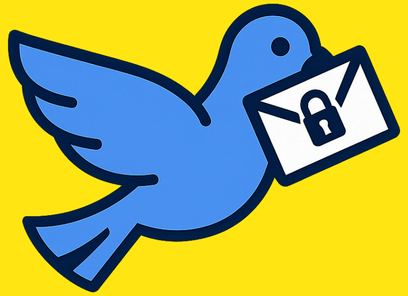
0 DEPENDENCY



LIGHTWEIGHT SECURE TCP

"JUST LIKE ITS NAME!"

C++ LIBRARY

A lightweight C++ library that wraps plain TCP with encrypted handshake, message integrity, and zero dependencies.

- XTEA256-based encryption
- Nonce + math challenge handshake
- Packets with strict layout and serialization
- Works on ESP32, RPi, Qt, or desktop
- No boost, no OpenSSL, only STD
- Drop-in integration via CMake

Why Lightweight-Secure-TCP?

Plain TCP lacks encryption and authentication, making it unsuitable for secure communication.

We needed a secure and minimal solution that doesn't require TLS or certificates.

Designed to run on a 240 MHz ESP32 with limited memory and processing power.

TLS provides security, but is too heavy and complex for embedded systems like ESP32.

The library works without OpenSSL or any third-party cryptographic dependencies.

Implements handshake and encryption in a single, portable C++ header file.

How Lightweight-Secure-TCP works?

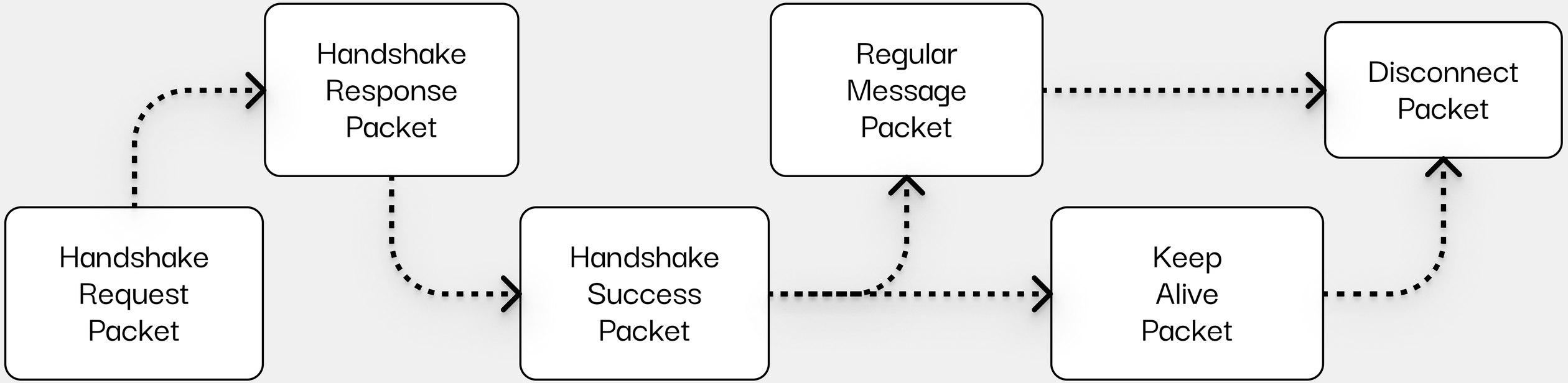
Before any message is exchanged, the client initiates a handshake containing a random nonce and a math challenge.

If the handshake fails, the connection is dropped. If it succeeds, a secure session is created.

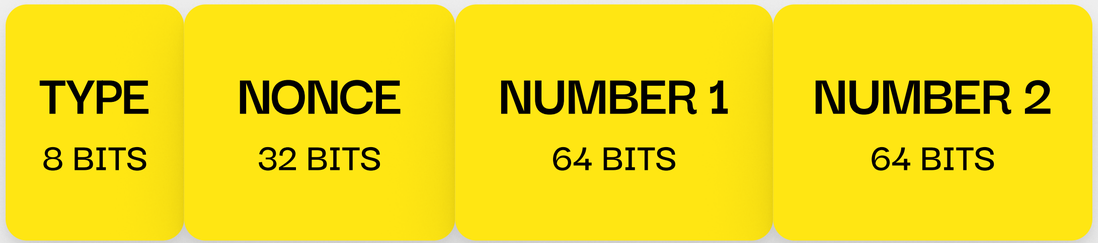
The handshake is encrypted with a static private key.

Every session afterward uses a unique session key.

6 different packets used to encapsulate data for the protocol

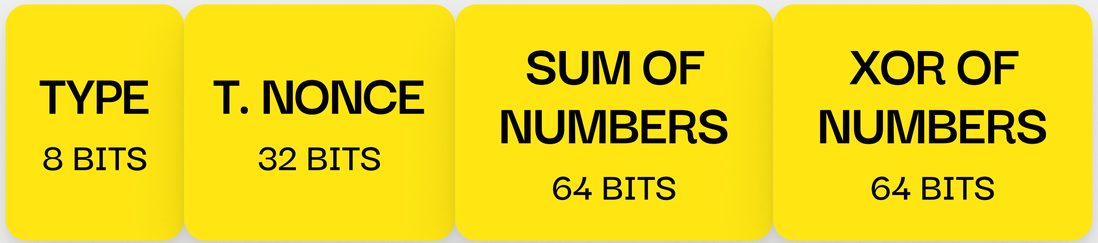


Handshake Request Packet (21 bytes)



Sent by the server upon connection.
Includes a 32-bit random nonce and two 64-bit random numbers.
Type (0x02) is unencrypted.
The remaining payload is encrypted using the static shared key.

Handshake Response Packet (21 bytes)

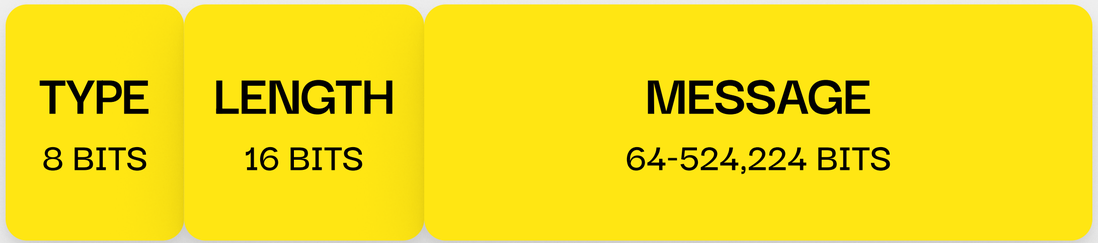


Sent by the client in response to the server's challenge.
Includes a transformed nonce, calculated by XOR'ing the original nonce with parts of the sum and XOR of the challenge numbers.
Also includes the sum and XOR of the two challenge numbers.
Type (0x03) is unencrypted.
The remaining payload is encrypted using the static shared key.

Handshake Success Packet (33 bytes)

Sent by the server after validating the client's response.
Contains a newly generated 256-bit (32-byte) XTEA256 session key used for encrypting all further communication.
Type (0x04) is unencrypted.
The session key is encrypted using the static shared key.

Regular Message Packet (11-65,531 bytes)



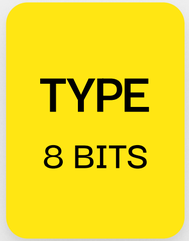
Sent bidirectionally during an active session.
Type (0x01) and message length (2 bytes) are unencrypted.
The message is encrypted using the session key and padded to reach a multiple of 8 bytes (64 bits).
Total packet size ranges from 11 to 65,531 bytes, depending on message length and padding.

Keep Alive Packet
(1 byte)

TYPE
8 BITS

Sent periodically to prevent connection timeouts.
Contains only the packet type (0x05), which is unencrypted.
Used in both directions to confirm the connection is still active.

Disconnect Packet (1 byte)



Sent to gracefully terminate a connection.
Contains only the packet type (0x06), which is unencrypted.
Used in both directions to signal a clean shutdown.

How XTEA256 encryption works

Encrypts 64-bit blocks (using 2 × 32-bit unsigned integers)

Uses a 256-bit symmetric key

Performs 32 rounds of encryption by default

All 8 key parts are used in each round

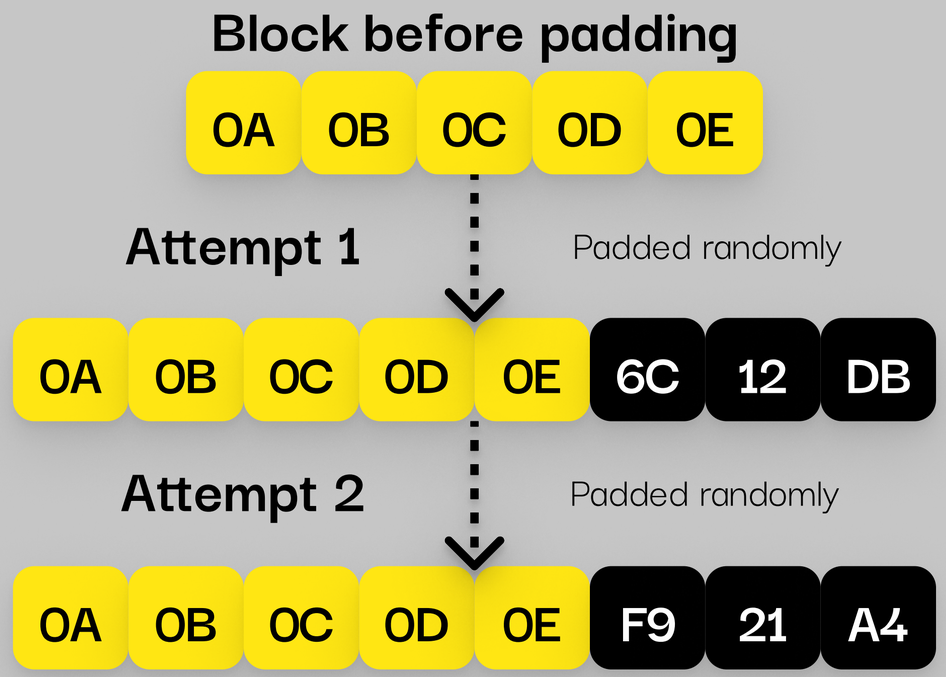
All 8 key parts are used in each round

Handshake packets use a static shared key

Session messages use a dynamic session key

No external libraries or dependencies required

How randomized padding prevents repetition



- XTEA256 encrypts data in 8-byte blocks. To fill incomplete blocks, random padding bytes are added.
- This ensures that identical messages result in different encrypted outputs.
- It makes it harder to compare packets or infer any session key.
- Each encryption attempt produces a unique result.
- This enhances privacy by hiding message patterns.

Quick Setup Guide
Installation is very simple

Download the Lightweight-Secure-TCP repository.

Copy the ‘lib’ folder into your project.

If you’re using PlatformIO, this is enough. For CMake-based desktop projects, add the following lines:

```
add_subdirectory(lib/LightweightSecureTCP)
target_link_libraries(your_target PRIVATE
LightweightSecureTCP)
```

3 example projects included in the repository

ESP32 Localhost Example – FizzBuzz

Runs client and server on the same ESP32. Tests full handshake, messaging, and disconnect locally.

ESP32 ↔ Desktop Example – FizzBuzz

ESP32 acts as client, desktop as server. Used to test real communication across devices.

Qt Desktop Example – Localhost Chat App

A basic GUI chat app built with Qt. Runs client and server on localhost to demonstrate packet exchange.

Setting the handshake key

```
// Set handshake key
Key handshakeKey({0x12345678, 0x9ABCDEF0, 0x13572468, 0xACEBDF01,
                  0x11223344, 0x55667788, 0x99AABBCC, 0xDDEEFF00});

LightweightSecureTCP::setHandshakeKey(handshakeKey);
```

The static key must be defined before constructing any client or server instance.

Setting the server

```
LightweightSecureTCP::setHandshakeKey(handshakeKey);

LightweightSecureServer server(PORT);
server.start();
server.setKeepAliveConfig(1000, 5000);
server.setOnHandshakeSuccess([&](HandshakeResult, Session &session)
                             { session.setOnMessageReceived([&](const std::string &message)
                                                            {
                                                                lwsdebug("[SERVER]") << "Received: " << message;
                                                                int received = std::stoi(message);
                                                                int next = received + 1;
                                                                std::string text = fizzBuzz(next);

                                                                std::string reply = std::to_string(next);
                                                                if (!text.empty()) reply += " " + text;

                                                                lwsdebug("[SERVER]") << "Sending: " << reply;

                                                                std::this_thread::sleep_for(std::chrono::milliseconds(500));
                                                                session.sendMessage(reply);
                                                            });
                             });
lwsdebug("[Main]") << "Server is running on port " << PORT << std::endl;
while(true){
    std::this_thread::sleep_for(std::chrono::seconds(1));
}
server.stop();
```

Initializes a LightweightSecureServer on Desktop
and handles messages using FizzBuzz.

Setting the client

```
LightweightSecureClient *client = new LightweightSecureClient(SERVER_IP, SERVER_PORT);
std::atomic<bool> reconnectNeeded = true;
client->setOnDisconnected([&]()
                          {
                              ESP_LOGW("[CLIENT]", "Disconnected from server. Reconnecting...");
                              vTaskDelay(pdMS_TO_TICKS(2000));
                                reconnectNeeded = true; });

client->setOnHandshakeSuccess([&](HandshakeResult)
                              {
                                  ESP_LOGI("[CLIENT]", "Handshake completed.");

                                  client->setOnMessageReceived([&](const std::string &message)
                                                               {
                                                                   ESP_LOGI("[CLIENT]", "Received: %s", message.c_str());
                                                                   int received = std::stoi(message);
                                                                   int next = received + 1;
                                                                   std::string text = fizzBuzz(next);

                                                                   std::string reply = std::to_string(next);
                                                                   if (!text.empty())
                                                                       reply += " " + text;

                                                                   ESP_LOGI("[CLIENT]", "Sending: %s", reply.c_str());
                                                                   vTaskDelay(pdMS_TO_TICKS(500));
                                                                   client->sendMessage(reply);
                                                               });
                                  client->sendMessage("1"); });
```

Initializes a LightweightSecureClient on ESP32
and handles messages using FizzBuzz.

Desktop server output

```
[LightweightSecureServer] Server started on port 12345
[Main] Server is running on port 12345
[SecurityHandshake][Server] Starting server handshake...
[Transporter] sendHandshakeRequest to Socket[fd=7]
[Transporter] sendHandshakeSuccess to Socket[fd=7]
[SecurityHandshake][Server] Server handshake successful!
[Session] Started.
[SERVER] Received: 1
[SERVER] Sending: 2     (:
[Transporter] sendRegularMessage to Socket[fd=7]
[SERVER] Received: 3 Fizz
[SERVER] Sending: 4   (:
[Transporter] sendRegularMessage to Socket[fd=7]
[SERVER] Received: 5 Buzz
[SERVER] Sending: 6 Fizz
[Transporter] sendRegularMessage to Socket[fd=7]
[SERVER] Received: 7      (:
[SERVER] Sending: 8   (:
[Transporter] sendRegularMessage to Socket[fd=7]
```

Shows the FizzBuzz interaction from the desktop server after handshake with the ESP32 client.

Setting the server

```
void MainWindow::setupServer()
{
    m_server = new LightweightSecureServer(PORT);
    m_server->setOnHandshakeSuccess([this](const HandshakeResult& result, Session& session) {
        session.setOnMessageReceived([this](const std::string& message) {
            QMetaObject::invokeMethod(this, [this, message]() {
                appendMessage(ui->listWidget_server, QString::fromStdString(message), false);
            }, Qt::QueuedConnection);
        });

        connect(ui->pushButton_server, &QPushButton::clicked, this, [this, &session]() {
            const QString messageText = ui->lineEdit_server->text();
            if (!messageText.isEmpty()) {
                session.sendMessage(messageText.toStdString());

                QMetaObject::invokeMethod(this, [this, messageText]() {
                    appendMessage(ui->listWidget_server, messageText, true);
                }, Qt::QueuedConnection);
            }
            ui->lineEdit_server->clear();
        });
    });
    m_server->start();
}
```

Initializes a LightweightSecureServer on
localhost and handles messages via GUI controls.

Setting the client

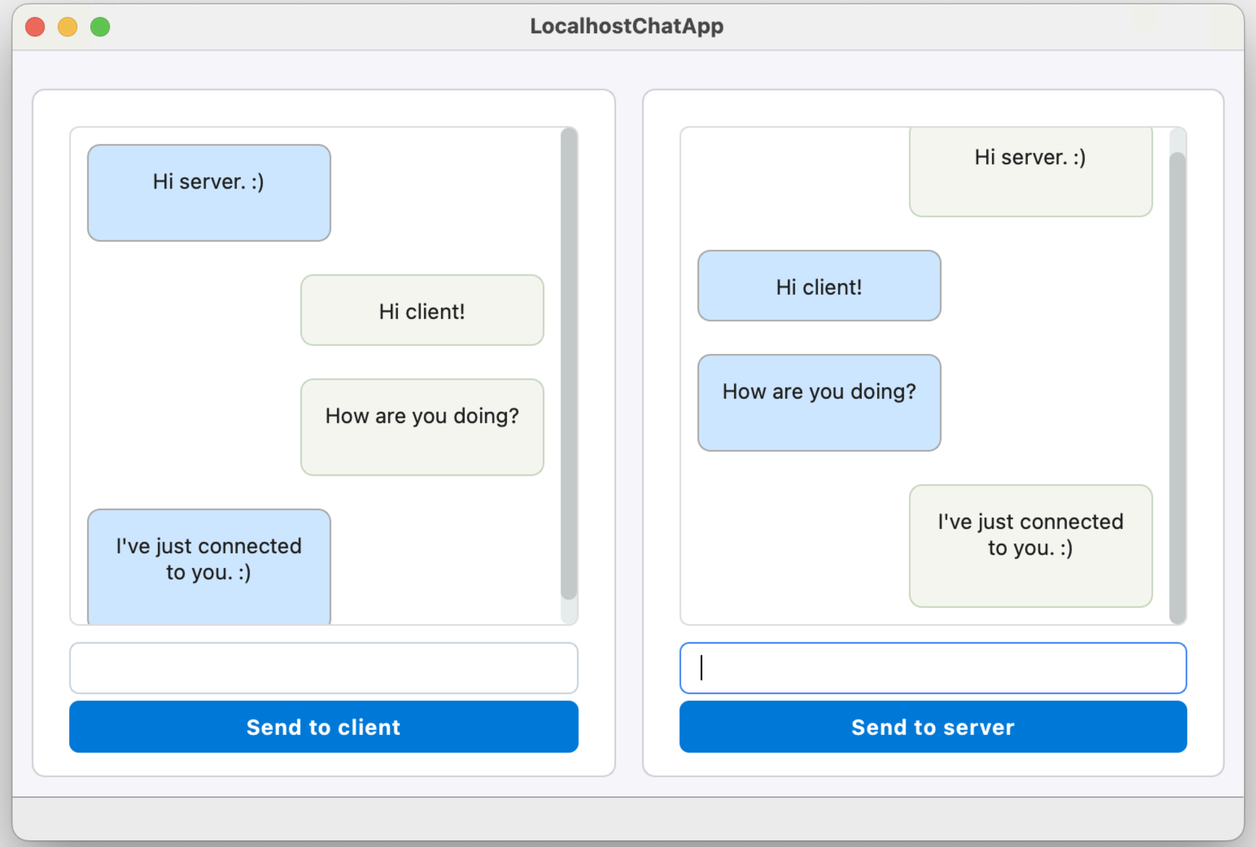
```
void MainWindow::setupClient()
{
    m_client = new LightweightSecureClient("127.0.0.1", PORT);
    m_client->setOnHandshakeSuccess([this](const HandshakeResult& result) {
        m_client->setOnMessageReceived([this](const std::string& message) {
            QMetaObject::invokeMethod(this, [this, message]() {
                appendMessage(ui->listWidget_client, QString::fromStdString(message), false);
            }, Qt::QueuedConnection);
        });

        connect(ui->pushButton_client, &QPushButton::clicked, this, [this]() {
            const QString messageText = ui->lineEdit_client->text();
            if (!messageText.isEmpty()) {
                m_client->sendMessage(messageText.toStdString());

                QMetaObject::invokeMethod(this, [this, messageText]() {
                    appendMessage(ui->listWidget_client, messageText, true);
                }, Qt::QueuedConnection);
            }
            ui->lineEdit_client->clear();
        });
    });
    m_client->connectToServer();
}
```

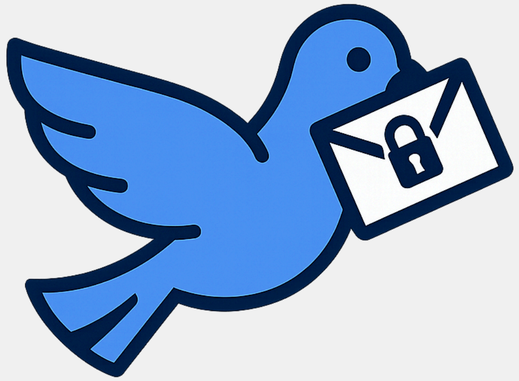
Initializes a LightweightSecureClient on localhost and handles messages via GUI controls.

Screenshot



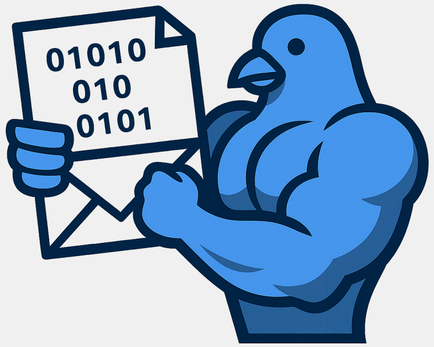
Displays a real conversation between the Qt-based client and server over localhost.

Thanks for exploring Lightweight Secure TCP. Feel free to use, contribute, or just say hi!



Full Documentation
https://sdenizozturk.github.io/lightweight-secure-tcp

GitHub Repository
https://github.com/sDenizOzturk/lightweight-secure-tcp



Contributing Guide
Documentation → Contributing