

Rust for Network Servers

Synchronous and asynchronous network communication

Simon Pannek

Technical University of Munich
Munich, Germany

June 25, 2021

Introduction

Back in the 1960s, the protocols for one of the first computer networks were developed.

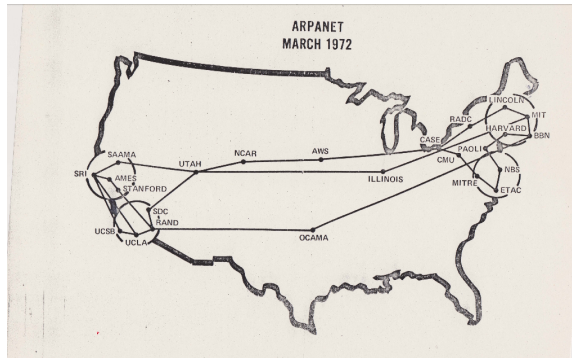


Figure: Ferris the Crab [1]

2 / 31

Notes:

1. Many modern devices used today are using the internet to communicate with each other
2. A foundation for this can be found when looking at the ARPANET
3. ARPANET: A packet-switching network deployed in the US
 - 3.1 One of the first networks to implement the TCP/IP protocol suite
 - 3.2 Evolved into the Internet as we know it today

What happened until now?

1. Additional security (encryption)

What happened until now?

1. Additional security (encryption)
2. Introduction of new protocols

What happened until now?

1. Additional security (encryption)
2. Introduction of new protocols
3. A lot was standardized (IEEE standards, W3C, ...)

What programming language to choose?

4 / 31

Notes:

1. Standardization → common used programming language usually support frequently deployed networking protocols
2. Depends on the use case: Certain trade offs comparing different languages
3. We are going to take a look at Rust and what it has to offer when it comes to running it as a backend of a network application

```
void unsecure_function(void) {  
    char buf[512];  
    read_from_network(buf);  
  
    ...  
}
```

Notes:

1. Let us take a step back: Currently, many networking libraries are written in C or C++
2. Is there any problem with this piece of code?
3. Buffer overflow attacks (Small explanation what can happen)
4. No guaranteed memory safety in C and C++
5. Mistakes of a programmer are not checked by the compiler and fall back to some default action (UB → corruption of memory)
6. Problematic for network applications (many concurrent reads and writes to buffers)

Buffer Overflow Attacks

```
void unsecure_function(void) {  
    char buf[512];  
    read_from_network(buf);  
  
    ...  
}
```

5 / 31

Notes:

1. Let us take a step back: Currently, many networking libraries are written in C or C++
2. Is there any problem with this piece of code?
3. Buffer overflow attacks (Small explanation what can happen)
4. No guaranteed memory safety in C and C++
5. Mistakes of a programmer are not checked by the compiler and fall back to some default action (UB → corruption of memory)
6. Problematic for network applications (many concurrent reads and writes to buffers)


```
def more_secure_function(socket):  
    buf = socket.recv(512)  
  
    ...
```

Notes:

1. Simplified example → Depends on the actual use case and implementation whether something can be considered secure
2. Interpreted languages like Python as a solution (background checks for undefined behavior for almost everything)
3. Scripting languages allow fast prototyping
4. Trade off of interpreted languages: Slower performance compared to executing compiled binary files (Instructions first have to get translated by the interpreter)
5. They do not run natively on the computer → you first have to install the interpreter

Interpreted languages

```
def more_secure_function(socket):  
    buf = socket.recv(512)  
  
    ...
```

6 / 31

Notes:

1. Simplified example → Depends on the actual use case and implementation whether something can be considered secure
2. Interpreted languages like Python as a solution (background checks for undefined behavior for almost everything)
3. Scripting languages allow fast prototyping
4. Trade off of interpreted languages: Slower performance compared to executing compiled binary files (Instructions first have to get translated by the interpreter)
5. They do not run natively on the computer → you first have to install the interpreter

Rust as a solution?

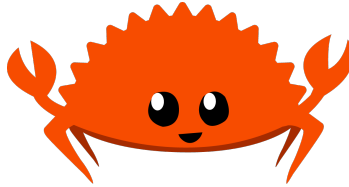


Figure: Ferris the Crab [3]

Notes:

1. Rust can be considered "the best of both worlds":
 - 1.1 Compiled language with a compiler guaranteeing memory- and thread-safety
 - 1.2 Fast speed of compiled languages and prevents most of the vulnerabilities mentioned before through its compiler checks
 - 1.3 Network operations are prone to failure → unsafe calls can be wrapped into enums of the type `std::result::Result`
 - 1.4 Package manager Cargo
2. Let us first take a look how to implement basic TCP and UDP communication in Rust

Ownership, borrowing, and lifetimes

8 / 31

Notes:

1. Ownership, borrowing and lifetimes play an important role in networking code
2. Ownership:
 - 2.1 A part of the code can own a piece of memory
 - 2.2 Function is called with an owned value → the piece of memory gets moved into that function
3. Borrowing:
 - 3.1 No complete access is needed → a reference can be borrowed
 - 3.2 There are also mutable references: Only one is allowed at the same time
 - 3.3 Useful when working with network applications (prevents the occurrence of data races if a thread tries to read or write to a buffer while another thread is already writing to it)
 - 3.4 We are not going to take a look at `std::sync::Arc`
4. Lifetimes:
 - 4.1 Lifetime determined by the code block it was created → variable gets out of scope → the corresponding piece of memory is freed
 - 4.2 If a value gets moved into another function, its lifetime changes with it

Ownership, borrowing, and lifetimes

→ No dangling references, illegal memory access, and memory leaks

8 / 31

Notes:

1. Ownership, borrowing and lifetimes play an important role in networking code
2. Ownership:
 - 2.1 A part of the code can own a piece of memory
 - 2.2 Function is called with an owned value → the piece of memory gets moved into that function
3. Borrowing:
 - 3.1 No complete access is needed → a reference can be borrowed
 - 3.2 There are also mutable references: Only one is allowed at the same time
 - 3.3 Useful when working with network applications (prevents the occurrence of data races if a thread tries to read or write to a buffer while another thread is already writing to it)
 - 3.4 We are not going to take a look at `std::sync::Arc`
4. Lifetimes:
 - 4.1 Lifetime determined by the code block it was created → variable gets out of scope → the corresponding piece of memory is freed
 - 4.2 If a value gets moved into another function, its lifetime changes with it

Ownership, borrowing, and lifetimes

→ No dangling references, illegal memory access, and memory leaks

(Except if you are explicitly working with `unsafe` code)

8 / 31

Notes:

1. Ownership, borrowing and lifetimes play an important role in networking code
2. Ownership:
 - 2.1 A part of the code can own a piece of memory
 - 2.2 Function is called with an owned value → the piece of memory gets moved into that function
3. Borrowing:
 - 3.1 No complete access is needed → a reference can be borrowed
 - 3.2 There are also mutable references: Only one is allowed at the same time
 - 3.3 Useful when working with network applications (prevents the occurrence of data races if a thread tries to read or write to a buffer while another thread is already writing to it)
 - 3.4 We are not going to take a look at `std::sync::Arc`
4. Lifetimes:
 - 4.1 Lifetime determined by the code block it was created → variable gets out of scope → the corresponding piece of memory is freed
 - 4.2 If a value gets moved into another function, its lifetime changes with it

The Transmission Control Protocol (TCP)

Notes:

1. TCP is used for reliable inter-process communication between two systems connected through a network
2. The sequence number can also be used to eliminate duplicates
3. `std::net` module includes networking functions for TCP

The Transmission Control Protocol (TCP)

- Active connection between two systems

Notes:

1. TCP is used for reliable inter-process communication between two systems connected through a network
2. The sequence number can also be used to eliminate duplicates
3. `std::net` module includes networking functions for TCP

The Transmission Control Protocol (TCP)

- Active connection between two systems
- Can be used to send a continuous stream of octets to another host

Notes:

1. TCP is used for reliable inter-process communication between two systems connected through a network
2. The sequence number can also be used to eliminate duplicates
3. `std::net` module includes networking functions for TCP

The Transmission Control Protocol (TCP)

- Active connection between two systems
- Can be used to send a continuous stream of octets to another host
- A sequence number is assigned to each octet in order to confirm it was received

Notes:

1. TCP is used for reliable inter-process communication between two systems connected through a network
2. The sequence number can also be used to eliminate duplicates
3. `std::net` module includes networking functions for TCP

The Transmission Control Protocol (TCP)

- Active connection between two systems
- Can be used to send a continuous stream of octets to another host
- A sequence number is assigned to each octet in order to confirm it was received
- If no acknowledgment (ACK) was received, the data is sent again

Notes:

1. TCP is used for reliable inter-process communication between two systems connected through a network
2. The sequence number can also be used to eliminate duplicates
3. `std::net` module includes networking functions for TCP

TCP client handling

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

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

    match reader.read_line(&mut buf) {
        ...
    }
}
```

TCP client handling

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

TCP listener

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

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

TCP client

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

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

    match stdin().read_line(&mut buf) {
        ...
    }
}
```

TCP client

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


TCP client

```
Ok(_) => {  
    let bytes = buf.as_bytes();  
  
    stream  
        .write_all(bytes)  
        .expect("Writing");  
}
```

The User Datagram Protocol (UDP)

13 / 31

Notes:

1. UDP is less reliable than the Transmission Control Protocol
2. No need to listen for new hosts or actively establish a connection
3. Typical example: Media streaming (package loss is acceptable)
4. `std::net` includes networking functions for UDP

The User Datagram Protocol (UDP)

- Commonly used if you do not need reliable delivery of streams

Notes:

1. UDP is less reliable than the Transmission Control Protocol
2. No need to listen for new hosts or actively establish a connection
3. Typical example: Media streaming (package loss is acceptable)
4. `std::net` includes networking functions for UDP

The User Datagram Protocol (UDP)

- Commonly used if you do not need reliable delivery of streams
- Tries to send messages to other programs with a minimal amount of protocol mechanism

Notes:

1. UDP is less reliable than the Transmission Control Protocol
2. No need to listen for new hosts or actively establish a connection
3. Typical example: Media streaming (package loss is acceptable)
4. `std::net` includes networking functions for UDP

The User Datagram Protocol (UDP)

- Commonly used if you do not need reliable delivery of streams
- Tries to send messages to other programs with a minimal amount of protocol mechanism
- No protection against package duplication

Notes:

1. UDP is less reliable than the Transmission Control Protocol
2. No need to listen for new hosts or actively establish a connection
3. Typical example: Media streaming (package loss is acceptable)
4. `std::net` includes networking functions for UDP

The User Datagram Protocol (UDP)

- Commonly used if you do not need reliable delivery of streams
- Tries to send messages to other programs with a minimal amount of protocol mechanism
- No protection against package duplication
- No checks whether the data sent has arrived and if it did in what order

Notes:

1. UDP is less reliable than the Transmission Control Protocol
2. No need to listen for new hosts or actively establish a connection
3. Typical example: Media streaming (package loss is acceptable)
4. `std::net` includes networking functions for UDP

UDP receiver

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

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

    match socket.recv_from(&mut buf) {
        ...
    }
}
```

14 / 31

Notes:

1. Why is the buffer size 1500? (Maximum Transmission Unit)
2. Why is a buffer needed at all? (`std::net::UdpSocket` does not implement the trait `std::io::Read`)

UDP receiver

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

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

14 / 31

Notes:

1. Why is the buffer size 1500? (Maximum Transmission Unit)
2. Why is a buffer needed at all? (`std::net::UdpSocket` does not implement the trait `std::io::Read`)

Networking in Rust

UDP sender

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

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

    match stdin().read_line(&mut buf) {
        ...
    }
}
```

UDP sender

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

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

Asynchronous Networking in Rust

Sequential programming	Asynchronous programming
<ul style="list-style-type: none">- Blocking functions- Result is returned immediately- A new thread is required for each task that should run independently	<ul style="list-style-type: none">- Non-blocking functions- Result is wrapped into futures- A scheduler dynamically assigns tasks to a limited amount of threads

Table: Comparison between sequential and asynchronous programming

Notes:

1. Previous examples: Current thread was blocked when waiting for new connections/incoming messages
2. Sequential programming: Scheduler swaps out the waiting task

The Tokio crate

17 / 31

Notes:

1. Easily imported and managed via Cargo
2. Includes feature flags → specify which parts of the library should be included
3. Parts of tokio:
 - 3.1 I/O event loop: Handles I/O events and dispatches them to the tasks waiting for them
 - 3.2 Timer: Runs tasks after a certain period of time
 - 3.3 Scheduler: Executes the tasks on the different threads
4. Enables the use of `async`, `await` and other features of asynchronous Rust
5. `tokio::net` includes networking functions similar to `std::net`

The Tokio crate

- Tokio is a runtime for writing reliable network applications [2]

Notes:

1. Easily imported and managed via Cargo
2. Includes feature flags → specify which parts of the library should be included
3. Parts of tokio:
 - 3.1 I/O event loop: Handles I/O events and dispatches them to the tasks waiting for them
 - 3.2 Timer: Runs tasks after a certain period of time
 - 3.3 Scheduler: Executes the tasks on the different threads
4. Enables the use of `async`, `await` and other features of asynchronous Rust
5. `tokio::net` includes networking functions similar to `std::net`

The Tokio crate

- Tokio is a runtime for writing reliable network applications [2]
- Tokio consists of three major parts:

17 / 31

Notes:

1. Easily imported and managed via Cargo
2. Includes feature flags → specify which parts of the library should be included
3. Parts of tokio:
 - 3.1 I/O event loop: Handles I/O events and dispatches them to the tasks waiting for them
 - 3.2 Timer: Runs tasks after a certain period of time
 - 3.3 Scheduler: Executes the tasks on the different threads
4. Enables the use of `async`, `await` and other features of asynchronous Rust
5. `tokio::net` includes networking functions similar to `std::net`

The Tokio crate

- Tokio is a runtime for writing reliable network applications [2]
- Tokio consists of three major parts:
 1. I/O event loop

Notes:

1. Easily imported and managed via Cargo
2. Includes feature flags → specify which parts of the library should be included
3. Parts of tokio:
 - 3.1 I/O event loop: Handles I/O events and dispatches them to the tasks waiting for them
 - 3.2 Timer: Runs tasks after a certain period of time
 - 3.3 Scheduler: Executes the tasks on the different threads
4. Enables the use of `async`, `await` and other features of asynchronous Rust
5. `tokio::net` includes networking functions similar to `std::net`

The Tokio crate

- Tokio is a runtime for writing reliable network applications [2]
- Tokio consists of three major parts:
 1. I/O event loop
 2. Timer

17 / 31

Notes:

1. Easily imported and managed via Cargo
2. Includes feature flags → specify which parts of the library should be included
3. Parts of tokio:
 - 3.1 I/O event loop: Handles I/O events and dispatches them to the tasks waiting for them
 - 3.2 Timer: Runs tasks after a certain period of time
 - 3.3 Scheduler: Executes the tasks on the different threads
4. Enables the use of `async`, `await` and other features of asynchronous Rust
5. `tokio::net` includes networking functions similar to `std::net`

The Tokio crate

- Tokio is a runtime for writing reliable network applications [2]
- Tokio consists of three major parts:
 1. I/O event loop
 2. Timer
 3. Scheduler

Notes:

1. Easily imported and managed via Cargo
2. Includes feature flags → specify which parts of the library should be included
3. Parts of tokio:
 - 3.1 I/O event loop: Handles I/O events and dispatches them to the tasks waiting for them
 - 3.2 Timer: Runs tasks after a certain period of time
 - 3.3 Scheduler: Executes the tasks on the different threads
4. Enables the use of `async`, `await` and other features of asynchronous Rust
5. `tokio::net` includes networking functions similar to `std::net`

Asynchronous Networking in Rust

Future demonstration

```
async fn async_function() {  
    println!("Started task1");  
    sleep(Duration::from_secs(5))  
        .await;  
    println!("Finished task1");  
}
```

18 / 31

Notes:

1. Complete asynchronous function declaration vs. `async` block
2. Execution in blocking environment:
 - 2.1 Started task1
 - 2.2 Finished task1
 - 2.3 Started task2
 - 2.4 Finished task2
3. Execution in non-blocking environment:
 - 3.1 Started task1
 - 3.2 Started task2
 - 3.3 Finished task2
 - 3.4 Finished task1

Asynchronous Networking in Rust

Future demonstration

```
#[tokio::main]
async fn main() {
    let future1 = async_function();

    let future2 = async {
        println!("Started task2");
        sleep(Duration::from_secs(3))
        .await;
        println!("Finished task2");
    };

    join!(future1, future2);
}
```

18 / 31

Notes:

1. Complete asynchronous function declaration vs. `async` block
2. Execution in blocking environment:
 - 2.1 Started task1
 - 2.2 Finished task1
 - 2.3 Started task2
 - 2.4 Finished task2
3. Execution in non-blocking environment:
 - 3.1 Started task1
 - 3.2 Started task2
 - 3.3 Finished task2
 - 3.4 Finished task1

Asynchronous Networking in Rust

Asynchronous TCP client handling

```
tokio::spawn(async move {  
    let mut reader = BufReader::new(&mut stream);  
  
    loop {  
        let mut buf = String::new();  
  
        match reader.read_line(&mut buf).await {  
            ...  
        }  
    }  
  
    Ok::<(), Error>::  
});
```

19 / 31

Notes:

1. `async move` → the asynchronous block will take the ownership of all variables referenced within it
2. Without this all variables would be bound to the scope of the code surrounding it
3. Full ownership of `stream` is required

Asynchronous Networking in Rust

Asynchronous TCP client handling

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

19 / 31

Notes:

1. `async move` → the asynchronous block will take the ownership of all variables referenced within it
2. Without this all variables would be bound to the scope of the code surrounding it
3. Full ownership of `stream` is required

Asynchronous TCP client handling

Asynchronous TCP listener

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

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

    handle_client(stream);
}
```

Asynchronous Networking in Rust

Asynchronous TCP client

```
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 {
        ...
    }
}
```

Asynchronous Networking in Rust

Asynchronous TCP client

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


Asynchronous Networking in Rust

Asynchronous TCP client

```
Ok(_) => {  
    let bytes = buf.as_bytes();  
  
    stream  
        .write_all(bytes)  
        .await?;  
}
```

What about Asynchronous UDP communication?

22 / 31

Notes:

1. Tokio offers `tokio::net::UdpSocket` → works similar to `std::net::UdpSocket`
2. Porting the synchronous UDP implementation to an asynchronous one works similarly to the TCP client
3. UDP does not implement actual connections → waiting for new connections is non-blocking anyway
4. The advantage of connecting to multiple clients for the asynchronous TCP client is not given for UDP
5. Client should also send messages? `std::sync::Arc` could be used → shared ownership to the socket (no mutable reference needed for `send_to` and `recv_from`)

Network Client Showcase

OSI layers

Application Layer
Presentation Layer
Session Layer

Transport Layer

Network Layer
Data Link Layer
Physical Layer

Notes:

1. Layers (just in case):
 - 1.1 Application Layer: Network Process to Application
 - 1.2 Presentation Layer: Data representation and Encryption
 - 1.3 Session Layer: Interhost communication
 - 1.4 Transport Layer: End-to-End connections
 - 1.5 Network Layer: Path Determination and logical addressing
 - 1.6 Data Link Layer: Physical addressing
 - 1.7 Physical Layer: Media, signal and binary transmission
2. Showcase how Tokio can be used together with higher level protocols
3. Hypertext Transfer Protocol (HTTP) for data transmission for the World Wide Web (usually based on TCP)

OSI layers

Application Layer

Presentation Layer

Session Layer

Transport Layer

Network Layer

Data Link Layer

Physical Layer

Notes:

1. Layers (just in case):
 - 1.1 Application Layer: Network Process to Application
 - 1.2 Presentation Layer: Data representation and Encryption
 - 1.3 Session Layer: Interhost communication
 - 1.4 Transport Layer: End-to-End connections
 - 1.5 Network Layer: Path Determination and logical addressing
 - 1.6 Data Link Layer: Physical addressing
 - 1.7 Physical Layer: Media, signal and binary transmission
2. Showcase how Tokio can be used together with higher level protocols
3. Hypertext Transfer Protocol (HTTP) for data transmission for the World Wide Web (usually based on TCP)

Data struct

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

25 / 31

Notes:

1. Crate "Serde" allows for simple serialize and deserialize implementation
2. Convert data instances from and to JSON (JavaScript Object Notation)
3. Required to send objects over HTML

Warp HTTP server

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

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

26 / 31

Notes:

1. Crate "Warp" is a simple web framework built on another crate called "Hyper"
2. Works together with Tokio
3. `warp::Filter`:
 - 3.1 Extracts data from a request, handles the data and sends a response back to the sender of the request
 - 3.2 `and-method`: Chains together two filters
 - 3.3 `map-method`: Takes a function as an argument, receives the extracted data and maps it to another value

Application Layer communication

Request HTTP client

```
let client = request::Client::new();

let response = client
    .post(URL)
    .json(&Data {
        number: 5,
        boolean: true,
    })
    .send()
    .await
    .expect("Sending request");

println!("{}", response.text().await.expect("Get text"));
```

27 / 31

Notes:

1. Crate "Request" enables sending simple HTTP requests
2. Requires some kind of runtime like Tokio

Client response

Received: Data { number: 5, boolean: true }

Rust is a very powerful language for writing network applications

Notes:

1. Today I have shown you how to run Rust on Network Servers
2. Often important to deploy a secure and efficient language as a backend
3. Underlying services can be considered the bottleneck most of the times
4. Slowing down or outages should be avoided
5. Security vulnerabilities can cause interruption of service or accidental disclosure of sensible user data
6. Rust's compile time checks can avoid many common security issues
7. Shifting checks into the process of compiling the code removes checking at runtime, making executables even faster

Are there any questions?

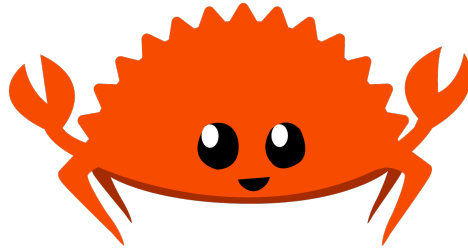



Figure: Ferris the Crab [3]

References

 UCLA and BBN (1972) [1]
The map of ARPANET in March 1972.
https://commons.wikimedia.org/wiki/File:Arpanet_1972_Map.png

 Docs.rs/tokio [2]
Tokio crate documentation.
<https://docs.rs/tokio/1.6.1/tokio/>

 Rustacean.net [3]
Ferris the Crab.
<https://rustacean.net/assets/rustacean-flat-happy.png>