COP4520 Project Rough Draft

Carlos Sanchez Ruiz Yosha Riley Kariel Sanchez Ruiz Andrew Ballen

Problem 2 Multi thread Server

***Abstract*—We created a functioning multithreaded server in the Rust programming language. The advantage of having a multithreaded server over a single threaded server is the allowance of multiple connections at the same time and being able to serve those connections as well. This will of course open the vulnerability of allowing an infinite number of connections, which would spawn too many threads and either crash the host system or slow it down to a state where it is not functioning. To mitigate this we implemented a thread pool to cap the number of threads the server is capable of spawning. After the clients exceed the amount of threads allotted that client will have to wait until another thread is open. To test we developed a program that will make several requests to the server at a webpage which will sleep before sending data to simulate a costly job. This program simulates user overload and displays the correct functionality as outlined before.**

I. Intro

In an effort to learn more about multithreaded programming and how servers work, we decided to make our own multi-threaded server. We followed a guide in the Rust Book to aid us in our implementation and understanding. In addition to the server we developed a load testing program to help show the multithreaded functionality of the server. This program also shows that having too many users cannot crash the server by overloading it as there is a finite number of threads that are spawned by the server. Our project development followed three main phases, developing a single threaded server, making that server multithreaded, then finally adding a thread pool to limit the number simultaneous connections and prevent crashing.

II. Problem Statement

We have utilized multi-threaded servers provided by third party companies for web application projects. However, we have little experience in designing them ourselves and, in particular, how to design one in Rust. In addition, we would like to see if we can find some improvements to prevent crashes due to overloading the thread capacity of the server, and to design an algorithm which tests that capability.

III. Related work and Discussion

What is a server?

A server is a software that is able to connect to the internet and accept requests made over a network. People describe it as a phone but for the internet. Just like in a phone, in a server you are able to play games, send messages, record videos, store data and stream. However to be more specific a server is a computer that's running meaning once the computer stops running there isn't a server anymore. While the server is online it is listening for request and passing that information to the person that made that request. Now to really visualize the server you have to think of it as like a web browser you would put the address of the server which could be "www.parallelfake.net" and you will end up at the server and that will initiate a request of the server. The server will give your web browser information on where to send the request and what to show based on that information. Furthermore the device that receives a response from the server is refer to as a client.

What are servers used for?

Servers are used for managing network resources. For example, a user may want to set up a server that sends/receives e-mails, manages print jobs, host a website or host's a game like stated before. Servers are actually very good at performing intense calculations. Due to this reason some servers are committed specifically to a certain task as it will be dedicated to a certain calculation and they are called dedicated servers. However, in today's day many servers are shared servers that take on the responsibility of communication between computers this could be in e-mails, DNS and etc.

Why are servers always on?

Servers are always on due to their services being constantly required, most servers are never turned off only for maintenance. I will give two examples. The first example would be a game you would want the servers to be up as long as possible since this means that there will be players playing the game making you more money however updates are required to any game. Therefore, it will need maintenance done so a notification is sent to all players when the servers will be turned off and when they will be relaunch so the players can hop back on and make the company money. In another case we have the bank website servers that need to be up even longer due to their importance in our economy but website servers still need maintenance which is why they also send a notification to their users when they will be turned off and back on this way people don't panic. If a server where to fail/turned off unintentionally without a notification it could cause the users and company many problems which is why most of the time they are on. To be able to achieve this servers are set up to be fault tolerant.

Examples of servers:

1) Application server: A program in a computer in a network that provides business logic for an application program.
2) Cloud server: Servers that run in a cloud computing environment that can be accessed on demand by unlimited amount of users.
3) Database server: Host's at least one database. Clients perform database queries that retrieve data from the database or write data to the database that is on the server.
4) Print server: A server that allows the user to connect to printers in the network. Once connected to the printers the users are able to use them to print or etc.
5) Web server: A program that serves requested HTML pages or files. In this case the web browser would be the client.
6) Mail Server: Application that receives email's from users and is also able to send email's to other users
7) Game server: A server that host's a program that is a game. It allows users to connect to a platform that has other connected user's if its a multiplayer game other wise it would be a platform with one user and will save the data of that user.

How do computers connect to a server

If we were to use a local network, the server connects to a switch/router that all other computers on the network are using. Once the server has established a connection any other computer is now able to connect to that server since it is "on". For example, If it's a web server a computer can connect to it and see the landing page. Now an internet server works kind of the same way but in a larger scale. The internet server would have to acquired a IP address by a web host or InterNIC. The users would connect to the server by typing the domain name of the server, which is registered by a domain name registrar. When the user is able to establish a connection to the server, the domain name is translated automatically to the server's IP address by a DNS resolver. This technique is easier than typing the IP address of the website as the users will remember the name of the website easier than an IP address. Furthermore, by using a name the operator of the server is able to change the IP address of the server if needed and they can keep the name.

Where are servers stored

Servers are stored in a closet or glass house for big companies. These servers are maintained and people make sure they are running properly and the machinery is up to date so it doesn't turned off. However, for remote servers they are located in a data center. This guy they are able to be managed and configured remotely by the operator of the server.

What can be a server

Any computer can be a server as long as it has the right software. However some conditions apply to any device that would like to be a server.

1) The server device must always be running to stay accessible
2) If the device is chosen as a server its resources are taken away. This refers to the processing and bandwidth
3) The server device would need a tighter or better security as now it is open to more attacks due to it being on at all times. This can be ddos, hackers and etc.
4) If the service of the server becomes really popular you might have to make more servers or limit the amount of users.

Single threaded server

A single threaded server is a server that consists of:

1) A single thread
2) One transport
3) One MRemote Dispatcher
4) A loop that reads in request

Advantages of single threaded server

1) Doesn't require many resources
2) Easy to debug due to there only being one thread
3) Easier to code and start running

Disadvantages of single threaded server

1) Can only process one request at a time
2) Less secure than a multithreaded server
3) Prone to crash if it becomes a high-traffic server

Single threaded servers are easy to understand and implement which is usually how beginners first make a server. However, single threaded servers can only process one request at a time. If the request were to be block for any reason any other request has to wait for that request to be done so it is able to access the server which is basically like the server being "offline" for a user even if it's one minute that is one minute where the user doesn't have access to the server. This is exactly what you want to avoid from your server which is where multithreaded servers come in.

Multithreaded server

A multithreaded web server is a server that unlike a single threaded web server is able to take multiple request simultaneously. This is a more efficient way of handling web request as it allows many users to have a server that is "on". This server is usually use in servers that are of high-traffic as it won't slow down or crash.

Advantages of multithreaded server

1) Can process multiple request simultaneously
2) Can handle spikes without crashing
3) Are more secure due to being able to handle more request

Disadvantages

1) Require more resources than a single threaded server to be able to make multiple threads
2) Difficult to debug due to multiple threads running simultaneously
3) Harder to code

Multithreaded web servers works wonders when there are multiple requests to a server since they will be able to handle all of them. However, even though it will be more efficient, reliable, and secure than a single threaded server it requires more resources and a deeper understanding of threads to be able to debug. Therefore, depending on the purpose of your situation you can choose either or.

Thread pool

A thread pool is a pool of threads that can be reused to perform task so that each thread can do more than one task. Instead of making a new thread per task the task can be passed to a thread pool that has been initialized an amount of threads. As soon as a thread is idle a task will be given to that thread so it can execute the task. These task are basically inserted into a blocking queue which the threads in the pool would dequeue from and once they are done with the task they will return to the pool to get ready for it's next task.

Other load balancing techniques and possible improvements

We decided to limit the number of simultaneous connections as it is a simple and easy to test load balancing technique that also usually does not affect users unless the server is lacking in capacity or there is a malicious actor. However there are many improvements that can be made to further secure the uptime of the server and overall stability for the clients. The leading alternative is rate limiting, this is where you record each unique IP that makes requests, then limit the number of requests they can make in a period of time. This prevents malicious actors or power users from hogging too many server resources. The weakness is that there still can be overload from too many unique users, so the optimal solution is to have a combination of rate limiting and a user cap. Rate limits are easily circumvented by crafty users by swapping accounts or IP when they get a 429 error instead of a 200 status code, therefore you need a banning function that will ban users and IPs that hit the rate limit too often

Advantages of a thread pool

1) Improved performance: By reusing threads in the pool it reduces overhead since you won't have to create or destroy threads.
2) Management of resources: By putting a limit of threads you are able to manage the resources of your application since it won't become overwhelmed by too many threads
3) Control over execution: Thread pools allow control of how many task you will like to execute
4) Enhances scalability: By reusing threads it allows the application to manage a large number of request without running into resource problems

```
let listener: TcpListener = TcpListener::bind(addr: "127.0.0.1:7878").unwrap();
```

Fig. 1. TCP listener

```
let listener: TcpListener = TcpListener::bind(addr: "127.0.0.1:7878").unwrap();

for stream: Result<TcpStream, Error> in listener.incoming() {
    let h: JoinHandle<()> = thread::spawn(move || {
        let stream: TcpStream = stream.unwrap();
        //handle_connection(stream);
    });
}
```

Fig. 2. Reading Request

Disadvantages of thread pool

1) No stable identity given to the thread
2) No control over the priority and state of the thread you are working with

IV. Technique

The first technique we started with to build our web server was to make a single threaded server to understand how to start a server. First our code had to listen to a TCP connection. *(Fig 1)* This code will listen at the local address of 127.0.0.1:7878 for any incoming TCP streams. The part before the colon is the IP address and the part after the colon would be the port. We selected this port because HTTP isn't normally accepted in it and this will result in no conflict with other web servers and 7878 is rust typed on a telephone. Now we would like to read the request from the listener and we do that with this code.*(Fig 2)* This code will listen to any request from the server by having a for loop that will create a stream per listener. The stream will represent an open connection between the client and the server. This connection is basically the name for the full requests and respond process in which the client connects to the server, the server will then give it a response and then the server will close that connection. So basically this for loop will process each connection and make streams for us to handle. We will handle this streams by using unwrap to stop our program if the stream runs into any issues and we have an output for that that it will print to the client. Now we want to see what we are actually reading from this request. This is why we have the print statement. We make a handle connection function that makes a buf reader for us that makes a reference to stream. The request variable will collect the lines of the request the browser send to our server. Furthermore, we indicate we want them in a vector by having Vec¡¿ and we have the unwrap again to handle any panics.*(Fig 3)*

Now we want to see what we are actually reading from this request. This is why we have the print statement. We make a handle connection function that makes a buff reader for us that makes a reference to stream. The request variable will collect the lines of the request the browser send to our server. Furthermore, we indicate we want them in a vector by having Vec¡¿ and we have the unwrap again to handle any panics. *(Fig 4)* This is the rest of the code in the single threaded server. This will write responses to the client based on what they

```rust
fn main() {
    let listener: TcpListener = TcpListener::bind(addr: "127.0.0.1:7878").unwrap();

    for stream: Result<TcpStream, Error> in listener.incoming() {
        let h: JoinHandle<()> = thread::spawn(move || {
            let stream: TcpStream = stream.unwrap();
            handle_connection(stream);
        });
    }
}


fn handle_connection(mut stream: TcpStream) {
    let buffered_reader: BufReader<&mut TcpStream> = BufReader::new(inner: &mut stream);
    let request: String = buffered_reader.lines().next().unwrap().unwrap();

    let request: Vec <_> = buffered_reader BufReader<&mut TcpStream>
        .lines() Lines<BufReader<&mut TcpStream>>
        .map(|result: Result<String, Error>| result.unwrap()) impl Iterator<Item = String>
        .take_while(|line: &String| !line.is_empty()) impl Iterator<Item = String>
        .collect();

        println!("Request: {:#?}", request);
```

Fig. 3. Request(stream) handling

```rust
if request == "GET / HTTP/1.1" {
    let status: &str = "HTTP/1.1 200 OK";
    let content: String = fs::read_to_string(path: "hello.html").unwrap();
    let length: usize = content.len();

    let response: String =
        format!("{status}\r\nContent-Length: {length}\r\n\r\n{content}");



    stream.write_all(buf: response.as_bytes()).unwrap();
} else  if request == "GET /sleep HTTP/1.1" {
    thread::sleep(dur: Duration::from_secs(8));

    let status: &str = "HTTP/1.1 200 OK";
    let content: String = fs::read_to_string(path: "hello.html").unwrap();
    let length: usize = content.len();

    let response: String =
        format!("{status}\r\nContent-Length: {length}\r\n\r\n{content}");

    stream.write_all(buf: response.as_bytes()).unwrap();

} else {
    let status: &str = "HTTP/1.1 404 NOT FOUND";
    let content: String = fs::read_to_string(path: "404.html").unwrap();
    let length: usize = content.len();

    let response: String =
        format!("{status}\r\nContent-Length: {length}\r\n\r\n{content}");

    stream.write_all(buf: response.as_bytes()).unwrap();
}
```

Fig. 4. Request(stream) handling

enter. Lets talk about each response first. The first response is an OK response if they have made a connection to our web server we make sure to give them the hello.html response.*(Fig 4)* and *(Fig 5)* Now the second response we have is a sleep which will print the same thing as the normal OK response but we will have it sleep for a certain period of time which we used to make sure our program could handle our max amount of threads for the web browser and didn't crash. If the client is not able to establish a connection to the server they will get the error response which will give them the error.html.

Now we will like to make it multithread so the clients don't have to wait for other clients. *(Fig 6)* So what we did was that we use a thread pool to spawn threads per each

```html
<!doctype html>
<style>
    body{
        background-image: url('background.png');
        font-family: 'Comic Sans MS', 'Chalkboard SE', 'Comic Neue', sans-serif;
        color: white;
        text-align: center;
    }
    h1{
        font-family: 'Comic Sans MS', 'Chalkboard SE', 'Comic Neue', sans-serif;
        color: red;
        font-size: x-largest;
    }
</style>
<html>
<head>
<title>Welcome to our page!</title>
<meta name="description" content="Our first page">
<meta name="keywords" content="html tutorial template">
</head>
<body>
    <div>
        <h1>Welcome to our page!</h1>
        <IMG src="https://web.archive.org/web/20091027090032/http://geocities.com/akachimchim/FIIMSTF/Bloodbar.gif">
    </div>
<br><br>
    <div>
        <p>
            Sick Gifs of cool stuff that we are about<br>
            <IMG src='https://web.archive.org/web/20091019160432/http://www.geocities.com/imbeharry/index/money.gif'>
                <br>
            <IMG src='https://web.archive.org/web/20091027124100/http://www.geocities.com/myfreehangout/internetmoney.gif'>
                <br>
            <IMG src='https://web.archive.org/web/20091018202535/http://www.geocities.com/letsdiscussjudging/dog-wagger.gif'>
                <br>
            <IMG src='https://web.archive.org/web/20090805082727/http://www.geocities.com/megamanta/moneybanners/mbanner2.gif'>
                <br>
            Heck Yea.<br>
            <IMG src='https://web.archive.org/web/20090904032715/http://geocities.com/sophianasa/smiles/bigsunglassessmile.gif'>
            </p>

    </div>
</body>
</html>
```

Fig. 5. Single threaded server finished

Fig. 6. What server will show

```
let pool: ThreadPool =   ThreadPool::new(size: 8);


for stream: Result<TcpStream, Error> in listener.incoming() {

    pool.execute(move || {
        let stream: TcpStream = stream.unwrap();
        handle_connection(stream);
    });
}
```

Fig. 7. Threadpool that will change our server to multithread

client. This thread pools will allow us to concurrently establish connections with the clients. The idea behind the usage of a thread pool here is to basically have a fixed number of threads to make sure we are able to handle a specific amount and if we can handle some overload.

V. Evaluation

We developed a program to simulate a user load, this helps display and test the functionality of our server and chosen load balancing technique. The program uses a rust library called Tokio which is a basic request library that is capable of multithreaded requests. We then use Tokio to make several requests to our server on a "sleep" end point. The sleep end point will cause the thread to sleep for a few seconds before sending the response, this simulates a taxing operation. After running the program, attempting to connect normally through a web browser will wait until there is a free thread. There is no slowdown to the users currently being served and it prevents too many users from attempting to connect at the same time and overloading or crashing the server. Since all functionality works as intended, the project is a success our implementation, while rudimentary, is a valid multithreaded server.

VI. Conclusion

We described the purpose and structure of servers and then built both a single threaded server and a multi threaded server. We also showed why multi threading is a necessary part of most functional servers. Additionally, we showed how to make our multi-threaded server safe against crashes through repeated requests, and showed our testing code for making those repeated requests. Future projects we could work on is to further our knowledge on protection against repeat requests to not only not crash, but also to refuse to connect at all to urls that attempt connections too many times in small time intervals, as well as an algorithm to detect attempted DDOS attacks and how to mitigate their effects on service.