Due Date: Check course website for due date.

This project must be done in groups of 2 students. Please read the syllabus for instructions, deadlines, penalties, and accommodations regarding group formation and management.

1 Introduction

This assignment introduces you to the principles of internetwork communication using the HTTP and TCP protocols, which form two of the most widely used protocols in today's Internet.

In addition, the assignment will introduce you to existing methods for securely representing claims between parties, using the example of JSON Web Tokens as described in RFC 7519 [2].

Last but not least, it will provide an example of how to implement a concurrent server that can handle multiple clients simultaneously.

2 Functionality

The goal of the project is to build a small personal web server that can serve files, stream MP4 video, and provides a simple token-based authentication API.

The web server should implement persistent connections as per the HTTP/1.1 protocol. HTTP/1.1 is specified in a series of “request for comments” (RFC) standards documents (RFC 7230-7237), though the earlier RFC 2616 [1] provides a shorter read.

You should use code we provide as a base from which to start. To that end, fork the repository at `https://git.cs.vt.edu/cs3984-systems-rust/project-3-peruse`. Be sure to set your fork to be private!

2.1 Serving Files

Your web server should, like a traditional web server, support serving files from a directory (the 'server root') in the server's file system. These files should appear under the root (`/`) URL. For instance, if the URL `/private/secure.html` is visited and the root directory is set to a directory `$DIR` that contains the directory `private`, the content of the file `$DIR/private/secure.html` should be served. You should return appropriate content type headers, based on the served file's suffix. Support at least .html, .js, .css, .mp4, and .svg files; see `/etc/mime.types` for a complete list.

Make sure that you do not accidentally expose other files by ensuring that the request url's path does not contain .. (two adjacent periods), such as `/public/../../../../../etc/passwd`.[1]

[1] Technically, RFC 3986 suggests that you remove those dot segments using an algorithm, but for the purposes of this project we'll assume that the client has applied this algorithm and our server will reject any URLs for which it hasn't.

You should return appropriate error codes for requests to URLs you do not support.

2.2 Authentication

You must, at a minimum, support a single user that will authenticate with a username and password. If the user is authenticated, they should have access to the secure portion of your server, which are all files located under `/private`. Otherwise, such access should be denied.

Your server should implement the entry point `/api/login` as follows:

- When used as the target of a POST request, the body of the request must contain

  `{"username":"<user>","password":"<pass>"}`

  where `<user>` and `<pass>` are placeholders for the name and password of the user. If the password is correct, your server should respond with a JSON object that describes claims that the client can later use to prove it has successfully authenticated. Your server should read the values for the username and password from the environment variables `USER_NAME` and `USER_PASS`, respectively.

  Send (at least) the following claims: (a) `sub` - to describe the subject (the principal as which the server will recognize the bearer of the claim), (b) `iat` - the time at which the claim was issued, in seconds since Jan 1, 1970, and (c) `exp` - the time at which the claim will expire.

  For example, a claim may look like this:

  `{"exp":1700231009,"iat":1700144609,"sub":"user2023"}`

  Returning the claims in the response, however, is not sufficient. The client must also obtain a signature from the server that certifies that the server issued the token (i.e., that the user's password was correct and thus the user has successfully authenticated).

  This signature is obtained in the form of a JSON Web Token, which the server should return as a cookie to the client. You should sign your JWT using HMAC with SHA-256. Your server should use the environment variable `SECRET` to obtain the signing key.

  You should use the serde_json and jsonwebtoken crates for JSON and JWT utilities respectively.

  See the MDN documentation for the format of the `Set-Cookie` header which you must follow. Make sure to set the cookie's path to / so that the client will send the cookie along for all URIs. The name of the cookie should be set to `auth_jwt`.

  You should also set an expiration time for the cookie via the `Max-Age` attribute, which you should set to the expiration time (in seconds) of the token. Your cookie should also be HTTP-only (set the `HttpOnly` attribute) and the `SameSite` attribute should be set to `Lax`.

  If the username/password does not match, your server should return 403 Forbidden.

- When used in a GET request, `/api/login` should return the claims the client presented in its request as a JSON object if the user is authenticated, or an empty object {} if not.

  Be sure to validate tokens before deciding whether the client is authenticated or not; do not accept tokens that have expired or whose signature does not validate.

  You should implement this without storing information about which cookies your server has issued server-side, but rather simply by validating the token the client presents.

  The JSON object shall be returned in the body of the response.

  Your server should implement the entry point `/api/logout` as well. When used in a POST request, your server should ask the client to clear the cookie from its cookie store by returning a `Set-Cookie:` header for the cookie in which the `Max-Age` attribute is set to 0.

  The type of “stateless authentication” can be used to provide a simple, yet scalable form of authentication. Unlike in traditional schemes in which the server must maintain a session store to remember past actions by a client, the presented token contains proof of past authentication, and thus the server can directly proceed in handling the request if it can validate the token. Moreover, this way of securely presenting claims allows authentication servers that are separate from the servers that provide a resource or service: for instance, if you log onto a website via Google or Facebook, their authentication server will present a signed token to you which you can later use to prove to a third server that Google or Facebook successfully authenticated you.

  However, such stateless authentication also has drawbacks: revoking a user’s access can be more difficult since a token, once issued, cannot be taken away. Thus, the server either has to keep revocation lists (in which case a session-like functionality must be implemented), or keep token expiration times short (requiring more frequent reauthentication or a token refresh scheme), or by changing the server’s key (which invalidates all tokens for all users). For this assignment, you do not need to implement revocation.

  We recommend you read the Introduction to JSON Web Tokens tutorial by Auth0. Note that JSON Web Tokens are not the only technology that make uses of cryptographically signed tokens. Others include PASETO and IRON Session.

2.3 Supporting HTML5 Fallback

Modern web applications exploit the History API, which is a feature by which JavaScript code in the client can change the URL that’s displayed in the address bar, making it appear to the user that they have navigated to a new URL when in fact all changes to the page were driven by JavaScript code that was originally loaded. This is also known as “client-side routing,” see React Router for how this is accomplished in the popular React framework.

When a URL that was modified in this way is bookmarked and later retrieved, or if the user refreshes the page while the modified URL is displayed, a request with this URL will be sent to the server, but it does not correspond to an existing server resource.

If the server is aware that this scenario can occur, it can respond with one or more suitable prerendered resources so that the user will not notice which routes existed only on client vs server. Such a resource is called a fallback resource.

This semester, we will implement a suitable fallback policy to host a Svelte application, and you should implement the following algorithm if the `-a` switch is given to your server to enable HTML5 fallback:

- First, check if the requested pathname represents an API endpoint or an existing file[2]; if so, handle it. Else:
- if the requested path is `/`, treat it as a request for `/index.html`.
- if the requested path is `/some/path` and a file `some/path.html` exists relative to your server's root directory, serve it.
- else, treat the request as if `/200.html` had been requested, returning this file if it exists in your server's root directory, or 404 otherwise.

2.4 Streaming MP4

To support MP4 streaming, your server should advertise that it can handle Range requests to transfer only part (a byte range) of a file. You should send an appropriate `Accept-Ranges` header and your server should interpret `Range` headers sent by a client.

To support a basic streaming server, it is sufficient to support only single-range requests such as `Range:  bytes=203232-` or `Range:  bytes=500-700`. Be sure to return an appropriate `Content-Range` header. Browsers will typically close a connection (and create a new one) if the user forwards or rewinds to a different point in the stream.

To let clients learn about which videos are available for streaming, your server should support an entry point `/api/video`. GET requests to this entry point should return a JSON object that is a list of videos that can be served, in the following format:

```
[
  {
    "size": 1659601458,
    "name": "LectureVirtualMemory.mp4"
  },
  {
    "size": 961734828,
    "name": "Boggle.mp4"
  },
  {
    "size": 1312962263,
    "name": "OptimizingLocking.mp4"
```

[2]Only existing files count, not existing directories.

```
  },
  {
    "size": 423958714,
    "name": "DemoFork.mp4"
  }
]
```

Use the `std::path::Path::read_dir` method to retrieve all files in the server's root directory (or a subdirectory, at your choosing), selecting those that carry the suffix `.mp4`. The `std::fs::DirEntry::metadata` method allows retrieving the size of each file.

2.5 Multiple Client Support

For all of the above services, your implementation should support multiple clients simultaneously. This means that it must be able to accept new clients and process HTTP requests even while HTTP transactions with already accepted clients are still in progress. You **must use** a single-process approach, either using multiple threads, or using an event-based approach.[3]

To test that your implementation supports multiple clients correctly, we will connect to your server, then delay the sending of the HTTP request. While your server has accepted one client and is waiting for the first HTTP request by that client, it must be ready to accept and serve additional clients. Your server may impose a reasonable limit on the number of clients it simultaneously serves in this way.

2.6 Robustness

Network servers are designed for long running use. As such, they must be programmed in a manner that is robust, even when individual clients send ill-formed requests, crash, delay responses, or violate the HTTP protocol specification in other ways. *No error incurred while handling one client's request should impede your server's ability to accept and handle future clients.*

2.7 Protocol Independence

The Internet has been undergoing a transition from IPv4 to IPv6 over the last 2.5 decades. To see a current data point, Google publishes current statistics on the number of users that use IPv6 to access Google's services. This transition is spurred by the exhaustion of the IPv4 address space as well as by political mandates.

Since IPv4 addresses can be used to communicate only between IPv4-enabled applications, and since IPv6 addresses can be used to communicate only between IPv6-enabled applications, applications need to be designed to support both protocols and addresses, using whichever is appropriate for a particular connection. For a TCP/UDP server, this requires accepting connections both via IPv6 as well as via IPv4, depending on which versions are

[3]For the purposes of this project, a multi-process approach is not acceptable.

available on a particular system. For a TCP/UDP client, this requires to identify the addresses at which a particular server can be reached, and try them in order. Typically, if a server is reachable via both IPv4 and IPv6, the IPv6 address is tried first, falling back to the IPv4 address if that fails, although it has also been proposed to try both addresses concurrently (see the Happy Eyeballs RFC[3].)

Ensuring protocol independence requires avoiding any dependence on a specific protocol in your code. The Linux kernel provides a convenience feature that provides a simple facility for accepting both IPv6 and IPv4 clients. This so-called dual-bind feature allows a socket bound to an IPv6 socket to accept IPv4 clients. Linux activates this feature if /proc/sys/net/ipv6/bindv6only contains 0. You may assume in your code that dual-bind is turned on. [4]

Therefore, binding your listener socket to the unspecified address for IPv6 (`::`) will also allow it to accept IPv4 clients (similar to binding to `0.0.0.0`).

2.8 Choice of Port Numbers

Port numbers are shared among all processes on a machine. To reduce the potential for conflicts, use a port number that is $10,000$ + last four digits of the student id of a team member.

If a port number is already in use, `TcpListener::bind()` will fail with `EADDRINUSE`. If you weren't using that port number before, someone else might have. Choose a different port number in that case. Otherwise, and more frequently, it may be that the port number is still in use because of your testing. Check that you have killed all processes you may have started on the machine you are working on while testing.

3 Strategy

3.1 Order of Implementation

1. Get the server up and running. Listen on an address for incoming connections which you then handle in order.
2. Parse each incoming request with `parser::parse_request`, producing a `http::Request`.
3. Handle the request with `server::handle_request`, producing a `http::Response`.
4. Send the `Response` back to the caller with `Response::write_sync`.

[4] I should point out, however, that this will make your code Linux-specific; truly portable socket code will need to resort to handling accepts on multiple sockets.

3.2 Tips

Make sure you understand the roles of DNS host names, IP addresses, and port numbers in the context of TCP communication.

Familiarize yourselves with the commands `wget(1)` and `curl(1)` and the specific flags that show you headers and protocol versions. These programs can be extremely helpful in debugging web servers.

Refresh your knowledge of `strace(1)`, which is an essential tool to debug your server's interactions with the outside world. Whenever you are in doubt about your server actually sends or receives, strace it. Use `-s 1024` to avoid cutting off the contents of reads and writes (or recv and send calls). Don't forget `-f` to allow strace to follow spawned threads.

4 Server Modes

Your server should allow changing the method of concurrency when handling requests. The two modes you need to support are *Multithreaded* and *AsyncAwait*.

1. The *Multithreaded* implementation should achieve concurrency by manually spawning threads to handle incoming connections.
2. The *AsyncAwait* implementation should achieve concurrency through Rust's asynchronous programming utilities. You will use the tokio async runtime.

5 Testing

The `tests/` directory contains test data and a script to ensure correctness of your server. Read the help message of `server_unit_test_pserv.py` to learn how to run the tests. It is recommended to start with the *Multithreaded* mode first. Converting the synchronous version to the asynchronous version will require minimal changes to your code.

6 Grading

Grading specific details will be found on the assignment page. The majority of your score will depend on the correctness of your server, which will be determined by the test script given.

Good Luck!

References

[1] Roy Fielding, Jim Gettys, Jeff Mogul, H. Frystyk, L. Masinter, P. Leach, and Tim Berners-Lee. Rfc 2616: Hypertext transfer protocol – http/1.1. http://www.w3.org/-Protocols/rfc2616/rfc2616.html.

[2] M. Jones, J. Bradley, and N. Sakimura. Json web token (jwt), 2015. RFC7519.

[3] D. Schinazi and T. Pauly. Rfc 8305: Happy eyeballs version 2: Better connectivity using concurrency. https://www.rfc-editor.org/rfc/rfc8305.