Rust for Network Servers

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

June 21, 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. Furthermore, the deployment of Rust on the application layer is explained.

***Index Terms*—Rust, computer network, TCP, UDP, asynchronous programming**

I. Introduction

Many modern devices 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 popular programming language supports these frequently deployed network protocols, whether natively in its standard library or through simple imports of third-party libraries. All these options raise the question of what language to use for own application purposes. There is no right answer to this question as it depends on many aspects such as the particular use case, the running environment, time, and cost of development. This paper highlights the 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 Transmission Control Protocol (TCP) and User Datagram Protocol (UDP).

There are usually different approaches when getting to pick a language for networking, especially when working with a transport protocol and a custom application protocol on top of it.

One option would be using a scripting language like Python or JavaScript. With such languages, it is possible to easily import libraries that offer a basic network API and a simple syntax allowing fast prototyping. As these languages usually are interpreted, they often operate 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 are important factors (as they are most of the time), a compiled language could be a good choice. This is the reason why many network libraries are written in C or C++. However, these two languages do not guarantee memory safety. Mistakes by the programmer can result in memory usage errors like invalid accesses or leaks.

This might be a problem, in particular when working with network applications as it is common to read from and write to buffers concurrently. For example, if incoming information is just written to a buffer without checking the buffer size, an attacker might be able to overwrite data in the memory and change the control 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. [2]

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

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 TCP/UDP communication, enabling socket communication and error handling out of the box. [3]

Rust is also a compiled language, but the compiler guarantees 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 that could introduce such issues.

Network operations are always prone to failure because the success of such an operation depends on outside factors like other hosts and a stable network connection. Rust cannot 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 enum forces the programmer to handle errors in some kind of way as it is required to unwrap the `Result` before being able to access the returned value, preventing the program to continue running in a faulty state.

Another selling point of Rust is its well-integrated package manager *Cargo*, making it possible to easily import, update and integrate third-party libraries called crates and ship them with an own application.

To see, how these benefits apply in basic TCP and UDP communication, chapter II will first give a short overview of ownership, borrowing and lifetimes. After that, it will explain the implementations of a TCP client and server as well as a UDP sender and receiver, each programmed in Rust using the standard library. Chapter III 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 are introduced. After that, an asynchronous server and client code example are shown to explain possible differences and advantages compared to a synchronous and blocking approach. Chapter IV 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, the discussed results will get summarized.

II. Networking in Rust

As it was already mentioned before, Rust uses verification at compile time to stop compiling the code when something possibly unsafe could happen at runtime without being explicitly marked as unsafe. To check efficiently for any of these unsafe memory accesses or possible data races, Rust uses the so-called ownership system as well as the principles of borrowing and lifetimes. It is crucial to be familiar with these concepts before actually starting to implement an own program communicating over the network. 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 the variable after it was moved. Sometimes a function does not need complete access to a value, in which case it is also sufficient to obtain a reference to the value by borrowing from it using the `&` operator. This enables the code calling the function to still be able to access the piece of memory afterward. If the callee needs to modify the value, a mutable reference has to be created. In order to avoid data races, it is only allowed to have a single mutable reference to a variable at once.

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

The Rust compiler does not check for illegal array accesses but unlike in C, the program exits in a controlled manner (this is called a ”panic”) if the program tries to access an index which is out of bounds. This does not prevent buffer overflows from taking 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 in. If a variable falls out of scope, the corresponding memory segment is freed automatically. If a value gets moved into another function, its lifetime changes to the scope of the new function. 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. It can be used to send a continuous stream of octets to another host. In order to check whether the data was received by the other system, 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 one can work with these abstractions independently of what platform the code is currently running on. To demonstrate this in a better way, we are going to implement a small TCP server that 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.

```rust
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 handling

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

As network operations might fail due to multiple reasons (for instance network failure, package loss, insufficient permissions, etc.), these 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)`. However, 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. Here we want to read whole lines from the stream, so the reader is able to keep 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 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 is successful but the amount of read bytes is equal to 0 it means that the connection was closed and we can break out of the loop. If at least one byte is 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.

If it 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 shut down the stream using `stream.shutdown(Both)?`. This method takes an enum of the type `std::net::Shutdown` and 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 method, returning from the whole function immediately if the result is equal to an `Err`. Otherwise, the program ignores the result and simply breaks out of the loop.

The example in 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 run the code, we still have to initialize the stream in the first place. As a matter of simplification, let us assume that the code of Figure 1 is now wrapped into a function called `handle_client`, taking the handled `TcpStream` as an argument. We are now able to call this function in the code of Figure 2.

First, a listener of the type `std::net::TcpListener` has to get created by binding it to `ADDRESS`. The constant `ADDRESS` is of 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 the message provided in the argument and the `Err` value formatted as a `String` in case of failure.

A `TcpListener` has an `accept` member function, which blocks the calling thread until a new TCP connection is found and returns the `TcpStream` and the address of the other host

```rust
let listener = TcpListener::bind(ADDRESS)
    .expect("Bind to address");

for stream in listener.incoming().flatten() {
    handle_client(stream)
        .expect("Handle client");
}
```

Fig. 2. TCP listener

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. As opening a TCP connection could fail, these connections are wrapped into the type `Result`. The method `flatten` called on the iterator flattens the iterator of results, implicitly converting `Err` values into empty iterators, `Ok` values into iterators of size 1, and chaining the inner iterators together again. This means invalid connections get filtered immediately. We can now use a `for` loop in order to continuously listen for a new TCP connection as soon as the contact to the current host is lost.

As soon as a connection was established successfully, our function `handle_client` is called with `stream` as an argument. Because the handling 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 in the server being more stable as it does not panic and crash if a single connection throws an error.

The last missing piece to use the 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 a `&str` `ADDRESS` as an argument. The constant `ADDRESS` is similar to the constant we have used in the server and corresponds to the address and port of the server we want to connect to. 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 return early with an error if something went wrong.

In order to let the user send messages from the command line, we have to use the `struct std::io::Stdin` which one can get by calling the function `std::io::stdin`. To prevent repeated calls of this function, it is called once in the beginning and saved in an immutable variable.

After that, the program enters a loop and keeps iterating until the connection was closed. In line 6 a mutable `String` called `buf` is created. The variable is used to save the console input of the user and then send it to the server.

In line 8 the method `read_line` of `stdin` receives the

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

let stdin = stdin();

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

next line from the user. It takes a mutable reference to the buffer into which the input should be read as an argument and returns a `Result` corresponding to whether reading was successful.

If so, the content of `buf` is trimmed to remove leading and trailing whitespaces and checked whether it matches ”exit”. In such a 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 sends it directly to the server. As this is another operation that might fail due to network issues, it again returns a `Result` which is unwrapped through the `expect` method.

If reading from the command line has failed, the program handles it the same way 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 this was successful, it breaks out of the loop resulting in the program terminating.

When handling a TCP connection in Rust, it can be difficult to detect whether the connection to the server was lost, because 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. Different from TCP, UDP tries to send messages to other programs with a minimal amount of protocol mechanism. This means that it offers no protection against package duplication and does neither check whether the sent data arrived nor in what order. As a consequence, it is unnecessary to listen for new hosts and actively establish a connection to another host before sending any data. UDP is commonly used if reliable delivery of streams and small amounts 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, one is able to use the **`struct`** `std::net::UdpSocket`. It provides methods to bind to an address, to 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 a receiver who receives these messages. Figure 4 gives an example of 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` function 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 occurs or it is closed explicitly. As there are no active 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 store the data of the other host. A `UdpSocket` does not implement the trait `std::io::Read`. A trait in Rust defines some kind of behavior, similar to interfaces in other languages. This means it is not possible to create a `BufReader` instance in order to read the data from the socket. Instead, 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 length `1500`. If the size of the data is known beforehand, one 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 Maximum Transmission Unit (MTU). If a message is larger than the buffer, it is going truncated.

In line 6 the method `recv_from` is used to read the next arriving package into `buf`. If the call is 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, using the TCP client implemented before will not work as it uses a different protocol. In Figure 5 there is a small implementation of a UDP sender.

```rust
let socket =
 ↪  UdpSocket::bind("0.0.0.0:0")?;

let stdin = stdin();

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

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

            let bytes = buf.as_bytes();

            socket
                .send_to(bytes, ADDRESS)
                .expect("Sending");
        }
        Err(e) => {
            eprintln!("{}", e);
            break;
        }
    }
}
```

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` function is given the static address "0.0.0.0:0" due to the UDP protocol. As there is no set connection between the two hosts, our sender needs its address and port in order to receive answers from the other host. By giving `bind` 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 which it is allowed to use. Setting the port explicitly is an alternative method, although this risks choosing a port that is already used by another program.

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

If the reading is successful, the input is first trimmed and then checked whether it matches "exit". In this case, the program simply breaks out of the loop. We do not have to shut down 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 the method call is unwrapped using `expect` to panic in case of an error.

III. Asynchronous networking using Tokio

In chapter II, basic examples for TCP and UDP communication in Rust were shown. In these programs the current thread was blocked when waiting for new connections or incoming messages. This not only results in the program having to wait for a new message to continue executing the code, but it also means that it is necessary to create a new thread for each connection if the server should listen to multiple connections at once. This principle is called synchronous 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. The result can be unwrapped when it is needed instead of waiting for the execution in advance. [7]

A. The Tokio crate

The core library of Rust is kept very small. However, the package manager *Cargo* allows an easy import of third-party packages. This way these extensions are not bound to the release cycle of Rust and breaking changes will only affect one single package instead of the whole library. For compatibility reasons, a project could 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. By adding the name and the version to the dependencies of a project, it gets imported automatically. Some crates also offer feature flags to decide which parts of the library should be included.

This prevents the whole library from getting compiled with the code.

To introduce asynchronous programming to Rust it is feasible to use a crate that implements a runtime for the execution of asynchronous functions. Rust already provides a minimal implementation for async code, but a lot of key parts like executors, tasks, and reactors are missing. Crates like *Tokio* build on top of these basic implementations, filling in the gaps of necessary parts which are not implemented yet. 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 such as sockets and filesystem operations.

A task in the context of a *Tokio* program is a non-blocking piece of code. The `tokio::task::spawn` method can be used 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. A task can be compared to a promise in the programming language JavaScript.

The *Tokio* runtime consists of three major parts: An I/O event loop, which handles I/O events and dispatches them to the waiting tasks, a timer to run tasks after a certain period of time and a scheduler, which executes the tasks on the different threads. One way to enable the *Tokio* runtime is to mark the main function with the **`async`** keyword and add the `#[tokio::main]` attribute macro above.

The scheduler swaps around all tasks which need to be run. This means there might be more tasks than threads at the same time 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, one 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.

The following chapters are going to use the crate *Tokio* mentioned before as a runtime for all the asynchronous code provided in the examples.

B. Programming with futures

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

The declared function types play an important role when working with *Tokio*. This is 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, `async_function` is called. The value returned by the function has the type `std::future::Future`, which wraps the result of the computation that is going to get evaluated later. The `Future` is stored in the variable `future1`. This way the *Tokio* runtime decides when to evaluate the function. If `async_function` was not marked with **`async`**, the main thread executing the program would enter the function,

```rust
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

completely evaluate it and after that continue executing the main function.

The type of `future2` is also a `Future`, the only difference is the use of 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 be evaluated completely while executing them concurrently.

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

In a blocking environment, this would result in the program taking more than eight seconds to execute, because the returned value of `future1` is evaluated first and only after that the value of `future2`. Therefore, 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 start two threads explicitly and execute both tasks concurrently. If multiple tasks have to run in parallel and include a lot of blocking operations, it would cause a lot of overhead to the program if a thread for each task was created.

On the other hand in an asynchronous environment, the scheduler stops the execution of the current task and swaps in another task which is currently waiting for execution as soon as 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 when 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 network API similar to the one of the standard library used in chapter II. These functions are marked as **async**, blending in with the rest of the *Tokio* ecosystem.

Let us first use *Tokio* to spawn a new task that handles a connection by reading all incoming messages and printing the output to the command line. The code in Figure 7 implements basic asynchronous client handling.

```
1   tokio::spawn(async move {
2       let mut reader =
      ↪ BufReader::new(&mut stream);
3
4       loop {
5           let mut buf = String::new();
6
7           match reader
8               .read_line(&mut buf).await {
9               Ok(0) => break,
10              Ok(size) =>
      ↪ print!("{}", buf),
11              Err(e) => {
12                  eprintln!("{}", e);
13                  stream
14                      .shutdown().await?;
15                  break;
16              }
17          }
18      }
19
20      Ok::<(), Error>(())
21  });
```

Fig. 7. Asynchronous TCP client handling

The current connection is represented by 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.

The function is given an asynchronous code block as an argument, which returns a `Future` as seen in Figure 6. Here additionally to the **async** keyword, **move** is used. This means the asynchronous block will take 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 works similar to the synchronous TCP client handling 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 read data from the connection repeatedly. 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 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 is 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 shut down 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. Different from 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 a `Result` wrapped by a `Future`. First, the `Future` gets unwrapped by calling **await** on it. Afterward, the `Result` is unwrapped using the `?` operator and the program breaks out of the loop.

At the end of the task, we have to specify the type of the `Result` returned explicitly. 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, it is possible for the compiler 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 or creating an extra asynchronous function instead of using **async** blocks. [7]

As in chapter II we are now able to 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 `handle_client` with the stream as an argument. As mentioned before, 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 needed to spawn a task for each TCP stream. This is why the code example in 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 named `ADDRESS`. The 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 returns a tuple 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 received stream can be given to our `handle_client` function, which spawns the new task handling the current user. As the client handling 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.

In the beginning a `tokio::net::TcpStream` instance is created by connecting to `ADDRESS`, a constant in which the address of the server is stored as a `&str`. Different to the synchronous implementation, we are not going to read from the command line directly but rather wrap it using a `tokio::io::BufReader` instance in order 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 in Figure 3: In line 6 a new `String` called `buf` is created to store 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 is written into the buffer. After new input has arrived, the scheduler continues to execute the current task and the return value of `read_line` is matched.

If reading is successful, the program enters the first match arm. In Figure 3 this match arm included a check whether the

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

let mut reader = BufReader::new(stdin());

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

    match reader
        .read_line(&mut buf).await {
        Ok(_) => {
            let bytes = buf.as_bytes();

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

Fig. 9. Asynchronous TCP client

read `String` is equal to ”exit” to offer a way to close the connection other than just by forcing it. This part is omitted in Figure 9 as it works in the same way. The 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 shut down 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 other hand is idle while waiting for input on the command line because it does not have any additional tasks. When expanding the implementation to a complete chat client which can both send and receive messages, a second task would be needed to listen for new data on the connection. Here an asynchronous environment would be more useful.

D. Asynchronous UDP communication

Concerning asynchronous UDP communication, *Tokio* offers the **`struct`** `tokio::net::UdpSocket` which works similar to the UdpSocket of the standard library used in Figures 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, **await** can be used.

As UDP does not implement persistent 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 when the sender and receiver are as strictly separated 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, it would not be possible to 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 chapters II and III, 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 transfer special data like a **struct** through a connection. This is where protocols belonging to the application layer are applicable.

The following chapter will give a small insight into the principle of how 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, a piece of data to 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.

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

Fig. 10. Data struct

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, using a principle called serialization and deserialization.

For this, we are going to use a crate called *Serde*. *Serde* allows us to let our **struct** derive the traits 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 a standard file format typically used for this exact purpose.

We additionally derive Debug to implement 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 we can convert the **struct** into a format that can be sent over HTTP, next, a server is needed to accept incoming requests. There are multiple crates that enable accepting and handling HTTP requests in Rust. For our HTTP server implementation, we are going to use a crate called *Warp*.

Warp is a simple web server framework, which builds on top of the crate Hyper, an HTTP implementation for Rust [9]. *Warp* makes use of asynchronous programming and seamlessly works together with *Tokio*. An important concept of *Warp* is the so-called filter: warp::Filter is a **struct** that 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 one can chain together multiple Filter instances and handle requests more precisely.

Secondly, the method map takes a function as an argument. This function receives the extracted data from the request and maps it to another value. In return, the value 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 these methods will not be needed for our example.

In Figure 11 there is a small example implementation of a web server written using *Warp*.

```rust
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 utilized. It 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, what the request looks like exactly. This enables us to reject invalid requests automatically. In this case, we only want to use a POST request, a common HTTP method 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 gets sent, we have 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` provides this functionality. It can be created using the function `warp::body::json()` and is again combined using `and`.

Finally, the extracted `Data` instance can be mapped to a response, which in result is sent back to the client. Here, simply formatting it to a `String` using the `format!` macro works fine. The `{:?}` brackets 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 our web server, filtering requests according to the filter given at initialization.

To run this server, the **`struct`** implements an 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. Here, `ADDRESS` has the type `([u8, 4], u16)`, a tuple 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 like *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.

In line 1 an instance of the type `reqwest::Client` is created in order to send asynchronous requests to the server. On this client, we 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 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`.

```
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

`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

One final question which might come up is why Rust is not used more frequently for network applications. In the introduction, it was mentioned that most network APIs are written in C or C++. The Rust Survey of 2020 suggests, that a huge issue is the adoption of the language in companies [10]. 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.

In the course of this paper, we have taken a look at general problems when working with network applications in other programming languages and the approach of using Rust to work around these boundaries. The standard library of Rust was introduced and it was demonstrated how it can be utilized to establish a connection between a client and a server using the Transmission Control Protocol (TCP). After that, a User Datagram Protocol (UDP) client and server implementation was shown, which allowed multiple clients to message the receiver at the same time. Next, we took a look at asynchronous programming and how the introduction of a custom scheduler can replace blocking operations by swapping out tasks, which have reached an **`await`** point and have to wait for another task to evaluate. In this context, the synchronous TCP implementation from earlier 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 demonstrated. 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 used in network applications on the application layer.

Many aspects were not considered in the previous chapters. To improve scalability one might consider to use a decentralized approach featuring load balancing with multiple servers handling connections and possibly communicating with each other in the background. Also a lot of possible security vulnerabilities were not considered yet: Rust is able to avoid security issues caused by null pointers, dangling pointers or data races, but these are not the only types of errors that can occur. For instance, if a server is accepting input from other clients, it is usually best to parse and verify the input received before handling the message instead of blindly trusting the incoming data.

In conclusion, Rust is a very powerful language for writing programs to run on network servers. It is often 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 time when it comes to verifying and handling user requests. Slowing down or even outages of such applications might cause a lot of damage and should be avoided at any cost. Security vulnerabilities could also cause interruption of service or even accidental disclosure of sensitive user data, depending on which 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 prevented in advance. In addition, 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 retaining all safety features.

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] 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.

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

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

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

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

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