Porting TCP/IP module (mbed TLS library) to Rust

Aman Choudhary amanc@iisc.ac.in
Shashank Singh shashanksing@iisc.ac.in
Indian Institute of Science, Bangalore, India

Rust is a multi-paradigm programming language designed for performance and safety, especially safe concurrency. Writing extremely fast code with a low memory footprint implicitly means using C or C++. However, using these languages requires us to manage memory manually and be wary of all the ways that can lead to undefined behavior. Rust provides a way around this by enforcing compile time safety guarantees. In this project we have tried to port the TCP/IP module of mbed TLS, a light-weight open source cryptographic and SSL library written in C to Rust.

1 Introduction

The *mbed TLS* library provides a complete protocol implementation for *SSL* and *TLS*. It is composed of seven modules: SSL/TLS communication, TCP/IP communication, Hashing, Random number generation (RNG), Symmetric cipher (Cipher), Public Key cryptography (PK) and X.509 public key infrastructure (X.509). We chose to translate entire TCP/IP module of the library. The TCP/IP module provides a reliable and ordered channel for the SSL/TLS module. It provides functionalities such as: listening on a port, accepting connections over it, reading / writing to that channel, and closing it. Finally, we have written an *Echo Server* and a *Client Application* as a sample use case, which also accomplishes the task of testing the API of the TCP/IP module.

2 TCP/IP Module

2.1 Overview

The TCP/IP module provides Network Sockets Abstraction Layer to integrate Mbed TLS into a BSD-style sockets API. It provides a context based interface for quickly setting up the underlying Transport Layer (TCP or UDP) to be used by application layer. This module is to be consumed by SSL/TLS module.

This module takes a context based approach to provide an equivalent of BSD-style sockets API. For every invoked operation, the context needs to be passed so that the library knows the network connection context which the caller is referring to.

2.2 MbedtlsNetContext Type

This is a wrapper type for sockets. It is an abstraction to remember the context which the user is referring to while doing Transport Layer operations.

2.3 API of the Module

The following is a list of library methods provided by the TCP/IP Module.

- **mbedtls_net_connect**: Initiate a connection with host:port in the specified protocol
- **mbedtls_net_bind**: Create a receiving socket on the provided interface and port in the specified protocol
- **mbedtls_net_accept**: Accept a connection on a listening socket (TCP) or wait for a message on the socket (UDP)
- **mbedtls_net_send**: Send a message in the stream (TCP) or the socket (UDP)
- **mbedtls_net_recv**: Receive a message from the stream (TCP) or call recv on UDP socket
- **mbedtls_net_free**: Gracefully shut down the stream (TCP), nothing needs to be done for UDP
- **mbedtls_net_close**: Simply calls mbedtls_net_free within itself

2.4 Working of the Module

In order to illustrate the architecture of our TCP/IP library, we consider a simple server and client application (Refer *Figure 1* and *Figure 2* for the diagrams).

Server Application

- The server initialises a MbedtlsNetContext, and issues *mbedtls_net_bind* call to bind it to the required port and network interface with the specified Transport Layer Protocol (TCP or UDP). A *TcpListener* or a *UdpSocket* object is allocated and saved in the context by the library.
- The server starts listening on the bound port for active remote connections and can wait for an incoming connection through *mbedtls_net_accept* call.
- In the case of TCP, when a connection request is received, the library creates a dedicated *TcpStream* object to carry out the communication with client. In the case of UDP, the library just clones a new socket and associates it with remote client address and modifies the context.
- The server can attempt to read and write to the connected client by issuing *mbedtls_net_recv* and *mbedtls_net_send* with the context.

Client Application

- The client initialises a MbedtlsNetContext, and issues *mbedtls_net_connect* call to connect to the server by providing its port and IP address along with the

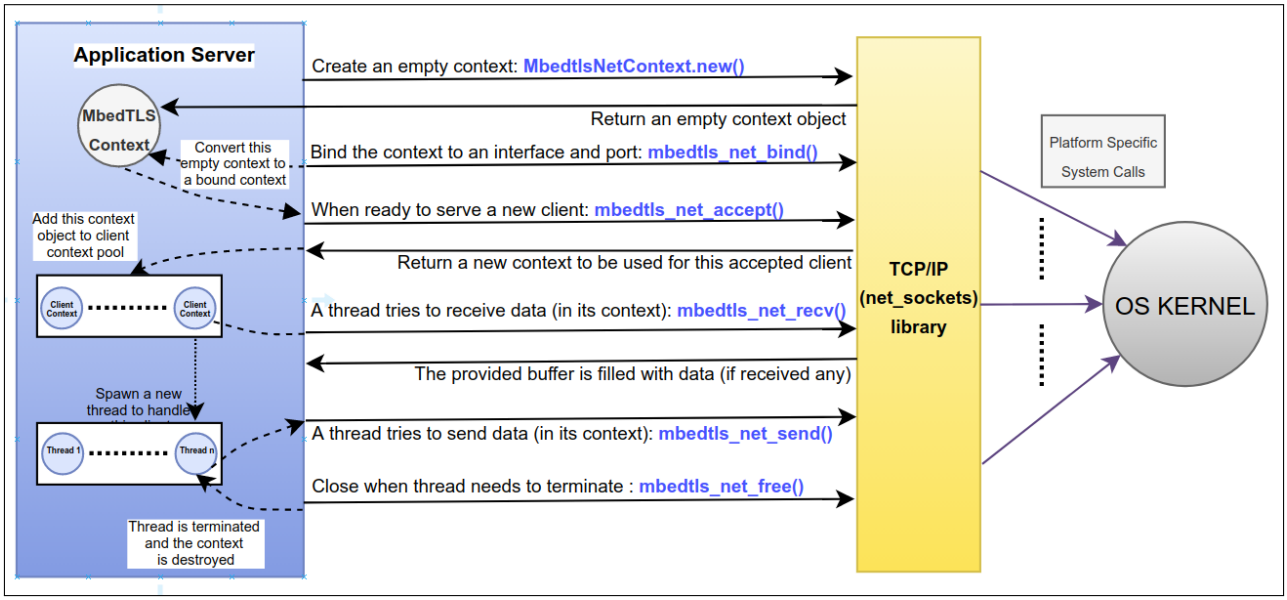

Figure 1: *A diagram illustrating working of a simple server application using the TCP/IP module*

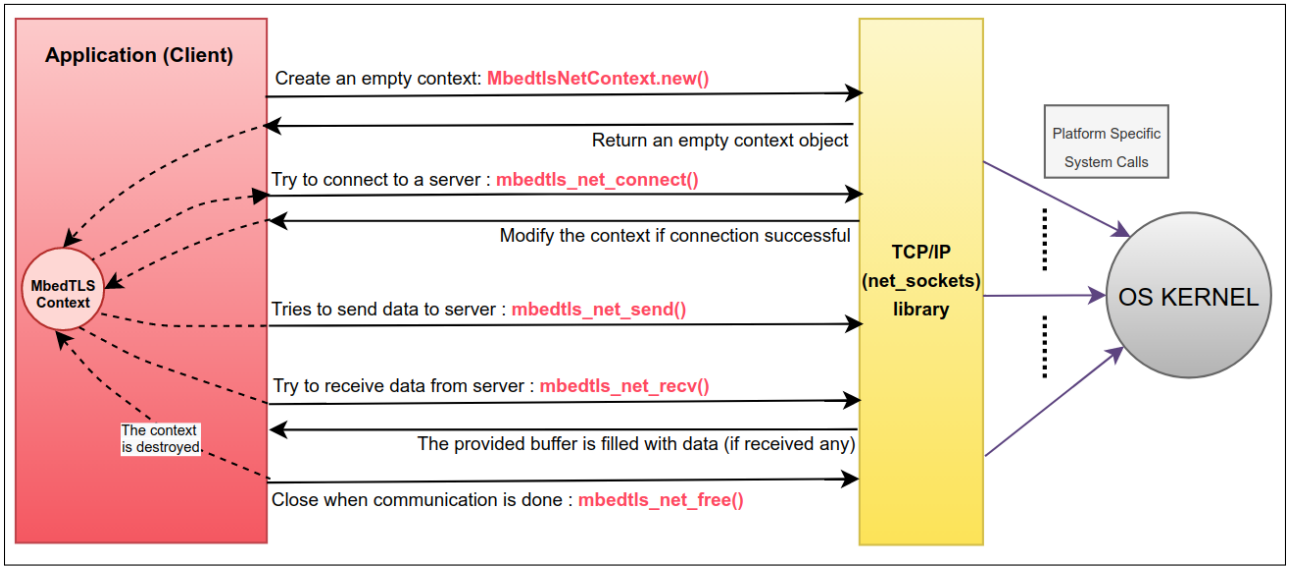

Figure 2: *A diagram illustrating working of a simple client application using the TCP/IP module*

Transport Layer Protocol. A *TcpListener* or a *UdpSocket* object is allocated and saved in the context by the library.

- Once *connect* succeeds, the client application can use the *recv* and *send* library calls to send and receive data to/from the server.

3 Rust Features Leveraged

While porting the module, several features of Rust (some unique to Rust) came in handy. Some features eased the effort while the others gave a guarantee or safety of some kind.

3.1 Ease of Implementation

The rich standard library provided by Rust eases our task as a programmer. One such example is **creation of a socket address structure** in C vs Rust. The Rust code is concise as it uses a standard library function to accomplish the task.

```
// C Code from original library
struct addrinfo hints, *addr_list;
memset( &hints, 0, sizeof( hints ) );
hints.ai_family = AF_UNSPEC;
hints.ai_socktype=proto==MBEDTLS_NET_PROTO_UDP?
                    SOCK_DGRAM : SOCK_STREAM;
hints.ai_protocol=proto==MBEDTLS_NET_PROTO_UDP?
                    IPPROTO_UDP : IPPROTO_TCP;
if(ip_addr==NULL )
    hints.ai_flags=AI_PASSIVE;
if(getaddrinfo(ip_addr,port,&hints,&addr_list)!=0)
    return(MBEDTLS_ERR_NET_UNKNOWN_HOST);

// Rust Code from ported version
let sock_addr = SocketAddr::new(ip_addr, port);
```

3.2 Memory Safety

Rust gives us several features that allow safe memory access. Some of these relevant to the project are as follows.

3.2.1 Implicit Out of Bounds Checks

Rust enforces implicit bound checking at runtime. This means that we don't have to worry about overstepping bounds. However, we do need to handle such a case as crashing is not always the solution.

```
/* In the Echo Server Program, we extract
the arguments from command line. If the user
forgets to provide this, we don't have to
worry about undefined behaviour as the
program will panic and crash */

let args: Vec<String> = env::args().collect();
let proto_str = (&args[1]).to_uppercase();

// If args[1] doesn't exist, we get a runtime
// panic for 'index out of bounds'
```

3.2.2 References preferred over pointers

The library code in C is plagued with pointers. Pointers come with a set of problems like dangling pointers, null pointer dereferencing etc. Rust provides us with references which can help us almost do away with pointers. This saves from a lot of potential vulnerabilities and bugs in the program.

3.2.3 Null Values Not Allowed

Many languages use *null* pervasively to indicate absence/lack of a value. Rust doesn't allow *null* which prevents many associated with it. However, Rust provides *Options* in case a value may be missing and it is safe to use as it enforces exception handling.

```
/* The context contains these fields defined as
Option to indicate that they might not have a
value. Every use of these requires an unwrapping
which enforces checks for whether value exists
*/
pub struct MbedtlsNetContext {
    protocol: Option<TLProtocol>,
    tcp_listener: Option<TcpListener>,
    tcp_stream: Option<TcpStream>,
    tcp_stream_remote_addr: Option<SocketAddr>,
    udp_socket: Option<UdpSocket>,
    udp_socket_remote_addr: Option<SocketAddr>,
}
```

3.3 Platform Independence

Rust compiler uses *LLVM* for target code generation and hence a variety of platforms are supported with fully functional standard library. Due to this, we could eliminate a lot of platform specific code and system calls from the original program. An example of this as follows.

```
/* For Windows Platforms */
#if (defined(_WIN32) || defined(_WIN32_WCE))
    #include <ws2tcpip.h>
    #include <winsock2.h>
    ...
#else /* For Linux Platforms */
    #include <sys/types.h>
    #include <sys/socket.h>
    ...
#endif
```

3.4 Mandatory Exception Handling

Let's assume a use case when a code tries to open a file via the *open* system call. The call may not return a valid file descriptor (by returning -1). In such a case, the code is not obliged to validate the descriptor before any further use, which may lead to undefined behaviour.

Rust has a *Result<T, E>* type, used for returning and propagating errors. It is an *enum* with the variants, *Ok(T)*, representing success and containing a value, and *Err(E)*, representing error and containing an error value. It makes it mandatory to cover the case of an error while unpacking the *Result* object and distinguish the pathways of valid and erroneous result.

```
match TcpStream::connect(sock_addr) {
    Ok(tcp_stream) => {
       println!("Connected to server.\n");
        ...}
    Err(e) => {
       println!("Couldn't connect to server.\n");
        ...}
};
```

3.5 The Rust Borrow Checker

The Borrow Checker is a highlight of Rust. It enforces compile-time memory-safety guarantees by an ownership based approach. It also takes care of automatic resource management. **Ownership Based Resource Management** means that an object is destroyed as soon as there is no owner for it. This means we don't need to worry about freeing resources like memory, threads, file handles, sockets.

As an example from the module, the library creates several sockets and streams contained in the context. As soon as the context is destroyed (when owner of the context goes out of scope), these resources are also automatically freed. So, we don't have to worry about manual resource management and **related leaks**.

3.6 Object Oriented Features

Rust is a multi-paradigm language and offers several OOP features that allow easy and robust development. Absence of these doesn't pose major security risks as such but they are good programming practices.

Access Specifiers : Rust allows access specifiers which enable us to dictate the visibility and privacy of classes, structures, methods, and other members. In the C version,

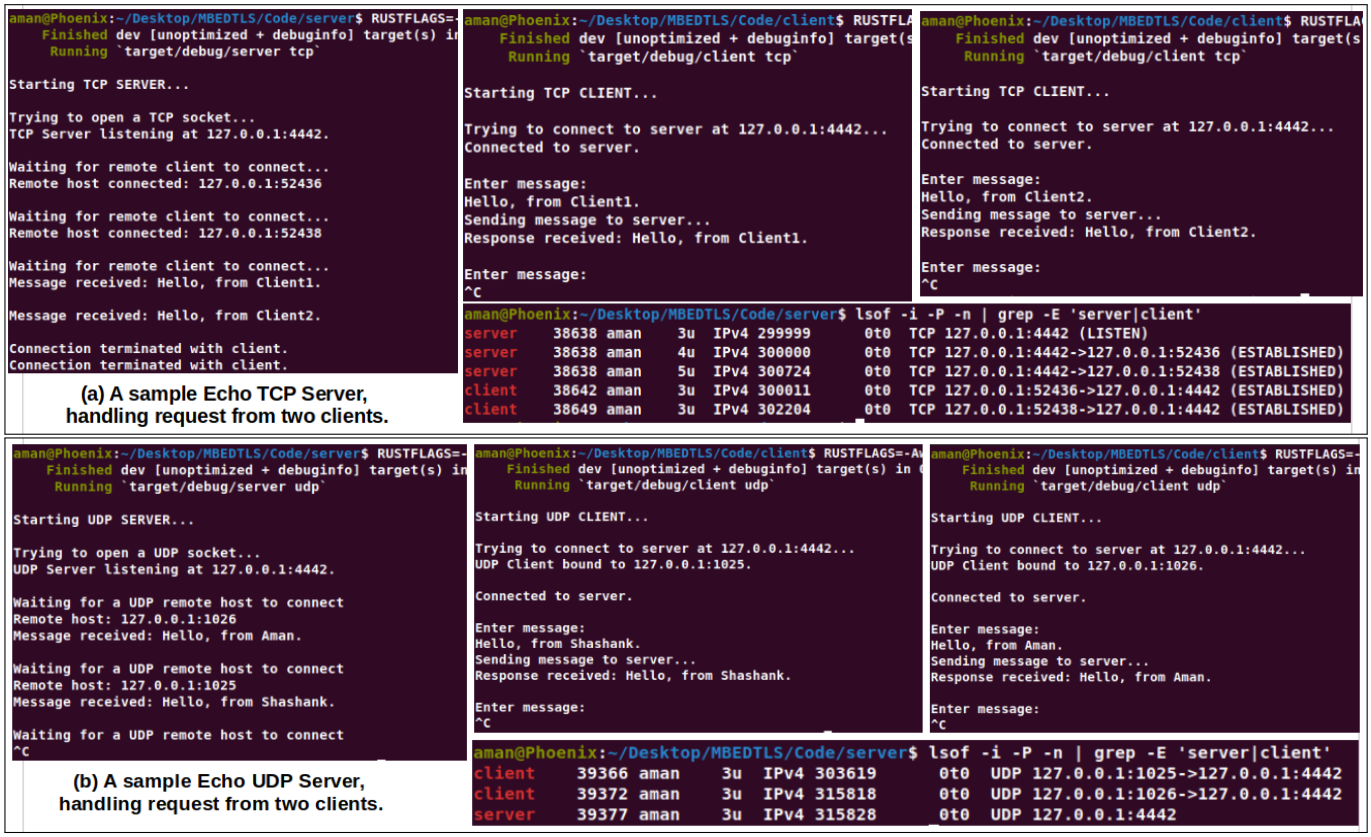

Figure 3: *A Multithreaded Echo Server - Client Communication using the library with (a) TCP and (b) UDP protocols*

the *mbedtls_net_context* contains a file descriptor which can be mistakenly modified in the application using the library. However, our Rust code specifies the members of the context as non-public (Refer code in Section 3.2.3) and hence rendering these members invisible to the application.

Associating behaviours with types : C doesn't allow wrapping code and data together as a single entity. Rust enables this by defining methods using *impl* keyword. We used this to write the constructor for the context type.

```
impl MbedtlsNetContext {
    pub fn new(proto: TLProtocol) -> Self {
        MbedtlsNetContext {
            protocol: Some(proto),
            tcp_listener: None, ....
        }}}
```

4 Challenges Encountered

The road to porting this module was not completely smooth. Some of the challenges we faced are highlighted below.

- We faced challenges in getting used to the new paradigm of *Ownership Based Resource Management* i.e. Rust's most dreaded *Borrow Checker*. We spent a lot of time fixing the error code $E0507$ that said *'Cannot move out of borrowed context'*.
- The original C code uses *file descriptors* in the *mBed TLS context* to store the underlying network sockets. However, Rust doesn't use file descriptors in the networking context. So, we had to redesign the context to suit the Rust way.
- Prima facie, understanding the original implementation was a difficult task. We were looking at the library code-base as a whole. Upon identifying the focus area for this module, things seemed to get clearer and made more sense.

5 Tests and Experiments

As a sanity test, we created an Echo Server and a simple Client Application in Rust using this library. The server simply waits for incoming connections, spawns a thread upon receiving a request and then echoes any message received from the client back to itself. *Figure 3* illustrates a test run for the same. This demo server was created using the same API calls as the original library and hence we can say that the ported library has kept the interfacing of the module intact.

GitHub Repository Link : `https://github.com/TPCSS-mbedTLS-Project-2020-22/mbedtls-source/tree/ac-ss-tcp-ip`

6 Conclusion

Porting this module to Rust was an educational experience. We were exposed to novel ideas on security, resource management and software engineering, some of which are quite unique to Rust. It is amazing to work on a systems programming languages which has a C like performance, yet gives far more guarantees and features. We conclude by a witty slogan that sums up Rust language -

'Rust: fast, reliable, productive — pick three!'

7 References

[1] *mBed TLS Documentation : https://tls.mbed.org/api/*

[2] *Rust Language Documentation : https://doc.rust-lang.org/book/*

[3] *Blog article on Rust's Memory Safety : https://blog.gds-gov.tech/appreciating-rust-memory-safety-438301fee097*

[4] *GitHub Documentation : https://docs.github.com/en*

[5] *Linux man pages : https://man7.org/*

[6] *Blog article on Rust's Borrow Checker : https://blog.logrocket.com/introducing-the-rust-borrow-checker/*