Министерство образования и науки Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«Волгоградский государственный технический университет»

Факультет электроники и вычислительной техники

Кафедра «ЭВМ и С»

Создание сервера (клиента) для протокола прикладного уровня HTTPS на основе протокола транспортного уровня TCP

Выполнили:
студенты группы ЭВМ-1.н:

Ноздренков С.В.

Наср Тарек М.

Проверил:

Ст. преп. Жариков Д.Н.

Волгоград, 2014

Содержание:

Введение

HTTPS (HyperText Transfer Protocol Secure) — расширение протокола HTTP, поддерживающее шифрование. Данные, передаваемые по протоколу HTTP, «упаковываются» в криптографический протокол SSL или TLS. В отличие от HTTP, для HTTPS по умолчанию используется TCP-порт 443.

Протокол был разработан компанией Netscape Communications для браузера Netscape Navigator в 1994 году. HTTPS широко используется в мире веб и поддерживается всеми популярными браузерами.

HTTPS не является отдельным протоколом. Это обычный HTTP, работающий через шифрованные транспортные механизмы SSL и TLS. Он обеспечивает защиту от атак, основанных на прослушивании сетевого соединения — от снифферских атак и атак типа man-in-the-middle, при условии, что будут использоваться шифрующие средства и сертификат сервера проверен и ему доверяют.

По умолчанию HTTPS URL использует 443 TCP-порт (для незащищённого HTTP — 80). Чтобы подготовить веб-сервер для обработки https-соединений, администратор должен получить и установить в систему сертификат для этого веб-сервера. Сертификат состоит из 2 частей (2 ключей) — public и private. Public-часть сертификата используется для зашифровывания трафика от клиента к серверу в защищённом соединении, private-часть — для расшифровывания полученного от клиента зашифрованного трафика на сервере. После того как пара ключей приватный/публичный сгенерированы, на основе публичного ключа формируется запрос на сертификат в Центр сертификации, в ответ на который ЦС высылает подписанный сертификат. ЦС при подписании проверяет клиента, что позволяет ему гарантировать, что держатель сертификата является тем, за кого себя выдаёт (обычно это платная услуга).

Существует возможность создать такой сертификат, не обращаясь в ЦС. Такие сертификаты могут быть созданы для серверов, работающих под

Unix, с помощью таких утилит, как ssl-ca от OpenSSL или gensslcert от SuSE. Подписываются такие сертификаты этим же сертификатом и называются самоподписанными (self-signed). Без проверки сертификата каким-то другим способом (например, звонок владельцу и проверка контрольной суммы сертификата) такое использование HTTPS подвержено атаке man-in-the-middle.

Эта система также может использоваться для аутентификации клиента, чтобы обеспечить доступ к серверу только авторизованным пользователям. Для этого администратор обычно создаёт сертификаты для каждого пользователя и загружает их в браузер каждого пользователя. Также будут приниматься все сертификаты, подписанные организациями, которым доверяет сервер. Такой сертификат обычно содержит имя и адрес электронной почты авторизованного пользователя, которые проверяются при каждом соединении, чтобы проверить личность пользователя без ввода пароля.

В HTTPS для шифрования используется длина ключа 40, 56, 128 или 256 бит. Некоторые старые версии браузеров используют длину ключа 40 бит (пример тому — IE версий до 4.0), что связано с экспортными ограничениями в США. Длина ключа 40 бит не является сколько-нибудь надёжной. Многие современные сайты требуют использования новых версий браузеров, поддерживающих шифрование с длиной ключа 128 бит, с целью обеспечить достаточный уровень безопасности. Такое шифрование значительно затрудняет злоумышленнику поиск паролей и другой личной информации.

В данной работе перед нами стоит задача реализации протокола HTTPS в виде клиент-серверного протокола.

1. Задание

Реализация протокола прикладного уровня HTTPS на основе протокола транспортного уровня TCP. Сервер должен предоставлять возможности работы следующих методов HTTPS: HEAD, GET POST, PUT, DELETE, AND OPTIONS. Сервер должен сообщать об ошибках запросов, соединений, данных в ответе клиенту. Клиент должен предоставлять возможности просмотра документов сервера.

2. Теоритическая часть

2.1. Протокол TCP

TCP (Transmission Control Protocol, Протокол управления передачей) был спроектирован в качестве связующего протокола для обеспечения интерактивной работы между компьютерами. TCP обеспечивает надежность и достоверность обмена данными между процессами на компьютерах, входящих в общую сеть. TCP, с одной стороны, взаимодействует с прикладным протоколом пользовательского приложения, а с другой, с протоколом, обеспечивающим \"низкоуровневые\" функции: маршрутизацию и адресацию пакетов, которые, как правило, выполняет IP.

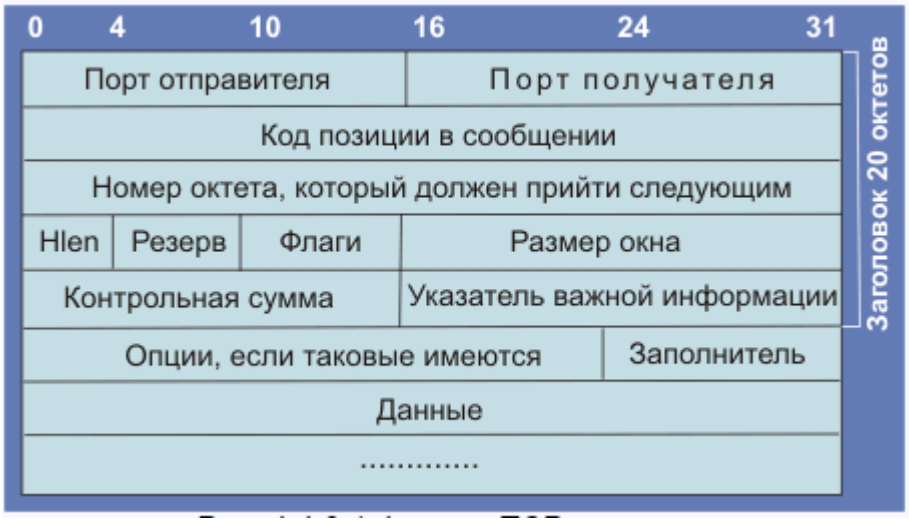


Рисунок 1 - Формат TCP-сегмента

В операционной системе реализация TCP представляет собой отдельный системный модуль (драйвер), через который, как правило, проходят все вызовы функций протокола. Интерфейс между прикладным процессом и TCP представляет собой библиотеку вызовов, такую же, как библиотека системных вызовов, например, для работы с файлами. Вы можете открыть или закрыть соединение (как открыть или закрыть файл) и отправить или принять данные из установленного соединения (аналогично операциям чтения и записи файла). Вызовы TCP могут работать с прикладным приложением в асинхронном режиме. Безусловно, реализация TCP в каждой системе может осуществлять множество собственных функций, но любая из этих реализации должна обеспечивать минимум функциональности, которая требуется стандартами TCP.

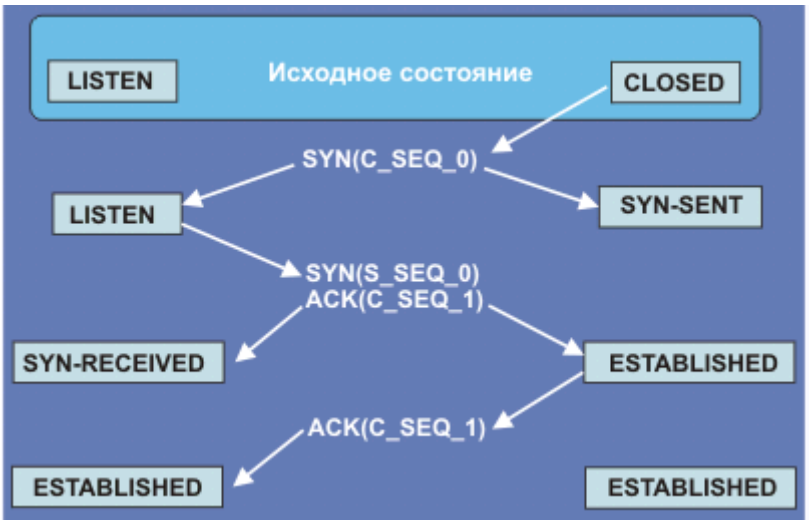


Рисунок 2 - Алгоритм установления связи.

Схема работы пользовательского приложения с TCP, в общих чертах, состоит в следующем. Для передачи данных пользовательскому процессу надо вызвать соответствующую функцию TCP, с указанием на буфер передаваемых данных. TCP упаковывает эти данные в сегменты своего стека и вызывает функцию передачи протокола нижнего уровня, например IP.

Для установления соединения между двумя процессами на различных компьютерах сети необходимо знать не только Internet-адреса компьютеров, но и номер TCP-порта, который процесс использует на данном компьютере.

В совокупности с Internet-адресом компьютера порты образуют систему гнезд (sockets). Пара гнезд уникально идентифицирует каждое соединение или поток данных в сети Internet, а порт обеспечивает независимость каждого TCP-канала на данном компьютере. Безусловно, несколько процессов на машине могут использовать один и тот же TCP-порт, но с точки зрения удаленного процесса между этими процессами не будет никакой разницы.

Одно гнездо на компьютере может быть задействовано в соединениях с несколькими гнездами на удаленных компьютерах. Кроме того, одно и то же гнездо может передавать поток данных в обоих направлениях. Таким образом, механизм гнезд позволяет на одном компьютере одновременно работать нескольким приложениям и уникально идентифицирует каждый поток данных сети. Это называется мультиплексированием соединений.

Модули TCP, UDP и драйвер Ethernet являются мультиплексорами типа n x 1. Действуя как мультиплексоры, они переключают несколько входов на один выход. Они также являются демультиплексорами типа 1 x n. Как демультиплексоры, они переключают один вход на один из многих выходов в соответствии с полем типа в заголовке протокольного блока данных. Когда Ethernet-кадр попадает в драйвер сетевого интерфейса Ethernet, он может быть направлен либо в модуль ARP, либо в модуль IP. (Значение поля типа в заголовке кадра указывает, куда должен быть направлен Ethernet-кадр).

Если IP-пакет попадает в модуль IP, то содержащиеся в нем данные могут быть переданы либо модулю TCP, либо UDP, что определяется полем \"Protocol\" в заголовке IP-пакета. Если TCP-сообщение попадает в модуль TCP, то выбор прикладной программы, которой должно быть передано сообщение, осуществляется на основе значения поля \"порт\" в заголовке TCP-сообщения.

Назначение портов приложениям на каждом компьютере происходит независимо друг от друга. TCP может самостоятельно выбирать порт, с

которым будет работать приложение, или приложение укажет, с каким портом на данном компьютере оно будет работать. Однако, как правило, часто используемые приложения - сервисы используют одни и те же номера портов, которые уже стали общеизвестными, например, такие как HTTP, FTP, SMTP и др., для того, чтобы к данному процессу на компьютере можно было присоединиться, указывая только адрес машины. Например, Internet браузер, если ему не указать дополнительно, ищет по указанному адресу приложение, работающее с портом 80, - это наиболее распространенный порт для серверов WWW.

Рисунок 3 – схема использования алгоритма скользящего окна.

Протокол TCP должен уметь работать с поврежденными, потерянными, дублированными или поступившими с изменением порядка пакетами. Это достигается благодаря механизму присвоения порядкового номера каждому передаваемому пакету данных и механизму проверок получения пакетов подтверждения доставки.

Когда TCP передает сегмент данных, копия этих данных помещается в очередь повтора передачи и запускается таймер ожидания подтверждения. Когда система получает подтверждение - сегмент TCP, содержащий управляющий флаг ACK, что этот пакет данных получен, она удаляет его из очереди. Сегмент подтверждения получения содержит номер полученного сегмента, на основании которого и происходит контроль доставки данных адресату. Если подтверждение не поступило до истечения срока таймера, пакет отправляется еще раз. Уведомление TCP о получении пакета данных

еще не означает, что он был доставлен конечному пользователю. Оно только означает, что TCP выполнил возложенные на него функции.

При передаче информации каждому байту данных присваивается порядковый номер, поэтому, в какой бы последовательности эти пакеты не достигали точки назначения, они всегда будут собраны в изначально заданной последовательности. Порядковый номер первого байта данных в передаваемом сегменте называется порядковым номером сегмента. Нумерация проводится \"с головы состава\", т. е. от заголовка пакета. TCP-пакет также содержит \"подтверждающий номер\" (acknowledgment number), который представляет собой номер следующего ожидаемого пакета данных передачи в обратном направлении. Иными словами, номер обозначает: \"до сих пор я все получил\". Механизм с использованием \"подтверждающего номера\" позволяет исключать дублирование пакетов при повторной отправке недоставленных данных.

Кроме определения порядка следования информационных пакетов, \"порядковый номер\" играет большую роль в механизме синхронизации соединения и контроле потерянных пакетов при разрывах соединения. Однако необходимо помнить, что величина счетчика - нумератора все же ограничена. Пакеты могут нумероваться числами от 0 до 2(32-1). Таким образом, все арифметические операции со счетчиком пакетов производятся по модулю 232. Это не означает, что гнезда, в процессе соединения, могут обмениваться только ограниченным количеством пакетов. Поскольку в процессе обмена получатель и отправитель знают предыдущий, последующий номера пакетов и длину пакета, а эти величины хранятся в структуре TCB при образовании соединения, все операции сравнения по модулю 232 проводятся корректно.

Здесь стоит сказать несколько слов о механизме предотвращения появления в сети пакетов с одинаковыми номерами. Они могут появиться, например, при установлении и быстром сбросе соединения или при сбросе

соединения и его быстром восстановлении, т. е. когда номер испорченного пакета может сразу использоваться новым пакетом данных. Механизм предотвращения подобных ситуаций основан на генерировании начального числа последовательности пакетов, а поскольку счетчик циклический, то не все ли равно, с какого места начинать отсчет.

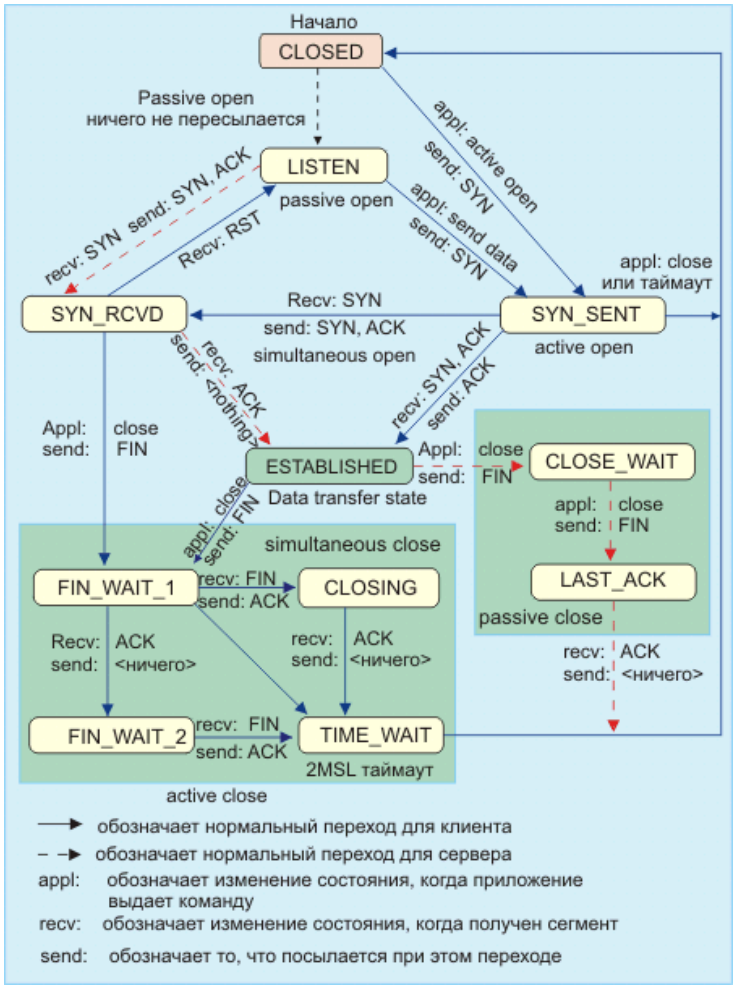


Рисунок 4 – схема состояний протокола TCP.

Так, при установлении нового соединения генерируется 32-битное число ISN (Initial Sequence Number). Генератор использует 32 младших разряда машинного таймера, который меняется каждые 4 микросекунды (полный цикл - 4,55 часа). Это число и служит отсчетом нумератора пакетов. Кроме того, каждая дейтаграмма в сети имеет ограниченное время жизни MSL - Maximum Life Time, которое значительно меньше периода генератора. Таким образом, в сети гарантируется невозможность возникновения конфликтов пакетов с одинаковыми номерами.

Поврежденные пакеты отсеиваются механизмом проверки величины контрольной суммы данных, которая размещается в каждом передаваемом пакете.

2.2 Протоколы HTTP/HTTPS

Протокол HTTP - это протокол запросов/ответов. Клиент посылает серверу запрос, содержащий метод запроса, URI, версию протокола, MIME-подобное сообщение, содержащее модификаторы запроса, клиентскую информацию, и, возможно, тело запроса, по соединению. Сервер отвечает строкой состояния, включающей версию протокола сообщения, код успешного выполнения или код ошибки, MIME-подобное сообщение, содержащее информацию о сервере, метаинформацию объекта, и, возможно, тело объекта.

HTTP соединение обычно происходит посредством TCP/IP соединений. Заданный по умолчанию порт TCP - 80, но могут использоваться и другие порты. HTTP также может быть реализован посредством любого другого протокола Интернета, или других сетей. HTTP необходима только надежная передача данных, следовательно может использоваться любой протокол, который гарантирует надежную передачу данных; отображение структуры запроса и ответа HTTP/1.1 на транспортные модули данных рассматриваемого протокола - вопрос, не решаемый этой спецификацией.

Стартовая (начальная) строка запроса для HTTP 1.1 составляется по следующей схеме:

«Метод URI HTTP/Версия»

Например (такая стартовая строка может указывать на то, что запрашивается главная страница сайта): «GET / HTTP/1.1»

Метод (в англоязычной тематической литературе используется слово method, а также иногда слово verb — «глагол») представляет собой

последовательность из любых символов, кроме управляющих и разделителей, и определяет операцию, которую нужно осуществить с указанным ресурсом. Спецификация HTTP 1.1 не ограничивает количество разных методов, которые могут быть использованы, однако в целях соответствия общим стандартам и сохранения совместимости с максимально широким спектром программного обеспечения, как правило, используются лишь некоторые, наиболее стандартные методы, смысл которых однозначно раскрыт в спецификации протокола.

Каждый сервер обязан поддерживать как минимум методы GET и HEAD. Если сервер не распознал указанный клиентом метод, то он должен вернуть статус 501 (Not Implemented). Если серверу метод известен, но он неприменим к конкретному ресурсу, то возвращается сообщение с кодом 405 (Method Not Allowed). В обоих случаях серверу следует включить в сообщение ответа заголовок Allow со списком поддерживаемых методов.

OPTIONS используется для определения возможностей веб-сервера или параметров соединения для конкретного ресурса. В ответ серверу следует включить заголовок Allow со списком поддерживаемых методов. Также в заголовке ответа может включаться информация о поддерживаемых расширениях.

Предполагается, что запрос клиента может содержать тело сообщения для указания интересующих его сведений. Формат тела и порядок работы с ним в настоящий момент не определён. Сервер пока должен его игнорировать. Аналогичная ситуация и с телом в ответе сервера.

GET используется для запроса содержимого указанного ресурса. С помощью метода GET можно также начать какой-либо процесс. В этом случае в тело ответного сообщения следует включить информацию о ходе выполнения процесса. Клиент может передавать параметры выполнения запроса в URI целевого ресурса после символа «?»:

HEAD аналогичен методу GET, за исключением того, что в ответе сервера отсутствует тело. Запрос HEAD обычно применяется для извлечения метаданных, проверки наличия ресурса (валидация URL) и чтобы узнать, не изменился ли он с момента последнего обращения.

POST применяется для передачи пользовательских данных заданному ресурсу. Например, в блогах посетители обычно могут вводить свои комментарии к записям в HTML-форму, после чего они передаются серверу методом POST и он помещает их на страницу. При этом передаваемые данные (в примере с блогами — текст комментария) включаются в тело запроса. Аналогично с помощью метода POST обычно загружаются файлы на сервер.

PUT применяется для загрузки содержимого запроса на указанный в запросе URI. Если по заданному URI не существовало ресурса, то сервер создаёт его и возвращает статус 201 (Created). Если же был изменён ресурс, то сервер возвращает 200 (Ok) или 204 (No Content). Сервер не должен игнорировать некорректные заголовки Content-*, передаваемые клиентом вместе с сообщением. Если какой-то из этих заголовков не может быть распознан или не допустим при текущих условиях, то необходимо вернуть код ошибки 501 (Not Implemented).

DELETE удаляет указанный ресурс.

URI (Uniform Resource Identifier, унифицированный идентификатор ресурса) — путь до конкретного ресурса (например, документа), над которым необходимо осуществить операцию (например, в случае использования метода GET подразумевается получение ресурса). Некоторые запросы могут не относиться к какому-либо ресурсу, в этом случае вместо URI в стартовую строку может быть добавлена звёздочка (астериск, символ «*»). Например, это может быть запрос, который относится к самому веб-серверу, а не какому-либо конкретному ресурсу.

Код состояния информирует клиента о результатах выполнения запроса и определяет его дальнейшее поведение. Набор кодов состояния является стандартом, и все они описаны в соответствующих документах RFC.

Каждый код представляется целым трехзначным числом. Первая цифра указывает на класс состояния, последующие - порядковый номер состояния (рис 1.). За кодом ответа обычно следует краткое описание на английском языке.

1xx Informational (Информационный). В этот класс выделены коды, информирующие о процессе передачи. В HTTP/1.0 сообщения с такими кодами должны игнорироваться. В HTTP/1.1 клиент должен быть готов принять этот класс сообщений как обычный ответ, но ничего отправлять серверу не нужно. Сами сообщения от сервера содержат только стартовую строку ответа и, если требуется, несколько специфичных для ответа полей заголовка.Прокси-сервера подобные сообщения должны отправлять дальше от сервера к клиенту.

2xx Success (Успешно). Сообщения данного класса информируют о случаях успешного принятия и обработки запроса клиента. В зависимости от статуса сервер может ещё передать заголовки и тело сообщения.

3xx Redirection (Перенаправление). Коды статуса класса 3xx сообщают клиенту, что для успешного выполнения операции нужно произвести следующий запрос к другому URI. В большинстве случаев новый адрес указывается в поле Location заголовка. Клиент в этом случае должен, как правило, произвести автоматический переход (жарг. «редеет»).

4xx Client Error (Ошибка клиента). Класс кодов 4xx предназначен для указания ошибок со стороны клиента. При использовании всех методов,

кроме HEAD, сервер должен вернуть в теле сообщения гипертекстовое пояснение для пользователя.

5xx Server Error (Ошибка сервера). Коды 5xx выделены под случаи неудачного выполнения операции по вине сервера. Для всех ситуаций, кроме использования метода HEAD, сервер должен включать в тело сообщения объяснение, которое клиент отобразит пользователю.

Заголовок HTTP (*HTTP Header*) — это строка в HTTP-сообщении, содержащая разделённую двоеточием пару вида «параметр-значение». Формат заголовка соответствует общему формату заголовков текстовых сетевых сообщений ARPA (RFC 822). Как правило, браузер и веб-сервер включают в сообщения более чем по одному заголовку. Заголовки должны отправляться раньше тела сообщения и отделяться от него хотя бы одной пустой строкой (CRLF).

Тело HTTP сообщения (message-body), если оно присутствует, используется для передачи сущности, связанной с запросом или ответом. *Тело сообщения* (message-body) отличается от *тела сущности*(entity-body) только в том случае, когда при передаче применяется кодирование, указанное в заголовке Transfer-Encoding. В остальных случаях тело сообщения идентично телу сущности.

Заголовок Transfer-Encoding должен отправляться для указания любого кодирования передачи, примененного приложением в целях гарантирования безопасной и правильной передачи сообщения. Transfer-Encoding - это свойство сообщения, а не сущности, и оно может быть добавлено или удалено любым приложением в цепочке запросов/ответов.

Присутствие тела сообщения в запросе отмечается добавлением к заголовкам запроса поля заголовка Content-Length или Transfer-Encoding.

Тело сообщения (message-body) может быть добавлено в запрос только когда метод запроса допускает тело объекта (entity-body).

2.3 Использование OpenSSL

Симметричные алгоритмы наряду со всеми своими достоинствами имеют большой недостаток: потеря секретного ключа грозит потерей всех данных, которые с помощью этого ключа зашифрованы (имеется в виду не физическая потеря, а потеря конфиденциальности).

Использование симметричной схемы шифрования требует передачи одного и того же ключа двум и более сторонам по независимому и надежному каналу. Этот канал должен исключать возможность утечки информации, обеспечивать полный контроль над процессом передачи ключа и гарантировать его доставку получателю. Когда количество абонентов невелико и расположены они недалеко друг от друга (например, в пределах одного города), гарантированно доставить ключ абоненту достаточно просто. А если абоненты находятся в разных городах или их количество очень велико? В таком случае организация надежного канала доставки ключевых данных - задача весьма непростая, требующая значительных затрат.

Для решения в первую очередь задачи распределения ключей была выдвинута концепция двухключевой (или ассиметричной) криптографии.

Для шифрования и дешифрования применяются различные ключи. Для шифрования информации, предназначенной конкретному получателю, используют уникальный открытый ключ получателя-адресата. Соответственно для дешифрования получатель использует парный, создаваемый одновременно с открытым, секретный ключ. Для передачи открытого ключа от получателя к отправителю секретный канал не нужен.

Алгоритм RSA. Теория

Криптосистема RSA, предложенная в 1977 году Ривестом (R. Rivest), Шамиром (A. Shamir) и Адлеманом (L. Adleman), предназначена для шифрования и цифровой подписи. В настоящее время RSA является наиболее распространенной криптосистемой - стандартом де-факто для многих криптографических приложений. Криптосистема RSA широко применяется в составе различных стандартов и протоколов Интернета, включая PEM, S/MIME, PEM-MIME, S-HTTP и SSL.

Криптографическая стойкость алгоритма RSA основана на трудоемкости разложения на множители (факторизации) больших чисел. Термин "большие" означает, что число содержит 100~200 и более двоичных разрядов. Открытый и секретный ключи являются функциями двух больших простых чисел. Рассмотрим на примере, как выполняется генерация ключей алгоритма RSA, но вместо больших чисел для простоты изложения будем использовать маленькие десятичные.

Для генерации парных ключей используются два случайных простых числа, p и q. Вычисляется произведение этих чисел n и значение функции Эйлера от числа n по формуле:

$$\varphi(n)=(p-1)(q-1) \quad [1]$$

Далее выбирается ключ шифрования e такой, что e и значение функции Эйлера $\varphi(n)$ являются взаимно простыми числами, т.е. числами, не имеющими общих делителей, кроме единицы (единицу еще называют тривиальным делителем). Теперь необходимо найти значение ключа дешифрования d такое, чтобы выполнялось равенство:

$$ed = 1(\mathrm{mod}\ \varphi(n)) \quad [2]$$

или

$d = e-1(\mod \varphi(n))$ [3]

Уравнение [2] означает, что остаток от деления произведения чисел e и d на значение функции Эйлера $\varphi(n)$ должен быть равен 1.

Условие [2] выполняется только в том случае, если e и $\varphi(n)$ являются взаимно простыми числами. Число d называется взаимно обратным к e по модулю $\varphi(n)$. Уравнение [2] эквивалентно обнаружению таких d и v, что:

$ed + \varphi(n)v = 1$ [4]

Поиск обратного значения числа по модулю выполняется при помощи алгоритма Эвклида. Этот алгоритм позволяет найти наибольший общий делитель (НОД) двух чисел.

Рассмотрим пример. Пусть у нас имеются два простых числа: p=13 и q=17. Найдем произведение этих чисел:

$n = 13 * 17 = 221$

и значение функции Эйлера от числа n=221:

$\varphi(n)=(p-1)(q-1)=(13-1)(17-1)=192$

Теперь выберем такое число e, чтобы оно было взаимно простым с $\varphi(n)$. Таким числом является, например, e=7. Далее надо найти обратное значение числа e, чтобы выполнялось уравнение [2]. Для этого с помощью алгоритма Эвклида ищем значения d и v, удовлетворяющие соотношению [4]. Суть алгоритма сводится к проведению последовательности операций деления с остатком. В соответствии с алгоритмом находим частное и остаток от деления $\varphi(n)$ на e:

$192 = 7 * 27 + 3$

Частное равно 27, остаток - 3. Теперь последовательно делим делитель на остаток (т. е. 7 на 3 в данном случае) до тех пор, пока в остатке не получим единицу:

$7 = 3 * 2 + 1$

А теперь распишем процесс получения остатка в обратном порядке:

$1 = 7 - 3 * 2 = 7 - (192 - 7 * 27) * 2 = 7 - (192 * 2 - 7 * 2 * 27)$

Раскроем скобки:

$1 = 7 + 7 * 54 - 192 * 2 = 7 * 55 - 192 * 2$

В итоге получаем искомые числа d=55 и v=2. Числа d=55 и e=7 являются взаимно обратными по модулю 192, что подтверждает равенство:

$7 * 55 = 1(\text{mod } 192)$

Учитывая, что e=7 - это наш ключ шифрования, то число d=55 будет ключом дешифрования.

Теперь разберемся, как выполняются операции шифрования и дешифрования информации по алгоритму RSA.

Для шифрования исходное сообщение необходимо представить в виде последовательности чисел, содержащихся в интервале от 0 до n. Для примера, разобьем аббревиатуру ABC на числа в интервале (0,221). Для этого достаточно каждый символ записать в десятичном представлении:

A=41h=65, B=42h=66, C=43h=67

Шифрование сводится к вычислению:

$Ci = Mi\ e(\text{mod } n)$

Здесь Mi - это i-й блок сообщения, Ci - результат криптопреобразования. Выражаясь простым языком, мы должны возвести значение Mi в степень e и найти остаток от деления на n.

Зашифруем нашу последовательность (65,66,67), зная, что e=7 и n=221:

C1 = 657(mod 221) = 91

C2 = 667(mod 221) = 144

C3 = 677(mod 221) = 50

В зашифрованном виде наша последовательность будет выглядеть как (91,144,50).

Для дешифрования необходимо выполнить следующую операцию:

Mi = Ci d(mod n)

Дешифруем последовательность (91,144,50) при d=55 и n=221:

M1 = 9155(mod 221) = 65

M2 = 14455(mod 221) = 66

M3 = 5055(mod 221) = 67

Таким образом, исходная последовательность восстановлена.

Шифрование RSA выполняется намного эффективнее, если правильно выбрать значение e. Чаще всего используются 3, 17 и 65537. Стандарт X.509 рекомендует 65537, PEM - 3, PKCS#1 - 3 или 65537.

Ознакомившись с теорией, приступим к рассмотрению средств, предоставляемых библиотекой для защиты информации по RSA алгоритму.

Функции библиотеки для защиты информации по RSA-алгоритму

Прежде чем изучить вышеозначенные функции, приостановимся на минуту и подумаем - если мы оперируем с числами, разрядность которых составляет ~200 битов, то мы должны их где-то хранить. А ведь их надо не только хранить, но и проводить над ними различные математические операции, такие как умножение, деление, возведение в степень и т. п. Очевидно, что стандартные типы языка программирования Си, например, long или double long, и прямое использование функций стандартной библиотеки этого языка, таких как "+", "*" и т. п. для этих целей совершенно непригодны. Поэтому библиотека OpenSSL содержит ряд специальных функций для работы с большими числами, разрядность которых превышает разрядность адресной шины и регистров процессора. Для хранения этих чисел используется динамическая память. Базовым объектом библиотеки для работы с такими числами является объект типа BIGNUM. Этот тип определен в файле openssl/bn.h:

```
#define BN_ULONG unsigned char
typedef struct bignum_st
{
    /* Pointer to an array of 'BN_BITS2' bit chunks. */
    BN_ULONG *d;
    int top; /* Index of last used d +1. */
    /* The next are internal book keeping for bn_expand. */
    int dmax; /* Size of the d array. */
    int neg; /* one if the number is negative */
    int flags;

} BIGNUM;
```

Приведем краткий перечень функций библиотеки для работы с большими числами:

n BIGNUM * BN_new(void) - создает объект типа BIGNUM и возвращает указатель на него;

n int BN_add(BIGNUM *r, const BIGNUM *a, const BIGNUM *b) - суммирует числа a и b, результат помещает в r (r=a+b);

n int BN_sub(BIGNUM *r, const BIGNUM *a, const BIGNUM *b) - выполняет операцию вычитания числа b из числа a, результат сохраняется в r (r=a-b);

n int BN_mul(BIGNUM *r, BIGNUM *a, BIGNUM *b, BN_CTX *ctx) - умножает число a на число b и сохраняет результат в r (r=a*b). Последний параметр BN_CTX *ctx используется для хранения промежуточных результатов вычисления (a BN_CTX is a structure that holds BIGNUM temporary variables used by library functions);

n int BN_div(BIGNUM *dv, BIGNUM *rem, const BIGNUM *a, const BIGNUM *d, BN_CTX *ctx) - делит число a на b, частное сохраняется в dv, остаток - в rem (dv=a/b, rem=a%b).

Подробную информацию по функциям типа BN_* смотрите на странице руководства man bn.

Все ассимметричные алгоритмы, реализованные в библиотеке, такие как RSA, DSA, Diffie-Hellman, используют эти функции.

Генерация ключей алгоритма RSA

Как в и случае симметричных алгоритмов, для криптографической защиты информации по алгоритму RSA необходимо вначале сгенерировать открытый и секретный ключи. Генерацию ключей алгоритма RSA выполняет функция RSA_generate_key следующего вида:

RSA *RSA_generate_key(int num, unsigned long e, void (*callback)(int,int,void *), void *cb_arg);

Параметры функции:

n int num - размер ключа в битах;

n unsigned long e - это то самое число e, с которым мы познакомились в пункте 2, когда рассматривали теоретические основы алгоритма RSA. Этот параметр обычно принимает значения 3, 17 или 65537.

Два последних параметра, указатели на функцию (*callback) и void *cb_arg, служат для предоставления обратной связи (feedback) с процессом генерации ключевой информации. Это, как правило, индикация хода выполнения операции генерирования ключей. Например, если мы выполним команду:

openssl genrsa -out outfile 2048

то увидим, как на экране начнут появляться символы "." и "+", отображающие процесс формирования ключей. Найдем в файле apps/genrsa.c исходных текстов библиотеки вызов функции RSA_generate_key:

rsa=RSA_generate_key(num,f4,genrsa_cb,bio_err);

Третий параметр - функция genrsa_cb - находится в этом же файле и имеет следующий вид:

```
static void MS_CALLBACK genrsa_cb(int p, int n, void *arg)
{
    char c='*';

    if (p == 0) c='.';
    if (p == 1) c='+';
    if (p == 2) c='*';
    if (p == 3) c='\n';
    BIO_write((BIO *)arg,&c,1);
    (void)BIO_flush((BIO *)arg);
#ifdef LINT
    p=n;
#endif

}
```

Как видно из текста этой функции, именно она выводит на экран символы "." и "+", отображающие ход операции генерирования ключей.

Результаты работы функции RSA_generate_key в виде открытого и закрытого ключа сохраняются в структуре типа RSA (см. include/openssl/rsa.h). Эти

ключи необходимо извлечь и записать в файлы для дальнейшей работы с ними. Делается это при помощи следующих двух функций:

int PEM_write_RSAPublicKey(FILE *fp, RSA *x);

int PEM_write_RSAPrivateKey(FILE *fp, RSA *x, const EVP_CIPHER *enc, unsigned char *kstr, int klen, pem_password_cb *cb, void *u);

Первая функция записывает в файл fp открытый ключ, который находится в структуре типа RSA, на которую указывает параметр x. Вторая функция изымает секретный ключ из структуры x и записывает его в файл fp. Как правило, секретный ключ зашифровывают, что способствует повышению его защищенности. Выбор алгоритма шифрования выполняется с помощью параметра const EVP_CIPHER *enc (контекст алгоритма шифрования, см. первую часть статьи). Но для шифрования необходимо задать ключевую фразу (пароль), и сделать это можно несколькими способами.

Можно указать в параметре cb (сокращение от callback) адрес функции, которая будет запрашивать пароль. Если указатель kstr не будет равен NULL, то в качестве пароля будут использованы первые klen символов из массива, на который указывает kstr, при этом параметр cb игнорируется. Если cb == NULL, а параметр u не равен NULL, то u интерпретируется как строка, заканчивающаяся нулем, и эта строка используется как пароль. Также можно все параметры (kstr, cb, u) установить в NULL, и библиотека запросит у нас парольную фразу самостоятельно, используя свои внутренние механизмы.

Обобщим все вышесказанное в виде фрагмента программы, генерирующей ключи по алгоритму RSA, при этом секретный ключ шифруется по алгоритму Blowfish с обратной связью по выходу. Длина ключа - 2048 бит. В целях экономии места код, выполняющий обработку ошибок, пропущен.

Листинг 1. Генерация ключей алгоритма RSA

```
#include <stdio.h>
```

```c
#include <openssl/rsa.h>
#include <openssl/pem.h>

/* Имена ключевых файлов */
#define PRIVAT "./privat.key"
#define PUBLIC "./public.key"

void main()
{
    /* указатель на структуру для хранения ключей */
    RSA * rsa = NULL;
    unsigned long bits = 2048; /* длина ключа в битах */
    FILE *priv_key_file = NULL, *pub_key_file = NULL;
    /* контекст алгоритма шифрования */
    const EVP_CIPHER *cipher = NULL;

    priv_key_file = fopen(PRIVAT, "wb");
    pub_key_file = fopen(PUBLIC, "wb");

    /* Генерируем ключи */
    rsa = RSA_generate_key(bits, RSA_F4, NULL, NULL);

    /* Формируем контекст алгоритма шифрования */
    OpenSSL_add_all_ciphers();
    cipher = EVP_get_cipherbyname("bf-ofb");

    /* Получаем из структуры rsa открытый и секретный ключи и сохраняем в файлах.
    * Секретный ключ шифруем с помощью парольной фразы "hello"
    */
    PEM_write_RSAPrivateKey(priv_key_file, rsa, cipher,
        NULL, 0, NULL, "hello");
    PEM_write_RSAPublicKey(pub_key_file, rsa);

    /* Освобождаем память, выделенную под структуру rsa */
    RSA_free(rsa);
}
```

Если в вызове функции PEM_write_RSAPrivateKey мы вместо "hello" оставим NULL, то библиотека самостоятельно попросит нас ввести парольную фразу для шифрования секретного ключа.

Шифрование и дешифрование по алгоритму RSA

Сформировав ключи, можно приступать к шифрованию данных. Для этого используется функция RSA_public_encrypt, которая имеет следующий прототип:

int RSA_public_encrypt(int flen, unsigned char *from, unsigned char *to, RSA *rsa, int padding);

Эта функция шифрует flen байт из буфера, на который указывает параметр from, используя ключ из структуры RSA * rsa, и помещает результат в буфер

to. Размер этого буфера должен быть равен размеру ключа, который определяется при помощи функции RSA_size(RSA *).

Параметр padding используется для выбора режима выравнивания данных. В большинстве случаев используется значение RSA_PKCS1_PADDING, что соответствует стандарту PKCS#1. При использовании этого режима размер буфера from должен быть не меньше (RSA_size(rsa) - 11), т.е. на 11 байт меньше размера ключа. Размер выходных данных всегда будет кратен длине ключа.

Следующая программа демонстрирует использование функции RSA_public_encrypt для шифрования данных.

Листинг 2. Шифрование данных по алгоритму RSA

```
 #include <openssl/rsa.h>
#include <openssl/pem.h>

void main(int argc, char **argv)
{
    /* структура для хранения открытого ключа */
    RSA * pubKey = NULL;
    unsigned char *ptext, *ctext;
    FILE * pub_key_file = NULL;

    /* Открываем входной и создаем выходной файлы */
    int inf = open(argv[1], O_RDWR);
    int outf = open("./rsa.file",
        O_CREAT|O_TRUNC|O_RDWR, 0600);

    /* Считываем открытый ключ */
    pub_key_file = fopen(PUBLIC, "rb");
    pubKey = PEM_read_RSAPublicKey(pub_key_file, NULL, NULL, NULL);

    /* Определяем длину ключа */
    int key_size = RSA_size(pubKey);
    ptext = malloc(key_size);
    ctext = malloc(key_size);

    /* Шифруем содержимое входного файла */
    while(1) {
        inlen = read(inf, ptext, key_size - 11);
        if(inlen <= 0) break;

        outlen = RSA_public_encrypt(inlen, ptext, ctext, pubKey, RSA_PKCS1_PADDING);
        if(outlen != RSA_size(pubKey)) exit(-1);

        write(outf, ctext, outlen);
    }

}
```

Дешифрование данных выполняет функция RSA_private_decrypt:

int RSA_private_decrypt(int flen, unsigned char *from, unsigned char *to, RSA *rsa, int padding);

Эта функция расшифровывает flen байт из буфера from, используя ключ rsa, и записывает результаты в буфер to.

Следующий фрагмент программы демонстрирует использование функции RSA_private_decrypt для дешифрования данных.

Листинг 3. Дешифрование файла, зашифрованного по RSA-алгоритму

```
#include <openssl/rsa.h>
#include <openssl/pem.h>

void main(int argc, char **argv)
{
    RSA *privKey = NULL;
    FILE *priv_key_file;
    unsigned char *ptext, *ctext;

    /* Открываем входной и создаем выходной файл */
    inf = open(argv[1], O_RDWR);
    outf = open("./test.rsa", O_CREAT|O_TRUNC|O_RDWR, 0600);

    /* Открываем ключевой файл и считываем секретный ключ */
    priv_key_file = fopen(PRIVAT, "rb");
    privKey = PEM_read_RSAPrivateKey(priv_key_file, NULL, NULL, NULL);

    /* Определяем размер ключа */
    key_size = RSA_size(privKey);
    ctext = malloc(key_size);
    ptext = malloc(key_size);

    /* Дешифруем файл */
    while(1) {
        inlen = read(inf, ctext, key_size);
        if(inlen <= 0) break;

        outlen = RSA_private_decrypt(inlen, ctext, ptext, privKey, RSA_PKCS1_PADDING);
        if(outlen < 0) exit(0);

        write(outf, ptext, outlen);
    }

}
```

Работоспособность программ была проверена для ОС Linux Slackware 10.2, библиотека OpenSSL версии 0.9.7c

Итак, вы разобрались с теоретическими основами алгоритма RSA и научились использовать некоторые функции библиотеки для защиты информации по этому алгоритму. Я говорю "некоторые", потому что этих функций в библиотеке очень много, и описать их в одной статье практически невозможно. Поэтому основным источником изучения средств библиотеки были и остаются официальная документация от разработчиков, а также любые исходные тексты программ, использующих эту библиотеку.

2. Проектирование и описание программы

Для создания данной системы необходимо для начала определится с функциями, которые будут выполняться на клиенте и сервере. Отобразим функции на диаграммах прецедентов.

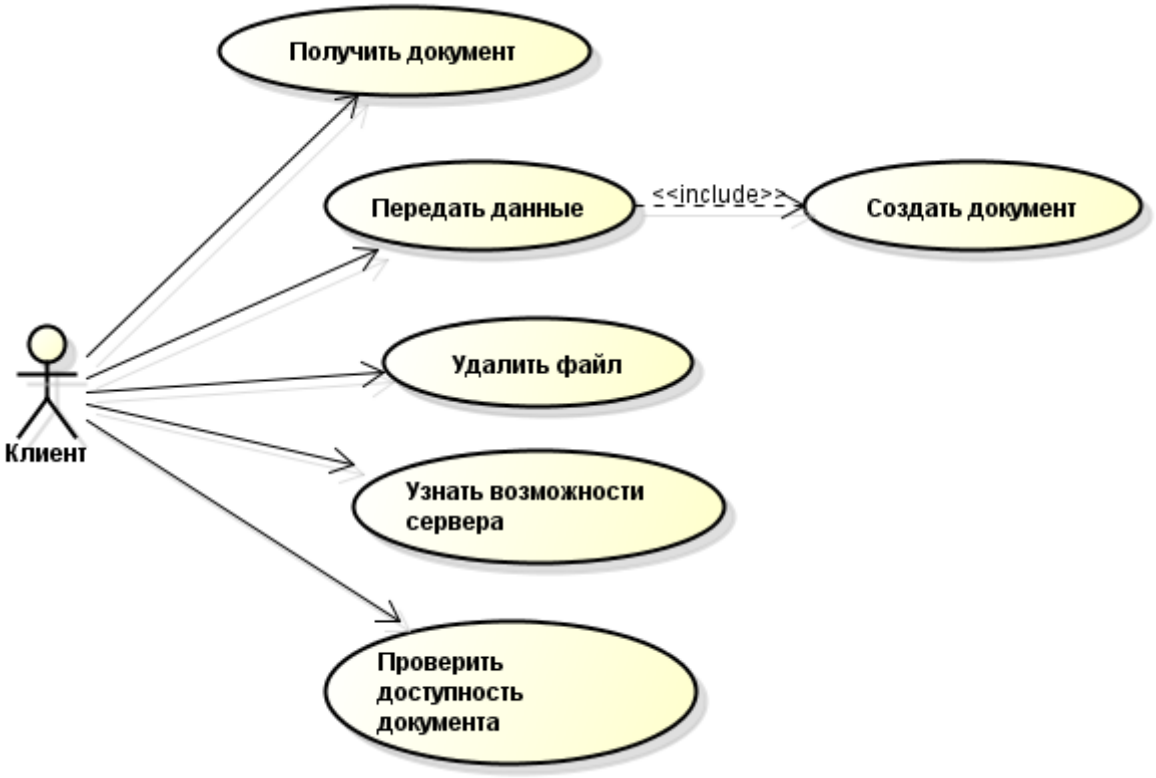


Рисунок 8 – Диаграмма прецедентов серверной программы

На рисунке 8 приведена диаграмма прецедентов серверной программы. Актор «Клиент» обозначает любую клиентскую программу или приложение, которая может подключиться к серверной программе, по протоколу TCP.

Прецедент «Получить документ» представляет собой событие оправки серверу метода GET, в следствии которого сервер должен либо выдать документ по данному URI, либо отправить код ошибки.

Прецедент «Передать данные» представляет собой событие оправки серверу методов POST или PUT. Различие методов состоит в том, что методом POST на сервер передается URI программы, которая должна обработать данные передаваемые клиентом на сервер. Метод PUT же сохраняет данные в теле запроса по передаваемому URI. В нашей реализации

сервер не ставится задача создать интерфейс запуска программ обработки клиентских данных (CGI или PHP), то есть для метода PUT существует прецедент «Создать документ», а метод POST выводить отправленные параметры на консоль.

Прецедент «Удалить документ» представляет собой событие оправки серверу метода DELETE, в следствии которого сервер должен либо удалить документ по данному URI, либо отправить код ошибки.

Прецедент «Узнать возможности сервера» представляет собой событие оправки серверу метода OPTIONS, в следствии которого сервер должен вернуть список доступных методов протокола HTTP на данном сервере.

Прецедент «Получить документ» представляет собой событие оправки серверу метода HEAD, вследствие которого сервер должен либо выдать успешности операции, либо отправить код ошибки.

Рисунок 9 – Диаграмма прецедентов клиентской программы

На рисунке 9 приведены функции клиентской программы. Эти методы интуитивно понятны.

Для более подробного описания системы приведем диаграммы активности для программы сервера.

Далее приводится алгоритм обработки поступившего запроса HTTP. Сначала производится загрузка заголовка запроса, происходит распознавание версии протокола, метода запроса и URI. Если в заголовке имеются ошибки, клиент получает сообщение о ошибке. Если ошибок нет, то сервер отправляет клиенту ответ с кодом 102 (продолжить).

Далее сервер получает параметры HTTP header и выполняет метод запроса.

Важной частью алгоритма анализа заголовка HTTP запроса является определение правильности параметров запроса, отделения параметров GET. После отделения параметров происходит парсинг строки с параметров и сохранения их в памяти в виде пар имя-значение.

В конце обработки запроса HTTP необходимо совершить действие согласно полученному методу. Для метода GET необходимо создать соответствующий запросу заголовок, в данном заголовке обязательно должен находиться параметр Content-Length, определяющий количество байт в передаваемом файле на клиент. Далее вызывается функция записи файла в TCP сокет. Метод HEAD состоит только лишь из заголовка, тело в данном случае не посылается, а заголовок идентичен заголовку в методе GET.

Метод POST может выполнять две функции в зависимости от значения параметра в запросе клиента. Либо он создает новый файл, и записывает в него тело полученное от клиента, либо в его теле содержатся параметры и он возвращает файл указанный в URI. Процесс создания нового файла аналогичен методу PUT. Метод DELETE удаляет с сервера файл с именем хранимым в URI.

Не маловажное значение в отправке ответа играет параметр «Content-type». Данный параметр задает какой тип файла содержится внутри ответа сервера. Например если Content-Type имеет значение «image/jpeg», то браузер распознает файл как картинку и будет отображать её соответственно.

3.2 Реализация

Язык программирования данной системы выбран C++, и библиотека Qt.

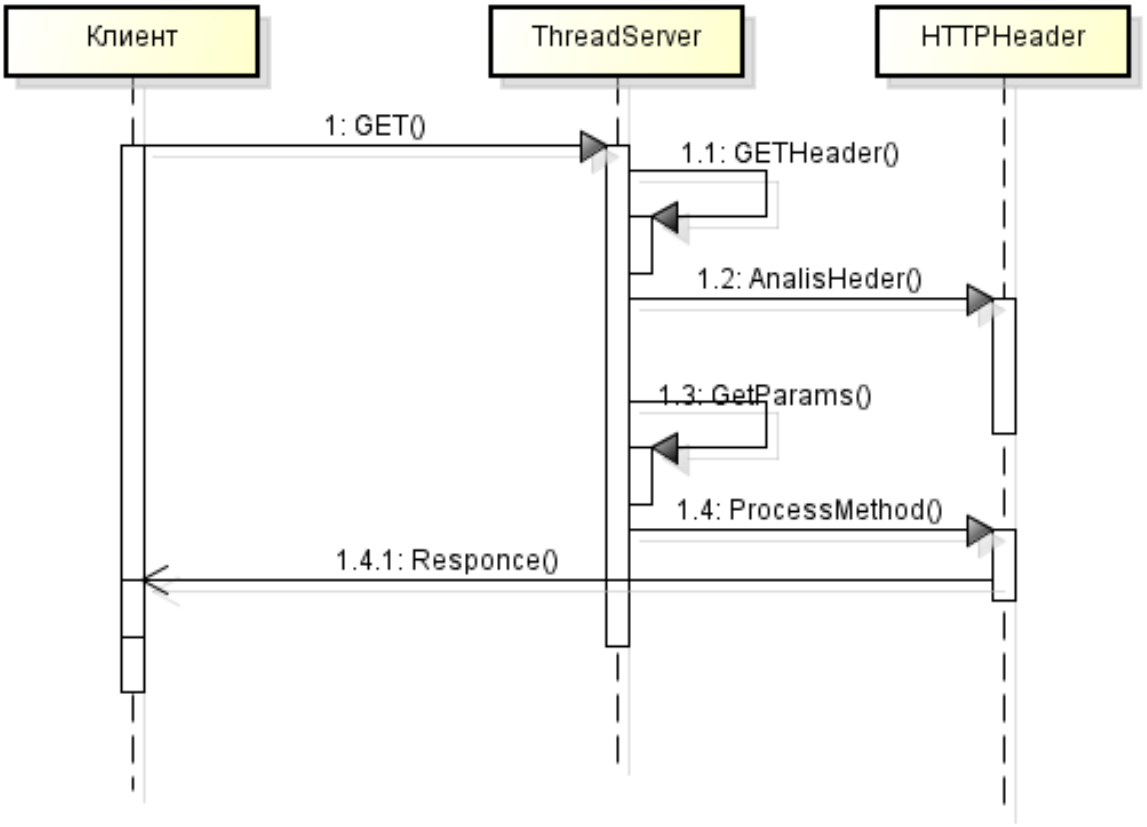


Рисунок 10 – Диаграмма последовательности для общей структуры работы клиент-серверного приложения.

Для передачи информации по каналу связи используются методы Read и Write. Для передачи информации данными методами необходимо создать буферы – байтовые массивы, указать количество передаваемый байт, и начальное смещение.

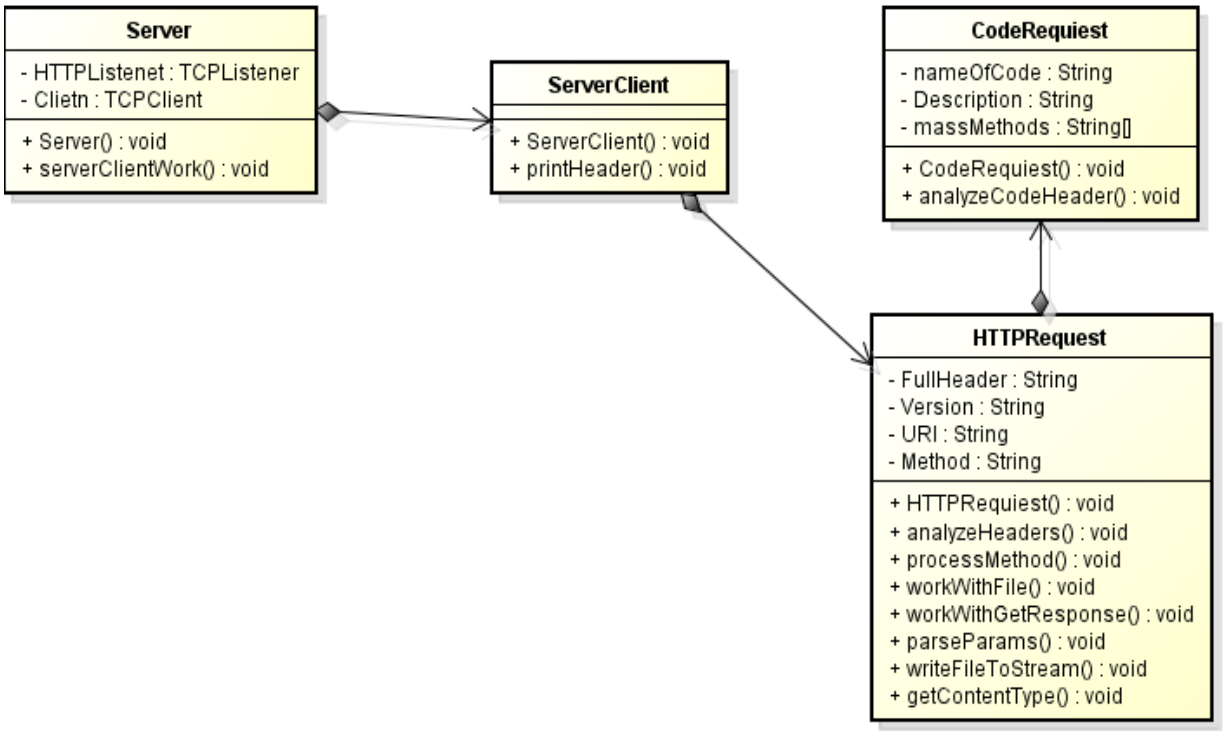


Рисунок 11 – Диаграмма классов сервера

Структура сервера состоит из 4 классов: Server, ServerClient, CodeRequiest, HTTPRequiest.

Класс Server представляет собой абстрацкию сервера. В своем конструкторе находится цикл создания и подтверждения подключений от клиентов. Для каждого нового подключения создается новый поток и анонимный экземпляр класса ServerClient.

Класс ServerClient представляет собой абстракцию адаптера передачи данных между сервером и клиентом. Все основные действия происходят в конструкторе этого класса. Именно конструктор класса ServerClient использует алгоритм изображенный на рисунке 3. Задача этого класса получать данные и вызывать необходимые методы из основного класса обработки запросов HTTPReguest. Также в данном классе есть метод печати в экран консоли параметров принятого запроса HTTP.

Класс HTTPReguest представляет собой класс, предоставляющий функционал для обработки запросов HTTP, а так же ответов на них клиенту. Метод analyzeHeader, получает на вход строку поступившего запроса, производит его разбор, определяет, если ли ошибки в запросе при помощи класса CodeRequest. Версия протокола, метод, URI и параметры get запроса, если они существуют, сохраняются в поля этого объекта.

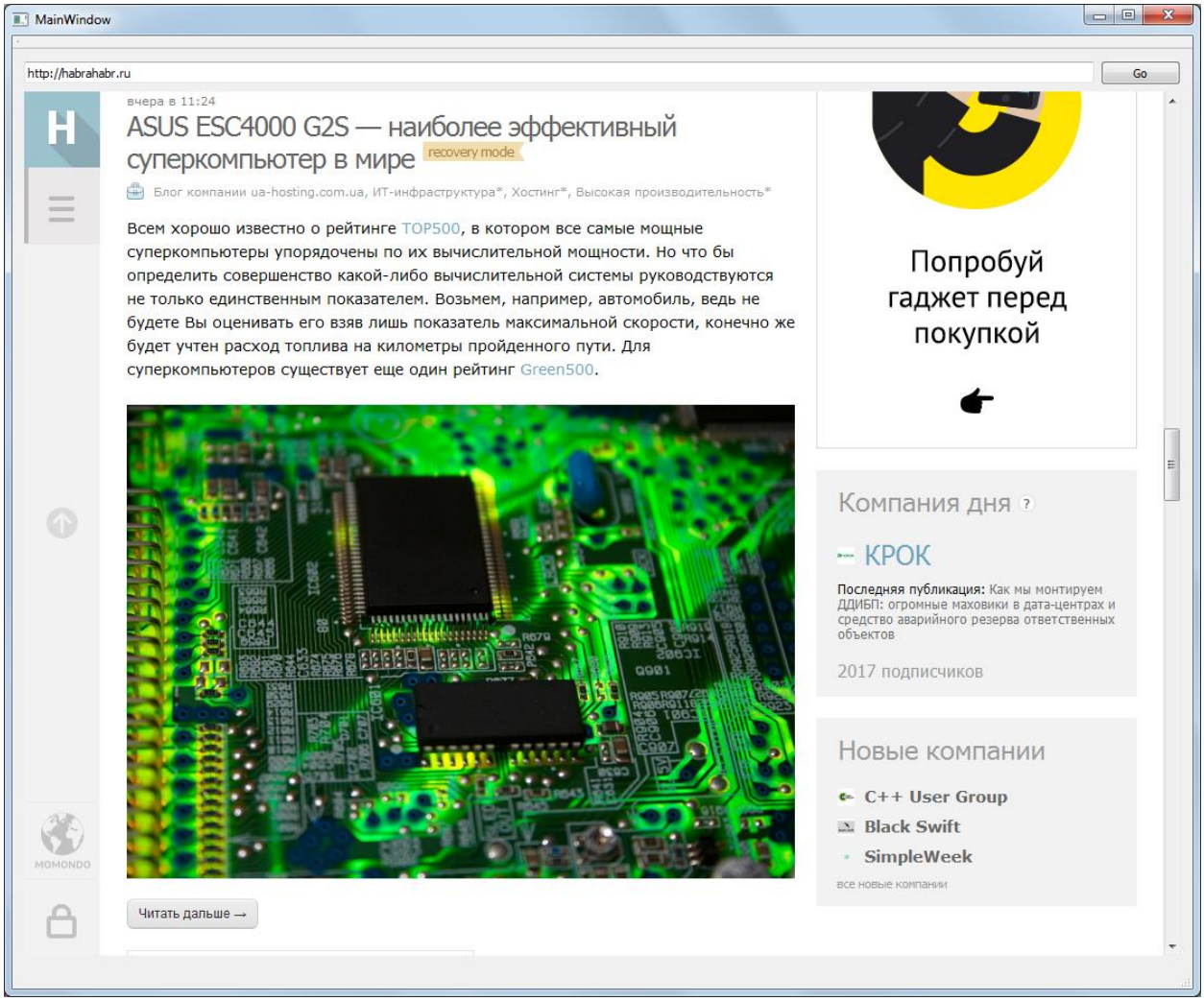
Метод processMethod выполняет специфические действия для каждого метода. Алгоритм данного метода изображен на рисунке 5. Так же в это методе вызываются внутрении методы данного класса: parseParams, writeFileToStream, getContentType, workWithFile, workWithGetRespose.

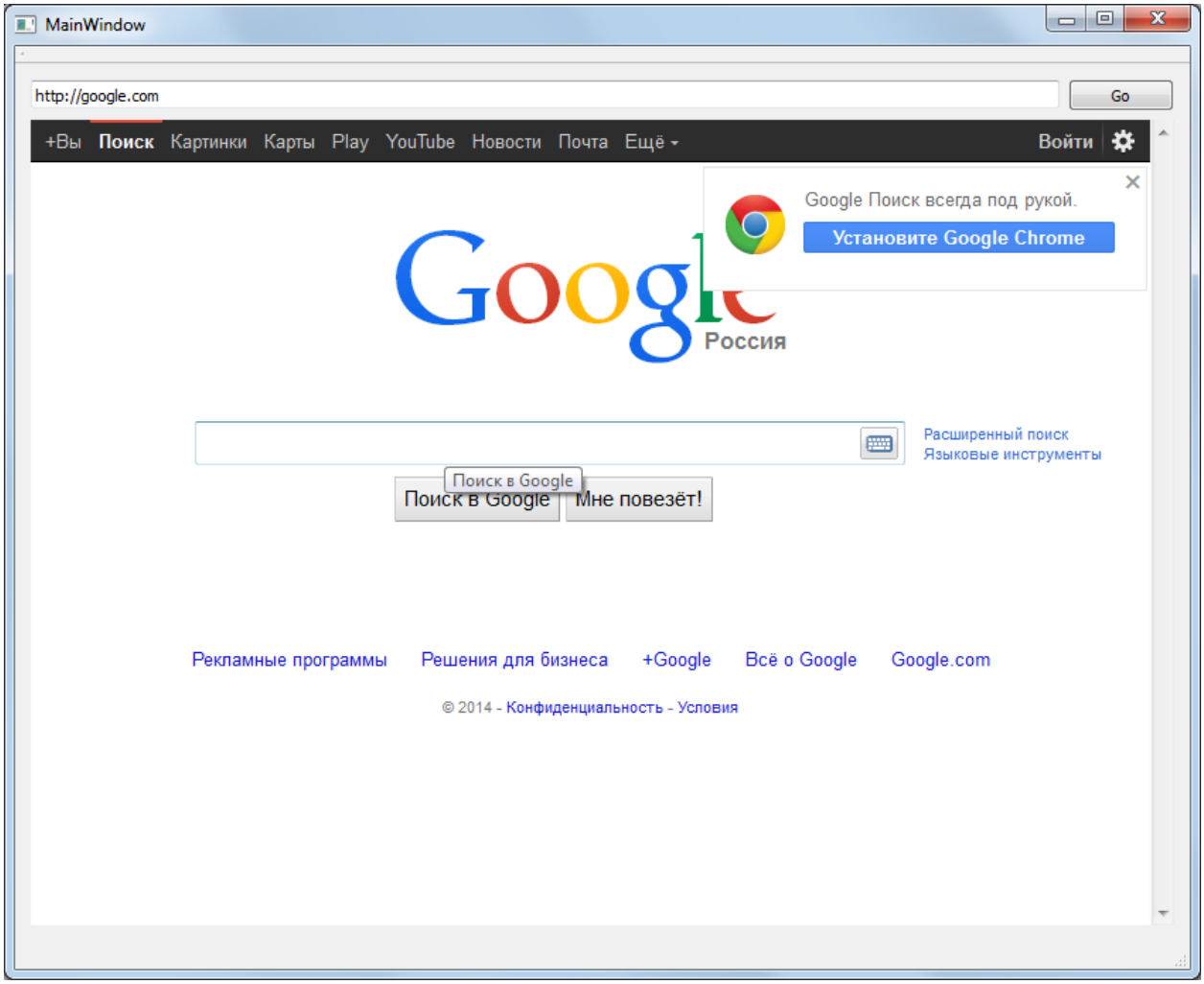
Метод workWithFile – реализует возможность создания или перезаписи файла. Для методов POST и PUT при передачи файлов возможны два случая: файл уже существует, файла нет. При создании файла, сервер отправляет код «201 Content create», если файл перезаписан, то «204 No content».

Алгоритм метода getContentType был описан на рисунке 6. Метод вызывается перед формированием параметров ответа.

Метод parseParams используется для преобразования считанной строки в массив ключ-значения для методов POST и GET.

3.3 Экранные формы




MainWindow
http://google.com
Go
+Вы
Поиск
Картинки
Карты
Play
YouTube
Новости
Почта
Ещё
Войти
Google Поиск всегда под рукой.
Установите Google Chrome
Google
Россия
Расширенный поиск
Языковые инструменты
Поиск в Google
Поиск в Google
Мне повезёт!
Рекламные программы
Решения для бизнеса
+Google
Всё о Google
Google.com
© 2014 - Конфиденциальность - Условия

3. Заключение

В ходе работы над системой были созданы две программы: клиент и сервер. Связь между клиентом и сервером реализуется при помощи протокола TCP. Сервер соответствует стандарту HTTP\1.0 и реализует методы GET, POST, HEAD, PUT, DELETE, OPTIONS. Данные защищены при помощи OpenSSL.

4. Исходный код

Engine.hpp

```
#ifndef ENGINE_HPP
#define ENGINE_HPP
#pragma comment(lib, "WS2_32.lib")
#pragma comment(linker, "/STACK:36777216")

#include <WinSock2.h>
#include <iostream>
#include <string>
#include <cstring>
using namespace std;

#define die(s) { echo(s); return; }
#define dief(s) { echo(s); return false;}

/**
@brief Universal class for working with sockets
*/
class engine_t
{
    string type;

    WSADATA wsaData;
    SOCKET mysock, remsock;
    sockaddr_in sai;
    char buf[2000000];
public:

    /**
    @brief Shows message
    @detailed We can overload this function for another way of log-messaging
    @param s - Message
    */
    void echo(const string &s) { cout << s << endl; }

    /**
    @brief Initialisation
    @param mtype - Application type. It can be: "client" or "server"
    @param ip - ip-address
    @param port - port
    */
    engine_t(const string &mtype, const string &ip, int port)
    {
        type = mtype;

        // Windows sockets initialisation
        if (WSAStartup(MAKEWORD(2, 0), &wsaData))
            die("Can't startup Windows Sockets");
            echo("Windows Sockets started");

        // Creates a socket that is bound to a specific transport service provider
        if ((mysock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)) == INVALID_SOCKET)
            die("Can't create socket");
            echo("Socket Created");

        memset(&sai, 0, sizeof(sockaddr_in));
        sai.sin_family = AF_INET;
        sai.sin_port = htons(port);
        sai.sin_addr.s_addr = type == "server" ? INADDR_ANY : inet_addr(ip.c_str());

        if (type == "server")
        {
```

```
            // Associates a local address with a socket
            if (bind(mysock, (sockaddr*)(&sai), sizeof(sai)) == SOCKET_ERROR)
                die("Bind error");
                echo("Bind OK!");

            // Places a socket in a state in which it is listening for an incoming
connection
            if (listen(mysock, 1) == SOCKET_ERROR)
                die("Listen error");
                echo("Listen OK!");
        }
    }

    /**
    @brief Connects to client/server for chatting
    */
    bool connect()
    {
        if (type == "client")
        {
            echo("Connecting...");
            if (::connect(mysock, (sockaddr*)(&sai), sizeof(sai)) == SOCKET_ERROR)
                dief("Connect error!");
                echo("Connection complete!");
        }
        else
        {
            echo("Accepting...");
            if ((remsock = accept(mysock, NULL, NULL)) == INVALID_SOCKET)
                dief("Accept error!");
                echo("Accepted!");
        }
        return true;
    }

    /**
    @brief Sends message
    @param s - message
    */
    bool write(const string &s)
    {
        int len = s.size();
        SOCKET to = type == "server" ? remsock : mysock;
        int f1 = send(to, (char*)(&len), sizeof(len), NULL);
        strcpy(buf, s.c_str());
        int f2 = send(to, buf, len + 1, NULL);
        return f1 == sizeof(int) && f2 == len + 1;
    }

    /**
    @brief Gets message
    @param s - message
    */
    bool read(string &s)
    {
        int len = 0;
        SOCKET from = type == "server" ? remsock : mysock;
        int f1 = recv(from, (char*)(&len), sizeof(len), NULL);
        int f2 = recv(from, buf, len + 1, NULL);
        s = string(buf);
        return f1 == sizeof(int) && f2 == len + 1;
    }

    /**
    @brief Destructor
    @detailed Closes sockets
```

```
    */
    ~engine_t()
    {
        closesocket(mysock);
        WSACleanup();
    }
};

#endif
```

MainWindow.ui

```
<?xml version="1.0" encoding="UTF-8"?>
<ui version="4.0">
 <class>MainWindow</class>
 <widget class="QMainWindow" name="MainWindow">
  <property name="geometry">
   <rect>
    <x>0</x>
    <y>0</y>
    <width>830</width>
    <height>652</height>
   </rect>
  </property>
  <property name="windowTitle">
   <string>MainWindow</string>
  </property>
  <widget class="QWidget" name="centralWidget">
   <layout class="QGridLayout" name="gridLayout">
    <item row="0" column="0">
     <layout class="QHBoxLayout" name="horizontalLayout">
      <item>
       <widget class="QLineEdit" name="lineEdit">
        <property name="text">
         <string>http://google.com</string>
        </property>
       </widget>
      </item>
      <item>
       <widget class="QPushButton" name="goButton">
        <property name="text">
         <string>Go</string>
        </property>
       </widget>
      </item>
     </layout>
    </item>
    <item row="1" column="0">
     <widget class="QWebView" name="webView">
      <property name="url">
       <url>
        <string>about:blank</string>
       </url>
      </property>
     </widget>
    </item>
   </layout>
  </widget>
  <widget class="QMenuBar" name="menuBar">
   <property name="geometry">
    <rect>
     <x>0</x>
     <y>0</y>
     <width>830</width>
```

```
     <height>21</height>
    </rect>
   </property>
  </widget>
  <widget class="QToolBar" name="mainToolBar">
   <attribute name="toolBarArea">
    <enum>TopToolBarArea</enum>
   </attribute>
   <attribute name="toolBarBreak">
    <bool>false</bool>
   </attribute>
  </widget>
  <widget class="QStatusBar" name="statusBar"/>
 </widget>
 <layoutdefault spacing="6" margin="11"/>
 <customwidgets>
  <customwidget>
   <class>QWebView</class>
   <extends>QWidget</extends>
   <header>QtWebKitWidgets/QWebView</header>
  </customwidget>
 </customwidgets>
 <resources/>
 <connections/>
</ui>
```

nsv_client_qt.pro

```
#-------------------------------------------------
#
# Project created by QtCreator 2014-12-14T23:34:47
#
#-------------------------------------------------

QT += core gui
QT += webkit
QT += webkit webkitwidgets

greaterThan(QT_MAJOR_VERSION, 4): QT += widgets

TARGET = nsv_client_qt
TEMPLATE = app


SOURCES += main.cpp\
        MainWindow.cpp

HEADERS  += MainWindow.hpp

FORMS    += MainWindow.ui
```

main.cpp

```
#include "MainWindow.hpp"
#include <QApplication>

int main(int argc, char *argv[])
```

```
{
    QApplication a(argc, argv);
    MainWindow w;
    w.show();

    return a.exec();
}
```

MainWindow.hpp

```
#ifndef MAINWINDOW_HPP
#define MAINWINDOW_HPP

#include <QMainWindow>
#include <QMessageBox>

namespace Ui {
class MainWindow;
}

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();

private:
    Ui::MainWindow *ui;

private slots:
    void onUpdate();
};

#endif // MAINWINDOW_HPP
```

nsv_client.cpp

```
#include <iostream>
#include "../common/engine.hpp"
using namespace std;
int main()
{
    cout << "CLIENT!" << endl;
    engine_t engine("client", "127.0.0.1", 5001);
    engine.connect();
    cout << endl;
    while (true)
    {
        string query, ans;
        getline(cin, query);
        engine.write(query);
        engine.read(ans);
        cout << "server: " << ans << endl << endl;
    }
    cout << "BYE!" << endl;
    return 0;
}
```

5. Литература

1. Marina del Rey, TRANSMISSION CONTROL PROTOCOL [Электронный ресурс]/ Information Sciences Institute University of Southern California .: California, 1981. – Режим доступа: http://rfc2.ru/793.rfc/original (Дата обращения 1.12.2014)
2. J.Postel , RFC 793 — Протокол управления передачей (TCP) [Электронный ресурс] / пер. Н. Малых .: Калифорния, 1981. – Режим доступа: http://rfc2.ru/793.rfc (Дата обращения 1.12.2014)
3. R. Fielding, ПРОТОКОЛ ПЕРЕДАЧИ ГИПЕРТЕКСТА HTTP/1.1 [Электронный ресурс] / пер. А. Симонов.: Калифорния, 1999. – Режим доступа: http://www.lib.ru/WEBMASTER/rfc2068/ (Дата обращения 1.12.2014)
4. А. Черных, Описание протокола HTTP [Электронный ресурс] / А. Черных.: 2007. – Режим доступа http://www.angel07.webservis.ru/internet/http.html (Дата обращения 1.12.2014)
5. Коды ответа сервера [Электронный ресурс] / Bretail . : 2010. – Режим доступа http://www.bertal.ru/help.php (Дата обращения 1.12.2014)