# Distributed Systems Lab 05

## Secure networking

During the previous lab, we presented asynchronous programming in Rust, which should facilitate efficient implementations of single-machine distributed systems. This lab presents network communication in Rust and discusses how to implement a secure communication channel, which is the key component of many multi-machine distributed systems. We will use files contained in this package, which you should download and extract locally.

# Learning section

## Links

**Link** is a name for an abstraction of communication between two distinct nodes of a distributed system. When software runs on more than one machine, the presence of a medium necessary to deliver messages cannot be ignored.

We will look at various types of links between nodes and analyze how they approximate the theory from the lectures.

### Best-effort links

The simplest abstraction, and arguably the closest one to the reality, is a **best-effort link**: when a message is inserted at one end, it either arrives at the other end or gets lost. Such a link can be implemented, for instance, by sending UDP or raw IP packets over the network. You can think of them as of *faulty links*. Note that there are no guarantees about the order in which the packets are delivered, and it might also happen that the same packet is delivered multiple times.

It is rather uncommon to send IP packets directly and thus, we will not discuss this approach.

#### UDP in Rust

**UDP** is supported by the Rust Standard Library. The main type is `std::net::UdpSocket`. It allows you to bind to a certain port and then send and receive messages, as with any other UDP implementation.

We provide some examples in `examples/udp-communication/`.

The interface from the standard library is synchronous. In asynchronous code, you should use `tokio::net`, which largely mimics the interface of `std::net`.

### Reliable links

Often, data delivery guarantees provided by best-effort links are too weak. A **reliable link** works as an invisible pipe, allowing inserting bytes at one of its ends and then reading them at the other end. The bytes are delivered in the order in which they are sent and without any duplicates.

In real distributed systems, reliable links can be approximated using TCP connections. However, it is impossible to implement a truly *reliable* (or *perfect*) link due to the possibility of technical issues. For example, no protocol can circumvent a cut Ethernet cable.

#### TCP in Rust

Rust supports **TCP** via two structures: `std::net::TcpListener` and `std::net::TcpStream`. The first one allows for binding to a port and has a convenient iterator over incoming connections. An incoming connection is nothing more than a `TcpStream` that allows for transmitting data by reading and writing bytes to it. A `TcpStream` can also be created by connecting to a remote host. This is similar to what other programming languages provide. Additionally, in Rust the connection is automatically closed when the `TcpStream` object is dropped.

Sending and receiving data over a TCP stream in Rust is implemented using the `Read` and `Write` traits from `std::io`. Functions `read` and `write` are the most basic ones and roughly correspond to `recv` and `send` syscalls. They are used by other functions, such as `read_exact`. Keep in mind that `read_to_end` will wait forever if the other side does not close the stream by calling `TcpStream::shutdown` or dropping the `TcpStream` object.

Examples are in `examples/tcp-communication/`.

Again, in asynchronous code, you should use `tokio::net` instead of the aforementioned primitives.

## Security of a system

Security is a feature that can often be overlooked, especially in university assignments. However, in many systems, security is a fundamental requirement, although it is tricky to satisfy. Just one mistake made when designing or implementing the system can render the whole effort pointless and allow an attacker to get access to some confidential information.

While this course is not about security, this lab will be dedicated to this topic. Any problem complex enough to require an implementation as a distributed system is likely to have very strict requirements regarding security. This might be because:

- the system is geo-distributed (or decentralized) and exchanges messages over the Internet,
- the system processes sensitive data about numerous individuals,
- the system can cause physical damage.

The remaining part of this lab will be focused on securing message exchanges between components of a system.

### Secure links

The basic link abstraction does not provide any kind of encryption. However, it is possible to build a **secure link** on top of an insecure one. One solution would be to use **public key cryptography**—for instance RSA—to encrypt messages with the public key of the recipient. However, this approach is too slow: RSA encryption is computationally expensive and allows for encryption of only messages shorter than the key. A common way of dealing with those issues is to use RSA to only securely exchange a key for some **symmetric key cryptography** scheme (e.g., AES) and then to encrypt the

following communication with this symmetric key. This approach results in a lower overhead and simplifies encryption of large portions of data.

### Transport Layer Security

**Transport Layer Security (TLS)** is a standard defined in RFC 8446 (*Request for Comments*) designed to provide secure links over insecure ones. Its latest version is 1.3 (as of October 2023), and this is what we will use during this lab, via the `rustls` crate. TLS is a client-server communication scheme. To proceed, the client must verify that it really talks with the designated server. That is, the public key of the server must be trusted by the client (you might have already seen some issues with this in the form of a web browser's warning about an *insecure connection*). This raises the issue of certificates.

### Certificates

The **certificate** is a *signed public key* from some public key cryptography scheme (like RSA). You can see one in the `examples/rustls/certs.rs` file. Certificates can be signed with other certificates, creating a **chain of trust**. Clients (e.g., web browsers) trust certificates, which have a known certificate in this chain. In practice, most clients have a built-in list of trusted *root* certificates: the ones that are not signed by other certificates, but are signed by themselves. Clients verify whether a chain of a certificate in question is valid and whether it ends with a trusted root certificate. Moreover, TLS allows optional certification of clients, making the verification process symmetric.

### Rustls

In directory `examples/rustls/`, we present an example how to create a secure TLS connection. The function `StreamOwned::new` is flexible enough to abstract away TCP, as it requires the underlying link to implement only the `Read` and `Write` traits.

Now we will discuss some lower level cryptography issues.

## Message Authentication Code, HMAC

**Message Authentication Code (MAC)** is used to ensure that a message comes from an authorized sender. It takes the form of a tag, a few additional bytes attached to the message. To implement a MAC, a common shared key is needed, and an algorithm that transforms an arbitrary stream of bytes into a fixed size signature using the provided key. Then, the recipient can use the shared key to verify whether the message was actually sent by an authorized sender. MACs also provide an ability to verify the integrity of the received message.

The advantage of MACs over simple integrity checks, such as CRC, is that without knowing the secret key it is infeasible to tamper with the message and obtain a MAC that will then pass the verification. This property, of course, depends on the actual algorithm. There are at least a few of them (you can easily find them on the Internet), but we will use **HMAC** during this course.

An example of HMAC calculation and verification using the `hmac` crate is provided in `examples/hmac.rs`.

## AES

Historically, symmetric key cryptography was developed first. The same secret sequence of bytes, the *key*, is used to both encrypt and decrypt a message. **AES (Advanced Encryption Standard)** is a

scheme of such symmetric encryption. It is very popular and constitutes a good default option. The keys can have sizes of 128, 192 and 256 bits–the longer the key, the harder it is to break the encryption. AES takes as an input 16 bytes, and outputs 16 bytes too.

### RSA

**RSA (Rivest–Shamir–Adleman)** is an algorithm widely used in public key cryptography. It is based on raising a number to large powers, modulo a product of two large primes. The underlying idea is the Euler's theorem about relatively prime numbers. Part of the public key is an exponent *e*. A plaintext is raised to the power of *e*, modulo some number *n*. The private key is mostly another exponent *d*. The ciphertext is then raised to the power of *d*, arriving at the original message (modulo *n*). The underlying math is not that complicated, so if you are interested, you should be able to learn about the details yourself. What is important from our perspective, is that being given a public key, computing a matching private key is considered to be a hard problem. This makes the scheme useful for encryption, and it is safe to publish freely public keys.

What is more, the two steps of RSA can be also executed in the opposite order. Encrypting a message (more precisely: a hash value of it) with a private key *signs* the message. Then decrypting the *signature* with the public key verifies whether it really was generated with the matching private key.

An example of RSA encryption and decryption using the `rsa` crate is provided in `examples/rsa.rs`.

### Encrypting blocks of data, CBC

Cryptography is full of traps. If you need to encrypt a large portion of bytes, but you are able to encrypt only a short chunk of bytes, it might be tempting to just divide the input and encrypt each part separately. However, this strategy is flawed, as can be easily seen when encrypting, for example, an image: after the transformation, the image will be distorted, but its structure will be still visible! Hence, encrypting large portions of data requires employing special algorithms (*modes of operation*), e.g., **CBC**, which xor-s a ciphertext of the previous block with the plaintext of the next block.

An example demonstrating AES-CBC encryption and decryption using the `aes` and `block-modes` crates is provided in `examples/aes_cbc.rs`.

# Small Assignment

Your task is to implement a *secure link* abstraction in Rust. You shall provide two types: `SecureClient` and `SecureServer`. `SecureClient` accepts plaintext data and sends it as encrypted messages over a link to `SecureServer`. `SecureServer` receives the messages (the receiving operation shall block when there are currently no incoming messages) and decrypts them.

Before being encrypted, the message shall be formatted as follows:

- 4 bytes: length (encoded in network byte order) of the message's content (i.e., the size of data passed to the `send_msg()` method of `SecureClient`),
- content of the message (i.e., the data passed to the `send_msg()` method),
- 32 bytes: a HMAC tag (SHA256 based) of the message's content (i.e., of the data passed to the `send_msg()` method).

The messages shall be encrypted with TLS and sent over the link provided as an argument to the `new()` method of `SecureClient`. `SecureServer` decrypts messages and validates their HMAC tags. If the tag is invalid, an error is returned. Otherwise, the server returns the content of the message (i.e.,

without the length and the HMAC tag). Certificates from the `certs` module will be used for testing, just as they are used in file `main.rs`.

You shall use the same hmac key, the one provided in the `new()` methods, for all exchanged messages. Messages sent over the provided link are guaranteed to arrive in the same order on the other side. Use the `hmac` crate for HMAC and `rustls` for TLS encryption. You can assume that the link for raw data never fails. Your solution may assume the input to functions not returning a `Result` is correct, i.e., the code may panic otherwise.

Note: a standard TLS stream already contains MACs internally to guarantee integrity. Therefore, you may consider the additional layer in this task as an alternative client authorization.

# Additional Homework

Play with Wireshark and `tcpdump`. You can read about them below.

For those interested, think how to verify that data is encrypted. How can one gain (some) confidence that a chunk of bytes is indeed encrypted?

For those really interested, read about Diffie–Hellman key exchange, an alternative way of building a secure channel.

## Wireshark

If you work with low level networking code, you may need to observe exactly what messages are exchanged. Operating systems provide various support for monitoring network interfaces for debugging purposes. Wireshark, a well known GUI tool, can be used to this end. However, because we want our examples to be maximally portable, they employ `tshark`, the terminal version of Wireshark. This lab is a good occasion to play with it. For instance, verify that that packets send by your implementation are indeed encrypted. To download Wireshark, visit the Wireshark website. Install also `tshark`, if it is not already bundled with Wireshark.

We do not cover Wireshark GUI here, but there are plenty of resources about it on the Internet if you are interested.

### Privileges for observing packets

Observing raw packets requires certain privileges in the operating system. If you are using your own computer, you can simply run `tshark` as root (administrator), or grant required privileges to a certain user. However, if you are using the students server or lab computers, you are not able to run processes as root there. In that case, you can easily install a virtual machine—preferably with Linux—and run `tshark` in it.

To observe packets on a network interface, running `tshark` as root, you can execute:

```
sudo tshark -i NETWORK_INTERFACE
```

Proper `NETWORK_INTERFACE` can be found via `ifconfig` or `ip a`. You can try first setting the interface which ‘connects’ you to the Internet and then visiting some websites in a web browser—`tshark` will log the network traffic you generate. In further examples, we will use `lo`: the loopback interface. It is the network interface used to send packets to the same host.

To observe TCP packets arriving at port `PORT`, you can execute:

```
tshark -i lo -f "dst port PORT" -n -T fields -e frame.time -e ip.src -e ip.dst -e
 tcp
```

To observer raw packets, for instance to check whether the data looks like it is being encrypted:

```
tshark -x -i lo -f "dst port PORT"
```

Wireshark has a lot of features: it can dump packets to a file and it can open the file later for an offline analysis, it supports very flexible packet filtering, and it can parse plenty of standard protocols. We do not cover all these features here, as during this lab Wireshark will be primarily useful for verification whether your solution indeed encrypts data.

### Tcpdump

On Linux devices (and macOS, but Linux is far more common in production environments) the standard tool for viewing various network packets is `tcpdump`. Its basic usage is similar to `tshark`'s.

---

Authors: F. Plata, K. Iwanicki, M. Banaszek, W. Ciszewski, M. Matraszek.