Simulation of TLS (42)

Leskovar Lukas Andreios (KatNr), 5BHIF

March 2022

Contents

1 Introduction

The goal of this project was to simulate communication over Transport Layer Security (TLS) by implementing the Diffie-Hellman Internet Key Exchange (IKE). Any further communication was to be encrypted by a symmetric encryption algorithm.

2 Transport Layer Security

The nowadays de-facto standard for communicating over the internet is Transport Layer Security. Its predecessor SSL was replaced by TLS 1.0 in 1999. In 2018 the newest version 1.3 was implemented which drastically reduced the overhead during handshakes.

2.0.1 Key Generation

2.0.2 Handshake

3 Software Architecture

3.1 Technologies

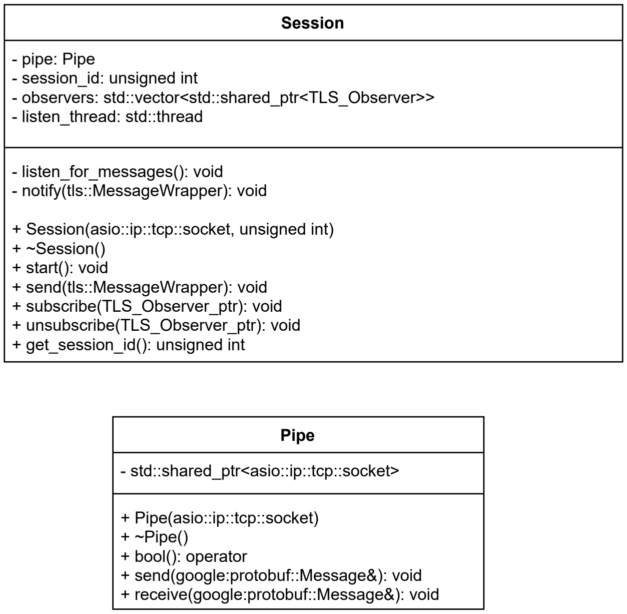
Purpose	Technology
Build Tool	Meson
Command line interface	CLI11
Configuration files	json
Data serialization	Protobuf
Logging	spdlog
Network Communication	asio
Programming Languages	C++ 17
Encryption	plusaes
Hashing	PicoSHA2
Large Integer Values	BigInt

Table 1: This table lists all the technologies used in this project.

3.2 Classes

3.2.1 Communication

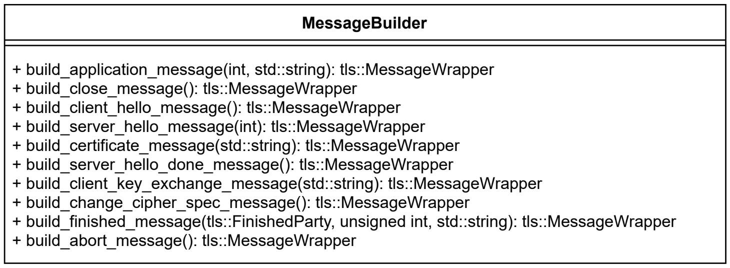
The following classes are utilized by any other part of the application trying to send or receive messages over TCP.



```
Session
- pipe: Pipe
- session_id: unsigned int
- observers: std::vector<std::shared_ptr<TLS_Observer>>
- listen_thread: std::thread

- listen_for_messages(): void
- notify(tls::MessageWrapper): void

+ Session(asio::ip::tcp::socket, unsigned int)
+ ~Session()
+ start(): void
+ send(tls::MessageWrapper): void
+ subscribe(TLS_Observer_ptr): void
+ unsubscribe(TLS_Observer_ptr): void
+ get_session_id(): unsigned int
```

```
Pipe
- std::shared_ptr<asio::ip::tcp::socket>

+ Pipe(asio::ip::tcp::socket)
+ ~Pipe()
+ bool(): operator
+ send(google:protobuf::Message&): void
+ receive(google:protobuf::Message&): void
```

Figure 1: UML Diagram on the Session and Pipe classes used for communication.



```
MessageBuilder

+ build_application_message(int, std::string): tls::MessageWrapper
+ build_close_message(): tls::MessageWrapper
+ build_client_hello_message(): tls::MessageWrapper
+ build_server_hello_message(int): tls::MessageWrapper
+ build_certificate_message(std::string): tls::MessageWrapper
+ build_server_hello_done_message(): tls::MessageWrapper
+ build_client_key_exchange_message(std::string): tls::MessageWrapper
+ build_change_cipher_spec_message(): tls::MessageWrapper
+ build_finished_message(tls::FinishedParty, unsigned int, std::string): tls::MessageWrapper
+ build_abort_message(): tls::MessageWrapper
```

Figure 2: The MessageBuilder is a utility class consisting of multiple static methods building Protobuf Messages.

3.2.2 Main Logic

To notify any main classes of new messages the observer pattern is implemented. The following classes each implement the TLS_Observer and get notified by their respective session object once any messages are receives.

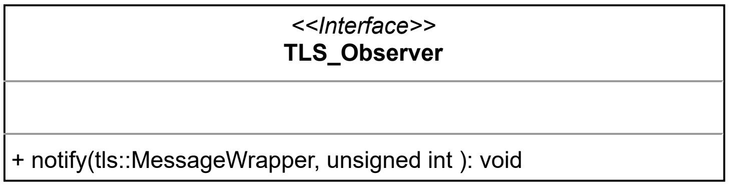


Figure 3: The TLS_Observer class implemented by any receiving class.

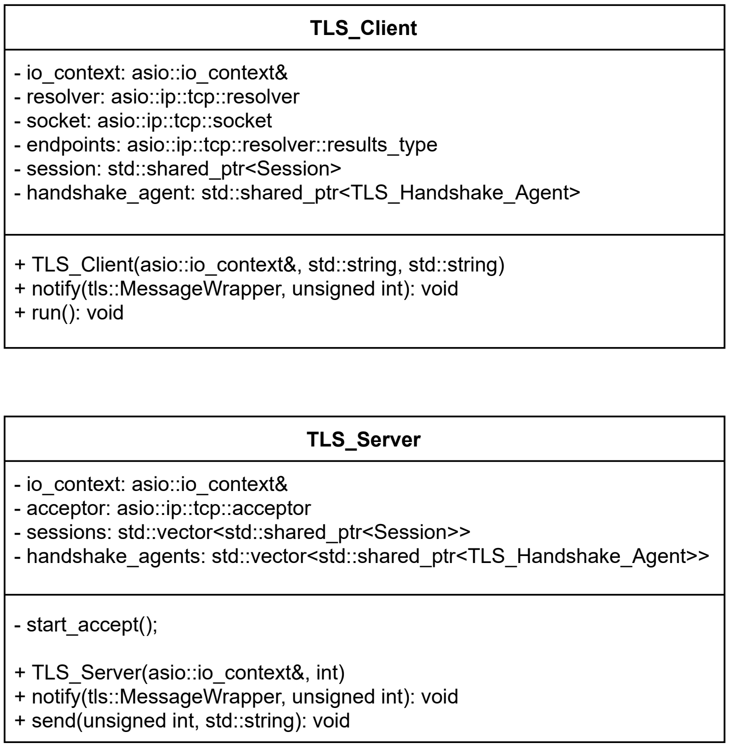

TLS_Client
- io_context: asio::io_context&
- resolver: asio::ip::tcp::resolver
- socket: asio::ip::tcp::socket
- endpoints: asio::ip::tcp::resolver::results_type
- session: std::shared_ptr<Session>
- handshake_agent: std::shared_ptr<TLS_Handshake_Agent>
+ TLS_Client(asio::io_context&, std::string, std::string)
+ notify(tls::MessageWrapper, unsigned int): void
+ run(): void
TLS_Server
- io_context: asio::io_context&
- acceptor: asio::ip::tcp::acceptor
- sessions: std::vector<std::shared_ptr<Session>>
- handshake_agents: std::vector<std::shared_ptr<TLS_Handshake_Agent>>
- start_accept();
+ TLS_Server(asio::io_context&, int)
+ notify(tls::MessageWrapper, unsigned int): void
+ send(unsigned int, std::string): void

Figure 4: The TLS_Client and TLS_Server classes containing main logic for the application.

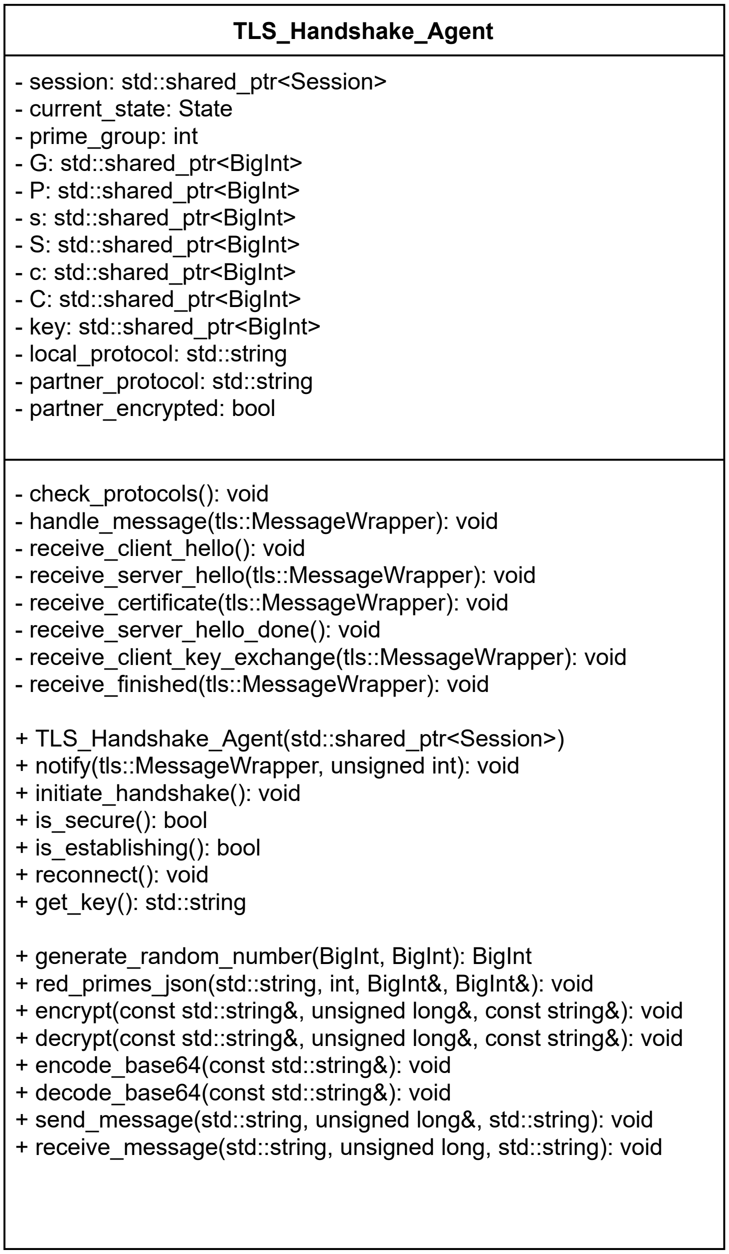

TLS_Handshake_Agent
- session: std::shared_ptr<Session>
- current_state: State
- prime_group: int
- G: std::shared_ptr<BigInt>
- P: std::shared_ptr<BigInt>
- s: std::shared_ptr<BigInt>
- S: std::shared_ptr<BigInt>
- c: std::shared_ptr<BigInt>
- C: std::shared_ptr<BigInt>
- key: std::shared_ptr<BigInt>
- local_protocol: std::string
- partner_protocol: std::string
- partner_encrypted: bool
- check_protocols(): void
- handle_message(tls::MessageWrapper): void
- receive_client_hello(): void
- receive_server_hello(tls::MessageWrapper): void
- receive_certificate(tls::MessageWrapper): void
- receive_server_hello_done(): void
- receive_client_key_exchange(tls::MessageWrapper): void
- receive_finished(tls::MessageWrapper): void
+ TLS_Handshake_Agent(std::shared_ptr<Session>)
+ notify(tls::MessageWrapper, unsigned int): void
+ initiate_handshake(): void
+ is_secure(): bool
+ is_establishing(): bool
+ reconnect(): void
+ get_key(): std::string
+ generate_random_number(BigInt, BigInt): BigInt
+ red_primes_json(std::string, int, BigInt&, BigInt&): void
+ encrypt(const std::string&, unsigned long&, const string&): void
+ decrypt(const std::string&, unsigned long&, const string&): void
+ encode_base64(const std::string&): void
+ decode_base64(const std::string&): void
+ send_message(std::string, unsigned long&, std::string): void
+ receive_message(std::string, unsigned long, std::string): void

Figure 5: The TLS_Handshake_Agent handles any key exchange messages. It also contains utility methods for encrypting/decrypting messages.

3.3 Class Association

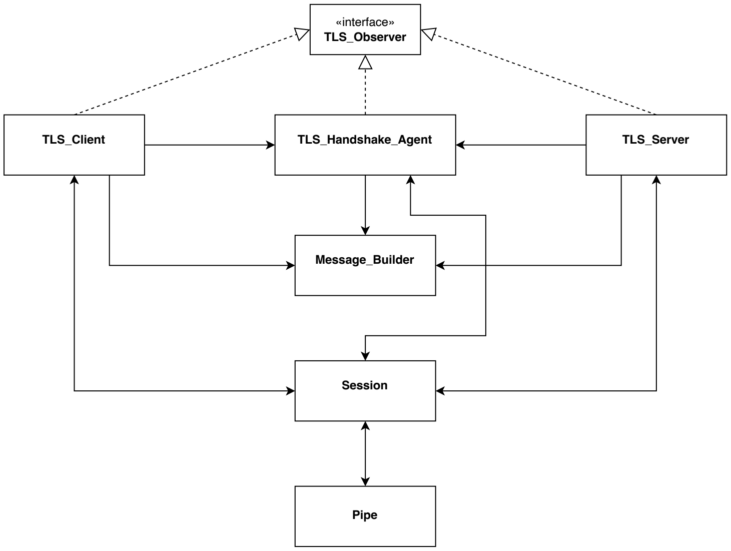


Figure 6: This UML diagram shows how the different classes of the application are associated to each other.

4 Description of code-blocks

4.1 Asio

Network communication between client and server is established by utilizing asio.

4.1.1 Client Connection

```
asio::io_context io_context;
asio::ip::tcp::resolver resolver(io_context);
asio::ip::tcp::socket socket(io_context);
asio::ip::tcp::resolver::results_type endpoints =
  resolver.resolve(host, port);

asio::connect(socket, endpoints);
```

Source Code 1: Creation of socket connection on client side.

4.1.2 Server Connection

```
asio::io_context io_context{1};
asio::ip::tcp::acceptor acceptor{
  io_context, asio::ip::tcp::endpoint(asio::ip::tcp::v4(), port)
};
acceptor.async_accept(
[this](const std::error_code& ec, asio::ip::tcp::socket socket) {
  if (!ec) {
    // handle socket
  } else {
    // throw error
  }
});
```

Source Code 2: Server asynchronously waiting for client connections.

4.2 Protobuf

Any data to be sent over TCP is serialized using Google Protobuf [3].

4.2.1 Message Serialization

```
void Pipe::send(google::protobuf::Message& message) {
  u_int64_t message_size{message.ByteSizeLong()};
  asio::write(*socket, asio::buffer(&message_size, sizeof(message_size)));

  asio::streambuf buffer;
  std::ostream os(&buffer);
  message.SerializeToOstream(&os);
  asio::write(*socket, buffer);
}
```

Source Code 3: Serialization of protobuf messages.

4.2.2 Message De-serialization

```
void Pipe::receive(google::protobuf::Message& message) {
  u_int64_t message_size;
  asio::read(*socket, asio::buffer(&message_size, sizeof(message_size)));

  asio::streambuf buffer;
  asio::streambuf::mutable_buffers_type bufs = buffer.prepare(message_size);
  buffer.commit(asio::read(*socket, bufs));

  std::istream is(&buffer);
  message.ParseFromIstream(&is);
}
```

Source Code 4: De-serialization of protobuf messages.

4.3 TLS Handshake

Whenever a new message is received the TLS_Handshake_Agent class is responsible for handling and responding to any handshake related message.

4.4 Message Handling

```
void TLS_Handshake_Agent::handle_message(tls::MessageWrapper message) {
  tls::MessageType messageType = message.type();

  if (messageType == tls::MessageType::CLIENT_HELLO) {
    receive_client_hello();

  } else if (messageType == tls::MessageType::SERVER_HELLO) {
    receive_server_hello(message);

  } else if (messageType == tls::MessageType::CERTIFICATE) {
    receive_certificate(message);

  } else if (messageType == tls::MessageType::SERVER_HELLO_DONE) {
    receive_server_hello_done();

  } else if (messageType == tls::MessageType::CLIENT_KEY_EXCHANGE) {
    receive_client_key_exchange(message);

  } else if (messageType == tls::MessageType::CHANGE_CIPHER_SPEC) {
    partnerEncrypted = true;

  } else if (messageType == tls::MessageType::FINISHED) {
    if (!partnerEncrypted) {
      session->send(Messagebuilder::build_abort_message());
      throw std::runtime_error("Partner did not send ChangeCipherSpec");
    }
    receive_finished(message);

  } else if (messageType == tls::MessageType::ABORT) {
    currentState = State::UNSECURED;
    throw new std::runtime_error("TLS connection aborted");
  } else {
    spdlog::error("Unknown message type: {}", messageType);
  }
}
```

Source Code 5: TLS Handshake Agent handling a message.

4.5 External Libraries

4.5.1 CLI11

CLI11 [1] implements a basic Command Line Interface (CLI) where users are able to specify parameters relevant for the program.

4.5.2 spdlog

To log important information the logging library spdlog [6] is employed.

```
spdlog::trace("Trace bugs during development");
spdlog::debug("Debug messages");
spdlog::info("User-facing messages");
spdlog::warn("Potential errors");
spdlog::error("Errors");
spdlog::critical("Critical errors");
```

Source Code 6: Usage of different log types.

4.5.3 JSON

Any further information, e.g. prime number for Diffie-Hellman IKE, is stored in a .json file which is read as follows. This is done using nlohman/json [5].

```
void TLS_Handshake_Agent::read_primes_json(
  std::string filename,
  int id,
  BigInt& g,
  BigInt& p
) {
  std::ifstream file(filename);
  if (!file.is_open()) {
    throw new std::runtime_error("Error opening file");
  }
  nlohmann::json primes;
  file >> primes;
  file.close();
  g = int(primes["groups"][id]["g"]);
  p = std::string(primes["groups"][id]["p_dec"]);
}
```

Source Code 7: nlohman reading the generator g and prime number p.

4.5.4 plusaes

The header-only library plusaes [4] is used to symmetrically encrypt/decrypt messages.

```
std::vector<unsigned char> key(32);

size = plusaes::get_padded_encrypted_size(message.size());
std::vector<unsigned char> encrypted(size);

plusaes::encrypt_cbc(
  (unsigned char*)message.data(), message.size(),
  &key[0], key.size(),
  &iv,
  &encrypted[0], encrypted.size(),
  true
);
```

Source Code 8: Plusaes encrypting a message

4.5.5 picoSHA2

Hashing is done using the header-only library picoSHA2 [7].

```
std::vector<unsigned char> key(32);
picosha2::hash256(key_string.begin(), key_string.end(), key);
```

Source Code 9: Plusaes encrypting a message

4.5.6 BigInt

The Diffie-Hellman IKE uses large integer values which are not supported in standard C++17. Therefore the header-only library BigInt [2] was included.

5 Usage

5.1 Command Line Arguments

-n, –hostname Hostname of the server (default: localhost)

-i, –ip IPv4 address of the server (to be preferred over hostname)

-p, –port The port of the server (default: 4433)

-l, –log-level Log-Level of the application (default: info)

6 Project Structure

```
/
├── LICENSE
├── meson_options.txt
├── meson.build
├── README.md
├── CHANGELOG.org
├── modp_primes.json
├── include
│   ├── tls_client.h
│   └── tls_server.h
├── src
│   ├── client.cpp
│   ├── server.cpp
│   ├── tls_client.cpp
│   └── tls_server.cpp
├── doc
│   ├── doc.tex
│   ├── references.bib
│   └── doc.pdf
├── tls_util
│   ├── include
│   │   ├── BigInt.hpp
│   │   ├── picosha2.h
│   │   ├── plusaes.hpp
│   │   ├── messagebuilder.h
│   │   ├── pipe.h
│   │   ├── session.h
│   │   ├── tls_handshake_agent.h
│   │   └── tls_observer.h
│   ├── src
│   │   ├── Message.proto
│   │   ├── pipe.cpp
│   │   ├── session.cpp
│   │   └── tls_handshake_agent.cpp
│   └── meson.build
└── build
```

References

[1] CLIUtils. Cli11. https://github.com/CLIUtils/CLI11, 2022.

[2] faheel. Bigint. https://github.com/faheel/BigInt, 2022.

[3] Google. Protocol buffers. https://developers.google.com/protocol-buffers, 2022.

[4] kkAyataka. plusaes. https://github.com/kkAyataka/plusaes, 2022.

[5] Niels Lohmann. Json for modern c++. https://nlohmann.github.io/json/, 2022.

[6] Gabi Melman. spdlog. https://github.com/gabime/spdlog, 2022.

[7] okdshin. Picosha2. https://github.com/okdshin/PicoSHA2, 2022.