Coap

Isaac Ntambu Kanda

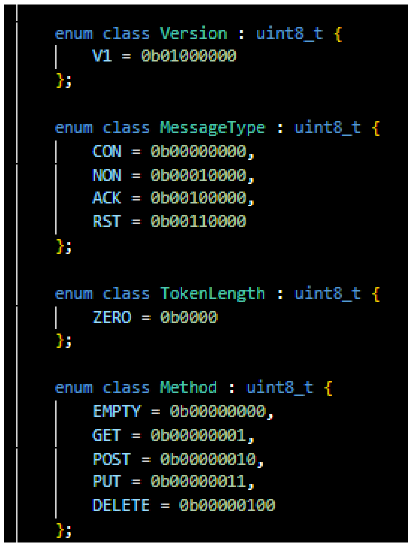
Background

The main concept of this lab is for us to build a Coap message that will later be used to communicate to the coap.me broker via a UDP connection on port 5683.

Implementation

Message Header

The implementation started by first designing the structure of a coap message, having the version, message type, TKL(token length), method, message ID, token, options and payload. This was done using enums as shown in figure 1.



```
enum class Version : uint8_t {
    V1 = 0b01000000
};

enum class MessageType : uint8_t {
    CON = 0b00000000,
    NON = 0b00010000,
    ACK = 0b00100000,
    RST = 0b00110000
};

enum class TokenLength : uint8_t {
    ZERO = 0b0000
};

enum class Method : uint8_t {
    EMPTY = 0b00000000,
    GET = 0b00000001,
    POST = 0b00000010,
    PUT = 0b00000011,
    DELETE = 0b00000100
};
```

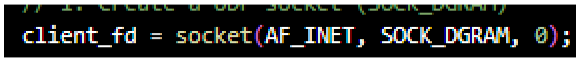
Figure 1: Implementation of the message parts using enums.

Coap client

The Coap client was responsible for making the connection to the coap.me broker this also included sending and receiving the data the coap.me broker sent back. The CoapClient class contained the DNS to the broker and port number which play a very vital roll in getting everything to work.

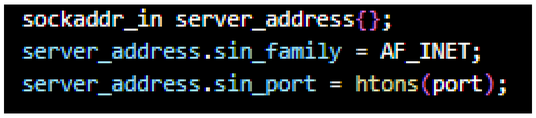
Connecting to broker

The client was able to create a UDP socket by using the “SOCKET_DGRAM” on port 5683 as shown in the figure below.



```
client_fd = socket(AF_INET, SOCK_DGRAM, 0);
```

Figure 2: creating a socket.

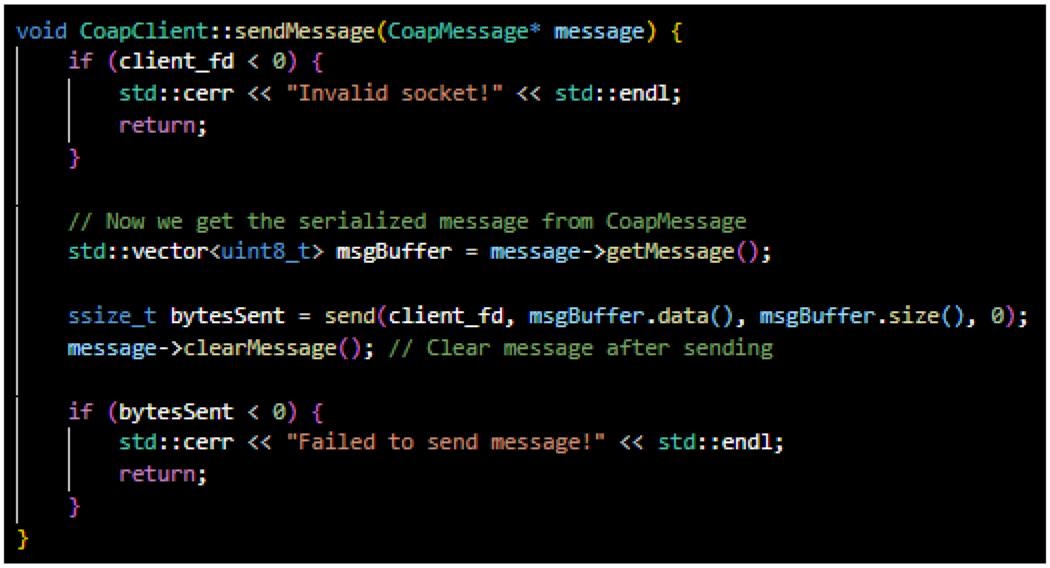


```
sockaddr_in server_address{};
server_address.sin_family = AF_INET;
server_address.sin_port = htons(port);
```

Figure 3: connecting the port.

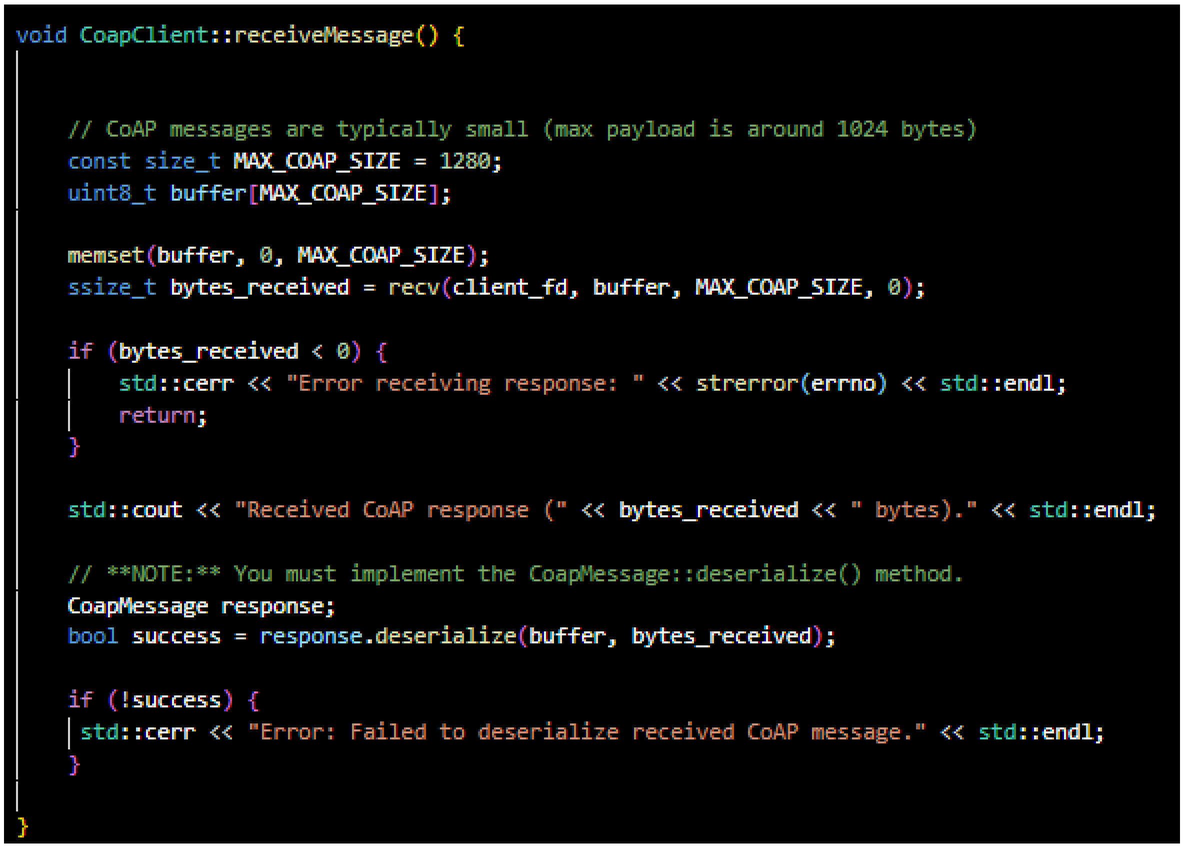
Send Message

As stated before, the client class also had to be able to send a coap message to the broker and this how it was done as shown below. Our SendMessage function takes in a message instance, and from that instance we collet the message vector by using the get function and store it in a uint8_t vector as shown in the image, and with the help of the socket function send, the message was sent to coap.me broker.



```
void CoapClient::sendMessage(CoapMessage* message) {
    if (client_fd < 0) {
        std::cerr << "Invalid socket!" << std::endl;
        return;
    }

    // Now we get the serialized message from CoapMessage
    std::vector<uint8_t> msgBuffer = message->getMessage();

    ssize_t bytesSent = send(client_fd, msgBuffer.data(), msgBuffer.size(), 0);
    message->clearMessage(); // Clear message after sending

    if (bytesSent < 0) {
        std::cerr << "Failed to send message!" << std::endl;
        return;
    }
}
```

Receive Message

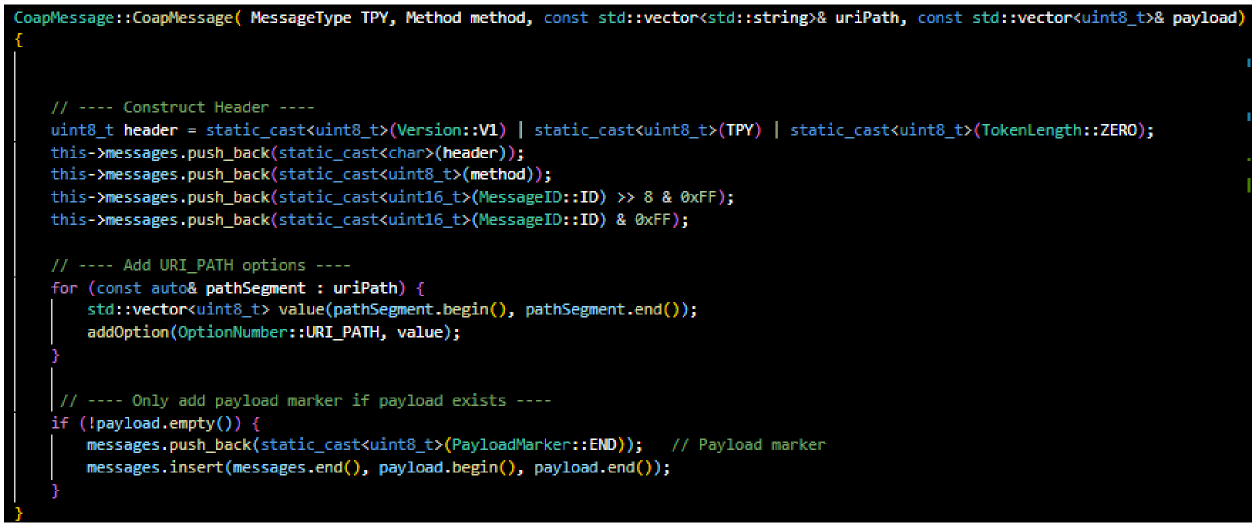


```
void CoapClient::receiveMessage() {


    // CoAP messages are typically small (max payload is around 1024 bytes)
    const size_t MAX_COAP_SIZE = 1280;
    uint8_t buffer[MAX_COAP_SIZE];

    memset(buffer, 0, MAX_COAP_SIZE);
    ssize_t bytes_received = recv(client_fd, buffer, MAX_COAP_SIZE, 0);

    if (bytes_received < 0) {
        std::cerr << "Error receiving response: " << strerror(errno) << std::endl;
        return;
    }

    std::cout << "Received CoAP response (" << bytes_received << " bytes)." << std::endl;

    // **NOTE:** You must implement the CoapMessage::deserialize() method.
    CoapMessage response;
    bool success = response.deserialize(buffer, bytes_received);

    if (!success) {
     std::cerr << "Error: Failed to deserialize received CoAP message." << std::endl;
    }

}
```

When coap.me responds, the client class receives the message and saves it inside an array and the bytes inside a ssize_t instance, which are later sent into the CoapMessage instance to be deserialized and extract the data.

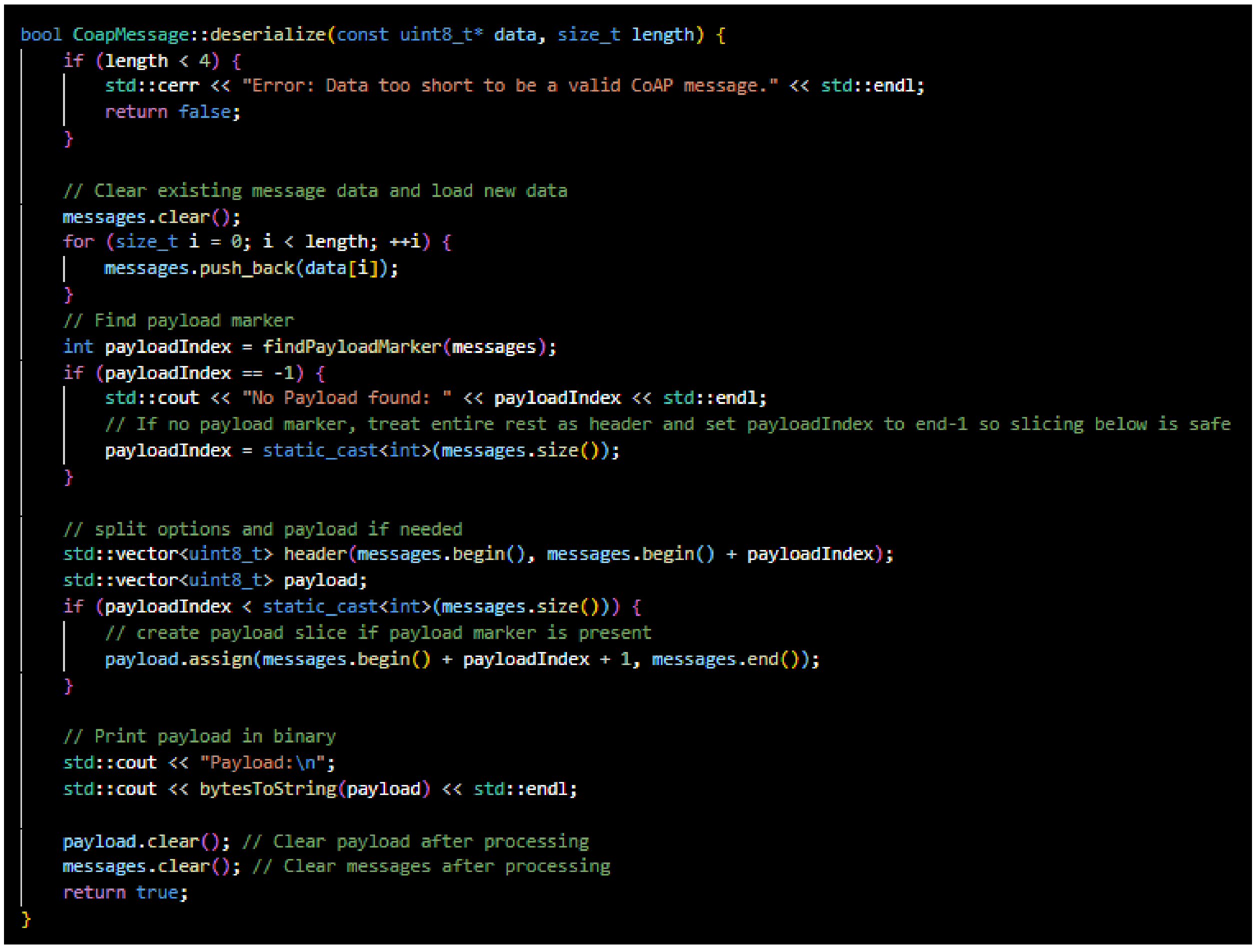
Coap Message

One of default constructors CoapMessage takes in the **method**, **path**, and **payload** as inputs. This constructor helps us in building the building the actual message as shown in the figure below.



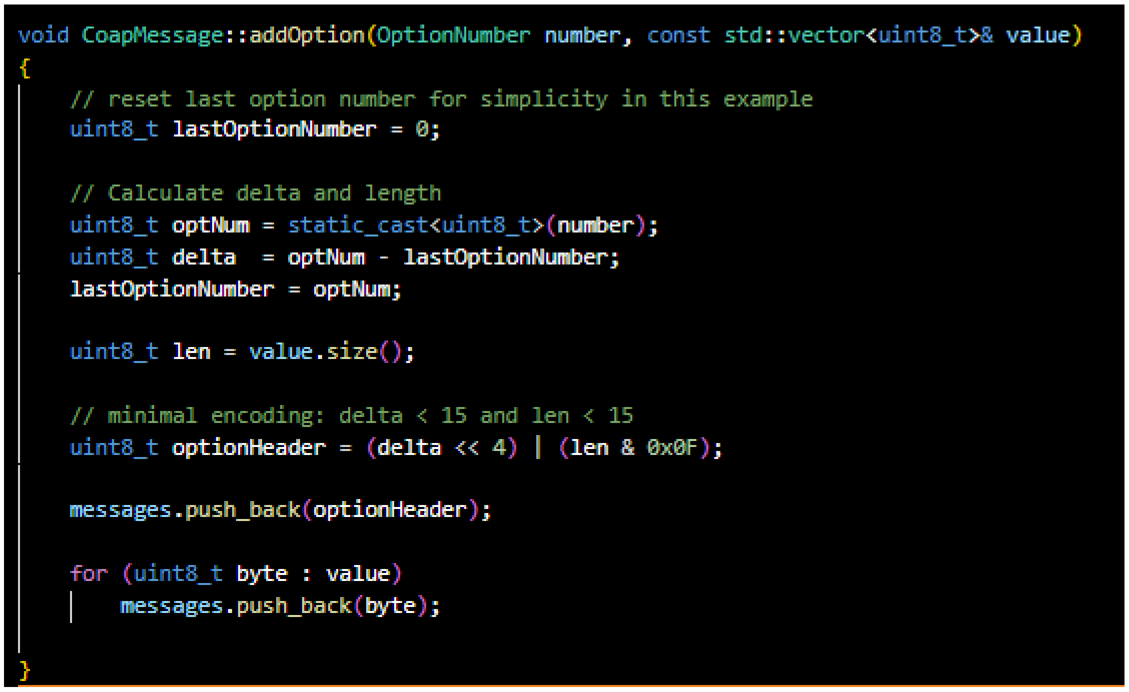
```cpp
CoapMessage::CoapMessage( MessageType TPY, Method method, const std::vector<std::string>& uriPath, const std::vector<uint8_t>& payload)
{


    // ---- Construct Header ----
    uint8_t header = static_cast<uint8_t>(Version::V1) | static_cast<uint8_t>(TPY) | static_cast<uint8_t>(TokenLength::ZERO);
    this->messages.push_back(static_cast<char>(header));
    this->messages.push_back(static_cast<uint8_t>(method));
    this->messages.push_back(static_cast<uint16_t>(MessageID::ID) >> 8 & 0xFF);
    this->messages.push_back(static_cast<uint16_t>(MessageID::ID) & 0xFF);

    // ---- Add URI_PATH options ----
    for (const auto& pathSegment : uriPath) {
        std::vector<uint8_t> value(pathSegment.begin(), pathSegment.end());
        addOption(OptionNumber::URI_PATH, value);
    }

     // ---- Only add payload marker if payload exists ----
    if (!payload.empty()) {
        messages.push_back(static_cast<uint8_t>(PayloadMarker::END));   // Payload marker
        messages.insert(messages.end(), payload.begin(), payload.end());
    }
}
```

The CoapMessage class also contains a function to deserialize the response received from the coap.me server and finds the payload with the help of the payload marker and later prints it out.



```cpp
bool CoapMessage::deserialize(const uint8_t* data, size_t length) {
    if (length < 4) {
        std::cerr << "Error: Data too short to be a valid CoAP message." << std::endl;
        return false;
    }

    // Clear existing message data and load new data
    messages.clear();
    for (size_t i = 0; i < length; ++i) {
        messages.push_back(data[i]);
    }
    // Find payload marker
    int payloadIndex = findPayloadMarker(messages);
    if (payloadIndex == -1) {
        std::cout << "No Payload found: " << payloadIndex << std::endl;
        // If no payload marker, treat entire rest as header and set payloadIndex to end-1 so slicing below is safe
        payloadIndex = static_cast<int>(messages.size());
    }

    // split options and payload if needed
    std::vector<uint8_t> header(messages.begin(), messages.begin() + payloadIndex);
    std::vector<uint8_t> payload;
    if (payloadIndex < static_cast<int>(messages.size())) {
        // create payload slice if payload marker is present
        payload.assign(messages.begin() + payloadIndex + 1, messages.end());
    }

    // Print payload in binary
    std::cout << "Payload:\n";
    std::cout << bytesToString(payload) << std::endl;

    payload.clear(); // Clear payload after processing
    messages.clear(); // Clear messages after processing
    return true;
}
```

Lastly a function that helps us build the option structure and later sends it also inside the message vector.



```cpp
void CoapMessage::addOption(OptionNumber number, const std::vector<uint8_t>& value)
{
    // reset last option number for simplicity in this example
    uint8_t lastOptionNumber = 0;

    // Calculate delta and length
    uint8_t optNum = static_cast<uint8_t>(number);
    uint8_t delta  = optNum - lastOptionNumber;
    lastOptionNumber = optNum;

    uint8_t len = value.size();

    // minimal encoding: delta < 15 and len < 15
    uint8_t optionHeader = (delta << 4) | (len & 0x0F);

    messages.push_back(optionHeader);

    for (uint8_t byte : value)
        messages.push_back(byte);

}
```