Приложение Б Листинг исходного кода

src\networking\udp\_receiver.hpp

#pragma once

#include <boost/asio.hpp>

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

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

#include <config/type/udp\_receiver.hpp>

#include <functional>

#include <models/udp\_buffer.hpp>

#include <optional>

#include <vector>

namespace networking::udp\_receiver {

namespace ip = boost::asio::ip;

class UdpReceiver {

public:

using HandlerType =

std::function<void(models::udp\_buffer::DataBuffer&)>;

UdpReceiver(const config::udp\_receiver::Configuration& config,

const HandlerType handler)

: started\_(false),

thread\_count\_(config.receive\_threads),

ip\_version\_(config.ip\_version),

buffer\_queue\_(config.buffer\_count, config.datagram\_max\_size),

socket\_(io\_service\_),

receiver\_endpoint\_(boost::asio::ip::address\_v4::any(),

config.target\_port),

expected\_source\_ip\_(config.target\_ip),

handler\_(handler){};

void Start();

class ReceiveFailed : std::runtime\_error {

using std::runtime\_error::runtime\_error;

};

class BufferCorrupted : std::runtime\_error {

using std::runtime\_error::runtime\_error;

};

class BufferSizeLessThanRequired : std::runtime\_error {

using std::runtime\_error::runtime\_error;

};

~UdpReceiver() {

io\_service\_.stop();

socket\_.close();

for (auto& t : receive\_threads\_) {

//End threads and supress exceptions if present;

try {

t.join();

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

}

}

private:

void WaitReceive();

void Handle(models::udp\_buffer::DataBuffer& data\_buffer,

ip::udp::endpoint& udp\_source,

const boost::system::error\_code& error, size\_t bytes\_transferred);

bool started\_;

int thread\_count\_;

ip::udp ip\_version\_;

models::udp\_buffer::BufferQueue buffer\_queue\_;

boost::asio::io\_service io\_service\_;

ip::udp::socket socket\_{io\_service\_};

boost::asio::io\_service::work work\_{io\_service\_};

ip::udp::endpoint receiver\_endpoint\_;

std::optional<ip::address> expected\_source\_ip\_;

std::vector<std::thread> receive\_threads\_;

HandlerType handler\_;

};

}// namespace networking::udp\_receiver

udp\_receiver.cpp

#include "udp\_receiver.hpp"

#include <fmt/core.h>

#include <boost/bind/bind.hpp>

#include <unordered\_set>

#include <utils/logging.hpp>

namespace networking::udp\_receiver {

namespace {

struct RAIIBufferContainer {

RAIIBufferContainer(

std::unique\_ptr<models::udp\_buffer::DataBuffer> buffer\_to\_return,

models::udp\_buffer::BufferQueue& queue\_to\_return)

: container(std::move(buffer\_to\_return)), queue(queue\_to\_return) {}

~RAIIBufferContainer() { queue.ReleaseBuffer(std::move(container)); }

std::unique\_ptr<models::udp\_buffer::DataBuffer> container;

models::udp\_buffer::BufferQueue& queue;

};

}// namespace

void UdpReceiver::Start() {

if (started\_) {

LOG\_INFO() << "UdpReceiver already started\n";

return;

}

started\_ = true;

socket\_.open(ip\_version\_);

socket\_.bind(receiver\_endpoint\_);

for (int i = 0; i < thread\_count\_; i++) {

receive\_threads\_.push\_back(std::thread([this] {

WaitReceive();

io\_service\_.run();

}));

}

}

void UdpReceiver::WaitReceive() {

auto buffer = buffer\_queue\_.AquireBuffer();

auto endpoint = std::make\_unique<ip::udp::endpoint>();

socket\_.async\_receive\_from(

boost::asio::buffer(buffer->buffer, buffer->buffer.size()), \*endpoint,

[data\_buffer = std::move(buffer), udp\_source = std::move(endpoint), this](

const boost::system::error\_code& error,

size\_t bytes\_transferred) mutable {

if (!data\_buffer) {

throw UdpReceiver::BufferCorrupted("Data buffer ptr is NULL");

}

if (!udp\_source) {

throw UdpReceiver::BufferCorrupted("UdpSource lost");

}

RAIIBufferContainer secured\_buffer(std::move(data\_buffer),

buffer\_queue\_);

Handle(\*secured\_buffer.container, \*udp\_source, error,

bytes\_transferred);

});

}

void UdpReceiver::Handle(models::udp\_buffer::DataBuffer& data\_buffer,

ip::udp::endpoint& udp\_source,

const boost::system::error\_code& error,

size\_t bytes\_transferred) {

static const std::unordered\_set<int> ignored\_errors{

boost::asio::error::operation\_aborted};

if (error) {

if (!ignored\_errors.contains(error.value())) {

LOG\_WARNING() << fmt::format("Receive failed: {} | Val: {} | Cat: {}\n",

error.message(), error.value(),

error.category().name());

}

return;

}

if (expected\_source\_ip\_) {

if (expected\_source\_ip\_ != udp\_source.address()) {

LOG\_INFO() << "Receive blocked for ip: " << udp\_source.address() << "\n";

return;

}

}

if (data\_buffer.buffer.size() < bytes\_transferred) {

throw UdpReceiver::BufferSizeLessThanRequired("Data buffer ptr is NULL");

}

data\_buffer.last\_datagram\_size = bytes\_transferred;

try {

handler\_(data\_buffer);

} catch (std::runtime\_error& e) {

LOG\_INFO() << "Caught untyped exception: " << e.what() << "\n";

}

}

}// namespace networking::udp\_receiver