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
```