Nginxxx Final Project Report

Table of Content

Overview

The concept of this project was inspired by popular web servers such as Apache HTTP Server and Nginx (after which the project is named). Under the current web development environment, the concept of web server has been largely abstracted away, whether that is by App Engine offered on Google Cloud, or Lambda function as a service on AWS or Firebase by Google.

These changes may be indicating a new direction to the development workflow web applications have to adapt with. However, just as the spawn of Python and other high-level languages alike does not mean the obsolescence of assembly, the new solutions offered by the big cloud providers do not mean the death of web servers, but simply an abstraction.

The purpose of this project is then to explore part of the infrastructures that had been abstracted away in recent years. The project's main objective is to implement a basic HTTP server. This means implementing the following functionalities:

- Accept HTTP requests

- Parse request method, URI, protocol, header, body
- Respond to requests with reasonable responses (code, header, body)

Above are some of the functionalities a basic HTTP server is expected to be able to process. We will be following RFC2616: Hypertext Transfer Protocol -- HTTP/1.1 guideline to the best of our understanding and ability as we implement our server, to ensure the best compatibility.

This is however a rather vague description, there are a couple more points we would like to achieve to push our project beyond the absolute bare minimum:

- Non-blocking request handling
  - This is a technical hurdle that came up during a discussion with professor Marcelo that we deemed important.
  - Consider the Google homepage. Countless individuals make request to the URL at the same time, yet no user would need another user's request to finish before receiving their response. We intend to achieve this
- Virtual Hosting
  - It is a waste to use an entire machine to only handle HTTP requests to for one domain. If possible, we would like to leverage all the computing power available to handle multiple domains

Besides the measurable objective, we also have things we hope to take away from:

- Improve proficiency with C
- Better understanding of the interaction between C and Linux, such as `unistd, sys/*, fcntl` , etc.
- System programming experiences with socket, processes, file descriptor, etc.
- Deep understanding of the structure of the HTTP protocol
- Ability to filter through complex documentation, and implement standards at low-level

Members and Contributions

Lianting Wang

- Designed and initialized the entire project framework
- Built the config reading system
- Wrote the socket file so that the program can respond to Http connections
- Built a sub-process multi-threaded listening system
- Built the virtual host listening system
- Built a file reading system and completed the GET method.
- Participate in write up

Youzhang (Mark) Sun:

- Implement POST, DELETE, OPTIONS
- Decision making on the behaviour of the majority of the methods
- Implement file system related functionality, including
    - File creation/appending
    - Conflict checking
    - Parent directory creation
- Debug server functionality
- Main contributor to final report write up

Hongxiao Niu

- Implement PUT method handling
- Implement HEAD method
- Complete status code
- Content-Type Recognition
- Participate in write up

Setup and Testing

Getting the code

Run following command

1. `git clone https://github.com/Lianting-Wang/Nginxxx.git`
2. `cd Nginxxx`
3. `git checkout dev`

In another word, checkout the repository at `dev` branch, which has the latest functionality and bug fixes

Config

One can modify the `conf/default.conf` with the following format

```
default.conf File Example

# not required, representing the maximum number of client connections.
# The default value is 1024
max_connections 1024;

# not required, representing the maximum time the client can stay after no operation.
# The default value is 120
keepalive_timeout 120;

server {
    #required, represents the port this server is listening on.
    listen 80

    # not required
    # represents the name of the virtual host
    # If the name is "_" then this port is the default server. The default value is _
    server_name _

    # not required
    # represents the default file to look for at this path if only the path is provided
    # The default value is index.html
    index index.html

    # not required
    # represents the directory where the subserver file is located
    # The default value is . /www/default
    root  ./www/default
}
```

Running

Execute `./run.sh`

If terminal outputs:

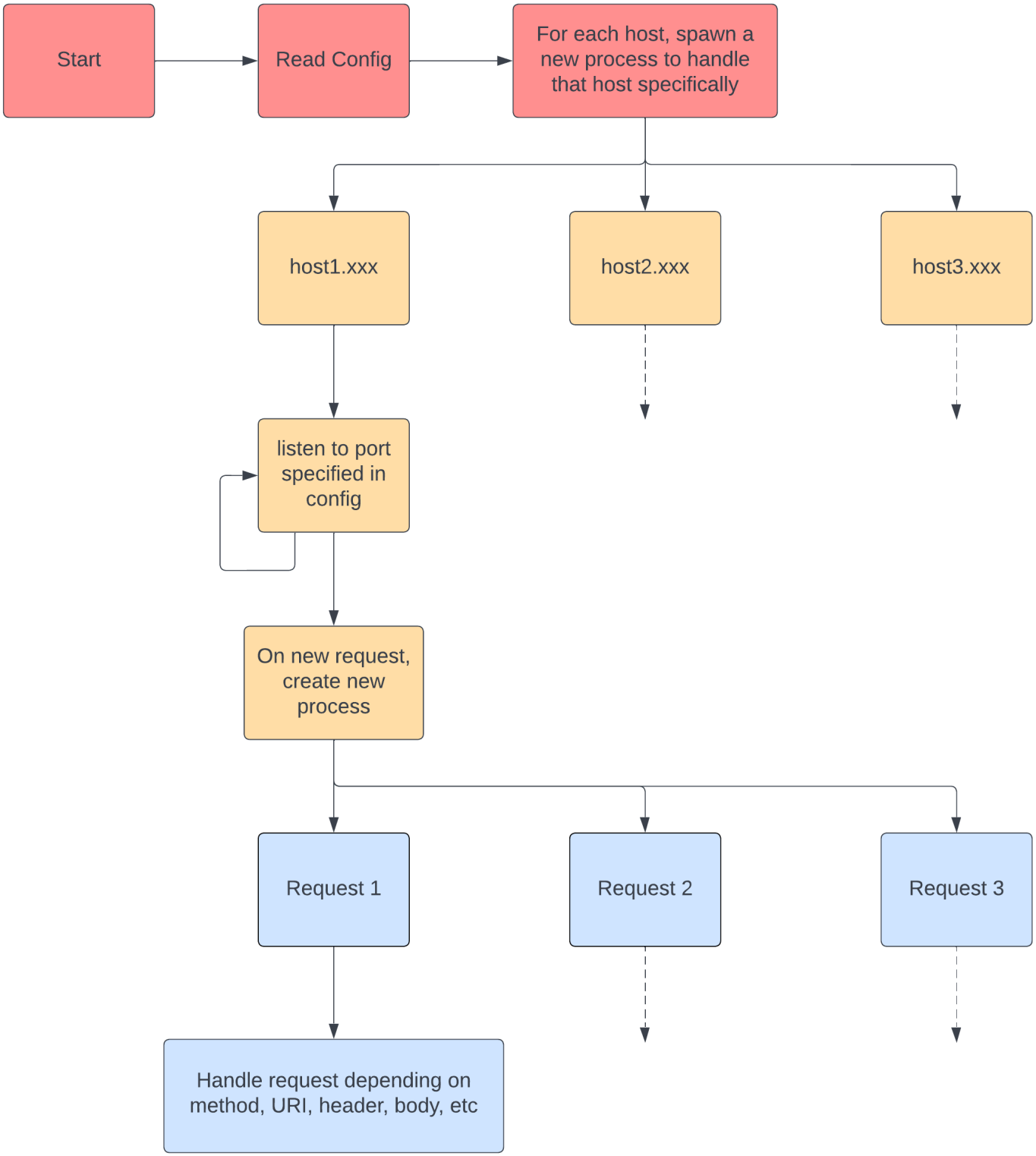
```
I am pid xxxx for port xx
Socket success! sock_fd=x listening...
```

Then the server is running and listening to port/s

Example Test Cases

```
# Test POST
> curl -d 'This is a test.' -X POST http://127.0.0.1/test.html
> curl http://127.0.0.1/test.html
This is a test.

# Test DELETE
> curl -X DELETE http://127.0.0.1/test.html

# Test PUT new location
> curl -d '1: hello; ' -X PUT  http://127.0.0.1/test.txt
> curl http://127.0.0.1/test.txt
1: hello;

# Test PUT append
> curl -d '2: hi; ' -X PUT http://127.0.0.1/test.txt
> curl http://127.0.0.1/test.txt
1: hello; 2: hi;

# Test POST conflict location
> curl -d '3: error' -X POST http://127.0.0.1/test.txt
409 Conflict

# Test HEAD
> curl -I http://127.0.0.1/test.txt
HTTP/1.1 200 OK
Server: Nginxxx
Content-Type: text/plain; charset=utf-8
Connection: keep-alive
Content-Length: 0

# Test HEAD non-existing
> curl -X DELETE http://127.0.0.1/test.txt
> curl -I http://127.0.0.1/test.txt
HTTP/1.1 404 NOT FOUND
Server: Nginxxx
Content-Type: text/html; charset=utf-8
Connection: keep-alive
Content-Length: 0
```

Main Workflow:



Implementation Detail

Element of Code and Purpose

- **`ng_main`**
  - Main entry point of the server, spawn child process for each virtual host
- **`ng_readfile`**
  - For server init
  - Initiate an nginxxx instance that represents the hosts
- **`ng_web_socket`**
  - Abstraction for socket binding, listening, and accepting.
- **`ng_http`**
  - `int handle_request(int client_fd, struct host_instance* hosts, struct host_list* host_lists)` :
    - This function is called on each new request
    - It will parse the request line and header section of the incoming HTTP request, make control flow decision based on method, URI, headers.
    - The function will read up to the body of the request, then call other functions to handle remaining operation
  - `int get_require_line(int client_fd, char * buf, int size)`
    - Helper function to get a line of `xxx CRLF` in the HTTP format
  - `int handle_response(int client_fd, int code, char* path, char method[])`
    - Handles GET and HEAD methods.
    - Given URI and its matching path, attempt to retrieve content at path
    - Exact behaviour listed below
- **`ng_file_util`**
  - Helper functions for file system related operations
  - `int append_content_type(char * buf, char * path)`
    - Abstraction for adding a `Content-Type` header to responding header
  - `int mkdir_parent(char * path)`
    - Create parent directory needed for a file creation

- **`ng_method`**
  - Main implementation of HTTP methods, most functionality are self explanatory, with their behaviour described later on.
  - `int handle_get(int client_fd, char * path)`
  - `int handle_post(int client_fd, char * path, int content_length, char * url_path)`
  - `int handle_delete(int client_fd, char * path)`
  - `int handle_put(int client_fd, char * path, int content_length, char * url_path)`
  - `int handle_options(int client_fd, char * path)`
  - `int backup_404(int client_fd)`
    - A hardcoded 404 error message in case the 404.html file cannot be found
  - `int send_internal_server_err(int client_fd)`
    - Helper function, abstracting sending 500 server error message

Virtual Hosting

The user can edit the `conf/default.conf` file to setup for more hosts. On web server start, the server attempt to read the said file, and with the correct format, would be able to extinguish virtual host, each identified by a `server { ... }` block in the config file.
The server then proceeds to start a child process for each of the hosts identified, each with its own socket to listen for requests meant for it.

Non-blocking Request Handling

On each `accept` operation with socket, the server spawns a child process, and delegates all further handling of the request to the said child. The child would proceed with parsing, processing, and responding to the said request.
Another way to understand the workflow is that on each new request, a new child process is spawned. The child process would handle the request as needed, and exit, freeing up resources.

HTTP Method Implementations

As the project is a basic HTTP server, and mainly a proof of concept, we mapped URI to the file system of the host machine. We append the URI of the request to the root of the host, and consider the operations that can be done on the path

- `GET`
  - If the requested path is a file, the server would respond with `200 OK` and the content of the file
  - If the requested path is a directory, the server would check if there exists an index file in the requested directory. If so, the said index file would be returned along with `200 OK`. Otherwise, a `404 NOT FOUND` would be returned

    - If the requested path doesn't exist in any form, a `404 NOT FOUND` would be returned
- `HEAD`
    - Return a header that is identical to if the request was called with `GET`, except the body is none
- `POST`
    - Quoting the RFC standard:

      > The actual function performed by the POST method is determined by the server and is usually dependent on the Request-URI

    - Thus we decided to have our POST have a simple behaviour like the following:
    - On receiving a request, the server would process the headers looking for header `Content-Length`. If one does not exists, a `411 LENGTH REQUIRED` response would be returned
    - Given the request path, the server would attempt to create a file at the path requested, creating parent directories recursively as needed.
    - During the process of creating parent directories, if a conflict is reached, namely trying to create a directory when a regular file of the same name already exists, a `409 CONFLICT` would be returned.
    - If the requested path endpoint is occupied, as in either there already exists a file or directory of the same name, a `409 CONFLICT` would be returned, and the POST operation dropped
    - If the writing is done successfully, a `201 CREATED` response would be returned along with the URI that can be used to GET the content in the future
- `PUT`
    - If the URL specified in `PUT` is unused, the method behave the same as `POST`
    - If the URL specified is a directory, a `405 METHOD NOT ALLOWED` response would be returned
    - If the URL specified is a file, the content inside the body of the request would be appended to the file at the given URI
- `DELETE`
    - If no file/directory exists at the requested path, a `404 NOT FOUND` would be returned
    - If the requested path is a directory, a `405 METHOD NOT ALLOWED` response would be returned
    - If the requested path is a file, the file would be deleted, and a `204 NO CONTENT` response returned
- `OPTIONS`
    - Given our interpretation of methods, we categorized possible operations as follows:
        - If the requested path does not exist, then the method possible are: `HEAD, POST, OPTIONS`
        - If the requested path is a directory, then the methods possible are: `GET, HEAD, OPTIONS`
        - If the requested path is a file, then the methods possible are:

          `GET, PUT, HEAD, DELETE, OPTIONS`

Analysis and Discussion

Overall, we are proud of the prototype we have completed. Technical challenges such as the virtual host and non-blocking request handling have been a success. We were also able to implement some of the major HTTP methods that are used often in the real world, within the context that we've defined for ourselves.
Due to time constrain, we were not able to implement all HTTP methods, namely `TRACE, CONNECT`. `TRACE` and `CONNECT` are going to be a great challenge if we were to implement it, as it involves other HTTP servers, and sadly we did not have time to put great amount of consideration into the subject, but from reading the standard, we expect them to be rather different from the previous methods (such as SSL tunneling specified in RFC).
Although not proven with statistic, a consequence of our child process approach is that it might be difficult to scale, as a machine could potentially need to manage tens of thousands of child processes. An alternative to our solution could be using `epoll` to handle requests as it comes, and limit them to a small number of processes.

However, we deemed the project a success overall. We've sieved through the complex RFC2616 and were able to implement functions we deem necessary, and were able to interact with our web server to get desired results. The project ends up being a great exercise for us to refine our knowledge of system programming, and exploration of an internet protocol.

The team also discussed areas to explore if this project were to continue. The current server implementation mostly returns content back to the client as it is stored in the server. An upgrade to the server would be to support a sense of "server-side rendering". An example would be the support of PHP rendering: On client request for `/.../xxx.php`, instead of returning the file content, we first evaluate the PHP tag within, this can be done by making a system call and executing a separate rendering executable and piping the output back to our server, and return the resulting HTML.

Another would be the exploration into HTTPS, however, with our lack of experience with security, this would likely require exponentially more time, and thus mostly remained as an afterthought.

Conclusion

Throughout the course, we've gone through the layers of computer networks, and have had hands-on experiences on multiple of them. Though we touched socket back in CSCB09, only now were we able to appreciate the physical, link, network, and transport that were all needed for us to scaffold up to the application layer that our project revolves around.
It was also interesting to see the flexibility application protocol allowed in comparison to IP, ICMP and ARP that we explored in the Simple-Router assignment. The flexibility is double-sided, as it allows for the “implementor” to design what is necessary, however, it also makes the standard less rigid, and more likely

for two implementations to be incompatible (consider all the possible headers an HTTP request could potentially contain). We experienced this during our development, and had to make decisions on what we believed to be important, and focus manpower on those aspects.

The project had been an interesting journey, however, we are also aware that the protocol is on the way out, and replaced completely by either a newer version of HTTP or HTTPS. Still, the project serves as a great exercise, and with our experience, we hope we would rise to the challenge of new protocols when needed.