `dns_helper.h`)

This module manages DNS resolution and caching, allowing the proxy to quickly resolve hostnames to IP addresses.

Key Functions:

- `int resolve_dns(const char* host, struct addrinfo** res, char* ipstr)`: Resolves the hostname to its IP address. First, it checks the cache; if the hostname is found in the cache, the cached result is returned. If not, the hostname is resolved, and the result is then added to the cache.
- `dns_cache_t dns_cache`: A global structure holding the cache of resolved DNS entries for efficient future lookups.

3.3 HTTP Helper (`http_helper.c` and `http_helper.h`)

The HTTP Helper module processes HTTP requests and responses. It contains functions to parse requests and generate standard HTTP responses (e.g., 404, 403).

Key Functions:

- `int is_http_method(const char* buffer)`: Checks if the incoming request contains a valid HTTP method (e.g., GET, POST).
- `int get_http_host(const char* buffer, char* host, size_t host_size)`: Extracts the target host from the HTTP request headers.

HTTP Response Macros:

- `HTTP_403_RESPONSE`: A predefined 403 Forbidden response used when the requested resource is filtered by the proxy.
- `HTTP_404_RESPONSE`: A predefined 404 Not Found response used when the requested resource cannot be found.

3.4 Logging (`logger.c` and `logger.h`)

This modules handles logging operations for the proxy. It write logs to a file specified in the configuration, recording important events such as errors, client requests, and proxy actions.

Key Functions:

- `int init_logger(const char* filename)`: Initializes the logging system and opens the specified log file.

- `void Log(LogLevel level, const char* format, ...)`: Logs messages with varying severity levels (INFO, WARN, ERROR).

An example of log entry can be found on Figure 2 page 14

3.5 Rules Management (`rules.c` and `rules.h`)

This module implements content filtering rules based on domain names, with plans to extend it to keywords in future steps. Keywords are not yet implemented, but the process will involve storing the server response in a file and filtering it for the specified keywords. This will serve as the foundation for a caching mechanism. Currently, the rules are loaded from the `proxy.rules` file and applied to each request to determine whether the request should be allowed or denied.

Key Functions:

- `int init_rules(const char* filename)`: Loads filtering rules from the `proxy.rules` file into the `rules_t` structure.
- `int is_host_deny(const char* host)`: Checks if a given hostname is on the deny list based on the loaded rules.

Example of Rules (`proxy.rules`):

```
[DarkWeb]
BAN_DOMAIN darkwebsite.com
BAN DOMAIN iamjmm.ovh
BAN_WORD darkweb
```

3.6 Server (`server.c` and `server.h`)

The Server module handles client connections, request forwarding, and response relaying. It initializes the server socket, accepts client connections, and facilitates communication between clients and target web servers.

Key Functions:

- `int init_listen_socket(const char* address, int port, int max_client)`: Creates a listening socket and binds it to the specified address and port.
- `int handle_connection(connection_t* conn)`: Processes the data exchange between the client and the server. If HTTP protocol is detected, it redirect to `handle_http`.
- `int handle_http(connection_t* conn)`: Handles HTTP communication for a specific connection, forwarding requests and relaying responses.

3.7 Utility Functions (`utils.h`)

This module provides common utility macros that are enable only when `DEBUG` flag is defined.

Key Functions:

- `#define INFO(fmt, ...) fprintf(stdout, "INFO: " fmt, ##__VA__ARGS__)`: It logs informational messages when debugging mode is **enabled**.
- `#define WARN...`: Same as INFO but with yellow color and on the `stderr` file descriptor.
- `#define ERROR...`: Same as INFO but with red color and on the `stderr` file descriptor.

4 Configuration and Usage

Effective configuration is critical for the deployment and operation of the HTTP proxy server. This is why we have choose to offer the option to reload the configuration without recompile the server. This section provides detailed instructions on how to compile the project, configure the server using the provided configuration files, and utilize the proxy server in various scenarios.

4.1 Compilation Instructions

The HTTP proxy server project utilizes a `Makefile` to autimize the build process, managing the compilation of sources files. The following steps outline how to compile the project.

1. **Prerequisites:**
   - **Compiler**: Ensure that `gcc` is installed on the system and its version is equals or higher than *13.2.0*
   - **Make utility**: Verify that `make` is also availabe for processing the `Makefile` and its version is equal or higher than *4.3*
2. **Building the Project:** Navigate to the root directory of the project and execute the `make` command.

   ```
   make
   ```

   This command performs the following actions:
   - Compiles all sources files located in the `src/` directory and the `main.c`.
   - Places intermediate object files in the `obj/` directory for organizational purposes.
   - Links the compiled objects files to crate the executable named `proxy`.
3. **Cleaning Build Output**: To remove compiled object files and the executable, use the `clean` target in the root directory of the project. This command also clean the log file.

   ```
   make clean
   ```

4. **(Option) Running the Debug Mode**: A debug mode is available and permit to have more output on the CLI about what happens inside of the program. To have it, run `make debug` in the root of the project. It will produce an executable named `proxy_debug`.

   ```
   make debug
   ```

4.2 Running the Proxy Server

Once compiled, the proxy server can be executed using the generated `proxy` (or `proxy_debug`) executable. The server relies on configuration files to determine its operational parameters. Below are the steps and options for running the proxy server:

1. **Basic Execution:**

   ```
   ./proxy
   ```

   Running the proxy will start the server using default settings specified in the `proxy.config` and `proxy.rules` file. located in the `conf/` directory. Look below to have more information about how to change the configuration files.
2. **Monitoring Server Status:**
   - **Logs:** The server writes operational logs to the file specified in the `proxy.config` (e.g., `logs/proxy.log`). Monitor this file to track server activities, errors, and other significant events.
   - **System Tools:** Use tools like `ps`, or `htop` to observe the proxy server's process and ressource usage. A huge amount of ressource used can be a warining!

3. **Stopping the Server**: To gracefully shut down the proxy server, we have bound a signal listener to our proxy. Our proxy can be killed using *CTRL+C* on the CLI or by using the `kill` command to imitate *CTRL+C* as shown below:

```
ps aux | grep proxy
kill -SIGINT <PID>
```

4.3 Configuration Files

Configuration files play a pivotal role in defining the behavior and capabilities of the HTTP proxy server. The project utilizes two configuration files `proxy.conf` and `proxy.rules`, both located within the `conf/` directory. These files allow users to customize server settings and define content filtering rules without modifying the source code.

4.3.1 `proxy.config`

The `proxy.config` file contains essential settings that dictate how the proxy server operates. Below is an example configuration with explanations for each parameter:

```
# Configuration file for the proxy

PORT 8081
ADDRESS 127.0.0.1
MAX_CLIENT 30
LOGGER_FILENAME logs/proxy.log
RULES_FILENAME conf/proxy.rules
```

Configuration Parameters:

- **port:** Specifies the port number on which the proxy server listens for incoming client connections. Default is `8080`.
- **address:** Defines the IP address to which the proxy server binds. Commonly set to `127.0.0.1` for localhost or `0.0.0.0` to listen on all available interfaces.
- **max_client:** Determines the maximum number of concurrent client connections the proxy server can handle.
- **logger_filename:** Path to the log file where the proxy server records operational logs, errors, and other significant events.
- **rules_filename:** Path to the `proxy.rules` file containing content filtering rules.

Note: The configuration can be reloaded without recompiling the server. Simply restart it, and it will automatically reload the configuration.

4.3.2 `proxy.rules`

The proxy.rules file defines content filtering policies, enabling the proxy server to block or allow specific domains. There is also some *BAN_WORD* because this is a future feature and we did the choice to already include them inside of the structure. The filtering rules functionality is crucial for enforcing organizational policies, enhancing security, and controlling access to certain web resources.

Example Rules

```
# rules for proxy

[gambling]
BAN_DOMAIN gamblingsite1.com
BAN_DOMAIN gamblingsite2.com
BAN_WORD bet
BAN_WORD casino
```

```
[darkweb]
BAN_DOMAIN darkwebsite.com
BAN DOMAIN iamjmm.ovh
BAN_WORD darkweb
```

Rules Definitions: Rules are stored by categories of 3 parameters

- **[categorie's name]:** The name of the categorie is between brackets (`[]`) and at the beginning of the set of rules.
- **BAN_DOMAIN:** Blocks access to the specified domain. Any request targeting a banned domain will be denied, and the client will receive a `403 Forbidden` response.
- **BAN_WORD:** Blocks requests containing the specified keyword. This can apply to URLs, query parameters, or content within HTTP requests. This feature is not yet implemented, but clients should also receive a `403 Forbidden` response.

Adding New Rules: To add new rules, follow the pattern and append your rules at the end of the file.

Comments and Formatting: Lines beginning with `#` are treated as comments and ignored by the proxy server. Ensure that each rule follows the correct syntax to be recognized and enforced properly.

Reloading Rules: After updating the `proxy.rules` file, restart the proxy server to load the new rules.

4.4 Usage Examples

To demonstrate the practical application of the HTTP proxy server, consider the following usage scenarios. These examples illustrate how to configure client applications to utilize the proxy server and how the server handles various types of requests.

4.4.1 Configuring a Web Browser to Use the Proxy

Most modern web browsers support proxy configuration. Below are steps to configure Firefox to route HTTP traffic through the proxy server.

1. Open Firefox and navigate to *Settings*.
2. Go to the bottom of general, in *Network Settings* and click on *Settings*.
3. Fill information with the *IP* and *Port* as show in Figure 3 page 14. Make sure to unselect *"Also use this proxy for HTTPS"*.

4.4.2 Blocking Access to Specific Domains

With the proxy server configured to block certain domains, attempting to access a blocked website will result in a *403 Forbidden* response.

1. **Define Blocked Domains:** Add the domains with the category you want to deny at the end of the `conf/proxy.rules` file.

   ```
   [Test]
   BAN_DOMAIN malware-site.org
   ```

2. **Restart the Proxy Server:**

   ```
   ./proxy
   ```

3. **Attempt to Access a Blocked Domain:**
   - Open a web browser configured to use the proxy.
   - Navigate to *malware-site.org*
   - The browser should display a *403 Forbidden* error page, and the `proxy.log` should record the blocked attempt.

4.4.3 Logging and Monitoring

The proxy server maintains detailed logs of all activities, which are invaluable for monitoring usage patterns, diagnosing issues, and auditing access.

1. **Accessing Logs:** Open the log file specified in `proxy.config`.

   ```
   cat logs/proxy.log
   ```

2. **Interpreting Log Entries:**

   - **Informational Logs**:

     ```
     [2024-10-14 18:01:48] [INFO] [LOGGER] Logger have
     been correctly initialized
     ```

   - **Error Logs:**

     ```
     [2024-10-14 18:07:10] [ERROR] [DNS] Failed to reso
     lve hostname: unknown.domain.com
     ```

3. **Filter Log Entries:** If you want to only see some logs, logs are stored by keyword, use `grep` to filter them.

   ```
   grep "\[CONFIG\]" logs/proxy.log
   ```

4.5 Best Practices for Configuration and Usage

To ensure optimal performance, security, and maintainability of the proxy server, adhere to the following best practices:

1. **Regularly Update Filtering Rules:** Periodically review and update the `proxy.rules` file to address emerging threats and evolving organizational policies.
2. **Monitor Logs Continuously:** Implement log rotation and archival strategies to manage log file sizes and retain historical data for analysis.
3. **Limit Exposure:** Bind the proxy server to specific network interfaces to restrict access to trusted networks, reducing the risk of unauthorized usage.
4. **Enable Debugging Judiciously:** Utilize the *DEBUG* mode for troubleshooting but disable it in production environments to minimize performance consumption.

4.6 Documentation

The proxy server possesses a complex architecture, necessitating comprehensive and clear documentation to facilitate understanding and maintenance. To achieve this, we chosen to utilize Doxygen. By configuring a Doxyfile, we create a detailed and structured documentation that reflects the system's design and functionality. To generate the latest documentation after making changes to the source code, simply navigate to the project's root directory and execute:

```
make docs
```

5 Challenges and Solutions

Developing a HTTP proxy server in C presented several technical and collaborative challenges. Managing these effectively was crucial to the project's success. This section outlines the primary challenges encountered and the solutions implemented to overcome them.

5.1 Technical Challenges

1. **Task Organization and Distribution**

   - **Challenge:** Coordinating work between team members to ensure efficient task distribution and progress by complexity. This was essential, especially as the project grew in complexity.
   - **Solution:** We utilized **Kanboard**, a Kanban-based project management tool, to organize and distribute tasks effectively. We used out own instance of Kanboard, running on one of our server. Kanboard allowed us to create visual boards with columns representing different stages of task completion (e.g., Idea, TODO, Done, etc.). This clearly facilitated visibility of each team member's responsibilities, helped prioritize tasks, and ensured that workload was evenly distributed. Look at the Figure 4 page 15

2. **Collaborative Code Development**

   - **Challenge:** Collaborating on code development requires careful management of dependencies and integration. It posed significant challenges. Ensuring that both team members could work simultaneously without causing conflicts was necessary for productivity.
   - **Solution:** We adopted **Git** as our version control system and used **Github** to manage it easily. By using Git and Github, we have ensured that both developers could work concurrently on different parts of the repository. To work on the same code, we also used template function that made this.

3. **Maintaining Code Integrity with Continuous Integration (CI)**

   - **Challenge:** As the project progressed, maintaining a stable and compilable program on the main branch became important to prevent integration issued and ensure that we had a workable program.
   - **Solution:** We discovered and implemented a **Continuous Integration (CI)** pipeline using tools with **Github Actions**. The CI system was configured to automatically build and test the code whenever changes were pushed to the main branch of the repository. This automation ensured that any compilation errors or failing tests were detected immediately.

4. **Memory Management and Leak Prevention**

   - **Challenge:** C language's manual memory management requires meticulous handling to prevent memory leaks and ensure efficient resource utilization. Memory-related issues can lead to unpredictable behavior and degraded performance, which are critical concerns for a network proxy server handling multiple connections. For example, at the beginning of our project, we make some mistake and the fds list overwrite the value of nfds.
   - **Solution:** To tackle memory management challenges, we employed **Valgrind**, a memory profiling and debugging tool. Valgrind helped us identify and diagnose memory leaks, invalid memory accesses, and other related issues by providing detailed reports on memory usage.

5.2 Outcomes

The strategies and tools implemented to overcome the challenges show us significant positive outcomes:

- **Enhanced Productivity:** Effective task management and collaboration tools allow us to complete tasks more efficiently and meet project deadlines. We also improve ourself in project management.
- **Improved Code Quality:** Continuous integration and memory management practices resulted in a stable and high-quality code, reducing the incidence of bugs and performance issues. Theses practices also improve our code style and makes it cleaner.
- **Scalability and Maintainability:** The organized code structure facilitated by Git and GitHub made the project more scalable and easier to maintain.
- **Team Cohesion:** The use of collaborative tools and regular communication fostered a strong sense of teamwork and shared responsibility, contributing to a more cohesive and motivated development team.

6 Conclusion

In this project, we successfully designed and implemented an HTTP proxy server using the C programming language, focusing on scalability, modularity, and efficiency. Through the development process, we gained valuable insights into the workings of proxy servers, network programming, and memory management in C. The use of tools like Git, GitHub, Kanboard for task management, and Valgrind for memory profiling allowed us to maintain a structured development workflow, ensuring that the project was both collaborative and high-quality.

By integrating Continuous Integration (CI), we ensured that the code remained stable and compilable at all times, preventing potential integration issues. This project has strengthened our understanding of system-level programming and equipped us with skills for tackling complex network-related challenges.

Looking forward, we believe this proxy server can be further enhanced with additional features, such as caching and keyword-based content filtering. The flexibility and modular design of the system provide a solid foundation for future improvements, making it a useful tool for various networking and security purposes.

Title of Figures

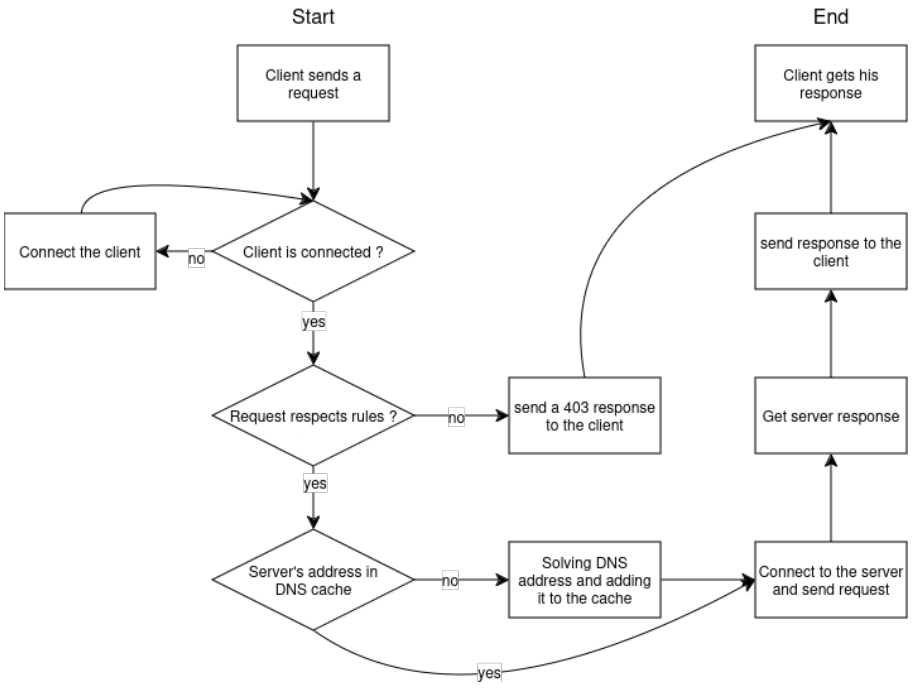

Figure 1: Flowchart of the HTTP Proxy Server's Request Handling Process.

```
1  [2024-10-14 16:50:59] [INFO] [LOGGER] Logger have been correctly initialized
2  [2024-10-14 16:50:59] [INFO] [CONFIG] Rules have been set.
3  [2024-10-14 16:50:59] [INFO] [CONFIG] Regex have been init.
4  [2024-10-14 16:50:59] [INFO] [CONFIG] DNS cache have been init.
5  [2024-10-14 16:50:59] [INFO] [SERVER] server starting....
6  [2024-10-14 16:50:59] [INFO] [SERVER] Socket created (fd: 4)
7  [2024-10-14 16:50:59] [INFO] [SERVER] Server is bind on 127.0.0.1:8081
8  [2024-10-14 16:50:59] [INFO] [SERVER] Server is now listening
9  [2024-10-14 16:50:59] [INFO] [SERVER] Socket open on fd 4
10 [2024-10-14 16:50:59] [INFO] [SERVER] Server polls are ready to run.
```

Figure 2: Initialization logs output example.

Figure 3: Proxy settings to use it with Firefox.

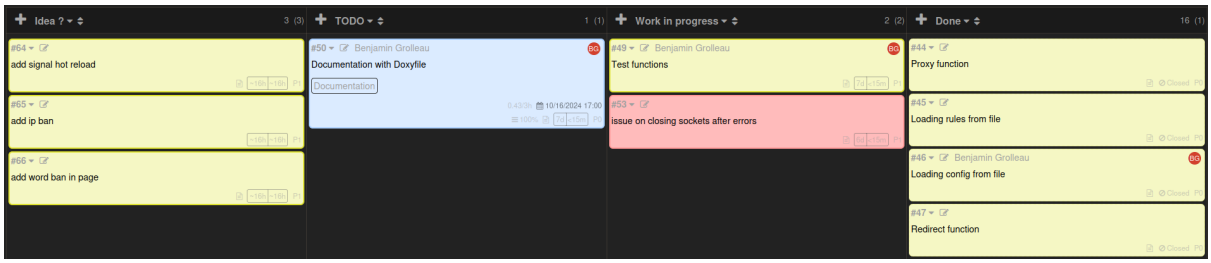

Figure 4: Kanboard screenshot three days before the end of the project.