ПРАВИТЕЛЬСТВО РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное
учреждение высшего образования

Национальный исследовательский университет
«Высшая школа экономики»

Московский институт электроники и математики
Факультет прикладной математики и кибернетики

Кафедра «Компьютерная безопасность»

**Отчет по работе**
по дисциплине
«Современные технологии программирования и обработки информации»

Темы работы: Программирование многопоточного SOCKS5 proxy сервера
с помощью библиотеки libuv на языке C++

**Выполнил(а):** студент группы СКБ162
Католиков А.С.
**Проверил:** ст. преп. Лебедев П.А.

**МОСКВА – 2019**

# Структура

# Введение

В этом задание я познакомился с написанием многопоточного Socks5 proxy сервера на языке C++ используя библиотеку libuv. Тестирование производилось через стандартный браузер имеющийся у каждого пользователя (в моем случае Mozila Firefox). На сегодняшний день данная технология очень востребована, так как почти что каждый день мы слышим о блокировке тех или иных интернет ресурсов, без которых мы не представляем нашу сегодняшний день. Прокси сервер предоставляет возможность обходить это с помощью простых механизмов. Зачем же нам нужна многопоточность? Ответ очень прост. Многопоточность приносит пользу при наличии нескольких задач, которые могут (хотя бы частично) работать одновременно. Данный сервер писался с помощью текстового редактора Atom на операционной системе  OSX.

# Постановка задачи

Реализовать многопоточный Socks5 прокси сервер с использованием кросс-платформенной библиотеки libuv на языке программирования C++ для взаимодействия с клиентами. Требуется единовременно обрабатывает большое количество TCP соединений с помощью сетевого протокола SOCKS версии 5 без аутентификация и предоставлять клиентам сервис без существенных задержек по времени. Данное исполнение программы не является оптимальным в силу особенности библиотеки libuv.

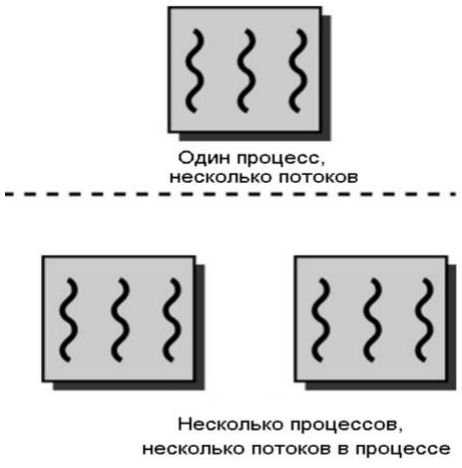
# Техническая информация

# Поток

Поток – ресурс ОС, для которого выделяется процессорное время на выполнение. Каждый поток создаётся для выполнения в рамках конкретного процесса и имеет равноправный доступ к ресурсам, связанным с процессом в целом, в первую очередь – его адресному пространству. Для отслеживания текущего места выполнения кода с потоком связывается состояние регистрового файла процессора с его точки зрения и отдельный аппаратный стек для хранения кадров, соответствующих вызываемым функциям. Первый поток программы создаётся самой ОС и начинает выполнения с точки, указанной в образе программы. Остальные потоки создаются по запросам к ОС от самого процесса. Потоки могут создаваться и завершаться в любом порядке, кроме первичного (выход из main()).

# Многопоточность

Способность центрального процессора или одного ядра в многоядерном процессоре одновременно выполнять несколько процессов или потоков, соответствующим образом поддерживаемых операционной системой. Этот подход отличается от многопроцессорности, так как многопоточность процессов и потоков совместно использует ресурсы одного или нескольких ядер.



# Прокси - Сервер

Это работающий на удаленном компьютере (сервере) сервер, позволяющий клиентам выполнять косвенные запросы к другим сетевым службам. Сначала запрос клиента будет передаваться на прокси-сервер, затем сам прокси-сервер подключится к указанному серверу и получит ресурс у него, или же вернет ресурс из собственного куша. Таким образом, в более простом смысле прокси-сервер представляет собой посредника между пользователем и ресурсами сети.

# SOCKS 5

**SOCKS** — сетевой протокол, который позволяет пересылать пакеты от клиента к серверу через прокси-сервер прозрачно (незаметно для них) и таким образом использовать сервисы за межсетевыми экранами.

Клиенты за межсетевым экраном, нуждающиеся в доступе к внешним серверам, вместо этого могут быть соединены с **SOCKS**-прокси-сервером. Такой прокси-сервер контролирует права клиента на доступ к внешним ресурсам и передаёт клиентский запрос внешнему серверу. **SOCKS** может использоваться и противоположным способом, осуществляя контроль прав внешних клиентов соединяться с внутренними серверами, находящимися за межсетевым экраном (брандмауэром).

*SOCKS 5* - предназначен для работы через межсетевой экран для приложений типа клиент-сервер, работающих по протоколу **TCP/UDP**, с обеспечением универсальных схем строгой аутентификации и методов адресации, с поддержкой **Ipv4**, доменных имен и адресов **Ipv6**.

*Начальная установка связи состоит из следующего:*

- Клиент подключается, и посылает приветствие, которое включает перечень поддерживаемых методов аутентификации
- Сервер выбирает из них один (или посылает ответ о неудаче запроса, если ни один из предложенных методов неприемлем)
- В зависимости от выбранного метода, между клиентом и сервером может пройти некоторое количество сообщений
- Клиент посылает запрос на соединение.
- Сервер отвечает.

**Методы аутентификации пронумерованы следующим образом:**

| | |
|---|---|
| 0x00 | Аутентификация не требуется |
| 0x01 | GSSAPI |
| 0x02 | Имя пользователя / пароль |
| 0x03-0x7F | Зарезервировано IANA |
| 0x80-0xFE | Зарезервировано для методов частного использования |

*Начальное приветствие от клиента:*

| Размер | Описание |
|---|---|
| 1 байт | Номер версии SOCKS (должен быть 0x05 для этой версии) |
| 1 байт | Количество поддерживаемых методов аутентификации |
| n байт | Номера методов аутентификации, переменная длина, 1 байт для каждого поддерживаемого метода |

*Сервер сообщает о своём выборе:*

| Размер | Описание |
|---|---|
| 1 байт | Номер версии SOCKS (должен быть 0x05 для этой версии) |
| 1 байт | Выбранный метод аутентификации или 0xFF, если не было предложено приемлемого метода |

**Последующая идентификация зависит от выбранного метода.**
*Запрос клиента:*

| Размер | Описание |
|---|---|
| 1 байт | Номер версии SOCKS (должен быть 0x05 для этой версии) |
| 1 байт | Код команды:<br>• 0x01 = установка TCP/IP соединения<br>• 0x02 = назначение TCP/IP порта (binding)<br>• 0x03 = ассоциирование UDP-порта |
| 1 байт | Зарезервированный байт, должен быть 0x00 |
| 1 байт | Тип адреса:<br>• 0x01 = адрес Ipv4<br>• 0x03 = имя домена<br>• 0x04 = адрес Ipv6 |
| Зависит от типа адреса | Назначение адреса:<br>• 4 байта для адреса Ipv4<br>• 1 байт - длинна, последующие domain name<br>• 16 байт для адреса Ipv6 |
| 2 байта | Номер порта, в порядке от старшего к младшему (big-endian) |

*Ответ сервера:*

| Размер | Описание |
|---|---|
| 1 байт | Номер версии SOCKS (0x05 для этой версии) |
| 1 байт | • 0x00 = запрос предоставлен<br>• 0x01 = ошибка SOCKS-сервера<br>• 0x02 = соединение запрещено набором правил<br>• 0x03 = сеть недоступна<br>• 0x04 = хост недоступен<br>• 0x05 = отказ в соединении<br>• 0x06 = истечение TTL<br>• 0x07 = команда не поддерживается / ошибка протокола<br>• 0x08 = тип адреса не поддерживается |
| 1 байт | 0x00 |
| 1 байт | • 0x01 = адрес Ipv4<br>• 0x03 = имя домена<br>• 0x04 = адрес Ipv6 |
| Зависит от типа адреса | • 4 байта для адреса Ipv4<br>• 1 байт - длинна, последующие domain name<br>• 16 байт для адреса Ipv6 |
| 2 байта | Номер порта, в порядке от старшего к младшему (big-endian) |

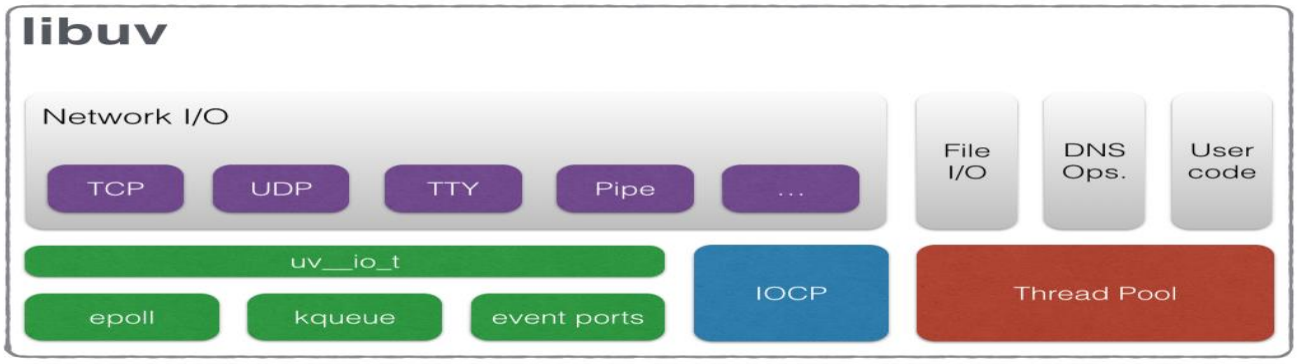
# Libuv

**Libuv** - является кросс-платформенной библиотекой, которая изначально была написана для Node.js . Она разработана на основе асинхронной модели I/O (ввода/вывода) на основе цикла событий. Библиотека, как и **libevent**, использует наиболее эффективный из доступных в системе способов асинхронной работы с сокетами, но в отличии от **libevent**, которая использует буферизированный ввод-вывод для операций с файловой системой, **libuv** для работы с файловой системой применяет синхронные вызовы в пуле потоков.

*Библиотека предоставляет:*

- Полнофункциональный цикл обработки событий, поддерживаемый epoll, kqueue, IOCP, event ports
- Асинхронные сокеты TCP и UDP
- Асинхронное разрешение DNS
- Асинхронные операции с файлами и файловой системой
- События файловой системы
- Код выхода ANSI, контролируемый TTY
- IPC с совместным использованием сокетов, с использованием доменных сокетов Unix или именованных каналов (Windows)
- Дочерние процессы
- Пул потоков
- Обработка сигналов
- Часы с высоким разрешением
- Примитивы потоков и синхронизации

*Диаграмма иллюстрирующая архитектуру библиотеки libuv:*



Libuv предоставляет пользователям **две абстракции для работы** в сочетании с циклом обработки событий: дескрипторы и запросы.

- *Дескрипторы представляют долгоживущие объекты*
- *Запросы представляют (как правило) краткосрочные операции.*

## *Цикл событий[1] (event loop):*

Центральная часть библиотеки. Он устанавливает контент для всех I/O операций, при этом они должны быть в одном потоке. Одно приложение может иметь несколько циклов событий, но каждый должен быть запущен в отдельном потоке.

## *API Libuv:*

- **uv_loop_t** — Event loop ( цикл событий )
- **uv_handle_t** — Base handle ( базовый дескриптер )
- **uv_req_t** — Base request ( базовый запрос )
- **uv_timer_t** — Timer handle ( дескриптер таймера )
- **uv_prepare_t** — Prepare handle ( дескриптер готовности )
- **uv_check_t** — Check handle ( проверочный дескриптер )
- **uv_idle_t** — Idle handle ( бездействующий дескриптер )
- **uv_async_t** — Async handle ( асинхронный дескриптер )
- **uv_poll_t** — Poll handle ( дескриптер опроса )
- **uv_signal_t** — Signal handle ( дескриптер сигналов )
- **uv_process_t** — Process handle ( дескриптер процессов )
- **uv_stream_t** — Stream handle ( дескриптер потоков )
- **uv_tcp_t** — TCP handle ( TCP дескриптер )
- **uv_pipe_t** — Pipe handle ( дескриптер PIPE )
- **uv_tty_t** — TTY handle ( дескриптер TTY )
- **uv_udp_t** — UDP handle ( дескриптер UDP )
- **uv_fs_event_t** — FS Event handle ( дескриптер событий FS )
- **uv_fs_poll_t** — FS Poll handle ( дескриптер опроса FS )

[1] *Цикл событий библиотеки libuv не потокобезопасный (thread-safe).*

# Трудности реализации

# Трудность выбора циклов событий

***Libuv*** *предоставляет пользователю несколько циклов событий:*

- **UV_RUN_DEFAULT:** Запускает цикл обработки событий, пока счетчик ссылок не упадет до нуля. Всегда возвращает ноль.
- **UV_RUN_ONCE:** Опрос для новых событий один раз. Эта функция блокируется, если нет ожидающих событий. Возвращает ноль, когда выполнено (нет активных дескрипторов или запросов не осталось), или ненулевое, если ожидается больше событий (это означает, что вы должны снова запустить цикл обработки событий в будущем).
- **UV_RUN_NOWAIT:** Цикл опрашивает новые события один раз, но не блокирует их, если нет ожидающих событий.

По каким критериям я сделал свой выбор,

**UV_RUN_ONCE** не подходит для реализации нашего серверного приложения, так как он опрашивает новые события всего один раз.

**UV_RUN_NOWAIT** работает в режиме “не ожидания” новых подключений, он подойдет для реализации клиентского приложения, но никак для серверной программы, которая ожидает подключения новых клиентов.

Остается только третий вариант, **UV_RUN_DEFAULT**, его и используем.

# Трудности обработки сообщений полученных от клиента

Сетевой протокол **Socks5** подразумевает отправку двух сообщений от клиента и двух сообщений от сервера. Вначале клиент отправляет сообщение длинной n-байт. Первый байт - эта версия сетевого протокола, последующий - количество поддерживаемых методов аутентификации, а затем и сами методы. Из-за трудности отображение данных байтов на стандартном потоке вывода

возникала трудность и не понимание вида данного передаваемого сообщения. Экспериментальным путем было установлено и затем обработано данное сообщение.

# Проблема отсоединения клиента от сервера

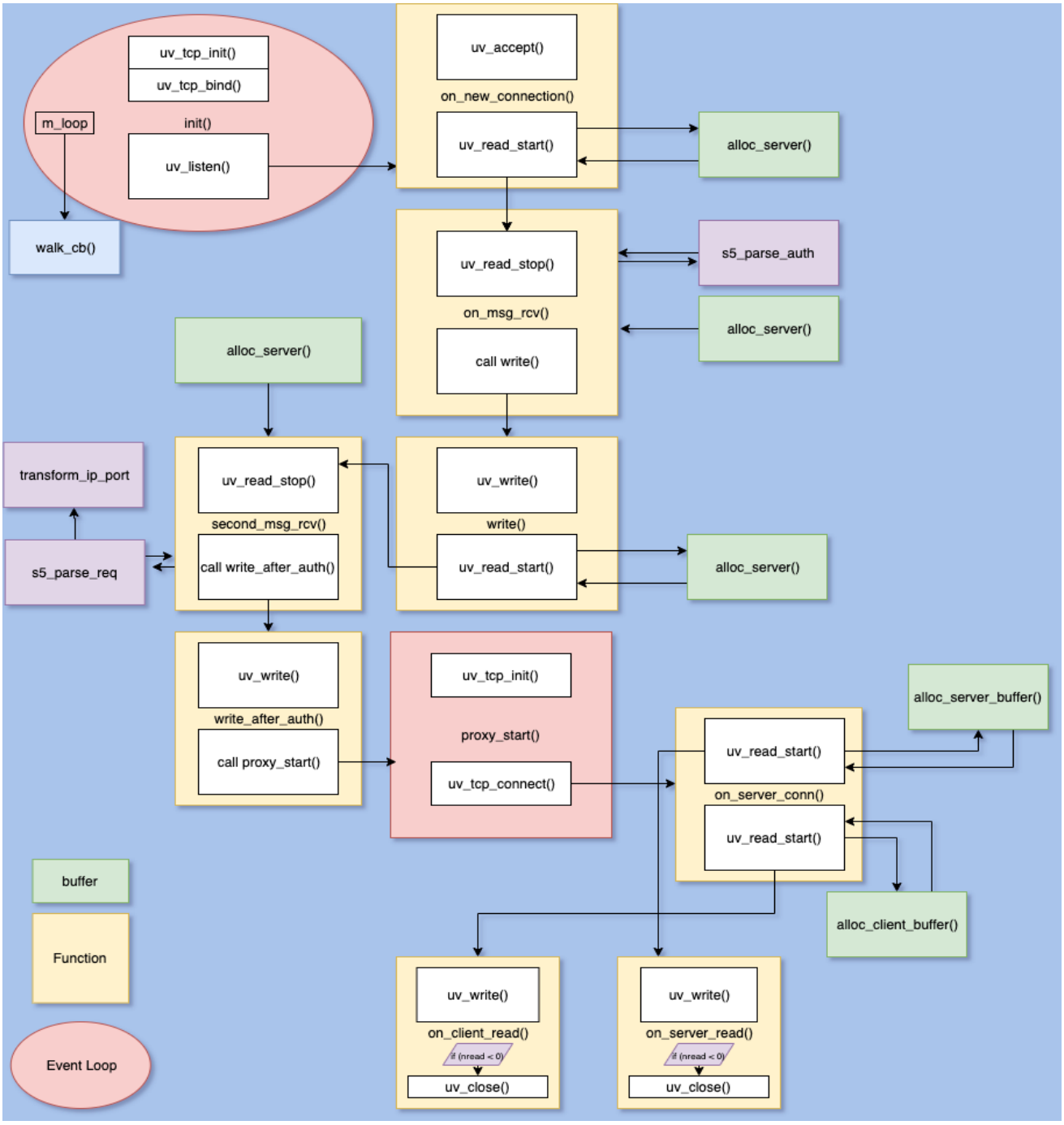
После приветствия и отправки информационных сообщений по правилам сетевого протокола Socks5, после пересылки пары пакетов возникала ошибка ECONNRESET (отсоединение клиента). Природа данной ошибки не до конца понятна. Возникает предположение, что это связана с криптографическими протоколами TLS/SSL, потому что данная ошибка проявляется на сайтах с протоколом **HTTPS** (HyperText Transfer Protocol Secure). Из-за недостаточного количества времени, экспериментальным путем не удалось установить точную причину данного явления.

# Описание реализации

# Архитектура написанного приложения

Libuv кроссплатформенная библиотека, реализующая паттерн Reactor. Паттерн Reactor - это шаблон обработки событий для отработки запросов на обслуживание, передаваемых одновременно обработчику услуг одним или несколькими входами (клиентами). Обработчик сервиса затем демультиплексирует входящие запросы и отправляет их синхронно связанным обработчикам запросов. Говоря более простым языком, Libuv предоставляет интерфейс составления цыпочек вызовов обработчика для работы над сеансом подключения нового клиента, обработки его запросов и завершением освобождая выделенные ресурсы.

Исходя из этого была выбрана следующая архитектура приложения: *(Смотри след.страницу)*


uv_tcp_init()
uv_tcp_bind()
m_loop
init()
uv_listen()
walk_cb()
uv_accept()
on_new_connection()
uv_read_start()
alloc_server()
uv_read_stop()
on_msg_rcv()
call write()
s5_parse_auth
alloc_server()
alloc_server()
transform_ip_port
uv_read_stop()
second_msg_rcv()
call write_after_auth()
s5_parse_req
uv_write()
write()
uv_read_start()
alloc_server()
uv_write()
write_after_auth()
call proxy_start()
uv_tcp_init()
proxy_start()
uv_tcp_connect()
alloc_server_buffer()
uv_read_start()
on_server_conn()
uv_read_start()
alloc_client_buffer()
buffer
Function
Event Loop
uv_write()
on_client_read()
if (nread < 0)
uv_close()
uv_write()
on_server_read()
if (nread < 0)
uv_close()

# Пошаговое описание приложения

Запускаем цикл событий в функции **init()**, создаем гнездо и назначаем адрес командой **uv_bind()**, затем выражаем готовность принимать соединения командой **uv_listen()**. После нового подключения эта команда вызывает соответствующий обработчик **(on_new_connection())**, который отвечает за обработку нового подключения. Мы принимаем подключение командой **uv_accept()** и включаем чтение из сокета клиента командой **uv_read_start().**

*Эта операция является асинхронной поэтому она требует несколько обработчиков:*

- **alloc_server()** - ответственен за выделяемую память
- **on_msg_rcv()** - ответственен за дальнейшую обработку сообщения

Обработчик **on_msg_rcv()** завершает поток на чтение командой **uv_read_stop()** от клиента, приводит к типу полученное сообщение из **buf->base** в **std::string** и передает строку в функцию **s5_parse_auth().** Функция проверяет полученное байтовое сообщение на принадлежность к сетевому протоколу **Socks5** и на наличие поддержки метода соединения с клиентом без аутентификации. Если все условия соблюдены, функция возвращает значение **true** и с помощью функции **write()** отправляется строка с названием поддерживаемого сетевого протокола и выбранного метода аутентификации ( в нашем случае **0x05 0x00** ). Если байтовая строка не удовлетворяет нашим условиям, то выводится сообщение об ошибке. Функция **write()** с помощью **uv_write()** отправляет клиенту сообщение длинной два байта. Затем с помощью **uv_read_start()** мы включаем чтение из сокета клиента.

*Опять же, операция асинхронна, поэтому имеем два обработчика:*

- **alloc_server()** - ответственен за выделяемую память
- **second_msg_recv()** - ответственен за дальнейшую обработку сообщения

После получения информации, обработчик (**second_msg_recv()**) получает сообщение, закрывает поток на чтение, обрабатывает полученное сообщение с помощью функций **s5_parse_req()** и преобразовывает **ip адрес и порт** в строку и соответственно в значение типа int для дальнейшей работы с **uv_ip4_addr()**. Если функция **s5_parse_req()** успешно завершается, то вызывается функция

**write_after_auth()**. Если второй байт отправляемого сообщения имеет значение **0x00**, то в теле функции вызывается **proxy_start()** о которой чуть позднее. Затем отправляется набор байтов о возможности или невозможности соединения с указанным ранее клиентом адресом с помощью **uv_write().** При неуспешном завершение функция **s5_parse_req()** возвращает **false** и клиенту возвращается код ошибки равный значению **0x08**.

Вернемся к **proxy_start().** Данная функция инициализирует **socket** и подключает к адресному пространству. Обработчик подключения открывает потоки на чтение из сокета клиента и из сокета сервера с помощью **uv_read_start().**

*Для каждой операции **uv_read_start()** имеем два обработчика:*

- **alloc_server/client_buffer()** - ответственен за выделяемую память
- **on_server/client_read()** - ответственен за дальнейшую обработку сообщения

Обработчик получения сообщения от клиента/сервера пересылает полученное сообщение **серверу/клиенту**. Если **nread** = EOF (клиента), то есть конец передаваемого сообщения, мы с помощью операции **uv_close()** закрываем соединение.