// main.cpp - WebSocket Client Application

#include <iostream>

#include <string>

#include <thread>

#include <atomic>

#include <mutex>

#include <condition\_variable>

#include <boost/beast/core.hpp>

#include <boost/beast/websocket.hpp>

#include <boost/asio/connect.hpp>

#include <boost/asio/ip/tcp.hpp>

#include <boost/asio/ssl/stream.hpp>

#include <boost/asio/ssl/context.hpp>

namespace beast = boost::beast;

namespace websocket = beast::websocket;

namespace net = boost::asio;

namespace ssl = net::ssl;

using tcp = net::ip::tcp;

// Class to handle WebSocket

communication

class WebSocketClient {

private:

net::io\_context ioc\_;

ssl::context ctx\_{ssl::context::tlsv12\_client};

std::unique\_ptr<websocket::stream<beast::ssl\_stream<tcp::socket>>> ws\_;

std::atomic<bool> is\_connected\_{false};

std::thread read\_thread\_;

std::mutex mutex\_;

std::condition\_variable cv\_;

bool shutdown\_ = false;

public:

WebSocketClient() {

// Set up SSL context

ctx\_.set\_default\_verify\_paths();

ctx\_.set\_verify\_mode(ssl::verify\_peer);

}

~WebSocketClient() {

disconnect();

if (read\_thread\_.joinable()) {

read\_thread\_.join();

}

}

bool connect(const std::string& host, const std::string& port, const std::string& target) {

try {

// Look up the domain name

tcp::resolver resolver{ioc\_};

auto const results = resolver.resolve(host, port);

// Create the WebSocket with SSL support

ws\_ = std::make\_unique<websocket::stream<bea

st::ssl\_stream<tcp::socket>>>(ioc\_, ctx\_);

// Set SNI hostname (required for SSL)

if (!SSL\_set\_tlsext\_host\_name(ws\_->next\_layer().native\_handle(), host.c\_str())) {

throw beast::system\_error(

beast::error\_code(static\_cast<int>(::ERR\_get\_error()),

net::error::get\_ssl\_category()),

"Failed to set SNI hostname");

}

// Connect to the TCP endpoint

net::connect(ws\_->next\_layer().next\_layer(), results.begin(), results.end());

// Perform SSL handshake

ws\_->next\_layer().handshake(ssl::stream\_base::client);

// Perform WebSocket handshake

ws\_->handshake(host, target);

is\_connected\_ = true;

// Start a thread to read incoming messages

read\_thread\_ = std::thread([this]() {

read\_messages();

});

return true;

} catch (const std::exception& e) {

std::cerr << "Error connecting to

WebSocket server: " << e.what() << std::endl;

return false;

}

}

void disconnect() {

if (is\_connected\_) {

try {

// Signal the read thread to stop

{

std::lock\_guard<std::mutex> lock(mutex\_);

shutdown\_ = true;

}

cv\_.notify\_all();

// Close the WebSocket connection

ws\_->close(websocket::close\_code::normal);

is\_connected\_ = false;

std::cout << "Disconnected from WebSocket server" << std::endl;

} catch (const std::exception& e) {

}

}

}

bool is\_connected() const {

return is\_connected\_;

}

bool send\_message(const std::string& message) {

if (!is\_connected\_) {

std::cerr << "Not connected to WebSocket server" << std::endl;

return false;

}

try {

ws\_->write(net::buffer(message));

return true;

} catch (const std::exception& e) {

is\_connected\_ = false;

return false;

}

}

private:

void read\_messages() {

try {

beast::flat\_buffer buffer;

while (is\_connected\_) {

// Check if we should shutdown

{

std::unique\_lock<std::mutex>

lock(mutex\_);

if (cv\_.wait\_for(lock, std::chrono::milliseconds(100),

[this]{ return shutdown\_; })) {

break;

}

}

// Check if there's a message to read

if (ws\_->is\_open() && ws\_->got\_binary() || ws\_->got\_text()) {

ws\_->read(buffer);

std::cout << "> ";

std::cout.flush();

buffer.consume(buffer.size());

}

// Sleep a bit to avoid busy waiting

std::this\_thread::sleep\_for(std::chrono::milliseconds(50));

}

} catch (const beast::system\_error& se) {

if (se.code() != websocket::error::closed) {

}

is\_connected\_ = false;

} catch (const std::exception& e) {

is\_connected\_ = false;

}

}

};

// Parse URL into components

bool parse\_url(const std::string& url, std::string& protocol, std::string& host,

std::string& port, std::string& target) {

// Find protocol delimiter

auto proto\_end = url.find("://");

if (proto\_end == std::string::npos) {

std::cerr << "Invalid URL format (no protocol)" << std::endl;

return false;

}

protocol = url.substr(0, proto\_end);

// Find host and target

auto host\_start = proto\_end + 3;

auto path\_start = url.find("/", host\_start);

if (path\_start == std::string::npos) {

host = url.substr(host\_start);

target = "/";

} else {

host = url.substr(host\_start, path\_start - host\_start);

target = url.substr(path\_start);

}

// Check for port specification

auto port\_start = host.find(":");

if (port\_start != std::string::npos) {

port = host.substr(port\_start + 1);

host = host.substr(0, port\_start);

} else {

// Default ports

if (protocol == "ws" || protocol == "http") {

port = "80";

} else if (protocol == "wss" || protocol == "https") {

port = "443";

} else {

return false;

}

}

return true;

}

int main() {

WebSocketClient client;

std::string server\_url = "wss://echo.websocket.events/.ws";

std::string input;

std::string protocol, host, port, target;

std::cout << "WebSocket Client" << std::endl;

std::cout << "----------------" << std::endl;

// Parse the default URL

if (!parse\_url(server\_url, protocol, host, port, target)) {

std::cerr << "Failed to parse default URL" << std::endl;

return 1;

}

bool running = true;

while (running) {

if (!client.is\_connected()) {

std::getline(std::cin, input);

if (input.empty()) {

input = server\_url;

}

if (!parse\_url(input, protocol, host,

port, target)) {

std::cerr << "Invalid URL format" << std::endl;

continue;

}

if (!client.connect(host, port, target)) {

std::cout << "Connection failed. Try again? (y/n): ";

std::getline(std::cin, input);

if (input != "y" && input != "Y") {

running = false;

}

continue;

}

}

std::cout << "\nCommands:" <<

std::endl;

std::cout << " /quit - Exit the application" << std::endl;

std::cout << " /disconnect - Disconnect from the server" << std::endl;

std::cout << " Any other input will be sent as a message" << std::endl;

while (client.is\_connected()) {

std::cout << "> ";

std::getline(std::cin, input);

if (input == "/quit") {

running = false;

break;

} else if (input == "/disconnect") {

client.disconnect();

break;

} else if (!input.empty()) {

if (!client.send\_message(input)) {

std::cout << "Failed to send

message. Connection lost." << std::endl;

break;

}

}

}

}

std::cout << "Exiting application" << std::endl;

return 0;

}