# Rust for Network Servers

Simon Pannek
*Technical University of Munich*
Munich, Germany
simon.pannek@tum.de

May 14, 2021

***Abstract*—Software running on network servers is a common target of malicious attacks trying to cause outages or leakage of user data. Oftentimes memory unsafety caused by human error when developing the application can be the reason for such issues to occur in the first place.**

**Therefore this paper focuses on the benefits of using Rust when deploying a backend for a service and how basic communication using the TCP and UDP protocols can look like, both using synchronous and asynchronous implementations as well as taking a look at the deployment of Rust on the application layer.**

***Index Terms*—rust, computer network, tcp communication, udp communication, asynchronous programming**

## I. Introduction

Many modern devices used today are using the internet to communicate with each other. This concept of connecting to each other dates back to the 1960s when the protocols for one of the first computer networks, the ARPANET, were developed. It allowed large existing host computers with different configurations to communicate with each other, even allowing indirect package transmission by using the hosts as a bridge between two not directly connected hosts [1]. Since then new protocols have been introduced, additional security, for example through encryption was added and a lot of these changes were standardized in order to let devices produced by diverse manufacturers seamlessly work together.

One advantage of standardization is that almost every common used programming language supports those frequently deployed networking protocols, whether natively in its standard library or through simple imports of third party libraries. All those options raise the question which language to use for own application purposes. There is no right answer to this question as it depends on the particular use case, the environment where it has to run, time and cost of development and even more. This paper goes into detail what advantages the programming language Rust has to offer when it comes to running it as a backend of a network application and showcases simple code examples about communication using the protocols TCP and UDP.

Rust is an open-source programming language developed by Mozilla introducing memory-safety without a runtime or garbage collection. Its standard library provides networking primitives for basic TPC/UDP communication, allowing socket communication and error handeling out of the box. [2]

There are usually different approaches when getting to picking a language for networking, especially if you directly want to work with a transport protocol and a custom application protocol on top of it.

As one option you can use a scripting language like Python or JavaScript. With such languages you are able to easily import libraries to give you a basic networking API and they offer a simple syntax allowing fast prototyping. As those languages usually are interpreted, they often perform much slower compared to executing compiled binary files as the instructions first have to get translated by the interpreter before being run.

If speed and scalability is an important factor (as it is most of the time), a compiled language can be a good choice. This is the reason why many networking libraries are written in C or C++. Those two languages do not guarantee memory safety. If a programmer makes a mistake, the compiler does not check for undefined behavior and just falls back to some default action and can lead to corruption of memory.

This can be a problem, in particular when working with network applications as you often have to read and write to buffers concurrently. For example if incoming information is just written to a buffer with no checks for the buffer size, an attacker might be able to overwrite data in the memory and change the flow of the program. This is called a buffer overflow attack and can lead to an attacker being able to control the host running the network application. [3]

Another problem with C and C++ is thread safety. Parallelization and preventing race conditions is completly up to the programmer. When sharing data between multiple threads without proper synchronization, many errors will only occur when in production with a lot of concurrent access and a higher load on the server.

Rust on the other hand is also a compiled language, but the compiler guarantees for memory- and thread-safety. This means it profits from the fast speed compiled languages have and also prevents most of the vulnerabilities mentioned above from happening, thanks to Rust rejecting to compile code which could introduce such issues.

Networking operations are always prune to failure, because the success of such an operation depends on outside factors like other hosts and a stable network connection. Rust can not stop these errors from happening in advance as it has to rely on system calls of the operating system, instead it wraps the returned information of such unsafe calls into enums of the type `std::result::Result`. This allows the use of pattern matching to check if an operation has failed and enables easy ways to handle this failure.

Another selling point of Rust is its well integrated package manager cargo, making it possible to easily import third party

libraries called crates, keeping them up to date, shipping and integrating them into an own application.

To see, how those benefits apply in basic TCP and UDP communication, chapter 2 will first give a short overview about ownership, borrowing and lifetimes and after that explain the implementations of a TCP client and server as well as an UDP sender and receiver, programmed in Rust using the standard library. Chapter 3 is going to explain, why the implementations of the previous chapter are not always optimal as the standard library uses blocking operations for network calls. As a solution, asynchronous programming using a library called tokio and the principle of futures is going to get introduced. After that an asynchronous server and client code example is shown to explain possible difference and advantages compared to a synchronous and blocking approach. Chapter 5 leaves the transport layer where protocols like TCP and UDP are explicitly used and is going to shift the focus to higher-level communication on the application layer. In the end, everything discussed is going to get summarized and concluded.

## II. Networking in Rust

It was already mentioned before that Rust uses verification at compile time to stop compiling the code if something unsafe could happen at runtime without it explicitly being marked as unsafe. To effeciently check for any of those unsafe memory accesses or possible data races, Rust uses the system of ownership, borrowing and lifetimes. It is very important to be familiar with those concepts before actually starting to implement an own network application, because otherwise many compiler errors will not seem rational.

### A. Ownership, borrowing and lifetimes

In Rust, a part of the code can own a piece of memory. When a function is called with an owned value, the piece of memory gets moved into that function and the code calling it is not allowed to access it after the function call. Sometimes a function does not need complete access to a value. If that is the case it can borrow a reference to the value. This enables the code calling the function to still be able to access the piece of memory afterwards. There are also mutable references in Rust, but it is only allowed to have a single mutable reference to a variable at once.

This can be really useful when working with network applications as this prevents the occurence of data races if a thread tries to read or write to a buffer while another thread is already writing to it.

The compiler of Rust does not check for illegal array accesses, but other than in C the program exits with an error code (this is called a panic) if the program tries to access an index which is out of bounds. This does not prevent buffer overflows to take place in advance, but at least the program stops executing immediately instead of letting it happen and possibly introducing security issues.

Variables usually have a lifetime determined by the code block it was created. If a variable gets out of scope, the corresponding piece of memory is freed automatically. If a value gets moved into another function, its lifetime changes with it. Combined with the system of ownership, this can prevent dangling references, illegal memory accesses and memory leaks from happening at compile time. [4]

### B. TCP communication

The Transmission Control Protocol (TCP) is used for reliable inter-process communication between two systems connected through a network. You can use it to send a continuous stream of octets to another host. In order to check whether the data was received by the other host, it assigns a sequence number to each octet sent and waits for an acknowledgment (ACK) from the receiving user. If no ACK is received in a certain amount of time, the data is sent again. The receiving host can also use the sequence number to eliminate duplicates and order the segments the right way. [5]

The module `std::net` of the Rust standard library already implements this protocol, so you can work with those abstractions independent of what plattform the code is currently running. To demonstrate this in a better way, we are going to implement a small TCP server which accepts connections and prints incoming messages as well as a TCP client, which connects to our server and sends messages from the command line to it. Figure 1 demonstrates how messages sent to a `std::net::TcpStream` can be received and printed for the user to read.

```
let mut reader =
 ↪  BufReader::new(&stream);

loop {
    let mut buf = String::new();

    match reader.read_line(&mut buf) {
        Ok(0) => break,
        Ok(_) => print!("{}", buf),
        Err(e) => {
            eprintln!("{}", e);
            stream.shutdown(Both)?;
            break;
        }
    }
}
```

Fig. 1. TCP client handeling

The variable `stream` has the type `TcpStream` and represents a connection between two hosts. Data is being transmitted when you write it to the stream and the other host can read the received information by reading from the stream on their side.

As networking functions can fail due to multiple reasons (by instance network failure, package loss, insufficent permissions, etc.), those functions usually return an instance of the enum `std::io::Result<T>`. If the call was successful, the returned value corresponds to the returned data, wrapped by

the enum variant `Ok(T)`. If the call has failed, the returned value corresponds to the error value, wrapped by the enum variant `Err(Error)`.

In line 1 a reader of the type `std::io::BufReader` is created. It can be inefficient to read from the stream with small and repeated calls. In this example we want to read whole lines from the stream, so the reader can maintain the data in its in-memory buffer until a whole line was received.

Now the example continues to loop and wait for a new message until the connection is closed. `buf` is a simple, mutable `String` new messages get written into. In line 6 the result of the call `reader.read_line(&mut buf)` is matched.

In case the call was successful but the amount of read bytes is equal to 0 this means that the connection was closed and we can break out of the loop. If at least one byte was read, the received line is written into `buf` and the program enters the second match arm, writing the content of the buffer to the command line using the `print!` macro.

In case reading has failed, the error value is returned and the program enters the third match arm. Here the error message is printed to the error output and the program tries to shutdown the stream using `stream.shutdown(Both)?`. This method takes an enum of the type `std::net::Shutdown` as an argument which decides, how the stream should get closed. `Both` means that both the reading and the writing portions of the stream should be shut down. The `?` operator tries to unwrap the `Result` returned by the function and returns immediately if the result is equal to an `Err`. If the unwrap was successful, the program ignores the result and simply breaks out of the loop.

The example of figure 1 can be used to handle a connection to a single host and print all the received data until the connection is closed. To actually run the code, we still have to initialize the stream in the first place. To make everything simpler, let us assume that the code of figure 1 is now wrapped into a method called `handle_client`, taking the `TcpStream` it handles as an argument. We are now able to call this function in the code of figure 2.

```
1 let listener =
  ↪ TcpListener::bind(ADDRESS)
2     .expect("Bind to address");
3
4 for stream in listener.incoming() {
5     if let Ok(stream) = stream {
6         handle_client(stream)
7             .expect("Handle client");
8     }
9 }
```

Fig. 2. TCP listener

First a listener of the type `std::net::TcpListener` has to be created by binding it to `ADDRESS`. The constant `ADDRESS` has the type `&str` and contains the address and port (e.g. "127.0.0.1:8080") on which the server should listen for new incoming connections. The method `expect` is another way to handle the return type `Result`. It tries to unwrap the `Ok` value if there is any and panics with a message including the argument and the `Err` value formatted as a `String` if it fails.

A `TcpListener` has an `accept` method, which blocks the calling thread until a new TCP connection is found and returns the `TcpStream` and the address of the other host if successful. The method `incoming` returns an iterator over new connections received by the listener. Iterating over it is equivalent to repeatedly calling `accept` on the same listener.

We can now use a for loop to keep listening for a new TCP connection as soon as the contact to the current host is lost. The if statement in line 5 utilizes pattern matching using the keyword `let` to check if the connection was successful. If the returned value of the `TcpListener` is an `Err`, it gets discarded and the listener tries to find another host.

In case a connection was established successfully, our function `handle_client` is called with `stream` as an argument. As the handeling of the client can also result in errors, the `expect` method is used once again to panic if something has failed. An alternative approach would be to check the returned value of `handle_client` and only print the error to the command line. This would result the server to be more stable and not panic completly if an error occurs in handeling the current connection.

Now the only thing we need to actually use our server is a client to connect to. A simple TCP client is implemented in figure 3.

First a connection to the server is established using the function `TcpStream::connect`, which takes `ADDRESS` as an argument. The constant `ADDRESS` is similar to the constant we have used in the server and corresponds to the address and the port of the server we want to connect to, represented as the type `&str`. This time no `TcpListener` is needed, because the other host is already listening for connections and we want to establish a connection explicitly to this host. The returned result of the function is unwrapped using `?` to panic if something went wrong.

After that the program enters a loop and keeps iterating until the connection was closed. In line 4 a mutable `String` called `buf` is created. This variable is used to save the console input of the user and then send it to the server.

In line 6 the method `read_line` of `std::io::Stdin` is used to read the next line from the command line. It takes a mutable reference to the buffer the input should be read into as an argument and returns a `Result` corresponding to whether reading was successful.

If it was, the content of `buf` is trimmed to remove leading and trailing whitespaces and checked whether it matches "exit". If that is the case, the `shutdown` method of the stream is used to close the current connection and the program breaks out of the loop. Otherwise the input string is converted into a byte slice using the `String`-method `as_bytes`. The method `write_all` of the stream takes this slice as an argument and

```rust
let mut stream =
    TcpStream::connect(ADDRESS)?;

loop {
    let mut buf = String::new();

    match stdin().read_line(&mut buf) {
        Ok(_) => {
            if buf.trim() != "exit" {
                stream.shutdown(Both)?;
                break;
            }

            let bytes = buf.as_bytes();

            stream
                .write_all(bytes)
                .expect("Writing");
        }
        Err(e) => {
            eprintln!("{}", e);
            stream.shutdown(Both)?;
            break;
        }
    }
}
```

Fig. 3. TCP client

sends it directly to the server. As this is another method which can fail due to networking issues, it again returns an `Result` which is unwrapped through the `expect` method.

If reading from the command line has failed, the program handles it exactly the same as the server handles read failure from the stream: The error gets printed to the error output and the program tries to shut down the current stream. If that was successful, it breaks out of the loop resulting the program to terminate.

When using TCP it can be really difficult to detect whether the connection to the server was lost, as the client is only sending and not receiving any information. This means if we try to close the connection from the server side, the client will not terminate immediately but usually only after sending a message and not receiving an ACK from the server in time. A way to resolve this problem would be to implement some kind of ping between the client and the server in a certain timeframe to check if the other side is still connected. If the time with no ping received exceeds this timeframe, the connection was probably lost and the stream can be closed.

### C. UDP communication

The User Datagram Protocol (UDP) is less reliable than the Transmission Control Protocol. Differently to TCP, UDP tries to send messages to other programs with a minimal amount of protocol mechanism. This means it offers no protection against package duplication and does not check whether the sent data arrived and if it did in what order. As a consequence you do not have to listen for new hosts and actively establish a connection to another host before sending any data. UDP is commonly used, if you do not need reliable delivery of streams and small amount of package loss does not matter. A typical example for UDP applications would be media streaming. [6]

When trying to communicate over UDP in Rust, you can use the struct `std::net::UdpSocket`. It provides methods to bind to an address, send and receive data. As we do not need a server to listen for new connections, the following example will be split into a sender who sends data and into a receiver who receives those messages. In figure 4 there is an example, how the code on the receiving end could look like.

```rust
let socket = UdpSocket::bind(ADDRESS)?;

loop {
    let mut buf = [0u8; 1500];

    match socket.recv_from(&mut buf) {
        Ok(_) => {
            let msg = from_utf8(&buf)
                .expect("Convert data");

            print!("{}", msg);
        }
        Err(e) => {
            eprintln!("{}", e);
            break;
        }
    }
}
```

Fig. 4. UDP receiver

In the beginning a socket is created by calling the `bind` method and giving the address the socket should listen to as an argument. The result is unwrapped using `?` and then moved into a variable. After that the program keeps looping until an error occured or it is closed explicitly. As there are not connections between two hosts, the socket cannot close when a connection is lost. Another difference to using TCP is that the socket is able to listen to incoming messages of multiple hosts.

When trying to receive an incoming message, we first have to initialize a buffer to save the data of the other host. A `UdpSocket` does not implement the trait `std::io::Read`. This means you cannot create a `BufReader` instance in order to read the data from the socket. Alternatively we have to read the messages as bytes and then parse them into a `String` later. The variable `buf` in this case is a byte slice of the length `1500`. If you know the size of the data beforehand, you can adjust the buffer length according to this, but as we do not know anything about the data we are going to receive, this example uses the size of a typical UDP MTU (Maximum

Transmission Unit). If a message is larger than the buffer, it is going to get cut off.

In line 6 the method `recv_from` is used to read the next arriving package into `buf`. If the call was successful, the program enters the first match arm, tries to convert the byte slice into a `&str` and panics in case of a failure using the `expect` method. After that the message gets printed to the command line. If the call has failed, the program prints the error and breaks out of the loop.

To send data to this receiver, you cannot just use the TCP client implemented before as it uses a different protocol. In figure 5 you can see a small implementation of an UDP sender.

```
1  let socket =
   ↪  UdpSocket::bind("0.0.0.0:0")?;
2
3  loop {
4      let mut buf = String::new();
5
6      match stdin().read_line(&mut buf) {
7          Ok(_) => {
8              if buf.trim() == "exit" {
9                  break;
10             }
11
12             let bytes = buf.as_bytes();
13
14             socket
15                 .send_to(bytes, ADDRESS)
16                 .expect("Sending");
17         }
18         Err(e) => {
19             eprintln!("{}", e);
20             break;
21         }
22     }
23 }
```

Fig. 5. UDP sender

Here the socket is not bound to the address of the host we want to send messages to. Rather the `bind` method is given the static address ”0.0.0.0:0”. This is because of the UDP protocol. As there is no set connection between two hosts, our sender needs an own address and port in order to receive answers from the other host. By giving the `bind` method the static address of ”0.0.0.0:0” as an argument, we implicitly state that the host should use its own ip address and choose any free, available port it is allowed to use. It would also work to set the port explicitly, but this risks choosing a port which is already used by another program.

From this point on the sender is really similar to the TCP client: In a continous loop the user input from the command line is written into `buf`.

If the reading was successful, the input is first trimmed and then checked whether it matches ”exit”. If that is the case, the program simply breaks out of the loop. We do not have to shutdown any stream here, as there is no connection in the first place. Otherwise the input is converted into bytes using the method `as_bytes` and then sent to the other host by calling `send_to` on the socket, giving it the byte slice and the address of the other host as arguments. The result of this method call is again unwrapped using `expect` to panic in case of an error.

## III. Asynchronous networking using Tokio

In the chapter before basic examples for TCP and UDP communication in Rust were shown. In those programs the current thread was blocked when waiting for new connections or incoming messages. This not only results into the program having to wait for a new message to continue executing the code, it also means that you have to create a new thread for each connection, if you want to listen to multiple connections at once. This principle is called sequential programming. In contrast to that there are no blocking functions in asynchronous programming. Instead results are wrapped in so called futures which represent a result of an otherwise blocking function. This result can be unwrapped when it is actually needed instead of waiting for the execution in advance. [7]

### A. The tokio crate

The core library of Rust is kept very small. Instead the package manager cargo allows for easy import of third party packages. This way those extensions are not bound to the release cycle of Rust and also breaking changes will just affect one single package instead of the whole library. For compatibility reasons a project can keep using an older version of a package while still upgrading the standard library. The packages of Rust are called crates.

A crate is similar to a binary or library. You can add the name and the version to the dependencies of a project in order to import it. If you do not want to compile and include a whole library, some crates also offer feature flags deciding which parts of the library should be included.

To introduce asynchronous programming to Rust, you usually want to use a crate which enables async methods. In this chapter the crate ”tokio” is used. As the documentation [8] states, tokio is a runtime for writing reliable network applications. It offers tools for working with asynchronous tasks as well as APIs for typical blocking operations suchs as sockets and filesystem operations.

A task in the context of a tokio program is a non-blocking piece of code. You can use the `tokio::task::spawn` method to schedule the task on the Tokio runtime and use `await` to wait for the returned value. This is a typical behavior in asynchronous programming and you can compare a task to a promise in programming languages like JavaScript.

The tokio runtime consists out of three major parts: An I/O event loop, which handles I/O events and dispatches them to the tasks which are waiting for them, a timer to run tasks after a certain period of time and a scheduler, which actually executes the tasks on the different threads. To enable the tokio

runtime, you can mark the main function with the **async** keyword and add the *#[tokio::main]* macro above.

The scheduler swaps around all tasks which need to be run. This means there can be more tasks than threads whilst still allowing concurrent execution of all tasks. To tell the scheduler that the program is currently waiting for another task to finish before it can continue running, you can call **await** on the method. This effectively stops the execution of the current task, waits for the other task to finish and then unwraps the returned result. Contrary to an execution without the tokio runtime, the scheduler knows when the current task can continue running and uses the otherwise unused CPU time by letting another task run instead.

### B. Programming with futures

Let us now take a look at the small demonstration of futures in figure 6.

```
1  async fn async_function() {
2      println!("Started task1");
3      sleep(Duration::from_secs(5))
4          .await;
5      println!("Finished task1");
6  }
7
8  #[tokio::main]
9  async fn main() {
10     let future1 = async_function();
11
12     let future2 = async {
13         println!("Started task2");
14         sleep(Duration::from_secs(3))
15             .await;
16         println!("Finished task2");
17     };
18
19     join!(future1, future2);
20 }
```

Fig. 6. Future demonstration

The declared types of functions play an important role when working with tokio. This is the reason why the example of figure 6 also includes the entire function definitions.

The main function is marked as **async** and the tokio macro is used to create a runtime for our program to run in. In line 10, the asynchronous function `async_function` is called. The value returned by the function has the type `std::future::Future`, which wraps the value going to be returned after the computation. The `Future` is saved in the variable `future1`. This way the tokio runtime actually decides when to evaluate the function. If `async_function` was not marked as an asynchronous function, the main thread executing the program would enter the function, completely evaluate it and after that continue executing the main function.

The type of `future2` is also a `Future`, the only difference being that it uses an **async** block instead of an extra function to define the asynchronous code.

In the end the macro `join!` is used to wait for both futures to evaluate completely while executing them concurrently.

Both functions first print a short message, that they have started executing, then they sleep a certain amount of time using `tokio::time::sleep` and then print that they have stopped executing. The key part in this is, that the function wrapped by `future1` is called first and sleeps for 5 seconds, the function wrapped by `future2` is called afterwards and only sleep for three.

In a blocking environment it would result the program to take more than eight seconds to execute, because first the returned value of `future1` is evaluated and only after that the value of `future2`. This way, the output would look like this:

```
Started task1
Finished task1
Started task2
Finished task2
```

The only way to avoid the synchronous execution would be to explicitly start two threads and execute both tasks concurrently. If multiple tasks have to run in parallel and include a lot of blocking operations, you usually do not want to create a thread for each task as the creation of threads adds a lot of overhead to the program.

In an asynchronous environment on the other hand, the scheduler stops the execution of the current task and swaps in another task which is waiting to be executed when the program reaches an **await** point. This way, the sleep function does not block the entire program flow and the function wrapped by `future2` finishes first, even if only executing on one thread. The resulting output of the program looks like this:

```
Started task1
Started task2
Finished task2
Finished task1
```

### C. Asynchronous TCP communication

Tokio offers the module `tokio::net`, which contains a TCP and UDP networking API similar to the one of the standard library used in chapter 2. Those functions are marked as **async**, blending in with the rest of the tokio ecosystem.

Let us first use tokio to spawn a new task which handles a connection by reading all incoming messages and printing the output to the command line. The code of figure 7 implements basic asynchronous client handeling.

The current connection is represented as the variable `stream`, which has the type `tokio::net::TcpStream`. The function `tokio::spawn` takes a `Future` as an argument and creates a new asynchronous task. When and how this task gets executed depends on the configuration of the current runtime.

```
tokio::spawn(async move {
    let mut reader =
 ↪  BufReader::new(&mut stream);

    loop {
        let mut buf = String::new();

        match reader
            .read_line(&mut buf).await {
            Ok(0) => break,
            Ok(size) =>
 ↪  print!("{}", buf),
            Err(e) => {
                eprintln!("{}", e);
                stream
                    .shutdown().await?;
                break;
            }
        }
    }

    Ok::<(), Error>(())
});
```

Fig. 7. Asynchronous TCP client handeling

As an argument the function is given an asynchronous code block, which returns a `Future` as seen in figure 6. Here additionaly to the **`async`** keyword, **`move`** is used. This means the asynchronous block will take the ownership of all variables referenced within it. Without this keyword, all variables would be bound to the scope of the code surrounding it. This is especially important as our asynchronous code needs full ownership of the `stream` variable in order to read from it and shut it down in case of an error.

The rest of the code is similar to the synchronous TCP client handeling of figure 1: In line 2 a buffered reader instance is created for more efficient reading from the `TcpStream`. After that the program enters a loop to repeatedly read the input of the connection. In line 5 a mutable `String` named `buf` is created to save the incoming data. Then the method `read_line` of `reader` is called with `buf` as an argument. As the reader instance has the type `tokio::io::BufReader` in this asynchronous implementation, `read_line` does not return a simple `Result` but rather a `Future` which wraps around the returned value. This means we can use an **`await`** point to tell the scheduler it can stop running the current program until new data has arrived.

If reading was successful and no bytes were read, it means that the connection was closed and we can break out of the loop. If at least one byte was read the program enters the second match arm and the read data gets printed to the command line.

If reading has failed, the program enters the third match arm and prints the error to the command line. After that it tries to shutdown the stream. Here the `shutdown` method writes all buffered data to the stream and shuts down the write instance of the stream using a system call. Other than the `shutdown` method of `std::io::TcpStream`, it does not take an enum as an argument to control how the stream should be shut down. The method also returns an `Result` wrapped by a `Future`. First the future gets unwrapped by calling **`await`** on it. Afterwards the result is unwrapped using the `?` operator and the program breaks out of the loop.

In the end of the task we have to explicitly specify the type of the result returned. The reason for this is the use of the `?` operator in line 14 to unwrap the returned value of the `shutdown` method. As **`async`** blocks do not have an explicitly specified return type, the compiler sometimes fails to infer the error type of the async block. This will trigger the compiler error `type annotations needed`. By specifying the exact type of the `Ok` enum variant, the compiler is able to infer the return type and we are able to use the `?` operator. Another way to handle this would be to use the `expect` method instead or create an extra asynchronous function instead of using **`async`** blocks. [9]

As in chapter 2 we can now wrap our code from figure 7 into a function called `handle_client` to be able to call it in figure 8. Next we are going to open the TCP streams and call the `handle_client` with the stream as an argument. It was already mentioned before that having TCP connections to multiple clients would require a thread for each connection. Tokio solves this problem as the event loop manages and listens to every connection. Instead of creating a thread for each connection it is only required to spawn a task for each TCP stream. That is why the code example of figure 8 supports connecting to multiple clients.

```
let listener =
 ↪  TcpListener::bind(ADDRESS).await?;

loop {
    let (stream, _) =
 ↪  listener.accept().await?;

    handle_client(stream);
}
```

Fig. 8. Asynchronous TCP listener

First a `tokio::net::TcpListener` is created by binding it to the address saved in the **`&str`** constant `ADDRESS`. This value corresponds to the ip address and port the server should listen to. The future result is then unwrapped using **`await`** as well as `?` and saved to the variable `listener`. As the `TcpListener` of tokio does not implement the `incoming` method, we have to call the `accept` method of the listener ourselves. This method again returns a tupel including the `TcpStream` and the address of the new connection, wrapped by a `Result` to handle errors and a `Future` to make the waiting operation non-blocking. After that the re-

ceived stream can be given to our `handle_client` function, which spawns the new task handeling the client. As the client handeling runs in an extra task, the program can continue to run and accept new connections without having to wait for the current one to disconnect.

To connect to this server one could just use the TCP client of figure 3, as both servers use the same protocol and can work together interchangeably. To showcase how the tokio library affects a client implementation, there is a smaller demonstration of an asynchronous TCP client in figure 9.

```
1  let mut stream =
   ↪  TcpStream::connect(ADDRESS).await?;
2
3  let mut reader =
   ↪  BufReader::new(stdin());
4
5  loop {
6      let mut buf = String::new();
7
8      match reader
9          .read_line(&mut buf).await {
10         Ok(_) => {
11             let bytes = buf.as_bytes();
12
13             stream
14                 .write_all(bytes)
15                 .await?;
16         }
17         Err(e) => {
18             eprintln!("{}", e);
19             stream.shutdown().await?;
20             break;
21         }
22     }
23 }
```

Fig. 9. Asynchronous TCP client

In the beginning a `tokio::net::TcpStream` instance is created by connecting to `ADDRESS`, a constant in which the address of the server is saved as a `&str`. Other than the synchronous implementation, we are not going to read from the command line directly but rather wrap it using a `tokio::io::BufReader` instance to be able to read from it without blocking the task.

The loop for reading messages on the command line and sending them to the server is also very similar to the synchronous implementation of figure 3: In line 6 a new `String` called `buf` is created to save the input of the user. Then the `read_line` method with `buf` as an argument is invoked. As this is an asynchronous method, we can use the **`await`** keyword to give up the computing resources until new input was written into the buffer. In case new input was written to the buffer and the scheduler continues to execute the task, the return value of `read_line` is matched.

If reading was successful, the program enters the first match arm. In figure 3 this match arm included a check whether the read string is equal to ”exit” to offer a way to close the connection other than just closing it forcefully. This part is removed in figure 9 as it works the same way. The read input is converted to bytes using `as_bytes` and saved in the variable `bytes`. After that the byte slice is given as an argument to the `write_all` method of `stream`. The returned `Future` is again unwrapped using **`await`** and the contained `Result` unwrapped using `?`.

If reading has failed, the error is printed to the error output, the program tries to shutdown the stream using its `shutdown` method, unwrap the `Future` and `Result` using **`await`** and `?` and then break out of the loop.

When running the TCP client, the asynchronous client does not differ as much from the synchronous client as the server implementations differ from each other. The server implementation can use the scheduling from tokio to handle multiple connections at once. The client on the otherhand is idle when it is waiting for input on the command line anyway, because it does not have any additional tasks. If you expanded the implementation to a complete chat client which can both send and receive message, a second task would be needed to listen for new data on the connection. Here an asynchronous environment would be of more use.

### D. Asynchronous UDP communication

When it comes to asynchronous UDP communication, tokio offers the struct `tokio::net::UdpSocket` which works similar to the `UdpSocket` of the standard library used in figure 4 and 5 for the UDP sender and receiver.

Porting the synchronous UDP implementation to an asynchronous one works the same as we have done for the TCP client and server in the previous subchapter: Blocking operations are replaced by non-blocking operations returning futures. If the current task has to wait for a future to evaluate, **`async`** can be used.

As UDP does not implement actual connections between the sender and the receiver and only binds to an address, the behavior of the asynchronous UDP implementation will also not differ as much from the synchronous example, similar to figure 9. At least this is the case if the sender and receiver are as strictly seperated from each other as they are in the synchronous UDP example. Otherwise the struct `std::sync::Arc` could be used to share the ownership of the `UdpSocket` to have a sending and receiving task in the same client. This is possible, because both the `send_to` and `recv_from` methods of the socket do not require a mutable reference to the socket. If this was not the case, you could not share one socket for two tasks as there can only be one mutable reference to a certain piece of data in Rust.

## IV. Application Layer Communication

In chapter 2 and 3, transport layer protocols were used to explicitly send text messages formatted as byte slices from a client to a server. This may be enough for simple applications

like a chat server, but in many cases another abstraction on top of that is needed to send special data like structs over a connection. This is where protocols belonging to the application layer come into place.

The following chapter will give a small insight into the principle how the asynchronous communication using tokio can also be used together with higher level protocols. In particular the Hypertext Transfer Protocol (HTTP), a protocol used for data transmission for the World Wide Web is going to be utilized in order to send a `struct` from a client to the server and receive a response as a confirmation. HTTP is usually based on the TCP protocol but can also be implemented using other transport protocols, which offer a confirmation, if sending data was successful.

### A. Serialization

Before sending anything over HTTP, first a piece of data to actually send is needed. For this we are going to introduce the `struct` called `Data` in figure 10, which will both be implemented in the HTTP client and server.

```
#[derive(Serialize, Deserialize, Debug)]
struct Data {
    number: u32,
    boolean: bool,
}
```

Fig. 10. Data struct

Even though we do not want to use raw TCP communication to send our packages, HTTP still is a text based protocol. This means before being able to send the struct from the client to the server, we have to transform it from a `struct` to a `String` and back, a principle called serialization and deserialization.

This is why we are going to use a crate called serde. Serde allows us to let our `struct` derive `Serialize` and `Deserialize`. This way the compiler is capable of providing a basic implementation to convert `Data` to a JSON `String` and the other way around. JSON stands for JavaScript Object Notation and is just a standard file format typically used for exactly this purpose.

The additional deriving of `Debug` implements the trait `std::fmt::Debug`. This serves the purpose of printing the struct in a human readable format, typically used for debugging purposes.

### B. HTTP communication

Now that it is possible to convert the struct into a format which is sendable over HTTP, next a server is needed to accept incoming requests. There are many crates, which enable you to accept and handle HTTP requests in Rust. For our HTTP server implementation, we are going to use a crate called warp.

Warp is simple web server framework, which builds on top of the crate hyper, a HTTP implementation for Rust [10]. Warp makes use of asynchronous programming and seamlessly works together with tokio. An important concept of warp are the so called filters: `warp::Filter` is a struct, which extracts data from a request, handles the data and sends a response back to the sender of the request. The `Filter` implements two important methods:

First the method `and`, which takes another `Filter` as an argument and returns them as a new, combined `Filter`. This way you can chain together multiple `Filter` instances and filter requests more precisely.

Secondly the method `map`, which takes a function as an argument. This function receives the extracted data from the request and can map it to another value. This value in return gets sent back to the client sending the request.

`Filter` also implements more methods useful for rejecting certain requests or customizing filters to take different sets of arguments in a request, but those methods will not be needed for our example.

In figure 11 you can find a small example implementation of a web server written using warp.

```
let filter = warp::path!("data")
    .and(warp::post())
    .and(warp::body::json())
    .map(|data: Data|
    ↪ format!("Received: {:?}", data));

warp::serve(filter).run(ADDRESS).await;
```

Fig. 11. Warp HTTP server

First a `Filter`, which matches a path of a possible request sent to the server, has to be created. Here the macro `warp::path!` can be used which, returns a `Filter` matching the exact path segment. For our example we are going to send the data to the path ”/data”.

Next we have to specify, how the request exactly looks like. This enables us to automatically reject invalid requests. In this case, we only want to use a POST request, a method supported by HTTP and used to accept data sent in the body of the incoming request. We can do this by creating a `Filter` requiring the POST method using `warp::post()` and then combining both filters using the previously mentioned `and` method.

When implementing the `struct` which actually should get sent, we used serialization into the JSON format. This is why the web server should expect a body encoded in JSON and try to parse the data into a `Data` instance. A `Filter` which does just that can be created using the function `warp::body::json()` and then again be combined using `and`.

Finally the extracted `Data` instance can be mapped to a response, which can be send back to the client. Here a simple formatting to a `String` using the `format!` macro works fine. The `{:?}` brackets the the formatting macro to use the `std::fmt::Debug` trait derived by `Data`.

The function `warp::serve` takes the `Filter` defined above as an argument and creates a new instance of the struct

`warp::Server`. This server represents a web server filtering requests according to the filter given at initialization.

To run this server, the struct implements the asynchronous method `run`, taking the address it should listen to as an argument and running the `Server` on the current thread until the program is closed. `ADDRESS` here has the type `([u8, 4], u16)`, a tupel with the ip address as a byte slice with a length of 4 and the port represented as an unsigned short.

To send the struct to this web server, we need to create a HTTP client. Reqwest is a crate often used for sending simple HTTP requests to a web server and similar to warp also makes use of asynchronous methods and requires some kind of runtime like tokio, at least when using its default API. In figure 12 there is an example implementation of a reqwest HTTP client sending a custom struct of the type `Data` to the warp server.

```
1  let client = reqwest::Client::new();
2
3  let response = client
4      .post(URL)
5      .json(&Data {
6          number: 5,
7          boolean: true,
8      })
9      .send()
10     .await
11     .expect("Sending request");
12
13 println!("{}", response.text()
    ↪   .await.expect("Get text"));
```

Fig. 12. Reqwest HTTP client

In line 1 an instance of the type `reqwest::Client` gets created in order to send asynchronous requests to the server. On this client we can call the method `post`, which creates a `reqwest::RequestBuilder` representing a post request. The argument `URL` is equal to the HTTP url of the targeted web server. To give it the data as a payload, the method `json` is called with a reference to our custom `Data` object. This method uses the `Serialize` trait of the struct to convert the data into a JSON `String`. Then the `send` method can be invoked to actually construct the request and send it to the target URL. To save the result of the request, the returned value is first unwrapped using **`await`** and `expect`.

`response` has the type `reqwest::Response`. To print the resulting text answered by the server, the asynchronous method `text` can be used. The `String` has to be unwrapped using **`await`** and `expect`, similar to the returned value of `send`.

If we run the example of figure 12 with the web server of figure 11 running at the same time, the client prints the response `Received: Data { number: 5, boolean: true }`.

## V. Conclusion

In conclusion, Rust can be a very powerful language for writing applications to run on network servers. It is often times very important to deploy a secure and efficient language in the backend of an application as the server underlying a service can be considered the bottleneck most of the times when it comes to verifying and handeling user requests. Slowing down or even outages of such applications can cause a lot of damage and should be avoided at any cost. Security vulnerabilities can also cause interruption of service or even accidental disclosure of sensible user data, depending on what type of application is affected.

Here Rust's checks at compile time come to shine, because many of the common security issues based on memory unsafety can be avoided in advance. Also shifting most of these checks into the process of compiling the code, Rust does not check for things like ownership and borrowing at runtime, thus making the executable even faster while still having all safety features in place.

In the course of the paper we have taken a look at general problems when working with network applications in other programming languages and the approach of Rust how to work around these boundaries. The standard library of Rust was introduced and it was demonstrated how it can be used to establish a connection between a client and a server using the Transmission Control Protocol (TCP). After that an User Datagram Protocol (UDP) client and server implementation was shown, which even allowed multiple clients to message the receiver at the same time. Next it was taken a look at asynchronous programing and how the introduction of a custom scheduler can replace blocking operations by swapping out tasks, which have reached a **`await`** point and have to wait for another task to evaluate. In this context the synchronous TCP implementation from before was converted into an asynchronous one and the advantage of not having to create a thread for each connection, that should be maintained at the same time, was mentioned. In the end a **`struct`** was sent from a client to a server using the Hypertext Transfer Protocol (HTTP) in order to show how asynchronous programming can also be applied in network applications on the application layer.

There are many aspects, which were not considered in the chapters before. When it comes to scalability one might consider to use a decentralized approach featuring loadbalancing with multiple servers handeling connections and possibly communicating with each other in the background. Also a lot of possible security vulnerabilities were not considered yet: Rust may be able to avoid security issues caused by null pointers, dangling pointers or data races, but this are not the only places where errors can occur. By instance if a server is accepting input from other clients, you usually want to parse and verify the input received, before handeling the message instead of blindly trusting the incoming data.

One final question, which might come up is why Rust is not used more for networking applications. In the introduction

it was mentioned that most networking APIs are writting in C or C++. The Rust Survey of 2020 suggests, that a huge issue is the adoption of the language in companies [11]. This makes sense as switching the language of an existing code base requires a lot of refactoring and introducing a new language adds the overhead of the employees first having to learn this new language. Another common reason is the steep learning curve related to the rather strict concepts of ownership and lifetimes the compiler enforces.

## REFERENCES


[1] F. E. Heart, R. E. Kahn, S. M. Ornstein, W. R. Crowther, and D. C. Walden, "The interface message processor for the arpa computer network," in *Proceedings of the May 5-7, 1970, Spring Joint Computer Conference*, ser. AFIPS '70 (Spring). New York, NY, USA: Association for Computing Machinery, 1970, p. 551–567.

[2] "The rust programming language." [Online]. Available: https://www.rust-lang.org/

[3] C. Cowan, F. Wagle, C. Pu, S. Beattie, and J. Walpole, "Buffer overflows: attacks and defenses for the vulnerability of the decade," in *Proceedings DARPA Information Survivability Conference and Exposition*, vol. 2, 2000, pp. 119–129 vol.2.

[4] R. Clipsham, "Safe, correct, and fast low-level networking," 2015.

[5] J. Postel, "Transmission control protocol," Internet Requests for Comments, RFC Editor, STD 7, September 1981. [Online]. Available: http://www.rfc-editor.org/rfc/rfc793.txt

[6] ——, "User datagram protocol," Internet Requests for Comments, RFC Editor, STD 6, August 1980. [Online]. Available: http://www.rfc-editor.org/rfc/rfc768.txt

[7] A. Chanda, *Network Programming with Rust*. Packt Publishing, 2018, ch. Asynchronous Network Programming Using Tokio.

[8] "Tokio crate documentation." [Online]. Available: https://docs.rs/tokio/1.5.0/tokio/

[9] "Asynchronous programming in rust." [Online]. Available: https://rust-lang.github.io/async-book/

[10] "Warp crate documentation." [Online]. Available: https://docs.rs/warp/0.3.1/warp/

[11] "Rust survey 2020 results." [Online]. Available: https://blog.rust-lang.org/2020/12/16/rust-survey-2020.html