UNIVERSIDADE
DE TRÁS-OS-MONTES
E ALTO DOURO

Trabalho Prático 1

Servidor de reserva de bilhetes de teatro

Licenciatura em Engenharia Informática

Sistemas Distribuídos

Hugo Paredes

Ivan Miguel Serrano Pires

Autores

Diogo Medeiros n.º 70633

Pedro Silva n.º 70649

Rui Pinto n.º 70648

Vila Real, maio 2022

ÍNDICE

1. INTRODUÇÃO

No âmbito da Unidade Curricular de Sistemas Distribuídos, foi solicitado um trabalho prático que consiste na implementação de um sistema cliente/servidor de reserva de bilhetes de espetáculos, usando a API Winsock 2.

2. PROTOCOLO DE COMUNICAÇÃO

O protocolo de comunicação entre o cliente e o servidor assenta em mensagens, serializadas em JSON, compostas por 2 atributos:

- Código
- Conteúdo da mensagem

O código da mensagem, à semelhança do protocolo de comunicação HTTP, corresponde a um valor numérico, associado a uma etiqueta representativa do tipo de mensagem:

- HELLO – 0
- GET_LOCATIONS – 1
- GET_GENRES – 2
- GET_SHOWS – 3
- BUY_TICKETS – 4
- QUIT – 5

O conteúdo em si, podendo tratar-se desde um valor inteiro a uma lista de objetos, é ele também serializado em JSON, para mais fácil transporte da mensagem.

Quando o cliente se conecta pela primeira vez, o servidor envia uma mensagem HELLO para confirmar a conexão.

Caso o cliente escolha comprar bilhetes, é enviada uma mensagem a solicitar as localizações disponíveis, à qual o servidor responde com uma lista. Após escolher a localização pretendida, o cliente solicita os géneros de espetáculos na mesma, recebendo do servidor, novamente uma lista dos géneros disponíveis.

Após escolher o género, o cliente solicita os espetáculos de acordo com as suas preferências. Já o servidor, com base nas preferências e histórico de compras do cliente, recomenda uma lista de espetáculos ao cliente.

Caso o cliente escolha comprar bilhetes, após indicar o espetáculo desejado e o n.º de bilhetes, envia essa informação ao servidor, que processa a compra.

Por fim, caso o utilizador opte por terminar a comunicação, é enviada uma mensagem ao servidor a informá-lo desse desejo, o qual responde com uma mensagem de confirmação.

Na figura seguinte, encontra-se ilustrado o fluxo de mensagens entre cliente e servidor.

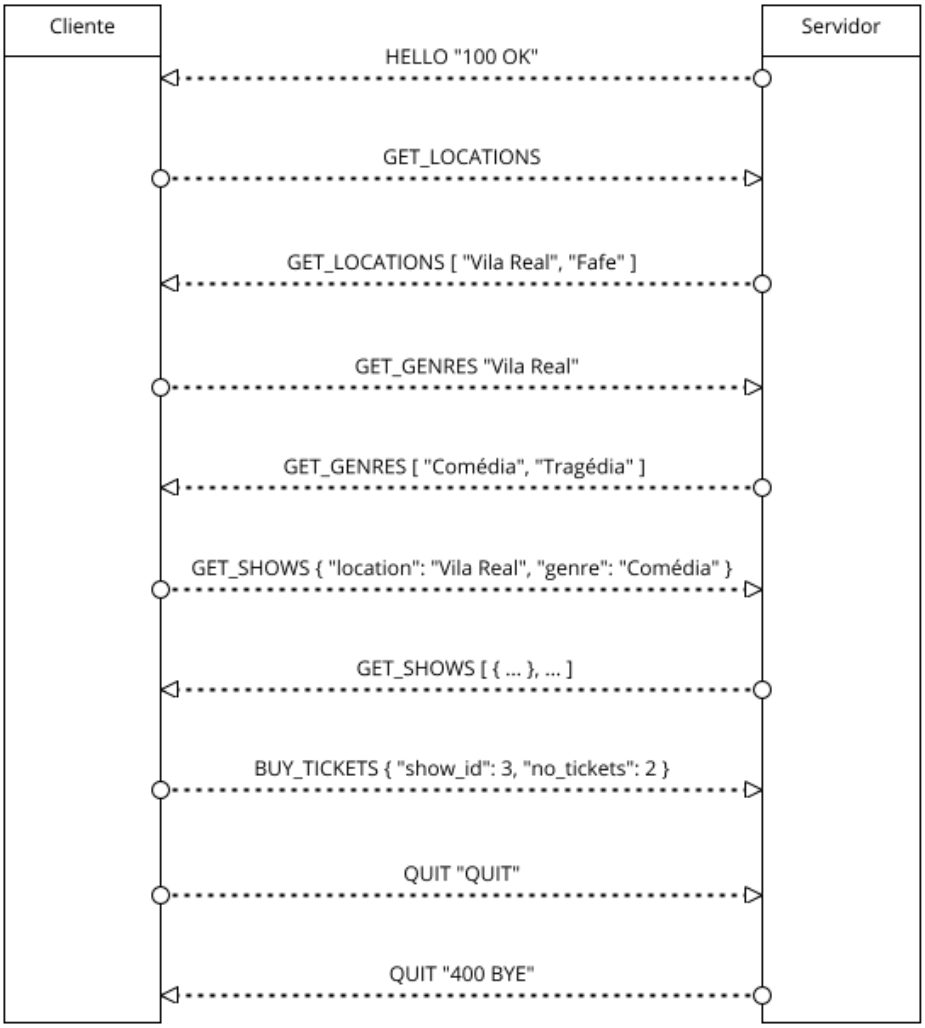

Fig. 1 – Troca de mensagens entre cliente e servidor

3. IMPLEMENTAÇÃO

O sistema de reserva de bilhetes de teatro foi implementado numa única solução, composta por três projetos distintos: duas aplicações de consola para cada agente de comunicação, cliente e servidor; uma biblioteca estática para partilha de classes.

A solução foi escrita, na sua totalidade, em C++20, podendo assim, usufruir das vantagens inerentes à utilização de classes, bem como das funcionalidades mais recentes da linguagem. Esta escolha prende-se por duas razões: segurança, dado que a maioria das funções em C encontram-se obsoletas; performance, dado tratar-se de uma linguagem compilada, o que permite manter o nível esperado de performance.

Quanto à informação gravada/lida pelo servidor em ficheiro, à semelhança do escolhido para as mensagens trocadas, optou-se pelo formato JSON, dado a sua simplicidade e fácil integração com a linguagem, graças à biblioteca "JSON for Modern C++" (Nlohmann, 2022).

Quanto ao atendimento simultâneo de múltiplos clientes, este é possível graças à criação de uma thread cada vez que um novo cliente se conecta, a qual executa a função principal, *main_call*, e recebe o socket e endereço do cliente.

```
// Read data from files
read_theaters();
read_clients();
// Wait for connection
sockaddr_in client_addr{};
int client_size = sizeof client_addr;
SOCKET client_socket;
std::list<std::thread> threads;
int i = 0;
while (i++ < max_threads && (client_socket = accept(listening,
reinterpret_cast<sockaddr*>(&client_addr), &client_size)) != INVALID_SOCKET)
{
	threads.emplace_back(main_call, client_socket, client_addr);
}
closesocket(listening);
for (auto& thread : threads)
{
	thread.join();
}
// Save data to files
write_theaters();
write_clients();
```

Já na função principal, o servidor obtém o endereço IP do cliente, criando um perfil para este caso seja a sua primeira conexão, e envia uma mensagem HELLO ao

cliente. De seguida, o servidor passa a processar os pedidos recebidos, executando o procedimento correspondente ao código da mensagem recebida. O ciclo pode terminar caso ocorra algum erro de comunicação, ou o cliente escolha terminar a conexão.

```
std::map<std::string, client> clients;
thread_local std::string ip_addr;

int main_call(const SOCKET client_socket, const sockaddr_in client_addr)
{
	/* client profile and HELLO message */
	while (ret_val > 0)
	{
		// Receive next message and parse it
		char reply[2000];
		ret_val = recv(client_socket, reply, 2000, 0);
		if (ret_val <= 0) break;
		auto msg = json::parse(reply).get<message>();
		log_message(msg, sender::client); // Log message
		// Call proper function, according to message code
		switch (msg.code)
		{
		case code::get_locations:
			ret_val = get_locations(client_socket);
			break;
		case code::get_genres:
			ret_val = get_genres(client_socket, msg.content);
			break;
		case code::get_shows:
			ret_val = buy_tickets(client_socket, msg.content);
			break;
		case code::quit:
			ret_val = quit_call(client_socket);
			break;
		case code::hello:
			std::cout << msg.content << "\n\n";
			break;
		case code::buy_tickets:
			break;
		}
	}
	// Close the socket
	closesocket(client_socket);
	return 0;
}
```

Após o cliente escolher a localização pretendida e o género de espetáculos desejado, o servidor prossegue para a função *buy_tickets*, responsável por enviar os espetáculos disponíveis ao cliente e por processar a compra.

O acesso sequencial à informação dos teatros e espetáculos foi conseguido com o recurso a mutexes, assegurando que cada cliente apenas consegue consultar a informação mais atualizada sobre os espetáculos ainda disponíveis.

```
std::map<std::string, client> clients;
std::mutex tickets_mutex;
thread_local std::string ip_addr;

int buy_tickets(const SOCKET& client_socket, const std::string& content)
{
	// Lock access to shows
	std::lock_guard guard(tickets_mutex);
	// Send available shows to client
	std::list<theater>::iterator it;
	int ret_val = get_shows(client_socket, content, it);
	if (ret_val < 0) return SOCKET_ERROR;
	// Get ticket info from client
	char reply[2000];
	ret_val = recv(client_socket, reply, 2000, 0);
	if (ret_val <= 0) return SOCKET_ERROR;
	// Parse ticket info
	auto msg = json::parse(reply).get<message>();
	log_message(msg, sender::client); // Log message
	json j = json::parse(msg.content);
	int id, no_tickets;
	j.at("id").get_to(id);
	j.at("no_tickets").get_to(no_tickets);
	if (id != -1 && no_tickets != -1)
	{
		// Update client's seen shows
		clients[ip_addr].shows_seen.emplace(id, no_tickets);
		// Update available seats
		const auto show_it = std::ranges::find_if((*it).shows,
			[&](const show& s) { return s.id == id; });
		(*show_it).available_seats -= no_tickets;
	}
	return ret_val;
}
```

Por outro lado, na função *get_shows*, após identificar o teatro pretendido, são os filtrados os espetáculos de acordo com o seu género e lotação, recomendando apenas aqueles aos quais o cliente ainda não assistiu, ou seja, comprou bilhetes.

```
// Filter shows
std::list<show> shows;
std::ranges::copy_if((*it).shows, std::back_inserter(shows),
	[&](const show& s) {
		return s.available_seats > 0 && s.genre == genre
		&& !clients[ip_addr].shows_seen.contains(s.id); });
```

Todas as mensagens trocadas entre o cliente e o servidor são registadas em logs, identificados pelo dia da mensagem, por exemplo “2022-05-07.log”, no diretório *theater_logs*. Para garantir que dois threads não tentam aceder ao mesmo ficheiro de log simultaneamente, recorreu-se a mutexes.

```
std::mutex log_mutex;
thread_local std::string ip_addr;

void log_message(const message& msg, const sender sender)
{
	// Lock access to logs
	std::lock_guard guard(log_mutex);
	std::ostringstream ss;
	ss << R"(.\theater_logs\)" << std::put_time(&msg.stamp, "%Y-%m-%d") <<
".log";
	// Log message
	std::ofstream ofs{ ss.str(), std::ios_base::app };
	ofs << std::put_time(&msg.stamp, message::fmt_str) << ',';
	switch (sender)
	{
	case sender::client:
		ofs << "From:";
		break;
	case sender::server:
		ofs << "To:";
		break;
	}
	ofs << ip_addr << ',';
	ofs << codename.at(msg.code) << ',';
	ofs << msg.content << '\n';
}
```

4. NOTAS FINAIS

Concluído o presente trabalho prático, todos os objetivos propostos foram cumpridos, pelo que a solução apresentada engloba todas as funcionalidades pretendidas, incorporando algumas alterações, devidamente justificadas, implementadas pelos elementos do grupo.

O desenvolvimento deste projeto permitiu adquirir competências e conceitos relacionados com a API Winsock 2, bem como aprofundar o nosso conhecimento sobre a linguagem C++.

De referir que em anexo encontra-se o código-fonte do cliente e do servidor, contudo foi tomada a decisão de não colocar o código referente às classes que suportam os projetos, dado este ser extenso e o seu teor já ter sido exposto em capítulos anteriores.

REFERÊNCIAS

Lohmann, N. (2022). JSON for Modern C++ (Version 3.10.5) [Computer software]. https://github.com/nlohmann

ANEXO A – CÒDIGO-FONTE DO CLIENTE

1. main.cpp

```
#include "utils.h"
#include <WS2tcpip.h>

constexpr auto ds_test_port = static_cast<u_short>(68000);

int main()
{
	WSADATA wsa;
	SetConsoleCP(CP_UTF8);
	// Initialise winsock
	std::cout << "\nInitializing Winsock...";
	if (WSAStartup(MAKEWORD(2, 2), &wsa) != 0)
	{
		std::cerr << "Failed. Error Code : " << WSAGetLastError() <<
'\n';
		return 1;
	}
	std::cout << "Initialized.\n";
	// Create a socket
	const SOCKET server_socket = socket(AF_INET, SOCK_STREAM, 0);
	if (server_socket == INVALID_SOCKET)
	{
		std::cerr << "Could not create socket : " << WSAGetLastError()
<< '\n';
		return 1;
	}
	std::cout << "Socket created.\n";
	// Ask client for server IP address
	std::string ip_addr;
	std::cout << "Server IP address: ";
	getline(std::cin, ip_addr);
	if (!validate_ip(ip_addr))
	{
		ip_addr = "172.30.217.49";
		std::cout << "Invalid IP address.\n";
		std::cout << "IP address will default to " << ip_addr << ".\n";
	}
	// create the socket address (IP address and port)
	sockaddr_in server_addr{};
	server_addr.sin_family = AF_INET;
	server_addr.sin_port = htons(ds_test_port);
	inet_pton(server_addr.sin_family, ip_addr.data(),
&server_addr.sin_addr.s_addr);
	// Connect to remote server
	if (connect(server_socket, reinterpret_cast<sockaddr*>(&server_addr),
sizeof server_addr) == SOCKET_ERROR)
	{
		std::cout << "Connection error\n";
		return 1;
	}
	std::cout << "Connected\n";
	main_call(server_socket);
	// Cleanup winsock
	WSACleanup();
	return 0;
}
```

2. utils.h

```
#pragma once
#include <list>
#include <regex>
#include <set>
#include <string>
#include <WinSock2.h>
#include "message.h"
#include "show.h"

#pragma comment(lib, "ws2_32.lib")

extern std::list<show> shows;

/**
 * @brief Main server call function
 * @param server_socket: server socket
 * @return integer: the call state
*/
int main_call(const SOCKET& server_socket);
/**
 * @brief Validates IP address
 * @param ip_address: IP address
 * @return boolean: whether IP address is valid or not
*/
bool validate_ip(const std::string& ip_address);
/**
 * @brief Asks client for location
 * @param server_socket: server socket
 * @param location: string to store location in
 * @return integer: call state
*/
int pick_location(const SOCKET& server_socket, std::string& location);
/**
 * @brief Asks client for show genre
 * @param server_socket: server socket
 * @param location: string containing location picked
 * @param genre: string to store genre in
 * @return integer: call state
*/
int pick_genre(const SOCKET& server_socket, const std::string& location,
std::string& genre);
/**
 * @brief Asks client to pick show and number of tickets
 * @return pair of integers: show ID and number of tickets
*/
std::pair<int, int> pick_show();
/**
 * @brief Buy tickets
 * @param server_socket: server socket
 * @return integer: call state
*/
int buy_tickets(const SOCKET& server_socket);
/**
 * @brief Asks client if he wishes to quit the call
 * @param server_socket: server socket
 * @return integer: 0 if client wishes to continue
*/
int quit_call(const SOCKET& server_socket);
```

3. utils.cpp

```
#include "utils.h"

std::list<show> shows;

bool validate_ip(const std::string& ip_address)
{
	const std::regex re(R"(^(([0-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-
9]|25[0-5])\.){3})"
		"([0-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-9]|25[0-5])$");
	return std::regex_match(ip_address, re);
}

int pick_location(const SOCKET& server_socket, std::string& location)
{
	// Send request for available locations
	const json j = message(code::get_locations, "");
	const auto msg = j.dump();
	int ret_val = send(server_socket, msg.data(),
static_cast<int>(msg.length()) + 1, 0);
	if (ret_val <= 0) return SOCKET_ERROR;
	// Receive available locations
	char reply[2000];
	ret_val = recv(server_socket, reply, 2000, 0);
	if (ret_val <= 0) return SOCKET_ERROR;
	// Parse response
	auto msg2 = json::parse(reply).get<message>();
	std::set<std::string> locations;
	json::parse(msg2.content).get_to(locations);
	// Pick location
	const auto n = static_cast<int>(locations.size());
	int option = 0;
	do {
		std::cout << "Available locations:\n";
		int i = 0;
		std::ranges::for_each(locations,
			[&](const auto& l) { std::cout << '\t' << i << " -> " <<
l << '\n'; i++; });
		std::cout << "Option: ";
		(std::cin >> option).ignore();
	} while (option < 0 || option >= n);
	location = *std::next(locations.begin(), option);
	return 0;
}

int pick_genre(const SOCKET& server_socket, const std::string& location,
std::string& genre)
{
	// Send request for available genres
	const json j = message(code::get_genres, location);
	const auto msg = j.dump();
	int ret_val = send(server_socket, msg.data(),
static_cast<int>(msg.length()) + 1, 0);
	if (ret_val <= 0) return SOCKET_ERROR;
	// Receive available genres
	char reply[2000];
	ret_val = recv(server_socket, reply, 2000, 0);
	if (ret_val <= 0) return SOCKET_ERROR;
	// Parse response
	auto msg2 = json::parse(reply).get<message>();
	std::set<std::string> genres;
```

```
	json::parse(msg2.content).get_to(genres);
	// Pick genre
	const auto n = static_cast<int>(genres.size());
	int option = 0;
	do {
		std::cout << "Available genres:\n";
		int i = 0;
		std::ranges::for_each(genres, [&](const auto& g) { std::cout <<
'\t' << i << " -> " << g << '\n'; i++; });
		std::cout << "Option: ";
		(std::cin >> option).ignore();
	} while (option < 0 || option >= n);
	genre = *std::next(genres.begin(), option);
	return 0;
}

std::pair<int, int> pick_show()
{
	std::pair show_info(-1, -1);
	// Check no of shows
	if (shows.empty())
	{
		std::cout << "No available shows.\n";
		return show_info;
	}
	// Pick show from available shows
	std::cout << "Available shows: \n";
	std::ranges::for_each(shows, [](const show& s) { s.write(); std::cout
<< '\n'; });
	int id;
	std::cout << "Choose show (id): ";
	(std::cin >> id).ignore();
	const auto it = std::ranges::find_if(shows,
		[&](const show& s) { return s.id == id; });
	if (it == shows.end())
	{
		std::cout << "No show with id " << id << " is available.\n";
		return show_info;
	}
	show_info.first = id;
	// Ask how many tickets the client wants
	int no_tickets;
	std::cout << "How many tickets? ";
	(std::cin >> no_tickets).ignore();
	if (no_tickets <= 0)
	{
		if (no_tickets < 0)
			std::cout << "Invalid number of tickets.\n";
		return show_info;
	}
	if (no_tickets > (*it).available_seats)
	{
		std::cout << "Not enough tickets available.\n";
		return show_info;
	}
	show_info.second = no_tickets;
	return show_info;
}

int buy_tickets(const SOCKET& server_socket)
{
	// Pick location
	std::string location;
```

```
	int ret_val = pick_location(server_socket, location);
	if (ret_val < 0) return ret_val;
	// Pick genre
	std::string genre;
	ret_val = pick_genre(server_socket, location, genre);
	if (ret_val < 0) return ret_val;
	// Ask for shows
	const json j = message(code::get_shows, json{ {"location", location},
	                                              {"genre", genre} }.dump());
	auto s = j.dump();
	ret_val = send(server_socket, s.data(), static_cast<int>(s.length()) +
1, 0);
	if (ret_val <= 0) return SOCKET_ERROR;
	// Parse response and get shows
	char reply[2000];
	ret_val = recv(server_socket, reply, 2000, 0);
	if (ret_val <= 0) return SOCKET_ERROR;
	auto msg = json::parse(reply).get<message>();
	json::parse(msg.content).get_to(shows);
	// Pick show
	auto [id, no_tickets] = pick_show();
	if (id == -1 || no_tickets == -1)
	{
		std::cout << "Error occurred during show pick.\n";
	}
	// Send show/ticket info
	shows.clear(); // Make sure shows is cleared
	const json k = message(code::buy_tickets, json{ {"id", id},
	                                                {"no_tickets", no_tickets}
}.dump());
	s = k.dump();
	ret_val = send(server_socket, s.data(), static_cast<int>(s.length()) +
1, 0);
	if (ret_val <= 0) return SOCKET_ERROR;
	return 0;
}

int main_call(const SOCKET& server_socket)
{
	// Receive HELLO message
	char reply[2000];
	int ret_val = recv(server_socket, reply, 2000, 0);
	if (ret_val <= 0)
	{
		closesocket(server_socket);
		return ret_val;
	}
	const auto msg = json::parse(reply).get<message>();
	std::cout << msg.content << "\n\n";
	// Main menu
	while (ret_val != SOCKET_ERROR)
	{
		int option;
		std::cout << "Menu:\n";
		std::cout << "\t1 -> Buy tickets\n";
		std::cout << "\t2 -> Quit\n";
		std::cout << "Option: ";
		(std::cin >> option).ignore();
		switch (option)
		{
		case 1:
			ret_val = buy_tickets(server_socket);
			break;
```

```
                case 2:
                        ret_val = quit_call(server_socket);
                        break;
                default:
                        std::cout << "Invalid option. Please try again.\n";
                        break;
                }
        }
        // Close the socket
        closesocket(server_socket);
        return ret_val;
}

int quit_call(const SOCKET& server_socket)
{
        // Confirm quit
        std::string option;
        std::cout << "Do you wish to quit? (y/n)\n";
        std::cout << "Option: ";
        getline(std::cin, option);
        if (option == "y")
        {
                // Send request to end call
                const json j = message(code::quit, "QUIT");
                const auto s = j.dump();
                int ret_val = send(server_socket, s.data(),
static_cast<int>(s.length()) + 1, 0);
                if (ret_val <= 0) return SOCKET_ERROR;
                // Receive response to end call
                char reply[2000];
                ret_val = recv(server_socket, reply, 2000, 0);
                if (ret_val <= 0) return SOCKET_ERROR;
                const auto msg = json::parse(reply).get<message>();
                std::cout << msg.content << '\n';
                return -1;
        }
        return 0;
}
```

ANEXO B – CÒDIGO-FONTE DO SERVIDOR

1. main.cpp

```
#include "utils.h"
#include <thread>

constexpr auto ds_test_port = static_cast<u_short>(68000);
constexpr auto max_threads = 5;

int main()
{
	SetConsoleOutputCP(CP_UTF8);
	// Initialise winsock
	WSADATA ws_data;
	std::cout << "Initializing Winsock...\n";
	if (WSAStartup(MAKEWORD(2, 2), &ws_data) != 0)
	{
		std::cerr << "\nWinsock setup failed! Error Code : " <<
WSAGetLastError() << '\n';
		return 1;
	}
	// Create a socket
	const SOCKET listening = socket(AF_INET, SOCK_STREAM, 0);
	if (listening == INVALID_SOCKET)
	{
		std::cerr << "\nSocket creation failed! Error Code : " <<
WSAGetLastError() << '\n';
		return 1;
	}
	std::cout << "\nSocket created.\n";
	// Bind the socket (ip address and port)
	sockaddr_in hint{};
	hint.sin_family = AF_INET;
	hint.sin_port = htons(ds_test_port);
	hint.sin_addr.S_un.S_addr = INADDR_ANY;
	if (bind(listening, reinterpret_cast<sockaddr*>(&hint), sizeof hint)
== SOCKET_ERROR)
	{
		std::cerr << "\nAddress binding failed! Error Code : " <<
WSAGetLastError() << '\n';
		return 1;
	}
	// Setup the socket for listening
	if(listen(listening, SOMAXCONN) == SOCKET_ERROR)
	{
		std::cerr << "\nListening failed! Error Code : " <<
WSAGetLastError() << '\n';
		return 1;
	}
	// Read data from files
	read_theaters();
	read_clients();
	// Wait for connection
	sockaddr_in client_addr{};
	int client_size = sizeof client_addr;
	SOCKET client_socket;
	std::list<std::thread> threads;
	int i = 0;
```

```
	while (i++ < max_threads && (client_socket = accept(listening,
reinterpret_cast<sockaddr*>(&client_addr), &client_size)) != INVALID_SOCKET)
	{
		threads.emplace_back(main_call, client_socket, client_addr);
	}
	closesocket(listening);
	for (auto& thread : threads)
	{
		thread.join();
	}
	// Save data to files
	write_theaters();
	write_clients();
	// Cleanup winsock
	WSACleanup();
	return 0;
}
```

2. utils.h

```
#pragma once
#include <fstream>
#include <mutex>
#include <set>
#include <WinSock2.h>
#include <WS2tcpip.h>
#include "client.h"
#include "message.h"
#include "theater.h"

#pragma comment(lib, "ws2_32.lib")

extern std::list<theater> theaters;
extern std::map<std::string, client> clients;
extern std::mutex tickets_mutex;
extern std::mutex log_mutex;
extern thread_local std::string ip_addr;

enum struct sender
{
	client,
	server
};
/**
 * @brief Logs message
 * @param msg: message send/received
 * @param sender: CLIENT or SERVER
*/
void log_message(const message& msg, sender sender);
/**
 * @brief Reads theaters from file
 * @param filename: path to JSON file
*/
void read_theaters(const char* filename = "shows.json");
/**
 * @brief Writes theaters to file
 * @param filename: path to JSON file
*/
void write_theaters(const char* filename = "shows.json");
/**
 * @brief Reads clients from file
 * @param filename: path to JSON file
*/
void read_clients(const char* filename = "clients.json");
/**
 * @brief Writes clients to file
 * @param filename: path to JSON file
*/
void write_clients(const char* filename = "clients.json");
/**
 * @brief Sends available locations to client
 * @param client_socket: client socket
 * @return integer: value of send
*/
int get_locations(const SOCKET& client_socket);
/**
 * @brief Sends available genres to client
 * @param client_socket: client socket
 * @param location: location received
 * @return integer: value of send
```

```
*/
int get_genres(const SOCKET& client_socket, const std::string& location);
/**
 * @brief Sends available shows to client, given location and genre
 * @param client_socket: client socket
 * @param content: message content
 * @param it: iterator pointing to chosen theater
 * @return integer: value of send
*/
int get_shows(const SOCKET& client_socket, const std::string& content,
std::list<theater>::iterator& it);
/**
 * @brief Processes client's ticket purchase
 * @param client_socket: client socket
 * @param content: last message content
 * @return
*/
int buy_tickets(const SOCKET& client_socket, const std::string& content);
/**
* @brief Main client call function
* @param client_socket: client socket
* @param client_addr: socket address
* @return integer: the call state
*/
int main_call(SOCKET client_socket, sockaddr_in client_addr);
/**
 * @brief Quits call with client
 * @param client_socket: client socket
 * @return integer: send return value
*/
int quit_call(const SOCKET& client_socket);
```

3. utils.cpp

```
#include "utils.h"

std::list<theater> theaters;
std::map<std::string, client> clients;
std::mutex tickets_mutex;
std::mutex log_mutex;
thread_local std::string ip_addr;

void log_message(const message& msg, const sender sender)
{
	// Lock access to logs
	std::lock_guard guard(log_mutex);
	// Get log file path
	std::ostringstream ss;
	ss << R"(.\theater_logs\)" << std::put_time(&msg.stamp, "%Y-%m-%d") <<
".log";
	// Log message
	std::ofstream ofs{ ss.str(), std::ios_base::app };
	ofs << std::put_time(&msg.stamp, message::fmt_str) << ',';
	switch (sender)
	{
	case sender::client:
		ofs << "From:";
		break;
	case sender::server:
		ofs << "To:";
		break;
	}
	ofs << ip_addr << ',';
	ofs << codename.at(msg.code) << ',';
	ofs << msg.content << '\n';
}

void read_theaters(const char* filename)
{
	std::ifstream ifs{ filename };
	json j;
	ifs >> j;
	j.get_to(theaters);
}

void write_theaters(const char* filename)
{
	std::ofstream ofs{ filename };
	const json j = theaters;
	ofs << std::setw(4) << j;
}

void read_clients(const char* filename)
{
	std::ifstream ifs{ filename };
	json j;
	ifs >> j;
	j.get_to(clients);
}

void write_clients(const char* filename)
{
	std::ofstream ofs{ filename };
	const json j = clients;
```

```
	ofs << std::setw(4) << j;
}

int get_locations(const SOCKET& client_socket)
{
	// Make set of (non-repeated) locations
	std::set<std::string> locations;
	for (auto& theater : theaters)
	{
		locations.insert(theater.location);
	}
	// Send locations to client
	const json j = locations;
	message msg(code::get_locations, j.dump());
	const json k = msg;
	const auto s = k.dump();
	const int ret_val = send(client_socket, s.data(),
static_cast<int>(s.length()) + 1, 0);
	log_message(msg, sender::server); // Log message
	return ret_val;
}

int get_genres(const SOCKET& client_socket, const std::string& location)
{
	// Find theater in given location
	const auto it = std::ranges::find_if(theaters, [&](const theater& t)
		{ return t.location == location; });
	if (it == theaters.end()) return 0;
	// Make set of (non-repeated) genres
	std::set<std::string> genres;
	for (auto& show : (*it).shows)
	{
		genres.insert(show.genre);
	}
	// Send genres to client
	const json j = genres;
	message msg(code::get_genres, j.dump());
	const json k = msg;
	const auto s = k.dump();
	const int ret_val = send(client_socket, s.data(),
static_cast<int>(s.length()) + 1, 0);
	log_message(msg, sender::server); // Log message
	if (ret_val <= 0) return SOCKET_ERROR;
	return 0;
}

int get_shows(const SOCKET& client_socket, const std::string& content,
std::list<theater>::iterator& it)
{
	// Parse message content for location and genre
	json j = json::parse(content);
	std::string location, genre;
	j.at("location").get_to(location);
	j.at("genre").get_to(genre);
	// Find theater in given location
	it = std::ranges::find_if(theaters, [&](const theater& t) { return
t.location == location; });
	if (it != theaters.end())
	{
		// Filter shows
		std::list<show> shows;
		std::ranges::copy_if((*it).shows, std::back_inserter(shows),
			[&](const show& s) {
```

```
                        return s.available_seats > 0 && s.genre == genre
                                &&
!clients[ip_addr].shows_seen.contains(s.id); });
            // Send shows to client
            const json js = shows;
            message msg(code::get_shows, js.dump());
            const json k = msg;
            const auto s = k.dump();
            const int ret_val = send(client_socket, s.data(),
static_cast<int>(s.length()) + 1, 0);
            log_message(msg, sender::server); // Log message
            if (ret_val <= 0) return SOCKET_ERROR;
      }
      return 0;
}

int buy_tickets(const SOCKET& client_socket, const std::string& content)
{
      // Lock access to shows
      std::lock_guard guard(tickets_mutex);
      // Send available shows to client
      std::list<theater>::iterator it;
      int ret_val = get_shows(client_socket, content, it);
      if (ret_val < 0) return SOCKET_ERROR;
      // Get ticket info from client
      char reply[2000];
      ret_val = recv(client_socket, reply, 2000, 0);
      if (ret_val <= 0) return SOCKET_ERROR;
      // Parse ticket info
      auto msg = json::parse(reply).get<message>();
      log_message(msg, sender::client); // Log message
      json j = json::parse(msg.content);
      int id, no_tickets;
      j.at("id").get_to(id);
      j.at("no_tickets").get_to(no_tickets);
      if (id != -1 && no_tickets != -1)
      {
            // Update client's seen shows
            clients[ip_addr].shows_seen.emplace(id, no_tickets);
            // Update available seats
            const auto show_it = std::ranges::find_if((*it).shows,
                  [&](const show& s) { return s.id == id; });
            (*show_it).available_seats -= no_tickets;
      }
      return ret_val;
}

int main_call(const SOCKET client_socket, const sockaddr_in client_addr)
{
      // Get client's IP address
      char buf[20];
      inet_ntop(client_addr.sin_family, &client_addr.sin_addr, buf, 20);
      ip_addr = std::string(buf);
      if (!clients.contains(ip_addr))
      {
            clients.emplace(ip_addr, client(ip_addr));
      }
      // Send HELLO message
      message m(code::hello, "100 OK");
      const json j = m;
      const auto s = j.dump();
      int ret_val = send(client_socket, s.data(),
static_cast<int>(s.length()) + 1, 0);
```

```
	log_message(m, sender::server); // Log message
	std::cout << "Hello, client!\n\n";
	while (ret_val > 0)
	{
		// Receive next message and parse it
		char reply[2000];
		ret_val = recv(client_socket, reply, 2000, 0);
		if (ret_val <= 0) break;
		auto msg = json::parse(reply).get<message>();
		log_message(msg, sender::client); // Log message
		// Call proper function, according to message code
		switch (msg.code)
		{
		case code::get_locations:
			ret_val = get_locations(client_socket);
			break;
		case code::get_genres:
			ret_val = get_genres(client_socket, msg.content);
			break;
		case code::get_shows:
			ret_val = buy_tickets(client_socket, msg.content);
			break;
		case code::quit:
			ret_val = quit_call(client_socket);
			break;
		case code::hello:
			std::cout << msg.content << "\n\n";
			break;
		case code::buy_tickets:
			break;
		}
	}
	// Close the socket
	closesocket(client_socket);
	return 0;
}

int quit_call(const SOCKET& client_socket)
{
	// Send 400 BYE message to client
	message msg(code::quit, "400 BYE");
	const json j = msg;
	const auto s = j.dump();
	send(client_socket, s.data(), static_cast<int>(s.length()) + 1, 0);
	log_message(msg, sender::server); // Log message
	std::cout << "Bye, client...\n\n";
	return 0;
}
```