

ОТВЕТЫ ДЛЯ ЭКЗАМЕНА ПО
ДИСЦИПЛИНЕ
“СЕТЕВЫЕ ТЕХНОЛОГИИ И
ТЕЛЕКОММУНИКАЦИИ”

Сделано: Евдокимовой Дарьей

По всем опечаткам: [сюда](#).

Инфа преимущественно взята из лекций Ипполитова В.Д, тырнета и что-то из общего файла.

Список лекций с cs-центра [на ютубчике](#).

Вопросы на 2 балла

1. [Возможности протоколов TCP и UDP, преимущества и недостатки. Применимость в приложениях.](#)
2. [Принципы работы системы DNS. Использование DNS для целей обеспечения работы других прикладных протоколов \(с примерами\).](#)
3. [Способы удалённого доступа к файлам. Принципы работы протоколов FTP, SMB, NFS. "Облачное" хранение файлов \(Dropbox, etc\).](#)
4. [Форматы хранения и передачи данных. Json, XML, protobuf. Преимущества и недостатки.](#)
5. [Сравнение HTTP 1.x, HTTP 2 и HTTP 3. Преимущества и недостатки.](#)
6. [Email: принципы функционирования, протоколы SMTP, POP3, IMAP. Вопросы безопасности: подделка адресов, модификация сообщений, спам. Технологии SPF, DKIM, DMARC.](#)

Вопросы на 3 балла

1. [Работа TCP: установка и завершение соединения, передача данных и подтверждение доставки, контроль размера сегмента и размера окна.](#)
2. [Однобайтовые и многобайтовые кодировки, их применимость, преимущества и недостатки. Юникод. Устройство кодировок UTF-8, UTF-16, UTF-32.](#)
3. [HTTP: общие принципы протокола, формат сообщений. URI. Заголовки Host, Content-Length, Content-Type, Content-Encoding, Transfer-Encoding, Accept, Range. Простые способы организации кэширования: Last-Modified и ETag. Cookies, аутентификация. Применение проксирования.](#)
4. [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.](#)

Дополнительная инфа

Полезные ссылки

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

Транспортный уровень отвечает за то, чтобы доставить данные от одного приложения на одном устройстве к другому приложению на другом устройстве.

Сетевой уровень доставляет данные от одного устройства до другого, которые, возможно, в разных сетях находятся.

Протоколы UDP and TCP относятся к **транспортному** уровню.

UDP - User Datagram Protocol

В udp есть порт и контрольная сумма. Можно сказать, что udp - тонкая обертка над протоколом сетевого уровня - ip. Если какой-то пакет потерялся, то мы об этом никак не узнаем. Будет ограничение на размер пакета (не сможем отправить больше 2^{16} бит).

MTU - Maximum Transmission Unit - максимальный размер блока данных, который можно передать.

В IPv4 есть 2 варианта: если маршрутизатор не хочет заниматься маршрутизацией, то он пакет может выбросить, и может опционально отправить ICMP отправителю ответ о том, что пакет был выброшен по причине его большого размера. Если маршрутизатор поддерживает фрагментацию, то тогда он может пакет разделить на несколько кадров.

Проблема фрагментации в том, что возрастаёт нагрузка на маршрутизатор. Т.е помимо маршрутизации, они теперь еще и должны заниматься "разрезанием" данных.

UDP может делать все вещи, которые делает и IP. То есть может делать мультикасты, бродкасты. Его преимущество в том, что он простой и быстрый: никаких обработок не происходит. Поэтому на транспортном уровне никаких дополнительных задержек при его использовании не вносится.

Заголовок UDP для IPv4

| Биты | 0 — 7 | 8 — 15 | 16 — 23 | 24 — 31 |
|------|------------------|----------|-------------------|---------|
| 0 | Адрес источника | | | |
| 32 | Адрес получателя | | | |
| 64 | Нули | Протокол | Длина UDP | |
| 96 | Порт источника | | Порт получателя | |
| 128 | Длина | | Контрольная сумма | |
| 160+ | Данные | | | |

TCP - Transmission Control Protocol

Самый популярный протокол транспортного уровня. Контролирует доставку пакетов.

** просто инфа пусть будет: протокол SMTP - протокол уровня приложения, работает поверх TCP. HTTP работает тоже поверх TCP **

Заголовок TCP

Заголовок сегмента TCP [[править](#) | [править код](#)]

Структура заголовка

| Бит | 0 — 3 | 4 — 6 | 7 — 15 | 16 — 31 |
|----------|--|-----------------|--------|--|
| 0 | Порт источника, Source Port | | | Порт назначения, Destination Port |
| 32 | Порядковый номер, Sequence Number (SN) | | | |
| 64 | Номер подтверждения, Acknowledgment Number (ACK SN) | | | |
| 96 | Длина заголовка, (Data offset) | Зарезервировано | Флаги | Размер Окна, Window size |
| 128 | Контрольная сумма, Checksum | | | Указатель важности, Urgent Point |
| 160 | Опции (необязательное, но используется практически всегда) | | | |
| 160/192+ | Данные | | | |

TCP контролирует доставку пакетов. Если данные вдруг не были доставлены до получателя, то TCP их отправляет заново. До тех пор, пока они не доставятся.

TCP контролирует порядок. Это значит, что нам не надо думать о том, в каком порядке придут пакеты. TCP основан на IP, поэтому в итоге все данные, которые отдаем TCP, он отправляет с помощью IP: все данные нарезает на какие-то фрагменты, кладет в IP пакеты и потом они путешествуют. TCP переставит пакеты в нужном порядке САМ. Здесь TCP гарантирует, что никакие байты, которые мы отправляли, не перемешаются.

TCP - это сессионный протокол (в том числе). В случае UDP нет никакой сессии между отправителем и получателем: любой отправитель в любой момент может отправить любой пакет любому получателю. И этот пакет может дойдет, а может и нет.

А чтобы общаться по TCP, сначала нужно договориться о соединении. И чтобы соединение завершить, надо тоже договориться о том, что соединение будет завершено, что данные передаваться больше не будет.

TCP - потоковый протокол.

Это значит, что мы оперируем не какими-то фрагментами, что вот мы хотим отправить 10 байтов, фигак-фигак, отправили их и получатель и получил ровно эти 10 байтов. То есть в случае UDP пакет является целостной сущностью транспортного протокола.

В случае TCP сущностью является не пакет, а поток. То есть мы устанавливаем соединение и говорим, что вот, начался поток байтов.

Потенциально, этот поток может быть бесконечным, нет никакого ограничения на его размер. В реальной жизни это не так, тк обязательно что-то пойдет не так: приложение умрёт, кабель перережут, связь нарушится и тд.

Короче говоря, приложение оперирует только потоком.

TCP - это протокол “один-к-одному”. У нас есть участник-приложение и еще один участник, который может расположен в другом приложении. И они между собой связываются. То есть нельзя сделать ни мультикаст, ни бродкаст. Потому что как это реализовывать, как обеспечить контроль доставки, если мы даже не знаем, кто есть в группе. TCP этого не умеет.

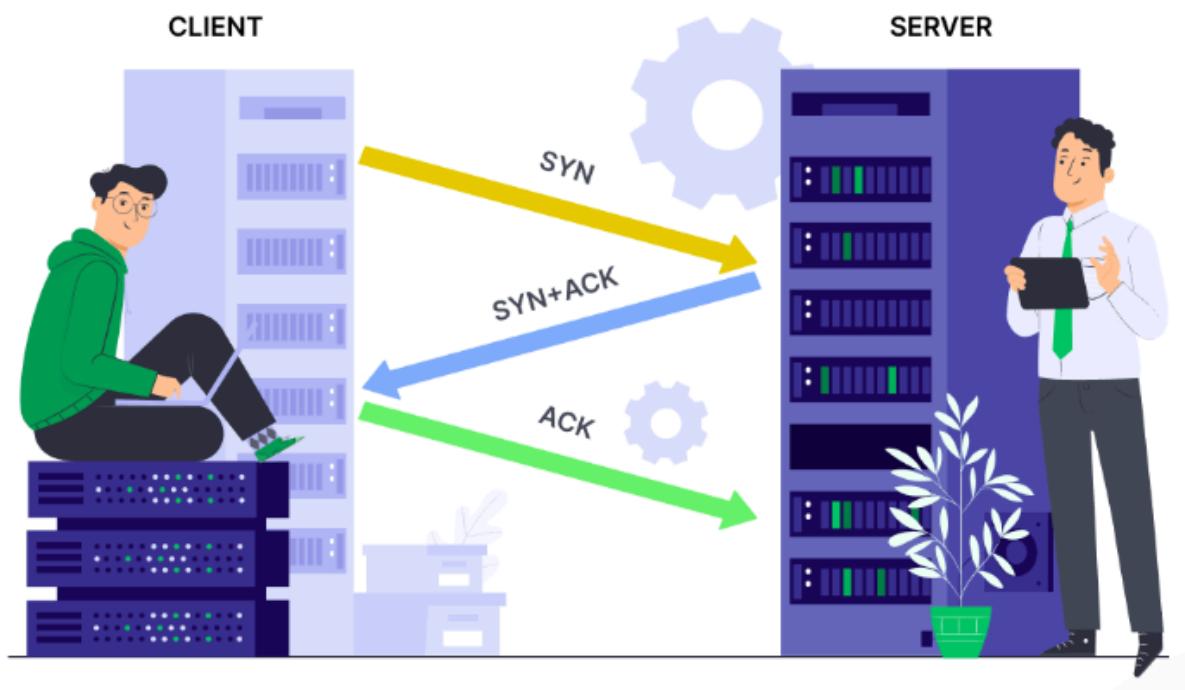
В TCP есть стороны, которые общаются, они имеют разные роли. В UDP роли одинаковые: любой участник может отправлять любому участнику что угодно; от любого, что угодно получить.

В TCP есть активная сторона (та, что устанавливает соединение) и пассивная (та, что сидит и ждет, пока к ней кто-нибудь подключится).

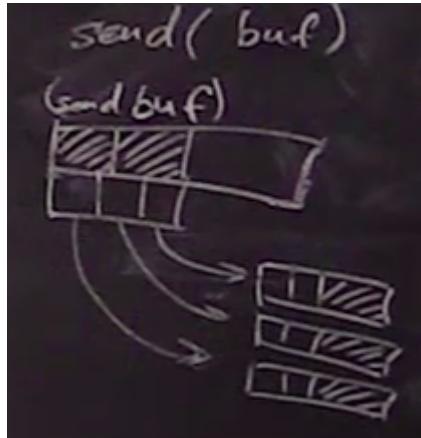
Пример: в http браузер - активная сторона, потому что будет пытаться установить соединение с каким-то сервером. А сервер - будет пассивной стороной, тк будет ждать, пока к нему придут клиенты, которых он их обслужит.

Тройное рукопожатие aka TCP three-way/triple handshake
Подробнее почитать [здесь](#).

| Схема тройного рукопожатия | | |
|------------------------------------|--------------------------------------|--|
| активная сторона (отправитель) | | пассивная сторона (получатель) |
| инициирует установку соединения | -----> (запрос на подключение) | |
| | <----- | если готова общаться, то отправляет ответ |
| договорились, будем общаться | -----> | |
| соединение установлено | | |



Можно рассказать более детально про то, как происходит отправка.

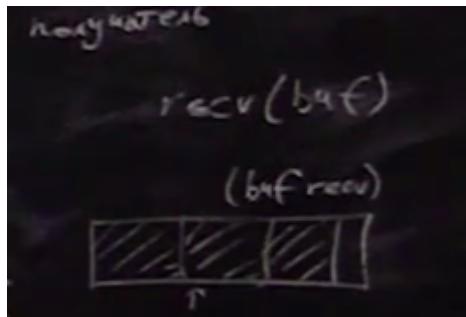


Есть какое-то приложение, которое говорит “давай че-нить отправим” и передает какой-то буфер с данными. То есть приложение хоть и оперирует с потоком, но оперирует с ним кусочками.

Операционная система складывает это сообщение в какой-то буфер на отправку (send buffer).

В какой-то момент ОС принимает решение о том, что сообщение пора отправлять, и тогда она нарезает все, что есть, на фрагменты нужного размера. То есть каким-то образом реализация TCP может узнавать (или угадывать всякими алгоритмами подбора), какое MTU на нашем маршруте. И в соответствии с MTU ОС нарезает на фрагменты (на 3 куска, например). Каждому добавляет заголовки, кладет туда кусочки

данных и отправляет. В каждом таком пакете написан номер байта, который будет первым в потоке.



У получателя есть буфер для получения. Он получает эти пакеты, смотрит, какой номер в потоке у получаемого пакета и кладет его в нужное место. Допустим, пришли все пакеты, и он их положил в свой буфер.

После этого приложение обращается к ОС и говорит “а что-нить мне прислали в рамках этого соединения?” и вызывает функцию recv() - сискол. И говорит “положи мне в этот буфер все то, что ты получил”. ОС смотрит, если че-то есть в буфере, то копирует в буфер приложения.



Допустим, пришли не все пакеты. Как узнаем, какой кусок не пришел? По номеру первого байта, который определяет сдвиг “разрезания” передаваемых данных. Получатель замечает, что образовалась дырка. Тогда получатель говорит отправителю: “вот дырка, дай мне данные”. Отправитель берет и отправляет нужные данные. Но! перед этим заново происходит нарезка данных. А не так, что вот с такого-то бита берем и отправляем данные, которые не дошли до получателя. А почему так? Почему нарезка происходит заново? Почему не запомнить, что вот с такого-то бита все норм отправилось, а с такого-то - нет. И просто переотправить нужный кусок. Ответ. Потому что ситуация может меняться. TCP может узнать о том, что на самом деле MTU, в соответствии с которым он сделал нарезки, не такой, а меньше, потому что пакет мог потеряться, потому что не пролез по какой-то линии связи. И тогда TCP может сделать вывод о том, что mtu надо уменьшить.

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

Про DNS

Domain Name System - штука, которая позволяет не запоминать интернет-адреса. Запоминать адреса - это фу. Адреса могут меняться, адреса привязаны к хостинг-провайдеру. Если поменяем провайдера, то адреса поменяются. Так вот чтобы сопоставить имена и IP адреса, существует DNS.

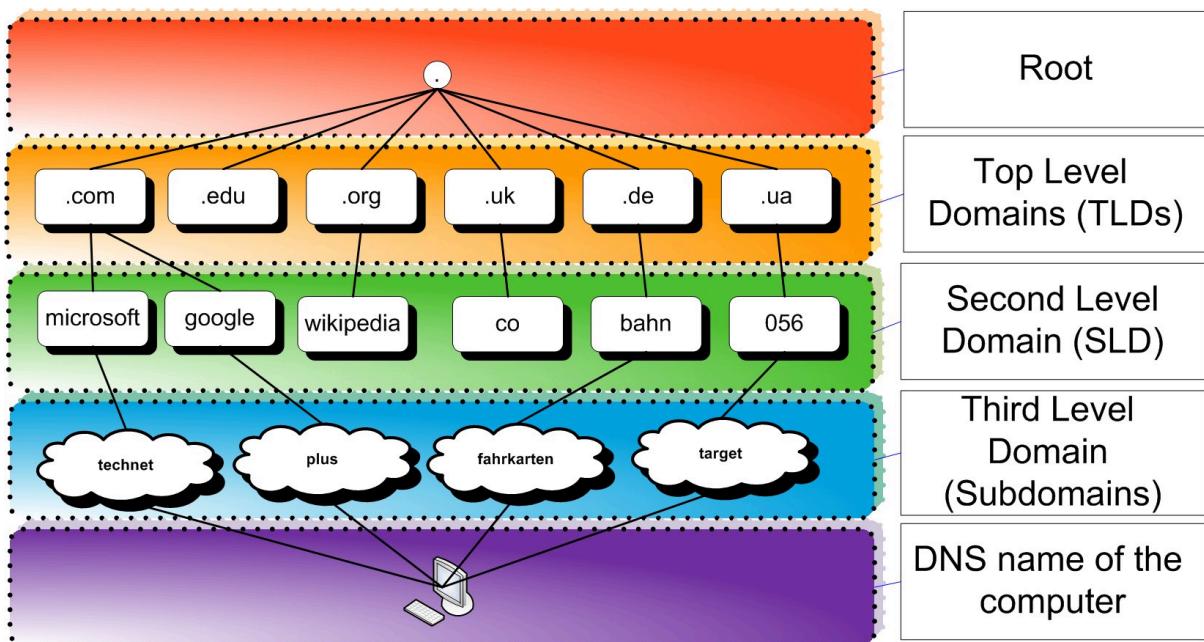
Каждое имя задает некое пространство.

| | | |
|-----------------------|----------------------|---|
| www. | google. | .com. |
| домен третьего уровня | домен второго уровня | домен первого уровня |
| | | это имя задает некое пространство, т.е. слева от него может быть все, что угодно, что принадлежит этому пространству .com |

Есть домены, принадлежащие странам: .ru, .uk, .su и тд

| domain name = com. | | |
|--------------------|------------------|----------------|
| name | type | content |
| google | NS = Name Server | ns1.google.com |
| google | NS | ns3.google.com |
| apple | NS | |

Структура DNS



Пример

| domain name = google.com. | | |
|---------------------------|--------------------------------------|----------------|
| name | type | content |
| www | A | 5.4.5.6 |
| www | A | 5.4.5.7 |
| www | AAAA | 240f: :05 |
| web | CNAME = canonical name | www.google.com |
| @ | MX создается для имени самой зоны | - |
| @ | A | 5.4.5.6 |
| @ | NS | ns1.google.com |
| @ | NS | ns2.google.com |

Для одного и того же имени может существовать несколько адресов. Это сделано для отказоустойчивости.

Для адреса IPv6 type = AAAA.

Чтобы вносить изменения в зону **.com**, требуется передать запрос организации, которая ей руководит. Читать [тут](#).

Инфа с сайтика

Система доменных имен состоит из следующих компонентов:

Иерархическая структура доменных имен:

- Доменные зоны верхнего уровня (первого уровня) – например: "ru", "com", или "org". Они включают в себя все доменные имена, входящие в эту зону. В любую доменную зону может входить неограниченное количество доменов.
- Доменные имена (доменные зоны второго уровня) – например: "google.com" или "yandex.ru". Т.к. система доменных имен является иерархичной, то "yandex.ru" можно также называть поддоменом вышестоящей зоны "ru". Поэтому, правильнее указывать именно уровень домена. Однако, на практике, доменную зону любого уровня называют просто «доменом».
- Поддомены (доменные зоны третьего уровня) – например: "api.google.com" или "mail.yandex.ru". Могут быть доменные зоны 4, 5 уровней и так далее.

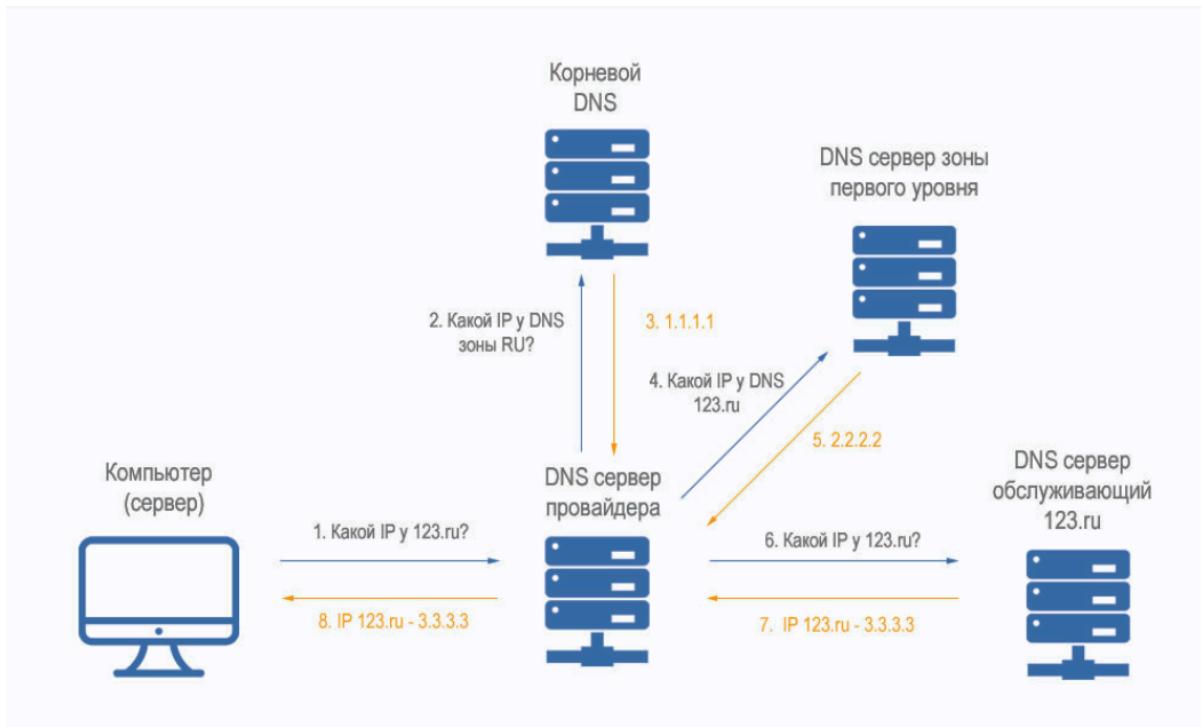
Обратите внимание, что "www.google.com" и "google.com" - это, фактически, разные домены. Надо не забывать указывать А-записи для каждого из них.

- **DNS сервер или NS (name server) сервер** – поддерживает (обслуживает) доменные зоны, которые ему делегированы. Он непосредственно хранит данные о ресурсных записях для зоны. Например, что сервер, на котором находится сайт "example.ru", имеет IP адрес "1.1.1.1". DNS сервер отвечает на все запросы, касательной этих доменных зон. Если ему приходит запрос о домене, который ему не делегирован, то он спрашивает ответ у других DNS серверов.
- **DNS записи (ресурсные записи)** – это набор записей о доменной зоне на NS сервере, которые хранят данные необходимые для работы DNS. На основании данных в этих записях, DNS сервер отвечает на запросы по домену. Список записей, и их значение, вы можете найти ниже.

- Корневые DNS сервера (на данный момент их 13 во всем мире) хранят данные о том, какие DNS сервера обслуживают зоны верхнего уровня.
- DNS сервера доменных зон верхнего уровня - хранят информацию, какие NS сервера обслуживают тот или иной домен.

Как работает DNS?

Для того, чтобы узнать IP адрес, домена компьютер / сервер обращается к DNS-серверу, который указан у него в сетевых настройках. Обычно, это DNS сервер Интернет провайдера. DNS сервер проверяет делегирован домен ему или нет. Если да, то сразу отвечает на запрос. Если нет, то запрашивает информацию о DNS сервере, обслуживающем этот домен, у корневого сервера, и затем у сервера доменных зон верхнего уровня. После этого, непосредственно делает запрос на NS сервер, обслуживающий этот домен, и транслирует ответ вашему компьютеру / серверу.



Кэширование данных используется на всех устройствах (компьютерах, северах, DNS серверах). То есть, они запоминают ответы на последние пришедшие к ним запросы. И когда приходит аналогичный запрос, они просто отвечают то же самое, что и в предыдущий раз. Например, если вы в браузере открыли сайт google.com первый раз после включения, то компьютер сделает DNS запрос, а при последующих запросах будет брать данные, которые ему были присланы DNS сервером в первый раз.

Таким образом, для популярных запросов не надо каждый раз проходить всю цепочку и генерировать запросы к NS серверам. Это значительно снижает нагрузку на них, и увеличивает скорость работы. Однако, как результат, обновление данных в системе DNS происходит не сразу. При изменении IP адреса домена, информацию об этом будет расходиться по сети Интернет от 1 до 24 часов.

Основные DNS записи

Существуют следующие основные DNS (ресурсные) записи:

A – содержит информацию об IPv4 адресе хоста (сервера) для домена.

Например, 1.1.1.1.

- AAA – содержит информацию об IPv6 адресе хоста (сервера) для домена. Например, 2001:0db8:11a3:09d7:1f34:8a2e:07a0:765d.
- MX – содержит данные о почтовом сервере домена. При этом указывается именно имя почтового сервера, например mail.example.com. Т.к. у домена может быть несколько почтовых серверов, то для каждого из них указывает приоритет. Приоритет задается числом от 0 до 65535. При этом «0» - это самый высокий приоритет. Принято по умолчанию для первого почтового сервера указывать приоритет «10».
- TXT – дополнительная информация о домене в виде произвольного текста. Максимальная длина 255 символов.
- SRV – содержит информацию об имени хоста и номере порта, для определенных служб / протоколов в соответствии с RFC 2782 <http://www.rfc-editor.org/rfc/rfc2782.txt>. Содержит следующие поля:
 - _Service._Proto.Name (Пример: _jabber._tcp.jabber), где:
 - Service: название службы (пример: ldap, kerberos, gc и другие).
 - Proto: протокол, при помощи которого клиенты могут подключиться к данной службе (пример: tcp, udp).
 - Name: имя домена, в котором размещена данная служба.
 - Приоритет – также как для MX записи указывает приоритет для данного сервера. Задается числом от 0 до 65535. При этом «0» - это самый высокий приоритет.
 - Вес – Относительный вес для распределения нагрузки между серверами с одинаковым приоритетом. Задается целым числом.
 - Порт – номер порта, на котором располагается служба на данном сервере.
 - Назначение - доменное имя сервера, предоставляющего данную службу.

- NS – имя DNS сервера, поддерживающего данный домен.
- CNAME (каноническое имя хоста / canonical name) – используется для перенаправления на другое доменное имя. Например, имя сервера изменилось с example.com на new.com. В таком случае в поле «Aliases» для записи cname надо указать - example.com, а в поле «Canonical name» - new.com. Таким образом, все запросы на example.com автоматически будут перенаправлены на new.com.
- SOA – базовая запись о домене. В ней хранится само имя домена и время жизни данных о домене - TTL. TTL (time-to-live) определяет какой период времени DNS сервер получив информацию о зоне будет хранить ее у себя в памяти (кэшировать). Рекомендуемое значение 86400 – 1 день. Значение указывается в секундах.

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

взято из [12й лекции](#)

Протокол FTP

File Transfer Protocol. Протокол работает поверх TSP. Используется 2 вида tsp-соединений: одно соединение - управляющее, через которое передаются команды, что нужно делать. А другое соединение - для передачи данных, в которых будут передаваться содержимое. По FTP передаются данные через 21 порт.

| client | | server |
|--------|--|--------|
| | (control) —> auth Можно задать свой логин и пароль (что не оч безопасно, тк пароль передается в чистом виде, без какой-либо обработки). | |
| | FWD cd <dir> mkdir <dir> chmod <> | |
| | ls fetch <file> upload <file> | |
| | (data) <-----TCP-----> | |

Когда клиент устанавливает соединение для управления, он выбирает какой-то порт свободный. И сообщает номер порта серверу.

Взято [отсюда](#).

Чтобы удостовериться, что клиент может скачивать файлы, при подключении сервер просит ввести логин и пароль. Но это не всегда обязательно — FTP поддерживает и анонимный режим, когда подключиться к серверу можно без авторизации.

Пример применения FTP

Давайте посмотрим, как работает FTP на примере жизненной ситуации. Допустим, вы работаете программистом в IT-компании. Весь код и все файлы хранятся в локальном хранилище — на внутреннем сервере, который доступен только сотрудникам. Чтобы начать работу, нужно перенести рабочие файлы себе на компьютер, а для этого — подключиться к серверу.

Задача понятна — открываем консоль и устанавливаем соединение через FTP для доступа к хранилищу. Перед этим нужно запросить логин и пароль от него, иначе не удастся подключиться. Как только вы авторизуетесь, вам будут доступны все файлы на сервере — или те, к которым вам выдадут доступ. Чтобы скачать их, нужно ввести ещё пару команд и дождаться загрузки. Готово!

Чем FTP отличается от HTTP

В интернете есть два популярных протокола для передачи данных: FTP и HTTP. Оба решают примерно одинаковые задачи, но всё же различаются в назначении.

- FTP изначально создавался для обмена данными между компьютерами. Он использует два соединения: одно для передачи, а другое — для управления. Идея в том, что FTP умеет следить, как выполняются длинные запросы — например, передача больших файлов или управление данными на сервере.
- HTTP заточен на передачу гипертекстовых документов — то есть сайтов и веб-страниц. Он использует всего одно соединение, по которому запросы летают между клиентом и сервером. Ему не нужно устраивать длинные сеансы обмена данными: просто отдал нужную страничку и забыл.

Иначе говоря, FTP ориентирован на долгое и «вдумчивое» взаимодействие сервера и клиента: он помнит, кто, кому, когда и что передал. У HTTP другая задача — управлять запросами в интернете. В отличие от FTP, он не хранит состояние сессии, зато работает быстрее — а это как раз то, что нужно для веба.

Плюсы и минусы FTP

Вот три главные задачи, для решения которых вам может понадобиться FTP:

- Передавать файлы на другие компьютеры. Это удобно, когда вам нужно, например, загрузить файлы на сервер сайта или

отправить видео другу. Просто подключились, отправили, остались довольны.

- Резервное копирование. Чтобы не потерять файлы, вы можете сделать их бэкап на удалённый компьютер.
- Удалённый доступ к файлам. FTP можно превратить в замену Google Drive или «Яндекс Диска»: берём удалённый компьютер, загружаем туда файлы и достаём по мере необходимости.

Теперь о том, почему использование FTP может быть не самой удачной идеей:

- Отсутствие адекватной защиты. Когда вы подключаетесь к серверу, данные отправляются по сети в незашифрованном виде: в том числе логины, пароли и сами файлы. Плюс в FTP нет никакой защиты от [брутфорса](#) — то есть пароль к серверу можно подобрать обычным перебором.
- Неэффективная передача файлов. Для каждой операции передачи FTP требует установки нового соединения. Если вы хотите передавать несколько файлов одновременно, это может привести к задержкам и увеличению нагрузки на сеть — особенно при передаче больших файлов.
- Разрыв соединения при ошибке. Если передача данных прервётся по какой-то причине, то весь процесс придётся начинать сначала.

Протокол SMB

SMB (сокр. от [англ.](#) Server Message Block) — [сетевой протокол прикладного уровня](#) для удалённого доступа к [файлам, принтерам](#) и другим сетевым ресурсам, а также для [межпроцессного взаимодействия](#). Первая версия протокола, также известная как Common Internet File System (CIFS) (Единая файловая система Интернета), была разработана компаниями [IBM](#), [Microsoft](#), [Intel](#) и [3Com](#) в 1980-х годах; вторая (SMB 2.0) была создана Microsoft и появилась в [Windows Vista](#). В настоящее время SMB связан главным образом с операционными системами [Microsoft Windows](#), где используется для реализации «Сети Microsoft Windows» ([англ.](#) Microsoft Windows Network) и «Совместного использования файлов и принтеров» ([англ.](#) File and Printer Sharing).

Как работает SMB?

SMB — это протокол, основанный на технологии [клиент-сервер](#), который предоставляет клиентским приложениям простой способ для чтения и записи файлов, а также запроса служб у серверных программ в

различных типах сетевого окружения. Серверы предоставляют файловые системы и другие ресурсы (принтеры, почтовые сегменты, именованные каналы и т. д.) для общего доступа в сети. Клиентские компьютеры могут иметь у себя свои носители информации, но также имеют доступ к ресурсам, предоставленным сервером для общего пользования.

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. Однако в случае использования SMB эти действия совершаются через сеть.

Как было сказано выше, SMB работает, используя различные [протоколы](#). В [сетевой модели OSI](#) протокол SMB используется как протокол Application/Presentation уровня и зависит от низкоуровневых транспортных протоколов.

С начала существования SMB было разработано множество различных вариантов протокола для обработки всё возрастающей сложности компьютерной среды, в которой он использовался. Договорились, что реальный вариант протокола, который будет использоваться клиентом и сервером, будет определяться командой negprot (negotiate protocol). Этот SMB обязан посыпаться первым до установления соединения. Первым вариантом протокола был Core Protocol, известный как SMB-реализация PC NETWORK PROGRAM 1.0.

Команды SMB

Он должным образом поддерживает весь набор основных операций, который включает в себя:

- 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.

Аутентификация SMB

Модель механизма защиты, которая используется в Microsoft SMB Protocol, в основном идентична модели любого другого варианта SMB-протокола. Она состоит из двух уровней защиты: **user-level** (пользовательский уровень) и **share-level** (уровень совместно используемого ресурса). Под **share** (опубликованный ресурс) понимается файл, каталог, принтер, любая услуга, которая может быть доступна клиентам по сети.

[Аутентификация](#) на уровне **user-level** означает, что клиент, который пытается получить доступ к ресурсу на сервере, должен иметь **username** (имя пользователя) и **password** (пароль).

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 определённым данным.

Аутентификация на уровне **share-level** означает, что доступ к ресурсу контролируется паролем, установленным конкретно на этот ресурс. В отличие от user-level, этот уровень защиты не требует имя пользователя для аутентификации и не устанавливается никакая уникальность текущего пользователя. Этот уровень используется в Windows NT, Windows 2000 и Windows XP для обеспечения дополнительного уровня контроля защиты сверх user-level.

Оба метода шифрования используют аутентификацию типа отклик-ответ, в которой сервер посылает клиенту случайную сгенерированную строку, а клиент возвращает в качестве отзыва обработанную строку, которая доказывает, что клиент имеет достаточный мандат для доступа к данным.

Протокол NFS

Network File System (NFS) — [протокол сетевого доступа](#) к [файловым системам](#), первоначально разработан [Sun Microsystems](#) в [1984 году](#). Позволяет монтировать (подключать) удалённые файловые системы через сеть.

NFS абстрагирован от типов файловых систем как [сервера](#), так и клиента. Существует множество реализаций серверов и клиентов NFS для различных [операционных систем](#) и аппаратных архитектур. NFS

предоставляет клиентам прозрачный доступ к файлам и файловой системе сервера.

FTP vs NFS

В отличие от [FTP](#), протокол NFS осуществляет доступ только к тем частям файла, к которым обратился процесс, и основное достоинство его в том, что он делает этот доступ прозрачным. Это означает, что любое приложение клиента, которое может работать с локальным файлом, с таким же успехом может работать и с NFS-файлом, без каких-либо модификаций самой программы.

Цель разработки

Изначальными требованиями при разработке NFS были:

- потенциальная поддержка различных операционных систем (не только [UNIX](#)), чтобы серверы и клиенты NFS возможно было бы реализовать в разных операционных системах;
- протокол не должен зависеть от каких-либо определённых аппаратных средств;
- должны быть реализованы простые механизмы восстановления в случае отказов сервера или клиента;
- приложения должны иметь прозрачный доступ к удаленным файлам без использования специальных путевых имён или библиотек и без перекомпиляции;
- для UNIX-клиентов должна поддерживаться семантика UNIX;
- производительность NFS должна быть сравнима с производительностью локальных дисков;
- реализация не должна быть зависимой от транспортных средств

Как работает?

Взято [отсюда](#).

Что такое NFS

Система NFS (Network File System) служит для доступа к информации, содержащейся на дисках других компьютеров. По назначению она аналогична системам SMB (Windows) и NCP (Novell).

У NFS есть существенное отличие от этих систем: при монтировании не требуется указывать пароль, а авторизация осуществляется по IP-адресу и идентификаторам пользователя и группы (UID/GID). Достоинством такого подхода является то, что монтирование по NFS может быть

осуществлено без участия пользователя -- например, при загрузке системы. Недостатком является невысокий уровень security -- отсюда шутливая расшифровка аббревиатуры NFS как "No File Security".

В отличие от SMB и NCP, NFS использует протокол связи **без состояния**. Это дает очень высокую устойчивость к сбоям как сети, так и другого компьютера. В практике автора была ситуация, когда программа, запущенная на одном компьютере и использующая по NFS файлы с другого, при потере связи с последним просто приостановилась, а после восстановления связи через два дня спокойно возобновила работу как ни в чем не бывало.

Для правильной работы NFS с правами доступа к файлам требуется, чтобы UID и GID пользователей на обоих компьютерах совпадали.

Монтирование по NFS

При монтировании файловой системы по NFS в качестве типа системы надо указать "nfs", а вместо устройства -- имя компьютера и монтируемую директорию, разделенные двоеточием. Например, чтобы смонтировать директорию /dist с компьютера Rdist, можно воспользоваться командой

```
mount -t nfs rdist:/dist /mnt/rdist
```

Имя компьютера можно указывать любым образом -- короткое (rdist), полное (rdist.inp.nsk.su), или IP-адрес (193.124.167.12).

При монтировании по NFS есть большое количество опций, описанных в man-странице по nfs. Обычно же достаточно указать из соображений security "-o nosuid,nodev".

Экспорт систем по NFS

Для того, чтобы дать другим компьютерам доступ по NFS к дискам своей машины, надо отредактировать файл /etc/exports. Формат этого файла в разных клонах Unix разный (а в SystemV и сам файл называется по другому), поэтому рассмотрим только Linux.

Каждая строка содержит имя точки монтирования и список компьютеров, которым разрешен доступ к ней. После имени компьютера в скобках может указываться список опций. Пример:

```
/home Tom(ro) Jerry(rw)
```

Здесь компьютеру Том разрешается доступ к /home только на чтение, а компьютеру Jerry -- на чтение/запись.

Хотя есть несколько форматов указания компьютера, в большинстве случаев для того, чтобы дать доступ нескольким соседним машинам, достаточно просто указать имя.

Из опций также в большинстве случаев достаточно знать "ro" и "rw".

Из соображений security лучше всегда придерживаться очень простого правила: давать доступ к как можно меньшему количеству директорий, как можно меньшему количеству компьютеров, и с как можно меньшими правами.

Чтобы после внесения изменений в /etc/exports они сразу начали действовать, надо выполнить команду exportfs.

Автомонтиrovщик amd

При монтировании систем с нескольких компьютеров становится довольно неприятно постоянно отслеживать все изменения и вносить их в /etc/fstab.

Кроме того, если компьютеры А и В дают друг другу доступ, то при включении или А не сможет смонтировать системы с В, потому, что тот еще не загрузился, или наоборот (а при большем числе компьютеров ситуация еще хуже).

Для решения этих проблем были придуманы **автомонтировщики**. При использовании такой программы нет необходимости монтировать системы с другого компьютера при загрузке – достаточно обратиться к директории типа /net/компьютер/, и в ней магическим образом появятся все директории, которые доступны с указанного компьютера.

В Linux автомонтиrovщик называется amd, и содержится он в пакете am-utils (вплоть до RedHat 4.2 включительно этот пакет назывался amd). Хотя amd умеет также автоматически монтировать съемные носители (CD-ROM, дискеты, Zip), в основном он используется именно для NFS.

Для использования amd достаточно установить соответствующий пакет -- запускаться автомонтиrovщик будет автоматически при загрузке системы.

Одна из самых ценных возможностей, предоставляемых автомонтированием -- то, что при установке новых пакетов не требуется предварительно скачивать .rpm-файлы по FTP. Можно устанавливать прямо по NFS с сервера, беря файлы в /net/. Например, для RedHat 5.2 в ИЯФ можно использовать путь

/net/rdist/dist/redhat-5.2/i386/RedHat/RPMS/

это та же самая директория, которая видна как

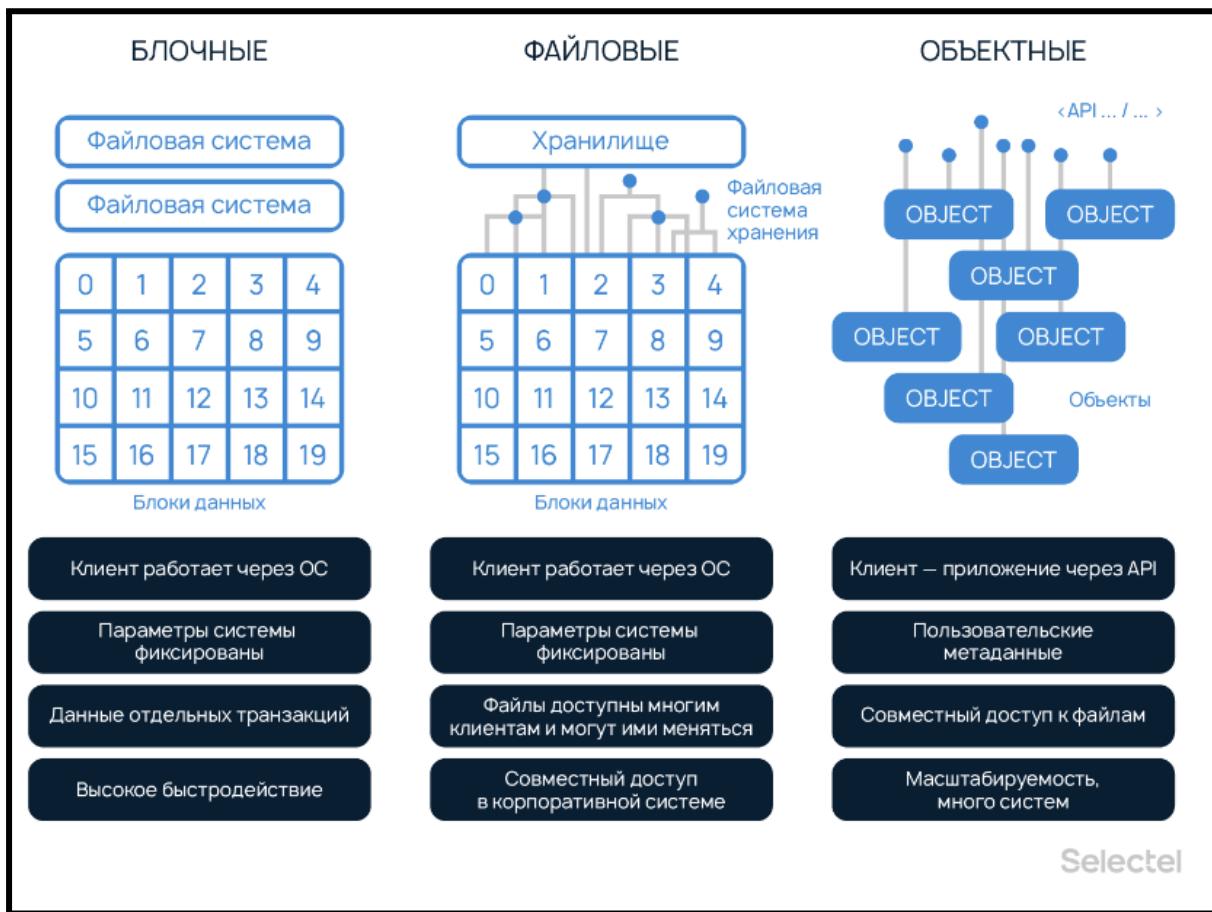
<ftp://rdist.inp.nsk.su/pub/Linux/redhat-5.2/i386/RedHat/RPMS/>

(более конкретно -- /net/rdist/dist/ -- это то же самое, что

<ftp://rdist.inp.nsk.su/pub/Linux/>).

"Облачное" хранение файлов

Виды хранилищ



Блочное

Весь объем информации делится на равные части — блоки с идентификаторами. Основное преимущество таких облачных хранилищ — разделение клиентских сред. Благодаря этому к каждой из них открывается быстрый отдельный доступ. Но платить нужно за весь выделенный объем памяти, даже если она ничем не занята.

Примеры хранилищ: Amazon Elastic Block Storage (EBS).

Файловое

Данные хранятся в иерархической системе. Это значит, что информация представляет собой файлы, объединяющиеся в папки, подкаталоги и каталоги. Основное преимущество — интуитивный интерфейс и легкость использования. Главный недостаток — плохая масштабируемость: с увеличением объема данных иерархия очень сильно усложняется и замедляет работу системы.

Примеры хранилищ: Яндекс.Диск, Dropbox, OneDrive, Google Диск.

Объектное

Это универсальный и современный способ хранения в облаке больших информационных массивов. Объектное хранилище используется для

данных любого вида: медиаконтента, программ, бухгалтерской/статистической отчетности и др. Главный недостаток — пользователь не может просто взять и переместить файл в нужную папку. Для загрузки информации нужно использовать специальный программный интерфейс — API (он позволяет двум независимым компонентам ПО обмениваться информацией).

Примеры хранилищ: Amazon Simple Storage Service (S3).

Плюсы облачного хранилища

- Доступ к данным с любого устройства, имеющего выход в интернет.
- Сохранение данных даже в случае сбоев.
- Организация совместной работы с информацией.
- Отсутствие необходимости покупать, поддерживать и обслуживать инфраструктуру по хранению данных (сервера).

Минусы облачного хранилища

- Необходимость качественного интернета.
- Замедление работы в облаке, если файлы весят много.
- Могут быть проблемы с безопасностью сохранности данных (например, однажды хакеры [взломали](#) 68 млн учетных записей Dropbox).

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

Общие слова

При разработке [требований к ПО](#) или в рамках проектов [интеграции информационных систем](#) у аналитика возникают вопросы про схему данных и формат файлов. Это связанные понятия, однако не одно и тоже.

Формат файла, отображаемый в его расширении, говорит о том, каким образом сообщение сериализуется, т.е. переводится в двоичный код, последовательность битов из 1 и 0, для передачи по сети или сохранения в постоянной памяти (на диске). Обратный процесс восстановления сообщения, имеющего практический смысл, из битовой последовательности, называется десериализацией.

За сериализацию и десериализацию отвечают приложения-источники и приемники сообщений, реализуя программный код работы с объектами и структурами данных. Этот код выполняет нужное преобразование данных каждого типа (числовые, символьные, дата и время, логические и пр.), которые встречаются в сообщении. На практике разработчик не часто пишет собственные сериализаторы и десериализаторы, а, в основном, пользуется готовыми библиотеками или встроенными во фреймворк классами.

Таким образом, **формат файла описывает правила сериализации данных, которые есть в сообщении**. Однако, этого недостаточно для верификации сообщения, пришедшего в систему-приемник из системы-источника.

Например, необходимо проверить, что пришли все нужные данные, и их тип соответствует ожидаемому.

За это отвечает схема данных, которая задает структуру сообщения: набор полей и их типы данных, а также обязательность каждого поля. В зависимости от используемого формата сериализации, схема данных может быть встроена в само сообщения или представлена отдельным файлом.

Например, бинарный формат Protobuf, который используется в технологии gRPC, включает описание схемы в само сообщение, а JSON, XML и YAML предполагают описание схемы в отдельном файле аналогичных форматов.

XML

XML – это открытый стандарт для хранения и обмена данными. Это язык разметки для описания структуры и содержания любого XML файла, такого как документы, веб-страницы или базы данных. Можно считать, что XML похож на HTML, но лучше: он позволяет прикреплять дополнительную информацию к узлам документа без изменения основного формата.

Существует множество преимуществ использования кода XML. Одно из главных - это то, что он более гибкий, чем HTML, а значит, вы можете легко создать собственный язык разметки для своего сайта или приложения. Кроме того, это облегчает форматирование данных и их корректное отображение в любом браузере или устройстве.

XML также позволяет создавать пользовательские теги, которые можно использовать в любых данных XML. Эти теги часто используются для определения конкретных XML данных. Это облегчает совместную работу разработчиков и дизайнеров при создании новых проектов!

Документ XML или данные XML - это набор элементов и атрибутов, которые могут быть вложены друг в друга. Элементы окружены открывающими и закрывающими тегами, а атрибуты - нет. Элемент может содержать вложенные элементы, символьные данные и текст. Обратите внимание, что в данных XML нет пробелов между тегами и между элементами; все должно быть заключено в пары скобок.

JSON

JSON является форматом обмена данными. Он не зависит от языка, то есть может использоваться с любым языком программирования, а лежащая в его основе структура данных не зависит от платформы. Независимая от языка природа JSON делает его идеальным для использования в веб-разработке, где может потребоваться обмен данными с другими языками программирования, такими как Ruby или [JavaScript](#).

JSON использует теги для разметки данных: `{"key": value, "otherKey": anotherValue}.` Ключи и значения всегда должны быть окружены фигурными скобками `{}` и квадратными скобками `[]` соответственно. Кроме того, каждая пара ключ-значение должна иметь равное количество кавычек вокруг нее - например: `{"name": "John"}` не будет корректным, потому что после тега `name` слишком мало кавычек.

JSON является легковесным благодаря использованию экономичного двоичного кодирования (эту технику мы рассмотрим более подробно позже). Это делает его идеальным для передачи небольших объемов информации по сети, например, при отправке запросов на оплату между интернет-магазинами или [веб-интерфейсами API](#), возвращающими результаты из баз данных.

JSON Библиотеки парсеров позволяют читать и записывать этот формат, не требуя знаний о том, что входит в каждое поле - все, что вам нужно, это несколько основных правил:

- 1) Поля должны иметь согласованные имена.
- 2) Все значения должны быть строками.
- 3) Знаки запятой должны разделять значения.

JSON также является человекочитаемым, то есть вы можете открыть файл и посмотреть, что в нем находится, не прогоняя его через синтаксический анализатор. Это делает отладку проблем в вашем коде более доступной и помогает документировать данные, полученные вами от других приложений (что особенно полезно, если они написаны на другом языке).

JSON Формат обмена данными используется во многих различных контекстах, от веб-разработки до хранения данных. Это также идеальный формат для обмена информацией между веб-интерфейсами и приложениями, поскольку это простой способ сериализации сложных данных в одну строку.

JSON vs XML: различия

1) JSON vs XML: первое отличие

JSON расшифровывается как JavaScript Object Notation. Это текстовый открытый стандартный формат обмена данными. JSON легкий и легко читаемый, но не предоставляет схем или информации о типах. Он отлично подходит для обмена данными между несколькими приложениями. XML означает расширяемый язык разметки. Это язык разметки, который определяет структуру любого файла XML в виде древовидной структуры. XML читается людьми, но не обязательно машинами. Его можно использовать для обмена структурированной информацией между программами и документами.

2) JSON vs XML: второе отличие

JSON и XML - популярные способы хранения структурированных данных в файле или базе данных. Как уже упоминалось, JSON - это легкий,

читаемый человеком способ представления структур данных, в то время как код XML - более объемный способ представления структурированных данных.

3) JSON vs XML: третье отличие

Одно из ключевых различий между этими двумя форматами данных заключается в том, что JSON может использоваться с JavaScript или обычными текстовыми файлами, в то время как XML может храниться только в виде текстового файла. Кроме того, при обработке информации JSON использует меньше памяти, чем программа XML. Эти ключевые различия в использовании памяти делают JSON идеальным форматом для быстрой обработки больших объемов данных.

4) JSON vs XML: четвертое отличие

Формат JSON - это способ компактного хранения данных, чтобы программы могли их прочитать. Как правило, его легче писать и читать, чем XML, поскольку в нем используется меньше символов. В то же время формат данных XML - это особая форма языка разметки для хранения данных в организованном виде. У него больше возможностей, чем у JSON, но он также сложнее, поскольку требует больше информации о структуре документа, прежде чем его можно будет прочитать.

5) JSON vs XML: пятая разница

Формат JSON используется для хранения и передачи данных, а XML - для представления данных в машиночитаемом виде. JSON набирает популярность как средство хранения данных для веб-приложений благодаря своей простоте. В отличие от него, XML все еще используется для передачи структурированных данных через Интернет.

6) JSON vs XML: шестая разница

Одно из ключевых различий между этими двумя форматами данных заключается в том, что JSON, как правило, более компактен, чем XML, а значит, его можно быстрее передавать по сетям. JSON также имеет меньше ограничений на структуру, что помогает программистам при попытке разобрать большой объем данных. Кроме того, многие языки программирования поддерживают оба формата, поэтому нет необходимости переключаться между ними при работе с разными платформами или языками программирования.

7) JSON vs XML: седьмая разница

Вы можете использовать JSON в своем веб-приложении или мобильном приложении, не беспокоясь о проблемах совместимости, поскольку он широко распространен в веб- и мобильных приложениях. С другой стороны, XML имеет некоторые проблемы, когда речь идет о кросс-платформенной совместимости. Он не поддерживается многими языками программирования (кроме Actionscript), поэтому разработчикам приходится выбирать между использованием таких инструментов, как Apache HttpComponents или Apache axis2, если они хотят, чтобы их приложения работали на нескольких платформах одновременно.

8) JSON vs XML: восьмая разница

XML файлы требуют больше места для хранения, чем файлы JSON (по крайней мере, если вы используете их с Node). В целом, это не является проблемой, если на вашем веб-сервере достаточно оперативной памяти для хранения этих файлов; в противном случае вам следует подумать об изменении архитектуры вашего приложения, чтобы вся обработка происходила на стороне клиента, а не за кулисами, где она будет занимать слишком много места в памяти.

9) JSON против XML: девятое отличие

Одно из ключевых различий между JSON и XML заключается в том, что XML имеет более жесткую структуру, чем JSON, что делает его более сложным для манипуляций без разрушения документа. Кроме того, большинство файлов XML нельзя редактировать в месте, как это делают документы JSON; поэтому, если вы хотите изменить значение элемента в документе JSON, вы можете отредактировать значение непосредственно в текстовом редакторе, и дело сделано. Это означает, что злоумышленник может изменить значение элемента, просто отредактировав сам документ, и это отразится в выводе вашей программы.

10) JSON vs XML: 10 разница

Синтаксис JSON и XML:

JSON синтаксис более компактный, чем у XML.

JSON синтаксис легче читать и писать.

Синтаксис JSON позволяет легко определять объекты, в отличие от более многословного способа работы с массивами или коллекциями в синтаксисе XML. Например:

```
``javascript function myFunction(date) { return { "date": date }; } var obj =  
Object.create(null); obj["date"] = new Date(); ````
```

Также важно помнить, что синтаксис для XML сложнее, чем для JSON из-за необходимости ссылок на сущности, которые могут не понадобиться в некоторых случаях (например, если вы создаете API-сервис). XML не является человекочитаемым. Читать JSON гораздо легче, чем XML, потому что в JSON используется меньше символов, что облегчает понимание смысла данных. JSON более лаконичен. В нем используется меньше символов для представления той же информации, что и в XML.

11)JSON против XML:11 разница

В JSON и XML тип данных значения кодируется как объект или элемент. В JSON в качестве типов данных поддерживаются только строки, числа, булевы и null. С другой стороны, в данных XML для описания данных XML могут использоваться многие другие типы, например, дата и время.

В объектной нотации JavaScript типы данных никак не кодируются. Разработчик сам решает, как он хочет представить свои данные в виде объектов и массивов, используя JSON. В результате в JSON не существует правил относительно того, что может быть использовано в качестве значения или имени атрибута.

Пример JSON vs XML

XML vs. JSON

| | |
|--|--|
| <pre><?xml version="1.0" encoding="UTF-8" ?> <dataset> <record> <id>1</id> <first_name>Kyle</first_name> <last_name>Danzey</last_name> <email>kdanze0@dedecms.com</email> </record> <record> <id>2</id> <first_name>Stanly</first_name> <last_name>Chaise</last_name> <email>schaise1@php.net</email> </record> <record> <id>3</id> <first_name>Valentine</first_name> <last_name>Vasler</last_name> <email>vvasler2@ifeng.com</email> </record> <record> <id>4</id> <first_name>Herve</first_name> <last_name>Tollet</last_name> </record></pre> | <pre>{ "dataset": { "record": [{ "id": "1", "first_name": "Kyle", "last_name": "Danzey", "email": "kdanze0@dedecms.com" }, { "id": "2", "first_name": "Stanly", "last_name": "Chaise", "email": "schaise1@php.net" }, { "id": "3", "first_name": "Valentine", "last_name": "Vasler", "email": "vvasler2@ifeng.com" }, { "id": "4", "first_name": "Herve", "last_name": "Tollet", "email": "htollet3@chronoengine.com" }] } }</pre> |
|--|--|

YAML

Взято [отсюда](#).

YAML — это специальный язык для структурированной записи информации, обладающий простым синтаксисом. Этот инструмент позволяет сохранять сложно организованные данные в формате (файл с расширением .yml), который компактен и легко читаем. Эта возможность может оказаться чрезвычайно полезной в контексте DevOps и виртуализации.

Сравнение YAML, JSON и XML

YAML (.yml)

Особенности:

- Обладает человекопонимаемым кодом;
- Имеет минималистичный синтаксис;
- Ориентирован на работу с данными;
- Включает в себя структуру, напоминающую JSON (YAML находится в расширенной версии JSON);
- Позволяет добавлять комментарии;

- Поддерживает использование строк без кавычек;
- Считается более «чистым» в сравнении с JSON;
- Предоставляет дополнительные возможности, такие как расширяемые типы данных, относительные якоря и сохранение порядка ключей.

Применение: YAML оптимально подходит для приложений с обширным объемом данных, которые основаны на DevOps-процессах или используют виртуальные машины. Повышенная читаемость данных оказывается особенно полезной в командах, где разработчики регулярно взаимодействуют с этими данными.

JSON

Особенности:

- Требует больше усилий для чтения;
- Синтаксис имеет жесткие и четкие требования;
- Подобен встроенному стилю YAML (некоторые парсеры YAML могут интерпретировать JSON-файлы);
- Отсутствует возможность добавления комментариев;
- Строки должны быть обрамлены двойными кавычками.

Применение: JSON применяется в веб-разработке, представляя наилучший формат для сериализации и передачи данных через HTTP-соединение.

XML

Особенности:

- Требует больше усилий для чтения;
- Обладает более сложной структурой;
- Выполняет функцию языка разметки, в то время как YAML используется для форматирования данных;
- Предоставляет больший спектр возможностей, например, использование атрибутов тегов;
- Обладает более жесткой схемой документа.

Применение: XML идеально подходит для сложных проектов, требующих тщательного контроля над валидацией, схемой и пространством имен. XML имеет низкую читаемость, потребляет больше пропускной способности и требует больше места для хранения данных, но при этом предоставляет непревзойдённый уровень контроля.

PROTOBUF

Отсюда

Protocol Buffers (Protobuf) - это формат сериализации данных, разработанный компанией Google. Он эффективно и компактно хранит

структурированные данные в двоичной форме, что позволяет быстрее передавать их по сети. Protobuf поддерживает широкий спектр выбранных языков программирования и является платформонезависимым, что означает, что программы, написанные с его использованием, могут быть легко перенесены на другие платформы.

В целом, Protocol Buffers - это мощный и эффективный способ обмена информацией между системами.

- Он позволяет разработчикам создавать эффективные API для передачи структурированных данных
- хранить данные в упорядоченном виде
- и создавать клиент-серверные приложения, которые могут взаимодействовать друг с другом.

Protocol Buffers является проектом с открытым исходным кодом и обеспечивает надежный и эффективный способ обмена данными между системами. С помощью Protobuf разработчики могут значительно уменьшить сложность, связанную с обменом данными между различными платформами, и повысить производительность своих приложений за счет снижения сетевых задержек. Это незаменимый инструмент для всех, кому необходимо быстро разрабатывать надежные и эффективные приложения, способные взаимодействовать друг с другом.

Оч кайфово написано про protobuf

Преимущества Protobuf

- 1 Fully type safe.
- 2 Data is auto compressed which reduces cpu and network bandwidth usage.
- 3 Provides backward and forward compatibility.
- 4 3-10x smaller, 20-100x faster than xml.
- 5 Provides ability to auto-generate client code in multiple languages, schema(.proto file) is used to generate code and read the data.
- 6 RPC frameworks like gRPC uses [Protocol Buffers](#) by default, which provides much better performance than JSON.
- 7 Easy to learn.

Недостатки Protobuf

- 1 Support for some languages might be lacking(most mainstream languages are fine).
- 2 Cant open the serialised data with a text editor.
- 3 Every field in a Protobuf message is optional and has a default value, it is impossible to differentiate a field that is missing in a protocol buffer from one that was assigned with the default value. ex; Int is defaulted to Zero, strings and arrays are defaulted to empty!

Синтаксис Protobuf-a

```
1 syntax = "proto3";
2
3 message Person{
4     string first_name = 1;
5     string last_name = 2;
6     int32 age = 3;
7     float weight = 4;
8     repeated string addresses = 5;
9     enum Gender{
10         Unknown = 0;
11         FeMale = 1;
12         Male =2;
13     }
14
15     Gender gender = 6;
16 }
```

```
1  syntax = "proto3"; → Protobuf Version
2
3  message Person{ → Protobuf message
Field type → string first_name = 1; → Field tag
5      string last_name = 2; → FieldName
6      int32 age = 3;
7      float weight = 4;
8      repeated string addresses = 5;
9      enum Gender{
10          Unknown = 0;
11          FeMale = 1;
12          Male =2;
13      }
14
15      Gender gender = 6;
16  }
17
```

Типы в Protobuf

Default values

If the value is not specified for any field in protobuf it always gets a default value

| Data Type | Default value |
|-----------|---------------|
| bool | false |
| number | 0 |
| string | empty string |
| bytes | empty bytes |
| enum | first value |
| repeated | empty list |

Сравнение Protobuf & JSON

Таким образом, различия между Protobuf и JSON можно разделить на четыре основные области: скорость, размер, типы данных и совместимость с платформами.

Скорость: Protobuf намного быстрее, чем JSON в отношении сериализации и десериализации данных. Поскольку формат бинарный, чтение и запись структурированных данных в Protobuf занимает меньше времени, чем в JSON.

Размер: Protobuf намного меньше, чем JSON, что может быть невероятно полезно при ограниченной пропускной способности сети. Благодаря компактности двоичных потоков данных, для хранения и передачи сообщения Protobuf требуется меньше места, чем для сообщения JSON.

Типы данных: Хотя оба формата поддерживают основные типы данных, такие как строки, числа и булевы, Protobuf поддерживает более сложные типы данных, такие как перечисления и карты, недоступные в JSON. Это позволяет разработчикам создавать более сложные приложения, требующие более богатых структур данных.

Совместимость с платформами: Поскольку Protobuf является форматом с открытым исходным кодом, он обладает лучшей совместимостью с платформами, чем JSON. Он может использоваться на различных платформах без трудностей и проблем совместимости, поскольку является независимым от языка и платформы.

Protobuf является предпочтительным форматом данных для приложений, требующих скорости, эффективности размера и сложных типов данных.

Protobuf vs XML vs JSON

Взято [отсюда](#).

Json

- human readable/editable
- can be parsed without knowing schema in advance
- excellent browser support
- less verbose than XML

XML

- human readable/editable
- can be parsed without knowing schema in advance
- standard for SOAP etc
- good tooling support (xsd, xslt, sax, dom, etc)
- pretty verbose

Protobuf

- very dense data (small output)
- hard to robustly decode without knowing the schema (data format is internally ambiguous, and needs schema to clarify)
- very fast processing
- not intended for human eyes (dense binary)

Сравнение HTTP 1.x, HTTP 2 и HTTP 3. Преимущества и недостатки.

HTTP (Hypertext Transfer Protocol) - это протокол передачи данных, который используется для обмена информацией между клиентом и сервером в Интернете. Существует несколько версий протокола HTTP, каждая из которых имеет свои преимущества и недостатки.

TL;DR

HTTP 1.x:

- Преимущества: универсальность, широкое распространение, простота реализации;
- Недостатки: медленная скорость передачи данных из-за ограниченного числа одновременных запросов и ответов, невозможность сжатия заголовков, недостаточная безопасность.

HTTP 2:

- Преимущества: увеличение скорости передачи данных благодаря множественным потокам, возможность сжатия заголовков, улучшенная безопасность;
- Недостатки: сложность реализации, несовместимость с некоторыми старыми браузерами и серверами.

HTTP 3:

- Преимущества: еще большее увеличение скорости передачи данных благодаря использованию протокола QUIC, улучшенная безопасность;
- Недостатки: пока еще находится в разработке и не поддерживается всеми браузерами и серверами.

В целом, каждая версия HTTP имеет свои преимущества и недостатки, и выбор версии зависит от конкретных потребностей и задач. Однако, в целом, HTTP 2 и HTTP 3 предпочтительнее использовать из-за улучшенной скорости передачи данных и безопасности.

Команды HTTP

Для разграничения действий с ресурсами на уровне HTTP-методов и были придуманы следующие варианты:

- [GET](#)

Метод GET запрашивает представление ресурса. Запросы с использованием этого метода могут только извлекать данные.

- [HEAD](#)

HEAD запрашивает ресурс так же, как и метод GET, но без тела ответа.

- [POST](#)

POST используется для отправки сущностей к определённому ресурсу. Часто вызывает изменение состояния или какие-то побочные эффекты на сервере.

- [PUT](#)

PUT заменяет все текущие представления ресурса данными запроса.

- [DELETE](#)

DELETE удаляет указанный ресурс.

- [CONNECT](#)

CONNECT устанавливает "туннель" к серверу, определённому по ресурсу.

- [OPTIONS](#)

OPTIONS используется для описания параметров соединения с ресурсом.

- [TRACE](#)

TRACE выполняет вызов возвращаемого тестового сообщения с ресурса.

- [PATCH](#)

PATCH используется для частичного изменения ресурса.

Пример работы HTTP

Когда мы набрали запрос и нажали ввод, браузер отправил этот запрос на веб-сервер.

На веб-сервере хранятся статьи в виде картинок, HTML-документов, файлов с CSS-стилями и JavaScript-файлами. Также на веб-сервере установлено ПО, которое понимает HTTP-протокол.

Веб-сервер принимает запрос, обрабатывает, определяет, какие файлы отправить, и отдает их в ответ. Браузер принимает эти данные, интерпретирует и показывает нам в «человеческом» виде.

Каждое действие в блоге, например, переход по ссылке на статью, — это новый запрос серверу и новый ответ. Запрос выглядит примерно так:

```
GET / HTTP/1.1
Host: selectel.ru
User-Agent: Mozilla/5.0
Accept: text/html
Connection: close
```

В запросе есть данные о браузере, версии HTTP-протокола, адресе, к которому обращается браузер, и о том, что именно нужно получить от сервера.

За последнее отвечает метод. Приведенном примере метод GET, который указывает, что браузер хочет прочитать страницу с сервера. Есть еще методы HEAD, PUT, PATCH, POST и другие. С их помощью можно отправить серверу команды удалить страницы, добавить на них новые данные или что-то скачать. Но эти методы встречаются гораздо реже.

HTTP-ответ выглядит примерно так:

```
HTTP/1.1 200 OK
Date: Tue, 05 Aug 2021 09:50:20 GMT
Server: Apache/1.3.26
X-Powered-By: PHP/4.1.2.
Content-Language: ru
Content-Type: text/html; charset=utf-8
Content-Length: 18
Connection: close
```

В ответе содержится:

версия протокола — HTTP/1.0 — и ответ: в примере это «200» — страница доступна;
время и дата ответа;
информация о сервере — в примере Apache-сервер;
инструкция, как браузеру отобразить страницу (content-type) — в примере это необходимо сделать в кодировке UTF-8.

В ответе мы также получаем HTML с данными страницы — ту самую гипертекстовую разметку из определения HTTP. Гипертекстовая разметка, например, для статьи про управление доменами в блоге Selectel будет выглядеть примерно так:

```
<!DOCTYPE html>
<html lang="ru-RU">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no, user-scalable=no">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <meta name="SKYPE_TOOLBAR" content="SKYPE_TOOLBAR_PARSER_COMPATIBLE">
    <meta name="yandex-verification" content="3c6eld4984cbafae">
    <meta name="google-site-verification" content="F89rXTMXRgew2SWiZ7CaN9ruT76FMNnaypsfFoU1fGA">
    <meta name="google-site-verification" content="muTMJ-pbpCyrKdDnWIfyxmHg1-yf_3KstSCV6W1E2VU">
    <link rel="icon" href="https://selectel.ru/blog/wp-content/themes/selectel/img/favicon.png">
    <script src="https://www.googleoptimize.com/optimize.js?id=GTM-NTN8ZRX"></script>
    <meta name='robots' content='index, follow, max-image-preview:large, max-snippet:-1, max-video-preview:-1' />

    <title>Управление доменами с Selectel DNS API - Блог компании Селектел</title>
    <meta name="description" content="Управление доменами с Selectel DNS API - Блог компании Селектел" />
    <meta property="og:locale" content="ru_RU" />
    <meta property="og:type" content="article" />
    <meta property="og:title" content="Управление доменами с Selectel DNS API - Блог компании Селектел" />
    <meta property="og:description" content="Управление доменами с Selectel DNS API - Блог компании Селектел" />
    <meta property="og:url" content="https://selectel.ru/blog/upravlenie-domenami-s-selectel-dns-api/" />
    <meta property="og:site_name" content="Блог компании Селектел" />
    <meta property="article:published_time" content="2015-06-04T15:30:53+00:00" />
    <meta property="article:modified_time" content="2022-03-31T16:09:48+00:00" />
    <meta property="og:image" content="https://selectel.ru/blog/wp-content/uploads/2015/06/PR-1056_preview.png" />
    <meta property="og:image:width" content="1600" />
    <meta property="og:image:height" content="800" />
    <meta name="twitter:card" content="summary_large_image" />
    <meta name="twitter:label1" content="Написано автором">
    <meta name="twitter:data1" content="T-Rex">
    <meta name="twitter:label2" content="Примерное время для чтения">
    <meta name="twitter:data2" content="5 минут">
    <script type="application/ld+json" class="yoast-schema-graph">
        {
            "@context": "https://schema.org",
            "@graph": [
                {
                    "id": "https://selectel.ru/blog/upravlenie-domenami-s-selectel-dns-api/",
                    "url": "https://selectel.ru/blog/upravlenie-domenami-s-selectel-dns-api/",
                    "name": "Управление доменами с Selectel DNS API - Блог компании Селектел",
                    "isPartOf": "https://selectel.ru/",
                    "image": "https://selectel.ru/blog/wp-content/uploads/2015/06/PR-1056_preview.png"
                }
            ]
        }
    </script>

```

Про HTTPS

HTTPS — это не совсем протокол. Это расширение HTTP-протокола — объединение двух протоколов: HTTP и SSL или HTTP и TLS.

Протоколы TLS (Transport Layer Security) и SSL (Secure Socket Layer) — криптографические. Это значит, что они позволяют шифровать данные, в нашем случае те, что передаются между браузером и сервером. Расшифровать эти данные могут только сервер и браузер, для всех остальных это будет набор нечитаемых символов.

Примечание: TLS основан на SSL, но второй уже устарел, и вместо него используют TLS.

Как работает шифрование

У ресурса/сайта, поддерживающего HTTPS, есть SSL/TLS-сертификат, который выдается центром сертификации.

Как правило, SSL/TLS-сертификат — это подтверждение, что ресурс настоящий. Но могут быть исключения: сертификат может быть выдан

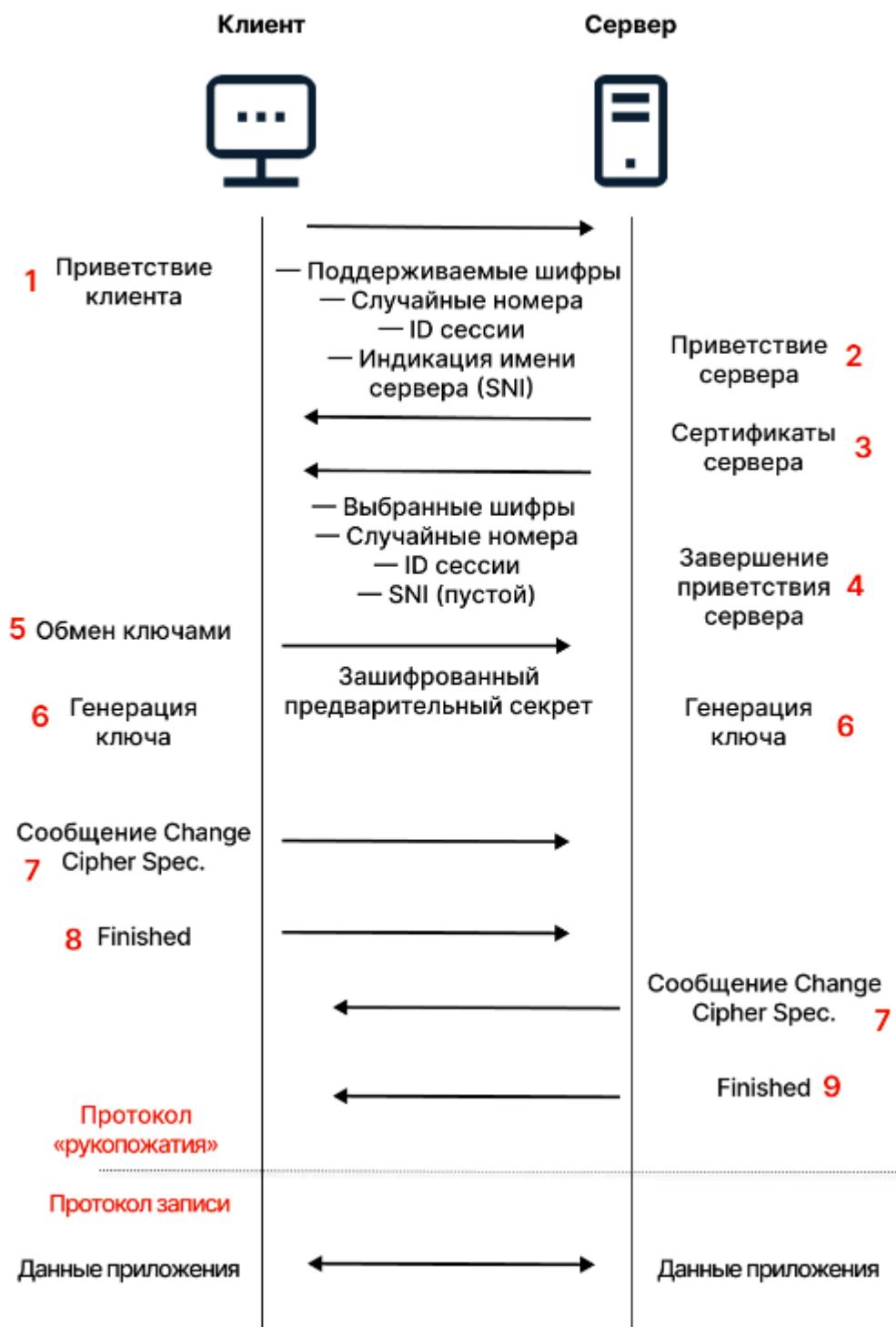
легитимным центром на фишинговый сайт. В таком случае важно совпадение CN в сертификате с доменным именем сайта и уверенность пользователя в этом имени.

Перед тем как запустить HTTP-соединение, браузер обращается к серверу, чтобы наладить защищенное соединение. Сервер отправляет копию сертификата безопасности в ответ.

Браузер проверяет данные по своим спискам доверенных центров (список есть в каждом браузере), проверяет совпадение CN с доменным именем, даты выпуска и срока окончания сертификата, отсутствие в CRL

(Certificate Revocation List - Списки отзываемых сертификатов — это список сертификатов, которые удостоверяющий центр пометил как отзываемые. Списки отзываемых сертификатов применяются для того, чтобы установить, был ли сертификат пользователя или удостоверяющего центра отзван в связи с компрометацией ключей),

поддерживаемые алгоритмы, наличие издателя в списке доверенных корневых сертификатов и в списке доверенных издателей. В случае проблем на любой из этих проверок сертификат считается невалидным. Если все хорошо, то браузер считает ресурс безопасным: они выбирают алгоритм шифрования, обмениваются ключом шифрования и потом данными по протоколу HTTP. Схематически это выглядит так:



Server Name Indication (SNI) — расширение компьютерного протокола TLS[1], которое позволяет клиенту сообщать имя хоста, с которым он желает соединиться во время процесса «рукопожатия». Это позволяет серверу предоставлять несколько сертификатов на одном IP-адресе и TCP-порту, и, следовательно, позволяет работать нескольким безопасным (HTTPS) сайтам (или другим сервисам поверх TLS) на одном IP-адресе без использования одного и того же сертификата на всех сайтах.

Сравнение HTTP1 & HTTP2

У HTTP есть следующие достоинства:

- использовать протокол можно в локальных сетях;
- страницы HTTP хранятся в кэше компьютера и они быстрее открываются;
- страницы HTTP открываются во всех браузерах;
- страницы HTTP мало весят

Создание HTTP2 решает проблему того, что в HTTP1 на каждый запрос, нужно открывать новое соединение. В http2 такого нет, тк в нём поддерживается мультиплексирование. [Смотреть здесь.](#)

Преимущества HTTP2 (перед http1)

- Мультиплексированная асинхронная передача данных: на одном соединении запросы разделяются на чередующиеся пакеты, сгруппированные в отдельные потоки.
- Запросы приоритизируются, благодаря чему снимается проблема с одновременной отправкой всех запросов.
- Реализовано сжатие HTTP-заголовков. Каждый отправленный заголовок содержит информацию об отправителе и получателе, а это – избыточные объёмы. Благодаря сжатию полная информация отправляется только в первом заголовке, в последующих отправленных заголовках такой информации уже нет.
- В отличие от текстового протокола HTTP, HTTP/2 - бинарный. Благодаря этому можно обрабатывать небольшие сообщения, из которых формируются более крупные.
- Server Push. Если в версии HTTP/1 браузер должен был сначала получить домашнюю страницу, и лишь из неё понять, какие ресурсы ему необходимы для рендеринга, то HTTP/2 позволяет отправить все необходимые ресурсы сразу, при первичном обращении к серверу.

Про HTTP3

Читать [отсюда](#).

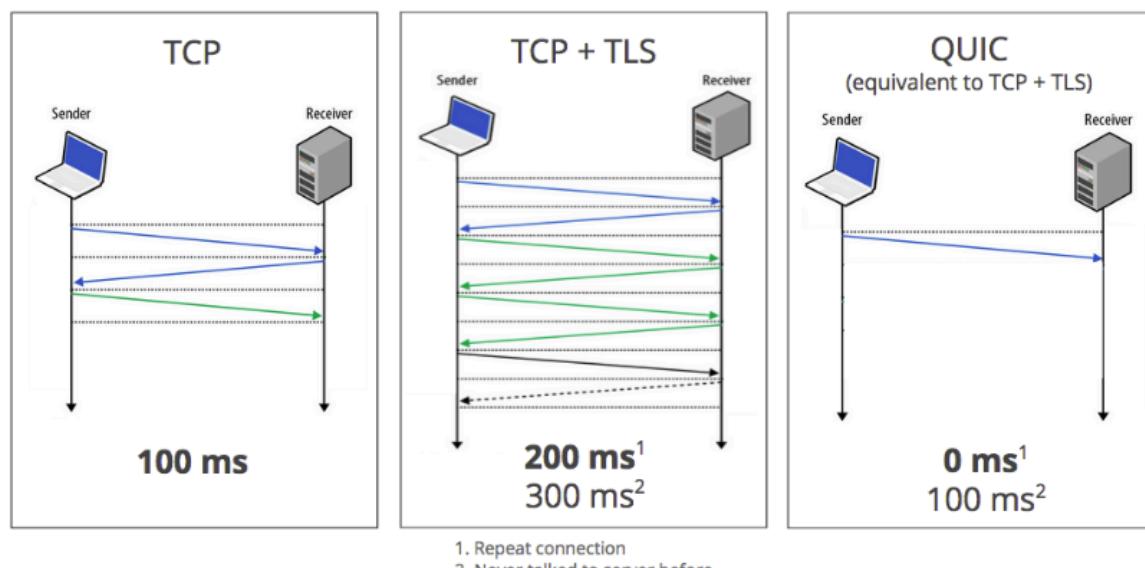
1. HTTP/3 — готовящаяся к стандартизации версия протокола HTTP. Её главное отличие от предыдущих версий в том, что она использует новый транспортный протокол QUIC и передаёт данные быстрее.

2. QUIC — транспортный протокол, работающий поверх UDP. Передаёт данные быстрее TCP, тратит меньше времени на установку соединения, но при этом такой же надёжный.
3. HTTP/3 использует QUIC на транспортном уровне и уровне безопасности. QUIC заменяет TCP и TLS.
4. Помимо QUIC, HTTP/3 отличается от HTTP/2 тем, что имеет другой алгоритм сжатия заголовков и исключает некоторые функции (например, мультиплексирование), так как их уже содержит QUIC.
5. HTTP/3 лучше реализовывает мультиплексирование. Если в HTTP/2 при потере TCP-пакета вся передача данных останавливалась до восстановления пропажи, то в HTTP/3 информация продолжает передаваться.
6. В использовании HTTP/3 пока есть некоторые сложности, связанные с определением типа протокола клиентом и сервером и блокировкой UDP-пакетов некоторыми брандмауэрами. Но эти проблемы решаются, когда веб перейдёт на новый протокол.
7. Пока HTTP/3 не стандартизован, но его уже поддерживают многие сервисы и браузеры, в том числе EdgeCDN.

Про QUIC

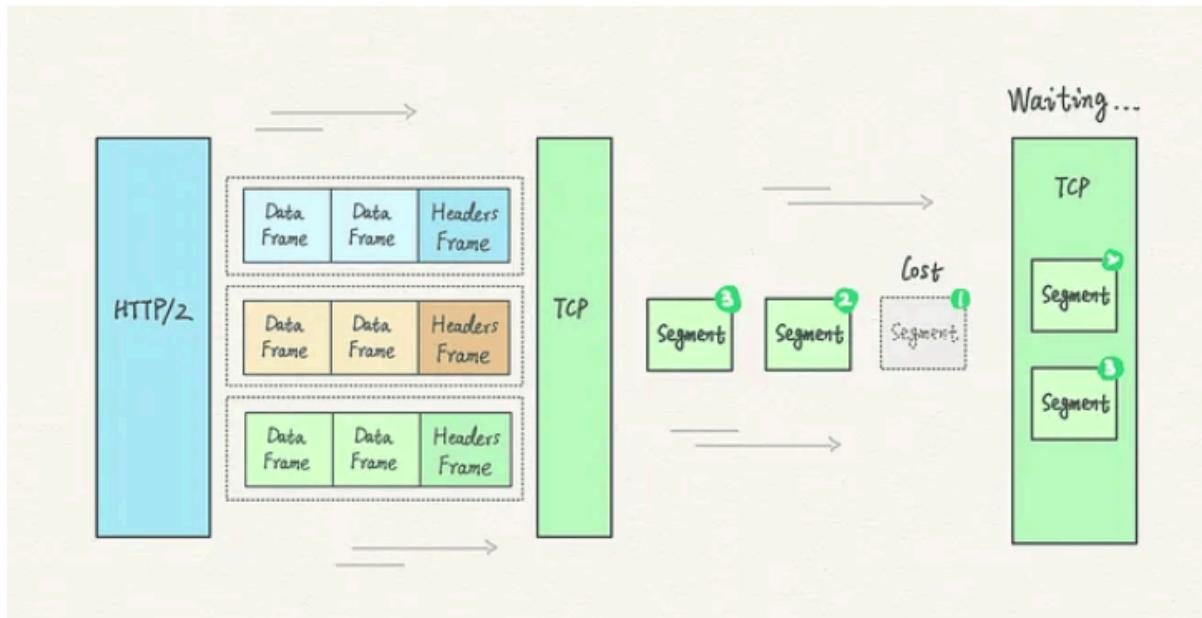
QUIC — экспериментальный интернет-протокол, разработанный Google в конце 2012 года. QUIC позволяет мультиплексировать несколько потоков данных между двумя компьютерами, работая поверх протокола UDP, и содержит возможности шифрования, эквивалентные TLS и SSL. Имеет более низкую задержку соединения и передачи, чем TCP.

Zero RTT Connection Establishment



Смотреть сюда: <https://habr.com/ru/companies/otus/articles/725902/>

Читать статью с [Хабра](#).



HTTP/2 решает проблему блокировки начала очереди на уровне HTTP с помощью фреймов (frames) и потоков (streams). Однако проблема остается на уровне TCP.

Получив фреймы со своего верхнего уровня, TCP разбивает их на сегменты.

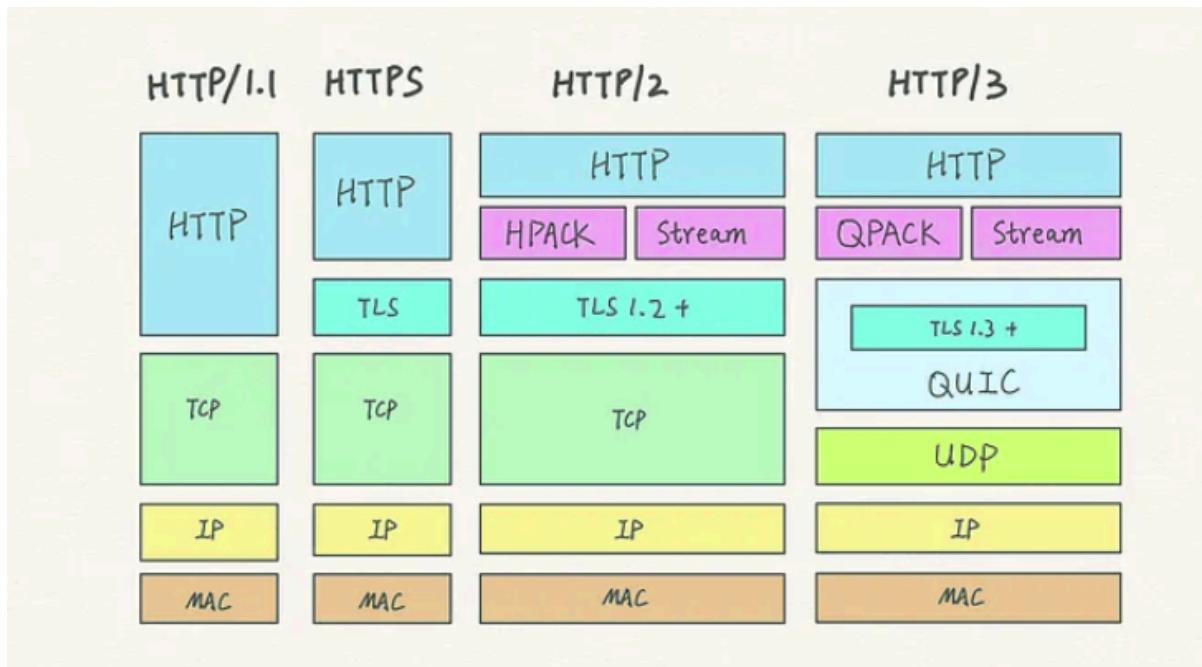
Если все пойдет хорошо, все сегменты будут доставлены на другой конец.

Однако интернет может быть нестабильным. Некоторые сегменты все-таки могут быть потеряны в процессе.

TCP имеет функцию, гарантирующую доставку. Он помещает полученные сегменты в буфер и ожидает повторной передачи потерянных сегментов, что и приводит к блокировке начала очереди.

Для решения этой проблемы нужно чем-нибудь заменить TCP — QUIC и UDP.

Обновленный стек протоколов



Мы видим значительное изменение стека протоколов: TCP был заменен на UDP.

В отличие от TCP, UDP не гарантирует доставку и не создает каких-либо зависимостей между сегментами. Это означает, что никаких блокировок начала очереди не может быть в принципе.

Кроме того, поскольку UDP является сетевым протоколом передачи без установления соединения, никакого рукопожатия не требуется. По этому он работает быстрее, чем TCP.

В дополнение к UDP был введен новый протокол QUIC. Он наследует сильные стороны TCP, среди которых управление соединениями и потоками данных. Кроме того, QUIC реализует функции, гарантирующие доставку данных, которых недостает UDP.

Другое важное изменение заключается в том, что TLS со всеми своими функциями обеспечения безопасности теперь интегрирован прямо внутри QUIC. Поскольку TLS 1.3 уже готов к работе в продакшнене, QUIC внедряет именно эту версию.

И последнее, но не менее важное: QPACK заменяет собой HPACK, что еще больше повышает производительность алгоритма сжатия заголовков. Количество его записей в статической таблице увеличено с 61 до 98, и теперь он имеет нулевой индекс.

Сравнение HTTP1 & HTTP2 & HTTP3

Взято [отсюда](#).

| Comparison of HTTP protocol stack changes from HTTP/1.1 to HTTP/3 | | |
|---|---|--|
| HTTP 1.1 | HTTP 2 | HTTP 3 |
| <p>Some methods and response codes are added.</p> <p>“Keep Alive” becomes officially supported. “Host” header is supported for Virtual Domain.</p> <p>Syntax and semantics are separated.</p> | <p>Support of parallel request transmission by “stream” (addressing HoL blocking issue of HTTP requests).</p> <p>Addition of flow-control function in units of “Stream”.</p> <p>Addition of prioritization function in units of “Stream”.</p> <p>Addition of PUSH function (send related file without request).</p> | <p>Lower protocol changes from TCP+TLS to UDP+QUIC.</p> <p>“Streams” and “flow-control function” are prioritized.</p> <p>Parallel request transmission is supported by QUIC stream (addressing HoL blocking issue of TCP packets).</p> |

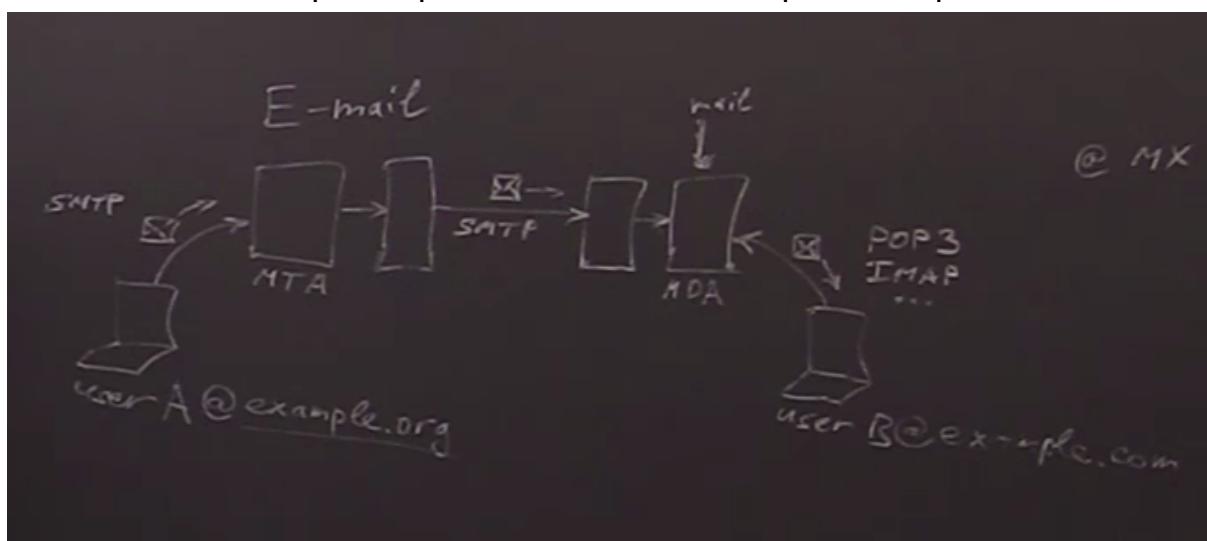
Keep-alive: Постоянное HTTP-соединение, также называемые HTTP keep-alive или повторное использование соединений HTTP — использование одного TCP-соединения для отправки и получения многократных HTTP-запросов и ответов вместо открытия нового соединения для каждой пары запрос-ответ.

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

взято из [11 лекции](#)

Про email

У нас есть 2 пользователя А & В. Один другому хочет отправить письмо, надо знать идентификаторы пользователей - адрес электронной почты.



Как пользователю А узнать, как отправить письмо получателю В? На самом деле надо понимать, что письмо дойдет до какого-то сервера, на котором оно хранится, пока пользователь В его не заберет. Сервер будет называться MDA (Mail Delivery Agent).

Пользователь А будет отправлять свое письмо на сервер MTA (Mail Transfer Agent), который должен будет понять, как доставить письмо до В. Понимать он будет это так: он возьмет в dns mx запись которая соответствует домену, который является частью email адреса. То есть он будет разрешить такую запись:

@MX 10 mail.example.com. Когда MTA разрешит эту запись, он отправит запрос на подключение к серверу MDA, и отправит ему это письмо. На этом сервере письмо будет лежать, пока его оттуда не заберет пользователь В.

Такое подключение-передача-забирание письма работает по разным протоколам.

Со стороны **отправителя** используется SMTP - Simple Mail Transfer Protocol.

Со стороны **получателя** используется ***POP3 - Post Office Protocol Version 3 or IMAP - Internet Message Access Protocol*** (более современный протокол).

Такая схема стала очень популярной, потому что любой человек/организация может стать участником этой схемы, для этого достаточно иметь доменное имя, настроить какой-то сервер и потом можно будет отправлять сообщения.

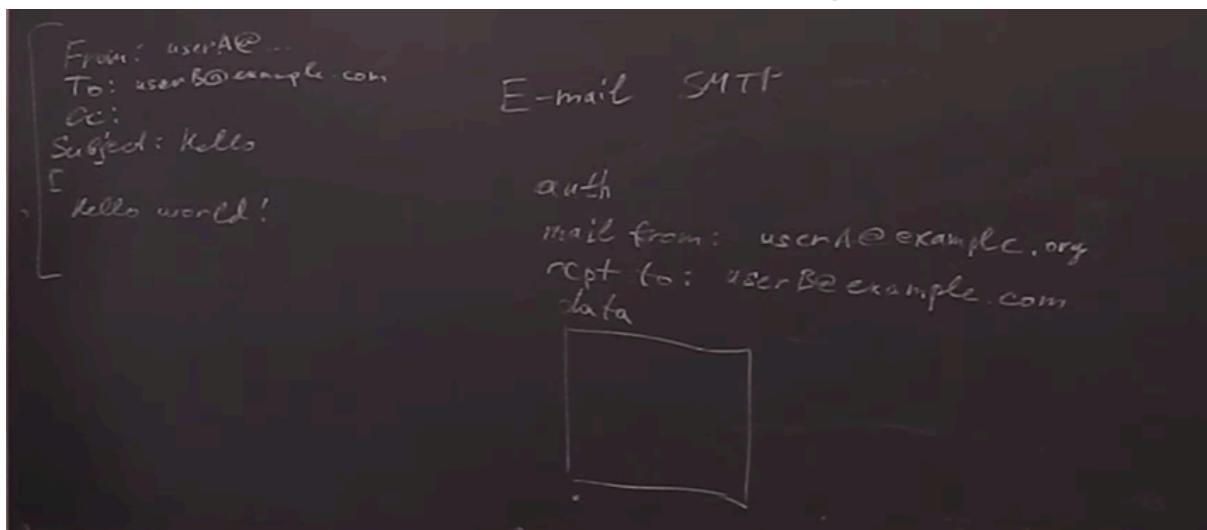
SMTP

Simple Mail Transfer Protocol

Используется для отправки между почтовыми серверами и для того, чтобы принимать письма от пользователя (клиента).

Простой пример взаимодействия: пользователь отправляет письмо. Что происходит?

- 1) Аутентификация (auth)
- 2) Говорим, что хотим отправить письмо пользователю.
указываем кто мы: **mailfrom: userA@example.org**
указываем получателей, которым это письмо хотим доставить, они могут быть в разных доменах: **mailto: userB@example.org**



Еще на заре развития протокола за ним закрешили два номера порта:

Первый — это порт 25, посредством которого почта передается между почтовыми серверами.

Второй — порт 587, благодаря которому почта передается от почтового клиента на сервер.

Сообщение электронной почты всегда состоит из трех элементов:

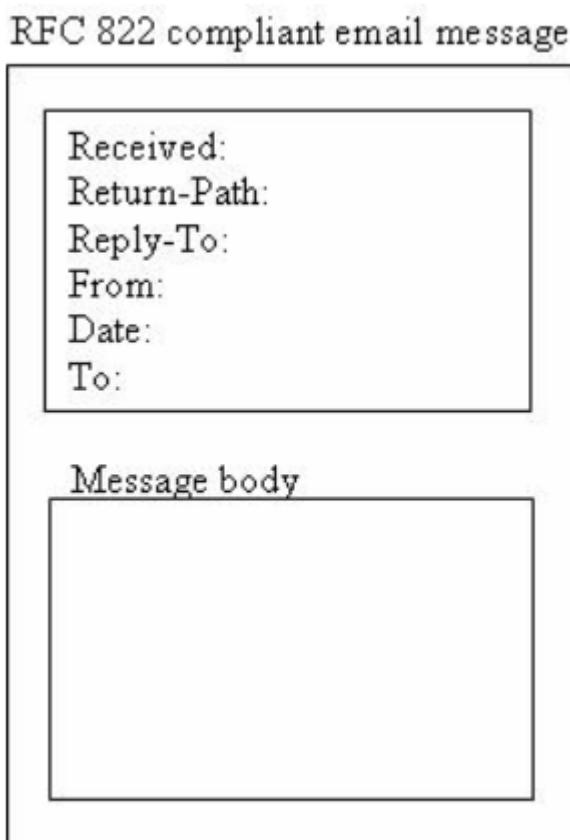
Так называемый конверт.

Заголовок.

Тело письма.

Конверт применяется для передачи сообщений электронной почты от клиенту к серверу и обратно.

При этом как клиент, так и сервер взаимодействуют друг с другом в обычном формате «запрос-ответ». В ходе взаимодействия между ними передаются текстовые строки. Что касается заголовка и тела письма, то их параметры прописаны в отдельном документе — RFC2822.



Формат поля заголовка

Received:

Received:
From host
by host
via physical-path
with protocol
id message-id
for final e-mail destination

Это поле используется для

идентификации тех SMTP-серверов, которые были задействованы в процессе доставки сообщения от отправителя к получателю. Любой сервер из цепочки добавляет к сообщению собственное поле Received, где можно видеть техническую информацию об этом сервере.

Return-Path — поле возврата, которое используется для определения маршрута, по которому прошло сообщение. Если оно было отправлено прямо на сервер получателя, то в поле отображается один адрес. Если же серверов несколько, они будут отображаться списком.

Команды SMTP

Все они состоят из четырех символов. Сакрального замысла здесь нет, просто в самом начале разработчики договорились, что длина будет

именно такой. Ниже — основные команды SMTP. Есть и другие, но они используются реже.

- 1) Команда **Hello** применяется для установки соединения. Эта операция будет выполнена только в том случае, если клиент указал свой домен и собственный почтовый адрес.
- 2) Команда **Mail** применяется для задания адреса отправителя.
- 3) Команда **RCPT** используется исключительно для того, чтобы прописать адрес получателя. Электронное сообщение можно передать сразу нескольким получателям, для чего требуется использовать команду RCPT несколько раз подряд.
- 4) Команда **DATA** нужна для уведомления принимающего сервера о завершении конверта, после чего идет само письмо.
- 5) Команда **QUIT** применяется для разрыва соединения с сервером сразу после завершения приема сообщения.

Ответы SMTP

Здесь все одновременно и проще, и сложнее. Ответы в случае SMTP состоят из двух частей:

- 1) Код сообщения. Дает возможность изучить корректность и правильность отправки.
- 2) Текстовое сообщение. Объясняет, что произошло в ходе отправки или получения. Как правило, сообщение формируется для того, что произошло. В подавляющем большинстве случаев такое сообщение предназначено для людей, а не компьютеров.

Коды сообщений начинаются на 2, 3, 5. Если сообщение начинается на 2, это значит, что предыдущая команда **успешно завершена**. «Тройка» в коде означает успешную отправку **с необходимостью предоставить дополнительные данные**.

Если сообщение начинается на 5, это означает **технический сбой**. Так, ошибка 502 — индикатор нереализованной команды, а 503 сообщает о неправильной последовательности команд.

Как работает SMTP — простыми словами

Давайте представим, что вы установили и настроили собственный SMTP-сервер. Далее вы планируете отправить письмо. Работает отправка по определенному алгоритму:

- 1) Указывается адрес отправителя, после чего система пользователя соединяется, к примеру, с SMTP почтового клиента Gmail.

- 2) Система передает серверу данные, включая email отправителя и получателя, тему письма, его содержимое.
- 3) Сразу после этого система начинает поиск SMTP-сервера получателя электронного сообщения.
- 4) Если этот сервер не найден или он не отвечает, SMTP-сервер пытается предпринять еще несколько попыток связи. Если ничего не получается, то система выдает ошибку отправки. При этом протокол сообщит, почему письмо не будет доставлено. Так, проблема может быть в несуществующем адресе или в блокировке сообщений.

Если все хорошо, то далее в работу вступают уже другие протоколы — POP и IMAP

Про MIME

Читать про протокол [MIME](#) - Multipurpose Internet Mail Extensions - это [стандарт](#), описывающий передачу различных [типов данных](#) по [электронной почте](#), а также, в общем случае, спецификация для кодирования информации и форматирования сообщений таким образом, чтобы их можно было пересылать по [Интернету](#).

Используют кодировку [base64](#). Украдено из вики:

Base64 — стандарт кодирования двоичных данных при помощи только 64 символов [ASCII](#). [Алфавит кодирования](#) содержит [латинские символы](#) A-Z, a-z, цифры 0-9 (всего 62 знака) и 2 дополнительных символа, зависящих от системы реализации. Каждые 3 исходных байта кодируются четырьмя символами (увеличение на $\frac{1}{3}$).

Эта система широко используется в [электронной почте](#) для представления бинарных файлов в тексте письма ([транспортное кодирование](#)).

В письме могут быть прикрепленные ресурсы: файлы, картинки и тд. Они кодируются с помощью multipart MIME

POP3 & IMAP Используются MUA для того, чтобы подключиться к MDA.

Про POP3

Post Office Protocol Version 3

Очень старый протокол, маленький набор функций.

```
auth  
list  
get N  
delete N  
quit
```

Нельзя пометить письмо как прочитанное/непрочитанное. Схема работы была такая: user agent подключается, просматривает письма. Те письма, которые ему не нужны, удаляет, те письма, что ему нужны, просматривает и потом тоже удаляет. А потом хранит их где-то у себя в харнилище. Короче, предполагалось, что письма на MDA не хранятся какое-то большое время.

Но потом (разрабы) поняли, что пользователи хотели бы хранить письма на сервере. Тогда придумали протокол IMAP.

Как работает протокол POP3

POP3 работает через стек TCP/IP и обычно использует порт 110 для незашифрованных соединений или порт 995 для зашифрованных соединений. Когда почтовый клиент настроен на использование POP3, он подключается к почтовому серверу и загружает все сообщения в папку входящих сообщений пользователя. Затем эти сообщения сохраняются на компьютере или устройстве пользователя и обычно удаляются с сервера электронной почты. Некоторые почтовые клиенты могут быть настроены так, чтобы оставлять копии сообщений на сервере. Важно отметить, что POP3 работает только с получением электронной почты и не поддерживает отправку электронной почты. Для отправки электронной почты обычно используют SMTP.

Команды POP3

USER — команда используется для определения имени пользователя для учетной записи электронной почты.

PASS — команда для указания пароля учетной записи email.

LIST — используется для получения списка сообщений электронной почты на сервере.

RETR — команда используется для получения определенного сообщения электронной почты с сервера.

DELE — используется для пометки определенного email-сообщения для удаления с сервера.

QUIT — команда для завершения сеанса POP3.

Распространенные ответы POP3

- OK — ответ означает, что предыдущая команда была выполнена успешно.
- ERR — ответ указывает на то, что предыдущая команда не была успешной.
- Ответ на команду LIST предоставляет список, уникальные идентификаторы и размеры почтовых сообщений на сервере.
- Ответ на команду RETR предоставляет полный текст указанного email-сообщения.
- Ответ на команду DELE подтверждает, что указанное сообщение было помечено на удаление.

Преимущества POP3

- POP3 предоставляет автономный доступ. Пользователи могут загружать свои email-сообщения и получать к ним доступ в автономном режиме, для этого не требуется активного подключения к интернету.
- Также пользователи могут контролировать объем памяти, используемой на сервере электронной почты, загружая и удаляя сообщения с сервера.
- Высокая совместимость — еще одно преимущество POP3. Широко используемый протокол совместим с большинством почтовых клиентов и серверов. Некоторые из них могут поддерживать расширенные версии POP3, такие как Authenticated POP или Secure POP. Они обеспечивают шифрование и проверку целостности сообщений.

Недостатки POP3

- POP3 не обеспечивает синхронизацию между почтовыми клиентами и серверами. После загрузки сообщения любые изменения, внесенные в него на почтовом сервере (например, удаление или пометка как прочитанное), не будут отражены на почтовом клиенте.
- Поскольку сообщения загружаются и удаляются с сервера, пользователь может получить доступ к своим сообщениям только с того устройства, на которое оно загружено. К тому же некоторые почтовые серверы могут иметь ограничения на объем дискового пространства, доступного для каждого пользователя. Это может ограничить количество сообщений, которые можно загрузить с помощью POP3.
- POP3 не предоставляет надежных средств защиты, таких как шифрование или двухфакторная аутентификация, что делает учетные записи электронной почты уязвимыми для взлома и других угроз безопасности. Но уровень безопасности могут повысить расширенные версии протокола.

Про IMAP

В нём есть

- 1) иерархическая система папок, которые можно создавать/удалять/переименовывать.
- 2) Есть метка: прочитано/не прочитано письмо.
- 3) Можно получить часть письма. Типа получаем часть письма, а аттач оставляем на сервере. В POP3 такой возможности нет.
- 4) Можем узнать, когда появляются новые письма. Опять же, в POP3 такой возможности нет. В POP3 чтобы узнать, что появились какие-то новые письма.

Как работает IMAP?

IMAP — протокол, используемый для доступа к email-сообщениям, хранящимся на сервере электронной почты. Протокол позволяет пользователям получать доступ к сообщениям электронной почты непосредственно на сервере электронной почты

IMAP работает путем поддержания постоянного соединения между клиентом и сервером. Когда клиент запрашивает доступ к электронному сообщению, сервер отправляет копию сообщения клиенту. Когда пользователь удаляет или перемещает сообщение на одном устройстве, сервер обновляет статус сообщения, чтобы все другие устройства, имеющие доступ к той же учетной записи электронной почты, отразили эти изменения.

IMAP обычно использует порт 143 для незашифрованных соединений или порт 993 для зашифрованных соединений.

Команды IMAP

- LOGIN — команда используется для аутентификации учетных данных пользователя.
- SELECT — команда для выбора почтового ящика для чтения или записи сообщений.
- FETCH — эту команду используют для получения содержимого определенного сообщения.
- STORE — используется для изменения состояния сообщения, например, пометить его как прочитанное или переместить в другую папку.
- EXPUNGE — команду используют для окончательного удаления всех сообщений, которые были помечены для удаления.
- LOGOUT — эта команда завершает сеанс IMAP.

Распространенные ответы IMAP

- OK — ответ означает, что предыдущая команда была выполнена успешно.
- NO — предыдущая команда не была успешной, но неудача не вызвана критической ошибкой.
- BAD — ответ указывает неудачу предыдущей команды, и ее не следует повторять.
- Ответ на команду SELECT предоставляет информацию о выбранном почтовом ящике, включая имя и количество сообщений.
- Ответ на команду FETCH предоставляет содержимое указанного сообщения.
- Ответ на команду STORE подтверждает, что указанное сообщение было изменено.

Преимущества IMAP

- IMAP позволяет юзерам получать доступ к своим email-сообщениям с различных устройств, поскольку все сообщения хранятся на сервере электронной почты. Протокол поддерживает синхронизацию в режиме реального времени: изменения, внесенные в сообщение на одном устройстве, отражаются на всех устройствах, имеющих доступ к учетной записи.
- IMAP позволяет создавать и управлять папками на сервере электронной почты, что облегчает организацию и поиск email-сообщений. Также протокол поддерживает расширенные возможности поиска, облегчая поиск определенных писем.

Недостатки IMAP

IMAP требует постоянного подключения к интернету, поскольку сообщения хранятся на сервере электронной почты и доступ к ним осуществляется в режиме реального времени. Поскольку все сообщения хранятся на сервере электронной почты, пользователи могут иметь ограниченный контроль над использованием пространства для хранения. Хранение всех сообщений электронной почты на сервере может представлять риск для безопасности в случае взлома сервера.

Чутка про безопасность

Протоколы POP3/IMAP - довольно старые, поэтому когда они создавались, они не использовали шифрование. Поэтому были сделаны их расширения с поддержкой TLS.

- 1) Клиент инициирует TLS-handshake и потом происходит обмен в рамках какого-то протокола, например, https.
Но для этого надо будет выделить отдельный порт, потому что если этого не сделать, то клиенты не смогут знать, как пользоваться этим TLS-ом. Пользователям придется угадывать, TLS перед ними или нет.
В http это 443 порт, 80 порт - для plain text.
- 2) Не хотим выделять отдельный порт. Во многих протоколах предусмотрена возможность апгрейда до TLS. И этот механизм называется [STARTTLS](#). Идея в том, что сначала подключаемся по исходному протоколу без шифрования и даем какую-то специальную команду, которая говорит “а давайте-ка перейдем в режим TLS”. Но для начала надо спросить, а поддерживает ли сервер такой переход в режим TLS. Если сервер не поддерживает, то придется либо отказаться от работы с этим сервером, либо продолжить работу в незашифрованном виде.

В SMTP, например, есть команда EHLO (типа extended hello), которая выдает список поддерживаемых сервером расширений. Там есть UTF-8, поддержка starttls.
После того, как выполняем команду STARTTLS, сервер говорит “OK”. Тогда то же самое TCP соединение используется для того, чтобы сделать TLS-handshake.

Про мгновенные сообщения

Например, ватсап - централизованная система, если ватсап умрет, то пользователи не смогут общаться, они не смогут даже свой сервер поднять, чтобы продолжить обмениваться сообщениями, тк все протоколы закрытые.

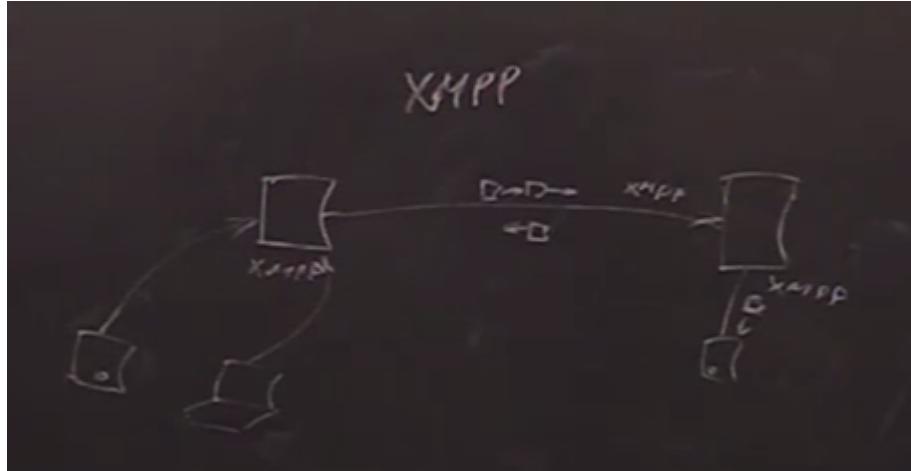
XMPP

Протокол для обмена быстрыми сообщениями.
Работает похоже на почту.
Должно быть доменное имя, в котором прописывается адрес xmpp-сервера,. Есть клиенты, которые аутентифицируются. И когда кто-то кому-то хочет отправить сообщение, отправитель добавляет получателя в адресную книгу и сервер проверяет, что сервер получателя правда существует. Как он это делает? Также берет dns-запись, и смотрит, если там есть запись, которая говорит о том, что есть

xmpp-сервер, который обслуживает пользователей в этом домене, то сервер позволяет с этим пользователем общаться. И когда отправитель отправляет сообщение, то тогда устанавливается соединение между этими серверами, и тогда передаются эти сообщения. Здесь соединение будет персистентное [умное слово]

Персистентное соединение - это постоянное соединение, когда и сервер,

и клиент в любой момент могут



отправить/принять данные.

То есть если в SMTP соединение устанавливается тогда, когда есть письма, то здесь (в xmpp) они достаточно существенное время будут жить, потому что предполагается, что отправка сообщений должна быть быстрой. При мгновенных сообщениях если каждый раз будут устанавливаться соединения между серверами, то это будет не очень быстро, что противоречит идеи создания мгновенных сообщений. Проблема в распространенности (популярности) мгновенных сообщений: раньше были +- популярны, но сейчас не очень. Возможно, дело в коммерции()

Про спам

Способы борьбы со спамом. Если говорить про электронную почту, то применяется

SPF - Sender Policy Framework

SPF - Sender Policy Framework - стандарт, который описывает правила, по которым можно определить, не подделан ли [домен](#) отправителя. Здесь проверяем отправителя, его ip адрес, что он соответствует тому серверу, который к нам подключился.

Правила задаются в ресурсных записях DNS. В том же самом домене, в котором у нас находятся пользователи, мы можем создать spf-записи и

указать, какие почтовые сервера могут отправлять письма, указывая в качестве отправителя пользователя из этого домена.

Тогда получится, что тот почтовый сервер, который принимает письмо, если он видит, что в этом письме в качестве отправителя указан какой-то адрес в этом домене, он идет и смотрит SPF-запись. Если ее нет, то он не может применить SPF-правила, записи-то нет. Тогда применяются какие-то другие правила. Если есть SPF-запись, то он смотрит: если тот, сервер, который ему это письмо приспал, совпадает с правилами, то тогда доставка этого письма разрешается. Если нет - то считается, что это спам.

Способ отслеживания спам-писем с помощью SPF работает на этапе приема письма сервером. Но а что если у нас цепочка серверов? Тогда мы не сможем узнать, с какого именно сервера пришла информация.

DKIM (DomainKeys Identified Mail)

(здесь проверяется идентичность каждого сервера в цепочке, через которую это письмо проходило. Это может быть как сервер отправителя, так и последующие сервера-отправители.)

Есть способ, который заключается в том, что мы публикуем публичный ключ в системе DNS и этим ключом наш сервер подписывает все исходящие из него письма. Такой способ называется digital key interchange. Здесь идет проверка цифровой криптографической подписи. И тогда любой получатель, который это письмо получит, сможет проверить публичный ключ с тем доменным ключом, который находится в ресурсных записях DNS. Если они совпадают, то значит, что это письмо было подписано тем сервером, который указан в DNS.

Но это тоже не столько гарантирует, что письмо не спам. Так кто-то мог заставить наш сервер отправить это письмо-спам, подписать его, кто-то украл учетную запись, например. Цифровая подпись публикуется в виде заголовка.

DMARC

Domain-based Message Authentication, Reporting and Conformance

(идентификация сообщений, создание отчетов и определение соответствия по доменному имени)

Это техническая спецификация, созданная группой организаций, предназначенная для снижения количества [спамовых](#) и [фишинговых](#) [электронных писем](#), основанная на идентификации почтовых доменов отправителя на основании правил и признаков, заданных на почтовом сервере получателя.

DMARC позволяет противостоять фишингу — мошенничеству, целью которого является кража конфиденциальных данных пользователя (логинов, паролей, данных кредитных карт). Главный инструмент фишинга — email-рассылки. Обычно злоумышленники маскируют свои письма под сообщения известных компаний, используя их домены. Если пользователь следует инструкциям из такого письма, он теряет личные данные и нередко деньги. А компания получает существенный репутационный ущерб.

Если же у компании настроен DMARC, то письмо, которое отправят мошенники от ее имени, либо вовсе не будет доставлено, либо будет помечено как подозрительное.

Как работает DMARC?

DKIM работает так: в письме есть зашифрованные данные о том, кем и когда было отправлено письмо. Почтовый провайдер, Gmail или Mail.ru, получает эти данные вместе с письмом. Провайдер расшифровывает их с помощью публичного ключа, выложенного на домене, с которого отправлено письмо. Если данные совпадают — значит, это честный отправитель, письмо можно пропускать во «Входящие». Если нет — мошенник, письмо отправляется в «Спам».

SPF показывает, разрешено ли конкретному серверу отправлять письма с этого домена. Сервер определяется по IP-адресу. Например, когда вы отправляете рассылку через Unisender или настраиваете корпоративную почту на Mail.ru, вы делегируете серверам Unisender и Mail.ru право отправлять письма с вашего домена.

Теперь разберемся с DMARC. Эта запись:

- указывает почтовому провайдеру, что делать с письмом в зависимости от результатов прочтения DKIM и SPF;
- говорит серверу отправить отчет на почту администратора домена (то есть вам или вашему системному администратору) с информацией, какие письма были отправлены и как провайдер поступил с письмами.

Как провайдер проверяет письма с учетом настроек DMARC

Допустим, вы отправили рассылку через Unisender пользователю на Mail.ru.

После того как письмо получает провайдер подписчика (Mail.ru), он проверяет репутацию домена, наличие email и домена в черных списках,

IP-адреса серверов, с которых отправлено письмо. В рамках этой проверки почтовый провайдер:

- Расшифровывает и верифицирует DKIM. Точно ли от этого домена отправлено письмо, или это подделка.
- Расшифровывает и верифицирует SPF. Разрешено ли слать письма от имени этого домена этому IP.
- Применяет политику, прописанную в DMARC. Допустим, в DMARC написано отправить в «Спам» тех, у кого DKIM не совпадает, и отослать отчет об этом администратору домена.

Далее к письму применяются стандартные спам-фильтры.

Варианты развития событий после проверки:

- Письмо пропущено и попадает во «Входящие». Если DKIM и SPF в порядке, а спам-фильтры пройдены.
- Письмо добавлено в карантин (в «Спам»). Если DKIM не совпадает и/или спам-фильтры не пройдены.
- Письмо отклонено (не доставлено). Индивидуальные причины: к примеру, у пользователя забит почтовый ящик.

После распределения писем отправителю высылается автоматический отчёт, где написано, что произошло с отправленными письмами.

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

Самый популярный протокол транспортного уровня. Контролирует доставку пакетов.

** просто инфа пусть будет: протокол SMTP - протокол уровня приложения, работает поверх TCP. HTTP работает тоже поверх TCP **

TCP контролирует доставку пакетов. Если данные вдруг не были доставлены до получателя, то TCP их отправляет заново. До тех пор, пока они не доставятся.

TCP контролирует порядок. Это значит, что нам не надо думать о том, в каком порядке придут пакеты. TCP основан на IP, поэтому в итоге все данные, которые отдаем TCP, он отправляет с помощью IP: все данные нарезает на какие-то фрагменты, кладет в IP пакеты и потом они путешествуют. TCP переставит пакеты в нужном порядке сам. Здесь TCP гарантирует, что никакие байты, которые мы отправляли, не перемешаются.

TCP - это сессионный протокол (в том числе). В случае UDP нет никакой сессии между отправителем и получателем: любой отправитель в любой момент может отправить любой пакет любому получателю. И этот пакет может дойдет, а может и нет.

А чтобы общаться по TCP, сначала нужно договориться о соединении. И чтобы соединение завершить, надо тоже договориться о том, что соединение будет завершено, что данные передаваться больше не будет.

TCP - потоковый протокол.

Это значит, что мы оперируем не какими-то фрагментами, что вот мы хотим отправить 10 байтов, фигак-фигак, отправили их и получатель и получил ровно эти 10 байтов. То есть в случае UDP пакет является целостной сущностью транспортного протокола.

В случае TCP сущностью является не пакет, а поток. То есть мы устанавливаем соединение и говорим, что вот, начался поток байтов. Потенциально, этот поток может быть бесконечным, нет никакого ограничения на его размер. В реальной жизни это не так, тк обязательно

что-то пойдет не так: приложение умрёт, кабель перережут, связь нарушится и тд.

Короче говоря, приложение оперирует только потоком.

TCP - это протокол “один-к-одному”. У нас есть участник-приложение и еще один участник, который может расположен в другом приложении. И они между собой связываются. То есть нельзя сделать ни мульти cast, ни бродкаст. Потому что как это реализовывать, как обеспечить контроль доставки, если мы даже не знаем, кто есть в группе. TCP этого не умеет.

В TCP есть стороны, которые общаются, они имеют разные роли. В UDP роли одинаковые: любой участник может отправлять любому участнику что угодно; от любого, что угодно получить.

В TCP есть активная сторона (та, что устанавливает соединение) и пассивная (та, что сидит и ждет, пока к ней кто-нибудь подключится).

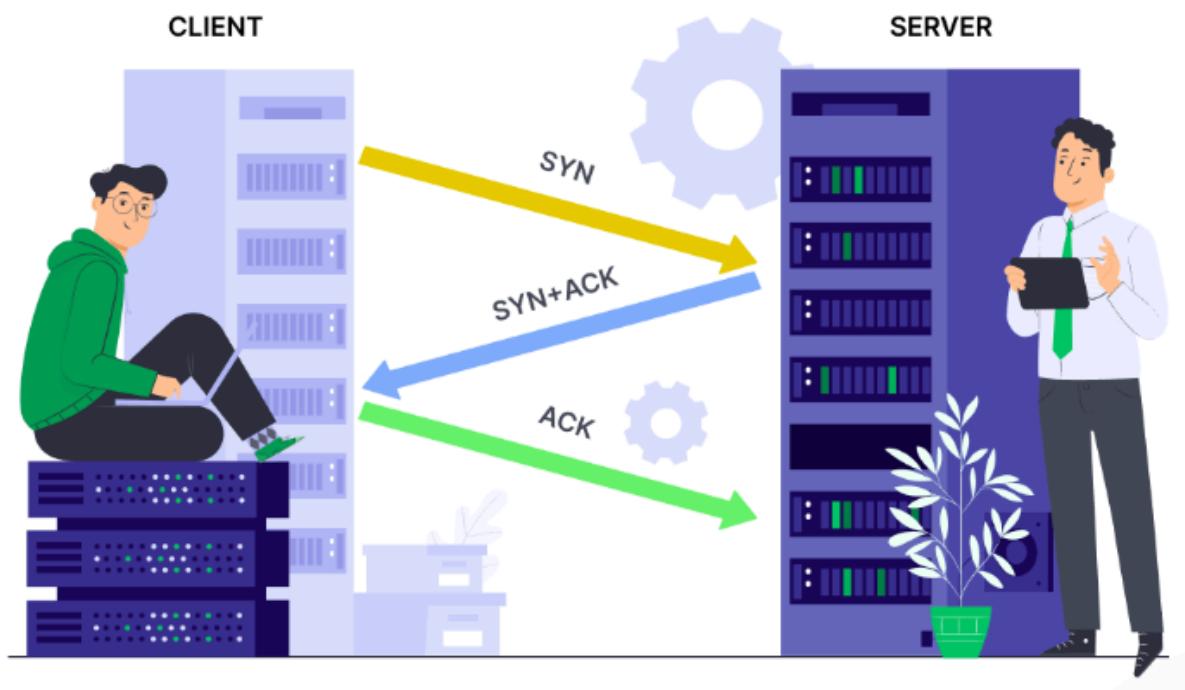
Пример: в http браузер - активная сторона, потому что будет пытаться установить соединение с каким-то сервером. А сервер - будет пассивной стороной, тк будет ждать, пока к нему придут клиенты, которых он их обслужит.

Установление соединения

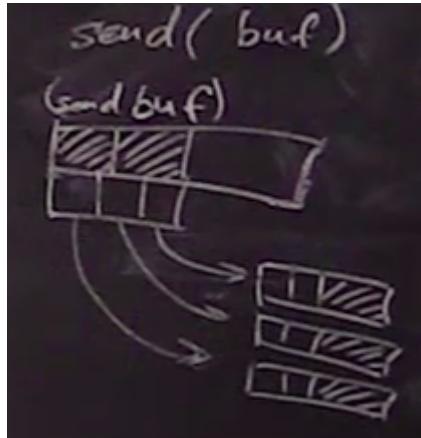
Тройное рукопожатие aka TCP three-way/triple handshake

Подробнее почитать [здесь](#).

| Схема тройного рукопожатия | | |
|------------------------------------|--------------------------------------|--|
| активная сторона (отправитель) | | пассивная сторона (получатель) |
| инициирует установку соединения | -----> (запрос на подключение) | |
| | <----- | если готова общаться, то отправляет ответ |
| договорились, будем общаться | -----> | |
| соединение установлено | | |



Можно рассказать более детально про то, как происходит отправка.

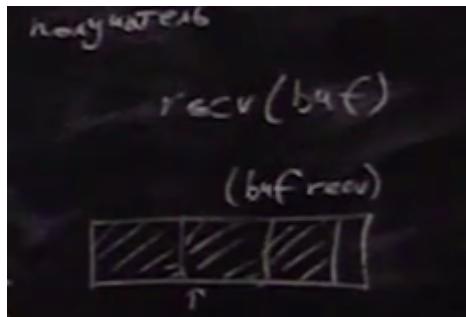


Есть какое-то приложение, которое говорит “давай че-нить отправим” и передает какой-то буфер с данными. То есть приложение хоть и оперирует с потоком, но оперирует с ним кусочками.

Операционная система складывает это сообщение в какой-то буфер на отправку (send buffer).

В какой-то момент ОС принимает решение о том, что сообщение пора отправлять, и тогда она нарезает все, что есть, на фрагменты нужного размера. То есть каким-то образом реализация TCP может узнавать (или угадывать всякими алгоритмами подбора), какое MTU на нашем маршруте. И в соответствии с MTU ОС нарезает на фрагменты (на 3 куска, например). Каждому добавляет заголовки, кладет туда кусочки

данных и отправляет. В каждом таком пакете написан номер байта, который будет первым в потоке.



У получателя есть буфер для получения. Он получает эти пакеты, смотрит, какой номер в потоке у получаемого пакета и кладет его в нужное место. Допустим, пришли все пакеты, и он их положил в свой буфер.

После этого приложение обращается к ОС и говорит “а что-нить мне прислали в рамках этого соединения?” и вызывает функцию `recv()` - сискол. И говорит “положи мне в этот буфер все то, что ты получил”. ОС смотрит, если че-то есть в буфере, то копирует в буфер приложения.



Допустим, пришли не все пакеты. Как узнаем, какой кусок не пришел? По номеру первого байта, который определяет сдвиг “разрезания” передаваемых данных. Получатель замечает, что образовалась дырка. Тогда получатель говорит отправителю: “вот дырка, дай мне данные”. Отправитель берет и отправляет нужные данные. Но! перед этим заново происходит нарезка данных. А не так, что вот с такого-то бита берем и отправляем данные, которые не дошли до получателя. А почему так? Почему нарезка происходит заново? Почему не запомнить, что вот с такого-то бита все норм отправилось, а с такого-то - нет. И просто переотправить нужный кусок. Ответ. Потому что ситуация может меняться. TCP может узнать о том, что на самом деле MTU, в соответствии с которым он сделал нарезки, не такой, а меньше, потому что пакет мог потеряться, потому что не пролез по какой-то линии связи. И тогда TCP может сделать вывод о том, что mtu надо уменьшить.

Завершение соединения

При нормальном завершении TCP-соединения в большинстве случаев инициализируется процедура, называемая двухсторонним рукопожатием,

в ходе которой каждая сторона закрывает свой конец виртуального канала и освобождает все задействованные ресурсы. Обычно эта фаза начинается с того, что один из задействованных процессов приложения сигнализирует своему уровню TCP, что сеанс связи больше не нужен. Со стороны этого устройства отправляется сообщение с установленным флагом FIN (отметим, что этот пакет не обязательно должен быть пустым, он также может содержать полезную нагрузку), чтобы сообщить другому устройству о своем желании завершить открытое соединение. Затем получение этого сообщения подтверждается (сообщение от отвечающего устройства с установленным флагом ACK, говорящем о получении сообщения FIN). Когда отвечающее устройство готово, оно также отправляет сообщение с установленным флагом FIN, и, после получения в ответ подтверждающего получение сообщения с установленным флагом ACK или ожидания определенного периода времени, предусмотренного для получения ACK, сеанс полностью закрывается. Состояния, через которые проходят два соединенных устройства во время обычного завершения соединения, отличаются, потому что устройство, инициирующее завершение сеанса, ведет себя несколько иначе, чем устройство, которое получает запрос на завершение. В частности, TCP на устройстве, получающем начальный запрос на завершение, должен сразу информировать об этом процесс своего приложения и дождаться от него сигнала о том, что приложение готово к этой процедуре. Инициирующему устройству не нужно это делать, поскольку именно приложение и выступило инициатором.

Заголовок TCP

Заголовок сегмента TCP [[править](#) | [править код](#)]

Структура заголовка

| Бит | 0 – 3 | 4 – 6 | 7 – 15 | 16 – 31 |
|----------|---------------------------------------|--|--------|--|
| 0 | Порт источника, Source Port | | | Порт назначения, Destination Port |
| 32 | | Порядковый номер, Sequence Number (SN) | | |
| 64 | | Номер подтверждения, Acknowledgment Number (ACK SN) | | |
| 96 | Длина заголовка, (Data offset) | Зарезервировано | Флаги | Размер Окна, Window size |
| 128 | | Контрольная сумма, Checksum | | Указатель важности, Urgent Point |
| 160 | | Опции (необязательное, но используется практически всегда) | | |
| 160/192+ | | | Данные | |

- Window size (16 бит): размер окна приема. В нем указывается количество байт данных, считая от последнего номера подтверждения, которые готов принять отправитель данного

пакета. Другими словами, отправитель данного пакета в этом поле сообщает другой стороне, каким доступным на данный момент размером буфера приема данных он располагает.

- Checksum (16 бит): контрольная сумма. Используется для проверки на наличие ошибок при передаче и/или приеме отправленного пакета. Рассчитывается с учетом заголовка (все поля заголовка, кроме самой контрольной суммы), полезной нагрузки (неслужебные данные с полезной информацией, которая, собственно, и передается), а также псевдо-заголовка (IP-адрес источника, IP-адрес назначения, номер протокола и длина TCP-сегмента, в которой учитывается как длина полей заголовка, так и длина данных полезной нагрузки). Более детально о расчете контрольной суммы вы можете прочитать здесь (http://www.tcpipguide.com/free/t_TCPChecksumCalculationandtheTCP_PseudoHeader-2.htm).
- Urgent pointer (16 бит): указатель срочности. Если установлен флаг URG, то это означает, что поле указателя срочности содержит численное значение положительного смещения от порядкового номера в сообщении, указывающее на последний байт срочных данных. После получения TCP-сегмента с флагом URG, установленным в значение «1», приемное устройство смотрит на поле указателя срочности и по его значению определяет, какие данные в сегменте являются срочными. Затем эти срочные данные сразу же направляются в приложение пользователя с указанием того, что отправитель пометил данные как срочные. Остальные данные в данном сегменте, как, к слову, и накопившиеся до этого в буфере приема, обрабатываются в нормальном режиме. Этим принцип обработки в сообщении флага URG отличается от обработки флага PSH, при получении которого вся информация из буфера, а не только срочная из сообщения, немедленно передается в приложение пользователя. Более детально об передаче данных в TCP при установленном указателе срочности вы можете узнать здесь (http://www.tcpipguide.com/free/t_TCPPriorityDataTransferUrgentFunction-2.htm).

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

взято из [7й лекции](#) и моей презентации, которую я когда-то делала
кодировки

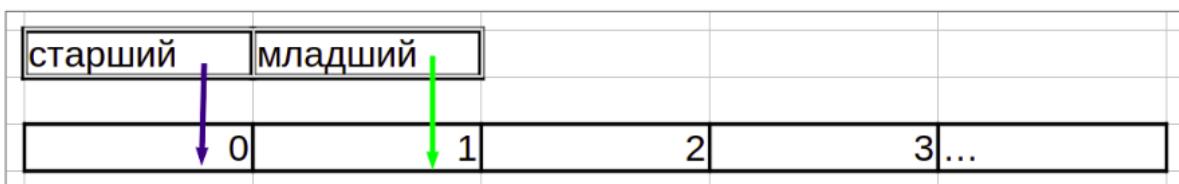
В качестве данных нужно отправлять тексты/числа на естественных языках и нужно понимать, как они представляются в виде байтов.

byteorder (endianness)

BYTEORDER (ENDIANNES)

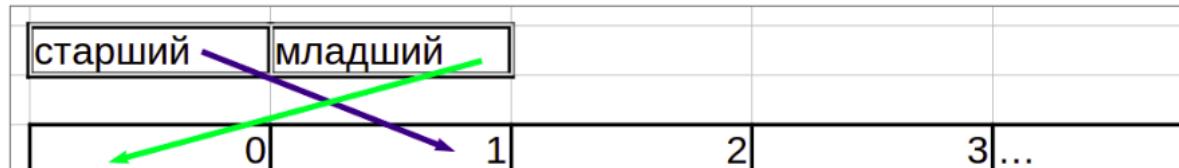
Старший – младший

BIG ENDIAN



Младший – старший

LITTLE ENDIAN



Есть число, которое занимает 2 байта. У этого числа есть младший байт и старший байт. И у нас есть какой-то поток байтов, в который мы хотим положить это число. Сначала можем положить старший байт, потом младший. Или сначала младший, потом старший.

Порядок от старшего к младшему (англ. big-endian):

запись начинается со старшего и заканчивается младшим. Этот порядок является стандартным для протоколов TCP/IP, он используется в заголовках пакетов данных и во многих протоколах более высокого уровня, разработанных для использования поверх TCP/IP. Поэтому, порядок байт от старшего к младшему часто называют сетевым порядком байт (англ. network byte order).

Порядок от младшего к старшему (англ. little-endian):

запись начинается с младшего и заканчивается старшим. Этот порядок записи принят в памяти персональных компьютеров с x86-процессорами, в связи с чем иногда его называют интеловский порядок байт

ВИДЫ ТЕКСТОВЫХ КОДИРОВОК

Есть текст, который мы хотим представить в виде байтов, а потом обратно декодировать их.

ASCII

ASCII — таблицы кодировок, в которых содержатся основные символы (английский алфавит, цифры, знаки препинания, символы национальных алфавитов(свои для каждого региона), служебные символы) и длина кода каждого символа

$n=8$ бит.

ASCII (AMERICAN STANDARD CODE FOR INFORMATION INTERCHANGE)



Кодировки стандарта ASCII (8 бит):

- **ASCII** — первая кодировка, в которой стало возможно использовать символы национальных алфавитов.
- **КОИ8-Р** — первая русская кодировка. Символы кириллицы расположены не в алфавитном порядке. Их разместили в верхнюю половину таблицы так, чтобы позиции кириллических символов соответствовали их фонетическим аналогам в английском алфавите. Это значит, что даже при потере старшего бита каждого символа, например, при проходе через устаревший семибитный модем, текст остается "читаемым".
- **CP866** — русская кодировка, использовавшаяся на компьютерах IBM в системе DOS.

«Альтернативная кодировка» («Альтернативная кодировка ГОСТ») — основанная на CP437 кодовая страница, где все специфические европейские символы во второй половине заменены на кириллицу, а псевдографические символы оставлены нетронутыми.

- **Windows-1251** — русская кодировка, использовавшаяся в русскоязычных версиях операционной системы Windows в начале 90-х годов. Кириллические символы идут в алфавитном порядке. Содержит все символы, встречающиеся в типографике обычного текста (кроме знака ударения).

● **ISO 8859** — семейство ASCII-совместимых кодовых страниц, разработанное совместными усилиями ISO (International Organization for Standardization) и IEC (Международная электротехническая комиссия (МЭК; англ. International Electrotechnical Commission, IEC)).

Применение:

Кодировки серии ISO 8859 применяются главным образом на юниксоподобных системах, а также для кодирования веб-страниц (поскольку большинство веб-серверов использует UNIX).

Состоит из 15 кодовых страниц.

Примеры кодовых страниц

ISO 8859-1 (Latin-1) - Расширенная латиница, включающая символы большинства западноевропейских языков

ISO 8859-2 (Latin-2) - Расширенная латиница, включающая символы центральноевропейских и восточноевропейских языков

....

ISO 8859-5 (Latin/Cyrillic) - Кириллица, включающая символы славянских языков (белорусский, болгарский, македонский, русский, сербский и частично украинский).

Преимущества ASCII

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

Недостатки ASCII

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

Область применения ASCII

- 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.

ПРО UNICODE

Юникод или Уникод (англ. *Unicode*) — это промышленный стандарт обеспечивающий цифровое представление символов всех письменностей мира, и специальных символов.

Коды в стандарте Unicode разделены на несколько областей.

Плоскость 0 (Основная многоязычная плоскость, англ. *Basic Multilingual Plane, BMP*) отведена для символов большинства современных письменностей и большого числа специальных символов.

Область с кодами от U+0000 до U+007F (от 0 до 127) содержит символы набора ASCII с соответствующими кодами - это сделано для совместимости.

Далее расположены области знаков различных письменностей, знаки пунктуации и технические символы.

Supplementary Multilingual Plan (SMP) - Дополнительная многоязычная плоскость отведена преимущественно для исторических письменностей, но включает также символы условных обозначений, такие как музыкальные и математические символы.

UCS-2 И UCS-4 - КОДИРОВКИ.

Universal Character Set

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.

Число показывает, сколько байт будет занимать один код поинт.

Недостатки UCS-2: не можем представить все код поинты.

Недостатки UCS-4: слишком большой размер. Даже если текст на английском языке, а там 90% ascii, то будет много излишних байтов. Конечно, можно сжать каким-нибудь алгоритмом, но это другая история.

Пример с UCS-2

ПРИМЕР С UCS-2

Дан текст в UCS-2, надо
переконвертировать в кириллицу.

```
004F0043005400410054004F004B002000380033002E00320  
03700200070002E0020003700200434043D04350439002004  
3C04430437044B043A0438002E0020041104350437043B043  
8043C04380442043D043E002004380020043104350441043F  
043B04300442043D043E003A0020002A0036003200360023
```

Псевдокод:

```
const s =  
'004F0043005400410054004F004B....';  
let result = '';  
for (let k = 0; k < s.length; k += 4) {  
    // берём очередные 4 символа  
    let hexcode = s.slice(k, k + 4);  
    // Преобразуем в число из 16-чного  
    // представления  
    let code = parseInt(hexcode, 16);  
    // Получаем букву по номеру в  
    // Unicode  
    result += String.fromCharCode(code);  
}
```

Это строка UCS-2 в шестнадцатеричной записи. В UCS-2 каждая «буква» (code point) кодируется двумя байтами, т.е. в этой строке каждые четыре символа это одна буква. Например 004F это латинская большая «О», а 0434 это русская буква «д».

Значит нужно просто разбить эту строку на четвёрки и преобразовать их в буквы. Например так как в примере выше

UTF-16

UTF-16 — один из способов кодирования **символов** (англ. *code point*) из Unicode в виде последовательности 16-битных **слов** (англ. *code unit*). Данная кодировка позволяет записывать символы Юникода в диапазонах U+0000..U+D7FF и U+E000..U+10FFFF (общим количеством 1 112 064), причем 2-байтные символы представляются как есть, а более длинные — с помощью суррогатных пар (англ. *surrogate pair*), для которых и вырезан диапазон.

Суррогатная пара — это две 16-битные кодовые единицы, используемые в UTF-16 (16-бит — два байта), которые представляют символ, превышающий максимальное значение, хранящееся в 16-битном формате. (т. е. 0xFFFF шестнадцатеричное или десятичное 65535).

Почему ? Поскольку весь набор Юникода имеет гораздо больше символов, чем 65535 (16 бит), поэтому для представления кодовой точки (символа) выше 0xFFFF (например, от 0x10000 до 0x10FFFF)

используются пары кодовых единиц, известные как суррогаты. И, к сожалению, использование UTF-16 в качестве набора символов потребует двух кодовых единиц для представления одного символа выше 0xFFFF.

Недостаток UTF-16:

-Усложнение декодирования, тк есть суррогатные значения, которые нужно собирать в пары и производить какие-то операции

UTF-32 — один из способов кодирования символов из Юникод, использующий для кодирования любого символа ровно 32 бита. Остальные кодировки, UTF-8 и UTF-16, используют для представления символов переменное число байт.

Главное преимущество UTF-32 перед кодировками переменной длины заключается в том, что символы Юникод непосредственно индексируемые. Получение n-ой кодовой позиции является операцией, занимающей одинаковое время.

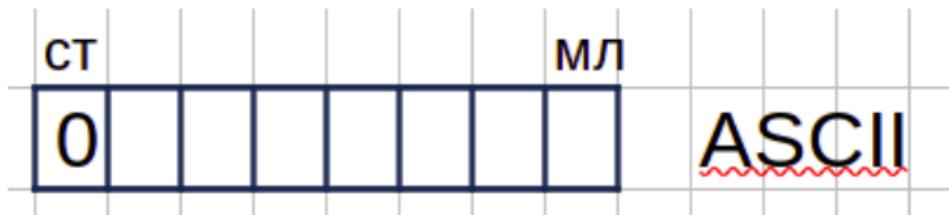
Главный недостаток UTF-32 — это неэффективное использование пространства, так как для хранения символа используется четыре байта.

Про UTF-8. Как работает?

Идея в том, что мы хотим представлять ASCII символы в том виде, в котором они есть. Хотим, чтобы текст, который состоит только из ASCII символов в кодировке UTF-8, выглядел также, как в кодировке ASCII. Поэтому для первых 128 код поинтов, которые совпадают с символами ASCII, и записываем их в том виде, в котором они есть.

Получается в ASCII старший бит равен чему? Нулю.

Если размер символа в кодировке UTF-8 = 1 байт, то код имеет вид (0aaa aaa), где «0» — просто ноль, остальные биты «a» — это код символа в кодировке ASCII;

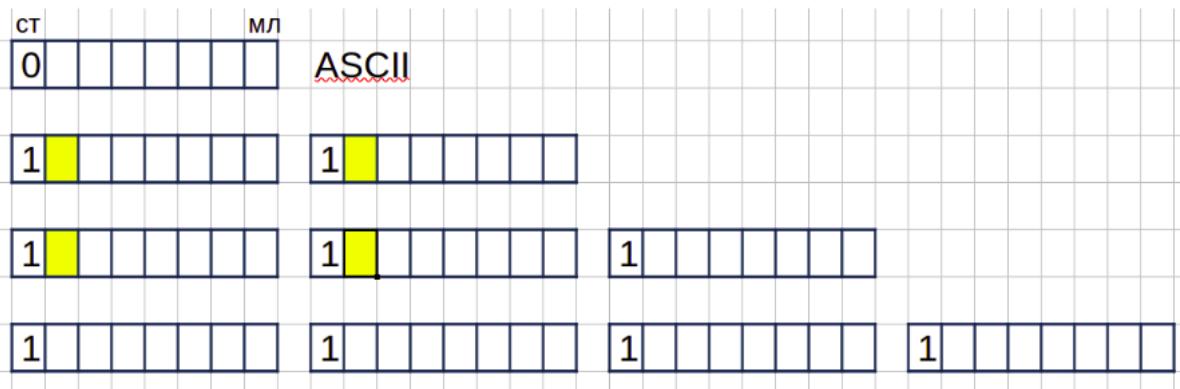


Во всех остальных байтах, старший символ равен 1, чтобы отличить от ASCII.

Следующий случай, когда идут 2 байта: там тож старший бит 1. Дальше хотим 3 байта. Как это сделать и что мы хотим?
Хотим, чтобы каждый вид байтов отличался от других.

UTF-8

Хотим, чтобы каждый вид байтов отличался от других.



То есть например по вот этому байту мы могли понять, что это первый байт представления код поинта, состоящего из 2x байтов. а другой- из 3х.

Также внутри каждой последовательности хотим понять, это первый байт или не первый. И все норм декодировать.

Как нам это сделать?

В тех, байтах, которые не первые, ставим 0. Теперь проблема в том, что эти не можем отличить друг от друга

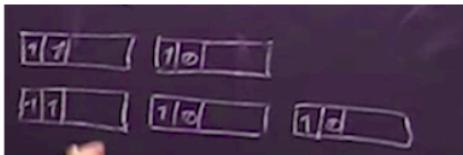
UTF-8

Расширяем так до 4х байтов (так написано в стандарте).

| СТ | МЛ |
|-----------|-------|
| 0 | ASCII |
| 1 1 0 | 1 0 |
| 1 1 1 0 | 1 0 |
| 1 1 1 1 0 | 1 0 1 |

Как нам это сделать?

В тех, байтах, которые не первые, ставим 0. Теперь проблема в том, что эти не можем отличить друг от друга



Как решить эту проблему? Там, где 2 байта, дописываем еще один нольник.



У этой кодировки нет байтордера, потому что построена на байтах, а не на словах.

1111

UTF-8 требует 8, 16, 24 или 32 бита (от одного до четырех байтов) для кодирования символа Unicode, UTF-16 требует 16 или 32 бита для кодирования символа, а UTF-32 всегда требует 32 бита для кодирования символа.

III

Про байт-ордер. Проблемы

- 1) Знаем, что перед нами UTF-16, но не знаем, какой там байт ордер. Нигде это не указано. Тогда мы можем применить какой-то эвристический алгоритм, чтобы выяснить, чему он равен.
- 2) UTF-16BE
- 3) В самом тексте можем указать, в каком он байт-ордере. Это делается с помощью кода поинта, который называется byte order mark (BOM). Такой код появляется в самой первой позиции текста.
В utf-8 нет BOM(!)

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 умолчанию 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](#)

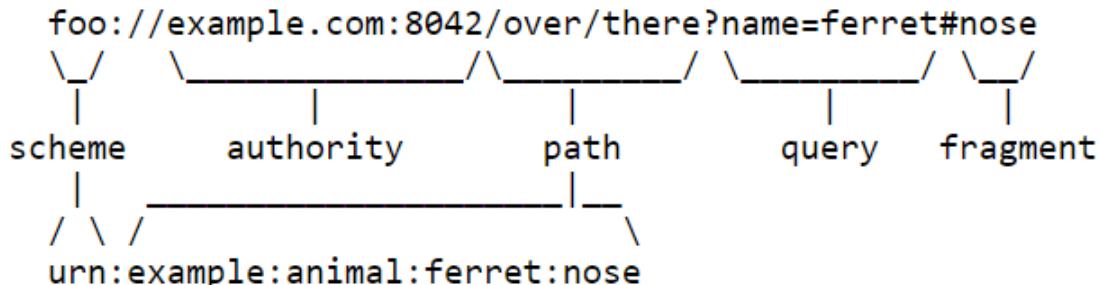
Uniform Resource Identifier - имя и адрес ресурса в сети, включает в себя URL и URN.

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.

- 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)

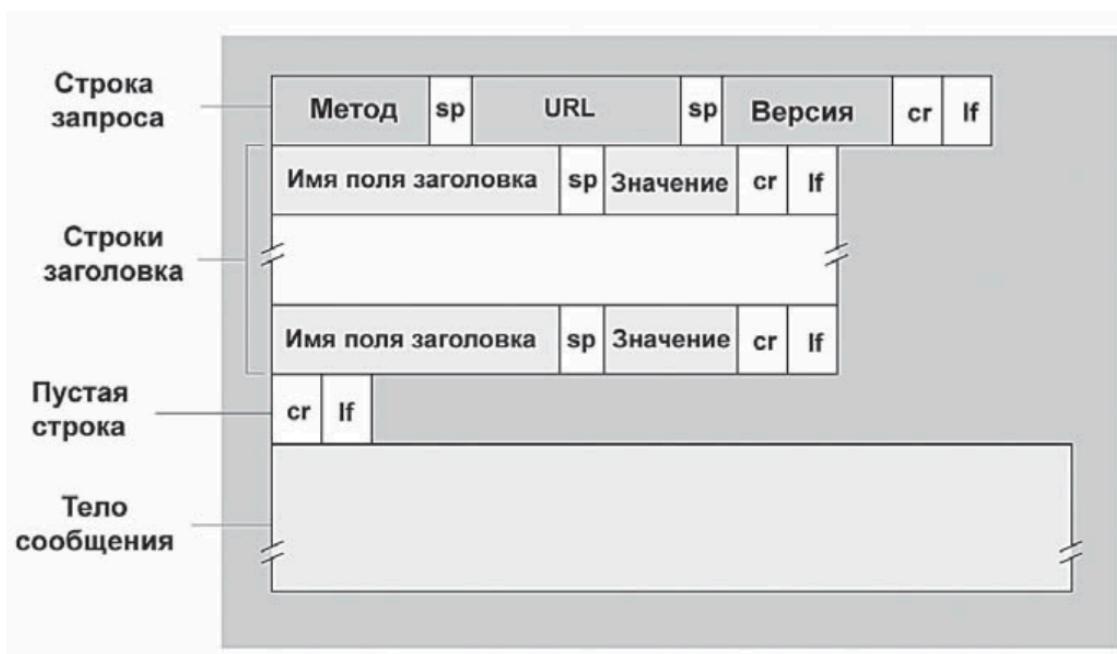


Рис. 2.8. Общий формат сообщения-запроса HTTP



Рис. 2.9. Общий формат сообщения-ответа протокола HTTP

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 нежележащий данных.
- **Transfer-Encoding : chunked.** Второй способ передачи информации о длине тела сообщения. Данный заголовок указывает на то, что данные будут передаваться кусками (chunkами), 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 возможные 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.

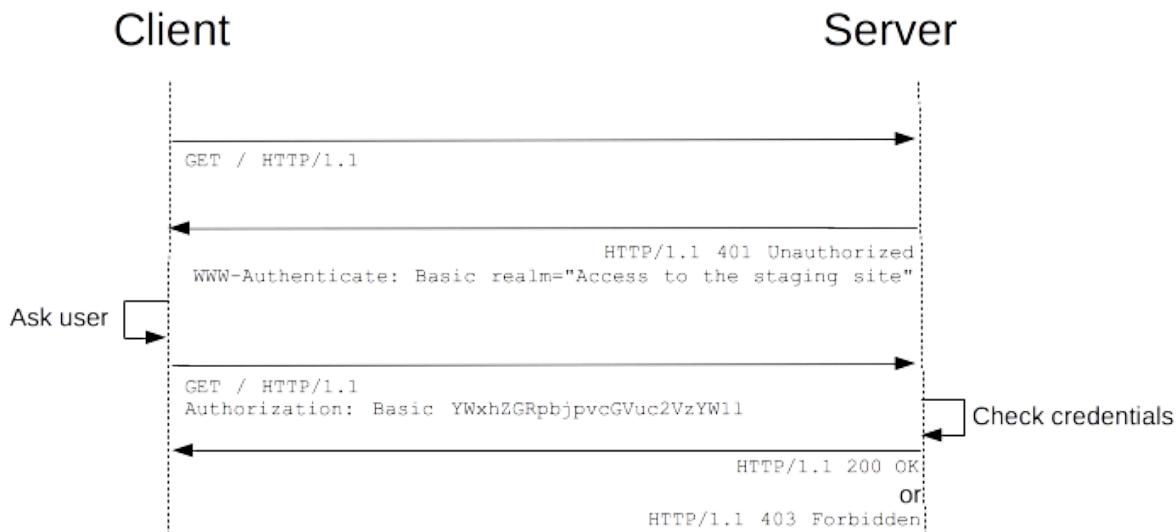
Допустим, что веб-сайту, который вы посещаете необходимо хранить какую-то информацию о вас (например, ваше имя или адрес вашего дома) или о вашей деятельности на этом сайте (к примеру хранить товары, которые вы добавили себе в корзину). Для этого и используются cookie. На сервере создается запись в базе 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 посыпает 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 единственным 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 посыпает 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 пересыпает 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.

взято из [10й лекции](#)

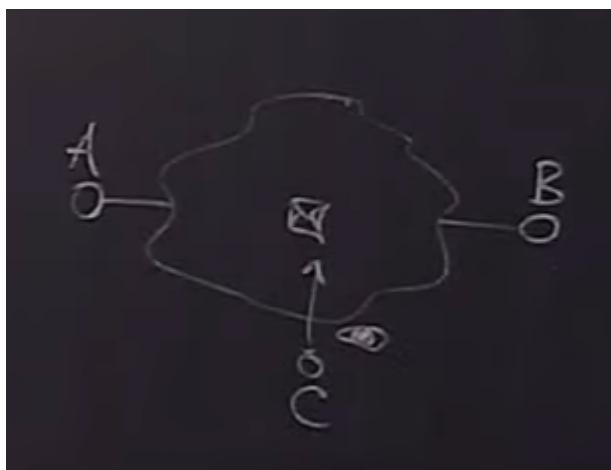
Тема про безопасность

Протокол TLS(SSL)

Transport Layer Security (Secure Sockets Layer)

Хотим обезопасить передачу данных в том случае, когда сеть не защищена.

Вот есть 2 чувака А&В, которые общаются через Интернет. Тогда может быть третья сторона, которая слушает/смотрит сообщения А&В.



Есть какое-то устройство в сети, которое удалось как-то так модифицировать, что оно стало способно копировать пересылаемые пакеты куда-то еще.

Или мы получили полный контроль над, например, маршрутизатором, и тогда мы можем настроить правила, удобные для нас.

Или отправляющая сторона А отправляет пакеты стороне В, точнее, думает, что отправляет пакеты стороне В. А на самом деле она общается с С. Происходит подмена ip адреса, например.

Как мы можем защитить данные?

- 1) Мы можем их зашифровать.

- 2) Аутентификация - проверка каким-то способами(может, например, криптографическими) идентичности того, с кем мы общаемся.

Начнем с шифрования.

- 1) С общим ключом (симметричное)

Симметричное шифрование

Есть функции

encrypt(startData, key) -> cryptoData

decrypt (cryptoData, key) -> startData

Есть стандарты: AES(читать [тут](#) и [тут](#)), Salsa и тд. Отличаются математической моделью, на которой они построены. Но по сигнатуре функций они схожи.

Обычно эти шифры являются биекцией между шифром и текстом. Имеют одинаковый размер. То есть например мы берем блок данных в 1Кб и получаем тоже 1Кб. Также такие тексты называются блочными, тк берут блок и преобразуют его в блок того же размера.

Как быть с ключом? А хочет шифровать, В хочет дешифровать. Нам нужен ключ на обеих сторонах диалога. Как можно сделать, чтобы у них обоих был один и тот же ключ? Кто-то генерирует этот ключ и отправляет ему по безопасному каналу получателю.

Но не всегда есть безопасный канал, по которому можем доставить этот ключ. Например, можем отправить по почте или позвонить по телефону и его сказать (клоунес).

Но для массового применения такое не очень удобно.

Для случая общения с непонятным сервером, нужен какой-то способ обмена ключами.

Например, это алгоритм DHE (Деффи-Хэлмана, читать [тут](#) и [тут](#)), который позволяет с помощью математических приемов по небезопасному каналу отправить ключ. И тогда добьемся того, что А и В будут иметь один и тот же ключ. И никто другой не получит доступ к нему. Это защищает только от просмотра. То есть если Зя сторона захочет стать "В", то алгоритм не заметит подмены.

Асимметричное шифрование

Как оно работает? Есть набор функций, который принимает наши данные, публичный ключ и приватный ключ.

encrypt(data, publicKey) -> crypto

decrypt(data, privateKey) -> data

Идея в том, что тот, чувак, который хочет, чтобы ему прислали какие-то зашифрованные данные, публикует свой ключ в открытом доступе. И любой желающий сможет зашифровать своё сообщение с помощью публичного ключа. Но расшифровать сможет только тот, кто имеет соответствующий этому публичному ключу приватный ключ, т.е. тот, кто его сгенерировал. Приватный ключ никому не сообщается.

Также эти алгоритмы могут использовать и “наоборот”: т.е. можно зашифровать с помощью приватного ключа, а зашифровать с помощью публичного. Но это не используется для засекречивания данных, тк если мы зашифровали с помощью приватного ключа, то любой сможет расшифровать с помощью публичного. Но этот способ можно использовать для проверки того, что сообщение было зашифровано именно этим ключом.

Примеры алгоритмов: RSA.

Особенность асимметричных алгоритмов в том, что они работают существенно медленнее симметричных, т.к. в них другая математика. Поэтому асимметричные не целесообразно использовать их для передачи больших данных.

Как можно организовать работу передачи больших данных?
Используем асимметричный алгоритм работы для передачи генерированного ключа, а потом используем симметричный алгоритм, который работает намного быстрее. Такой вариант работы предусмотрен в TLS (не только он, но такой тож есть).

В ассиметричном шифровании нет проблемы обмена ключами, тк отправить зашифрованные данные можно, зная публичный ключ, который знают все.

НО здесь другая проблема: мы должны проверить, что этот публичный ключ принадлежит именно той стороне, которой мы хотим что-то отправить.

Поэтому должен быть способ верификации публичного ключа. Есть 2 способа:

- 1) Web of Trust - сеть доверия. Идея в том, что мы проверяем корректность публичного ключа путем опрашивания тех сторон, кому мы уже доверяем. Если эти стороны (не обязательно все, а какое-то их количество) подтвердило, что чуваку можно доверять, то все ок.
В этом подходе нет централизованной авторитетной организации, которая говорила бы, что вот этим чувакам можно доверять, а этим нельзя.

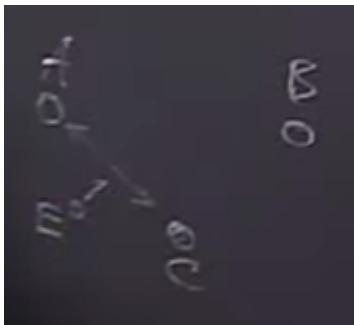
Пример: keybase.io

- 2) Когда есть доверенные организации, которые сами проверяют можно ли доверять какому-то участнику или нет. Крупные организации дорожат своей репутацией, поэтому обманывать кого-то им нет смысла. Допустим, есть веб сайт, и организация проверяет, что вот именно этот веб-сайт владеет таким-то ключом. Ну а мы, пользователи, раз доверяем организации, то мы тоже считаем, что именно этому сайту принадлежит именно этот публичный ключ.

Для работы этих подходов нужен механизм, который будет позволять проверять участников на их “хорошесть”, типа что они не хацкеры.

В случае Web Of Trust

Мы(A), хотим чет отправить B, есть третий участник C, который должен нам подтвердить, что B норм чел. То есть мы доверяем C, C доверяет B. И через C мы должны это доверие к B подтвердить. С не может просто так сказать нам (A), что вот, B - норм чел, потому что может возникнуть какой-то левый чувак-перехватчик, который может подменить обмен A с C. Т.е. подтверждение A-C должно быть уже безопасным.



В случае доверенной организации, С - это доверенная организация. Нам надо, чтобы С могло нам подтвердить свое доверие к В.

И такой механизм подтверждения есть. Он называется алгоритмом вычисления цифровой подписи.

Цифровая подпись - механизм подтверждения авторства.

Для этого есть функции

- создания подписи: `sign(data, privateKey) -> signature`
функция принимает какие-то данные, из которых считается криптографический хэш по какому-то алгоритму и шифруется с помощью приватного ключа, а на выходе получаем подпись - число, которое примерно (по размеру) совпадает с размером хеша.
- функция проверки подписи: `check(data, publicKey, signature) -> OK/NOT_OK`
здесь предполагается, что данные уже зашифрованы каким-то из алгоритмов, либо они не зашифрованы. В зависимости от ситуации. Передается публичный ключ. Хотим проверить, что данные были подписаны приватным ключом, который соответствует публичному. Также передаем подпись саму. Функция возвращает: корректная подпись или нет.
Как работает? Берутся данные, от которых вычисляется хеш, после этого цифровая подпись дешифруется с помощью функции `decrypt`, используя публичный ключ и сравнивается: если хеш совпадает, то все ок (значит подпись была создана тем, кто имеет публичный ключ), если нет - не ок.

Цифровой сертификат

Введем понятие цифрового сертификата.

** Еще раз: у нас есть А, В, С. А хочет обратиться к В безопасно, для этого А обращается к С, чтобы С подтвердил, что он знает В **

```
certificate(  publicKeyB,  
              signature = sign (cerificate, privateKeyC),  
              ...  
            )
```

Как мы можем проверить этот сертификат? Берем сертификат, зануляя в нём сигнатуру, берем публичный ключ С и сигнатуру из сертификата.

Если все ок, то это значит, что сертификат был подписан С.

`check(certificate, publicKeyC, sign) -> OK/NOT_OK`

Но в случае, когда А - это клиент, В - это сервер, нам будет недостаточно подтвердить сертификат, потому что может быть такая ситуация, что мы изначально не знаем, какой там должен быть публичный ключ.

Например, мы знаем, что у нас ключ (какой-то) подписан системой сертификации, который мы доверяем, но мы не знаем, что он от (например) Яндекса. Т.е нам

надо еще указывать доменное имя сервера, к которому мы обращаемся в сертификате. Если доменного имени сервера не будет, то будем просто подтверждать, что сертификат выдан кому-то, но неизвестно кому.

Поэтому добавляем сюда subject, в котором хранится строка, которая идентифицирует того участника обмена данными, с которыми мы общаемся.

```
certificate( publicKeyB,  
             signature = sign (cerificate, privateKeyC),  
             subject  
             ...  
           )
```

То есть к проверке сертификата, мы добавляем проверку того subject-а, который указан внутри. Т.е. например мы обращались к веб-сайту Yandex.ru, и тогда мы получаем от этого сервера сертификат. И помимо того, что мы проверяем его валидность, мы еще проверяем, что имя, к которому мы обращались, совпадает.

Чтобы не получилось так, что Вася Пупкин, который получил тоже сертификат от доверенного центра сертификации С, но при этом Вася != Яндекс, а мы думаем, что общаемся с Яндексом.

subject подменить тоже нельзя, тк это часть сертификата, который подписан центром сертификации.

Центр сертификации не могут выдавать сертификаты кому попало. Они должны проверять, что у того, кто запрашивает сертификат, имеет контроль над subject -ом - предметом сертификации.

Как центр сертификации может проверить, что доменное имя принадлежит именно конкретному пользователю, который запрашивает сертификат?

Хотим проверить <http://vasya.ru/file.txt>. Это можно сделать 3мя способами:

- 1) В этом файле есть какая-то секретная строчка, которую центр сертификации предоставил нам в личном кабинете. Центр смотрит, если он получил корректный ответ, то он считает, что мы имеем контроль над этим сайтом и доменным именем.
- 2) Используем систему DNS. Если у нас есть доменное имя, то у нас есть контроль над доменной зоной. Тогда центр может попросить сделать какую-то запись на TXT-зоне, в которой будет строчка “veify=...”. Если запрос от центра совпал с тем, что мы разместили, то все ок, мы имеем контроль надо доменом.
- 3) В системе WHO IS указан технический контакт, например, email, по которому можно обращаться по вопросам работы сайта. Тогда центр сертификации отправляет на этот емайл какое-то письмо, которое мы потом отправляем центру. Если содержание писем совпало, то значит все ок. Но такой способ, есесна, не оч.

В системе WWW используется протокол HTTP, если хотим безопасно отправлять данные, то используем протокол TLS.

Мы не хотим каждое приложение существенно переписывать под TLS, реализация у нас такая, что практически каждый протокол мы можем завернуть поверх TLS. И от этого программа не изменится от того, как будет работать. Потому что TLS фактически предоставляет тот же набор данных, что и TCP. Короче, если у нас есть протокол, который работает поверх TCP, то не сложно добавить TLS библиотеку и кой-какие функции заменить и все будет норм работать.

Для того, чтобы все работало норм, добавляется некая фаза, в которой происходит аутентификация собеседника и изначальная настройка этого соединения.

Поэтому сначала открывается TCP соединение, потом в нем происходит TLS-соединение.

Тк TLS работает поверх TCP, то у нас есть активная и пассивная стороны, которые работают по-разному.

| | | |
|---|--|---|
| A | | B |
|---|--|---|

| | | |
|--|--|--|
| | -----> TCP handshake | |
| | -----> TLS handshake | |
| | -----> | |
| | <p>А к В посыпает параметры. Активная сторона(А) говорит, какие параметры хочет: может выбирать алгоритмы, которые будут использоваться в соединении: алгоритмы хеширования, симметричного/асимметричного шифрования, алгоритмы обмена ключами, алгоритмы вычисления цифровой подписи.</p> | |
| | <----- | |
| | <p>В отвечает параметрами. Посыпает сертификат. Может запросить аутентификацию от клиента (А).</p> | |
| | <p>Когда А всё это получает, то валидирует сертификат, алгоритмы, которые были выбраны. Предоставляет клиентский сертификат.</p> | |
| | -----> | |
| | <p>Обе стороны договорились о том, какие алгоритмы они будут использовать. И тогда происходит обмен ключами для симметричного шифрования, чтобы дальнейшие данные были зашифрованы.</p> | |
| | <p>Обмен зашифрованными данными.</p> | |
| | <p>*кстати*</p> <p>Внутри соединения могут быть служебные сообщения о том, что произошла какая-то ошибка; сообщения о том, что пора менять ключ.</p> | |

** Если сервер и клиент поддерживают восстановление сессии, то они могут сохранить между собой секретный ключ, который позволит клиенту при следующем подключении сессии. То есть в первом пакете, в котором клиент передает параметры с секретным ключом, то сервер его видит и сразу возобновляет сессию. Обмен данными начинается сразу. Короче,

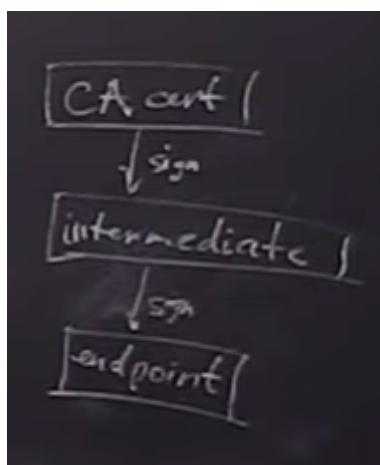
такой механизм был сделано для ускорения передачи данных. Но не все клиенты и серверы могут это поддерживать, поэтому иногда такой механизм не применяется. **

Есть вариант, когда используется pre shared key - ключ, который заранее известен серверу и клиенту. *редко используется такой способ, для веб-сайтов такое не используют.*

Цепочка сертификатов

В отдаёт А сертификат, который подписан центром сертификации и мы должны доверять этому центру. Для этого достаточно иметь его публичный ключ (где-то у себя хранить список ключей, которым мы доверяем). Этот список может быть в ОС, в браузере (user-agent), в Джаве (JRE), вручную зарегистрировать можно, добавляя куда можем/хотим.

| | |
|---|--|
| Корневые сертификаты root | должны быть подписаны своим же собственным приватным ключом. |
| Промежуточные сертификаты (intermediate) | подписаны корневым. |
| endpoint (какой-то сайт, например) | подписан промежуточным |



Зачем так сделано? Почему не напрямую?

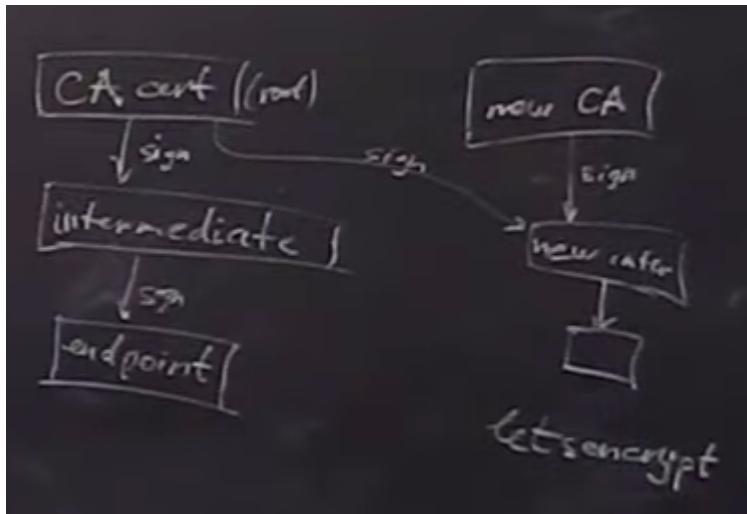
- 1) Корневой сертификат нужно очень безопасно хранить. Потому что если он будет утерян, то будет проблема и в ОС, и в User-Agent-е, и в JRE. Типа этот корневой сертификат зашит везде. И если этот ключ кто-то украдет, то вор сможет получить большие полномочия:

сможет подписывать любые сертификаты, на любые имена. И мы им будет доверять. Для защиты от этого есть механизм отзыва сертификатов.

Поэтому корневой сертификат хранится на каком-то носителе, который не подключен никуда. То есть это может быть диск или чет такое, физическое. Короче, взломав какой-то сервер, такой корневой сертификат получить нельзя.

А вот intermediate и endpoint сертификаты хранятся на серверах, защищенных, конечно, но все-таки серверах. И когда центру сертификации отправляется запрос на подтверждение авторства, то используется intermediate сертификат. В случае, если intermediate будет утерян, то центр сертификации выпишет новый сертификат. И ему все будут доверять, тк корневой сертификат не потерялся!

- 2) Имеет возможность подписать один и тот же промежуточный сертификат несколькими корневыми. Это сделано для того случая, когда появляется новый



центр сертификации,

которому пока не все еще доверяют: есть старые ОС, версии Джавы, которые не знают корневой сертификат (новый). Но есть старый доверенный.

Про отзыв сертификата

Что, если мы потеряли контроль над своим сертификатом? Т.е. потеряли приватный ключ, который соответствует нашему сертификату. Тогда есть проблема в том, что тот, кто его получил, может подделывать наш сервер (ресурс).

Поэтому нужно узнать, что произошла компрометация этого ключа и клиенты не должны теперь доверять этому сертификату.

Надо проверить, не отозвал ли сертификат центр сертификации, который его выдал?

Для этого есть специальные механизмы. Каждый центр должен выдавать API, в которой можно отправить сертификат и центр должен ответить, был ли сертификат отзван или нет.

Получается, что клиент, прежде чем доверять сертификату, должен проверить, не был ли он отзван.

Такая проверка делается не всегда. Результат проверки кэшируется, чтобы не было необходимости проверки заново при каждом запросе.

Отзыв сертификата стоит существенных денег. Типа его отзыва стоит дороже его получения (лол).

https://ru.wikipedia.org/wiki/Server_Name_Indication

БАЗА

Email-аутентификация — процедура проверки подлинности отправителя.

Авторизация — предоставление определенному лицу прав на выполнение определенных действий.

Чем отличается WWW от Интернет?

WWW - один из сервисов, которые есть в Интернет. Занимает большой процент использования. Работает на уровне приложений, т.е. это более высокоуровневая вещь. Это сеть документов. Размещается на ресурсах сети Интернет, но логически может представлять собой структуру, никак не связанную с её физическим размещением.

Интернет - некоторая сеть сетей, которая объединяет компьютеры между собой. Работает на сетевом уровне: все протоколы, которые в ней работают являются протоколами сетевого уровня.

Дополнительная инфа

Взято из [1й лекции](#).

Про физический уровень

Отвечает за доставку данных благодаря физическим процессам.

За счет чего происходит передача данных? При помощи электромагнитных волн.

3 способа передачи данных:

- Проводник
- Световые сигналы (оптоволокно)
- Радиоволны

По каким критериям можно сравнивать эти 3 способа?

- 1) Пропускная способность - количество передаваемых бит в секунду.
- 2) Скорость распространения сигнала.

Чем пропускная способность отличается от скорости распространения сигнала? Читать [тут](#).

“В отличие от скорости передачи информации, которая показывает как быстро передается информация от источника к получателю, пропускная способность показывает как много этой информации можно передать по конкретному каналу”.

Пропускная способность зависит от того, с какой частоты мы используем несущий сигнал.

Скорость света в стекле меньше, чем в воздухе.

Оптоволокно - кварцевая нить, которая имеет эффект полного внутреннего отражения света, поэтому свет наружу не выходит. Также она изолирована светонепроницаемыми оболочками. Поэтому вероятность в нее проникновения какого-то света (извне) очень маленькая.

Помехи самые большие возникают тогда, когда частота сигнала помех совпадает с частотой несущего сигнала.

| | | | |
|------------------------|------------------------|---------------------------|------------|
| | проводник | свет(оптоволокно) | радиоволны |
| Пропускная способность | + средняя на ПС влияют | самая большая, тк высокая | + средняя |

| | | | |
|--|---|--------------------|--|
| | еще и помехи, которые появляются в канале | частота | |
| Скорость распространения | + - средняя | - , небольшая | ++ выигрывают |
| Защищенность от помех (помехоустойчивость) | + - средняя | самая защищенная | - , не оч защищены, тк распространяются в открытой среде |
| Цена | средняя цена | самая высокая цена | средняя цена, материальные линии связи не нужны |

Модель OSI/ISO

Иерархическое представление уровней позволяет каждому уровню выполнять какую-то свою конкретную задачу. И эти уровни могут между собой взаимодействовать.

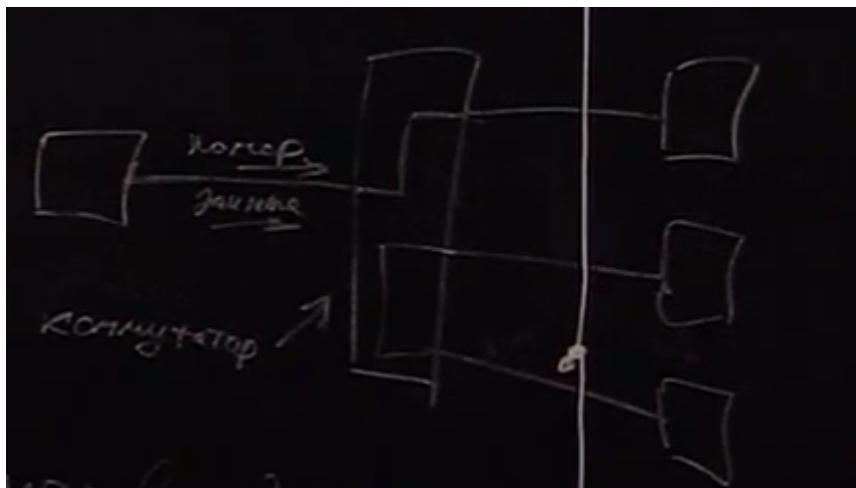
Преимущество многоуровневого подхода позволяет придумывать различные способы работы/реализации этого уровня. Если бы не было иерархии, то не было бы между уровнями никакой стандартизации взаимодействия. На каждом уровне есть конкретные реализации, которые называются протоколами.

Протокол - набор правил, которые описывают взаимодействие >2x сторон. Это могут быть описание последовательных действий, форматы сообщений и тд.

1. Физический уровень - позволяет передавать цифровые данные при помощи физических процессов.
2. Канальный уровень (LINK) - его задача передать фрагменты данных в пределах многопользовательской среды. Здесь появляются заголовки, в которых прописан адрес получателя. Доставка осуществляется с помощью коммутирующего устройства.

| Канальный подход | Пакетный подход |
|--|---|
| Данные можно передавать как непрерывный поток. | Данные разделяют на фрагменты. Каждый фрагмент передаем кому-то конкретному получателю. |
| Например, телефонная связь. | Например, почта. Подписываем письмо, кладем в ящик. И потом это письмо доходит до конкретного получателя. |

Коммутация каналов



Здесь происходит коммутация цифровых или аналоговых данных.

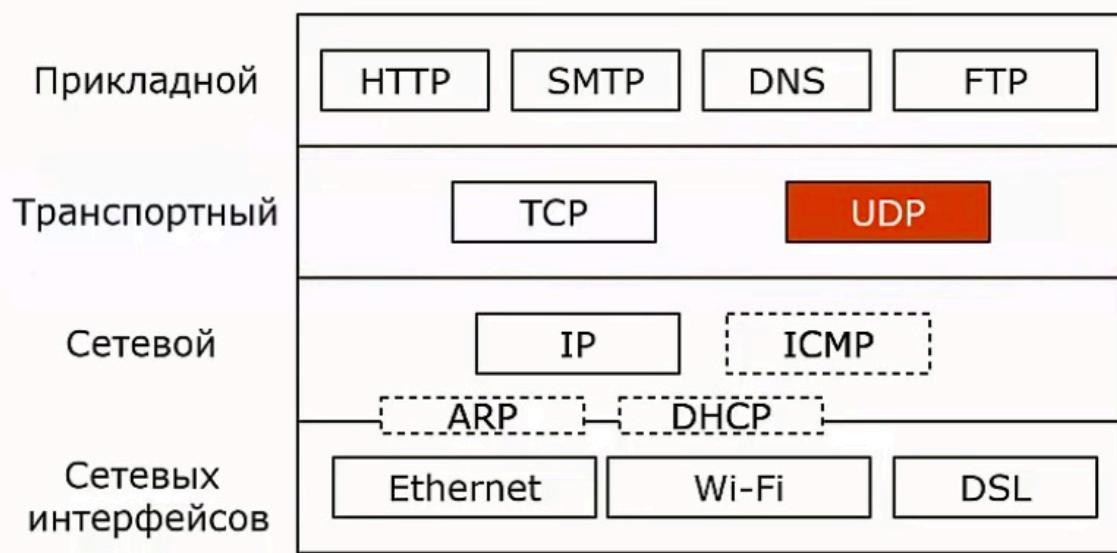
Преимущество: нам выделен конкретный канал, который зарезервирован нами, можем им пользоваться когда захотим.

Недостаток: канал связи простояивает, когда абоненты не общаются.

Коммутация пакетов

Здесь происходит коммутация цифровых данных. Данные разделяем на фрагменты. В каждый такой фрагмент добавляем заголовок, в который прописываем адрес получателя. И каждый фрагмент отдельно коммутируется - доставляется до коммутатора, который смотрит в заголовок и отправляет по нужному адресу. Таким образом, можем общаться с несколькими людьми одновременно, что в случае канальной коммутации невозможно.

Примеры протоколов на разных уровнях



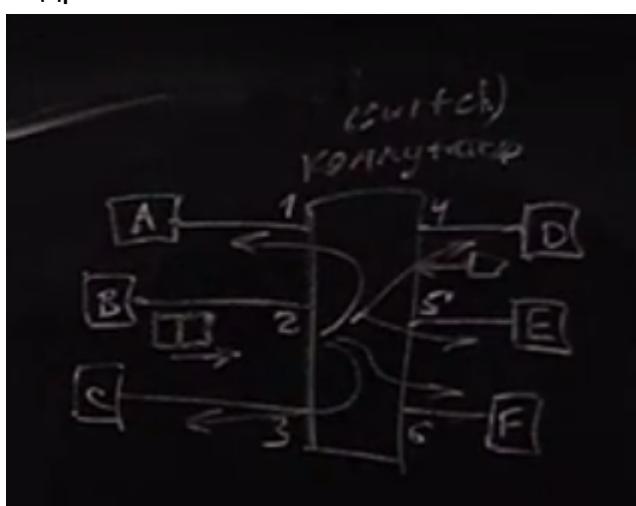
Про канальный уровень

Отвечает за доставку пакетов с двоичными данными в среде с множественным доступом (ну вообще может быть и соединения точка-точка).

Ethernet - набор стандартов, который описывает взаимодействие физического и канального уровней. В качестве адресов используются 48-битные числа - MAC-адреса (Media Access Control). У каждого устройства свой уникальный MAC-адрес.

Коммутатор = switch (свитч).

Кадры = frames.



Устройства обозначены буквами. У каждого устройства есть mac-адреса (одноименные с буквами). Цифрами обозначим порты.

Сначала отправляем данные на коммутатор, который берет из заголовка нужную информацию и передает получателю.

А какая инфа содержится в заголовке, чтобы можно было коммуницировать между устройствами?

Ну, по-первых, это адрес получателя (mac-address). Еще адрес отправителя. Может быть указана контрольная сумма, размер кадра, флаги какие-то и тд.

Пусть мы отправляем пакет В в D на коммутатор.

| dst (mac) | src (mac) |
|-----------|-----------|
| D | B |

Как коммутатор узнает, что получатель D расположен на порту 4?

Таблица, в которой ставится соответствие мак-адреса и порта, называется мак-таблицей.

А как коммутатору составить эту таблицу? Если не знает кому отправлять пакет, то отправляет его всем устройствам (во все порты). Раз у нас есть устройство с мак-адресом D, то оно сможет этот пакет обработать. А другие устройства должны этот пакет послать нафиг, потому что он им не предназначается.

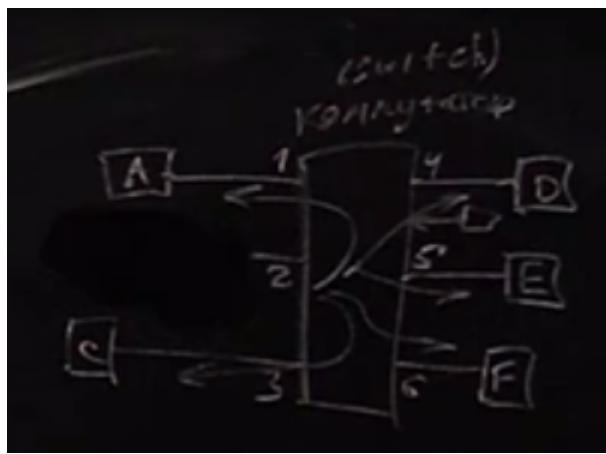
[Эта процедура может занять некоторое время, поэтому она реализована аппаратно!]

Окей, D получил этот пакет. Теперь на коммутатор он отправляет такой заголовок:

| dst (mac) | src (mac) |
|-----------|-----------|
| B | D |

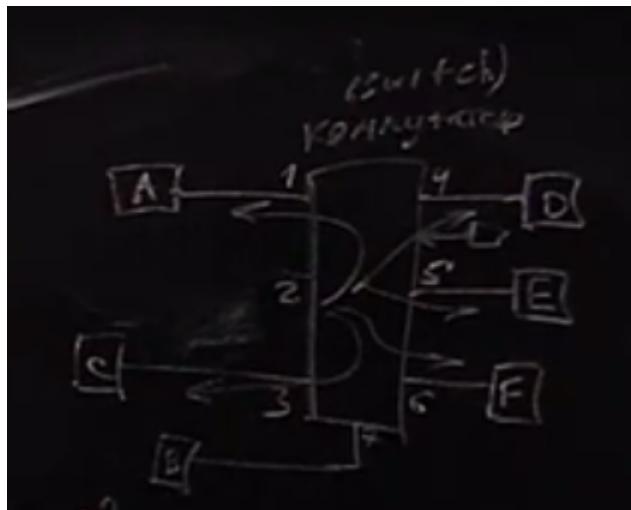
Ну и пока такое взаимодействие происходит, то мак-таблица заполняется дальше кста.

А что если B вышел из сети? D до сих пор отправляет пакеты до B. Что будет в таком случае?



Коммутатор обнаружит, что физическое устройство вышло из сети и либо уберет строку из таблицы, либо нет.

И получается D отправляет пакеты всем. Тк может оказаться так, что по каким-то причинам устройство D подключили к другому (например, 7)



порту.

В случае, если на 2й порт подключили какое-то новое устройство G, то его тож записываем в табличку.

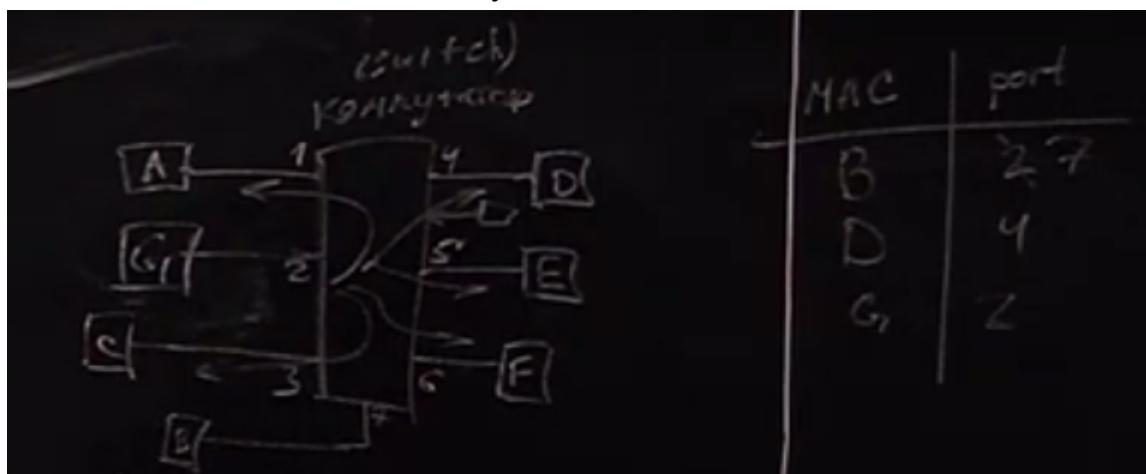


Таблица не бесконечная. Коммутаторы предусматривают механизмы её очистки. Если место в таблице закончилось, то можем убрать самое старое добавленное устройство. Или каким-то другим способом очищаем табличку (зависит от реализации короче). Например, если время последнего действия устройства слишком большое, тогда удаляем.

Взято из [лекции 1.2](#).

По сути материал этой лекции - из 4 семака, так что тут кратенько чёт напишу.

VLAN - virtual local area network.

Про vlan можно почитать [тут](#).

Стандарт для сетей IEEE 802.X - почитать можно [тут](#).

Коммутаторы должны поддерживать этот стандарт.

А что делает этот стандарт? Берет заголовок, который был в Ethernet, и добавляет туда еще и номер влана. Это называется тегированием.

Пакеты с тегами - тегированные пакеты.

Нетегированные порты называются access портами, тегированные - через которые могут передаваться тегированные пакеты называются trunk-ами.

Коммутатор не должен принимать тегированные пакеты из access-портов. Почему? Потому что если будет это делать, то нехорошее устройство может начать посыпать тегированные пакеты и оно сможет отправлять эти пакеты в любой из вланов, который захочет. А это противоречит идее того, что мы эти вланы хотим изолировать.

Зачем в рамках влана возможность отправить пакет всем в сети?

Например, мы реализуем протокол, где мы хотим опросить всех пользователей нашей сети (влане). И тогда кто-то конкретный отзовется или все отзовутся.

broadcast address - широковещательная рассылка.

Бывают ситуации, когда пакет был отправлен, но до получателя не дошел.

Причины:

- проблемы с коммутацией (устройство подключили на другой порт, но запись в таблице не успела обновиться)
- помехи в среде, которые нарушают передачу

- коммутатор - устройство с внутренней памятью и для того, чтобы обработать пакеты, надо их прочитать в память, проанализировать заголовок и понять, в какой порт дальше отправлять. То есть пакет храниться в памяти какое-то время. Получается входящих пакетов так много, что коммутатору не хватает памяти, чтобы их одновременно все в память прочитать.

Про сетевой уровень

Посредством физического и канального уровней нельзя построить какую-то большую сеть, тк появляются проблемы с масштабированием, потому что система (доставки пакетов, получатель которых не пойми где находится) не структурирована. А мы хотим структурированную, устроенную таким образом, чтобы можно было понять, чтобы были какие-то осмысленные правила, что если адрес вот такой-то, то отправлять туда-то. За это отвечает сетевой уровень.

Теперь мы можем доставлять пакеты сегментов на канальном уровне, а сегменты - на сетевом.

Про транспортный уровень

На транспортном уровне можем работать с потоками данных.
Смотреть вопрос про UDP/TCP.

Про сессионный уровень

Его задача - структурировать данные сессии(сеанса).
TCP обладает свойствами не только транспортного, но и сессионного уровня.

Про уровень представления

Его задача - представить данные в каком-то другом виде, изменить, зашифровать, закодировать таким образом, чтобы данные обладали каким-то другими свойствами.

Про уровень приложений

Решает конкретную бизнес-задачу для какого-то конкретного приложения.

Сводная таблица уровней

| | Доставка происходит | Реализации протокола | |
|--------------------|--|----------------------|------------|
| физический уровень | | | TCP/ IP |
| канальный | внутри сегмента сети (влан) | Ethernet | |
| сетевой | между устройствами в разных сегментах | IPv4, IPv6 | |
| транспортный | между приложениями на этих устройствах | TCP, UDP | |
| сессионный | | | |
| представления | | TLS | |
| приложений | | DNS, HTTP | |

Про IPv4

Адреса устроены следующим образом:
32 битное число.

Заголовок такой:

IPv4 Header Format

| Отступ | Октет | 0 | | | | | | | | 1 | | | | | | | | 2 | | | | | | | | 3 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|------------------------|-----------------|-----------------------------------|------------------|------------------------------------|----------------------------------|------------------------|---|---|---|-------|---|---|---|-----------------------------|---|---|---|--------------------|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
| | | Бит | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | | | | | | | | | | | | | | | | | | | | | |
| 0 | 0 | Версия | Размер заголовка | Differentiated Services Code Point | Explicit Congestion Notification | Размер пакета (полный) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 4 | 32 | Идентификатор | | | | | | | | Флаги | | | | | | | | Смещение фрагмента | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 8 | 64 | Время жизни | | | | Протокол | | | | | | | | Контрольная сумма заголовка | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 12 | 96 | IP-адрес источника | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 16 | 128 | IP-адрес назначения | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 20 | 160 | Опции (если размер заголовка > 5) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 20 или 24+ или 192+ | 160 или 192+ | Данные | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

(условно)



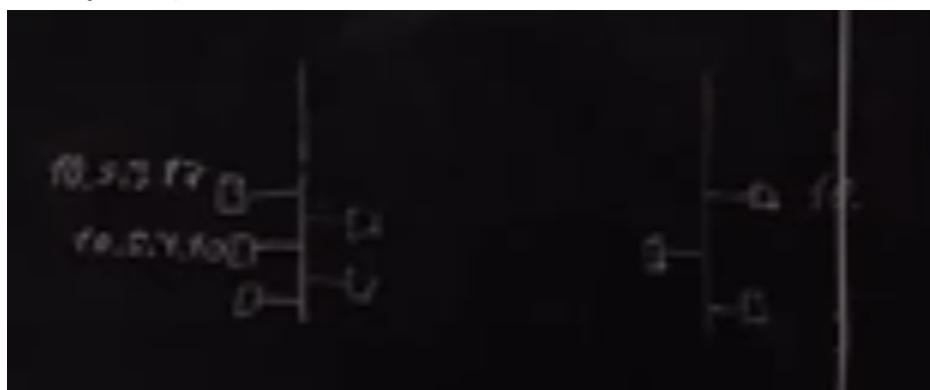
адрес сети адрес хоста

Мы должны каждому хосту в сегменте выдать адрес сетевого уровня таким образом, чтобы часть, которая соответствует адресу сети у них была одинаковая, а часть, которая соответствует адресу хоста - разная. И тогда, имея адрес устройства, мы сможем, посмотреть на адрес сети, чтобы понять, в какой сети оно[устройство] находится.

Маска сети.

Адресное пространство сети - это набор адресов, которые могут иметь хосты в конкретной сети.

Пример, который мне лень подробно расписывать. Про разные коммутаторы в сетях.



Идея задания адресов в том, чтобы адресные пространства разных сетей не пересекались между собой. Иначе для тех адресов, которые оказались в пересечении, мы не будем знать, в каком сегменте находится получатель. То есть должны всегда точно знать, в каком именно сегменте оказался наш получатель.

Когда была доставка на канальном уровне, мы предполагали, что уже знаем мак-адрес получателя.

Протокол ARP преобразует адреса сетевого уровня в адреса канального уровня. Все устройства, которые поддерживают протокол IPv4 должны поддерживать реализацию протокола ARP, иначе как будем узнавать мак-адреса...

ARP - это протокол сетевого уровня.

Работает ARP так: если мы хотим узнать мак-адрес у устройства с известным сетевым IPv4 адресом, мы отправляем запрос всем участникам сети с вопросом “у кого из вас такой ip адрес?” И устройство с таким адресом отвечает. Если устройства с таким адресом в сети нет, то будет тишина, и по какому-то тайм-ауту поймем, что такого устройства в сети нет.

Как отправить запрос всем участникам? С помощью широковещательной рассылки.

Любое устройство сетевого уровня, прежде чем использовать протокол ARP, понять, находится ли ip адрес получателя в том же самом фрагменте сети, что и оно само.

Как это понять? По маске сети.

Если в сети нет ни одного маршрутизатора, то пакеты из разных сегментов никак нельзя будет передать.

Каждому хосту помимо маски и ip адреса, нужно задать адрес маршрутизатора, которым хост сможет воспользоваться, если захочет отправить пакет за пределы своей сети.

Маршрутизатор, который используется хостами для связи с другой сетью, называется default gateway. Часто адрес маршрутизатора: 10.4.0.1. (Но это необязательно)

Forwarding - процесс пересылки из одного маршрутизатора в другой.

Взято из [лекции 2](#)

Ну это опять повторение инфы с 4го семинара + прошлой лекции.

Нулевой адрес зарезервирован под адрес сети, а ff...ff - для бродкаста.
Эти адреса нельзя использовать для адресов хостов.

| Таблица маршрутизации | | |
|-----------------------|-------------------|---------------------------------|
| адрес сети (Net) | маска сети (Mask) | адрес маршрутизатора (Next Hop) |

RIP - Routing Internet Protocol - указывает за сколько переходов хост может добраться из конкретной сети в какую-то другую.

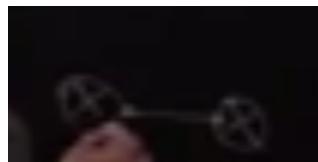
Взято из [лекции 3](#)

Про протоколы маршрутизации

Дистанционно -векторные протоколы (distance-vector): RIP, EIGRP.
Эти протоколы устроены таким образом, что маршрутизаторы сети знают только расстояние до неё и следующий шаг, т.е. они не знают весь маршрут, знают только свою окрестность.

Link state: OSPF.

В таком протоколе сохраняется состояние канала связи. У канала есть метрика, которая задает вес ребра в графе. Мы храним метрику и точки, к



которым она подключена.

И если мы знаем все такие ребра, то сможем восстановить весь график. А потом применяется алгоритм поиска кратчайшего пути в графике. В OSPF используется алгоритм Дейкстры.

В чём плюс? Работает быстрее. Передаем состояния канала связи. Это открытый протокол.

Минус: каждый маршрутизатор должен знать больше информацию, знать не только о своих соседях инфу, но вообще обо всех каналах, которые

где-то есть. То есть для этого нужно больше памяти + вычисления делать. Если что-то где-то поменялось (добавилась / удалилась линия), то каждый маршрут надо заново пересчитать, тк какой-то маршрут мог стать лучше, а какой-то хуже. При любом изменении надо все пересчитывать, что занимает процессор.

Если граф очень большой, то каждый раз пересчитывать по алгоритму Дейкстры очень жирно, и тогда на помощь приходит механизм зонирования - граф разбивается на отдельные зоны, в каждой из них отдельно работает алгоритм. А потом уже зоны каким-то образом потом все вместе связываются.

Протоколы RIP, EIGRP, OSPF работают внутри административной единицы - автономной системы.

| IGP | EGP |
|---|--|
| Interior Gateway Protocol | Exterior Gateway Protocol |
| Внутри шлюза используется: для связи внутри автономной системы. | Используется для внешней маршрутизации: для связи автономных систем между собой. |
| RIP, EIGRP, OSPF | BGP (Border Gateway Protocol) |

Взято из [лекции 4](#)

Про настройку хостов на сетевом уровне.

Работаем с протоколом IP, т.е. рассматриваем либо IPv4 и/или IPv6.

Надо адрес, маску, default gateway.

Настройка вручную плохо масштабируется. Поэтому придумали протоколы автоматической конфигурации.

Самый распространенный DHCP: Dynamic Host Configuration Protocol - протокол динамической настройки хостов.

DHCP

По умолчанию DHCP включен на каждом компе в настройках.

| host(client) | | DHCP(server) |
|--|----------------------------------|--|
| | -----> discover (broadcast) | |
| | <----- offer | <p>если есть сервер в сети, то отправляет оффер, в котором будет писать свободные адреса (точнее их диапазон) в текущем адресном пространстве.</p> <p>Как сервер определяет, что клиент новый? По mac-адресу клиента. То есть сервер ведет таблицу с мак-адресами (DHCP clients table)</p> |
| клиент может оффер принять или не принять. Если принимает, то отправляет следующий запрос, в котором просит сервера выдать ему те настройки, которые сервер ему предложил. | -----> request (broadcast) | |
| | <----- confirm | сервер подтверждает, что эти настройки клиент может использовать |

Зачем нам в протоколе такое двухэтапное подтверждение надо?

- Например, если discover/offer потерялся, то клиент через какой-то момент времени видит, что никто не ответил на offer, и отправляет discover еще раз. И так делает несколько раз, пока ему кто-нибудь не

ответит. Если никто не отвечает в течение какого-то времени после нескольких попыток, то клиент считает, что DHCP сервера в сети нет. Или он есть, но сервер не хочет высыпать никакие offer-ы, тогда клиент не может воспользоваться услугами DHCP-сервера.

- Может оказаться ситуация, что у нас оказалось несколько DHCP-серверов и они оба прислали офферы, тогда клиент может выбрать из них оффер, который ему больше нравится. Чаще будет брать первый, который ему пришел.

DHCP lease time - время аренды настроек - клиент должен отправить запрос с тем, чтобы сервер подтвердил, что эти настройки еще актуальны. Так сервер продлит время аренды хоста, тк увидит, что тот еще жив-здоров.

| DHCP clients table | | |
|--------------------|------------|---|
| mac-address | IP address | lease time (время, когда последний раз продлевалась аренда) |
| | | |

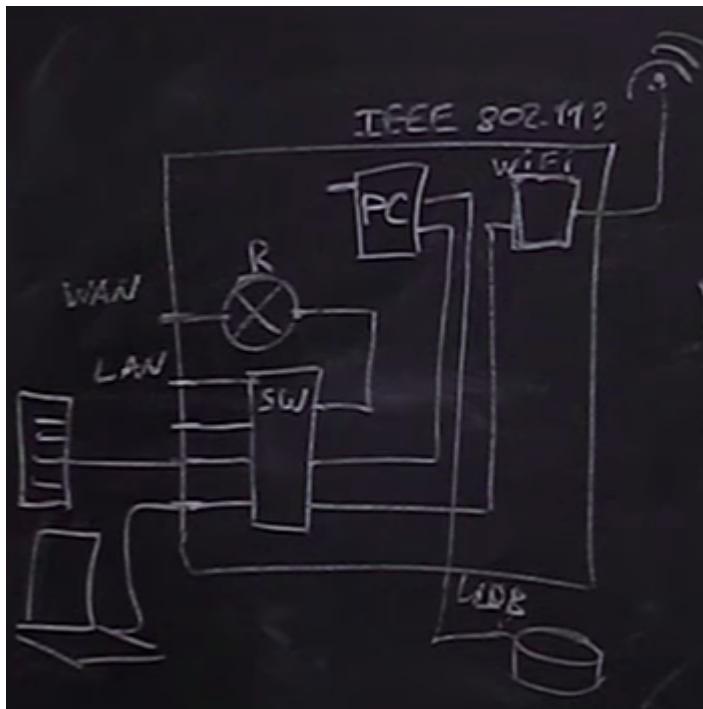
Если хост отключился, то время аренды истекает, и его из таблицы можно удалить. Но в большинстве каких-нить небольших сетей (домашних/небольших организаций) сервер не удаляет запись хоста из таблицы. А добавляется еще столбец со статусом: действующий / недействующий.

| DHCP clients table | | | |
|--------------------|------------|---|--------|
| mac-address | IP address | lease time (время, когда последний раз продлевалась аренда) | status |
| | | | |

Для чего это делается? Почему не удаляется запись? Ведь можно было бы удалить, тк таблица не бесконечная, занимает память. Тк в бытовых сетях все устройства те же, так что каждый раз изменять их адреса и тд не имеет особого смысла.

Как выглядит домашний маршрутизатор?

WLAN; CAN



IPv6

Адресное пространство 2^{128} .

Для адресных пространств конкретной сети, в которой есть устройства, используется адресное пространство /64. Половина битов под адрес сети, половина - под адрес хоста. Всегда.

DHCPv6.

В IPv6 встроена автоматическая конфигурация хостов. В IPv6 есть NDP (Network Discovery Protocol), который работает поверх ICMPv6. Как работает? Отправляется запрос на группу мультикаста, которая называется over routers, к которой присоединяются все маршрутизаторы.

Link Local Address - на одном канальном уровне.

Local Address - адреса, аналогичные тем, что в IPv4 заразервированы под локальные сети по типу 10.0.0.0/1.

Разделение адресов - для того, чтобы быстрее понять, к какой стране/региону и тд принадлежит адрес
global, multicast, region, provider

Вот [тут](#) можно потыкаться.

Мульти cast в IPv6 “клёвый”. А почему? Бродкаста нет, кста.

Для работы IPv6 нужен мульти cast, с помощью которого работает NDP.

Для IPv4 мульти cast включен не везде, в IPv4 сложно получить какой-то адрес.

Короче в IPv6 всегда включен мульти cast, иначе как устройство будет работать.

взято из [6й лекции](#)

socket:

- ресурс ОС
- IP:port

Сокеты Беркли, читать [тут](#).

взято из [8й лекции](#)

HTTP/1.1

Полезные ссылки

1. Сайт курса с какого-то уника:
<https://math.gsu.by/wp-content/uploads/courses/networks/index.html>
2. Сайт какого-то курса: <https://lectures.net.ru/>
3. Статья с хабра про DNS: <https://habr.com/ru/articles/303446/>
4. Про доменные зоны на вики:
https://ru.wikipedia.org/wiki/%D0%94%D0%BE%D0%BC%D0%B5%D0%BD%D0%BD%D0%BE%D0%B5_%D0%B8%D0%BC%D1%8F
5. Про TCP [статья](#)
6. Про SMTP [статья](#)
7. Про POP3 & IMAP [статья](#)
8. Про сериализацию и десериализацию данных [статья](#)
9. JSON vs XML [сравнение](#)
10. Про Yaml [статья](#)