CSC 412 Assignment

Topic : Implementation of a reliable UDP Protocol

Name : Ayan Das

Date : 12/10/2024

Introduction

The reliable UDP protocol was implemented using rust. Partly because rust is a programming language that I have wanted to learn for a long time and I wanted to use this assingment as an opportunity to do so. In regards to the starter code, I did not use the starter code provided by the textbook, since it was written in python and I already have familiarity regarding python.

Description

To handle the underlying logic of ensuring the UDP protocol is reliable, I am using a library named **laminar**. It is an application-level transport protocol which is built on top of UDP and allows for configuration regarding reliabillity vs unreliabillity. Meaning, we can configure the library to send reliable or unreliable data. Original use cases of the library has been for any kind of multiplayer games (such as first person shooters), as well as chatrooms for end to end messaging. The two types of packets that can be sent using this library are reliable and unreliable packets with the added support for keeping track of the sequence. However, the logic for implementing the sequence number is relatively simple and to keep the scope of the project simple, I handled the packet sequencing logic through basic for loop. The for loop iterates between the range of 1 to 10 (inclusive) when sending the package, attaching the sequence number to the packet during request from the client and as well as during response from the server upon recieving request from the client.

Given that there was no constraint on the assignment regarding what PORT we may use, again, to retain simplicity, I used port 3000 for the server side and port 4000 for the client side. However, any available port that isn't reserved can be utilized for sending **ping** messages and recieving **pong** messages. Additinally, because the client and servers are both within my personal device, to self reference my device, the code for setting client and server are the following shown below:

```
const SERVER : &str = "127.0.0.1:3000";
const CLIENT : &str = "127.0.0.1:4000";
```

To break down the code above, the **const** keyword in rust is used to declare global variables. In this case, the variables **SERVER** and **CLIENT** are both global variables. Additionally **&str** is the datatype in this case. This is something exclusive to Rust, unlike high level languages such as Python or TypeScript, where the type is generally inferred as **str** or **string**, respectively, Rust has two types of string datatypes. There's borrowed string, represented as **&str** and owned string, which is represented as String. Borrowed strings is the default type of string within the language, and as the name suggest, rather than allocating space within the memory for the string, the string is being referenced. On the other hand, if we wanted to initialize the string in the form of owned string, the syntax would differ and would look like the following below:

```
const SERVER : String = "127.0.0.1:3000".to_owned();
const CLIENT : String = "127.0.0.1:4000".to_owned();
```

Generally speaking, the convention is to use **&str** for strings that will not be **modified**. In this case, the server and client addresses remains static throughout the entirety of the program.

In order to improve readabillity of the code, I have divided the server and client code into two seperate functions of their own. Which interacts with one another within the main function. In total, the program contains three functions, as shown below:

```
/// NOTE : This is just an high-level abstraction, I will go more in-depth regarding
the implementaiton in the later portion of the report
fn main() {
  // recieve user input
  // similar to std::cin in C++
  // here, the retrieved user input is saved
  // within the variable stdin
  let stdin = std::io::stdin();

  // do something with client
  client();
  // do something with server
  server();
}
```

After declaring the addresses, we need to bind the socket to the port, otherwise we will not be able to recieve data. In order to bind a socket, we need two things : an ip address and a port number. In the context of this program, the IP address happens to be an special purpose IP address, and popularly known as “loopback address” or “local host”. All computers use this address as their own, however, this IP address cannot be used to communicate with other devices.

In terms of the function definitions, the first function I defined was the **server()** function. Given the nature of Rust, similar to C++, the default return type is void, meaning the function doesn’t expect anything to be returned. However, in the context of the problem that needed to be solved, I needed to specify a specific return type. It will enable me to catch any kind of potential errors. The function declaration line is the following:

```
fn server() -> Result<(), ErrorKind>
```

The **Result** type is used to return and propagate errors. The Result type takes in two parameters, the first parameter, **()** represents the success type parameter, which is similar to **void** in C++, meaning no value is returned at the end of the function execution. Because the goal is to perform in-place modification rather than return a value in this scenario. In the event that the function does return an error, the error will be of type **ErrorKind**, which also happens to be the second parameter Result expects, the type of errors that should be returned. ErrorKind comes from the same library that handles reliable data transfer using UDP : **laminar**, and it handles all possible network errors that could occur.

The implementaiton of ErrorKind according to the documentation is the following:

```
/// import the enum
/// we use "use" for importing libraries in Rust
use laminar::ErrorKind;

// within the enum represents all possible types of errors that may occur
// the "pub" keyword is used to make the variable/class/functions public

pub enum ErrorKind {
  // Error in decoding the packet
  DecodingError(DecodingErrorKind),

  // Error related to recieving or pasing a fragment
```

```
  FragmentError(FragmentErrorKind),

  // Error relating to recieving or pasing a packet
  PacketError(PacketErrorKind)

  // Wrapper around a std::io::Error (meaning it's built on top of the standard error
message when there's an error related to input/output)
  IOError(Error)

  // Did not recieve enough data, this error is triggered when the requirement for
minimum number of data was not recieved
  RecievedDataToShort,

  // This error is raised in the event that the protocol version did not match
  // for example if one device attempts to send data through UDP whereas the
recieving end only accepts TCP based data
  ProtocolVersionMismatch,

  // occurs when the message could not be sent because the channel is disconnected

  // this error message also helps recover the message that could not be transmitted
successfully due to the channels being disconnected
  SendError(SendError<SocketEvent>),

  //  Error that occurs when reciever expected header but data could not be read from
the buffer
  CouldNotReadHeader(String)
}
```

As part of the function definition, as mentioned previously, we first start by binding the socket to the IP address **127.0.0.1** and port **3000**. The code for this binding logic was implemented in the following manner:

```
// the "?" at the end is used to catch potential errors that may occur during the
process of socket binding
// "mut" is used to state that the variable can be modified
// in Rust, variables are immutable by default, meaning they cannot be modified after
being defined unless we include the "mut" keyword
let mut socket = Socket::bind(SERVER)?;
```

The **socket** is also an object, and objects can access the methods that has been defined within the class. In this case, consider bind to be an class, and within bind contains various methods. The two methods that we are concerned with is **get_packet_sender()** and **get_event_reciever()**, this will help us identify the sender and reciver sockets.

According to the documentation, the **get_packet_sender()** “returns a handle to the packet sender which provides a thread-safe way to enqueue packets to be processed”. This method is used to pre-process the packets that should be sent over the network.

Similarly, the **get_event_reciever()** “returns a handle to the event reciever which provides a thread-safe way to retrieve events from the socket”. Meaning this method is used to recieve the packets by the destination that are sent over the network. The code for this is the following:

```
// syntax for how we can define multiple variables in a single line
//
// note that these variables cannot be modified, if we attempt to modify them, the
```

```
compiler will panic and throw an error
let (sender, reciever) = (socket.get_packet_sender(), socket.get_event_reciever());
```

Due to the client server nature of the connection, we need to ensure that the socket are polled. **Polling** "refers to a technique where a client repeatedly sends a request to a server at regular intervals to check for new data". It is another form of the client asking if there's any new data/ updates available from the server. Polling is a pull-based mechanism and helpful for retrieving data from the server, similar to HTTP protocol, which is also a pull based protocol. Rather than the server actively sending updates, the client checks for updates from the server in this scenario every 1ms (1ms is the default configuration of the polling method built into the socket in the context of this program). For optimization and ensuring there are no memory leaks, the socket's polling mechanism occurs within a new thread that is spawned. The code for this is the following:

```
// import the built in thread library
// comes with rust's standard library
use std::thread;

// define thread as a closure
// a new thread is spawned
// thread takes in a closure of type FnOnce (this is the default type for closure)
// closure is a concept of functional programming
// || indicates that no parameters are being passed into the closure based function
thread::spawn(move || socket.start_polling());
```

The next part of the code is used to keep track of the sequence number. The sequnece number, for the sake of simplcity of logic, tracked the sequence number by a seperate mutable variable. The code for this is the following:

```
// seq keeps track of the current sequence value of the packet
let mut seq = 1;
```

In terms of the logic of sending and sending and recieving the packets, the goal of the server is to be on the lookout for messages (requests) from the client side. The client should be the one initiating the request throguh sending ping messages, while the server remains on the lookout for incoming messages. To keep track of the string buffer, a basic **String** type variable is being used. Given the nature of the logic, the server can be started and will remain open (persistent connection), until the user manually decides to quit the server. The server can be quit by pressing **control + C** on windows and **cmd + c** on macOS. Or we can also input **"Bye!"** within the terminal and the server will close (we will break out of the loop as well). Additionally, the loop is helpful because the network may not be able to send all the messages at once (the packet may need to be broken down into smaller segments), this is dependent on the capacity of the link layer. If the server successfully recieves the message, the event will be of success type (we verify this using the **match** keyword). Extract the payload data (as we would typically do in an API endpoint for any kind of mobile/web based applications). The message is contained in the payload, the string may have bit error rates. Rust has a built in method to detect corrupted/invalid UTF 8 characters and replace them with valid UTF characters withint the String library. This is the "reconstruction" part of the message on the destination end. Additionally, the packet's header also contains the ip address, allowing us to determine the original source port and ip addresses from which the message originated. Once this process ends, server will send a confirmation message stating that the message has been recieved on the server side of the eterminal, specifying the message that has been recieved as well as information regarding the IP and port addresses from which the message originated. The server subsequnely sends a Pong message back. The Pong message is wrapped around the **reliable_sequenced** wrapper, which ensures that the response message is sent in an reliable and

orderly manner over the network. The reliable_sequenced method takes in 3 paramters, the address to which the message will be sent, we already have this stored as part of the header attached to the packet, the response message that is being sent (converted into bytes and each bit being a vector element), and an error handler to catch for any errors during transmission from the server side. Lastly, in the event that the server remains idle for over 5 seconds, meaning it doesn't recieve any incoming connections for the five seconds, it will assume client has closed it's connection (becuase 5 seconds is the timeout interval), this can lead to temporary socket timeout before connection gets re-established. Note that the connection will re-estbalish as long as the server side terminal remains open and client starts resending requests to server. After the server timeout window occur, the first request from client will not be sent successfully, that's where I manually re-transmit Ping messages from the client side a second time due the first transmission failing, and the second transmission will be sent successfully and responded to in accordance.

The code for the above logic is the following:

```
// the loop is used to recieve the packets
    // the network may not be capable of sending all the packets at once
    // this depends on how much data it can send within a small timeframe
    // therefore, we need to use a loop based statement
    loop {
        // equivalent to using try/catch in javascript
        // logic defining what the server should do upon successfully recieving the
packets
        // the message displayed here is what we see on the server side as a response
        if let Ok(event) = receiver.recv() {
            match event {
                SocketEvent::Packet(packet) => {
                    // the payload contains the message of the packet
                    let msg = packet.payload();

                    // no response will be sent if user inputs Bye!
                    if msg == b"Bye!" {
                        break;
                    }

                    // this is where the package lost logic occurs
                    // replaces invalid UTF-8 characters
                    // with valid UTF-8 characters
                    let msg = String::from_utf8_lossy(msg);

                    // extracts the ip from the packet
                    let ip = packet.addr().ip();

                    println!("Received message {:?} from {:?}", msg, ip);
                    // increment the sequence number by 1
                    //seq = seq + 1;
                    let response = format!("Pong!");
                    sender
                        .send(Packet::reliable_sequenced(
                            // sends packet in reliable + orderly manner
                            packet.addr(),
                            // TODO : this needs to be modified
                            response.as_bytes().to_vec(),
                            Some(seq),
                        ))
```

```
                        .expect("This should send"); // error handler in the event
packets aren't send
                }
                // temporary socket timeout before connection is re-established
                // occurs if the thread idles for too long
                SocketEvent::Timeout(address) => {
                    println!("Client timed out: {}", address);
                }
                _ => {}
            }
        }
    }
```

Lastly, we need to specify the return value, which in this case is void type with a **Result** wrapper. To satisfy the compiler during program execution, the last line of the function execution must contain the following:

```
Ok(())  // indicates the end of a successful function call with no errors
```

The complete code the server would look like the following:

```
fn server() -> Result<(), ErrorKind> {
    /**
     * In Rust, "binding a socket" means associating specific network address
     * like an IP address and port with a newly created socket
     * it essentially tells the operating system that this socket should be reachable
at that particular address
     *
     * allowing other applications to connect to it o the network
     * -> Result<(), ErrorKind> allows us to use ? instead of .unwrap()
     * unwrap() is older form of error handling, ? is a newer form of handling errors
     *
     * socket.get_packet_sender() --> Returns a handle to the packet sender which
provides a thread-safe way to enqueue packets to be processed. This could be used
when the socket is busy running it's polling loop in a seperate thread
     *
     * socket.get_event_reciever() --> returns a handle to the event reciever which
provides a thread-safe-way to retrieve events from the socket. The use case is
similar to get_packet_sender() method.
     */
    let mut socket = Socket::bind(SERVER)?;
    let (sender, receiver) = (socket.get_packet_sender(),
socket.get_event_receiver());

    // define _thread as a closure
    // a new thread is spawned
    // thread takes in a closure of type FnOnce
    // meaning socket.start_polling() should only execute once
    // this is automatically executed when the program runs
    thread::spawn(move || socket.start_polling());

    // seq keeps track of the current sequence value of the packet
    let mut seq = 1;

    // the loop is used to recieve the packets
    // the network may not be capable of sending all the packets at once
    // this depends on how much data it can send within a small timeframe
    // therefore, we need to use a loop based statement
```

```
    loop {
        // equivalent to using try/catch in javascript
        // logic defining what the server should do upon successfully recieving the
packets
        // the message displayed here is what we see on the server side as a response
        if let Ok(event) = receiver.recv() {
            match event {
                SocketEvent::Packet(packet) => {
                    // the payload contains the message of the packet
                    let msg = packet.payload();

                    // no response will be sent if user inputs Bye!
                    if msg == b"Bye!" {
                        break;
                    }

                    // this is where the package lost logic occurs
                    // replaces invalid UTF-8 characters
                    // with valid UTF-8 characters
                    let msg = String::from_utf8_lossy(msg);

                    // extracts the ip from the packet
                    let ip = packet.addr().ip();

                    println!("Received message {:?} from {:?}", msg, ip);
                    // increment the sequence number by 1
                    //seq = seq + 1;
                    let response = format!("Pong!");
                    sender
                        .send(Packet::reliable_sequenced(
                            // sends packet in reliable + orderly manner
                            packet.addr(),
                            // TODO : this needs to be modified
                            response.as_bytes().to_vec(),
                            Some(seq),
                        ))
                        .expect("This should send"); // error handler in the event
packets aren't send
                }
                // temporary socket timeout before connection is re-established
                // occurs if the thread idles for too long
                SocketEvent::Timeout(address) => {
                    println!("Client timed out: {}", address);
                }
                _ => {}
            }
        }
    }

    Ok(())
}
```

After defining the server, we need to define the client. The first part of the implementation of the client is similar to the implementation of the server. The logic for defining the Ip and Port number, alongside the binding process and initializing the sequence number. In fact the process of quitting the persisting connection of the client works similar to server, we can quit using the same

keybindings mentioned previously. Given the nature of the client function design, the client is capable of not just sending **ping** based inputs but any kind of inputs are allowed, the server will still respond with a Pong message upon successfully recieving request from the client side. The request that gets sent is what the user provides, the input is submitted by the user. The code for up to this implementaiton is the following:

```
fn client(number_of_requests: u8) -> Result<(), ErrorKind> {
    // this is the client side socket/ip
    // @127.0.0.1 : is a special ip address known as "the loopback address"
    // it is used by the computer to refer to itself
    // @:* represents the port number, this can be any available port within the
operating system
    let addr = "127.0.0.1:4000";
    let mut socket = Socket::bind(addr)?;
    println!("Connected on {}", addr);
    let mut seq = 1;

    // when we bind a socket
    // the address gets "wrapped" around, essentially imagine wrapping a physical
object with a gift wrapper or placing a content withing a container
    // therefore, we need to "unwrap" it
    // thus the use of unwrap()
    // parse() helps convert
    let server = SERVER.parse().unwrap();

    println!("Type a message and press Enter to send. Send `Bye!` to quit.");

    let stdin = stdin();
    let mut s_buffer = String::new();
```

Next comes the outer loop, this loop also exists within the **server()** function as well. The user input is read and stored in the buffer, the buffer is then parsed to remove any dangling non UTF-8 characters and newlines that may have been included in the user input. The code for this is the following:

```
loop {
        s_buffer.clear();
        stdin.read_line(&mut s_buffer)?;
        let mut line = s_buffer.replace(|x| x == '\n' || x == '\r', "");

        // additional logic work
        // will be explained in the subsequent portion of the report.
}
```

Once the string has been cleared out of unneccessary values/whitespaces that may have existed as part of the user input, the message is now ready to be converted into bytes and sent over the network to the server over the reliable and sequenced UDP protocol. In order to adhere to the requirement of the assignment of sending 10 ping messages back to back, I used a for loop, and I also adjusted the **client()** function by including a parameter, allowing for added customizabillity regarding how many messages can be sent back to back. Meaning during the function call, whatever number is specified as part of the parameter, the total number of requests sent back to back will be upto the number specified in the parameter, starting at 1. This means we can not only send 10 ping messages back to back, but even 30, 100 or more, but the server can get overloaded and there's no congestion control built in within the library, the server will tmeout if congestion on the destination occurs due to influx of request and will stop accepting requests and responding unless the sockets

are manually restarted. Similar to before, evey time a message is sent out from the client side, sequence number is incremented by 1, that's how the sequence number is kept track of. When sending the request, the same method from the library is used, **reliable_sequenced**, and it accepts 3 parameters, as those 3 are all required parameters. The code for thisis the following:

```
// send 10 ping messages back to back
        // it's not limited to just ping messages
        // after sending the specified number of requests here back to back
        // the server will be sending waiting for acknowledgement for
        for i in 1..number_of_requests + 1 {
            // start the timer at the beggining of each iteration
            let now = Instant::now();
            let string = i.to_string();
            //line.push_str(&string);

            // send reliable sequence data
            socket.send(Packet::reliable_sequenced(
                server,
                // creates a copy of the string, as the name implies
                // converts it into bytes
                // so that it can be sent over the network to the server port
                line.clone().into_bytes(),
                Some(i),
            ))?;

  // Additional logic, not shown here as that has not yet been discussed.
        }
```

The sequence number is wrapped around Some() since Some is a "container" around the sequence number, so in the event that the sequence number did not arrive on the server side, it will default to **None**, meaning the packet was not sent successfully and re-transmission is required, that will handle the loss of packets and simply recover by resending the same packet, ensuring the proper sequnece number is recieved on the server side.

Next, we start the polling session for the socket just like before, while also using the time library to start from the current timestamp, the sequence number gets increment by 1, since that will be the new sequence number for the subsequent packet, and the socket waits and checks if it recieves a response. The timestamp is being kept track of to calculate the RTT value of sending the ping message and recieving back pong message. If some form of socketEvent does occur, we check if the recieved packet happens to contains the same server address as the server itself, if so, that means we have recieved a response from the server. The response also contains a payload (the payload contains the "Pong" message), which is then extracted and validated to ensure there are no invalid UTF-8 values, the message from the payload (which is Pong, as that's what was configured on the **server()**) function is displayed on the terminal alongside the time that was elapsed (which is the RTT) and the sequnce number based on the current iteration. Otherwise, if the address doesn't match the server, rather than printing out the message, RTT and sequence number, we instead simply print out **"unknown sender"** on the terminal, because we don't want to print out/acknowledge the response from anything other than the server socket we created. The code for this is the following:

```
seq = seq + 1;
            match socket.recv() {
                Some(SocketEvent::Packet(packet)) => {
                    if packet.addr() == server {
                        // prints out the message recieved on the server side
                        // handles packet loss and reconstructs packets as needed
```

```
                    // unpack what the server sent
                    // server should respond with Ping
                    println!(
                        "{}, {} RTT {:?}",
                        String::from_utf8_lossy(packet.payload()),
                        i,
                        // check the timestamp it took to send the message
                        now.elapsed()
                    );
                } else {
                    // if sender cannot be verified
                    // print out unknwon sender
                    println!("Unknown sender.");
                }
            }
```

Lastly, we simply return an **Ok(())** indicating successful execution of the function. The complete code for the client() side is the following:

```
fn client(number_of_requests: u8) -> Result<(), ErrorKind> {
    // this is the client side socket/ip
    // @127.0.0.1 : is a special ip address known as "the loopback address"
    // it is used by the computer to refer to itself
    // @:* represents the port number, this can be any available port within the
operating system
    let addr = "127.0.0.1:4000";
    let mut socket = Socket::bind(addr)?;
    println!("Connected on {}", addr);
    let mut seq = 1;

    // when we bind a socket
    // the address gets "wrapped" around, essentially imagine wrapping a physical
object with a gift wrapper or placing a content withing a container
    // therefore, we need to "unwrap" it
    // thus the use of unwrap()
    // parse() helps convert
    let server = SERVER.parse().unwrap();

    println!("Type a message and press Enter to send. Send `Bye!` to quit.");

    let stdin = stdin();
    let mut s_buffer = String::new();
    loop {
        s_buffer.clear();
        stdin.read_line(&mut s_buffer)?;
        let mut line = s_buffer.replace(|x| x == '\n' || x == '\r', "");
        // send 10 ping messages back to back
        // it's not limited to just ping messages
        // after sending the specified number of requests here back to back
        // the server will be sending waiting for acknowledgement for
        for i in 1..number_of_requests + 1 {
            // start the timer at the beggining of each iteration
            let now = Instant::now();
            let string = i.to_string();
            //line.push_str(&string);

            // send reliable sequence data
```

```
            socket.send(Packet::reliable_sequenced(
                server,
                // creates a copy of the string, as the name implies
                // converts it into bytes
                // so that it can be sent over the network to the server port
                line.clone().into_bytes(),
                Some(i),
            ))?;

            socket.manual_poll(Instant::now());

            // if user inputs Bye!
            // no message gets sent
            if line == "Bye!" {
                break;
            }

            seq = seq + 1;
            match socket.recv() {
                Some(SocketEvent::Packet(packet)) => {
                    if packet.addr() == server {
                        // prints out the message recieved on the server side
                        // handles packet loss and reconstructs packets as needed
                        // unpack what the server sent
                        // server should respond with Ping
                        println!(
                            "{}, {} RTT {:?}",
                            String::from_utf8_lossy(packet.payload()),
                            i,
                            // check the timestamp it took to send the message
                            now.elapsed()
                        );
                    } else {
                        // if sender cannot be verified
                        // print out unknwon sender
                        println!("Unknown sender.");
                    }
                }

                // specify what to do if the client times out, there could be
instance message has been sent but the connection may have been lost after
establishment
                // if so, cnnection will be re-established and message will be sent
                // ensuring that the server sends the appropriate response back
                // the connection will timeout if the thread remains idle for too
long
                // the disconnect method from the library isn't entirely functional
                Some(SocketEvent::Disconnect(_)) => {}
                _ => println!("Pong! {:?}, RTT : {:?}", i, now.elapsed()),
            }
        }
    }

    // this must be returned at the end of the function execution
    // otherwise the compiler will panic
```

```
    Ok(())
}
```

Lastly, now that we have defined our client and server function, it is a matter of connecting them with one another, meaning ensuring that a connection has been established with one another. To do this, we prompt the user on whether they want to start up the client or server, the user input is saved on a mutable value of owned string type, and we simply check the first character of the user input, if it starts with an **s**, then we start up the server, meaning the **server()** function gets executed, otherwise, we start up the **client()** instead. The return type for **main()** is the same as the `client()` and `server()` functions.

The code for the main function is the following:

```
fn main() -> Result<(), ErrorKind> {
    // used to take in user input
    // store the user input within the variable stdin
    // immutable by default
    let stdin = stdin();

    // prompt the user to type in whether they want to start client or server
    println!("Please type in `server` or `client`.");

    let mut s = String::new();
    stdin.read_line(&mut s)?;

    // basic conditional statement to check if we should start server or client
    // we only have to check the first letter of the user input
    // if it doesn't start with an s
    // start the client instance instead
    if s.starts_with('s') {
        println!("Starting server at port 3000...");
        server()
    } else {
        println!("Starting client at port 4000...");
        // the value sepcified within client is used to determine how many requests
should be sent back-to-back
        client(10)
    }
}
```

The demo (gif) can be checked in the link provided below: (it's part of the github repository where the code is hosted):

Reliable UDP Demo

Link to repository, please refer to the READ ME for instructions on how to get the code up and running on local computer if interested:

Github Repository Link