Rust Port Scanner: Detailed Code Documentation

This document provides a comprehensive, line-by-line explanation of the given Rust code, which implements a basic multi-threaded port scanner.

1. Introduction

This Rust program functions as a simple command-line port scanner. It takes an IP address as input and, optionally, the number of threads to use for scanning. It attempts to connect to a range of TCP ports on the specified IP address to identify which ones are open.

2. Overall Structure

The code is composed of the following main parts:

- **use statements**: Imports necessary modules from the Rust standard library.
- **const MAX**: Defines a constant for the maximum port number to scan.
- **struct Arguments**: Defines a structure to hold parsed command-line arguments (IP address and number of threads).
- **impl Arguments**: Implements methods for the Arguments struct, specifically new for parsing command-line input.
- **fn scan**: The core function that performs the port scanning logic for a given range of ports within a thread.
- **fn main**: The entry point of the program, responsible for argument parsing, thread creation, and collecting/displaying results.

3. Line-by-Line Explanation

3.1. Module Imports (use statements)

```
use std::env;
use std::io::{self, Write};
use std::net::{IpAddr,TcpStream};
use std::str::FromStr;
use std::process::exit;
use std::sync::mpsc ::{Sender, channel};
use std::thread;
```

- use std::env;: Imports the env module, which provides functions to interact with the environment, such as accessing command-line arguments (env::args()).
- use std::io::{self, Write};: Imports the io module for input/output operations.

Specifically, self brings std::io itself into scope, and Write brings the Write trait (used for flushing output to the console).
- use std::net::{IpAddr, TcpStream};: Imports types from the net module.
  - IpAddr: Represents an IP address (IPv4 or IPv6).
  - TcpStream: Represents a TCP connection, used for attempting to connect to ports.
- use std::str::FromStr;: Imports the FromStr trait. This trait allows converting a string into another type, which is used here to parse a string into an IpAddr.
- use std::process::exit;: Imports the exit function, used to terminate the program with a specified exit code.
- use std::sync::mpsc::{Sender, channel};: Imports components for message passing between threads.
  - mpsc: Stands for "Multiple Producer, Single Consumer".
  - Sender: The sending half of a channel.
  - channel: A function that creates a new (Sender, Receiver) pair.
- use std::thread;: Imports the thread module, which provides functions for working with threads, such as thread::spawn to create new threads.

3.2. Constants

```
const MAX: u16 = 65535;
```

- const MAX: u16 = 65535;: Defines a constant named MAX with the type u16 (an unsigned 16-bit integer). This constant represents the maximum possible TCP port number (2^16 - 1), indicating the upper limit for the port scan.

3.3. Arguments Struct

```
struct Arguments {
   ipaddr: IpAddr,
   threads: u32,
}
```

- struct Arguments { ... }: Defines a structure named Arguments. This struct is used to encapsulate the command-line arguments that the program needs:
  - ipaddr: IpAddr: Stores the target IP address, using the IpAddr type.
  - threads: u32: Stores the number of threads to use for scanning, as an unsigned 32-bit integer.

3.4. impl Arguments Block (Argument Parsing)

```
impl Arguments {
    fn new(args: &[String]) -> Result<Arguments, &'static str> {
        if args.len() < 2 {
            return Err("Not enough Arguments");
        } else if args.len() > 4 {
            return Err("Too many Arguments");
        }
        let f = args[1].clone();
        if let Ok(ipaddr) = IpAddr::from_str(&f) {
            return Ok(Arguments { ipaddr, threads: 4 });
        } else {
            let flag = args[1].clone();
            if (flag.contains("-h") || flag.contains("-help")) && args.len() == 2 {
                println!("usage: -j to select the number of threads you want
                \r\n      -h or -help to this help message");
                return Err("Help requested");
            } else if flag.contains("-h") || flag.contains("-help") {
                return Err("Too many Arguments");
            } else if flag.contains("-j") {
                if args.len() < 4 {
                    return Err("Not enough arguments for -j flag");
                }
                let threads = match args[2].parse::<u32>() {
                    Ok(s) => s,
                    Err(_) => return Err("Failed to parse the number of threads"),
                };
                let ipaddr = match IpAddr::from_str(&args[3]) {
                    Ok(s) => s,
                    Err(_) => return Err("Not a Valid Ip Address it must be IPv4 or IPv6 format"),
                };
                return Ok(Arguments { threads, ipaddr });
            } else {
                return Err("Invalid Arguments passed");
            }
        }
    }
}
```

- impl Arguments { ... }: This block implements methods associated with the Arguments struct.
- fn new(args: &[String]) -> Result<Arguments, &'static str>: This is an associated function (like a static method in other languages) for Arguments. It's responsible for parsing the raw command-line arguments (passed as a slice of Strings) and constructing an Arguments instance.
  - It returns a Result<Arguments, &'static str>. Result is an enum that represents either success (Ok(value)) or failure (Err(error_value)). Here, Ok contains the parsed Arguments, and Err contains a static string literal describing the error.
- if args.len() < 2 { ... }: Checks if there are at least two arguments (program name + at least one user-provided argument). If not, it's "Not enough Arguments".
- else if args.len() > 4 { ... }: Checks if there are more than four arguments. If so, it's "Too many Arguments".
  - *Note*: The maximum of 4 arguments corresponds to program_name -j num_threads ip_address.
- let f = args[1].clone();: Clones the second argument (index 1), which could be an IP address or a flag.
- if let Ok(ipaddr) = IpAddr::from_str(&f) { ... }: Attempts to parse f directly as an IpAddr.
  - IpAddr::from_str(&f): Tries to convert the string f into an IpAddr using the FromStr trait. This returns a Result.
  - if let Ok(ipaddr) = ...: This is a pattern matching syntax. If the Result is Ok, it extracts the IpAddr value into the ipaddr variable.
  - If successful, it returns Ok(Arguments { ipaddr, threads: 4 }), meaning the IP was provided directly, and the default 4 threads will be used.
- else { ... }: If f (the second argument) is not a valid IP address, it proceeds to check for flags.
  - let flag = args[1].clone();: Clones the argument again, treating it as a potential flag.
  - if (flag.contains("-h") || flag.contains("-help")) && args.len() == 2 { ... }: Checks for -h or -help flag when *only* two arguments are provided (program name + flag).
    - println!: Prints the usage message to the console.
    - return Err("Help requested");: Returns an error indicating help was requested. This error is specifically caught in main to exit(0).
  - else if flag.contains("-h") || flag.contains("-help") { ... }: If -h or -help is present but there are *more* than two arguments, it's an error ("Too many Arguments").
  - else if flag.contains("-j") { ... }: Checks for the -j flag, which is used to specify the number of threads.

        - if args.len() < 4 { ... }: Requires at least 4 arguments (program_name -j num_threads ip_address).
        - let threads = match args[2].parse::<u32>() { ... };: Attempts to parse the third argument (args[2]) as a u32 for the number of threads.
            - match ...: This is another pattern matching construct.
            - Ok(s) => s: If parsing is successful, assign the parsed value s to threads.
            - Err(_) => return Err(...): If parsing fails, return an error message.
        - let ipaddr = match IpAddr::from_str(&args[3]) { ... };: Attempts to parse the fourth argument (args[3]) as an IpAddr.
            - Similar match logic as above for error handling.
        - return Ok(Arguments { threads, ipaddr });: If all parsing for -j is successful, return the Arguments struct with the specified threads and IP.
    - else { return Err("Invalid Arguments passed"); }: If none of the above conditions match, it means an unrecognized argument or flag was provided.

3.5. scan Function (Port Scanning Logic)

```
fn scan(tx: Sender<u16>, start_port: u16, addr: IpAddr, num_threads: u16) {
  let mut port: u16 = start_port + 1;
  loop {
    match TcpStream::connect((addr, port)){
      Ok(_) => {
        print!(".");
        io::stdout().flush().unwrap();
        tx.send(port).unwrap();
      }
      Err(_) => {}
    }
    if (MAX - port) <= num_threads {
      break;
    }
    port += num_threads;
  }
}
```

- fn scan(tx: Sender<u16>, start_port: u16, addr: IpAddr, num_threads: u16): This function performs the actual port scanning. It's designed to be run in a separate thread.

  - tx: Sender<u16>: The sender half of the MPSC channel. This allows the thread to send open port numbers back to the main thread.
  - start_port: u16: The initial port number from which this thread should start scanning. Each thread will have a unique start_port (e.g., thread 0 starts at port 0, thread 1 at port 1, etc.).
  - addr: IpAddr: The target IP address to scan.
  - num_threads: u16: The total number of threads being used in the scan. This is used to determine the step size for scanning (explained below).
- let mut port: u16 = start_port + 1;: Initializes a mutable port variable. It starts scanning from start_port + 1 because the loop condition port += num_threads will effectively start from start_port if the first port is start_port. For example, if start_port is 0 and num_threads is 4, the first thread will check 1, 5, 9...
- loop { ... }: This creates an infinite loop that continues until explicitly break'd out of.
- match TcpStream::connect((addr, port)) { ... }: Attempts to establish a TCP connection to the specified addr and port.
  - TcpStream::connect is a non-blocking operation on some systems or with specific configurations, but generally, it will block until a connection is established or fails. For port scanning, a brief timeout would typically be added in a more robust scanner, but for this example, it's a direct attempt.
  - This returns a Result.
  - Ok(_): If the connection is successful (meaning the port is open).
    - print!(".");: Prints a dot to the console to indicate progress.
    - io::stdout().flush().unwrap();: Flushes the standard output buffer. This ensures that the dot is immediately visible on the console, rather than being buffered until a newline character or the buffer is full. unwrap() will panic if flushing fails, which is generally acceptable for simple console output.
    - tx.send(port).unwrap();: Sends the port number (which is open) through the channel back to the main thread. unwrap() will panic if sending fails (e.g., if the receiver has been dropped).
  - Err(_): If the connection fails (meaning the port is likely closed or filtered).
    - {}: Does nothing; the error is ignored.
- if (MAX - port) <= num_threads { break; }: This is the loop termination condition.
  - MAX - port: Calculates how many ports are left to scan from the current port up to MAX.
  - If the remaining ports are fewer than or equal to the total num_threads, it means all threads will have processed their assigned ranges soon, so this thread can stop. This prevents threads from trying to scan beyond MAX and

helps distribute the remaining ports evenly as the scan approaches the end.

- port += num_threads;: Increments the port number by num_threads. This is the crucial part for distributing the work across threads. If num_threads is 4, thread 0 scans ports 1, 5, 9, ...; thread 1 scans 2, 6, 10, ...; and so on. This ensures each thread scans a distinct set of ports without overlap.

3.6. main Function (Program Entry Point)

```
fn main() {
    let args: Vec<String> = env::args().collect();
    let program = args[0].clone();
    let arguments = Arguments::new(&args).unwrap_or_else(
        |err| {
            if err.contains("help") {
                exit(0);
            } else {
                eprintln!("{} Problem parsing the arguments: {}", program, err);
                exit(0);
            }
        }
    );
    let num_threads = arguments.threads;
    let addr = arguments.ipaddr;
    let (tx, rx) = channel();
    for i in 0 .. num_threads {
        let tx = tx.clone();
        thread::spawn(move  || {
            scan(tx, i as u16, addr, num_threads as u16);
        });
    }

    let mut out = vec![];
    drop (tx);
    for p in rx {
        out.push(p);// Close the sender to stop the loop when all threads are done
    }
    println!("");
    out.sort();
    for v in out {
        println!("Port {} is open", v);
    }
}
```

- fn main() { ... }: The main function, where program execution begins.
- let args: Vec<String> = env::args().collect();:
  - env::args(): Returns an iterator over the command-line arguments. The first argument is typically the program's name.
  - .collect(): Collects the items from the iterator into a Vec<String> (a dynamically sized list of strings).
- let program = args[0].clone();: Stores the program's name (the first argument) in the program variable.
- let arguments = Arguments::new(&args).unwrap_or_else(...);: Calls the new function of the Arguments struct to parse the command-line arguments.
  - &args: Passes a reference to the args vector.
  - .unwrap_or_else(|err| { ... }): This is a common way to handle Result types. If Arguments::new returns Ok, its value is unwrapped and assigned to arguments. If it returns Err, the provided closure (the |err| { ... }) is executed with the error message (err).
    - if err.contains("help") { exit(0); }: If the error message indicates "help requested", the program exits gracefully with status code 0.
    - else { eprintln!("{} Problem parsing the arguments: {}", program, err); exit(0); }: For any other parsing error, it prints an error message to standard error (eprintln!) along with the program name and then exits with status code 0.
- let num_threads = arguments.threads;: Extracts the number of threads from the parsed arguments.
- let addr = arguments.ipaddr;: Extracts the target IP address from the parsed arguments.
- let (tx, rx) = channel();: Creates a new MPSC channel.
  - tx: The Sender end. This will be cloned and given to each scanning thread.
  - rx: The Receiver end. This stays in the main thread to collect results.
- for i in 0 .. num_threads { ... }: Loops num_threads times to create and spawn each scanning thread.
  - let tx = tx.clone();: Clones the Sender. Each thread needs its own Sender instance to send messages independently.
  - thread::spawn(move || { ... });: Spawns a new thread.
    - move || { ... }: This is a closure (an anonymous function). The move keyword indicates that the closure takes ownership of any variables it

        captures from its environment (tx, addr, num_threads in this case). This is essential for safe multi-threading in Rust.
        - scan(tx, i as u16, addr, num_threads as u16);: Calls the scan function within the new thread, passing its specific Sender, its starting port offset (i as a u16), the target IP address, and the total number of threads.
- let mut out = vec![];: Initializes an empty, mutable vector named out. This vector will store the open port numbers discovered by the scanner.
- drop (tx);: **Crucial for terminating the receiver loop.** This explicitly drops the *original* tx (sender) handle in the main thread. When all other tx clones (held by the spawned threads) are also dropped (i.e., when all threads finish execution), the rx (receiver) end will know that no more messages will be sent.
- for p in rx { out.push(p); }: Iterates over the rx (receiver).
    - This loop will block and wait for messages (open port numbers) to be sent from the scan threads.
    - It continues to receive messages until all Sender instances connected to this Receiver have been dropped. Once the last Sender (including the original tx dropped above) is dropped, the rx iterator will complete, and the loop will terminate.
- println!("");: Prints an empty line for better formatting, separating the progress dots from the final results.
- out.sort();: Sorts the collected open port numbers in ascending order.
- for v in out { println!("Port {} is open", v); }: Iterates through the sorted list of open ports and prints each one to the console.

4. How to Use/Run

To compile and run this program:

1. **Save the code**: Save the code into a file named main.rs.
2. **Compile**: Open a terminal or command prompt, navigate to the directory where you saved main.rs, and compile the code using Rust's package manager, Cargo:
   ```
   rustc main.rs
   ```
   This will create an executable file (e.g., main on Linux/macOS, main.exe on Windows).
3. **Run**:
    - **Basic scan (default 4 threads)**:
      ```
      ./main <IP_Address>
      ```
      Example:
      ```
      ./main 127.0.0.1
      ```

- **Specify number of threads**:
  ./main -j <num_threads> <IP_Address>

  Example (scan 192.168.1.1 with 20 threads):
  ./main -j 20 192.168.1.1

- **Help message**:
  ./main -h

  or
  ./main -help

5. Conclusion

This Rust program effectively demonstrates concurrent programming using threads and message passing (MPSC channels) to perform a basic TCP port scan. It provides a command-line interface for specifying the target IP and the number of threads, offering a foundational understanding of network scanning techniques in Rust.