**МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ**

Федеральное государственное бюджетное образовательное

учереждение высшего образования

«Шадринский государственный педагогический университет»

Факультет информатики математики и физики

Кафедра программирования и автоматизации бизне-процессов

«**Разработка прототипа HTTP клиента для POSIX совместимых OC**»

Курсовая работа по дисциплине «Проектный практикум»

направление подготовки 09.03.01 Информатика и вычислительная техника

образовательная программа « Программное обеспечение вычислительной техники и автоматизированных систем»

Квалификация - бакалавр

**Выполнил**:

Стерхов А.И

Студент 330Б группы

очная форма обучения

**Научный Руководитель**:

Канд. Пед. Наук, доцент

Слинкин Д.А

“\_\_”\_\_\_\_\_\_\_\_\_\_\_\_2024 г.

оценка\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Содержание**

Введение………………………………………………………………………..3

1. Теоретическая часть
   1. HTTP протокол ………………………………………………………...4
   2. HTTP запросы ….………………………………………………………6
   3. HTTP ответы……………………………………………………………8
   4. Структура клиента…………………………………………………….11
   5. Используемые инструменты разработки…………………………….12
2. Практическая часть
   1. Настройка и установка сервера Apache……………………………...16
   2. Реализация HTTP клиента на языке Си…………………………..….18
   3. Создание PHP скрипта для проверки запросов ……………………..24

Заключение…………………………………………………………………25

Список используемых материалов………………………………………. 25

Приложение……………………………………………...…………………26

**Введение**

Протокол HTTP (Hypertext Transfer Protocol) является одним из главных протоколов, используемых при взаимодействии клиента и сервера в сети интернет. Он позволяет клиенту отправлять запросы на сервер и получать от него ответы, c помощью которых происходит передача данных.

**Актуальность** темы курсовой работы связана с тем, что веб-сервисы и API становятся все более нужными и важными в современном программном обеспечении. Взаимодействие с ними по протоколу HTTP является неотъемлемой частью таких систем. Прототип может быть использован как основа для разработки более сложного и функционального HTTP клиента.

**Цель** данной курсовой работы заключается в разработке программного инструмента - прототипа HTTP клиента на языке программирования C для POSIX совместимых операционных систем. Главная задача состоит в том, чтобы создать гибкий и функциональный продукт, который позволит взаимодействовать с веб-сервисами и осуществлять передачу данных по протоколу HTTP.

Для достижения поставленной цели необходимо решить несколько задач.

**Задачи**:

1. Изучить структуру и основные принципы работы HTTP протокола.
2. Разработать базовую структуру HTTP клиента.
3. Реализовать рабочий прототип HTTP клиента с обработкой запросов, ответов и управлением соединений.
4. Тестировка и проверка функционала с помощью PHP скрипта.

В заключении подведены основные итоги написания курсовой работы, а также представлен результат.

**1. Теоретическая часть**

**1.1 HTTP Протокол**

Обмен данными между веб-серверами и их клиентами обеспечиваются протоколом прикладного уровня под названием Hypertext Transfer Protocol (HTTP, протокол передачи гипертекстовых файлов). HTTP используется в World Wide Web (WWW) начиная с 1990 года. Первой версией HTTP, известной как HTTP/0.9, был простой протокол для передачи необработанных данных через интернет. HTTP/1.0 как описано в RFC 1945, был улучшенной версией этого протокола, позволяя сообщениям иметь MIME (Multipurpose Internet Mail Extensions — многоцелевые расширения интернет почты) подобный формат, содержащий информацию о передаваемых данных. Однако, HTTP/1.0 недостаточно хорошо работал с иерархическими прокси-серверами, кэшированием, постоянным соединением и виртуальными хостами. На смену ему вышла спецификации протокола HTTP под номером 1.1 которая была опубликована в качестве документа RFC 2616 в июне 1999 года. На сегодняшний день наиболее распространенной версией протокола является версия HTTP/2.

HTTP представляет собой относительно простой протокол, использующий на транспортном уровне услуги протокола TCP. TCP (Transmission Control Protocol — протокол управления передачей), который используется для доставки файлов от серверов к клиентам. Когда клиент устанавливает соединение с веб-сервером, указывая URL в браузере или щелкая мышью на гиперссылку, система генерирует сообщение - запрос протокола HTTP. Этот процесс протекает на прикладном уровне, но прежде чем он станет возможным должно быть налажено взаимодействие на более низких уровнях.

Первым шагом в установлении соединения между двумя системами будет выяснение IP — адреса путем отправки запроса на разрешение имени сервера. Адрес позволит протоколу IP направить трафик серверу. Как только клиентской системе становится известен IP — адрес сервера, она устанавливает соединение по протоколу TCP с портом сервера 80, хотя вместо него может использоваться и другой порт, многое зависит от конфигурации веб-сервера.

Как только TCP — соединение установлено, браузер и сервер могут обмениваться сообщениями HTTP.

Каждое сообщение состоит из нескольких элементов.

1. Начальная строка (Start Line) — используется для описания версии используемого протокола или индикатора статуса ответа.

2. Заголовки (Header) — несколько строчек текста в определенном формате, которые уточняют запрос или описывают содержимое тела ответа.

3. Пустая строка (Empty Line) — содержит пустую строку, определяющая окончание раздела заголовка, а также сообщает, что все данные для конкретного запроса или ответа были успешно отправлены.

4. Тело сообщения (Message body) — содержит данные, связанные с запросом, либо документ (например HTML — страницу), передаваемый в ответ.

Сообщение HTTP бывают всего двух типов: запросы и ответы.

- Запросы (HTTP Requests) — сообщения, которые отправляются клиентов на сервер, чтобы вызвать выполнение каких либо действий. Часто используются для получения доступа к определенному ресурсу. Главной частью запроса является HTTP заголовок.

Структура начальной строки всех HTTP - запросов выглядит следующим образом:

*RequestType / RequestURI / HTTPVersion*

Версия 1.1 стандарта протокола HTTP определяет семь основных типов сообщений — запросов, которые используют несколько значений переменной RequestType.

**1.2 HTTP запросы**

**GET**. Позволяет получать информацию с сервера. Он отправляет запрос га сервер с использование URI (Uniform Resource Indentifier) или URL (Uniform Resource Locator), которые определены значением переменной RequestURI.

GET — запросы являются безопасными и имподентными, что означает, что они не должны изменять состояние сервера и могут повторены многократно без влияния на результат. Самое важное, что GET — запрос не должен содержать чувствительных данных, так как информация передающаяся через URL может быть видима в логах сервера или истории браузера.

**HEAD**. Функционально идентичен предыдущему запросу (GET), кроме того, что ответ должен содержать только начальную строку и заголовки, тело сообщения не должно включаться в ответ. Метод HEAD обычно используется в ситуациях, когда клиенту не получать содержимое ресурса, а он нуждается только в такой информации как, заголовки ответа, контроль кэширования, информация о сервере и так далее.

**POST**. Позволяет отправлять данные на сервер. Поддерживает отправку различных типов файлов, среди которых текст, PDF документы и другие типы данных в двоичном виде. Обычно метод POST использую при отправке информации (например заполнение форм логина, пароля, номеров банковских карт и. тд.) и загрузке данных на веб-сайт, таких как изображения и документы. Также метод POST поддерживает тип кодирования данных multipart/form-data, что позволяет передавать файлы.

**OPTIONS**. Позволяет запросить информацию о сервере, в том числе информацию о допускаемых к использованию методов HTTP.

**PUT**. Используется для создания (размещения) новых ресурсов на сервере. Если данный запрос разрешен на сервере без надлежащего контроля, то это может привести к серьезным проблемам безопасности. Например, если сервер принимает PUT — запросы без проверки, злоумышленники могут отправить PUT — запрос для создания или замены исполняемого файла на сервере.

**DELETE**. Запрос состоит из строки запроса содержащий URL ресурса, который требуется удалить. Он может также содержать заголовки, которые предоставляют дополнительную информацию о запросе.

Однако следует отметить, что DELETE запросы могут быть опасными и необратимыми операциями, которые должны выполняться с осторожностью. Также необходимо учесть, что сервер может иметь ограничения на использование данного метода, и в некоторых случаях может потребоваться использовать другие методы запроса для удаления ресурса.

**TRACE**. Запрос может использоваться как администратором сети для оптимизации и отладки сетевого оборудования, так и обычными пользователями для проверки вашего подключения к интернету или отслеживания маршрутов сообщений в сети.

**1.3 HTTP ответы**

HTTP — ответ является сообщением, которое сервер отправляет клиенту в ответ на его запрос. Его структура равна структуре HTTP — запроса: стартовая строка, заголовки и тело.

Изначальная строка HTTP — ответа выглядит так:

*HTTPVersion StatusCode StatusPhrase*

Переменная HTTPVersion определяет стандарт поддерживаемый сервером. Переменные StatusCode и StatusPhrase показывают был ли запрос успешно обработан сервером, если нет, то почему это произошло.

Код представляет собой трехзначное число, а фраза текстовую строку. Значение кодов определяются стандартом протокола HTTP и строго соблюдаются при использовании всеми веб-серверами.

Сервер может вернуть код, принадлежащий одной из пяти категорий кодов состояния. Самые распространенные действующие значения кодов перечислены ниже.

1xx — Коды из данной категории носят исключительно информативный характер и никак не влияют на обработку запроса.

2xx — Коды успешного завершения операции содержат два в качестве первой цифры и подразумевают, что сообщение — запрос клиента было успешно получено, распознано и принято.

200 — OK (все в порядке). Показывает, что запрос был успешно обработан и ответ содержит информацию, соответствующую типу запроса.

201 — Created (создан). Показывает, что запрос был успешно обработан и был создан новый ресурс.

202 — Accepted (принят). Показывает, что запрос был принят, однако его обработка еще не завершена.

3xx — Коды перенаправления содержат три в качестве первой цифры и демонстрируют, что требуется дальнейшие действия со стороны клиента для успешной обработки запроса. Самые распространенные действующие значения этого типа кодов перечислены ниже.

300 — Multiple Choices (множественный выбор). Показывает, что ответ содержит список ресурсов, которые могут быть использованы для удовлетворения запроса, но пользователь должен выбрать одно из них.

301 — Moved Permanently (перемещен окончательно). Показывает, что запрашиваемому ресурсу был присвоен новый URI и все дальнейшие ссылки на этот ресурс должны использовать один из новых URI, указанных в ответе.

302 — Found (найден). Показывает, что запрашиваемый ресурс находится постоянно под другим URI, но клиент должен продолжать использование старого значения URI в своих следующих запросах.

4xx — Коды ошибок клиента содержат четыре в качестве первой цифры и указывают, что запрос не удается успешно обработать из-за ошибок на стороне клиента.

400 — Bad Request (неправильный запрос). Сервер не может распознать запрос клиента из-за синтаксических ошибок.

404 — Not Found (не найден). Сервер не может обнаружить ресурс, определенный значением переменной RequestURI.

405 — Method Not Allowed (недопустимый метод). Данный тип запроса не может быть использован для заданного значения переменной RequestURI.

409 — Conflict (конфликт). Запрос не может быть обработан, так как имеется конфликт с текущим состоянием запрашиваемого ресурса, например, когда команда PUT пытается провести запись данных, в то время как ресурс уже задействован другим процессом.

5xx — Коды ошибок сервера содержат пять в качестве своей первой цифры и показывают, что запрос не может быть обработан из-за неполадок в работе сервера.

500 — Interal Server Error (внутренняя ошибка сервера). Сервер столкнулся с непредвиденными событиями, которые не позволили ему удовлетворить запрос.

501 — Not Implemented (возможность не реализована). Сервер не поддерживает функцию, необходимую для удовлетворения запроса.

503 — Service Unavailable (сервис недоступен). Сервер не может обработать запрос, так как в настоящий момент перегружен или закрыт на техническое обслуживание.

505 — HTTP Version Not Supported (версия HTTP не поддерживается). Сервер не поддерживает или отказывается поддерживать ту версию протокола HTTP, которая указана в сообщении-запросе.

После начальной строки сообщение-ответ может содержать серии заголовков. Как и те, что присутствуют в запросе, они обеспечивают дополнительную информацию о самом сервере и данном ответе. Если размер файла превышает величину одной стандартной дейтаграммы, сервер создает добавочное ответ-сообщение, содержащее тела сообщений, но у них нет начальной строки и заголовков.

**1.4 Структура клиента**

HHTP клиент должен состоять из следующих базовых элементов:

1. Установка TCP соединения: Клиент должен установить TCP-соединение с сервером, к которому он хочет отправить HTTP запросы. Для этого клиент должен знать IP-адрес сервера и номер порта, по которому он должен будет устанавливать соединение.

2. Отправка HTTP запросов: Клиент должен сформировать корректный HTTP запрос, включающий метод запроса (GET, POST, DELETE и.т.д), путь к ресурсу на сервере, заголовки запроса и при необходимости тело запроса.

3. Прием HTTP ответа: Клиент должен дождаться ответа от сервера после отправки запроса. Когда ответ получен, клиент должен прочитать заголовки ответа, из которых можно узнать статус запроса (Например, 200 OK, 404 Not Found) и в случае успеха, данные которые отправил сервер в ответе.

4. Обработка ошибок: Клиент должен быть способен обрабатывать различные типы ошибок, которые могут произойти во время работы. Например клиент должен обработать ошибку подключения к серверу, ошибку сети и.т.д.

5. Закрытие соединения: После завершения работы клиент должен закрыть соединение с сервером, чтобы освободить ресурсы.

Данная структура является базовой, ее можно использовать для создания HTTP-клиента. При необходимости может быть расширена и дополнена в зависимости от конкретных задач.

**1.5 Используемые инструменты разработки**

Для разработки приложения была выбрана платформа поддерживающая стандарт POSIX(Portable Operating System Interface for Unix), который является платформо-независимым системным интерфейсом для компьютерного окружения.

POSIX стандарт был разработан в IEEE (Institut of Electrical and Electronics Engineers) и впервые был опубликован в 1988 году. Он устанавливает набор спецификаций для обеспечения совместимости между различными операционными системами, основанными на архитектуре Unix, что позволяет разработчикам программного обеспечения переносить свои приложения без необходимости переписывать код с нуля.

**Задачи Posix**

* Упрощение переноса реализации программ на различным платформам.
* Установление и согласование интерфейсов в процессе проектирования, а не при их выполнении.
* Сохранение и учёт всех основных созданных и используемых приложений.
* Определение минимально необходимых интерфейсов для повышения эффективности создания, утверждения и подтверждения документации.
* Развитие стандартов для обеспечения сетевых коммуникации, распределения обработки данных и защита информации.

**Стандарт состоит из четырех основных разделов**:

1. Основные определения — список основных соглашений, используемых в спецификациях, и список заголовочных файлов языка Си.

2. Оболочки и утилиты — описание утилит и командной оболочки sh.

3. Системные интерфейсы — список системных интерфейсов языка Си.

4. Обоснование — объяснение принципов которые используются в стандарте.

Так же для создания HTTP клиента нам потребуется веб-сервер.

Веб-сервер является программным обеспечением, которое позволяет обслуживать HTTP — запросы, принимая и отвечая на них. Он осуществляет обработку запросов от клиентов (Web-браузеров) и возвращает соответствующие запросу ответы.

В данной курсовой работе будет использоваться один из более популярных веб -серверов в мире — Apache. Он обладает широким набором нужных функций и гибкостью, что делает его идеальным выбором для множества проектов и приложений. Apache является кроссплатформенным веб-сервером и поддерживает различные операционные системы, включая Microsoft Windows, Linux, macOS и другие.

**Основные преимущества использования Apache** :

1.Простая настройка — Apache имеет интуитивно понятную конфигурацию, позволяя быстро и легко настроить сервер в соответствии с нужными задачами.

2. Гибкость: Apache, поддерживает множество модулей и расширений, которые расширяют функционал сервера. Это позволяет пользователю настраивать сервер в соответствии с требованиями конкретной задачи.

3. Безопасность: Apache имеет ряд функций, таких как поддержка SSL — шифрования, контроль доступа на основе IP — адресов и паролей, блокировка запросов от злоумышленников и многое другое. Это помогает обеспечить безопасность веб-приложений и данных.

4. Высокая производительность: Apache обладает оптимизированным кодом, что позволяет достичь высокой производительности даже при обработке большего количество запросов.

5. Богатая документация: Apache имеет обширную документацию, которая позволяет пользователям настраивать, оптимизировать сервер в соответствии с их потребностями.

В целом, использование Apache Веб - сервера предоставляет надежное и гибкое решение для обслуживания HTTP — запросов и обеспечения доступности веб-приложений.

Для написания самого клиента будет использоваться язык программирования Си. Использование языка программирования Си для написания HTTP - клиента обладает рядом преимуществ и особенностей.

1. Производительность: язык Си является низкоуровневым языком, который позволяет максимально эффективно использовать ресурсы компьютера. Код написанный на Си, выполняется очень быстро и имеет малый объем памяти, что особенно важно при обработке большего объема HTTP — запросов.

2. Богатая стандартная библиотека: язык Си предоставляет обширную стандартную библиотеку, которая содержит различные функции и инструменты для обработки сетевых соединений, включая функции для создания и управления HTTP — запросами.

3. Портативность: язык Си является платформо-независимым, что означает, что код, написанный на Си, может быть скомпилирован на различных устройствах и архитектурах.

Для проверки работы клиента необходимо написать скрипт, который будет проверять запросы. Данный скрипт будет реализован на серверном языке PHP.

Для написания серверного скрипта для проверки работоспособности приложения был выбран язык PHP, так как PHP — это распространенный язык программирования общего назначения с открытым исходным кодом. PHP специально создан для веб — разработок, и его код можно внедрить непосредственно в HTML.

Хотя PHP, главным образом предназначен для работы в среде веб-разработки, область его применения не ограничивается только этим.

Основные области применения языка PHP:

1. Создание серверных скриптов: PHP широко используется для создания скриптов, которые выполняются на стороне сервера. Это позволяет создавать динамические веб-страницы, работать с базами данных и выполнять другие серверные задачи.

2. Клиентские приложения: PHP может быть использован для создания оконных приложений, которые выполняются на стороне клиента. PHP не является самым распространенным языком для разработки клиентских приложений, можно использовать PHP-GTK для создания некоторых форм приложений. Это полезно, когда нужно создать приложения с функциями, которые будут доступны для нескольких операционных систем.

Язык программирования PHP доступен на многих операционных системах, включая Linux, многие модификации Unix, Microsoft Windows, macOS и многие другие. Также в PHP подключена поддержка веб-серверов таких как Apache, Nginx и многих других.

**2. Практическая часть**

**2.1 Настройка и установка веб-сервера Apache**

Шаг 1: Загрузка Apache.

Первым делом необходимо загрузить Apache HTTP Server. Для установки Apache открываем терминал UNIX системы и прописываем команду для установки веб-сервера, через репозитории. Подтверждаем установку нажав, Y(Yes), N (NO).

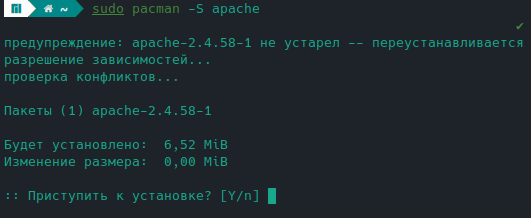


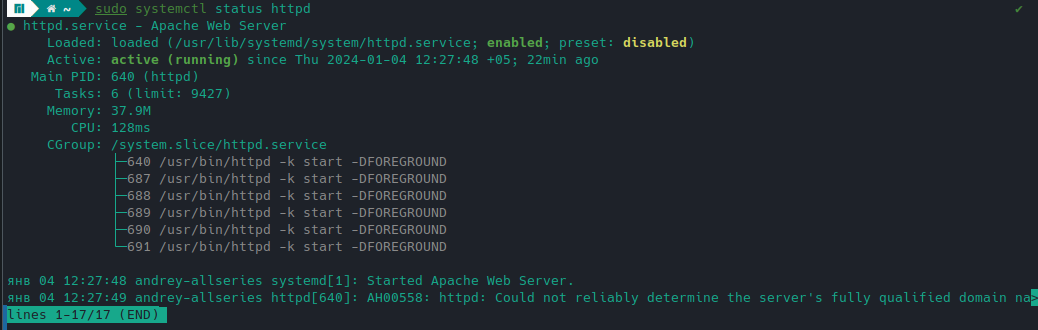
Рис. 1. Уcтановка веб-сервера Apache.

Следующим шагом нам нужно запустить веб-сервер, используем эту команду:

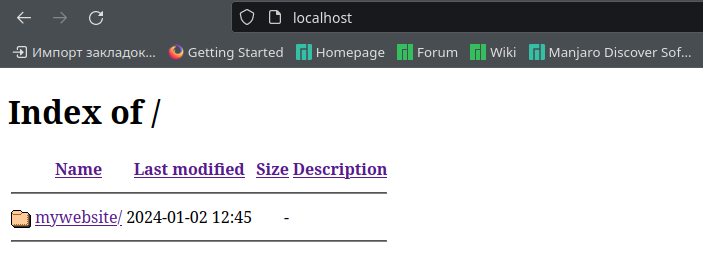


Рис. 2. Запуск веб-сервера Apache.

После установки нам нужно проверить активен ли наш веб-сервер, для этого используем команду status. Если статут enabled, значит все хорошо установилось.

 Рис. 3. Проверяем, активен ли веб-сервер.

Откроем веб-браузер и введем в адресной строке «[http://localhost](http://localhost/)».Если все настройки выполнены правильно, вы должны увидеть страницу приветствия Apache или директорию ваших проектов.

 Рис. 4. Проверяем работу localhost.

**2.2 Реализация HTTP клиента на языке Си.**

Для создания клиента будем пользоваться следующим алгоритмом:

Клиентский алгоритм.

1. Создание сокета.

2. Определение адреса сервера.

3. Подключение.

4. Чтение и отображение сообщений.

5. Разрыв соединения.

Для клиента будем использовать не специализированные библиотеки языка Си.

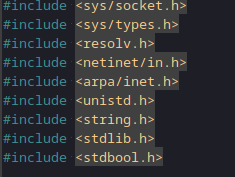
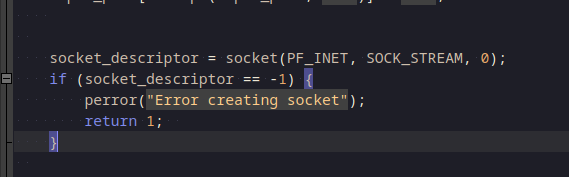


Рис. 5. Используемые библиотеки.

Рис. 6. Создание сокета.

Функция socket создает новый сокет и возвращает его дескриптор. Если создание сокета прошло успешно, то в переменную socket\_descriptor будет записан его дескриптор (целое число, кторое будет использоваться для обращения к сокету). Функция принимает 3 аргумента:

PF\_INET — это константа указывающая на использование протокола IPV4.

SOCK\_STREAM — это константа указывающая на использование протокола TCP.

«0» это флаг, который указывает на использование протокола по умолчанию.

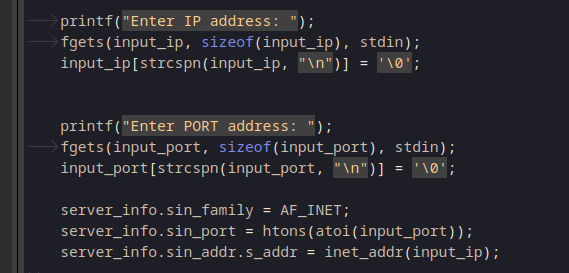
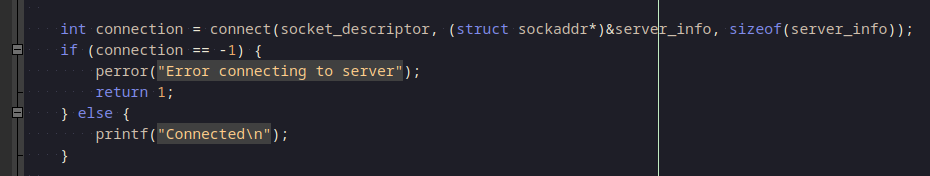


Рис. 7. Пользовательский ввод IP и порта.

В данном коде адрес сервера определяется с помощью пользовательского ввода. При выполнении программы пользователю предлагается ввести IP-адрес и порт сервера. Эти значения сохраняются в переменных input\_ip и input\_port соответственно. Далее, при настройке структуры server\_info, используется функция inet\_addr(input\_ip) для преобразования введенного пользователем IP-адреса в формат, который можно использовать для настройки соединения. Адрес сервера определяется в строке server\_info.sin\_addr.s\_addr = inet\_addr(input\_ip).

 Рис. 8. Создание соединения.

1. Создается переменная connection, которая будет содержать результат вызова функции connect() для установки соединения с сервером.

2. В функцию connect() передается следующая информация:

- socket\_descriptor : дескриптор сокета (указатель на открытый сокет);

- (struct sockaddr\*)&server\_info: указатель на структуру server\_info приводится к типу struct sockaddr\* с помощью оператора (struct sockaddr\*);

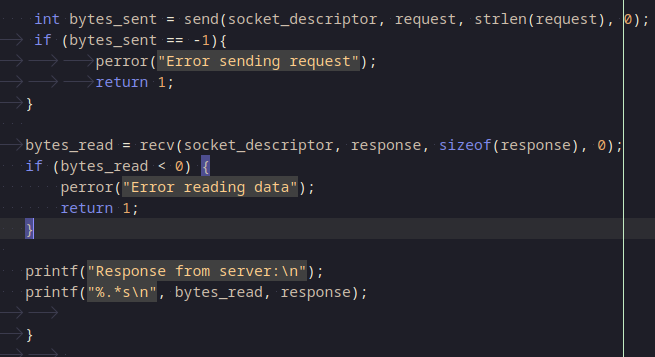
- sizeof(server\_info): размер структуры server\_info.

3. Возвращаемое значение функции connect() присваивается переменной `connection`.

4. В блоке if проверяется, равно ли connection -1 (это означает, что установка соединения завершилась с ошибкой).

- Если connection равно -1, то выводится сообщение об ошибке с помощью функции perror(), и программа завершается с кодом 1 с помощью оператора return 1.

- Если connection не равно -1, значит соединение успешно установлено, и выводится сообщение "Connected" с помощью функции printf(). Таким образом, программа продолжит свое выполнение.

 Рис. 9. Чтение и отображение сообщений.

Определяется переменная "bytes\_sent", которой присваивается количество отправленных байтов данных по сокету "socket\_descriptor" с помощью функции "send".

Проверяется, равно ли значение "bytes\_sent" -1. Если да, то выводится сообщение об ошибке "Error sending request" с помощью функции "perror", и функция возвращает 1.

Определяется переменная "bytes\_read", которой присваивается количество прочитанных байтов данных из сокета "socket\_descriptor" в переменную "response" с помощью функции "recv".

Проверяется, меньше ли значение "bytes\_read" нуля. Если да, то выводится сообщение об ошибке "Error reading data" с помощью функции "perror", и функция возвращает 1.

Выводится сообщение "Response from server:".

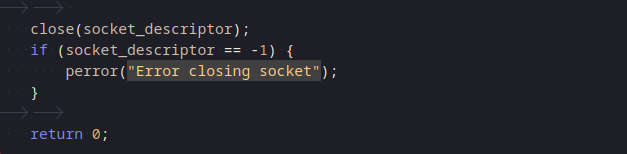
Выводится содержимое переменной "response" с учетом количества прочитанных байтов "bytes\_read" с помощью функции "printf".

Рис. 10. Закрытие соединения.

Код закрывает сокет с помощью функции close и проверяет значение возвращаемое этой функцией. Если возвращаемое значение равно -1, выводится сообщение об ошибке с помощью функции perror. Затем код возвращает 0.

**2.3 Создание PHP скрипта для проверки запросов.**

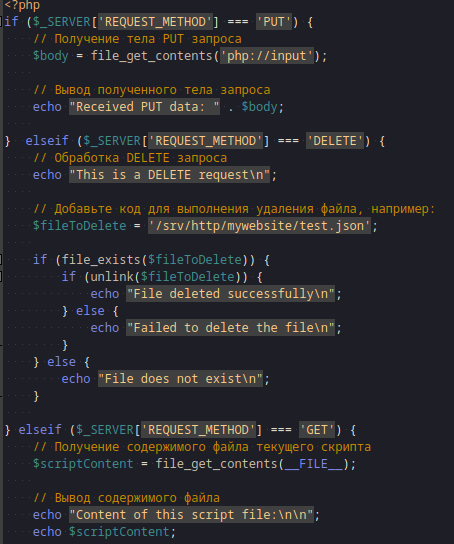


Рис. 10. Проверка PUT, DELETE и GET запросов.

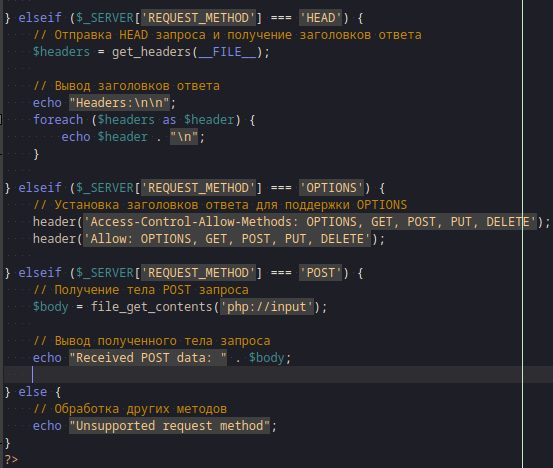


Рис. 11. Проверка HEAD, OPTIONS и POST запросов.

**Заключение**

В рамках данной курсовой работы были достигнуты следующие цели:

1. Создание прототипа HTTP клиента на языке Си. В ходе работы были изучены основы протокола HTTP, а также различные библиотеки и инструменты для работы с HTTP запросами на языке Си.

2. Также в ходе курсовой работы был создан скрипт для проверки HTTP запросов на стороне сервера с использованием языка PHP. Скрипт принимает запросы от HTTP-клиента и выполняет необходимые операции.

Полученные результаты являются хорошей основой для дальнейшей разработки и улучшение клиента и сервера.

**Список используемой литературы**

1. POSIX [Электронный ресурс] // Википедия: свобод. энцикл. – Режим доступа: <https://ru.wikipedia.org/wiki/POSIX> – 04.01.2024
2. HTPP [Электронный ресурс] // Википедия: свобод. энцикл. – Режим доступа: https://ru.wikipedia.org/wiki/HTTP– 03.01.2024

**Приложение**

Код Клиента:

// Создание http клиента

#include <sys/socket.h>

#include <sys/types.h>

#include <resolv.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <unistd.h>

#include <string.h>

#include <stdlib.h>

#include <stdbool.h>

#define MAXBUF 1024

int main() {

int socket\_descriptor, bytes\_read;

struct sockaddr\_in server\_info;

char input\_ip[16];

char input\_port[6];

socket\_descriptor = socket(PF\_INET, SOCK\_STREAM, 0);

if (socket\_descriptor == -1) {

perror("Error creating socket");

return 1;

}

printf("Enter IP address: ");

fgets(input\_ip, sizeof(input\_ip), stdin);

input\_ip[strcspn(input\_ip, "\n")] = '\0';

printf("Enter PORT address: ");

fgets(input\_port, sizeof(input\_port), stdin);

input\_port[strcspn(input\_port, "\n")] = '\0';

server\_info.sin\_family = AF\_INET;

server\_info.sin\_port = htons(atoi(input\_port));

server\_info.sin\_addr.s\_addr = inet\_addr(input\_ip);

int connection = connect(socket\_descriptor, (struct sockaddr\*)&server\_info, sizeof(server\_info));

if (connection == -1) {

perror("Error connecting to server");

return 1;

} else {

printf("Connected\n");

}

char request[MAXBUF];

char response[MAXBUF];

char method[8];

printf("Enter method\n");

printf("1. GET\n");

printf("2. HEAD\n");

printf("3. OPTIONS\n");

printf("4. POST\n");

printf("5. TRACE\n");

printf("6. PUT\n");

printf("7. DELETE\n");

printf("Method number: ");

fgets(method, sizeof(method), stdin);

if (method[strlen(method) - 1] == '\n') {

method[strlen(method) - 1] = '\0';

}

switch (method[0]) {

case '1':

snprintf(request, sizeof(request), "GET /mywebsite/put.php HTTP/1.1\r\nHost: %s\r\n\r\n", "localhost");

break;

case '2':

snprintf(request, sizeof(request), "HEAD /mywebsite/put.php HTTP/1.1\r\nHost: %s\r\n\r\n", "localhost");

break;

case '3':

snprintf(request, sizeof(request), "OPTIONS /mywebsite/put.php HTTP/1.1\r\nHost:%s\r\n\r\n", "localhost");

break;

case '4':

snprintf(request, sizeof(request), "POST /mywebsite/put.php HTTP/1.1\r\nHost:%s\r\nContent-Type: application/x-www-form-urlencoded\r\nContent-Length: 14\r\n\r\nHello world!!!\r\n\r\n", "localhost");

break;

case '5':

snprintf(request, sizeof(request), "TRACE /mywebsite/ HTTP/1.1\r\nHost:%s\r\n\r\n", "localhost");

break;

case '6':

snprintf(request, sizeof(request), "PUT /mywebsite/put.php HTTP/1.1\r\nHost: localhost\r\nContent-Type: text/plain\r\nContent-Length: 15\r\n\r\nHello, Server!\r\n");

break;

case '7':

snprintf(request, sizeof(request), "DELETE /mywebsite/put.php HTTP/1.1\r\nHost:%s\r\n\r\n", "localhost");

break;

default:

printf("Invalid method number\n");

break;

}

int bytes\_sent = send(socket\_descriptor, request, strlen(request), 0);

if (bytes\_sent == -1) {

perror("Error sending request");

}

bytes\_read = recv(socket\_descriptor, response, sizeof(response), 0);

if (bytes\_read < 0) {

perror("Error reading data");

} else if (bytes\_read == 0) {

printf("Connection closed by server\n");

}

printf("Response from server:\n");

printf("%.\*s\n", bytes\_read, response);

close(socket\_descriptor);

if (socket\_descriptor == -1) {

perror("Error closing socket");

}

return 0;

}

PHP скрипт:

<?php

if ($\_SERVER['REQUEST\_METHOD'] === 'PUT') {

// Получение тела PUT запроса

$body = file\_get\_contents('php://input');

// Вывод полученного тела запроса

echo "Received PUT data: " . $body;

} elseif ($\_SERVER['REQUEST\_METHOD'] === 'DELETE') {

// Обработка DELETE запроса

echo "This is a DELETE request\n";

// Добавьте путь для выполнения удаления файла, например:

$fileToDelete = '/srv/http/mywebsite/test.json';

if (file\_exists($fileToDelete)) {

if (unlink($fileToDelete)) {

echo "File deleted successfully\n";

} else {

echo "Failed to delete the file\n";

}

} else {

echo "File does not exist\n";

}

} elseif ($\_SERVER['REQUEST\_METHOD'] === 'GET') {

// Получение содержимого файла текущего скрипта

$scriptContent = file\_get\_contents(\_\_FILE\_\_);

// Вывод содержимого файла

echo "Content of this script file:\n\n";

echo $scriptContent;

} elseif ($\_SERVER['REQUEST\_METHOD'] === 'HEAD') {

// Отправка HEAD запроса и получение заголовков ответа

$headers = get\_headers(\_\_FILE\_\_);

// Вывод заголовков ответа

echo "Headers:\n\n";

foreach ($headers as $header) {

echo $header . "\n";

}

} elseif ($\_SERVER['REQUEST\_METHOD'] === 'OPTIONS') {

// Установка заголовков ответа для поддержки OPTIONS

header('Access-Control-Allow-Methods: OPTIONS, GET, POST, PUT, DELETE');

header('Allow: OPTIONS, GET, POST, PUT, DELETE');

} elseif ($\_SERVER['REQUEST\_METHOD'] === 'POST') {

// Получение тела POST запроса

$body = file\_get\_contents('php://input');

// Вывод полученного тела запроса

echo "Received POST data: " . $body;

} else {

// Обработка других методов

echo "Unsupported request method"; } ?>