# Возможности протоколов TCP и UDP, преимущества и недостатки. Применимость в приложениях.

## Про транспортные протоколы в целом

Основная задача транспортного уровня заключается в обеспечении логического соединения между процессами, запущенными на различным хостах. Следовательно, при отправке пакета необходимо указать информацию, идентифицирующую процесс-получатель на хосте-получателе, которую будет использовать последний при приемке пакета. Работа, которую должен выполнить хост-отправитель, называется *мультиплексированием* и заключается в сборе фрагментов данных, поступающих на транспортный уровень из разных сокетов, создании сегментов путем присоединения заголовков и передаче сегментов сетевому уровню.
Работу, которую должен выполнить хост-получатель, называется *демультиплексированием* и заключается в доставке данных сегмента нужному сокету.

Службы, которые протокол транспортного уровня может предоставлять приложениям:

- Надежная доставка данных
- Пропускная способность
- Время доставки
- Безопасность

| Приложение | Протокол прикладного уровня | Базовый транспортный протокол |
|---|---|---|
| Электронная почта | SMTP[554] | TCP |
| Удаленный терминальный доступ | Telnet[425] | TCP |
| Всемирная паутина | HTTP[483] | TCP |
| Передача файлов | FTP[427] | TCP |
| Потоковый мультимедийный контент | HTTP (например, YouTube) | TCP |
| IP-телефония | SIP[502], RTP[518] или проприетарное (например, Skype) | UDP или TCP |

**Рис. 2.5.** Популярные Интернет-приложения и используемые ими протоколы прикладного и транспортного уровней

| Приложение | Протокол прикладного уровня | Нижерасположенный транспортный протокол |
|---|---|---|
| Электронная почта | SMTP | TCP |
| Удаленный терминальный доступ | Telnet | TCP |
| Всемирная паутина | HTTP | TCP |
| Передача файлов | FTP | TCP |
| Удаленный файловый сервер | NFS | Обычно UDP |
| Потоковый мультимедийный контент | Обычно проприетарный | UDP или TCP |
| Интернет-телефония | Обычно проприетарный | UDP или TCP |
| Сетевое управление | SNMP | Обычно TCP |
| Протокол маршрутизации | RIP | Обычно TCP |
| Трансляция имен | DNS | Обычно TCP |

**Рис. 3.6.** Популярные Интернет-приложения и используемые ими протоколы транспортного уровня

Комментарий к таблице:
Большая часть DNS-транзакций использует UDP, TCP всегда используется для трансфера зоны (ранее широко распространенный механизм передачи баз данных между DNS-серверами).

## UDP

Протокол UDP изначально задумывался как протокол транспортного уровня, которые работает с минимальными издержками как по времени, так и по объему занимаемой памяти. Следовательно, он

реализует лишь минимум необходимой функциональности, а именно описанные выше операции мультиплексирования и демультиплексирования.

**Структура UPD-сегмента**

- Номера портов отправителя и получателя, каждое из этих полей занимает 16 бит (для выполнения функций мультиплексирования и демультиплексирования)
- Длина сегмента (заголовок + данные) в байтах, это поле занимает 16 бит, следовательно, максимальный размер UPD-дейтаграммы - 64 Кбайт.
- Контрольная сумма для проверки ошибок, это поле также занимает 16 бит.


**Рис. 3.7.** Структура UDP-сегмента

Преимущества:

★ *Более полный и точный контроль приложения за процессом передачи данных*. UPD просто упаковывает данные в UPD-сегмент и отправляет его на сетевой уровень, в то время, как протокол TCP может снижать скорость передачи данных для предотвращения перегрузок в сети, а также пытается повторять передачу сегмента, пока не будет получено подтверждение о его приеме. *При этом для UDP некоторые подобные проверки можно реализовать на уровне приложений*. По вышеописанным причинам UDP лучше подходит для систем реального времени, где зачастую скорость передачи данных и устойчивость к потере части данных не так важны, как отсутствие задержек.

★ *Отсутствует установление соединения*. Следовательно, UDP не вносит дополнительную задержку в процесс передачи. Возможно по этой причине DNS-приложения используют UDP.

- ★ *Не заботится о состоянии соединения*. Не нужно хранить информацию необходимую для надежной передачи данных, которую хранит TCP, вследствие чего серверы при работе по UDP способны поддерживать работу с гораздо большим количеством клиентов, чем серверы при работе по TCP.
- ★ *Небольшой заголовок пакета*. 8 байт против 20 у TCP.

Недостатки:

- Несмотря на широкое применения UDP в мультимедийных приложения, вопрос о его рациональности остается открытым. Представим себе ситуацию, при которой большое количество пользователей просматривают потоковое видео высокого качества, сеть может стать сильно перегруженной из-за чего потери станут настолько велики, что пользователи просто ничего не увидят.

## TCP

TCP - надежный протокол транспортного уровня с установлением соединения. TCP обеспечивает управление перегрузкой сети, обнаружение ошибок и повторную передачу пакетов. TCP-соединение всегда является двухточечным и дуплексным (то есть предоставляет возможность передавать данные в обоих направлениях).
В отличие от UDP TCP использует для идентификации сокета не только порт получателя и порт отправителя, но еще IP-адрес получателя и IP-адрес отправителя.

Преимущества:

- ★ TCP предоставляет службу надежной доставки данных, из-за чего широко применяется в прикладных протоколах, в которых критически важно получить все данные в правильном порядке.
  - SSH, FTP, Telnet: в данных протоколах TCP используется для обмена файлами.
  - SMTP, POP, IMAP: в данных протоколах TCP отвечает за передачу сообщений электронной почты.
  - HTTP/HTTPS: в данных протоколах TCP отвечает за загрузку страниц из интернета.
- ★ TCP предоставляет службу предотвращения перегрузок, что способствует минимизации задержек в сети и позволяет предоставить каждому TCP-соединению примерно равную долю пропускной способности канала.
- ★ TCP предоставляет службу управления потока, а также службу выбора размера TCP-сегмента, что способствуют минимизации

потери сегментов из-за переполнения буфера на стороне получателя или прохождения сегмента через “узкие” каналы связи (с малым MTU).

Недостатки:

- Наличие механизма управления перегрузкой негативно влияет на соединения, которым нужно передавать данные с максимально возможной скоростью.
- TCP нуждается в установлении соединения, выделении буфера и специальных переменных соединения.
- Размер заголовка TCP (обычно 20 байт) > размер заголовка UDP (8 байт).

# Принципы работы системы DNS. Использование DNS для целей обеспечения работы других прикладных протоколов (с примерами)

DNS - Domain Name System
DNS - это, во-первых, распределенная база данных, реализованная с помощью иерархии DNS-серверов, а, во-вторых, это протокол прикладного уровня, позволяющих хостам обращаться к этой базе данных.
Протокол работает поверх UDP (см. здесь), использует порт 53.

**Иерархия серверов DNS**

- На самом верхнем уровне находятся так называемые **13 корневых серверов**. На самом деле, серверов гораздо больше (на момент 2011 года их было 247), 13 - количество организаций, которым эти сервера принадлежат.
- **DNS-серверы верхнего уровня**. Эти серверы отвечают за домены верхнего уровня, такие как com, org, net, edu, а также gov и национальные домены верхнего уровня, такие как uk, fr, ca, jp, ru и т.д.
- **Авторитетные DNS-серверы**. Каждая организация, имеющая публично доступные хосты (веб-серверы или почтовые серверы) в Интернете, должна предоставить также доступные DNS-записи, которые сопоставляют имена этих хостов с IP-адресами и которые находятся на авторитетном DNS-сервере. Организация может

хранить эти записи либо на своем авторитетном DNS-сервере, либо на авторитетном DNS-сервере своего Интернет-провайдера.

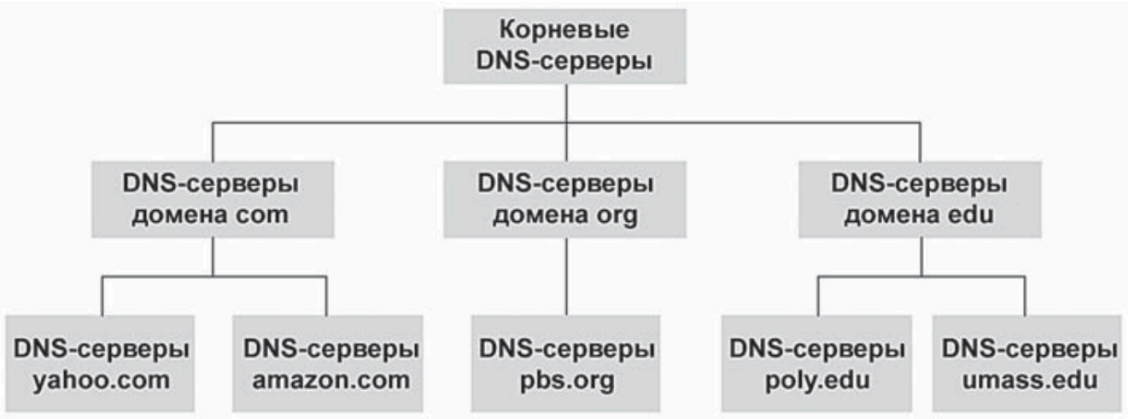



**Рис. 2.19.** Часть иерархической структуры DNS-серверов

**Принцип работы DNS**

Принцип работы DNS заключается в следующем:
когда хосту необходимо выполнить трансляцию доменного имени в IP-адрес, он отправляет запрос локальному DNS-серверу, который выполняет серию запросов к другим DNS-серверам (либо берет запись из своего кэша) с целью нахождения целевого IP-адреса, который в случае успеха возвращается первоначальному хосту.

Рассмотрим пример трансляции доменного имени в IP-адрес (предположим, что ни на одном из DNS-серверов, через которых прошел запрос, подходящая запись в кэше не нашлась, поэтому запрос прошел всю иерархию DNS-серверов).

- Доменное имя запрашивающего хоста: cis.poly.edu.
- Доменное имя, IP-адрес которого запрашивающий хост желает получить: gaia.cs.umass.edu.

1. Хост делает запрос к своему локальному DNS-серверу (который обычно является ближайшим к нему DNS-сервером).
2. Локальный DNS-сервер делает серию итеративных запросов к другим DNS-серверам.
    a. Сначала он делает запрос к корневому DNS-серверу (.), чтобы получить список серверов верхнего уровня, ответственных за домен edu.
    b. Затем локальный DNS-сервер делает запрос к одному из DNS-серверов верхнего уровня, полученных на предыдущем

шаге с целью получения IP-адреса авторитетного DNS-сервера, содержащего IP-адрес целевого хоста.

c. Наконец, локальный DNS-сервер производит запрос к авторитетному DNS-серверу с целью получения IP-адреса хоста gaia.cs.umass.edu.

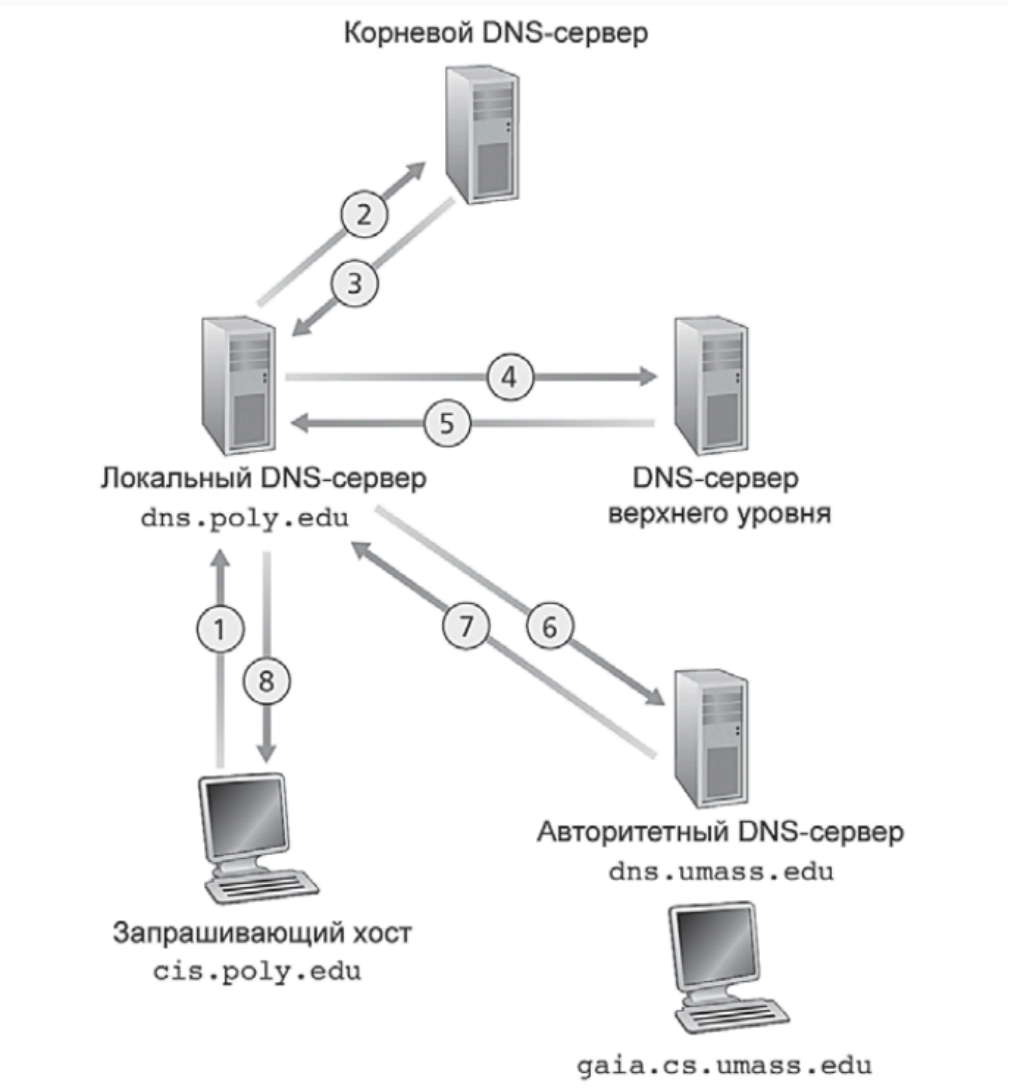



**Рис. 2.21.** Взаимодействие различных DNS-серверов

Несколько важных замечаний по поводу предыдущего примера:

- В текстовом описании данного примера описаны только DNS-запросы, но необходимо понимать, что всего по сети было переслано 8 DNS-сообщений, ведь на каждый DNS-запрос был DNS-ответ (см. рисунок 2.21). Дабы сузить такое количество DNS-сообщений активно применяется кэширование DNS-записей на разных уровнях иерархии. То есть в предыдущем примере вполне возможна ситуация, когда незадолго до рассматриваемого запроса уже производился запрос к хосту gaia.cs.umass.edu. Тогда с

высокой долей вероятности DNS-запись о хосте  gaia.cs.umass.edu сохранилась в кэше локального DNS-сервера. В таком случае запросы a.-c. не потребовались бы и резолвинг доменного имени gaia.cs.umass.edu ограничелся одним запросом к локальному серверу и одним ответом от него.
- В данном примере присутствовал лишь один промежуточный DNS-сервер, на самом деле, их конечно же может быть несколько.

Помимо резолвинга доменных имен DNS-сервера также имеют еще одну крайне полезную службу: *службу распределения нагрузки между серверами*.
Для дублированных (реплицированных) веб-серверов набор IP-адресов привязан к одному каноническому имени хоста. Этот набор содержится в базе данных сервера DNS. Когда клиент делает запрос DNS-серверу по имени, привязанному к набору адресов, сервер в ответ выдает весь набор IP-адресов, но с каждым ответом делает ротацию их порядка. Так как клиент обычно отправляет запрос к первому из списка адресов, то такая DNS-ротация помогает распределять трафик среди реплицированных серверов.

**Ресурсные записи**

Ресурсный записи - это, собственного говоря, и есть содержимое DNS-сервера. Ресурсная запись представляет собой набор из четырех полей: (имя, значение, тип, время жизни).
*Время жизни* определяет, когда ресурсная запись должна быть удалена из кэша. Обычно прослеживается тенденция к возрастанию времени жизни при прохождении вверх по иерархии (от авторитетных серверов к корневым).

Типы ресурсных записей:

| **Тип** | **Имя** | **Значение** | **Пример (без времени жизни)** |
|---|---|---|---|
| A (Address) | имя хоста | IPv4-адрес хоста | (relay1.bar.foo.com, 145.37.93.126, A) |
| AAAA | имя хоста | IPv6-адрес хоста | (relay1.bar.foo.com, ABCD:1234:4321:5555:6666:FFFF:4C5D:4545, AAAA) |

| | | | |
|---|---|---|---|
| NS (Name Server) | имя домена | имя авторитетного DNS-сервера, отвечающего за получение IP-адресов соответствующего домена | (foo.com, dns.foo.com, NS) |
| CNAME (Canonical Name) | псевдоним | каноническое имя для данного псевдонима | (foo.com, relay1.bar.foo.com, CNAME) |
| MX (Mail eXchanger) | псевдоним для почтового сервера | каноническое имя для почтового сервера | (foo.com, mail.bar.foo.com, MX) |

Несколько замечаний по поводу предыдущей таблицы:

- Если DNS-сервер является авторитетным для какого-то определенного имени хоста, то он содержит запись типа A для этого имени хоста (он может содержать запись типа A в своем кэше, даже если не является авторитетным). Если сервер не является авторитетным для имени хоста, тогда он будет содержать запись типа NS для домена, который включает это имя хоста; а также запись типа A, в которой находится IP-адрес DNS-сервера, содержащегося в поле Значение записи типа NS.
- Если в поле Имя указана “@”, то эта запись относится к той доменной зоне, в таблице которой она указана. То есть, если на сервере, соответствующем доменной зоне .google.com указана запись

  | | | |
  |---|---|---|
  | @ | NS | ns2.goog.com |

  то это эквивалентно случаю, когда на этом же сервере указана запись

  | | | |
  |---|---|---|
  | .google.com. | NS | ns2.goog.com |

  (В поле Имя может находится как относительное имя (без точки в конце), так и абсолютное (с точной в конце))

**DNS-сообщение**

DNS-сообщение бывает двух типов: DNS-запрос и DNS-ответ. Оба вида имеют одинаковую структуру.

- Первые 12 байт отведены под заголовок, в котором присутствуют следующие поля (каждое по 2 байта):
    - Идентификатор запроса
    - Несколько флагов:
        - Однобитный флаг “запрос/ответ” указывает, является ли сообщение запросом или ответом.
        - Однобитный флаг «авторитетный ответ» устанавливается в ответном сообщении, если DNS сервер является авторитетным для запрашиваемого хоста.
        - Однобитный флаг «требуется рекурсия» устанавливается, когда клиент (хост или другой DNS-сервер) требует от сервера обработать этот запрос самому (рекурсивный запрос).
        - Однобитный флаг «рекурсия возможна» устанавливается в 1 в отклике DNS-сервера в случае, если он поддерживает рекурсию.
    - В заголовок также включены 4 числовых поля, указывающих количество секций данных четырех различных типов, которые идут после заголовка.
- Секция запросов содержит информацию о выполняемом запросе, которая включает, во-первых, поле имени, содержащее запрашиваемое имя хоста, и, во-вторых, поле типа, которое указывает тип запроса, например, адрес, связанный с именем (тип A) или имя почтового сервера (тип MX).
- В ответе DNS-сервера секция откликов содержит ресурсные записи для запрашиваемых имен. Напомним, что каждая ресурсная запись содержит тип (например, A, NS, CNAME и MX), значение и время жизни. DNS-ответ способен возвращать несколько ресурсных записей, так как имя хоста может быть привязано не к одному IP-адресу, например, для реплицированных веб-серверов, обсужденных ранее.
- Секция серверов имен содержит записи других авторитетных серверов.
- Секция дополнительной информации включает другие полезные записи. Например, в отклике на запрос MX содержится ресурсная запись, представляющая каноническое имя хоста почтового

сервера. В секции дополнительной информации будет содержаться запись типа A, в которой находится IP-адрес для канонического имени почтового сервера.


**Рис. 2.23.** Формат сообщений DNS

**Примеры использования DNS другими протоколами прикладного уровня**

- Резолвинг доменных имен для отправки HTTP-запросов.
- Резолвинг псевдонимов почтовых серверов для отправки почты.

# Способы удаленного доступа к файлам. Принципы работы протоколов FTP, SMB, NFS. "Облачное" хранение файлов (Dropbox, etc)

## FTP

FTP (File Transmission Protocol) - супер устаревший протокол передачи файлов по сети. Работает поверх TCP, использует порт 21.
При работе по FTP используется несколько TCP-соединений:

- Одно *управляющее соединения*, которое устанавливает клиент и которое используется для передачи shell-подобных команд от клиента к серверу.
- Несколько *соединений данных*, по одному на каждый передаваемый файл. Бывают два режима установления подобного соединения:
    - В обычном случае используется *активный режим*. В таком случае сервер устанавливает соединение данных с клиентом.
    - Но бывают ситуация, когда подобная схема взаимодействия невозможно, например, по причине того, что маршрутизатор, за которым находится клиент, использует NAT. В таком случае используется *пассивный режим*: клиент сам подключается к серверу на указанный порт.

  Очевидно, что слишком много клиентов одновременно сервер обслуживать не может, ведь ему необходимо хранить состояние всех сеансов, а именно нужно как-то связывать управляющее соединение с конкретной пользовательской учетной записью, сервер должен отслеживать текущий каталог пользователя по мере того, как он пробегает по дереву каталогов, но самое главное, что у сервера могут банально закончится свободные порты.

Основные команды, которые поддерживает FTP:

- USER username: используется для передачи идентификатора пользователя серверу.
- PASS password: используется для передачи пользовательского пароля серверу.
- LIST: используется для запроса у сервера списка всех файлов, находящихся в текущем удаленном каталоге. Список файлов передается через соединение данных (новое и непостоянное), а не через управляющее TCP-соединение.
- RETR filename: используется для вызова (то есть, получения) файла из текущего каталога на удаленном хосте. Данная команда приводит к тому, что удаленный хост инициирует соединение данных и отправляет запрашиваемый файл через это соединение.
- STOR filename: используется для сохранения (то есть отправки) файла в текущий каталог удаленного хоста

! Важно отметить, что в стандартном версии FTP никакого шифрования не происходит, имя пользователя и пароль передаются в незащищенном

виде. Для решения этой проблемы существует протокол FTPS, которые позволяет использовать FTP поверх TLS.

Недостатком данного протокола протокола помимо небольшого количества одновременно поддерживаемых клиентов и отсутствия шифрования является также то, что данный протокол совершенно неудобен в случае, если необходимо, например, внести небольшие изменения в большое количество файлов, ведь для этого каждый файл придется скачать, внести в него изменения и загрузить обратно.

## NFS

NFS - Network File System
Протокол, используемый для локального монтирования удаленным хостом, в основном применяется на UNIX-системах, хотя поддержка для Windows тоже присутствует.
Принцип работы протокола следующий:
пользователь монтирует в своей файловой системе сетевую файловую систему, после чего прикладные программы могут обращаться к файлам на ней, как к обычным локальным файлам. На самом деле, попытки чтения или записи в файл из данной файловой системы будут обрабатываться специальным драйвером с помощью передачи сообщения удаленному хосту - владельцу примонтированной файловой системы.

При чтении файла применяется read-ahead кэширование, то есть запрашивается больше байт данных, чем требуется прикладной программе для увеличения эффективности последовательного чтения файлов.

## SMB

SMB - Server Message Block
Аналог NFS для windows.
Серверы предоставляют клиентам файловые системы и другие ресурсы (принтеры, почтовые сегменты, именованные каналы и т. д.) для общего доступа в сети.
Клиенты соединяются с сервером, используя протоколы TCP/IP (а, точнее, NetBIOS через TCP/IP), NetBEUI или IPX/SPX. После того, как соединение установлено, клиенты могут посылать команды серверу (эти команды называются SMB-команды или SMBs), который даёт им доступ к ресурсам, позволяет открывать, читать файлы, писать в файлы

и вообще выполнять весь перечень действий, которые можно выполнять с файловой системой. Короче, как в NFS
Операции, которые должна поддерживать реализация SMB

- присоединение к файловым и принтерным ресурсам и отсоединение от них;
- открытие и закрытие файлов;
- открытие и закрытие принтерных файлов;
- чтение и запись файлов;
- создание и удаление файлов и каталогов;
- поиск каталогов;
- получение и установление атрибутов файла;
- блокировка и разблокировка файлов.

Модель механизма защиты состоит из двух уровней защиты: user-level и share-level.
Аутентификация на уровне user-level означает, что клиент, который пытается получить доступ к ресурсу на сервере, должен иметь username и password. Если данная аутентификация прошла успешно, клиент имеет доступ ко всем доступным ресурсам сервера, кроме тех, что с share-level-защитой. Этот уровень защиты даёт возможность системным администраторам конкретно указывать, какие пользователи и группы пользователей имеют доступ к определенным данным.
Аутентификация на уровне share-level означает, что доступ к ресурсу контролируется паролем, установленным конкретно на этот ресурс.

## “Облачное” хранение файлов

Облачное хранение файлов – это способ хранения файлов в облаке, позволяющий серверам и приложениям получать доступ к данным через совместно используемые файловые системы.
Собственно говоря, “облачное” хранение файлов предоставляет для пользователя все так же возможности, что и вышеописанные NFS и SMB, то есть создавать, удалять, записывать и считывать файлы, а также упорядочивать их логически в древовидную структуру директорий.
Одним из главным преимуществ облачного хранения файлов является простота использования для пользователя. Существует большое количество пользователей ПК, которые никогда не слышали аббревиатур NFS или SMB, но, несмотря на это, активно пользуются, например, google drive для описанных в предыдущем абзаце целей, а

также для того, чтобы делиться своими файлами с коллегами или друзьями.

### WebDAV

WebDAV (Web Distributed Authoring and Versioning) - расширение для протокола HTTP 1.1, позволяющее выполнять следующие операции:

- Создание добавления/удаления информации о веб-страницах: ее создателе, дате создания и т.д.
- Создание коллекций документов и получения иерархического листинга (подобно листингу каталога файловой системы)
- Блокировка документа для предотвращения “проблемы потерянного обновления“.
- Передачи инструкции серверу о копировании или передвижении веб-ресурсов.

Подробнее можно ознакомиться в RFC 4918.

# Форматы хранения и передачи данных. Json, XML, protobuf. Преимущества и недостатки

### XML

XML (eXtensiable Markup Language) - язык разметки, предназначенный для хранения структурированных данных, обмена этими данными между приложениями, а также создания специализированных языков разметки.

В простейший XML-документ входят две части:

- **Пролог**. Пролог содержит объявление XML-документа, указывающее на то, что это XML-документ, и указывает версию XML-документа. Пролог также может содержать необязательные компоненты такие, как, например, кодировка.
- **Корневой элемент**.

**Синтаксис**

- <element_name> - start_tag, </element_name> - end_tag. Все, что содержится между start_tag и end_tag - содержимое *элемента*.
- Элемент может содержать *атрибуты*, которые предоставляют о нем дополнительную информацию. Нет четких правил, когда

следует использовать атрибуты, а когда элементы, однако считается, что метаданные следует записывать, как атрибуты (см. пример ниже).

```
<messages>
   <note id="501">
      <to>Tove</to>
      <from>Jani</from>
      <heading>Напоминание</heading>
      <body>Не забудь обо мне в эти выходные!</body>
   </note>
   <note id="502">
      <to>Jani</to>
      <from>Tove</from>
      <heading>Re: Напоминание</heading>
      <body>Не забуду</body>
   </note>
</messages>
```

Примеры ниже иллюстрируют разные способы представления одной и той же информации.

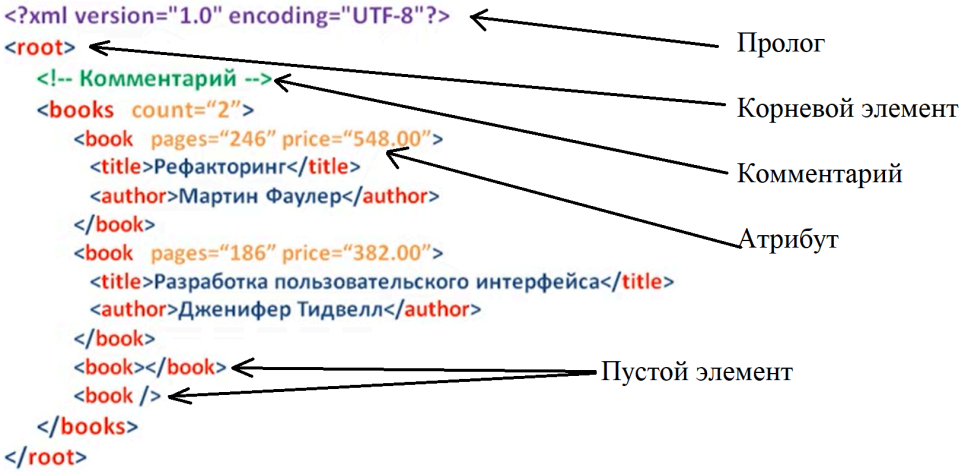
### Пример №1

```
<person sex="female">
   <firstname>Anna</firstname>
   <lastname>Smith</lastname>
</person>
```

### Пример №2

```
<person>
   <sex>female</sex>
   <firstname>Anna</firstname>
   <lastname>Smith</lastname>
</person>
```

- XML-документ может содержать комментарии.
- Для пустого элемента есть краткая форма записи: <element_name/>.




Преимущества:

- ★ Человекочитаемое представление данных.
- ★ Формат является унифицированным и соответствует стандарту.
- ★ Не зависит от платформы.
- ★ Поддерживает различные кодировки.
- ★ Поддерживает комментарии.

Недостатки:

- Синтаксис избыточен (названия полей повторяются дважды).
- Неоднозначность моделирования объектов (один и тот же объект может быть представлен разными способами, например, с помощью элементов или с помощью атрибутов).
- Отсутствие строгой типизации.
- Сложность парсинга.
- Занимает много места.
- Большое время парсинга.

## Json

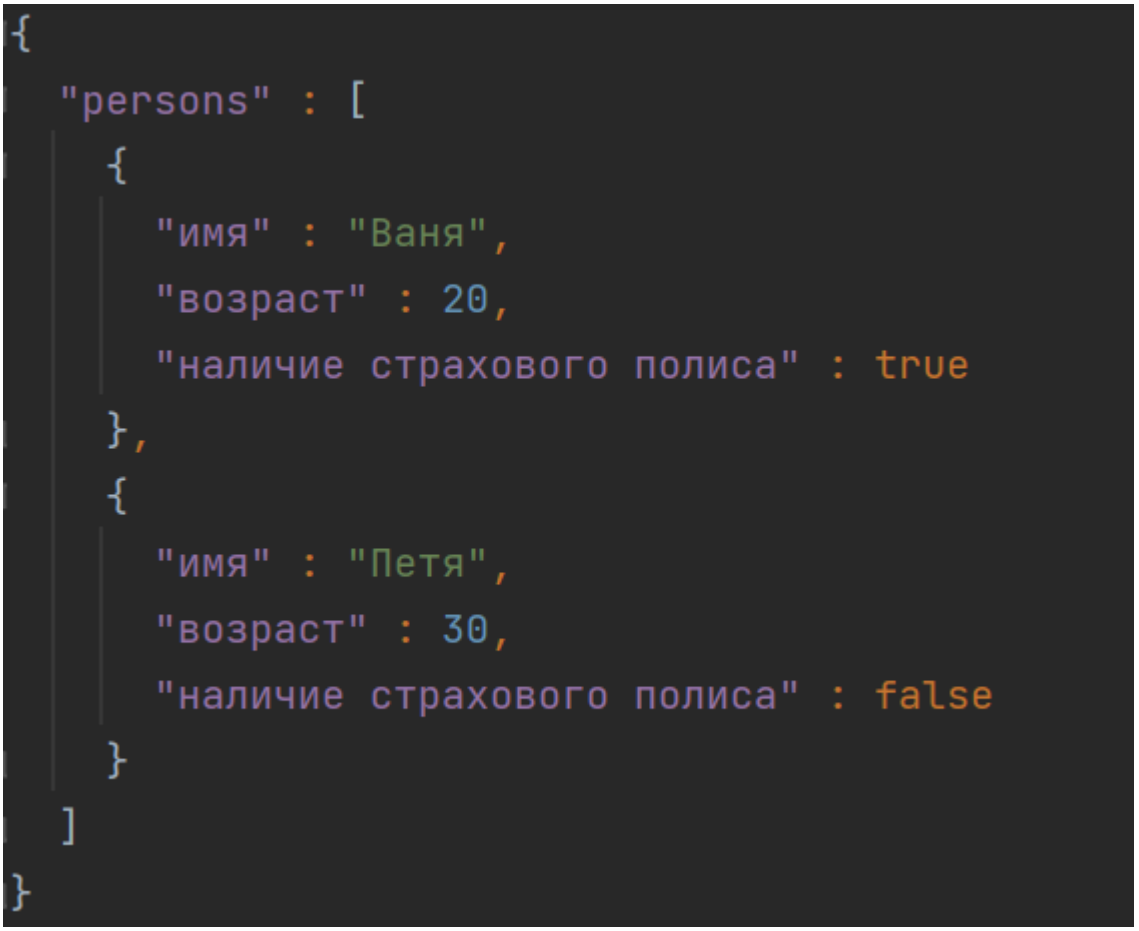
JSON - Java Script Object Notation

**Синтаксис**

JSON-документ состоит из одного JSON-объекта или из одного JSON-массива.

- JSON-объект - это неупорядоченное множество пар ключ-значение. В качестве ключа выступает строка, в качестве значения может выступать другой JSON-объект, JSON-массив, целое или вещественное число, литералы true, false и null, а также строка.
- JSON-массив - это взятый в квадратный скобки, перечисленный через запятую набор значений.

Пример:




Преимущества:

- ★ Человекочитаемость JSON лучше, чем у XML.
- ★ Присутствует типизация.
- ★ Простота парсинга.
- ★ Занимает мало места.

Недостатки:

- Можно использовать только UTF-8.
- Не поддерживает комментарии.

## Protobuf

Protobuf - независимый от языка и платформы расширяемый механизм Google для сериализации структурированных данных, хранения сериализованных данных и передачи их по сети.
В отличие от XML или JSON данные передаются в двоичном формате.
Официальный веб-сайт посвященный protobuf.

**Формат протокола**

Формат протокола определяется в специальном .proto файле.
Основными сущностями, определяемыми в .proto файле являются

сообщения, представляющие собой набор типизированных полей. Поля бывают следующих типов: bool, int32, int64, uint32, uint64, fixed32, fixed64, sfixed32, sfixed64, float, double и string (UTF-8), за что отвечают эти типы данных можно посмотреть вот здесь. Можно также добавить дополнительную структуру данных к своим сообщениям, используя другие типы сообщений в качестве полей, с некоторыми дополнительными структурными типами данных, уже реализованными в protobuf, можно ознакомиться здесь. Также у каждого поля внутри сообщения есть уникальный числовой тэг. Каждое поле может быть аннотировано одним из следующих модификаторов:

- **optional**. Поле может быть установлено, а может быть не установлено. Если поле не установлено, то используется значение по умолчанию.
- **repeated**. Поле может повторяться любое количество раз, в том числе 0.
- **required**. Значение для поля должно быть указано, иначе сообщение будет считаться “неинициализированным”. Помечая поле аннотаций required вы должны понимать, что дороги назад не будет. Если в какой-то момент в будущем вы решите, что вы больше не хотите использовать required-поле, то убрать его будет проблематично, так как приложения, использующие старую версию вашего протокола, будут считать сообщения без этого поля неполными и отбрасывать их, и тогда ни о какой forward compatibility речи быть не может. В proto 3 аннотация required была и вовсе удалена.

**Что делает компилятор с .proto файлом**

- Для **C++** компилятор создает файл .h и .cc из каждого .proto с классом для каждого типа сообщения, описанного в вашем файле.
- Для **Java** компилятор генерирует файл .java с классом для каждого типа сообщения, а также специальные классы Builder для создания экземпляров класса сообщений.
- **Python** немного отличается — компилятор Python генерирует модуль со статическим дескриптором каждого типа сообщения в вашем .proto , который затем используется с метаклассом для создания необходимого класса доступа к данным Python во время выполнения.
- Для **Go** компилятор создает файл .pb.go с типом для каждого типа сообщения в вашем файле.

Преимущества:

- ★ Занимает мало места.
- ★ Автоматический быстрый парсинг.
- ★ Доступен для многих языков программирования.
- ★ Backward compatible
- ★ Forward compatible! (при соблюдении некоторых правил)

Недостатки:

- Не подходит для работы с большими данными (больше нескольких Мбайт).
- Два сообщения нельзя сравнить, не проводя парсинг, так как одни и те же данные могут быть представлены разными бинарными сериализациями.
- Не подходит для работы с большими многомерными массивами чисел с плавающей точкой.
- “Не очень хорошо поддержаны” не ОО языками программирования.
- Сообщения не описывают данные внутри себя, то есть нельзя полностью интерпретировать сообщение без доступа к .proto файлу.
- Не является официальным стандартом ни одной организации.

# Сравнение HTTP 1.x и HTTP 2

В HTTP 2 структура сообщений, да и сама модель сообщений “запрос-ответ” не претерпела больших изменений. Что поменялось, так это способ использования TCP-соединений, а также кадрирование сообщений.

## TCP-соединения и виртуальные каналы

Дело все в том, что на запрашиваемой пользователем веб-странице может быть достаточно большое количество ресурсов (допустим, в среднем 100). И запрашивать их все последовательно было бы совсем неэффективно, нужно как-то распараллелить работу.
В HTTP 1.x клиент открывает для каждого сервера, с которым желает взаимодействовать, несколько TCP-соединений (не больше определенного количества, заданного в настройках клиента, для браузера 5-10), конкретное количество которых зависит от количества запрашиваемых ресурсов. Естественно, такой способ взаимодействия нес за собой массу издержек: на установление каждого соединения

необходимо время, для каждого соединения нужно выделить буфер и переменные соединения, из-за механизма управления перегрузкой TCP при большим количестве соединений, скорость каждого из них может быть очень маленькой, да и вообще страдать производительность всей системы. Мало того, некоторым создателям веб-сервисов не нравится наличие ограничения на количество открытых TCP-соединений, и они применяют специальную хитрость: для некоторых наборов ресурсов создаются отдельные доменные имена, вследствие чего количество соединений, открытых для получения ресурсов из одного набора никак не будет влиять не количество соединений, открытых для получения ресурсов из другого набора. В таком случае эффект уменьшения скорости передачи каждого соединения может начать играть еще более значимую роль.
В HTTP 2 вышеперечисленные проблемы были решены при помощи введения *виртуальных каналов*. Для взаимодействия клиента и сервера создается только одно TCP-соединение, которые может быть поделено на сколько угодно (не больше, чем $2^{31}$ (см. структуру кадра)) виртуальных каналов (например, для каждого ресурса отдельный канал), сообщения кадрируются, каждый кадр содержит информацию о том, к какому каналу он относится. Кадру нет необходимости содержать свой порядковый номер, так как TCP обеспечивает передачу данных в правильном порядке.

**Структура кадра**

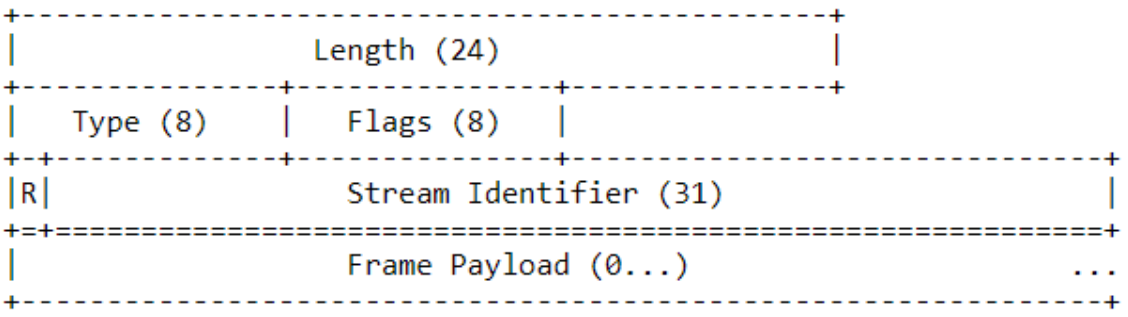



Figure 1: Frame Layout

- Длина полезной нагрузки (Frame Payload) в байтах.
- Тип кадра и 8 флагов, специфичных для каждого типа. Типы кадров:
    - **DATA** - используется для передачи переменного количество байтов, ассоциированных с виртуальным каналом, например, для передачи полезной нагрузки HTTP-сообщений. Специфические флаги:

- **END_STREAM**. Указывает на то, что данный кадр является последним в данном виртуальном канале, и последний переводится в закрытое или полузакрытое состояние.
- **PADDED**. Указывает на то, что к данным было применено padding, в таком случае поле Pad Length является валидным и указывает на количество байт, добавленным при padding'е.

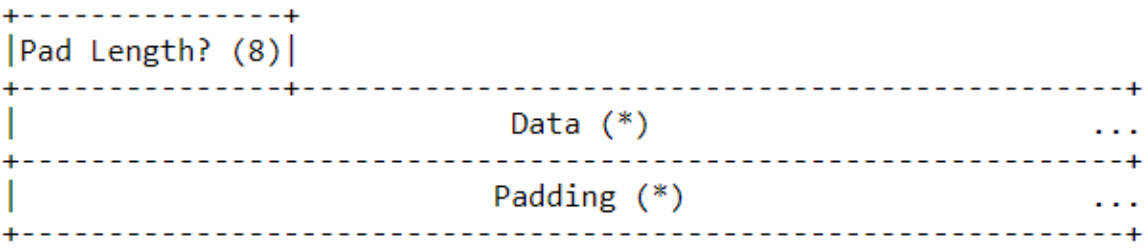



Figure 6: DATA Frame Payload

- **HEADERS** - используется для открытия виртуального канала и передачи заголовков (в HTTP 2 метод и целевой URI передаются в виде заголовков).
  - Как и кадр DATA, кадр HEADERS может содержать Padding.
  - Флаг **PRIORITY** указывает наличие флага E и полей Stream Dependency и Weight. Эти поля указывает на наличие зависимости данного канала от другого виртуального канала и его приоритет (подробнее можно ознакомиться в RFC 7540).
  - Флаг **END_STREAM** играет ту же роль, что и в кадре DATA.
  - Флаг **END_HEADERS** указывает на то, что данный кадр является последним, содержащим заголовки, и после него не будет отправлено кадров CONTINUATION. Обратите внимание на то, что если указан флаг END_STREAM, но не указан влаг END_HEADERS, это говорит о том, что данный кадр содержит не все заголовки, а ожидаются дополнительные кадры CONTINUATION, после получения которых виртуальный канал перейдет в закрытое или полузакрытое состояние, и данные на него поступать уже не будут.

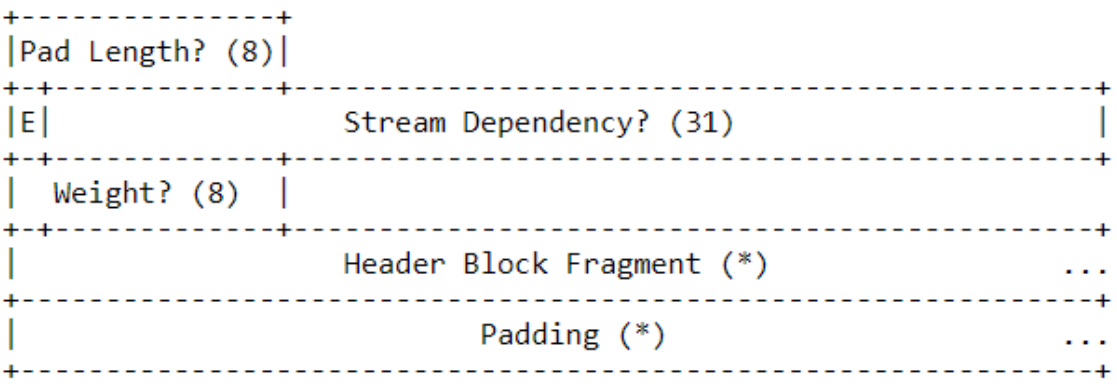



Figure 7: HEADERS Frame Payload

  - **SETTINGS** - используется для передачи конфигурационной информации между участниками соединения.
  - **PING** - используется для измерения минимального Round-trip Time.
  - **CONTINUATION** - используется для продолжения последовательности заголовочных блоков. Так же, как и в случае заголовка HEADERS содержит флаг END_STREAM и END_HEADERS.
- 1 зарезервированный бит (должен быть установлен в 0).
- Идентификатор виртуального канала. Идентификатор 0 используется для сообщений, которые относятся к соединению, как к таковому и не имеют принадлежности к какому-то виртуальному каналу.

## HTTP 2: server push

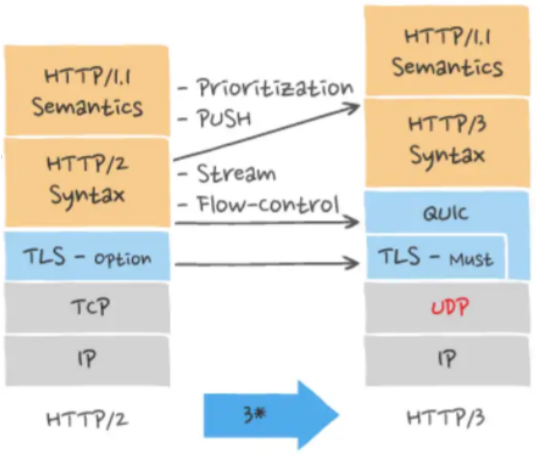
Также у HTTP 2 появилась такая фишка, как Server push. Представим себе ситуацию: клиент общается с сервером, используя HTTP 1.1. Сервер посылает клиенту HTML-документ, при его декодировании клиент обнаруживает ссылки на еще 10 ресурсов и отправляет на них запросы серверу, потом в этих ресурсах также есть ссылки на другие ресурсы и т.д. Server push предназначен для того, чтобы сэкономить время, которое тратится на декодирование ответов. Вместо того, чтобы ждать запрос на очередной ресурс, сервер, осведомленный о наличии ссылки на него из только что запрошенного клиентом ресурса, может сразу же послать его.

## HTTP 3

Какие недостатки есть у HTTP 2:

- HTTP-мультиплексирование (разбиение TCP-соединения на несколько виртуальных каналов) несомненно принесло много пользы клиентам, но не серверам, так как большое количество одновременно приходящих запросов увеличило среднюю загруженность их центральных процессоров и, как следствие, задержку.
- Еще одна проблемой, связанной с мультиплексированием, является его невидимость для нижележащего TCP. Из-за его механизма обеспечения надежной передачи данных может появиться задержка получения сообщения из-за повторной отправки сегмента, который на самом деле к этому сообщению вообще не относится. Данная проблема называется Head of Line Blocking (подробнее можно посмотреть здесь).
- Дополнительной проблемой стали push'и от серверов, например, в случае, когда ресурс, отправляемый с помощью пуша, уже находится в кэше клиента. Конечно, можно использовать специальные механизмы для предотвращения повторной загрузки в кэш, но издержки от отправления лишнего push-сообщения от сервера никуда не денутся.

Для решения вышеописанных проблем был придумал HTTP 3.
Самым главным отличием HTTP 3 от HTTP 2 стало то, что вместо TCP стал использоваться протокол QUIC поверх UDP.




QUIC - Quick UDP Internet Connections. Разработан Google.

- Благодаря использованию UDP вместо TCP:
  - Потерянный сегмент задерживал только один канал, а не все.
  - Обработка ошибок стала лучше, количество лишних перенаправлений данных уменьшилось.

    - Появился более гибкий механизм управления перегрузкой.
- QUIC предоставляет свой механизм мультиплексирования с отдельным для каждого канала контролем потока.
- Обязательным было использование TLS 1.3, что увеличило скорость handshake (см. здесь).

Преимущества:

★ Уменьшение задержки благодаря использованию UDP и QUIC вместо TCP.

Недостатки:

- Переход с HTTP 2 на HTTP 3 сложнее, чем переход с HTTP 1.x на HTTP 2, так как помимо изменений на прикладной уровне, произошли изменения на транспортном уровне.

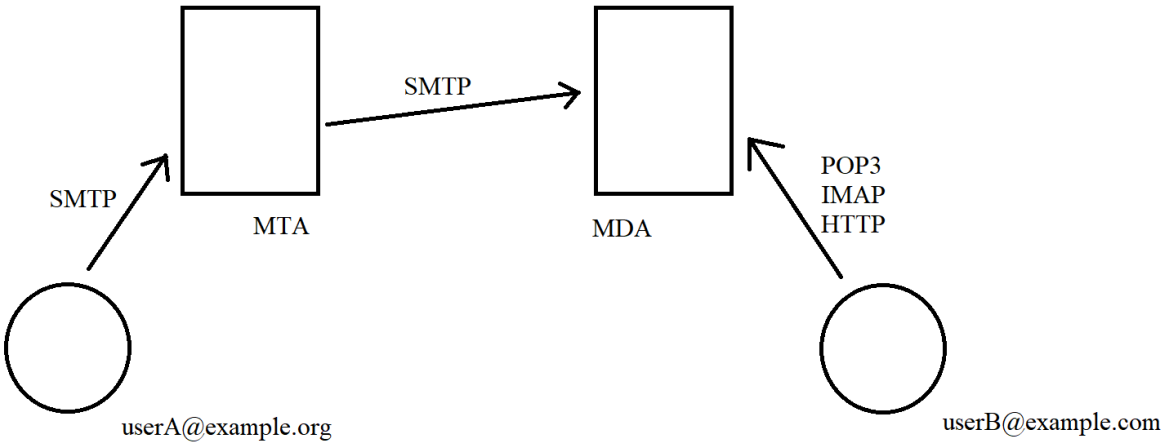
Подробнее можно почитать здесь и в RFC 9114.

# Email: принципы функционирования, протоколы SMTP, POP3, IMAP. Вопросы безопасности: подделка адресов, модификация сообщений, спам. Технологии SPF, DKIM, DMARC

## Email: принципы функционирования

Посмотрим, что происходит, когда userA@example.org отправляет электронное письмо пользователю userB@example.org (см. рисунок):

1. Клиентское приложение пользователя A отправляет письмо на почтовый сервер своей доменной зоны, который выступает в роли MTA (Mail Transfer Agent) с помощью протокола SMTP (Simple Mail Transfer Protocol).
2. MTA принимает это письмо и устанавливает соединение с почтовым сервером из доменной зоны пользователя B, который выступает в роли MDA (Mail Delivery Agent), и передает ему письмо также по протоколу SMTP.
3. Когда пользователь B желает проверить почту, он запускает свой почтовый агент, который устанавливает соединение с MDA и забирает письма пользователя B.




Несколько важных замечаний:

1. В данной цепочке могут присутствовать промежуточные серверы.
2. Если пользователи A и B находятся в одной доменное зоне, то в качестве MTA и MDA может выступать один и тот же сервер.

## SMTP

Simple Mail Transfer Protocol - командный протокол прикладного уровня с сохранением состояния соединения. Порт по умолчанию: 25.
В общем случае для передачи письма серверу клиент должен выполнить следующие действия:

- Выполнить аутентификацию.
- Указать отправителя письма (MAIL FROM).
- Указать получателя письма (RCPT TO).
- Указать само тело письма.

**Тело письма**

В начале тела письма идут заголовки:

- Обязательные:
    - From - отправитель.
    - To - получатель или получатели.
- Необязательные:
    - Subject - тема письма.
    - Cc (Carbon Copy) - вторичные получатели письма.

Затем идет текст. Тело письма оканчивается точкой на отдельной строке. А что произойдет в случае, когда пользователь захочет включить в тело письма точку на отдельной строке? Для

предотвращения преждевременного окончания письма используется следующий механизм (RFC 5321):

1. Перед отправкой очередной строчки текста клиент проверяет ее первый символ. Если этот символ - точка, тогда еще одна точка вставляет в начала этой строки.
2. SMTP-сервер при получении строки текста также проверяет ее. Если она состоит из единственной точки, то это воспринимается как индикатор конца письма. Если в начале строки стоит точка, а за ней идет еще какие-то символы, то первая точка удаляется.

Вышеописанный формат описывает отдельный стандарт: MIME (*Multipurpose Internet Mail Extensions*), а тело письма называется MIME-сообщением (RFC 2045).
Основная проблема данного протокола заключается как раз таки в том, что дальше идет именно текст в семиразрядной кодировке ASCII. SMTP не поддерживает передачу данных  в других кодировках или передачу двоичных данных. Для этого применяются различного рода костыли.

**Quoted-printable**

Использование системы Quoted-printable - это один из способов представления произвольных данных в виде ASCII-текста.
При использовании этой системы байты, кодирующие ASCII-символы, остаются неизменными, остальные представляются тремя ASCII-символами: “=” и двумя шестнадцатеричными цифрами, которые собственно и представляют кодируемый байт. Данный способ позволяет эффективно кодировать тело письма, если большая его часть - текст в кодировке ASCII, в противном случае использование будет неэффективным, так как каждый байт кодируется тремя ASCII символами, т.е. тремя байтами.

**Base64**

Другой способ представления произвольных данных в SMTP - использования base64. Размер сообщения увеличивается в 4/3 по сравнению с оригиналом. Какой способ кодирования выбирать, quoted printable или base64, зависит от данных в теле письма, если это в основном ASCII-текст, то quoted printable, иначе base64.

**Приложения**

Для прикрепления к телу письма приложений, например, аудио- или видеофайлов применяется следующий метод: в заголовке письма указывается длинная строка произвольной длины. Данная строка будет использоваться в качестве границы приложения в теле письма: все, что

находится между двумя подобными строками следует интерпретировать как приложение.

## Получение письма

### POP3

Post Office Protocol - протокол прикладного уровня для доступа к почте. Порт по умолчанию: 110.
После установления соединения работа с POP3 происходит в три этапа:

- **Авторизация**. Агент пользователя отправляет учетные данные (логин и пароль) пользователя серверу.
- **Транзакция**. Пользовательский агент получает сообщение, а также может помечать сообщения для удаления, удалять эти пометки, запрашивать почтовую статистику. Пользовательский агент может быть сконфигурирован для работы в двух состояниях: “загрузить и удалить” и “загрузить и сохранить”. После фазы авторизации агент использует только 4 команды: list, retr, dele и quit, последняя из которых переводит работу протокола в состояние “обновление”.
- **Обновление**. Почтовый сервер удаляет помеченные сообщения и закрывает соединение.

Одной из проблем POP3 является отсутствие структуры писем. Конечно, после того, как пользователь загрузит письма с почтового сервера на свой агент, он может рассортировать полученные сообщения по каталогам, но что если на следующий день ему потребуется посмотреть свои письма с другого устройства?
Отсутствие каталогов в POP3 это, в том числе, следствие того, что длительное хранение писем на MDA в принципе не предполагалось. Со временем стало понятно, что пользователи хотят хранить свои письма на сервере и получать к ним доступ с различных устройств. Для этого и еще кое-чего был разработан IMAP.

### IMAP

Internet Message Access Protocol - протокол прикладного уровня для доступа к электронной почте. По умолчанию используется порт 143. Как уже было сказано протокол IMAP поддерживает каталоговую структуру хранения электронной почты. Еще немаловажной особенностью IMAP является то, что он позволяет пользовательскому агенту получать только отдельные части MIME-сообщения, например, при

низкоскоростном соединении получать только заголовки и текст письма без получения аудио- или видеоприложений.

## Борьба со спамом

### SPF

SPF (Sender Policy Framework) - тип DNS-записи, разработанный специально для борьбы со спамом. В SPF указывается список почтовых серверов, которые могут отправлять письма, указывая в качестве отправителя пользователя из этого домена.
Недостатком SPF является то, что в случае прохождения письма через промежуточные серверы MDA не сможет проверить доменное имя отправителя. Конечно, последний промежуточный сервер может указать доменное имя сервера-отправителя в специальном заголовке, но в таком случае мы должны доверять последнему промежуточному серверу, а так бывает не всегда.

### DKIM

DKIM (DomainKeys Identified Mail) — технология проверки электронной почты, с помощью которой можно вычислить поддельные письма. В доменной зоне сервера-отправителя создается TXT-запись, содержащая публичный ключ. С помощью приватного ключа создается цифровая подпись на основе данных письма и помещается в само письмо. Получатель может расшифровать цифровую подпись с помощью вышеописанного публичного ключа и сравнить полученные данные с данными из письма.

### DMARC

DMARC (Domain-based  Authentication, Reporting and Conformance) расширяет SPF и DKIM с помощью введения настраиваемых на MDA политик, которые описывают, что нужно сделать с письмом, если какие-то из проверок завершились неудачно. В этих политиках также может быть указано, какие отчеты необходимо посылать владельцу (администратору) домена. Отправителю писем также предоставляется отчет об их статусе.

# Работа TCP: установка и завершение соединения, передача данных и подтверждение доставки, контроль размера сегмента и размера окна.

TCP-соединение всегда является двухточечным и дуплексным (то есть предоставляет возможность передавать данные в обоих направлениях).
*MTU* (Maximum Transmission Unit, максимальный передаваемый блок) - длина наибольшего фрагмента канального уровня, который может быть передан текущим хостом.
*MSS* (Maximum Segment Size, максимальный размер сегмента) - максимальный объем данных, который может быть помещен в сегмент. Обычно определяется как (MTU - длина заголовка TCP/IP).
*RTT* (Round-Trip Time) - время, которое проходит между отправкой TCP-сегмента и приемом подтверждения о его получении.

## Структура TCP-сегмента

- 16-разрядные номера портов отправителя и получателя для мультиплексирования и демультиплексирования (см. здесь).
- 16-разрядное поле контрольной суммы для проверки ошибок.
- 32-битные поля порядкового номера и номера подтверждения используемые для реализации службы надежной передачи данных.
- 4-разрядное поле длины заголовка, определяющее длину TCP-заголовка в 32-разрядных слова.
- 16-разрядное поле окна приема, свидетельствующее о количестве свободного места в буфере получателя.
- Поле флагов, состоящее из 6 бит:
    - Бит ACK используется для указания на корректность значения, размещенного в поле подтверждения. То есть, говоря, простым языком, если данный бит установлен в 1, то это сигнализирует о том, что данный сегмент является подтверждением успешного приема сообщения.
    - Установка бита PSH означает, что получатель должен незамедлительно отправить данные на верхний уровень (на практике не используется).
    - Установка бита URG указывает на то, что в сегменте присутствуют данные, которые объект верхнего уровня

стороны отправителя отметил как срочные, в таком случае указатель на конец срочных данных хранится в 16-разрядном поле указателя срочных данных (на практике не используется).
    - Биты SYN и FIN нужны соответственно для установки и для завершения соединения.
    - Бит RST устанавливается в сегменте, которые является ответом на попытку подключения к порту, для которого нет сокета.
- Необязательное поле параметров, которое используется, когда отправитель и получатель договариваются о максимальном размере сегмента (MSS) или при увеличении размера окна в высокоскоростных сетях.


**Рис. 3.29.** Структура TCP-сегмента

### Поля порядкового номера и номера подтверждения

*Порядковый номер сегмента* - это номер первого байта данных в сегменте.
На практике стороны протокола TCP выбирают начальный номер сегмента случайным образом. Это объясняется необходимостью минимизировать вероятность нахождения в сети сегмента, который сформирован другим (уже закрытым) TCP-соединением между этими же двумя хостами, но который может быть по ошибке отнесен к существующему TCP-соединению.

Необходимо отметить, что даже если сегмент не содержит никаких данных, а нужен только для того, чтобы подтвердить получение другого сегмента, он все равно содержит порядковый номер сегмента, который на единицу больше, чем порядковый номер предыдущего отправленного данным хостом сегмента.
Пример для telnet-приложения:

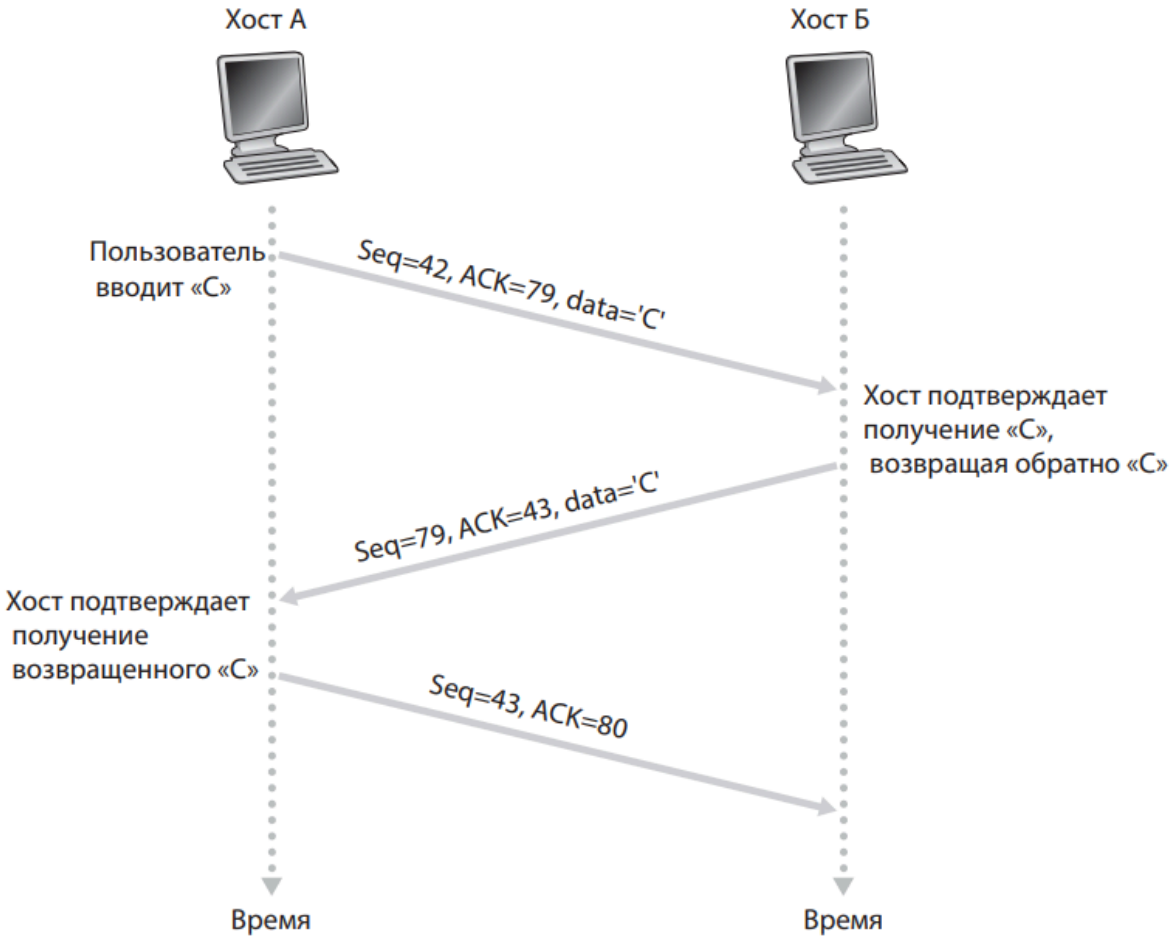



**Рис. 3.31.** Порядковые номера и номера квитанций для простого Telnet-приложения работающего по протоколу TCP

*Номер подтверждения* - это порядковый номер следующего байта, ожидаемого хостом.
Представим себе ситуацию: хост А пересылает хосту Б 1000 байт, разделенных на 10 сегментов по 100 байт. Хост Б сначала принял байты с 0 по 299, потом с 500 по 699, в следующем сегменте, отправляемым хостом А, номер подтверждения будет равен 300.
Как TCP должен реагировать на подобное получение сегментов в неправильном порядке?

Спецификация протокола предоставляет программистам свободу в решении этого вопроса. Выделяются два основным подхода:

1. Сегменты, пришедшие в неверном порядке, немедленно игнорируются. Этот подход упрощает программирование.
2. Сегменты, пришедшие в неверном порядке, сохраняются до тех пор, пока недостающие данные не будут доставлены. Этот подход повышает эффективность использования линии.

## Время оборота и интервал ожидания

Есть определенная вероятность потери сегмента при его передаче по сети. Для того, чтобы гарантировать доставку всех сегментов до получателя, используется таймер, по истечении времени которого в случае неполучения подтверждения о приеме производится еще одна попытка отправить сегмент (сегменты). Далее будет описан способ, с помощью которого определяется время таймаута.

SampleRTT (Sample Round-Trip Time) - время, прошедшее с момента отправки пакета, до момента, когда было получено подтверждение о его приеме.
EstimatedRTT = (1 - a) * EstimatedRTT + a * SampleRTT. Данная величина в статистике называется экспоненциальным весовым скользящим средним. EstimatedRTT обновляется каждый раз при обновлении SampleRTT. Данная величина введена для того, чтобы усреднить SampleRTT, которое может изменяться скачкообразно. Обычно a = 0,125. То есть
EtimatedRTT = 0,875 * EstimatedRTT + 0,125 * SampleRTT. При таком значении a последние значения SampleRTT имеют большие вес, чем полученные ранее.
DevRTT = (1 - b) * DevRTT + b * |SampleRTT - EstimatedRTT|.
Обычно b = 0,25. То есть
DevRTT = 0,75 * DevRTT + 0,25 * |SampleRTT - EstimatedRTT|.

Каким же должно быть время ожидания?
Оно точно должно быть больше, чем EstimatedRTT, так как иначе будут производится лишние повторные передачи, но и сильно больше, чем EstimatedRTT время ожидания быть не должно, ведь в таком случае TCP не сможет своевременно отреагировать на потерю сегмента.
TimeoutInterval = EstimatedRTT + 4 * DevRTT.

Изначально TimeoutInterval = 1 с. После получения подтверждения о приеме сегмента, для которого засекалось время, TimeoutInterval пересчитывается по вышеописанной формуле. При превышении таймаута происходит повторная отправка сегмента, а само значение времени ожидания увеличивается вдвое.

## Выборочное подтверждение

Представим себе ситуацию хост А пересылает хосту Б 5 сегментов по 200 байт. Сегмент с 200 по 399 байт был потерян. Стандартный версия TCP предписывает хосту Б послать сообщение с номером подтверждения равным 200, таким образом, хост А будет вынужден переотправить 4 сегмента, начиная с 200 байта, ведь у него нет никакой информации об успешности доставки сегментов с 400 по 999 байт. Для избежания подобных лишних повторных передач используется модификация TCP, которая называется выборочное подтверждения, позволяющая принимающей стороне подтверждать прием сегментов, пришедших в неверном порядке.

## Управление соединением

### Установление соединения

Процесс установления соединения в TCP называется тройным рукопожатием.

1. Клиентская сторона желает инициализировать соединение с сервером и отправляет ему так называемые SYN-сегмент, в этом сегменте бит SYN выставлен в единицу, также в поле порядкового номера установлено случайное число client_isn, с которого будут нумеровать все дальнейшие посылаемые клиентом байты данных.
2. Серверная сторона принимает SYN-сегмент и отправляет ответный SYNACK-сегмент. У этого сегмента бит SYN также установлен в единицу, в поле номера подтверждения установлено значение client_isn + 1, в поле порядкового номера установлено случайное значение, выбранное сервером, server_isn. В этот момент сервер также выделяет буфер для TCP-соединения и переменные соединения.
3. При получении SYNACK-сегмента клиент также выделяет буфер для соединения, а также переменные соединения, после чего высылает серверу третий сегмент, в котором поле номера подтверждения установлено в server_isn + 1, бит SYN установлен в

0. Также третий сегмент,  в отличие от первых двух, уже может содержать прикладные данные.

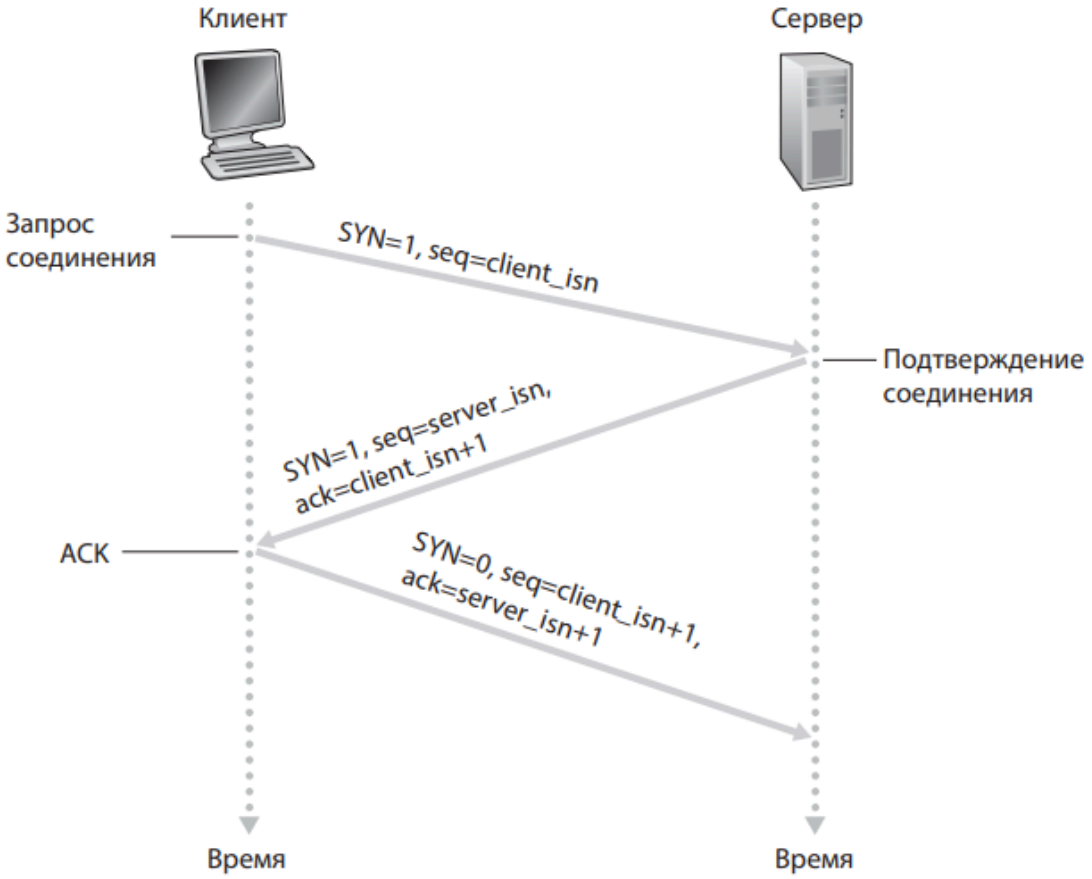



**Рис. 3.39.** Обмен сегментами в тройном рукопожатии протокола TCP

**Завершение соединения**

Одна из сторон посылает сегмент с выставленным в единицу битом FIN, вторая сторона подтверждает приемку FIN-сегмента, после чего посылает свой FIN-сегмент. Потом первая сторона посылает свое подтверждение, после чего переходит в состояние TIME_WAIT, которое в зависимости от реализации может длиться разное время (обычно 30 сек, 1 мин или 2 мин). В это состояние первая сторона переходит для того, чтобы переотправить последнее подтверждение, если оно будет потеряно.

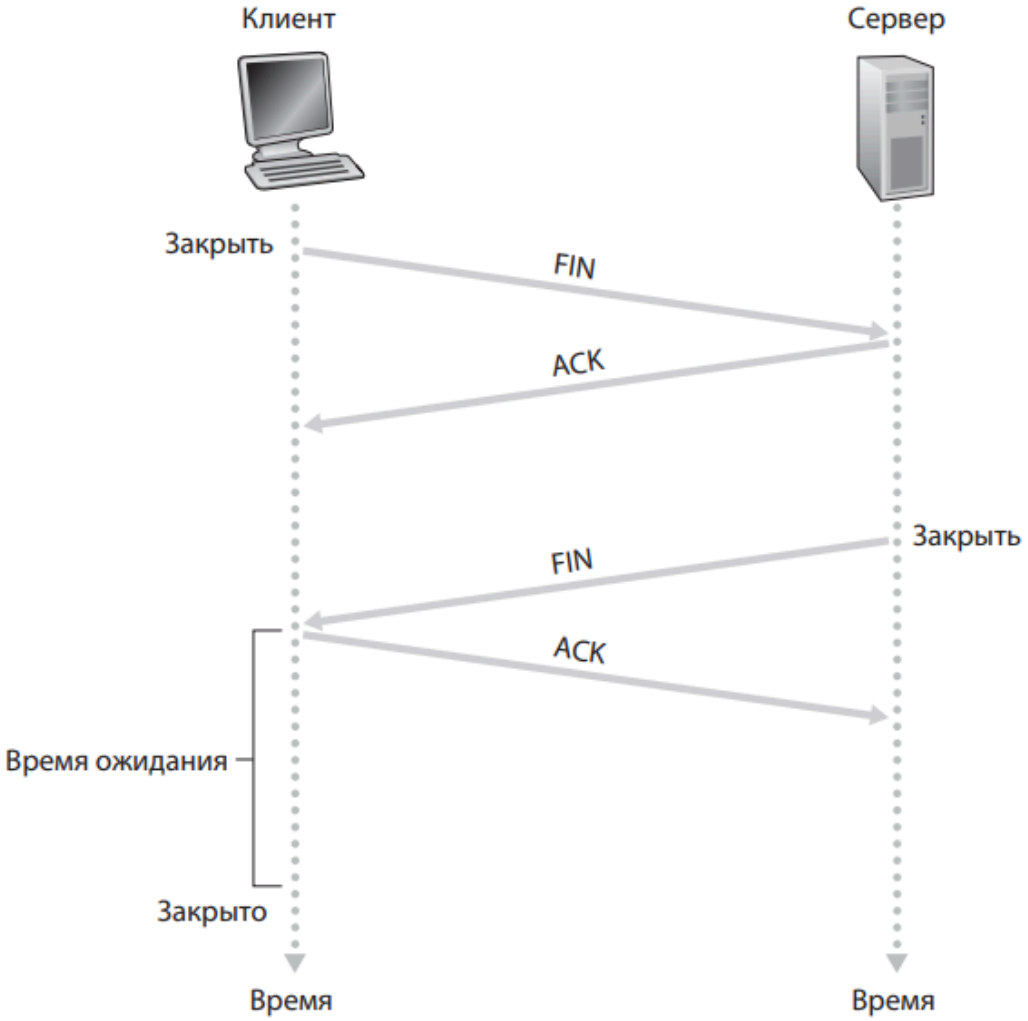



**Рис. 3.40.** Завершение TCP-соединения

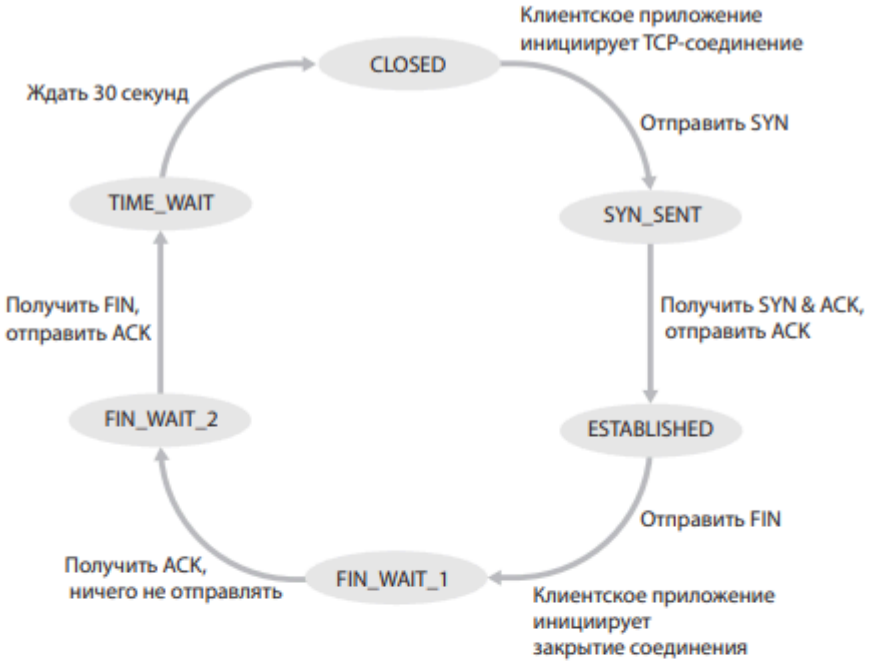



**Рис. 3.41.** Обычная последовательность TCP-состояний клиента

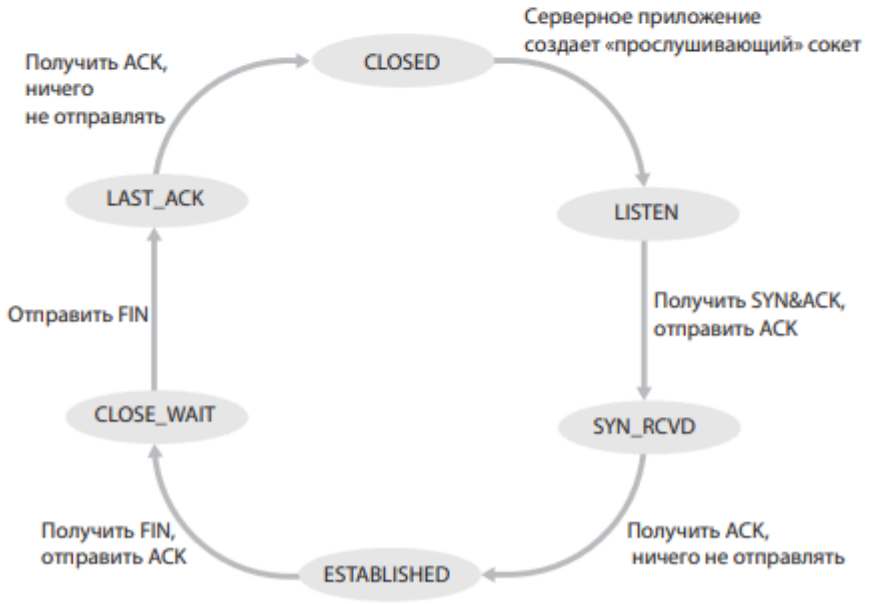



**Рис. 3.42.** Обычная последовательность состояний TCP проходимых TCP серверной стороны

## Управление потоком

Управление потоком - это механизм, предназначенный для предотвращения переполнения буфера на стороне получателя. Реализован данный механизм с помощью поля окна приема, которое, неформально говоря, сообщает отправителю количество свободного места в буфере получателя. В соответствии со значением данного поля отправитель регулирует поставку сегментов таким образом, что

количество неподтвержденных байтов было меньше либо равно, чем окно приема.
Есть одна проблема. Допустим буфер получателя полностью заполнен, отправитель знает об этом и больше сегментов не отправляет. Со временем буфер очищается, но нет никакого способа сообщить отправителю об этом, и отправитель до конца своих дней будет думать, что буфер получателя забит до отказа, и не будет посылать данные. Для предотвращения подобной ситуация протокол предписывает отправителю периодически посылать сегменты размером 1 байт получателю в случае, если буфер последнего был заполнен. Со временем буфер очистится, и отправитель узнает об этом.

## Управление перегрузкой

- Чем ближе скорость передачи данных к пропускной способности линии связи, тем большими становятся задержки ожидания пакетов.
- Повторные передачи в сочетании со значительными задержками ожидания приводят к бесполезной трате значительной доли пропускной способности линий связи.
- При отбрасывании пакета на одном из маршрутизаторов работа каждого предыдущего маршрутизатора по передаче пакета получателю считается проделанной впустую.

Перед началом описания механизма управления перегрузкой необходимо отметить, что в TCP существует еще один механизм обеспечения надежной передачи данных, который называется *ускоренная повторная передача*. Принцип действия данного механизма заключается в том, что получение четырех ACK-сегментов с один и тем же номером подтверждения (один исходный и три дублирующихся) воспринимается отправителем, как потеря следующего сегмента, что побуждает его повторно отправить данные из этого сегмента, не дожидаясь истечения таймаута.

В алгоритме управления перегрузкой TCP лежит в принципе не сложная идея: пока данные не теряются, скорость передачи нужно повышать, как только данные начинают теряться (о чем свидетельствует истечение таймаута, либо получение четырех подтверждающих ACK-сегментов с одинаковым номером подтверждения), скорость передачи нужно понижать.

**Алгоритм управления перегрузкой TCP (Якобсон, RFC 5681)**

*cwdn* - окно перегрузки, количество байт, которое отправитель может послать за один RTT.
*ssthresh* (slow start threshold) - переменная соединения TCP, смысл которой будет понятен в дальнейшем описании алгоритма.

Данный алгоритм состоит из трех этапов:

1. **Медленный старт**. Значение cwnd устанавливается равным одному MSS, то есть начальная скорость передачи данных MSS/RTT. При каждом получении подтверждения о приеме сегмента cwnd увеличивается на один MSS. Таким образом, происходит удвоение cwnd каждый RTT. Этот процесс заканчивается в двух случаях:
    a. Произошло событие потери (истечение интервала ожидания). Тогда cwnd снова устанавливается в один MSS, а значение переменной ssthresh устанавливается равным половине предыдущего значения cwnd.
    b. Достигнуто значение ssthresh / 2. Тогда TCP переход на этап предотвращения перегрузки.
    c. Произошло получение трех дублирующихся ACK-сегментов. TCP выполняет ускоренную повторную передачу, значение ssthresh устанавливается равным половине cwnd, значение cwnd устанавливается равным (ssthresh + 3 * MSS), после TCP чего переходит в состояние быстрого восстановления.

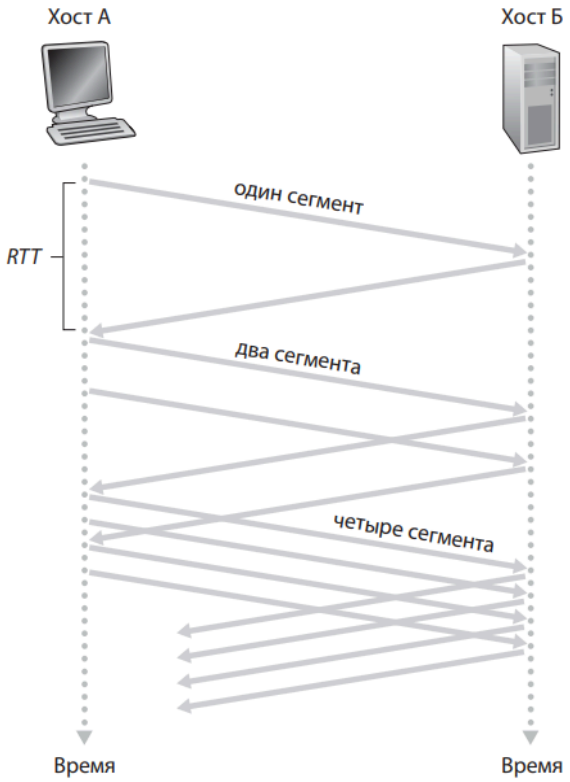



**Рис. 3.51.** Медленный старт TCP

2. **Предотвращение перегрузки**. В этом состоянии TCP увеличивает cwnd на 1 MSS каждый RTT. Обычный это делается так: В начале очередного RTT фиксируется значение cwnd - cwnd0, при каждом приеме подтверждающего сегмента значение cwnd увеличивается на (MSS / cwnd0) * MSS байт. То есть, если MSS равно 1460 байт и cwnd равно 14 600 байт, тогда 10 сегментов будут отправлены за один RTT. Каждый принятый ACK-пакет (предположим, что на сегмент данных приходит один сегмент ACK) увеличивает размер окна перегрузки на 1/10 MSS, и тогда значение окна перегрузки будет увеличено на один MSS после получения всех ACK пакетов на все 10 сегментов данных. Из данного состояния TCP выходит из-за истечения времени ожидания или из-за получения трех дублирующихся ACK-сегментов, действия, которые выполняются в этих случаях, аналогичны действиям состояния медленного старта.
3. **Этап быстрого восстановления** (является необязательным). Когда приходит подтверждающий сегмент для пропущенного пакета, TCP переходит в состояние предотвращения перегрузки, при этом устанавливая значение cwnd равным ssthresh. При истечения интервала ожидания выполняются те же действия, которые

выполнялись бы в состоянии предотвращения перегрузки или медленного старта.

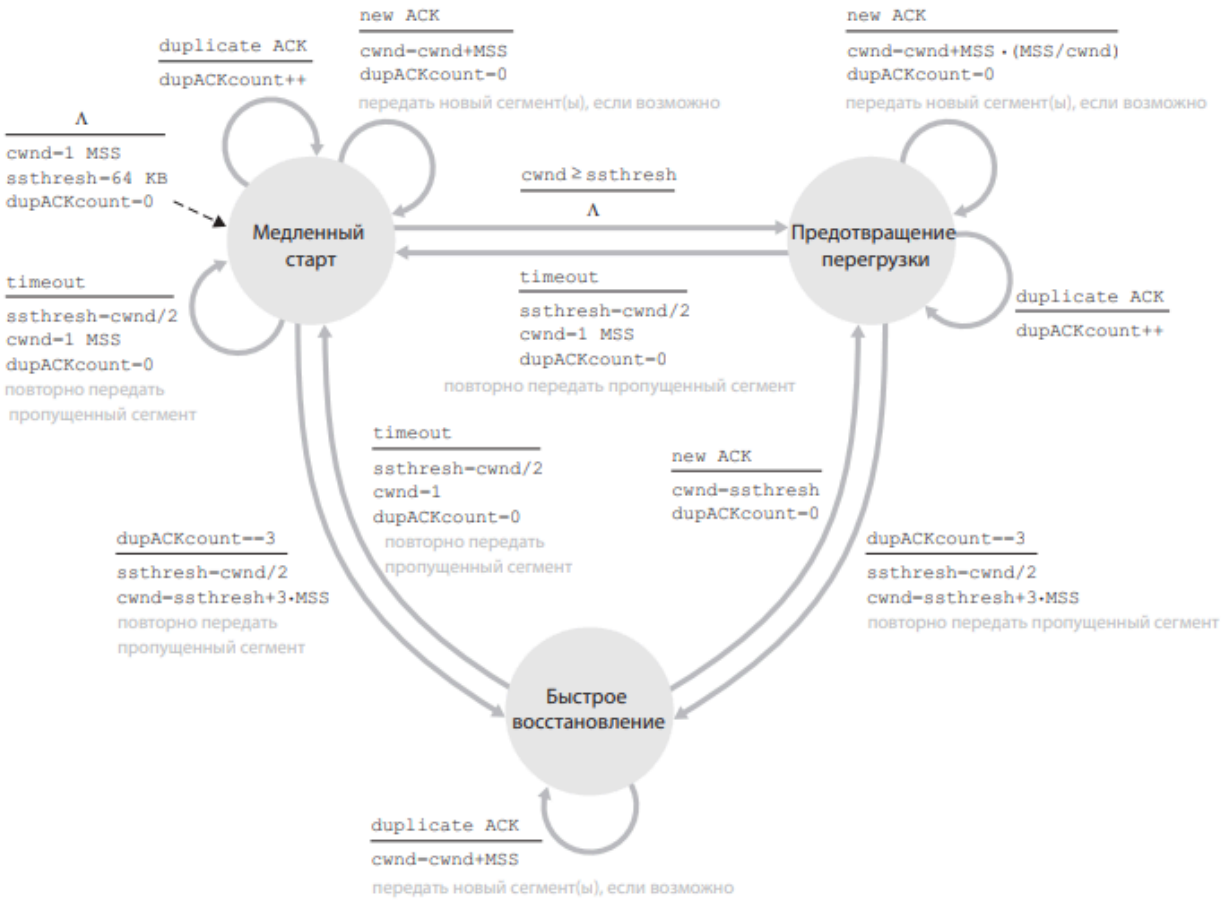



**Рис. 3.52.** FSM-схема управления перегрузкой протокола TCP

# Однобайтовые и многобайтовые кодировки, их применимость, преимущества и недостатки. Юникод. Устройство кодировок UTF-8, UTF-16, UTF-32

Общая идея кодирования текста очень проста. Берется некоторый набор символов, каждому из которых в соответствие ставится некоторое целое число. Затем каждый символ в тексте записывается в виде последовательности байт, представляющей соответствующее данному символу число.

## Endianess

Порядок следования байтов от старшего к младшему.
*Big endian* - самый старший бит слева, самый младший бит справа (такой порядок используется в IP, TCP, UDP).
Little endian - самый младший бит слева, самый старший бит справа.

## ASCII

- Символы кодируются 1 байтом.
- Всего можно закодировать 128 символов, первый бит в каждом байте всегда равен 0.
- Символы, которые можно закодировать: английские буквы (строчные и заглавные), цифры, служебные символы и знаки препинания.

Существует ряд кодировок, дополняющих ASCII символами из национальных алфавитов разных стран, эти символы также кодируются 7 битами, самый старший бит равен 1 (всего можно закодировать 128 “дополнительных” символов). Например, для кириллицы: KOI8-R, ISO8859-5, CP 866.

Преимущества:

- ❖ Простота декодера.
- ❖ Текст занимает минимально возможный объем памяти.
- ❖ Имеется возможность начинать декодирование с произвольного байта.

Недостатки:

- В стандартной ASCII очень мало символов.
- Огромное количество “дополняющих ASCII” кодировок вносят неразбериху в процесс кодирования/декодирования.

Область применения:

- MIME-сообщения хранятся в 7-разрядной ASCII кодировке.
- Доменное имя может состоять только из ASCII символов.
- request-line и status-line в HTTP представлены в кодировке ASCII.

## UNICODE

Это не кодировка, это стандарт кодирования codepoint’ов (не символов).
Всего представлено примерно 1 млн codepoint’ов.
Unicode разделяет все codepoint’ы на две большие группы:

- *Basic Multilingual Plane*: codepoint’ы из BMP имеют номера от 0 до $2^{16}$ - 1, причем первые 128 codepoint’ов- символы из ASCII, в BMP

входят цифры, знаки препинания, буквы из национальных алфавитов европейских стран и т.д.
- *Extended Multilingual Plane*: codepoint'ы из EMP имеют номера больше либо равные $2^{16}$.

Важно отметить, что codepoint'ы в данных группах расположены не в виде непрерывной последовательности, а в виде некоторых блоков, в которых есть незаполненные области, чтобы была возможность добавления новых codepoint'ов в будущем.

## UCS-4 (UTF-32)

- 1 codepoint кодируется четырьмя байтами, можно было бы кодировать и тремя, но сделано четырьмя для выравнивания.

Преимущества:

- ❖ Имеется возможность начинать декодирование с произвольного codepoint'a.
- ❖ Легко найти codepoint по номеру его позиции в тексте.
- ❖ Легко можно определить количество codepoint'ов в тексте.

Недостатки:

- Текст закодированный в UCS-4 занимает слишком много места. Даже если 98% символов в нашем тексте входят в ASCII, их все равно нужно кодировать 4 байтами.
- Нужно учитывать endianness.

## UCS-2

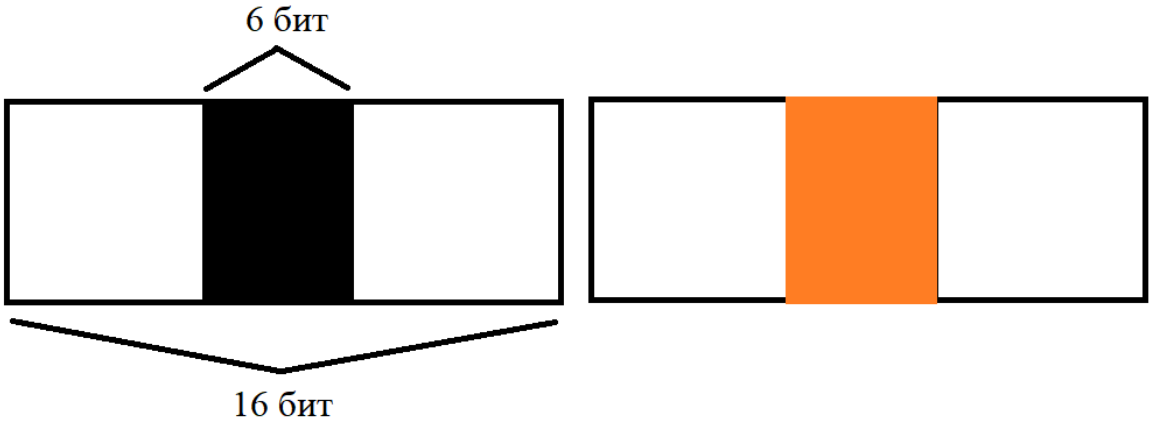
- Специальная кодировка для codepoint'ов из BMP. Если codepoint из BMP декодируем его, иначе рисуем ◆.
- Каждый codepoint кодируется 2 байтами.
- Преимущества и недостатки в целом аналогичны USC-4.

## UTF-16

Кодирование производится следующим образом:

- Если символ из BMP, то он кодируется 2 байтами аналогично UCS-2 (следовательно, UTF-16 совместим с UCS-2).
- Если символ из EMP, то применяется следующий способ: четыре байта разбиваются на так называемую суррогатную пару. У каждого элемента пары 6 бит зарезервировано под специальную комбинацию, причем у первого элемента комбинация одна, у второго элемента комбинация другая. С помощью этих зарезервированных полей декодер поймет, что перед ним codepoint не из BMP, закодированный 2 байтами, а из EMP,

закодированный 4 байтами. Остальные (32 - 12) = 20 бита используются для кодирования самого codepoint'а.
Важно понимать, что поскольку имеются зарезервированные комбинации, codepoint'ы из BMP нельзя кодировать 2 байтами, в которых одна из этих комбинаций содержится. То есть ($2^{16}$ - 2 * $2^{16-6}$) мест для codepoint'ов из BMP занято.




Преимущества:

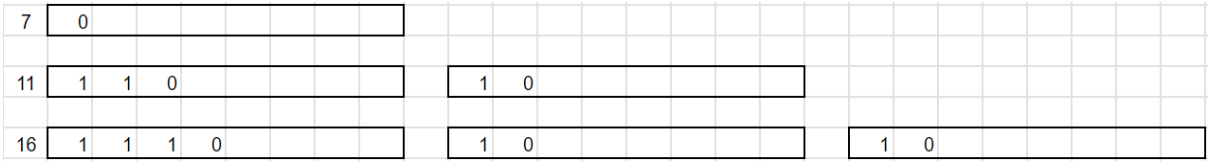
- ❖ Более экономное использование памяти по сравнению с UTF-32

Недостатки:

- Усложнение декодера.
- Все равно недостаточно экономно.
- Нужно учитывать endianness.
- Нет возможности проводить декодирование с произвольного байта (хотя возможно применения эвристических алгоритмов для выполнения этой задачи).

## UTF-8

Кодирование происходит следующим образом:




В самом левом столбце указано количество бит, которыми кодируется сам codepoint.
Количество единиц в префиксе первого байта равняется количеству байт, кодирующих данный codepoint (кроме случая, когда codepoint кодируется одним байтом). Префикс каждого байта составлен таким образом, чтобы по нему можно было однозначно определить, на первой позиции он находится или нет.

Преимущества:

- ❖ Не нужно учитывать endianness.
- ❖ Максимальная экономия памяти.
- ❖ Имеется возможность начинать декодирование с произвольного байта.
- ❖ Совместима с ASCII.

Недостатки:

- Декодер усложняется еще сильнее.
- Нет возможности найти codepoint по номеру его позиции в тексте.
- Нет возможности определить сколько codepoint'ов в тексте.

## Modifiers

Модификаторы - это codepoint'ы, которые каким-то образом изменяют отображения других codepoint'ов.
В их число входят, например, диалектические знаки: ˆ, ~, ´, ¯, ˘, `.

Возникает проблема:

| | | |
|---|---|---|
| е | ¨ | ж |

и

| | |
|---|---|
| ё | ж |

Строки то одни и те же, но при сравнение мы будем получать обратный результат.

Для того, чтобы правильно сравнивать подобные строки, необходимо выполнить процесс *нормализации*, то есть приведения текста к одной или другой форме.

# HTTP: общие принципы протокола, формат сообщений. URI. Заголовки Host, Content-Length, Content-Type, Content-Encoding, Transfer-Encoding, Accept, Range. Простые способы организации кеширования: Last-Modified и ETag. Cookies, аутентификация. Применение проксирования

HTTP - HyperText Transmission Protocol
HTTP - текстовый (использующий кодировку ASCII) протокол прикладного модели OSI, работающий по схеме “запрос/ответ” без сохранения состояния соединения.  По умолчания этот протокол работает на порте 80, чаще всего поверх TCP.

Неформальное описание схемы работы:
Клиент, желающий получить какой-то ресурс с сервера, посылают серверу сообщение-запрос, на которое сервер посылает сообщение-ответ, содержащее в случае успеха запрашиваемый ресурс. Часто запрашиваемым ресурсом является html-документ, после парсинга которого клиент понимает, что ему нужно запросить с сервера еще несколько ресурсов, например, несколько картинок.

При этом никакого состояния соединения сервер не сохраняет (для HTTP 1.x), то есть отправляются ли запросы в рамках одного соединения или в рамках нескольких, никак не влияет на выполнение самого запроса. На запросы может повлиять, например, порядок выполнения (например, метод GET  выполняется после метода PUT), но то, в рамках каких соединений выполняются эти запросы, никак не повлияет на результат.

## URI

RFC 3986
*Uniform Resource Identifier* - имя и адрес ресурса в сети, включает в себя URL и URN.

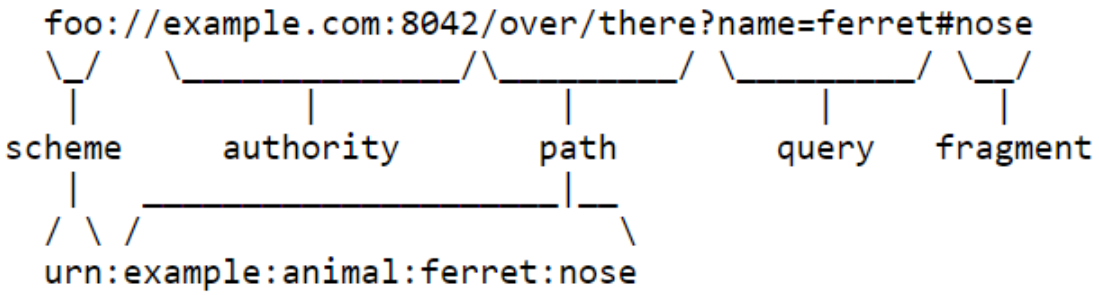
Шаблон:

```
URI         = scheme ":" hier-part [ "?" query ] [ "#" fragment ]

hier-part   = "//" authority path-abempty
            / path-absolute
            / path-rootless
            / path-empty
```

authority = [ userinfo "@" ] host [ ":" port ]

Два примера:




1. Схема - определяет метод доступа к ресурсу (например, http, https, ftp).
2. Authority может включать в себя информацию о пользователе, такую как, например, его имя и пароль. Host - это либо доменное имя, либо IP-адрес. Также authority может содержать порт, если порт не указан, то используется порт по умолчанию для заданной схемы (80 - для http, 443 - для https, 21 - для ftp).
3. Далее идет путь до ресурса.
4. Query - неиерархическая структура данных (обычно в виде пар “ключ=значение”, разделенных “&”), которая вместе с путем определяет ресурс (пример: search?q=cats).
5. Fragment - некоторая дополнительная информация для идентификации ресурса, например, глава книги (! никогда не передается на сервер, всегда обрабатывается клиентом).

*Uniform Resource Locator* - адрес ресурса в сети, определяет местонахождения и способ обращения к ресурсу. Грубо говоря, URL - это то, что идет до path в URI.
*Uniform Resource Name* - определяет только название ресурса. Грубо говоря, URN - это path в URI.

## Формат сообщений

Далее будет рассмотрен формат сообщений для версии HTTP 1.x, версия 2 привнесла незначительные изменения в этот формат, связанные с появлением кадрирования HTTP-сообщений (см. вопрос про сравнение HTTP 1.x и HTTP 2).

- В шаблонах запроса и ответа отличаются только первые строки.
    - В запросе указан метод (будут рассмотрены позже), целевой URI, а также запрашиваемая версия HTTP.
    - В ответе указана версия HTTP, код состояния (будут рассмотрены позже), а также текстовая интерпретация этого кода.
- Далее идут заголовки, которые позволяют передавать дополнительную информацию (заголовок имеет вид: имя заголовка:значение), пустая строка и тело сообщения (в том числе нулевой длины)

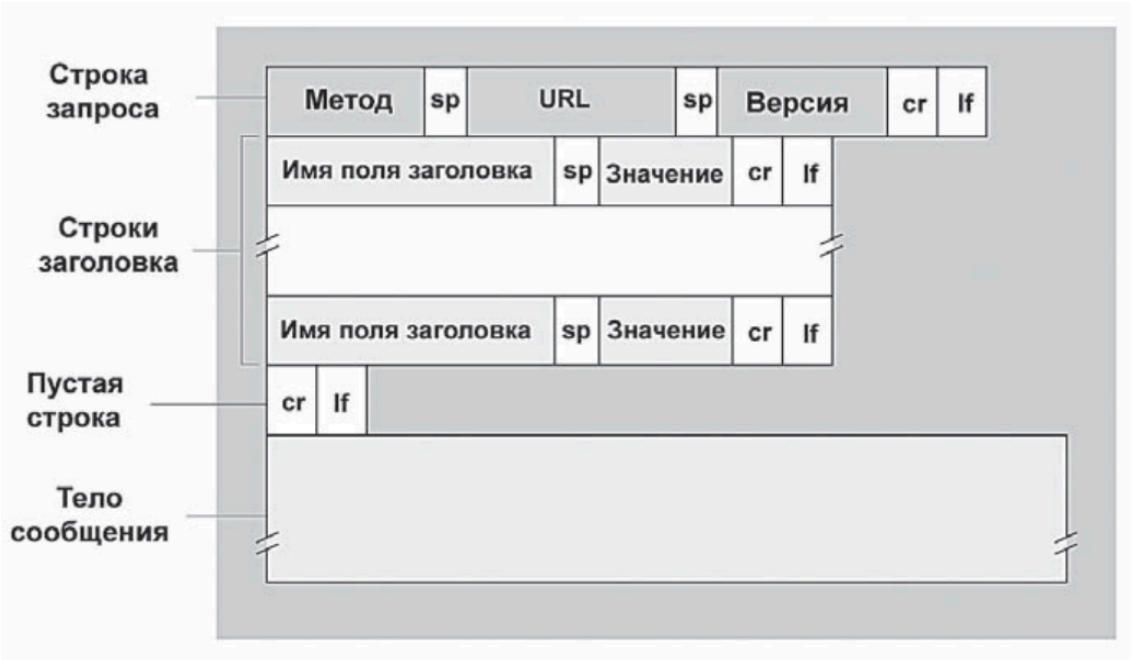



**Рис. 2.8.** Общий формат сообщения-запроса HTTP

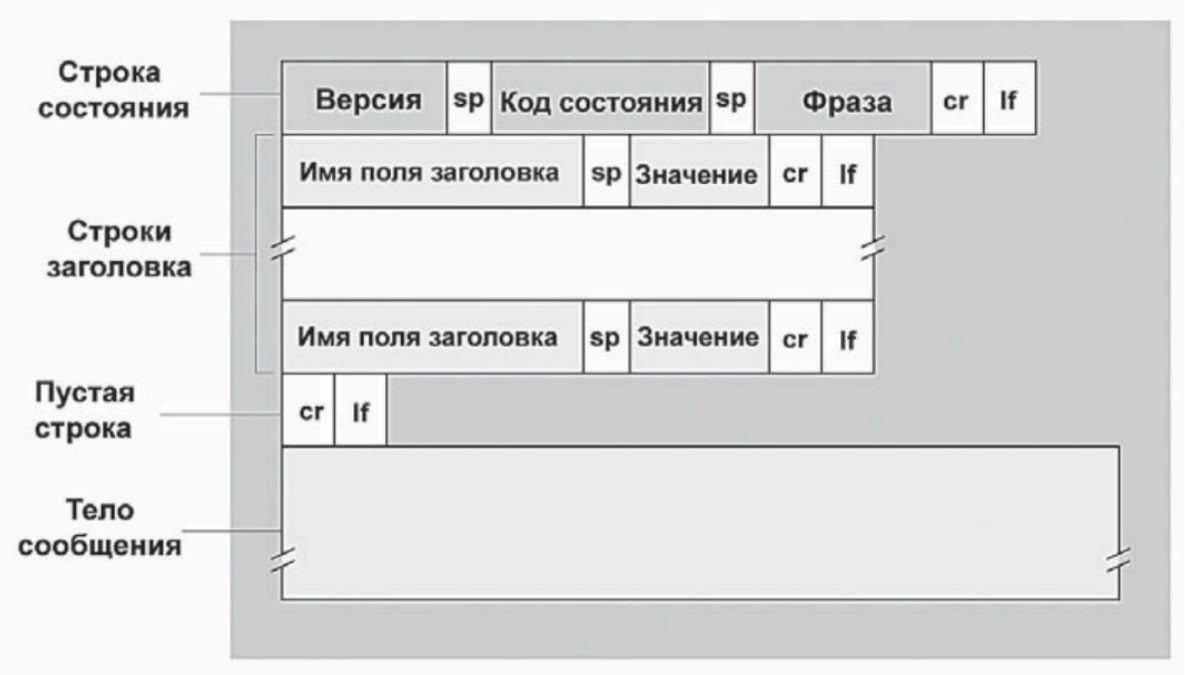



**Рис. 2.9.** Общий формат сообщения-ответа протокола HTTP

**Методы**

- GET - чаще всего используемый метод, применяется для получения ресурса (! запрос, используемый метод GET никак не должен изменять бизнес-состояние сервера, то есть по результатам запроса GET может быть сделана запись в логах или нечто подобное, но не более того, никакого изменения самого ресурса или настроек сервера)
- HEAD - метод, используемый для получения того же самого, что вернул бы ответ на запрос с методом GET только без тела ответа (может быть полезен при отправке).
- PUT - запрос на помещение ресурса на сервер (ресурс должен быть доступен по тому же адресу, по которому он помещается + сервер никак не должен изменить этот ресурс).
- DELETE - удалить ресурс.
- POST - модификация чего-либо на сервере.
- OPTIONS - запрос возможностей ресурса или сервера в целом (если в качестве Request URI указан “*”).

**Коды возврата**

- 1xx - информационные
- 2xx - успех
    - 200 (OK)

    - 201 (Created) - запрос был обработан, и новый ресурс был создан
    - 202 (Accepted) - запрос был принят, но еще не был обработан
- 3xx - переадресация
    - 301 (Moved Permanently)
    - 304 (Not Modified) - используется для кэширование (будет рассмотрено позже)
    - 307 (Temporary Redirect)
- 4xx - ошибка клиента
- 5xx - ошибка сервера

**Заголовки**

- **Host**. В HTTP 1.1 является обязательным для сообщения-запроса заголовком и содержит то же самое, что содержит authority после user-info (см. URI), то есть имя хоста и опционально порт. Нужен для того, чтобы идентифицировать хост на сервере, на котором несколько веб-сервисов расположены на одном IP-адресе, либо же при прохождении запроса через прокси-сервер. Если для запрашиваемой услуги не требуется имя хоста, тогда значение данного заголовка оставляется пустым.
- **Content-Length**. Содержит длину тела сообщения.
- **Content-Type**. Содержит тип нижележащий данных.
- **Transfer-Encoding : chunked**. Второй способ передачи информации о длине тела сообщения. Данный заголовок указывает на то, что данные будут передаваться кусками (chank'ами), в начале каждого куска будет указана его длина, когда данные закончатся будет отправлен кусок, длина которого 0.
- **Accept**. Указывает, данные какого типа клиент ожидает получить. Если для клиента нет разницы, то заголовок не указывается, если сервер не может отправить данные запрашиваемого типа, он должен отправить 406 (Not Acceptable).
- **Accept-Encoding**. Указывает, какое кодирование данных поддерживает клиент, например, gzip или compress. Если данный заголовок не указан, сервер может предположить, что клиент поддерживает все возможно виды кодирования. Если клиент не поддерживает никакого дополнительного кодирования данных, то значение данного заголовка должно быть установлено в "identity".
- **Content-Encoding**. Указывает на то, что к телу сообщения было применено дополнительное кодирование.

- **Range**. Указывает серверу, какую часть документа нужно вернуть.

## Кэширование

Есть несколько способ определения, когда ресурс устарел и нужно запросить его новую версию:

1. Одним из заголовков ответа от сервера может быть Last-Modified, значением которого является дата, когда запрашиваемый ресурс последний раз был модифицирован. После чего, когда клиенту вновь необходимо запросить тот же ресурс, он отправляет так называемый условный GET-запрос, одним из заголовков которого является If-Modified-Since, значением которого является в точности то, что было указано в ранее полученном от сервера заголовке Last-Modified. Сервер получает отвечает на этот запрос по разному в зависимости от того, когда ресурс последний раз был изменен. Если ресурс был изменен после времени, указанного в If-Modified-Since, это будет указывать на то, что в кэше клиента в данный момент находится устаревшая версия ресурса, и сервер должен отправить ресурс заново. В противном случае, в кэше клиента находится актуальная версия ресурса, сервер должен отправить 304 (Not Modified), и клиент получить ресурс из кэша.
2. Сервер может использовать в ответе заголовок ETag, идентифицирующий специфическую версию ресурса с помощью тэга, после чего клиент при запросе может передать серверу те тэги, которые помечают имеющиеся у него ресурсы, в заголовке If-None-Match и получить новую версию ресурса, если его версии устарели. Заголовок If-None-Match также используется, например, для операции PUT, чтобы перед ее выполнением проверить наличие ресурса на сервере.
3. С использованием Заголовка Cache-Control в ответе, в котором может быть указано, например, значение max-age, определяющее время, в течение которого эта версия ресурса остается валидной, либо там может быть указано значение no-cache, означающее, что данный ресурс вообще не должен кэшироваться.
4. С использованием заголовка Expires, с помощью которого сервер может сообщить в какой момент времени версия данного ресурса становится “несвежей”.

## Cookies


Cookie - это механизм, который появился в HTTP 2.
Допустим, что веб-сайту, который вы посещаете необходимо хранить какую-то информацию о вас (например, ваше имя или адрес вашего дома) или о вашей деятельности на этом сайте (к примеру хранить товары, которые вы добавили себе в корзину). Для этого и используются cookie. На сервере создается запись в базе данных, которая идентифицируется по специальному номеру - cookie, cookie возвращается клиенту с помощью заголовка Set-Cookie. Позже клиент может передать свой cookie, используя заголовок Cookie.

## Уведомления от сервера

Что делать, если клиент не посылает никаких запросов, а серверу нужно послать ему какое-то уведомление? Эта проблема решается двумя способами:

1. **Long polling**. В самом начале обмена сообщениями между клиентом и сервером клиент посылает запрос серверу, на который тот отвечает только в тот момент, когда нужно отправить какое-то уведомление. В HTTP 1.1 недостатком этого метода является то, что независимо от того, что пользователь делает в данный момент: активно работает с веб-сервисом (клиент активно взаимодействует с сервером) или же отошел на обед на несколько часов (клиент не взаимодействует с сервером), соединение будет открыто до тех пор, пока одна из сторон не разорвет соединение. В HTTP 2 такого недостатка нет, ведь единственном расходуемым ресурсом при использовании этого метода будет виртуальный канал.

2. Второй метод появился в HTTP 2 и называется **SSE** (Server-Sent Events). Он заключается в том, что клиент делает запрос на отдельный URL в виртуальном канале того же TCP-соединения, в рамках которого происходит основное взаимодействие. Сервер же при необходимости отправки уведомления посылает ответ по этому виртуальному каналу. Таким образом, клиент всегда будет ожидать получение очередного уведомления. Преимуществом данного метода по сравнению с предыдущим является то, что у клиента нет необходимости каждый раз при получении уведомления от сервера отправлять очередной запрос для ожидания следующего уведомления.

## Аутентификация


## Проксирование

Бывает два вида HTTP-прокси:

- **Обычный**. HTTP-прокси устанавливается на стороне клиента, принимает каждый запрос от него и пересылает серверу от своего имени. Применяется такая схема чаще всего для того, чтобы скрыть источник запроса.
- **Reverse**. HTTP-прокси устанавливается на стороне сервера. Делается это, например, для балансировки нагрузки между серверами, если их больше одного.

Оба вида прокси также применяются для кэширования.

# Применение симметричного и асимметричного шифрования, цифровой подписи. Принципы работы протокола TLS. Центры сертификации. Верификация принадлежности субъекта сертификации. Цепочки сертификатов. Отзыв сертификатов. Работа HTTP поверх TLS

Для обеспечения безопасности в сети нужно наличие *аутентификации* (то есть какого-то способа однозначной идентификации хоста), а также *шифрования*.

## Симметричное и асимметричное шифрование

Цель шифрования: защитить передачу данных между хостами от просмотра и/или модификации.

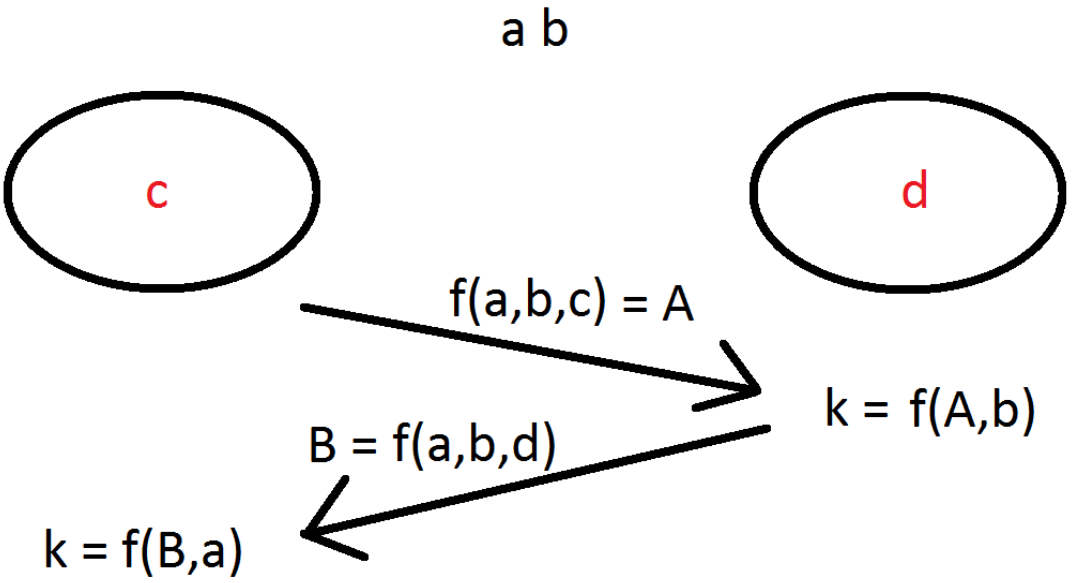
**Симметричное шифрование**

Есть некий ключ, который знают все участники обмена. Хост-отправитель шифрует отправляемые данные с помощью этого ключа и некой криптографической функции, отправляет зашифрованные данные хосту-получателю, а тот с помощью того же самого ключа дешифрует их и получает первоначальные данные. Обычно эти криптографические функции взаимно-однозначные, то есть размер изначальных и зашифрованных данных одинаковый.

Возникает вопрос: а как передать этот самый ключ? Тут есть несколько вариантов:

1. Передать их по другому (защищенному или незащищенному) каналу. Например, послать бумажной почтой, продиктовать по телефону или передать написанный на бумажке ключ лично в руки своему собеседнику.
2. С помощью алгоритма Diffie–Hellman key exchange, который позволяет передать ключ по незащищенному каналу. Неформальное описание этого алгоритма: есть два хоста, желающие получить ключ для симметричного шифрования. Есть два публичных числа: a и b, которые известны обоим хостам, а

также у каждого из хостов есть свое секретное число: c и d. На рисунке показан способ получения ключа k. Подробнее про данный алгоритм можно почитать на [википедии](#).




3. Использовать асимметричное шифрование для передачи ключа.

Примеры: Шифр Цезаря, шифр Вернера, AES, NaCl.

**Асимметричное шифрование**

У каждого хоста есть пара из публичного и приватного ключа. Публичный, как следует из названия, хранится в открытом доступе, приватный - есть только у хоста-владельца. Хост A, желающий отправить данные хосту B, шифрует их с помощью публичного ключа хоста B и отправляет. Хост B получает зашифрованные данные и расшифровывает их с помощью своего приватного ключа. Асимметричные способы шифрования работают намного медленнее симметричных, поэтому их использование при передачи данных больших объемов неоправданно, поэтому чаще всего используется схема, при которой ключ для симметричного шифрования, которое и будет использоваться для обмена данными, передается с помощью асимметричного.

## Web of trust, цифровые сертификаты, цифровые подписи

Симметричное и асимметричное шифрование, как следует из названия, обеспечивают шифрование передаваемых данных, но никак не

обеспечивают аутентификацию. То есть при отсутствии каких-то дополнительных методов нет возможности определить, что обмен происходит действительно с тем хостом, с которым мы хотим обмениваться данными. Злоумышленник может выдать себя за сервер, предоставить свой публичный ключ, а дальше получать и расшифровывать конфиденциальные данные клиента. Для обеспечения аутентификации есть несколько методов, мы рассмотрим два: web of trust и центры сертификации.

**Web of trust**

У хоста-источника есть список хостов, которым он доверяет. При получении у целевого хоста публичного ключа, он отправляет всем своим доверенным хостам запрос с целью получения информации о том, кто из них доверяет целевому хосту. При получении достаточного количества подтверждений хост-источник решает, что целевой хост это действительно тот, за кого себя выдает, добавляет его в свой список доверенных хостов, и начинается обмен данными.

**Цифровые подписи**

Цифровые подписи в отличие от симметричного и асимметричного шифрования решают проблему аутентификации. Как и при асимметричном шифровании, у сервера есть публичный и приватный ключи. Берутся какие-то данные и шифруются с помощью приватного ключа сервера. Обычно перед этим от этих данных вычисляется криптоустойчивый хэш, и только потом этот хэш шифруется с помощью приватного ключа, но это в данном случае не сильно важно. Зашифрованные таким образом данные - это и есть цифровая подпись. Потом клиент, желающий подтвердить подлинность публичного ключа, берет те же самые данные и сравнивает их с расшифрованной с помощью публичного ключа цифровой подписью, если данные оказываются одинаковыми, то публичному ключу можно доверять. Проблема возникает лишь с тем, как определить, можно ли доверять полученной цифровой подписи. Тут на сцену выходят цифровые сертификаты.

**Цифровые сертификаты**

Выдачей цифровых сертификатов занимаются *центры сертификации*, принадлежащие крупным компаниям, которым доверяют все (почти все) хосты, например, Google. В цифровой сертификат входят публичный ключ какого-то ресурса, цифровая подпись центра

сертификации, полученная с помощью кодирования самого этого сертификата, и еще много что, что сейчас не имеет особого значения.
Пусть хост A желает обмениваться данными с хостом B, предварительно проверив его публичный ключ у центра сертификации C. Хост A запрашивает публичный ключ у хоста B и сертификат у центра сертификации или хоста B (в TLS используется второй вариант). Цифровая подпись из сертификата дешифруется с помощью публичного ключа C и сравнивается с самим сертификатом, если они совпадают, а также совпадают публичный ключ, предоставленный хостом B, и публичный ключ из сертификата, то хост B тот, за кого себя выдает, и можно начинать обмен данными.
На самом деле понятно, что еще в сертификате обязательно должно присутствовать доменное имя хоста B, для того, чтобы идентифицировать принадлежность сертификата хосту B.

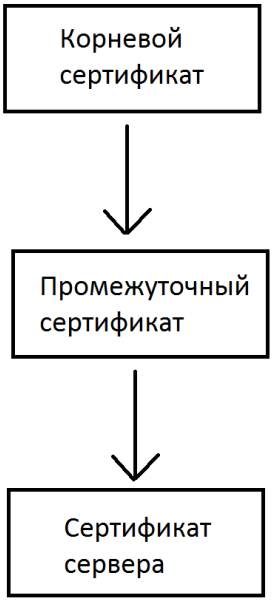
Как зарегистрировать свой собственный сертификат?
Нужно обратиться к центру сертификации, который попросит вас подтвердить, что вы владелец сервера, для которого делаете сертификат. Это может делаться несколькими способами, вот некоторые из них:

- Сервер может попросить разместить на сервере определенный файл.
- Сервер может попросить разместить в вашей доменной зоне TXT запись с какой-нибудь фиксированной строкой.
- В системе whois для вашего доменного имени может быть указан email владельца, в таком случае подтверждение происходит с помощью отправки письма на этот Email.

Классы сертификатов
На самом деле, сертификаты бывают разных классов. Вышеописанный сертификат - это сертификат самого нижнего уровня, его получение самое дешевое, так как подтверждение о владении сервером может быть произведено автоматически.
Сертификат более высокого уровня содержит название организации, которой сервер принадлежит. Такой сертификат уже стоит значительно больше денег, так как для его создания требуется проверка документов и т.д., что выполняется человеком.

Иерархия сертификатов

На самом деле, ранее описанная модель получения сертификата немного упрощена.  До этого мы представляли, что цифровая подпись в сертификате сервера сделана с помощью приватного ключа корневого сертификата, но на самом деле, присутствует еще промежуточный сертификат.




Стрелка на данной картинке означает, что цифровая подпись в сертификате, в который стрелка входит, сделан с помощью приватного ключа сертификата, из которого стрелка выходит. Цифровая подпись корневого сертификата сделана с помощью его же приватного ключа. Сделано так по нескольким причинам:

- В угоду безопасности. Корневой сертификат вообще обычно хранится на защищенном физическом носителе, который не подключен ни к одному вычислительному устройству, так как его компрометация вызовет большие проблемы: очень многие сертификаты станут компротированными. Использовать его каждый раз, когда необходимо было выдать сертификат какому-то серверу, ставило бы его безопасность под угрозу.
- В случае, когда появляется новый центр сертификации, которому еще не все доверяют, его промежуточный сертификат также может подписать старый доверенный центр сертификации.

## TLS

Представим себе, что хост A хочет подключиться к серверу B по TLS, для этого происходят следующие действия:

- Установление TCP-соединения, TCP-handshake.

- от A к B: набор алгоритмов, которые A желает использовать для TLS-соединения: алгоритмы хэширования, симметричного и асимметричного шифрования, обмена ключами и вычисления цифровой подписи.
- B выбирает алгоритмы из тех, которые прислал ему A, и посылает их хосту A вместе со своим цифровым сертификатом и возможно запросом цифрового сертификата A (обычно это используется только в условиях повышенной безопасности).
- A посылает свой сертификат и в этом же сообщении может начать обмен ключами (может, потому что для обмена данными может использоваться уже известный обеим сторонам ключ, тогда этот этап пропускается).

Так было в TLS 1.2. В TLS 1.3 рукопожатие происходит быстрее, так как рекомендованный способов обмена ключами всего один (с помощью алгоритма Диффи-Хеллмана), соответственно, о нем не нужно договариваться. Поэтому клиент в первом же сообщении начинает обмен ключами, и весь handshake проходит за один RTT.
Возможность использования заранее переданных (pre-shared) ключей в TLS 1.3 никуда не делась, в таком случае в качестве процедуры обмена ключами выступает обмен идентификаторами этих ключей.

**Восстановление соединения**

На самом деле, еще до TLS 1.3 был способ оптимизации процесса установления соединения. Этот способ называется восстановлением соединения и заключается он в том, что на сервере сохранялся ключ, с помощью которого происходил обмен данными с клиентом, который мог быть использован при повторном установлении соединения, благодаря чему не было нужды повторно выполнять процедуру обмена ключами. При установлении первого соединения во время первого RTT сервер посылает клиенту 32-разрядное число, идентификатор сессии, которое клиент при повторном подключении может указать в своем первом сообщении для того, чтобы сервер по нему нашел ключ.
Тут возникает проблема: представим себе сервер очень популярного веб-сервиса, к нему пытаются подключиться тысячи, а то и миллионы клиентов за секунду, вряд ли он будет своевременно справляться с поиском ключей для каждого клиента. Для решения этой проблемы используется session ticket - зашифрованные приватным ключом сервера параметры сессии, которые сервер отправляет клиенту во время handshake, а тот пересылает при установлении повторного соединения,

таким образом, необходимость сохранять эту информацию на сервере отпадает.

## Отзыв сертификата

Если сертификат был скомпрометирован, то центр сертификации его отзывает. Клиентские приложения при каждом запросе (редко), время от времени или никогда (редко) делают запрос к центру сертификации с целью проверки отозванных сертификатов. В обычном случае, когда проверки происходят время от времени, вероятность попасть на скомпрометированный веб-сайт есть.

## Скомпрометированный центр сертификации

Компрометация центра сертификации - это серьезная проблема, для ликвидации ее последствий был разработан новый вид DNS записи: CAA (Certification Authority Authorization), в которой указаны центры сертификации, которые имеют право выдавать данному доменному имени сертификаты (если такой записи нет, то считается, что все центры сертификации могут выдавать сертификат данному доменному имени). Важно понимать, что данная мера предосторожности будет работать только в том случае, если клиент при получении сертификата будет проверять, совпадает ли центр сертификации, который его выдал с центром сертификации, указанным в CAA.