Михайлович Олександр ТВ-22

Практична робота №1.3

Програмне забезпечення мереж

**Інтерфейс**

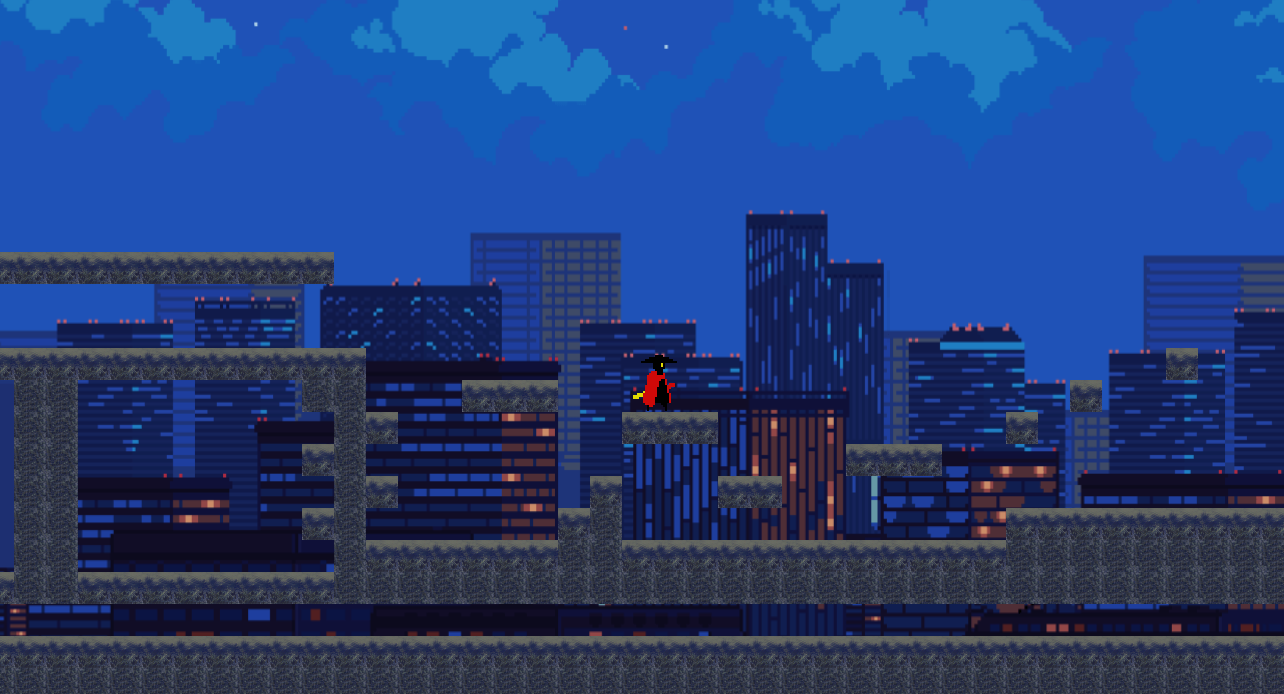


Рисунок 1. Персонаж не рухається

A video game with two people on a bridge

Description automatically generated

Рисунок 2. Персонаж у повітрі

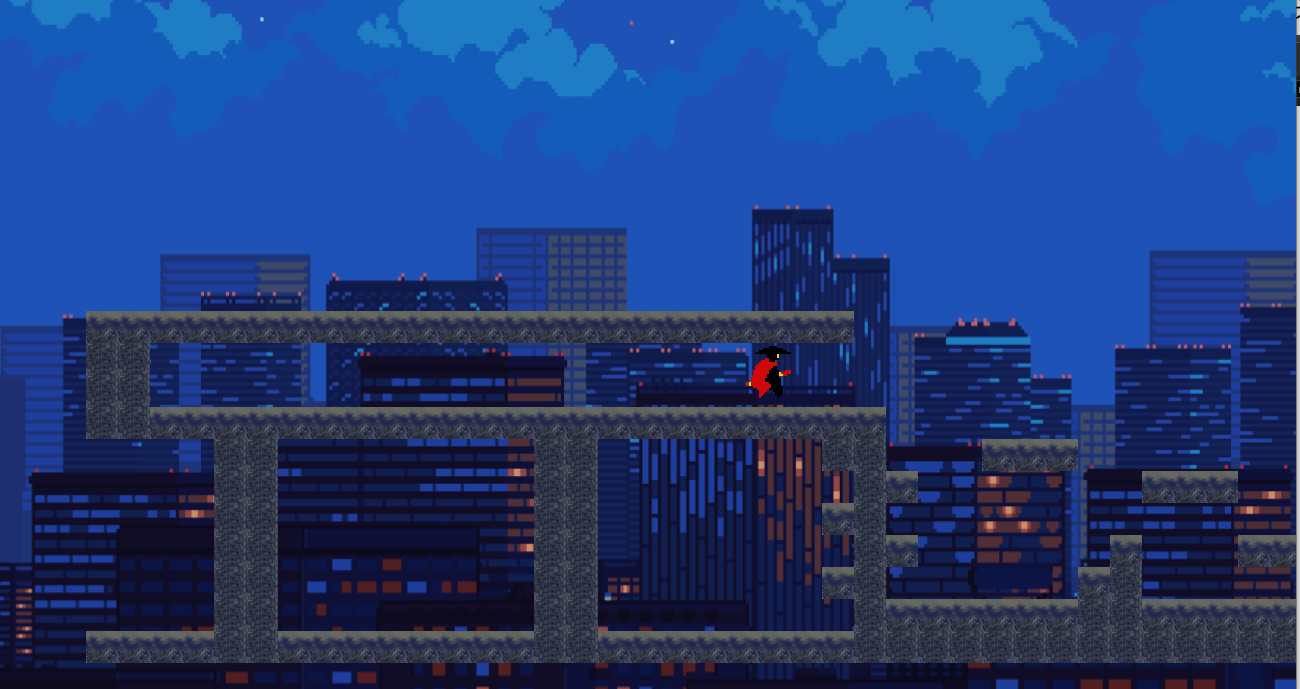


Рисунок 3. Персонаж біжить

Для створення мапи розроблено систему позначок – тому зміна мапи може бути виконана в окремому текстовому файлі.

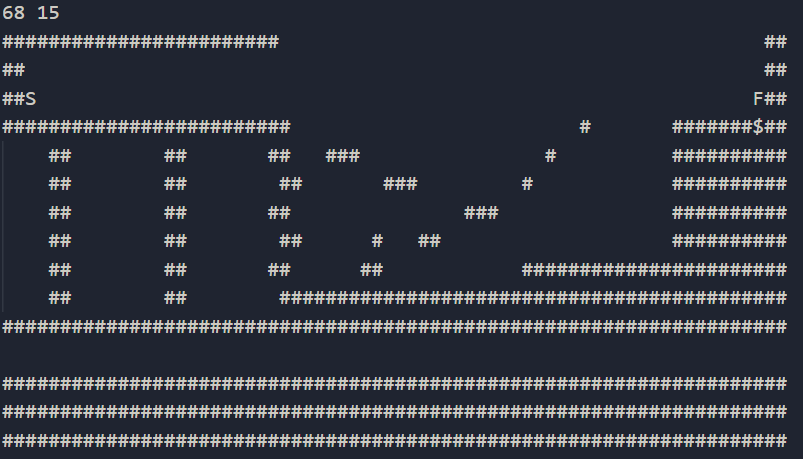


Рисунок 4. Мапа в текстовому форматі

**Програмна реалізація**

Було реалізовано клієнт-серверну архітектуру на мові C++. Для проміжного тестування було написано термінальний додаток - клієнт на python.

Для пересилання пакетів (у сенсі коди для виправлення – crc32 хешування й навпаки розхешування й відновлення) було написано бібліотеку (.dll) на C++ з додатковими вхідними точками у форматі C (було використано в python клієнті). Проєкт виконано у форматі CMake проєкту з трьома підпроєктами – клієнт, сервер й бібліотека.

У клієнтській частині для вираховування фізики використовується система з 3-х векторів – позиції, швидкості й прискорення. У клієнт-серверній взаємодії налаштована (що виходить із наявності фізики) система інтерполяції (також підготовлена база для написання екстраполяції – що теоретично має ще більш покращити роботу системи).

«Двигун» гри зроблено таким чином – що доволі просто можна змінити частоту обробки фізики рухомих персонажів (наприклад 30ms – приблизно 33 рази в секунду (використовується/може використовуватись гравцем), 100ms – 10 разів в секунду (може використовуватись сервером, й відповідно мережею)).

Для мережі є можливість надсилати змінну кількість параметрів – пакети представлені у власному бінарному форматі (наявний прапор для типу пакету та наприклад може бути наявним число, що відповідає кількості певних повторюваних елементів).

Як основу для відображення було використано родину бібліотек/фреймворк SDL3. Розробка виконана на MinGW.

Використано систему наслідування (наприклад персонаж гравця це клас гравець<->об’єкт що рухається<->об’єкт).

Для обробки колізії (й системи шляху – дозволяє оновлювати персонажа не кожного разу, а накопичувати певну кількість часу (delta time)) використано алгоритм DTA.

Що клієнт, що сервер має декілька потоків – від цього з’явилась також необхідність обробляти race conditions.

На мапі можна побачити відсутність визначення звичайного блоку й блоку поверхні (порівняймо різні типи блоків на зображеннях) – тип блоку визначається автоматично за зчитування мапи.

Під час малювання виконано перевірки на область видимості – тому дуже далекі об’єкти (приблизно 1-6 блоків поза зоною камери) не надсилаються для відображення.

Математики в системі представлена у вигляді векторів (як вже було зазначено) та дуалістичної позиції – це означає, що позиція деяких об’єктів має не один, а два вектори – цілий вектор (вектор цілих чисел) і реальний вектор (вектор з плаваючими комами). Перший тип вектору використовується для визначення відношень, колізій й рендерінгу – у той час як другий покращує точність пересування (яке потім транслюється в цілий вектор для перевірок).

Використано «смузінг» (smooth, згладжування) камери. Камера прив’язана до персонажу – проте не є статично прив’язаною. Навколо персонажу є блок повз який не центр камери вийти не може – центр камери математично «прив’язаний» до центру персонажу (камера рухається не з персонажем, а за персонажем).

Виконано деякі перевірки (значна кількість) на мережеве з’єднання.

Сервер працює в режимі слухача й обробника/письменника – слухач зчитує дані й надає їх обробнику, письменник очікує даних від обробника (використано алгоритм spin-lock для зменшення затримки за рахунок збільшення вимог до сервера).

Клієнт займає не значну кількість процесору (на мої системі це було в проміжка 1.5-4.5 відсотки навантаження).

File: Network.cpp (Server)

001: #include "Network.h"

002: #include <cstdint>

003: #include <iostream>

004: #include <mutex>

005: #include <optional>

006: #include <string>

007: #include <array>

008: #include <winsock2.h>

009: #include <syncstream>

010: #include <ws2ipdef.h>

011: #include <error\_repairing.h>

012:

013: using std::to\_string;

014:

015: Network::Network() {

016: WSADATA wsaData;

017: if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0) {

018: std::osyncstream(std::cerr) << "Socket startup failed: " << WSAGetLastError() << '\n';

019: exit(1);

020: }

021: std::osyncstream(std::cout) << "Socket startup success" << '\n';

022: }

023:

024: Network::~Network () {

025: if (WSACleanup() != 0) {

026: std::osyncstream(std::cerr) << "Socket shutdown failed: " << WSAGetLastError() << '\n';

027: exit(1);

028: }

029: std::osyncstream(std::cout) << "Socket shutdown success" << '\n';

030: }

031:

032: bool Network::setup\_socket () {

033: // get server info

034: addrinfo hints;

035: memset(&hints, 0, sizeof(hints));

036: hints.ai\_family = AF\_INET; // AF\_INET or AF\_INET6 (IPv4 or IPv6)

037: hints.ai\_socktype = SOCK\_DGRAM; // UDP

038: hints.ai\_protocol = 0; // use default protocol implementation for UDP

039: hints.ai\_flags = AI\_PASSIVE; // used resolution for socket creation and binding

040: addrinfo\* result;

041: int error = getaddrinfo(nullptr, "20123", &hints, &result);

042: if (error != 0) {

043: std::osyncstream(std::cerr) << "Self address resolution failed: " << gai\_strerror(error) << '\n';

044: return false;

045: }

046:

047: // UDP socket

048: m\_server\_socket = socket(result->ai\_family, result->ai\_socktype, result->ai\_protocol);

049: if (m\_server\_socket == INVALID\_SOCKET) {

050: std::osyncstream(std::cerr) << "Socket creation failed: " << WSAGetLastError() << '\n';

051: freeaddrinfo(result);

052: closesocket(m\_server\_socket);

053: return false;

054: }

055:

056: // set socket options

057: int opt = 1;

058: if (setsockopt(m\_server\_socket, SOL\_SOCKET, SO\_REUSEADDR, (char\*)&opt, sizeof(opt)) == SOCKET\_ERROR) {

059: std::osyncstream(std::cerr) << "Socket modification failed: " << WSAGetLastError() << '\n';

060: freeaddrinfo(result);

061: closesocket(m\_server\_socket);

062: return false;

063: }

064:

065: // binding

066: error = bind(m\_server\_socket, result->ai\_addr, int(result->ai\_addrlen));

067: freeaddrinfo(result); // freeing memory - unneaded after binding

068: if (error == SOCKET\_ERROR) {

069: std::osyncstream(std::cerr) << "Socket binding failed: " << WSAGetLastError() << '\n';

070: closesocket(m\_server\_socket);

071: return false;

072: }

073:

074: return true;

075: }

076:

077: void Network::process\_error (const SOCKADDR\_IN& client\_address) {

078: int wsa\_error = WSAGetLastError();

079:

080: // this seems to be get when client is disconnects - in connection less protocol ;(

081: if (wsa\_error == 10054) return; // FIXME try to find a const

082:

083: std::osyncstream(std::cerr) << "Socket receive failed: " << wsa\_error << '\n' << '\n';

084: response\_bad\_formed(inet\_ntoa(client\_address.sin\_addr), std::to\_string(client\_address.sin\_port));

085: }

086:

087: void Network::socket\_main () {

088: // setup

089: if (not setup\_socket()) return;

090:

091: // get data cycle

092: SOCKADDR\_IN client\_address;

093: int client\_address\_size = sizeof(client\_address);

094: std::array<char, 1024> buffer;

095: int buffer\_size = static\_cast<int>(buffer.size());

096:

097: while (server\_running) {

098: memset(&client\_address, 0, sizeof(client\_address));

099: int data\_size = recvfrom(m\_server\_socket, buffer.data(), buffer\_size, 0, (SOCKADDR\*)&client\_address, &client\_address\_size);

100: client\_address.sin\_port = htons(client\_address.sin\_port);

101:

102: if (data\_size == SOCKET\_ERROR) {

103: process\_error(client\_address);

104: continue;

105: }

106:

107: push\_message(inet\_ntoa(client\_address.sin\_addr),

108: to\_string(client\_address.sin\_port),

109: std::vector<uint8\_t>(buffer.data(), buffer.data() + static\_cast<size\_t>(data\_size)));

110: }

111:

112: // cleaning

113: closesocket(m\_server\_socket);

114: }

115:

116:

117: // ===================================

118: // start of thread-made functions

119: // ===================================

120:

121: bool Network::is\_server\_running() const {

122: return server\_running;

123: }

124:

125: void Network::stop\_server () {

126: if (not server\_running) return;

127: server\_running = false;

128: }

129:

130:

131: bool Network::has\_message() {

132: return not messages.empty();

133: }

134:

135: bool Network::decode\_message (std::vector<uint8\_t>& message) {

136: return custom\_utils::decode\_package(message);

137: }

138:

139: std::optional<Network\_package> Network::pop\_message () {

140: std::unique\_lock<std::mutex> lock(mutex\_messages); // use message queue

141:

142: cv\_has\_message.wait(lock, [this] { return not messages.empty() or not server\_running; });

143: if (messages.empty()) { // server was stopped

144: std::osyncstream(std::cout) << "Attempt to pop message from empty queue when server was stopped" << '\n';

145: return std::nullopt;

146: }

147:

148: Raw\_message raw\_message = messages.front();

149: messages.pop\_front();

150:

151: // decode

152: bool was\_decoded = decode\_message(raw\_message.package);

153: if (not was\_decoded) {

154: // change to error

155: response\_bad\_formed(raw\_message.ip, raw\_message.port);

156: return std::nullopt;

157: }

158:

159: // parse

160: std::optional<Package> package = parse\_message(raw\_message.package);

161: if (not package.has\_value()) {

162: // change to error

163: response\_bad\_formed(raw\_message.ip, raw\_message.port);

164: return std::nullopt;

165: };

166:

167: return Network\_package{.package=package.value(), .ip=std::move(raw\_message.ip), .port=std::move(raw\_message.port)};

168: }

169:

170: // TODO: make atomic dequeue

171: void Network::push\_message (std::string&& client, std::string&& port, std::vector<uint8\_t>&& message) {

172: std::lock\_guard<std::mutex> lock(mutex\_messages);

173: messages.emplace\_back(std::move(message), std::move(client), std::move(port));

174: cv\_has\_message.notify\_one();

175: return;

176: }

177:

178: // ===================================

179: // end of thread-made functions

180: // ===================================

181:

182: void Network::registered\_acknowledge (const std::string& ip, const std::string& port) {

183: Answer answer;

184: answer.type = Answer::Type::REGISTERED\_ANSWER;

185: answer.finish = std::nullopt;

186: answer.other = std::nullopt;

187: answer.id = std::nullopt;

188:

189: send\_answer(ip, port, answer);

190: }

191:

192: void Network::acknowledge (const std::string& ip, const std::string& port, uint64\_t id) {

193: Answer answer;

194: answer.type = Answer::Type::ACKNOWLEDGE;

195: answer.finish = std::nullopt;

196: answer.other = std::nullopt;

197: answer.id = id;

198:

199: send\_answer(ip, port, answer);

200: }

201:

202: void Network::response\_bad\_formed (const std::string& ip, const std::string& port) {

203: Answer answer;

204: answer.type = Answer::Type::BAD\_FORMED;

205: answer.finish = std::nullopt;

206: answer.other = std::nullopt;

207: answer.id = std::nullopt;

208: send\_answer(ip, port, answer);

209: }

210:

211: void Network::deleted\_acknowledge (const std::string& ip, const std::string& port) {

212: Answer answer;

213: answer.type = Answer::Type::BREAK\_SESSION;

214: answer.finish = std::nullopt;

215: answer.other = std::nullopt;

216: answer.id = std::nullopt;

217: send\_answer(ip, port, answer);

218: }

219:

220: void Network::send\_info\_of\_other (const std::string& ip, const std::string& port, uint64\_t id, const std::vector<Answer::Other>& other) {

221: Answer answer;

222: answer.type = Answer::Type::OTHER;

223: answer.finish = std::nullopt;

224: answer.other = other;

225: answer.id = id;

226:

227: if (other.empty()) answer.other = std::nullopt;

228:

229: for (const auto& x : other) {

230: std::osyncstream(std::cout) << "Sending info of other: " << x.id << '\n';

231: }

232:

233: send\_answer(ip, port, answer);

234: }

235:

236: std::optional<Package> Network::parse\_message (const std::vector<uint8\_t>& message) {

237: // Ensure that the message size is exactly what we expect - must be less than max amount of data

238: if (message.size() > sizeof(uint8\_t) + 2 \* sizeof(uint64\_t) + 6 \* sizeof(double) or message.size() == 0) {

239: return std::nullopt;

240: }

241:

242: Package package;

243: uint64\_t llong\_value;

244: size\_t data\_disposition = 0;

245: const uint8\_t\* data = message.data();

246:

247: // 1. Type (1 byte)

248: if (data\_disposition + sizeof(uint8\_t) > message.size()) return std::nullopt; // no enough data left

249: package.type = static\_cast<Package::Type>(data[0]);

250: if (int(package.type) < Package::MIN\_TYPE || int(package.type) > Package::MAX\_TYPE) {

251: return std::nullopt;

252: }

253: data\_disposition += sizeof(uint8\_t);

254:

255: // End types - types without additional payload

256: bool is\_type\_only\_package = (

257: (package.type == Package::Type::BREAK\_SESSION) or

258: (package.type == Package::Type::LOGIN )

259: );

260:

261: if (is\_type\_only\_package) return package;

262:

263: // 2. Id (8 bytes, network to host order)

264: if (data\_disposition + sizeof(uint64\_t) > message.size()) return std::nullopt; // no enough data left

265: memcpy(&llong\_value, data + data\_disposition, sizeof(llong\_value));

266: package.id = static\_cast<unsigned long>(ntohll(llong\_value));

267: data\_disposition += sizeof(uint64\_t);

268:

269: // Id package - packages with type and id

270: if (package.type == Package::Type::GET\_OTHER) return package;

271:

272: // [3, 9]

273: if (data\_disposition + 6 \* sizeof(double) + sizeof(uint64\_t) != message.size()) return std::nullopt; // no correct amount of data

274:

275: // 3. x (8 bytes, double in network byte order)

276: memcpy(&llong\_value, data + data\_disposition, sizeof(llong\_value));

277: package.x = network\_to\_host\_double(llong\_value);

278: data\_disposition += sizeof(double);

279:

280: // 4. y (8 bytes, double in network byte order)

281: memcpy(&llong\_value, data + data\_disposition, sizeof(llong\_value));

282: package.y = network\_to\_host\_double(llong\_value);

283: data\_disposition += sizeof(double);

284:

285: // 5. dx (8 bytes, double in network byte order)

286: memcpy(&llong\_value, data + data\_disposition, sizeof(llong\_value));

287: package.dx = network\_to\_host\_double(llong\_value);

288: data\_disposition += sizeof(double);

289:

290: // 6. dy (8 bytes, double in network byte order)

291: memcpy(&llong\_value, data + data\_disposition, sizeof(llong\_value));

292: package.dy = network\_to\_host\_double(llong\_value);

293: data\_disposition += sizeof(double);

294:

295: // 7. ddx (8 bytes, double in network byte order)

296: memcpy(&llong\_value, data + data\_disposition, sizeof(llong\_value));

297: package.ddx = network\_to\_host\_double(llong\_value);

298: data\_disposition += sizeof(double);

299:

300: // 8. ddy (8 bytes, double in network byte order)

301: memcpy(&llong\_value, data + data\_disposition, sizeof(llong\_value));

302: package.ddy = network\_to\_host\_double(llong\_value);

303: data\_disposition += sizeof(double);

304:

305: // 9. timestamp (8 bytes, unsigned long long in network order)

306: memcpy(&llong\_value, data + data\_disposition, sizeof(llong\_value));

307: package.time = static\_cast<unsigned long>(ntohll(llong\_value));

308: // Unneaded: data\_disposition += sizeof(uint64\_t);

309:

310: return package;

311: }

312:

313: bool Network::encode\_message (std::vector<uint8\_t>& message) {

314: return custom\_utils::encode\_package(message);

315: }

316:

317: void Network::serialize\_answer (const Answer& answer, std::vector<uint8\_t>& buffer) {

318: uint8\_t type = \*reinterpret\_cast<const uint8\_t\*>(&answer.type);

319: buffer.insert(buffer.end(), reinterpret\_cast<uint8\_t\*>(&type), reinterpret\_cast<uint8\_t\*>(&type) + sizeof(type));

320:

321:

322: if (answer.id.has\_value()) {

323: uint64\_t id = ntohll(answer.id.value());

324: buffer.insert(buffer.end(), reinterpret\_cast<uint8\_t\*>(&id), reinterpret\_cast<uint8\_t\*>(&id) + sizeof(id));

325: }

326: if (answer.finish.has\_value()) {

327: uint8\_t finish = answer.finish->has\_finished;

328: buffer.push\_back(finish);

329:

330: uint64\_t x = ntohll(answer.finish->x);

331: buffer.insert(buffer.end(), reinterpret\_cast<uint8\_t\*>(&x), reinterpret\_cast<uint8\_t\*>(&x) + sizeof(x));

332:

333: uint64\_t y = ntohll(answer.finish->y);

334: buffer.insert(buffer.end(), reinterpret\_cast<uint8\_t\*>(&y), reinterpret\_cast<uint8\_t\*>(&y) + sizeof(y));

335:

336: uint64\_t time = ntohll(answer.finish->time);

337: buffer.insert(buffer.end(), reinterpret\_cast<uint8\_t\*>(&time), reinterpret\_cast<uint8\_t\*>(&time) + sizeof(time));

338: }

339: if (answer.other.has\_value()) {

340: uint64\_t size = ntohll(answer.other.value().size());

341: buffer.insert(buffer.end(), reinterpret\_cast<uint8\_t\*>(&size), reinterpret\_cast<uint8\_t\*>(&size) + sizeof(size));

342:

343: for (const auto& other : answer.other.value()) {

344: uint64\_t value = ntohll(other.id);

345: buffer.insert(buffer.end(), reinterpret\_cast<uint8\_t\*>(&value), reinterpret\_cast<uint8\_t\*>(&value) + sizeof(value));

346:

347: value = ntohll(\*reinterpret\_cast<const unsigned long long\*>(&other.x));

348: buffer.insert(buffer.end(), reinterpret\_cast<uint8\_t\*>(&value), reinterpret\_cast<uint8\_t\*>(&value) + sizeof(value));

349: value = ntohll(\*reinterpret\_cast<const unsigned long long\*>(&other.y));

350: buffer.insert(buffer.end(), reinterpret\_cast<uint8\_t\*>(&value), reinterpret\_cast<uint8\_t\*>(&value) + sizeof(value));

351:

352: value = ntohll(\*reinterpret\_cast<const unsigned long long\*>(&other.dx));

353: buffer.insert(buffer.end(), reinterpret\_cast<uint8\_t\*>(&value), reinterpret\_cast<uint8\_t\*>(&value) + sizeof(value));

354: value = ntohll(\*reinterpret\_cast<const unsigned long long\*>(&other.dy));

355: buffer.insert(buffer.end(), reinterpret\_cast<uint8\_t\*>(&value), reinterpret\_cast<uint8\_t\*>(&value) + sizeof(value));

356:

357: value = ntohll(\*reinterpret\_cast<const unsigned long long\*>(&other.ddx));

358: buffer.insert(buffer.end(), reinterpret\_cast<uint8\_t\*>(&value), reinterpret\_cast<uint8\_t\*>(&value) + sizeof(value));

359: value = ntohll(\*reinterpret\_cast<const unsigned long long\*>(&other.ddy));

360: buffer.insert(buffer.end(), reinterpret\_cast<uint8\_t\*>(&value), reinterpret\_cast<uint8\_t\*>(&value) + sizeof(value));

361:

362: value = ntohll(\*reinterpret\_cast<const unsigned long long\*>(&other.time));

363: buffer.insert(buffer.end(), reinterpret\_cast<uint8\_t\*>(&value), reinterpret\_cast<uint8\_t\*>(&value) + sizeof(value));

364: }

365: }

366: // TODO: add check on size of array

367: }

368:

369: void Network::send\_answer (const std::string& ip, const std::string& port, const Answer& answer) {

370: // prepare data

371: std::vector<uint8\_t> buffer;

372:

373:

374: serialize\_answer(answer, buffer);

375: if (not encode\_message(buffer)) {

376: response\_bad\_formed(ip, port);

377: return;

378: }

379:

380: // get user info

381: addrinfo hints;

382: memset(&hints, 0, sizeof(hints));

383: hints.ai\_family = AF\_INET; // AF\_INET or AF\_INET6 (IPv4 or IPv6)

384: hints.ai\_socktype = SOCK\_DGRAM; // UDP

385: hints.ai\_protocol = 0; // use default protocol implementation for UDP

386: addrinfo\* result;

387: int error = getaddrinfo(ip.c\_str(), port.c\_str(), &hints, &result);

388: if (error != 0) {

389: std::osyncstream(std::cerr) << "User address resolution failed: " << gai\_strerror(error) << '\n';

390: return;

391: }

392:

393: // FIXME case of long buffer - int overflow

394: error = sendto(m\_server\_socket, reinterpret\_cast<char\*>(buffer.data()), static\_cast<int>(buffer.size()), 0, result->ai\_addr, static\_cast<int>(result->ai\_addrlen));

395: freeaddrinfo(result);

396: if (error == SOCKET\_ERROR) {

397: std::osyncstream(std::cerr) << "Failed to send data to user: " << WSAGetLastError() << '\n';

398: return;

399: }

400: }

401:

# ----------------------------------------------------------------------

File: Network.cpp (Client)

001: #include "Network.h"

002: #include <cstdint>

003: #include <optional>

004: #include <syncstream>

005: #include <iostream>

006: #include <error\_repairing.h>

007:

008:

009: Network::Network() : m\_server\_address{nullptr} {

010: WSADATA wsaData;

011: if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0) {

012: std::osyncstream(std::cerr) << "Socket startup failed: " << WSAGetLastError() << '\n';

013: exit(1);

014: }

015: std::osyncstream(std::cout) << "Socket startup success" << '\n';

016: find\_server();

017: }

018:

019: Network::~Network () {

020: if (WSACleanup() != 0) {

021: std::osyncstream(std::cerr) << "Socket shutdown failed: " << WSAGetLastError() << '\n';

022: exit(1);

023: }

024: std::osyncstream(std::cout) << "Socket shutdown success" << '\n';

025:

026: if (m\_server\_address != nullptr) freeaddrinfo(m\_server\_address);

027: }

028:

029: bool Network::is\_connected () const { return m\_is\_connected\_to\_server; }

030:

031: bool Network::setup\_socket () {

032: if (m\_is\_socket\_setuped) return true;

033:

034: // get server info

035: addrinfo hints;

036: memset(&hints, 0, sizeof(hints));

037: hints.ai\_family = AF\_INET; // AF\_INET or AF\_INET6 (IPv4 or IPv6)

038: hints.ai\_socktype = SOCK\_DGRAM; // UDP

039: hints.ai\_protocol = 0; // use default protocol implementation for UDP

040: hints.ai\_flags = AI\_PASSIVE; // used resolution for socket creation and binding

041: addrinfo\* result;

042: int error = getaddrinfo("localhost", nullptr, &hints, &result);

043: if (error != 0) {

044: std::osyncstream(std::cerr) << "Self address resolution failed: " << gai\_strerror(error) << '\n';

045: return false;

046: }

047:

048: // UDP socket

049: m\_client\_socket = socket(result->ai\_family, result->ai\_socktype, result->ai\_protocol);

050: if (m\_client\_socket == INVALID\_SOCKET) {

051: std::osyncstream(std::cerr) << "Socket creation failed: " << WSAGetLastError() << '\n';

052: freeaddrinfo(result);

053: closesocket(m\_client\_socket);

054: return false;

055: }

056:

057: // binding

058: error = bind(m\_client\_socket, result->ai\_addr, int(result->ai\_addrlen));

059: freeaddrinfo(result); // freeing memory - unneaded after binding

060: if (error == SOCKET\_ERROR) {

061: std::osyncstream(std::cerr) << "Socket binding failed: " << WSAGetLastError() << '\n';

062: closesocket(m\_client\_socket);

063: return false;

064: }

065:

066: m\_is\_socket\_setuped = true;

067: return true;

068: }

069:

070: void Network::find\_server () {

071: // get server info

072: addrinfo hints;

073: memset(&hints, 0, sizeof(hints));

074: hints.ai\_family = AF\_INET; // AF\_INET or AF\_INET6 (IPv4 or IPv6)

075: hints.ai\_socktype = SOCK\_DGRAM; // UDP

076: hints.ai\_protocol = 0; // use default protocol implementation for UDP

077: hints.ai\_flags = AI\_PASSIVE; // used resolution for socket creation and binding

078: int error = getaddrinfo("127.0.0.1", "20123", &hints, &m\_server\_address);

079: if (error != 0) {

080: std::osyncstream(std::cerr) << "Self address resolution failed: " << gai\_strerror(error) << '\n';

081: exit(1);

082: }

083: }

084:

085: void Network::process\_error () {

086: int wsa\_error = WSAGetLastError();

087:

088: // this seems to be get when client is disconnects - in connection less protocol ;(

089: if (wsa\_error == 10054) return; // FIXME try to find a const

090:

091: std::osyncstream(std::cerr) << "Socket receive failed: " << wsa\_error << '\n' << '\n';

092: }

093:

094:

095: // ========================================================

096: // receive data

097: // ========================================================

098: bool Network::decode\_message (std::vector<uint8\_t>& message) {

099: return custom\_utils::decode\_package(message);

100: }

101:

102: std::optional<Answer> Network::parse\_answer (const std::vector<uint8\_t>& message) {

103: // Ensure that the message size is exactly what we expect

104: if (message.size() == 0) {

105: return std::nullopt;

106: }

107:

108: // Ensure the message size is valid

109: if (message.size() < sizeof(uint8\_t)) {

110: return Answer{.type=Answer::Type::BAD\_FORMED};

111: }

112:

113: Answer answer;

114: uint64\_t llong\_value;

115: size\_t data\_disposition = 0;

116: const uint8\_t\* data = message.data();

117:

118: // 1. Type (1 byte)

119: answer.type = static\_cast<Answer::Type>(data[0]);

120: if (int(answer.type) < int(Answer::Type::BAD\_FORMED) || int(answer.type) > int(Answer::Type::BREAK\_SESSION)) {

121: return Answer{.type=Answer::Type::BAD\_FORMED};

122: }

123: data\_disposition += sizeof(uint8\_t);

124:

125: // Handle specific answer types

126: switch (answer.type) {

127: case Answer::Type::FINISH: {

128: // 2. Finish (x: 8 bytes, y: 8 bytes, time: 8 bytes, has\_finished: 1 byte)

129: if (message.size() != data\_disposition + sizeof(uint64\_t) \* 3 + sizeof(uint8\_t)) {

130: return Answer{.type=Answer::Type::BAD\_FORMED};

131: }

132:

133: Answer::Finish finish;

134:

135: memcpy(&llong\_value, data + data\_disposition, sizeof(llong\_value));

136: llong\_value = ntohll(llong\_value);

137: finish.x = static\_cast<uint64\_t>(llong\_value);

138: data\_disposition += sizeof(uint64\_t);

139:

140: memcpy(&llong\_value, data + data\_disposition, sizeof(llong\_value));

141: finish.y = ntohll(llong\_value);

142: data\_disposition += sizeof(uint64\_t);

143:

144: memcpy(&llong\_value, data + data\_disposition, sizeof(llong\_value));

145: finish.time = ntohll(llong\_value);

146: data\_disposition += sizeof(uint64\_t);

147:

148: finish.has\_finished = static\_cast<bool>(data[data\_disposition]);

149: data\_disposition += sizeof(uint8\_t);

150:

151: answer.finish = finish;

152: break;

153: }

154: case Answer::Type::OTHER: {

155: // 3. Other (id: 8 bytes, cound: 8 bytes, x/y/dx/dy/ddx/ddy: 6 \* 8 bytes each)

156: if (message.size() < data\_disposition + sizeof(uint64\_t)) { // has id

157: return Answer{.type=Answer::Type::BAD\_FORMED};

158: }

159:

160: memcpy(&llong\_value, data + data\_disposition, sizeof(llong\_value));

161: answer.id = ntohll(llong\_value);

162: data\_disposition += sizeof(uint64\_t);

163:

164: // ---

165:

166: if (message.size() < data\_disposition + sizeof(uint64\_t)) { // has count

167: return Answer{.type=Answer::Type::BAD\_FORMED};

168: }

169:

170: uint64\_t count\_net;

171: memcpy(&llong\_value, data + data\_disposition, sizeof(llong\_value));

172: count\_net = ntohll(llong\_value);

173: size\_t count = \*reinterpret\_cast<size\_t\*>(&count\_net);

174: data\_disposition += sizeof(uint64\_t);

175:

176: // sizeof(Answer::Other) = sizeof(uint64\_t) + 6 \* sizeof(double)

177: if (message.size() != data\_disposition + count \* sizeof(Answer::Other)) { // enough data for count

178: return Answer{.type=Answer::Type::BAD\_FORMED};

179: }

180:

181: std::vector<Answer::Other> others;

182: others.reserve(count);

183:

184: for (uint64\_t i = 0; i < count; ++i) {

185: Answer::Other other;

186:

187: memcpy(&llong\_value, data + data\_disposition, sizeof(uint64\_t));

188: other.id = ntohll(llong\_value);

189: data\_disposition += sizeof(uint64\_t);

190:

191: memcpy(&llong\_value, data + data\_disposition, sizeof(uint64\_t));

192: other.payload.x = network\_to\_host\_double(llong\_value);

193: data\_disposition += sizeof(uint64\_t);

194:

195: memcpy(&llong\_value, data + data\_disposition, sizeof(uint64\_t));

196: other.payload.y = network\_to\_host\_double(llong\_value);

197: data\_disposition += sizeof(uint64\_t);

198:

199: memcpy(&llong\_value, data + data\_disposition, sizeof(uint64\_t));

200: other.payload.dx = network\_to\_host\_double(llong\_value);

201: data\_disposition += sizeof(uint64\_t);

202:

203: memcpy(&llong\_value, data + data\_disposition, sizeof(uint64\_t));

204: other.payload.dy = network\_to\_host\_double(llong\_value);

205: data\_disposition += sizeof(uint64\_t);

206:

207: memcpy(&llong\_value, data + data\_disposition, sizeof(uint64\_t));

208: other.payload.ddx = network\_to\_host\_double(llong\_value);

209: data\_disposition += sizeof(uint64\_t);

210:

211: memcpy(&llong\_value, data + data\_disposition, sizeof(uint64\_t));

212: other.payload.ddy = network\_to\_host\_double(llong\_value);

213: data\_disposition += sizeof(uint64\_t);

214:

215: memcpy(&llong\_value, data + data\_disposition, sizeof(uint64\_t));

216: other.payload.time = ntohll(llong\_value);

217: data\_disposition += sizeof(uint64\_t);

218:

219: others.push\_back(other);

220: }

221:

222: answer.other = others;

223: break;

224: }

225: case Answer::Type::ACKNOWLEDGE: {

226: // 4. ID (8 bytes)

227: if (message.size() != data\_disposition + sizeof(uint64\_t)) {

228: return Answer{.type=Answer::Type::BAD\_FORMED};

229: }

230: memcpy(&llong\_value, data + data\_disposition, sizeof(llong\_value));

231: answer.id = ntohll(llong\_value);

232:

233: break;

234: }

235: case Answer::Type::REGISTERED\_ANSWER: {

236: if (message.size() != data\_disposition) {

237: return Answer{.type=Answer::Type::BAD\_FORMED};

238: }

239: break;

240: }

241: default: { break; }

242: }

243:

244: return answer;

245: }

246:

247: // ========================================================

248: // send data

249: // ========================================================

250:

251:

252: bool Network::encode\_message (std::vector<uint8\_t>& message) {

253: return custom\_utils::encode\_package(message);

254: }

255:

256: void Network::serialize\_package (const Package& package, std::vector<uint8\_t>& buffer) {

257: uint8\_t type = \*reinterpret\_cast<const uint8\_t\*>(&package.type);

258: buffer.insert(buffer.end(), reinterpret\_cast<uint8\_t\*>(&type), reinterpret\_cast<uint8\_t\*>(&type) + sizeof(type));

259:

260: // due to how type works - in current implementation, only one payload can be in the answer

261: if (package.id.has\_value()) {

262: uint64\_t id = ntohll(package.id.value());

263: buffer.insert(buffer.end(), reinterpret\_cast<uint8\_t\*>(&id), reinterpret\_cast<uint8\_t\*>(&id) + sizeof(id));

264: }

265:

266: if (package.payload.has\_value()) {

267: uint64\_t d\_value;

268: d\_value = ntohll(\*reinterpret\_cast<const unsigned long long\*>(&package.payload.value().x));

269: buffer.insert(buffer.end(), reinterpret\_cast<uint8\_t\*>(&d\_value), reinterpret\_cast<uint8\_t\*>(&d\_value) + sizeof(d\_value));

270: d\_value = ntohll(\*reinterpret\_cast<const unsigned long long\*>(&package.payload.value().y));

271: buffer.insert(buffer.end(), reinterpret\_cast<uint8\_t\*>(&d\_value), reinterpret\_cast<uint8\_t\*>(&d\_value) + sizeof(d\_value));

272:

273: d\_value = ntohll(\*reinterpret\_cast<const unsigned long long\*>(&package.payload.value().dx));

274: buffer.insert(buffer.end(), reinterpret\_cast<uint8\_t\*>(&d\_value), reinterpret\_cast<uint8\_t\*>(&d\_value) + sizeof(d\_value));

275: d\_value = ntohll(\*reinterpret\_cast<const unsigned long long\*>(&package.payload.value().dy));

276: buffer.insert(buffer.end(), reinterpret\_cast<uint8\_t\*>(&d\_value), reinterpret\_cast<uint8\_t\*>(&d\_value) + sizeof(d\_value));

277:

278: d\_value = ntohll(\*reinterpret\_cast<const unsigned long long\*>(&package.payload.value().ddx));

279: buffer.insert(buffer.end(), reinterpret\_cast<uint8\_t\*>(&d\_value), reinterpret\_cast<uint8\_t\*>(&d\_value) + sizeof(d\_value));

280: d\_value = ntohll(\*reinterpret\_cast<const unsigned long long\*>(&package.payload.value().ddy));

281: buffer.insert(buffer.end(), reinterpret\_cast<uint8\_t\*>(&d\_value), reinterpret\_cast<uint8\_t\*>(&d\_value) + sizeof(d\_value));

282:

283: uint64\_t time = package.payload.value().time;

284: buffer.insert(buffer.end(), reinterpret\_cast<uint8\_t\*>(&time), reinterpret\_cast<uint8\_t\*>(&time) + sizeof(time));

285: }

286: }

287:

288: bool Network::send\_package (const Package& package) {

289: // setup

290: if (not setup\_socket()) {

291: std::osyncstream(std::cerr) << "Socket wasn't setuped before getting answer (internal error - send)" << '\n';

292: exit(1);

293: }

294:

295: // prepare data

296: std::vector<uint8\_t> buffer;

297: serialize\_package(package, buffer);

298: if (not encode\_message(buffer)) {

299: std::osyncstream(std::cerr) << "Failed to encode message - inner error\n";

300: exit(1);

301: }

302:

303: // FIXME case of long buffer - int overflow

304: int error = sendto(m\_client\_socket, reinterpret\_cast<char\*>(buffer.data()), static\_cast<int>(buffer.size()), 0, m\_server\_address->ai\_addr, static\_cast<int>(m\_server\_address->ai\_addrlen));

305: if (error == SOCKET\_ERROR) {

306: std::osyncstream(std::cerr) << "Failed to send data to server: " << WSAGetLastError() << '\n';

307: return false;

308: }

309:

310: return true;

311: }

312:

313: // ========================================================

314: // data receivers

315: // ========================================================

316:

317: std::optional<Answer> Network::get\_answer () {

318: // setup

319: if (not setup\_socket()) {

320: std::osyncstream(std::cerr) << "Socket wasn't setuped before getting answer (internal error - answer)" << '\n';

321: exit(1);

322: }

323:

324: // get data cycle

325: int server\_address\_size = sizeof(\*m\_server\_address->ai\_addr);

326: static std::array<char, 1024> buffer;

327: constexpr int buffer\_size = static\_cast<int>(buffer.size());

328:

329: int data\_size = recvfrom(m\_client\_socket, buffer.data(), buffer\_size,

330: 0, m\_server\_address->ai\_addr, &server\_address\_size);

331:

332: if (data\_size == SOCKET\_ERROR) {

333: process\_error();

334: return std::nullopt;

335: }

336:

337: // process data

338: std::vector<uint8\_t> data(buffer.data(), buffer.data() + static\_cast<size\_t>(data\_size));

339: decode\_message(data);

340: std::optional<Answer> answer = parse\_answer(data);

341:

342: if (not answer.has\_value()) return answer; // error - return

343:

344: // check if connected (registered) or disconnected (break session)

345: if (answer->type == Answer::Type::REGISTERED\_ANSWER) {

346: m\_is\_connected\_to\_server = true;

347: } else if (answer->type == Answer::Type::BREAK\_SESSION) {

348: m\_is\_connected\_to\_server = false;

349: }

350:

351: return answer;

352: }

353:

354: // ========================================================

355: // data senders

356: // ========================================================

357:

358: std::optional<bool> Network::send\_connection () {

359: if (m\_is\_connected\_to\_server) return std::nullopt;

360:

361: // make data for connection

362: Package data {.type=Package::Type::LOGIN};

363:

364: // send

365: return send\_package(data);

366: }

367:

368: std::optional<bool> Network::send\_disconnection () {

369: if (not m\_is\_connected\_to\_server) return std::nullopt;

370:

371: // make data for disconnection

372: Package data {.type=Package::Type::BREAK\_SESSION};

373:

374: // send

375: return send\_package(data);

376:

377: }

378:

379: std::optional<id\_game\_t> Network::send\_ask\_other () {

380: if (not m\_is\_connected\_to\_server) return std::nullopt;

381:

382: // make data for ask other

383: Package data {.type=Package::Type::GET\_OTHER, .id=m\_next\_package\_id};

384: ++m\_next\_package\_id;

385:

386: // send

387: if (not send\_package(data)) return std::nullopt;

388:

389: return m\_next\_package\_id - 1;

390: }

391: std::optional<id\_game\_t> Network::send\_message (const Package::Package\_Payload& payload) {

392: if (not m\_is\_connected\_to\_server) return std::nullopt;

393:

394: // make data for message

395: Package data {.type=Package::Type::MESSAGE, .id=m\_next\_package\_id, .payload=payload};

396: ++m\_next\_package\_id;

397:

398: // send

399: if (not send\_package(data)) return std::nullopt;

400:

401: return m\_next\_package\_id - 1;

402: }

403: std::optional<id\_game\_t> Network::send\_finish (const Package::Package\_Payload& payload) {

404: if (not m\_is\_connected\_to\_server) return std::nullopt;

405:

406: // make data for finish

407: Package data {.type=Package::Type::FINISH, .id=m\_next\_package\_id, .payload=payload};

408: ++m\_next\_package\_id;

409:

410: // send

411: if (not send\_package(data)) return std::nullopt;

412:

413: return m\_next\_package\_id - 1;

414: }

415:

416:

417: // ========================================================

418: // helpers

419: // ========================================================

420:

421: std::optional<std::map<id\_game\_t, Answer::Other\_Payload>> Network::parse\_type\_other\_answer (const Answer& answer) {

422: if (answer.type != Answer::Type::OTHER) return std::nullopt;

423: if (not answer.other.has\_value()) return std::nullopt;

424:

425: std::map<id\_game\_t, Answer::Other\_Payload> result;

426: for (const auto& other : answer.other.value()) {

427: result.insert({other.id, other.payload});

428: }

429:

430: return result;

431: }

432: