

Компьютерные сети

Прикладные протоколы

Соотношение прикладных протоколов TCP/IP и верхних уровней модели OSI/ISO. Задачи сеансового уровня, уровня представления. SMTP. HTTP. Представление о RPC и API

[Введение](#)

[Обзор верхних уровней модели OSI](#)

[Прикладной уровень \(Application layer\)](#)

[Уровень представления \(Presentation Layer\)](#)

[ASCII и Telnet](#)

[Проблема перекодирования кодировок](#)

[Порядок байтов \(byte order\)](#)

[Сжатие данных](#)

[Стек H.323](#)

[MIDI](#)

[XDR](#)

[SSL \(Secure Socket Layer\)](#)

[TLS \(Transport Layer Security\)](#)

[Сеансовый уровень \(Session Layer\)](#)

[TLS на примере HTTPS. сертификаты](#)

[QUIC как альтернатива HTTP+TLS+TCP](#)

[Примеры протоколов прикладного уровня](#)

[Сервисы электронной почты: SMTP/POP3/IMAP](#)

[Процесс доставки электронной почты](#)

[Протокол SMTP](#)

[Основные понятия HTTP](#)

[Методы HTTP](#)

[Коды состояния](#)

[Заголовки HTTP](#)

[Заголовки в HTML](#)

[User Agent](#)

[HTTP-прокси](#)

[SPDY и HTTP/2](#)

[Примеры API](#)

[SOAP \(Simple Object Access Protocol — простой протокол доступа к объектам\)](#)

[XML-RPC](#)

[JSON-RPC](#)

[REST API](#)

[WebDAV](#)

[Практическое задание](#)

[Литература и ресурсы для дальнейшего изучения](#)

[Дополнительная литература](#)

[Используемая литература](#)

Введение

Модель TCP/IP выиграла в конкурентной борьбе у модели OSI/ISO. Но задачи, решаемые на тех уровнях модели OSI, которые не имеют соответствия в модели TCP, остались. Например, задача создания соединения, управления соединением, завершения и возобновления соединения решается и в стеке TCP/IP, но частично (создание, управление и завершение) — транспортным протоколом TCP, частично (возобновление соединения, механизмы сеансов, такие как куки и сессии, механизм докачки) — прикладным протоколом HTTP. Другой пример — аутентификация и шифрование, которые изначально не были заложены в прикладные механизмы. Интернет 90-х годов, без общепринятых механизмов аутентификации и шифрования трафика, прекрасно описывался четырьмя уровнями модели DOD (TCP/IP). Но с ростом числа узлов в сети Интернет и широким распространением WWW стало понятно, что необходимо решать вопросы безопасности. Реализация безопасности плохо укладывается в модель TCP/IP, дополнительные протоколы шифрования удобнее описывать с помощью модели OSI/ISO. Поэтому кроме прикладного уровня мы также коснемся верхних уровней модели OSI/ISO, их изначального предназначения и современных задач, относящихся к этому уровню (в частности задач аутентификации и шифрования). Стоит отметить, что, так как модель OSI/ISO на практике не была реализована, имеет смысл говорить не столько о прямом соответствии верхних уровней OSI/ISO используемым протоколам, сколько о задачах, решаемых теми или иными протоколами, и связи этих задач с соответствующими уровнями.

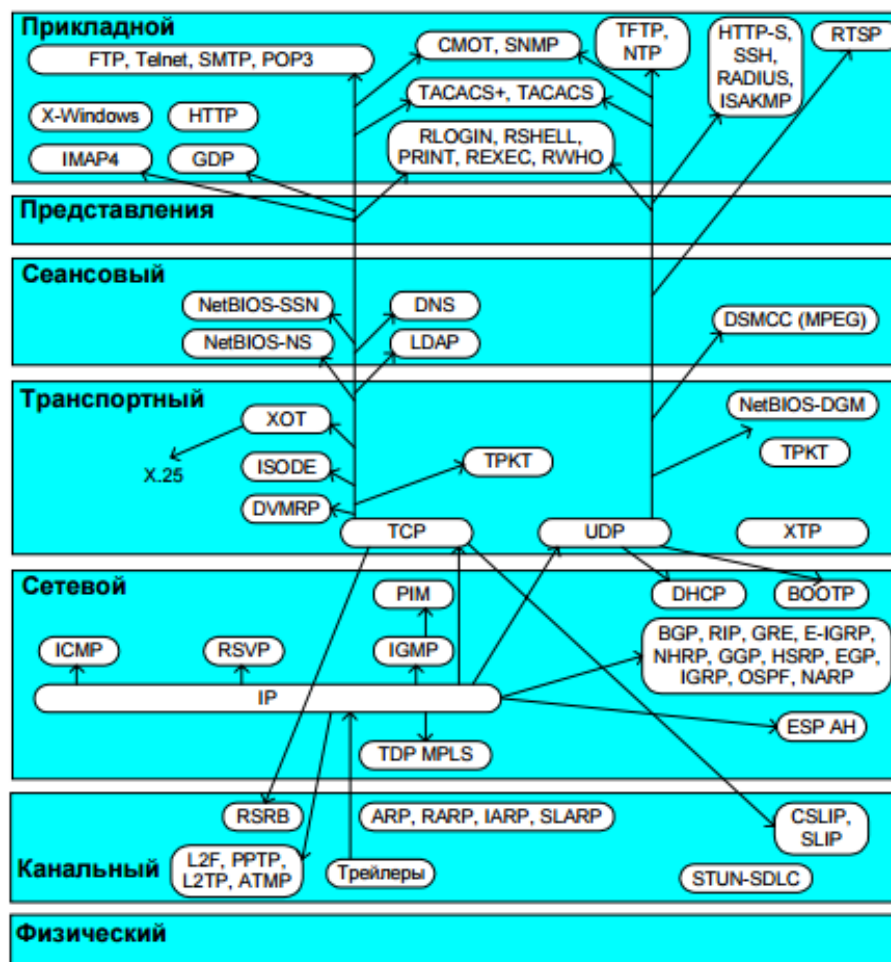
Обзор верхних уровней модели OSI

Уровень	Название	TCP/IP	Уровень
7 уровень	Прикладной уровень	Прикладной уровень	4 уровень
6 уровень	Уровень представления		
5 уровень	Сеансовый уровень		
4 уровень	Транспортный уровень	Транспортный уровень	3 уровень
3 уровень	Сетевой уровень	Межсетевой уровень	2 уровень
2 уровень	Канальный уровень	Сетевых интерфейсов	1 уровень
1 уровень	Физический уровень		

Отнести определенный вид сетевого оборудования, алгоритм или протокол к одному из нижних (L1–L4) уровней модели OSI несложно. Но при попытке соотнести протоколы с уровнями L5–L7 могут возникнуть проблемы. Наиболее традиционная концепция предполагает, что сеансовый, представления и прикладной уровни модели OSI/ISO соотносятся с прикладным уровнем стека TCP/IP. Но есть и трактовки, согласно которым протокол TCP относится не только к транспортному, но и к сеансовому уровню (см. [«Анализ трафика мультисервисных сетей»](#)).

7	WWW	SNMP	FTP	Telnet	TFTP	SMTP	I
6							
5	TCP			UDP			II
4							
3	IP	ICMP	RIP	OSPF	ARP	III	
2	Ethernet, Gigabit Ethernet, Token Ring, PPP, FDDI, X.25, SLIP, Frame Relay...						IV
1							
уровни OSI	Протоколы						уровни TCP/IP

С другой стороны, вопрос отношения протоколов к сеансовому или уровню представления в целом оказывается довольно спорным. Трактовки в разных источниках могут сильно отличаться. Некоторые служебные протоколы, такие как протоколы маршрутизации, RIP, OSPF, BGP, часто относятся к сетевому уровню модели OSI/ISO. При этом технически из них поверх IP работает только OSPF, протокол BGP работает поверх TCP (179 порт), протокол RIP работает поверх UDP. Протокол DNS, традиционно относящийся к прикладному уровню, в источнике отмечен как сеансовый. Протокол DHCP также относится к прикладному уровню, но ведет свое происхождение от протокола ARP, относящегося к сетевому, а иногда даже к канальном уроню (как видите, даже когда речь идет о нижних уровнях модели OSI, тоже не всегда можно однозначно ответить, к какому уровню что относится). Чаще всего протоколы DHCP и DNS относят к прикладному уровню, но встречается мнение, что DHCP относится к сетевому, а DNS — к сеансовому. Например, по версии <http://www.protocols.ru/files/Protocols/TCP/IP.pdf>.



В этом есть смысл. Мы тоже рассматривали DHCP на уроке по сетевому уровню (кроме того, протокол DHCP является далеким потомком протокола RARP). Фактически можно наблюдать два подхода в соотношении протоколов с соответствующими уровнями — по уровню инкапсуляции (тогда DHCP, DNS и даже BGP и RIP оказываются прикладными) либо по решаемым задачам (в таком случае DHCP, BGP, RIP оказываются сетевыми, а DNS и LDAP — сеансовыми). Распространенная точка зрения на протоколы, как показывает практика, где-то посередине. Отметим, что попытка строго ориентироваться на инкапсуляцию также не всегда может быть успешной, особенно когда окажется, что число уровней больше семи и уровни повторяются, а не только не следуют традиционному порядку. Такие протоколы, как RPC, могут относиться к сеансовому уровню, но на практике работать выше, чем протоколы прикладного уровня (например, XML-RPC поверх HTTP). Фактически модель OSI во многих случаях соблюдается очень плохо.

В случае использования традиционного стека TCP/IP (без шифрования) соотношение протоколов с моделью OSI/ISO остается довольно прозрачным: в выдаче Cisco Packet Tracer можно видеть, что уровни L5 и L6 остаются незаполненными. Нешифрованный протокол HTTP (инкапсулированный в TCP, который инкапсулирован в IPv4, который следует, например, через Ethernet) прекрасно описывается моделью TCP/IP. Но уже протокол HTTPS (шифрованный HTTP, инкапсулированный в безопасный протокол TLS, который уже вкладывается в TCP) в модель TCP/IP полностью не укладывается. Точнее, с точки зрения TCP/IP именно HTTPS является прикладным протоколом, но с

точки зрения технической реализации это не так: на самом деле HTTPS — не что иное как тот же самый протокол HTTP, который вложен в протокол TLS.

Сравните:

Уровень	Модель OSI	Модель TCP/IP	Уровень
7. Прикладной уровень	HTTP	HTTPS	4. Прикладной уровень
6. Уровень представления	TLS (шифрование)		
5. Сеансовый уровень	TLS (аутентификация и обмен ключами)		
4. Транспортный уровень	TCP	TCP	3. Транспортный уровень
3. Сетевой уровень	IPv4	IPv4	2. Межсетевой уровень
2. Канальный уровень	IEEE 802.3	IEEE 802.3	1. Уровень сетевых интерфейсов
1. Физический уровень	IEEE 802.3		

Сделаем небольшой обзор каждого из вышестоящих уровней по отдельности.

Прикладной уровень (Application layer)

Большинство протоколов относятся как раз к прикладному уровню. По замыслу разработчиков OSI/ISO, прикладной уровень дает доступ к сетевым услугам к приложению. На практике же сами приложения и реализуют соответствующий протокол. То есть веб-серверы nginx и Apache2 самостоятельно и независимо реализуют протокол HTTP, получая доступ к услугам TCP/IP ядра через механизм сокетов Беркли. Любой клиент, любой чат, любой браузер имеет в своем составе код, отвечающий за реализацию соответствующего прикладного протокола (как самостоятельно, так и с подключением соответствующих библиотек).

Прикладной протокол имеет нижнюю границу, по замыслу создателей, передавая запросы уровню представления, либо, в соответствии с моделью TCP/IP, через механизм сокетов Беркли, транспортному уровню, а на практике — любому другому нижестоящему протоколу (TCP, TLS, SSH, даже в HTTP возможна инкапсуляция). Именно поэтому Wireshark не следует той или иной сетевой модели, отображая инкапсуляцию просто по уровню вложения. При этом можно отметить, что прикладной протокол не имеет верхней границы. Фактически после заголовков прикладного протокола могут содержаться данные, которые сами могут управлять соединением. Так, файл HTML,

передаваемый в поле данных протокола HTTP, сам может содержать HTTP-заголовки: `<meta http-equiv="Content-Type" content="text/html; charset=utf-8">` или `<meta http-equiv="Pragma" content="no-cache">`; а наличие таких тегов, как или `<link href="style.css" rel="stylesheet" type="text/css">` и/или `<script type="text/javascript" src="main.js">` и/или `` будет заставлять браузер открывать новые сокет, то есть инициировать новые TCP-соединения, через которые параллельно для ускорения отправлять соответствующие GET-запросы для получения указанных в атрибутах тегов файлов.

По назначению протоколы прикладного уровня можно разделить на служебные и пользовательские. Служебные протоколы реализованы на уровне компонентов операционной системы и работают прозрачно для пользователя. К ним относятся DHCP, DNS, TFTP, NTP, SNMP. Они часто реализуются на основе протокола UDP и не все их относят к прикладному уровню. Остальные протоколы — пользовательские, связанные с теми или иными программами. Мы запускаем браузер, чтобы получить доступ к WWW, почтовый клиент, чтобы получить доступ к электронной почте, мессенджер, чтобы получить доступ к протоколу чата, и т. д.

Для использования соответствующего прикладного протокола приложения открывают соответствующий TCP- или UDP-сокет и отправляют/получают данные, не беспокоясь о том, каким образом они будут доставлены, и используя соглашение о том, что TCP дает гарантированную доставку, а UDP — нет.

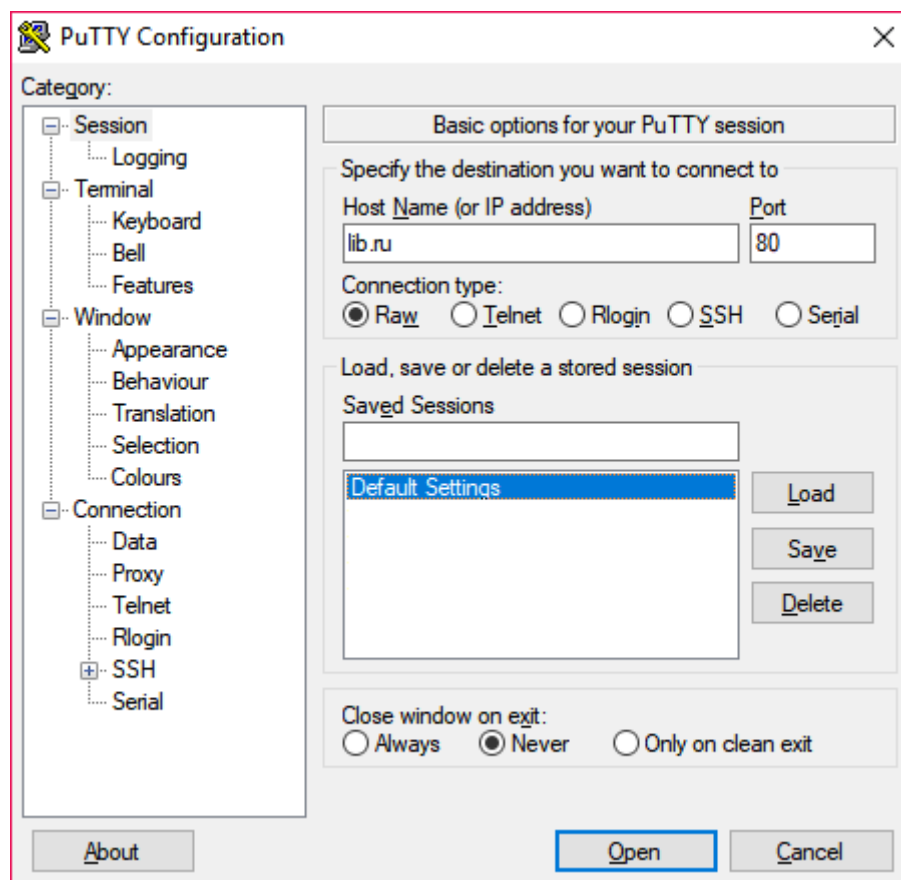
Технически протоколы прикладного уровня также можно разделить на несколько групп:

1. Простые текстовые: формат протокола больше похож на командную строку для работы с оболочкой операционной системы, такой как shell, или с SQL-клиентом. Клиент отправляет текстовые команды-запросы, сервер передает ответы и данные. Такой подход наследуется от работы с терминалами на заре компьютерной эры, и до сих пор для отладки и тестирования таких протоколов может применяться Telnet (хотя это не всегда и не совсем верно). К ним относятся многие прикладные протоколы, такие как HTTP, SMTP, FTP.
2. Текстовые протоколы на базе языка разметки — как правило, на базе XML (eXtended Markup Language) или JSON. К таковым протоколам относится, например, XMPP (eXtensible Messaging and Presence Protocol), используемый в Jabber. Технически сюда можно отнести XML-RPC, JSON-RPC и SOAP, но эти протоколы используют заголовки HTTP, а не непосредственно вкладывают XML в TCP-сессию, как это делает XMPP.
3. Двоичные протоколы. Открывая TCP-сессию или отправляя данные по протоколу UDP, можно передавать и двоичные структуры.

Но стоит отметить, что протокол HTTP, будучи текстовым, может передавать и двоичные данные, а вот протокол SMTP уже не может, и требуется перекодирование данных в base64. Для представления в ASCII по стандарту MIME (Multipurpose Internet Mail Extensions).

Особняком стоит Telnet. Фактически он реализует небольшую надстройку для удаленной консоли, самостоятельно лишь добавляя символы телетайпного ввода новой строки (CR/LF) и добавляя дополнительные управляющие команды. В отличие от FTP, который сам исполняет команды, входящие в реализацию сервера, Telnet передает команды другой программе, поэтому команды будут разные в зависимости от того, к чему вы подключаетесь: к shell на машине с UNIX-подобной ОС или к CLI на маршрутизаторе/коммутаторе Cisco.

Пример простого текстового протокола: в PuTTY можно выбрать Raw-соединение и не закрывать окно после завершения сеанса. Мы подключимся к серверу lib.ru по протоколу HTTP, непосредственно вводя с клавиатуры методы и заголовки запросов.



В UNIX-подобных системах (GNU/Linux и Mac OS X) можно воспользоваться Telnet для подключения к сырой сессии TCP и корректной обработки Enter «по-телетайпному» как CR LF (если вы пробовали в Wireshark отследить HTTP- или FTP-трафик, то могли заметить, что строки разделяются с помощью `\r\n`).

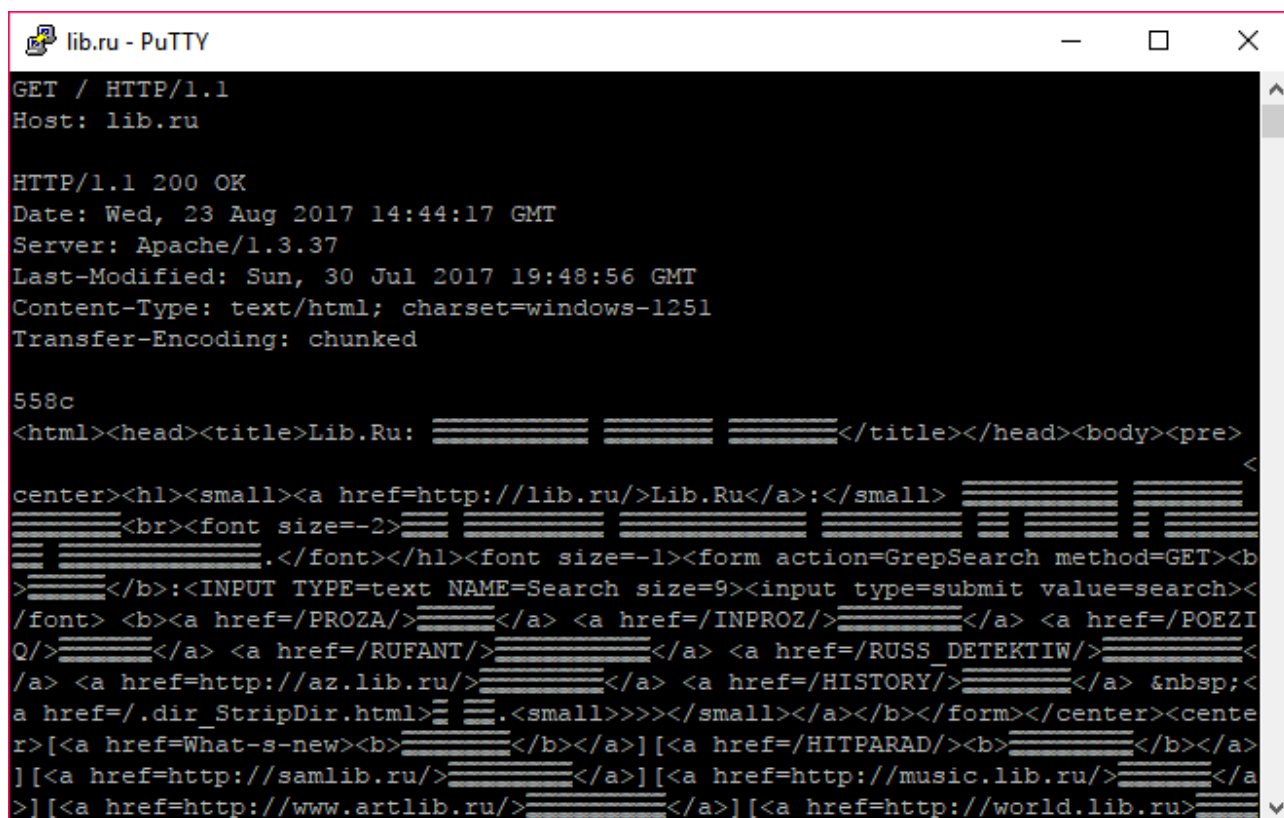
```
$ telnet lib.ru 80
```

Подключимся к lib.ru или другому веб-серверу, использующему незащищенный HTTP, на порт 80 (стандартный для нешифрованного HTTP).

```
GET / HTTP/1.1
Host: lib.ru
```


Вводим две строки и обязательно пустую строку.

Мы получим ответ. Так как telnet lib.ru 80 фактически эквивалентно telnet 81.176.66.163 80, PuTTY или Telnet выполнит запрос механизма сокетов Беркли gethostbyname("lib.ru") и получит у локального или кеширующего сервера (или в ряде случаев из /etc/hosts) IP-адрес для указанного доменного имени или имени хоста. Обратите внимание, что сервер, на котором хостится несколько сайтов, фактически не знает, какой из них, кроме дефолтного, вы хотите получить. Поэтому в спецификации HTTP/1.1 был добавлен заголовок host, позволяющий указать домен в явном виде.



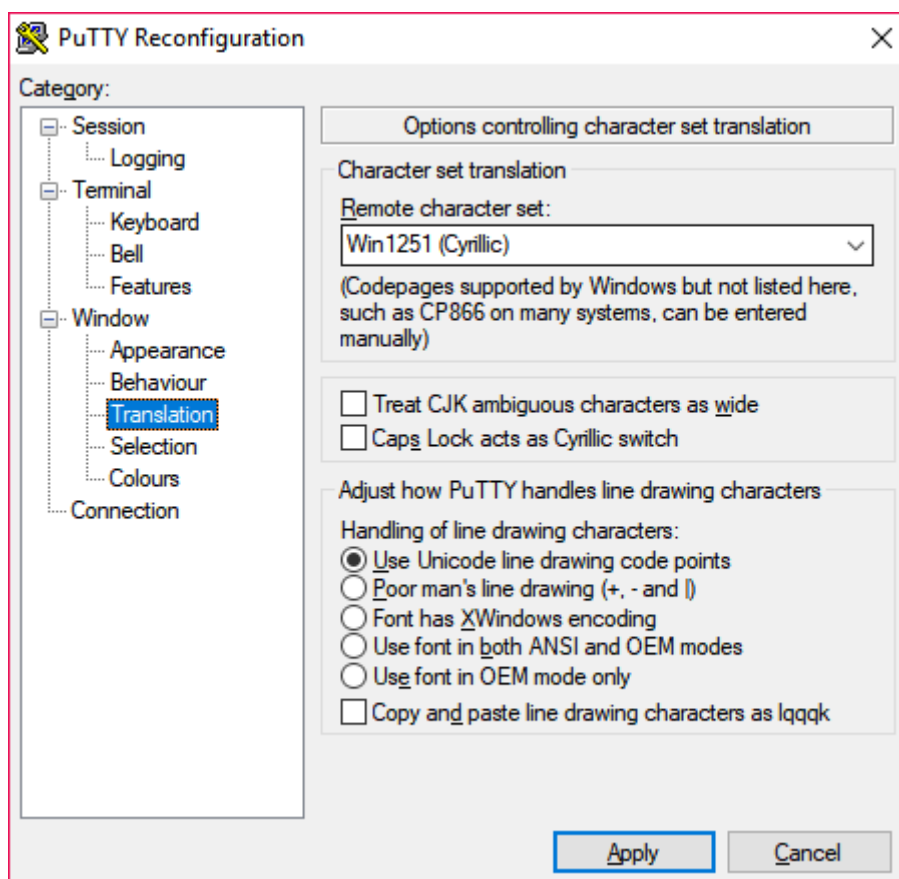
```
lib.ru - PuTTY
GET / HTTP/1.1
Host: lib.ru

HTTP/1.1 200 OK
Date: Wed, 23 Aug 2017 14:44:17 GMT
Server: Apache/1.3.37
Last-Modified: Sun, 30 Jul 2017 19:48:56 GMT
Content-Type: text/html; charset=windows-1251
Transfer-Encoding: chunked

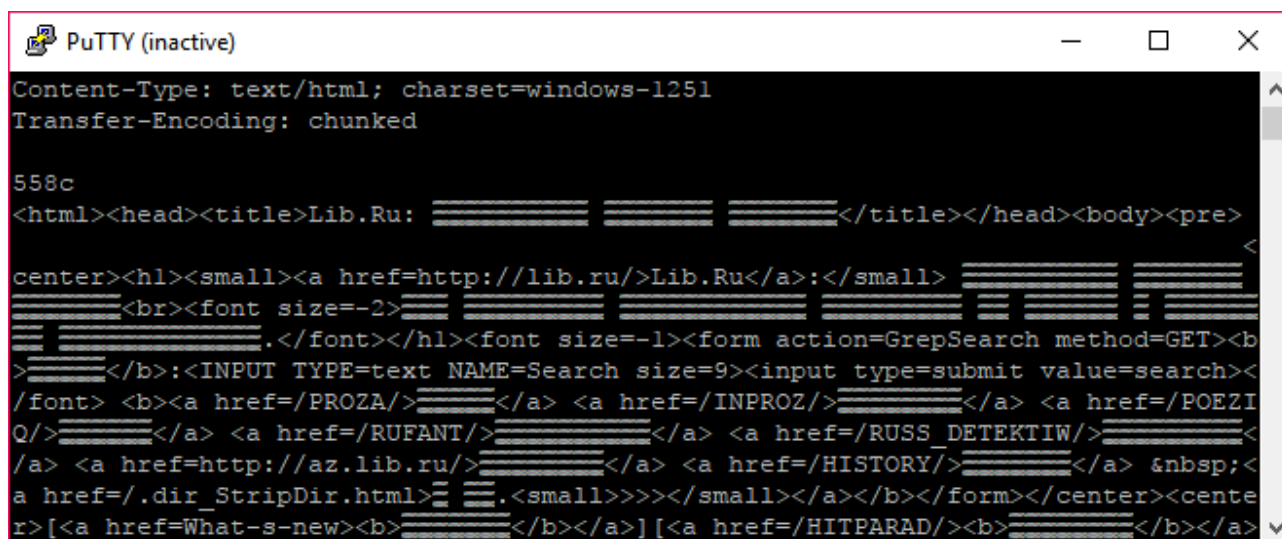
558c
<html><head><title>Lib.Ru: </title></head><body><pre>
<
center><h1><small><a href=http://lib.ru/>Lib.Ru</a>:</small>
<br><font size=-2>
</font></h1><font size=-1><form action=GrepSearch method=GET><b>
</b><input type=text NAME=Search size=9><input type=submit value=search>
</font> <b><a href=/PROZA/></a> <a href=/INPROZ/></a> <a href=/POEZI
Q/></a> <a href=/RUFANT/></a> <a href=/RUSS_DETEKTIW/></a>
<a href=http://az.lib.ru/></a> <a href=/HISTORY/></a> &nbsp;<b>
<a href=/.dir_StripDir.html>
</b></small></b></a></b></form></center><cente
r>[<a href=What-s-new><b>
</b></a>][<a href=/HITPARAD/><b>
</b></a>][<a href=http://samlib.ru/>
</a>][<a href=http://music.lib.ru/>
</a>][<a href=http://www.artlib.ru/>
</a>][<a href=http://world.lib.ru/>
```

Мы получили ответ. Кириллическая кодировка не совпадает (браузер решает эту проблему, но реализация сырой сессии PuTTY или Telnet является более низкоуровневой технологией).

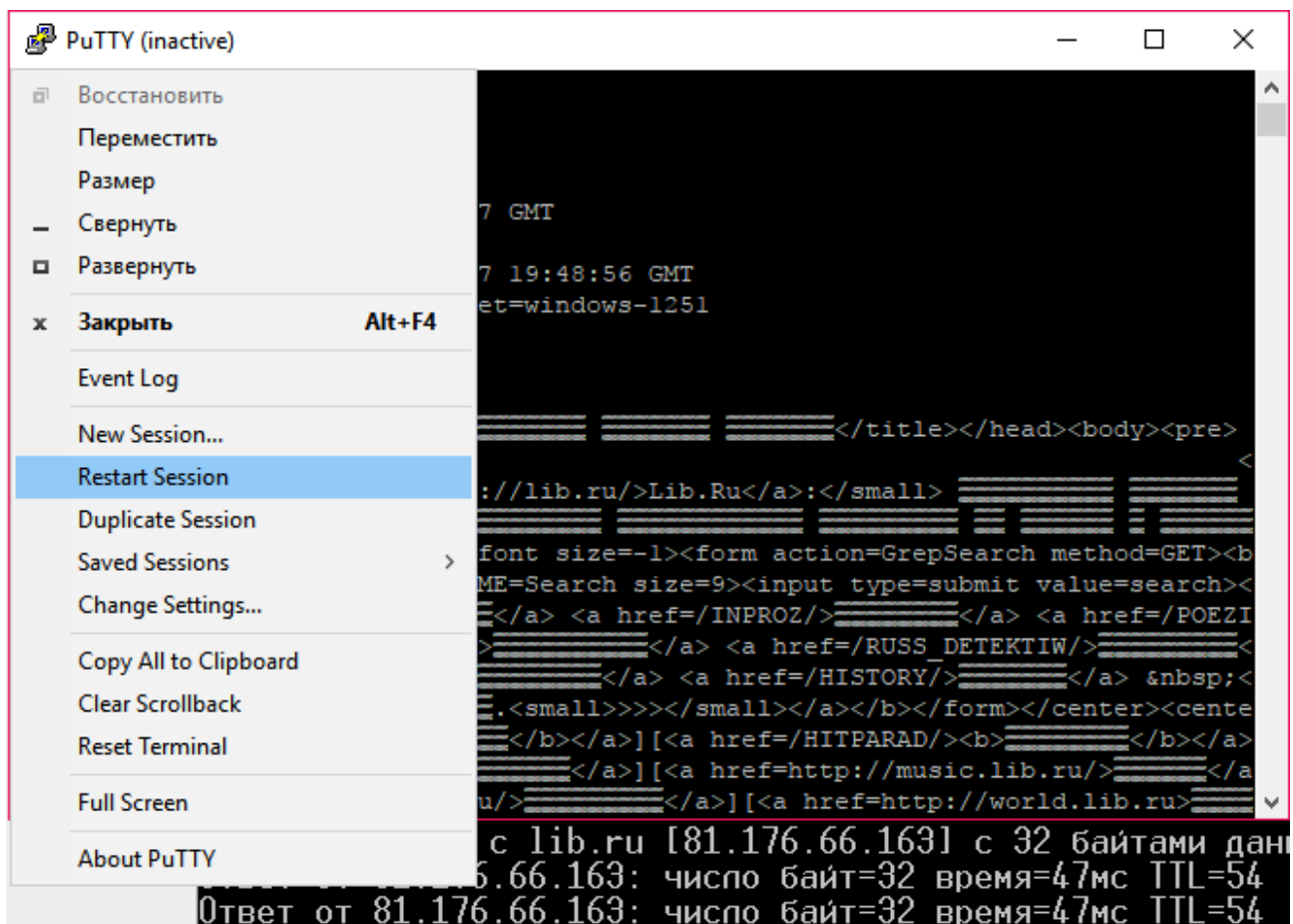
Протокол указывает, что нужно использовать windows-1251. Если мы работаем в PuTTY, можно указать кодировку.



Но уже отображенные данные не изменятся.



И сама сессия неактивна. Необходимо сделать restart сессии и повторить запрос.



Снова вводим:

```
GET / HTTP/1.1
Host: lib.ru
```

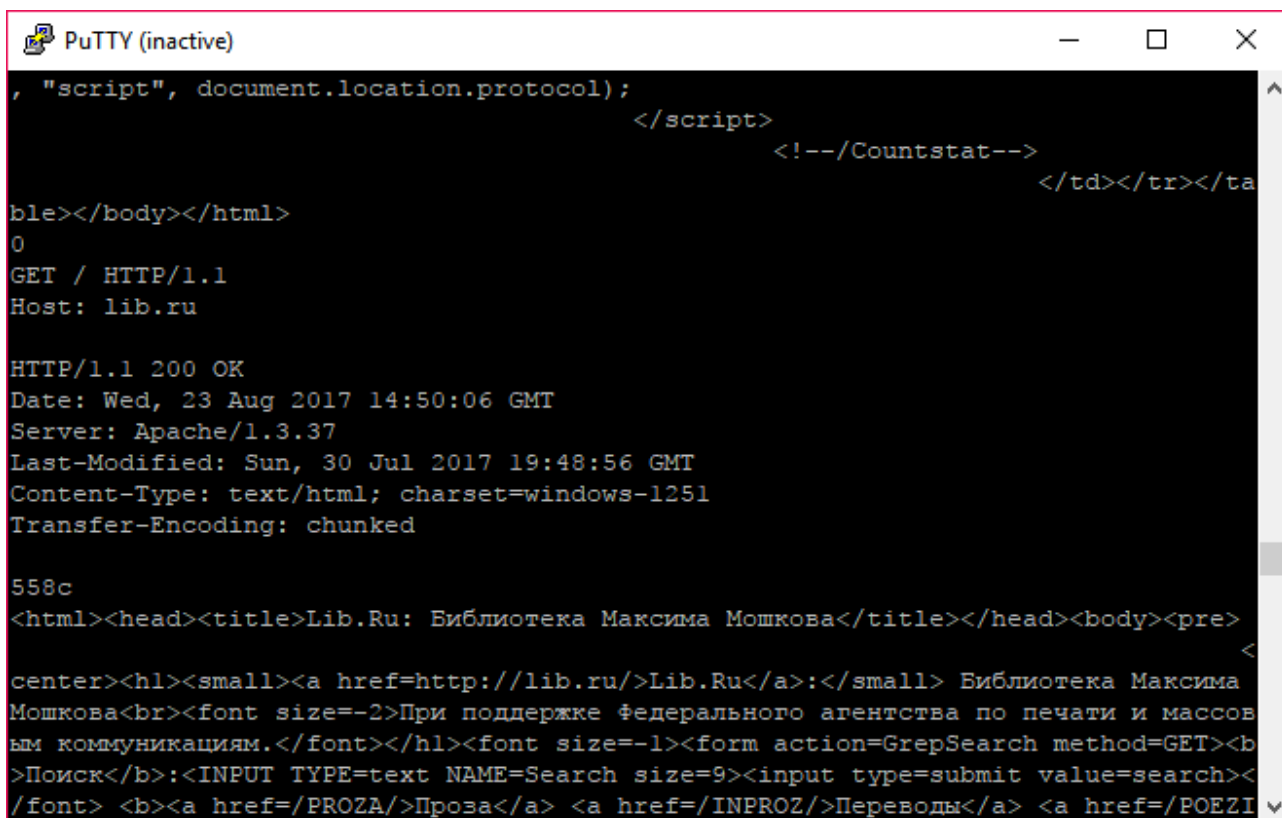
Не забываем пустую строку.

Фактически ввод выглядит так:

```
GET / HTTP/1.1\r\n
Host: lib.ru\r\n
\r\n
```

Вы легко можете в этом убедиться, если прослушаете трафик с помощью Wireshark.

После того, как вы ввели строку запроса GET и заголовок Host, а затем пустой строкой указали конец ввода, если при этом не истекло время таймаута сессии, можно посмотреть ответ.



```
PuTTY (inactive)
, "script", document.location.protocol);
</script>
<!--/Countstat-->
</td></tr></table>
</body></html>
0
GET / HTTP/1.1
Host: lib.ru

HTTP/1.1 200 OK
Date: Wed, 23 Aug 2017 14:50:06 GMT
Server: Apache/1.3.37
Last-Modified: Sun, 30 Jul 2017 19:48:56 GMT
Content-Type: text/html; charset=windows-1251
Transfer-Encoding: chunked

558c
<html><head><title>Lib.Ru: Библиотека Максима Мошкова</title></head><body><pre>
<
center><h1><small><a href=http://lib.ru/>Lib.Ru</a>:</small> Библиотека Максима
Мошкова<br><font size=-2>При поддержке федерального агентства по печати и массов
ым коммуникациям.</font></h1><font size=-1><form action=GrepSearch method=GET><b
>Поиск</b>:<INPUT TYPE=text NAME=Search size=9><input type=submit value=search><
/font> <b><a href=/PROZA/>Проза</a> <a href=/INPROZ/>Переводы</a> <a href=/POEZI>
```

Мы видим, что одного протокола прикладного уровня недостаточно, чтобы не просто получить, а еще и обработать (прочитать) информацию. Таким образом, мы подходим к уровню представления.

Уровень представления (Presentation Layer)

Уровень представления — самый неуловимый, почти мифический уровень. Его все знают, но почти каждый затрудняется ответить, что этот уровень делает и зачем он нужен.

Если найти примеры протоколов сеансового уровня ещё не слишком сложно (обычно туда относят тоннельные протоколы, такие как L2TP, PPTP, даже сам протокол TCP и иногда TLS), то с уровнем представления всё сложнее. На одной из картинок, демонстрирующих соответствие уровней OSI и протоколов, он вообще оказался пуст. Но не всё так плохо. Очень часто к уровню представления относят TLS. (Стоп! Мы же его видели на сеансовом. Это правда. Он выполняет задачи сразу на двух уровнях модели OSI.)

Несмотря на то, что очень сложно выделить протоколы, работающие на уровне представления, задачи, решаемые этим уровнем, остаются. Здесь проще всего обратиться за комментарием непосредственно к стандарту. Посмотрим в отечественном ГОСТе ЭМВОС: Взаимосвязь открытых систем | [ГОСТ Р ИСО/МЭК 7498-1-99](#):

7.2.3 Услуги, предоставляемые прикладному уровню

7.2.3.1 Уровень представления данных обеспечивает следующие возможности:

- а) идентификацию набора синтаксисов передачи;

- b) выбор синтаксиса передачи;
- с) доступ к услугам сеансового уровня.

7.2.3.2 Идентификация набора синтаксисов передачи обеспечивает одно или несколько средств представления абстрактного синтаксиса. Выбор синтаксиса предоставляет средства первоначального выбора синтаксиса передачи и последующего изменения сделанного выбора.

7.2.3.3 Услуги сеансового уровня предоставляются логическим объектам прикладного уровня в виде услуг уровня представления данных.

Эта идея родом из 70-х годов, когда начали разрабатывать OSI. Тогда компьютеры были не столь унифицированными, как сейчас, и предыдущий опыт использования показывал, что все разнородное оборудование (разные архитектуры, кодировки, типы данных) требует использования соглашений, которые бы позволили работать совместно. По замыслу авторов модели, протоколы сеансового уровня решили бы проблему представления данных из одних видов (форматов, кодировок, видов синтаксиса и т. д.) в другие.

Давайте посмотрим, что такое уровень представления по версии Microsoft.

На странице <https://technet.microsoft.com/en-us/library/cc959885.aspx> имеется следующий список:

- трансляция кодировок, например из ASCII в EBCDIC;
- преобразование данных, таких как порядок байтов, CR в CR/LF или целых чисел в вещественные;
- сжатие данных, уменьшающее число передаваемых битов;
- шифрование/дешифрование данных.

Разберем эти пункты подробнее. EBCDIC (Extended Binary Coded Decimal Interchange Code) — семейство двоично-десятичных кодировок, используемых в мэйнфреймах IBM. FTP умеет работать не только с ASCII, но и с EBCDIC. (Вообще говоря, у FTP имеется еще два режима передачи: двоичный и без перекодировки, если обе машины работают в одной и той же кодировке. Как вы понимаете, все эти задачи очень хорошо соотносятся с уровнем представления.)

ASCII и Telnet

Кодировка ASCII (American Standard Code for Information Interchange) — далекий потомок кода Бодо, который, в свою очередь, является потомком азбуки Морзе. Все трое родом из телеграфа. Этим и определяется специфика кодировки. Он содержит не только алфавитно-цифровые символы, но и управляющие. Отметим, что кодировка ASCII-семибитная, потому позволяла закодировать 128 (от 0 до 127) символов и управляющих кодов. Символы с 0 по 31 являлись управляющими, остальные символы служили для представления латинских букв в верхнем и нижнем регистре, цифр и знаков препинания. 127-й символ тоже считался управляющим.

ASCII Code Chart

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2		!	"	#	\$	%	&	'	()	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{		}	~	DEL

Как это работало? Представим две печатные машинки, связанные телеграфом. После установки соединения все действия, совершенные на первой машинке, отправляются на вторую.

Вы печатаете символ **H** — удаленная машинка также печатает символ **H**. Обе машинки смещают барабан влево на одну позицию для печати следующего символа.

Вы печатаете символ **i**. Человек на той стороне видит надпись «Hi».

Но вы подумали, что ошиблись, и нажимаете кнопку Backspace. Если в первых двух случаях отправлялись символы с кодами 48 и 69, теперь будет отправлен «символ» с кодом 08 («**BS**»). Это приводит к тому, что на той стороне вместо печати символов каретка смещается влево (или барабан вправо — в зависимости от конструкции). То же происходит и на вашей печатной машинке.

Теперь вы нажимаете символ **e**, и передается код 65. Так постепенно появляется надпись «Hello, world!», причем символ **e** будет напечатан поверх **i**. Теперь надо начать печатать с новой строки. Для этого служат две команды. Первая, с кодом 0D («**CR**» — Carriage Return — возврат каретки), — перевод печатающей каретки в самую левую позицию. Следующая команда, с кодом 0A («**LF**» — Line Feed — перевод строки), — барабан с бумагой прокручивается на одну строку вверх, позволяя печатать с новой строки.

А что будет, если передать символ с кодом 09 («**HT**» — горизонтальная табуляция)? Будет напечатано 8 пробелов.

Связь может быть двусторонней. То есть та сторона тоже может что-то написать или отправить нам управляющий код. Поведение обеих печатных машинок будет идентичным.

В таблице можно увидеть и другие коды команд: **NUL** или **NOP** — ничего не делать, **BEL** — звонок, **VT** — вертикальная табуляция, **SYN** и **ACK**, знакомые нам по протоколу TCP.

Если одну машинку заменить на компьютер, мы получим простейший терминал для работы с компьютером, который может как получать от нас сообщения, так и писать нам ответы. Эта логика во многом определила механизм работы с ЭВМ и по настоящее время. Более того, работа с клавиатурой не отличается от отправки сообщения с печатной машинки.

Кнопка Tab генерирует символ с кодом 09, который операционная система обрабатывает по-разному. В графическом интерфейсе это будет переключение окон или полей ввода. В редакторе nano в Linux на экране будет напечатано 8 пробелов, но в самом файле будет записан всего лишь один символ с кодом 09.

Кнопка Backspace генерирует символ с кодом 08, который обрабатывается драйвером клавиатуры и приводит к смещению одного символа влево и затиранию ранее напечатанного на экране символа.

Кнопка Esc генерирует символ с кодом 1B, который будет обработан операционной системой, как правило, как отмена некоторого действия.

Кнопка Enter генерирует символ с кодом 0D, который приводит к выполнению на экране действий, фактически соответствующих последовательности 0D 0A, и т. д.

Также отметим, что в текстовых протоколах строки обязательно разделяются последовательностью 0D 0A. А вот хранение текстовых файлов может отличаться. И если в Windows разделителем является комбинация символов 0D 0A, то в UNIX — только 0A. Таким образом, для корректной работы файл, переданный из Windows в Linux, должен быть перекодирован, чтобы вместо 0D 0A остались только 0A. И сделать это можно, например, утилитой dos2unix. Об этом говорят на сайте Microsoft, относя данную задачу также к уровню представления.

Теперь можно рассмотреть протокол Telnet. По сути он представляет собой сырую TCP-сессию, в которой для управления используются управляющие коды ASCII и особые управляющие коды Telnet. В 7-битной ASCII управляющим символом был также символ с кодом 127. Похожим назначением в Telnet является 255-й символ после расширения семибитной кодировки до восьми бит (задача перекодировки из восьмибитной в семибитную при использовании соответствующего оборудования также относится к уровню представления).

ASCII таблица с кодировкой CP866

	00	10	20	30	40	50	60	70		80	90	A0	B0	C0	D0	E0	F0
0		►		0	@	P	`	p	0	А	Р	а	⌘	⌘	⌘	Р	≡
1	☐	◄	!	1	А	Q	a	q	1	Б	С	б	⌘	⌘	⌘	С	±
2	☐	↑	"	2	В	R	b	r	2	В	Т	в	⌘	⌘	⌘	Т	>
3	♥	!!	#	3	С	S	c	s	3	Г	У	г		⌘	⌘	У	<
4	♦	¶	\$	4	Д	T	d	t	4	Д	Ф	д	⌘	⌘	⌘	Ф	⌘
5	⌘	§	%	5	Е	U	e	u	5	Е	Х	е	⌘	⌘	⌘	Х	J
6	⌘	=	&	6	Ф	V	f	v	6	Ж	Ц	ж	⌘	⌘	⌘	Ц	÷
7	*	⌘	'	7	Г	W	g	w	7	З	Ч	з	⌘	⌘	⌘	Ч	≈
8	☐	†	<	8	Н	X	h	x	8	И	Ш	и	⌘	⌘	⌘	Ш	°
9	○	↓	>	9	И	Y	i	y	9	Й	Щ	й	⌘	⌘	⌘	Щ	-
A	☐	→	*	:	J	Z	j	z	A	К	Ь	к	⌘	⌘	⌘	Ь	•
B	♂	←	+	;	К	[k	[B	Л	М	л	⌘	⌘	⌘	М	√
C	♀	⌘	,	<	Л	\	l	l	C	Н	Ь	н	⌘	⌘	⌘	Ь	n
D	♂	↔	-	=	М]	m	>	D	Н	Э	н	⌘	⌘	⌘	Э	2
E	♂	▲	.	>	Н	^	n	~	E	О	Ю	о	⌘	⌘	⌘	Ю	●
F	※	▼	/	?	О	_	o	Δ	F	П	Я	п	⌘	⌘	⌘	Я	

Таким образом, каждый символ воспринимался либо как управляющий код (первые 32 символа, например, CR и LF), либо как алфавитно-цифровой (или псевдографический) символ, либо как команда. По-особому трактовался символ #255 (FF в шестнадцатеричной системе счисления). За символом с кодом 255 следовал код команды либо еще раз 255 для передачи символа 255: #255#код команды.

Например:

- #255#242 — синхронизация (Synch) обмена данными. Эта команда всегда сопровождается TCP Urgent notification;
- #255#249 — ожидание передачи данных;
- #255#255 — символ с кодом #255.

Но Telnet оказался мало совместим с кодировкой CP-1251, используемой в Windows.

Á	à	,	è	„	...	†	‡	€	‰	É	«	Й	Ї	Ó	Ú
128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
ă	‘	’	“	”	•	—	—	ë	™	é	»	ò	í	ó	ú
144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
nbsp	ÿ	Ы	Э	Я	Ы	І	Š	Ě	©	Ю	«	¬	shy	®	ђ
160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
•	±	Ы	Э	’	μ	¶	•	ě	№	Ю	»	Э	Ю	Я	ђ
176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
А	Б	В	Г	Д	Е	Ж	З	И	Й	К	Л	М	Н	О	П
192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
Р	С	Т	У	Ф	Х	Ц	Ч	Ш	Щ	Ъ	Ы	Ь	Э	Ю	Я
208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
а	б	в	г	д	е	ж	з	и	й	к	л	м	н	о	п
224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
р	с	т	у	ф	х	ц	ч	ш	щ	ъ	ы	ь	э	ю	я
240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

Кодировка CP-1251 («Windows-1251»), вторая половина таблицы. Первая совпадает с ASCII

В примере выше мы показали, что можно использовать Telnet для подключения к текстовому протоколу транспортного уровня TCP. Главным достоинством Telnet в данном случае является правильная обработка символов CR LF. Но стоит указать, что, в отличие от использования HTTP поверх TCP, у нас фактически получается использование HTTP поверх Telnet поверх TCP, и Telnet-клиент может попытаться отправить Telnet-команды серверу или, наоборот, интерпретировать символ «я» из кодировки CP-1251 и следующий за ним как Telnet-команду (а букву Я, соответственно, отображать не будет). Потому нужно использовать PuTTY в режиме RAW, который также корректно обрабатывает CR LF, либо netcat (но netcat не обрабатывает CR LF, потому требуется использование дополнительных инструментов, например, stty -icrnl).

В некоторых случаях использовать Telnet для отладки протоколов правильно. Так, поток управления FTP фактически работает поверх Telnet-протокола, потому его можно использовать для отладки, и в отличие от случая с HTTP или SMTP, он будет работать корректно. Но буква Я не будет обрабатываться в кодировке CP-1251. Впрочем, она по тем же самым причинам не обрабатывается и в FTP (проверьте через любой клиент, используя настоящий FTP).

Расширение кодировок с 7-битной ASCII до 8-битной добавило возможность использовать еще 128 символов. Это достаточно, чтобы закодировать один национальный алфавит и символы псевдографики, но недостаточно, чтобы закодировать несколько разных национальных алфавитов (в Windows такая концепция называлась Code Page). Более того, даже для одного алфавита существовали разные кодировки (KOI-8R, происходящая от семибитной KOI-7R, и используемая в UNIX CP-866 в DOS, обе с псевдографикой, CP-1251 в Windows без псевдографики).

Проблема перекодирования кодировок

Поскольку кодировок стало много, возникла потребность передавать данные об используемой на странице кодировке. Одна из неудачных попыток была с применением DNS. Предполагалось, что если по субдомену `www` не удалось получить кодировку по умолчанию (не совпала с локальной), можно было запросить ее в явном виде через указание соответствующего субдомена. Получив запрос на `www-koi8-r`, веб-сервер выдал бы файл в кодировке `koi8-r`. Концепция не прижилась. Более того, не было стандарта именования субдоменов (так, кодировке CP-1251 могли соответствовать субдомены `cp-1251`, `www-cp-1251`, `windows-1251` и т. д.).

Кое-какие из этих доменов существуют до сих пор, иные же можно найти в кеше поисковиков и архиве Интернета.

Примеры.

- <http://www-koi8-r.rambler.ru;>
- <http://www-koi8r.comch.ru;>
- [http://windows-1251.www.stankin.ru/rus_ver/std_prc/inteh/itvs/;](http://windows-1251.www.stankin.ru/rus_ver/std_prc/inteh/itvs/)
- <http://cp1251.www.orgland.ru/~maxim/Kopku/actors.html;>
- [http://cp1251.deol.ru/search/;](http://cp1251.deol.ru/search/)
- <http://www-cp1251.comch.ru/~viart/mir/zhur/lat/proza/bilina.htm;>
- [http://koi8-r.www.orgland.ru/.](http://koi8-r.www.orgland.ru/)

Исторический пример:

```
546% netcat www-koi8-r.rambler.ru 80
HEAD / HTTP/1.0
Host: www.rambler.ru
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, */*
accept-charset: koi8-r

HTTP/1.1 200 OK
Date: Sat, 07 Nov 1998 14:54:26 GMT
Server: Apache/1.3.3 (Unix) Rambler/1.4 rus/PL26.5
Connection: close
Content-Type: text/html; charset=koi8-r
Expires: Thu, 01 Jan 1970 00:00:01 GMT
Last-Modified: Sat, 07 Nov 1998 14:54:31 GMT
Vary: accept-charset
```

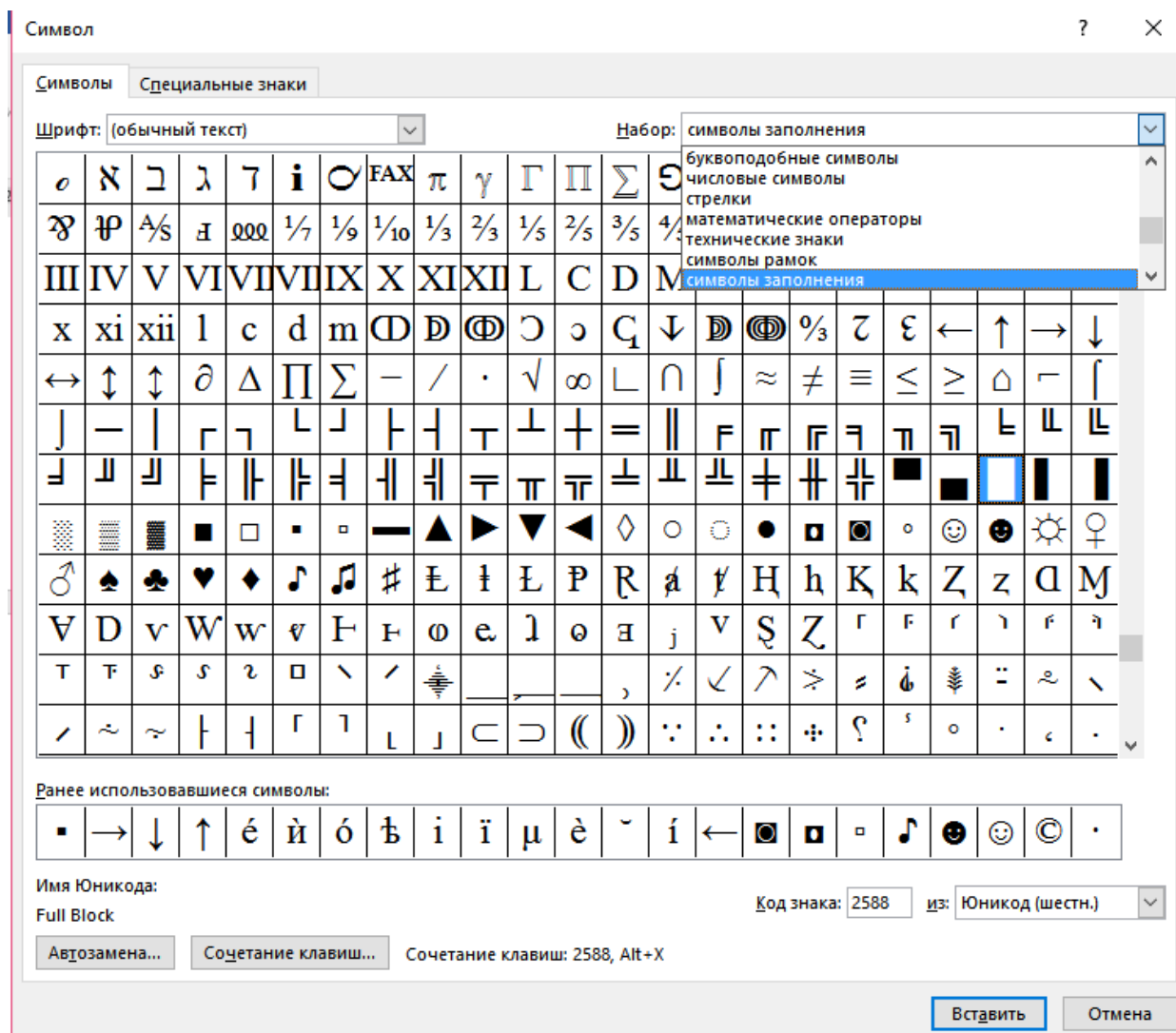
<http://apache.lexa.ru/mail-archive/msg01824.htm>

Фактически кодировка указывается в соответствующих заголовках HTTP (`accept-charset`, `content-type` и в HTML).

На практике вопросы кодирования/перекодирования/указания кодировок решают сами прикладные протоколы или даже передаваемые файлы (FTP, SMTP, HTTP, HTML, XML). Все равно в Интернете

еще можно иногда встретить «кракозябры» — неправильно перекодированный файл. Более того, попытка его исправить неправильной раскодировкой может сильно испортить файл, сделав невозможной или сложной обратной перекодировку. Для этого понадобятся уже соответствующие алгоритмы, вплоть до применения статистического анализа либо путем перебора возможных кодировок. Ранее существовали соответствующие прикладные утилиты.

Одна из попыток решить проблему «кодовых страниц» — Юникод. Сам Юникод появился давно, существуют вариации UTF-7, UTF-8 (переменной длины), UTF-16 и даже UTF-32. В сети, как правило, используется UTF-8, в Windows — UTF-16. Юникод позволяет закодировать множество символов и использовать их совместно.



Порядок байтов (byte order)

Не менее важная проблема, указываемая на сайте Microsoft, — порядок байтов.

Существуют два (на самом деле больше, но два — наиболее используемые) порядка байтов. Один из них привычен нам: разряды записываются слева направо. Это big-endian, или тупоконечный,

называемый часто «сетевым порядком байтов» (network byte order), так как используется в том числе в TCP/IP и компьютерных сетях, и little-endian, остроконечный, или «интеловский». Он неудобен для восприятия человеком, но эффективен при выполнении арифметических операций, почему и применяется в процессорах Intel. Существуют процессоры, работающие в разном порядке байтов или позволяющие переключателем аппаратно задать режим работы. ARM и MIPS существуют в little-endian вариациях.

```

C:\test\hello.exe  ↓FRO -----  a32 PE .00400043  www.hiew.ru
.00400026: 0000          add     [eax], al
.00400028: 0000          add     [eax], al
.0040002A: 0000          add     [eax], al
.0040002C: 0000          add     [eax], al
.0040002E: 0000          add     [eax], al
.00400030: 0000          add     [eax], al
.00400032: 0000          add     [eax], al
.00400034: 0000          add     [eax], al
.00400036: 0000          add     [eax], al
.00400038: 0000          add     [eax], al
.0040003A: 0000          add     [eax], al
.0040003C: 800000      add     b, [eax], 0
.0040003F: 000E          add     [esi], cl
.00400041: 1F          pop     ds
.00400042: 8A000000    mov     edx, 009B4000E ; 'o! B'
.00400047: CD21       int     021 ; '! '
.00400049: B8014CCD21 mov     eax, 021CD4CD1 ; '! =L@'
.0040004E: 54          push    esp
.0040004F: 6869732070 push    070207369 ; 'p si'
.00400054: 726F       jc      .0004000C5 --↓1
.00400056: 677261     jc      .0004000B8 --↓2
.00400059: 6D          insd
.0040005A: 206361     and     [ebx][061], ah
.0040005D: 6E          outsb
.0040005E: 6E          outsb
.0040005F: 6F          outsd
.00400060: 7420       jz      .000400082 --↓3
.00400062: 626520     bound  esp, [ebp][020]
.00400065: 7275       jc      .00040000C --↓4
1Help 2PutBk 3Edit 4Mode 5Goto 6Refer 7Search 8Header 9Files 10Quit
  
```

Число 09 B4 00 0E в машинном представлении на процессорах Intel выглядит как 0E 00 B4 09.

Поэтому при передаче данных необходимо использовать соответствующие функции (на самом деле макросы). Если порядок байт машины не сетевой, функция в данной реализации будет преобразовывать его. Если сетевой — в библиотеках для этой архитектуры данный макрос будет просто заглушкой и ничего не будет делать.

- ntohs() — network to home short.
- htons() — home to network short.
- ntohl() — network to home long.
- htonl() — home to network long.

Данные функции позволяют преобразовать из сетевого порядка в домашний и из домашнего в сетевой двухбайтные и четырехбайтные слова. Они используются в реализации серверов, сетевых драйверов. В частности, при передаче IP-адреса и номера порта в функцию bind их также необходимо

преобразовать в сетевой порядок байт. Это один из примеров реального применения преобразования представления в реализации TCP/IP.

Другой пример — UTF. Как быть, если текст будет преобразован? В этом случае возникнет ситуация, похожая на передачу файла с неверной кодировкой: символ 0E20 станет символьном 200E и т. д.

Для этого был придуман особый символ BOM (Byte Order Mark), который должен стоять самым первым. BOM воспринимается как неразрывный пробел нулевой длины и не виден при просмотре текста. Для UTF-16 этот символ имеет код FEFF. Если в полученном файле текст начинается с FFFE, то можно понять, что требуется преобразование порядка байтов (символ FFFE в UTF-16 не существует). Следовательно, он должен быть преобразован прикладным приложением в исходный порядок байтов.

Сжатие данных

Также на сайте Microsoft упоминается сжатие (компрессия и декомпрессия) и шифрование. Исходно этих определений нет в стандарте, но теоретически шифрование и сжатие подпадают под преобразование синтаксиса и семантики и должны быть выполнены на обоих узлах.

Сжатие бывает с потерями и без потерь. Сжатие с потерями применяется для картинок, видео, музыки, когда скорость передачи важнее качества. Без потерь — когда текст должен быть получен с точностью до байта (исполняемые программы, сырые мультимедиа-данные без потери качества и т. д.).

Сжатие может быть выполнено с применением соответствующих протоколов (кодеков в стеках IP-телефонии), самим протоколом (HTTP позволяет сжимать данные с помощью gzip, если и сервер и клиент поддерживают сжатие) либо прикладными программами (когда сразу передается сжатый файл jpeg, gif, mp4 или mp3 — примеры сжатия с потерями, архивы .tar.gz, .zip, .rar — примеры сжатия без потерь).

В HTTP можно указать, что передается сжатый архив, с помощью соответствующего MIME.

- application/zip: ZIP;
- application/gzip: Gzip.

С другой стороны, есть и возможность сжимать данные, которые внешнее приложение передает/получает в несжатом виде.

Клиент при запросе передает информацию о возможности получать данные в сжатом виде.

```
GET / HTTP/1.1
Host: www.example.com
Accept-Encoding: gzip, deflate
```

В ответе будет сообщено, что применяется сжатие.

```
HTTP/1.1 200 OK
Date: mon, 26 Jun 2016 22:38:34 GMT
Server: Apache/1.3.3.7 (Unix) (Red-Hat/Linux)
Last-Modified: Wed, 08 Jan 2003 23:11:55 GMT
Accept-Ranges: bytes
Content-Length: 438
Connection: close
Content-Type: text/html; charset=UTF-8
Content-Encoding: gzip
```

Браузер отобразит index.html, но непосредственно в поле данных протокола HTTP будут переданы сжатые файлы. Это сжатие без потерь.

Доступны следующие алгоритмы сжатия:

- Compress — программный метод UNIX «compress» (исторический, считается устаревшим и нежелательным для использования в большинстве приложений, вместо него используется gzip или deflate);
- Deflate — сжатие на основе алгоритма Deflate (описано в RFC 1951), использующего комбинацию алгоритма LZ77 и кодирования Хаффмана, упакованного в формат данных zlib (RFC 1950);
- Exi — W3C Efficient XML Interchange (Эффективный обмен XML);
- Gzip — формат GNU zip (описан в RFC 1952). Использует алгоритм Deflate для сжатия, но формат данных и алгоритм контрольной суммы отличаются от кодирования контента в Deflate. Используется довольно часто;
- Pack200-gzip — формат передачи данных для по сети для Java-архивов;
- Br — Brotli — новый алгоритм сжатия с открытым исходным кодом, специально разработанный для кодирования содержимого HTTP, реализованный в Mozilla Firefox 44 и Chromium 50.

Если данные не сжаты, то Content-Encoding должно указываться Identity, который указывает, что преобразование не используется. Это значение по умолчанию для кодирования содержимого.

В дополнение к этим некоторые неофициальные или нестандартизированные алгоритмы могут использоваться на практике серверами/клиентами:

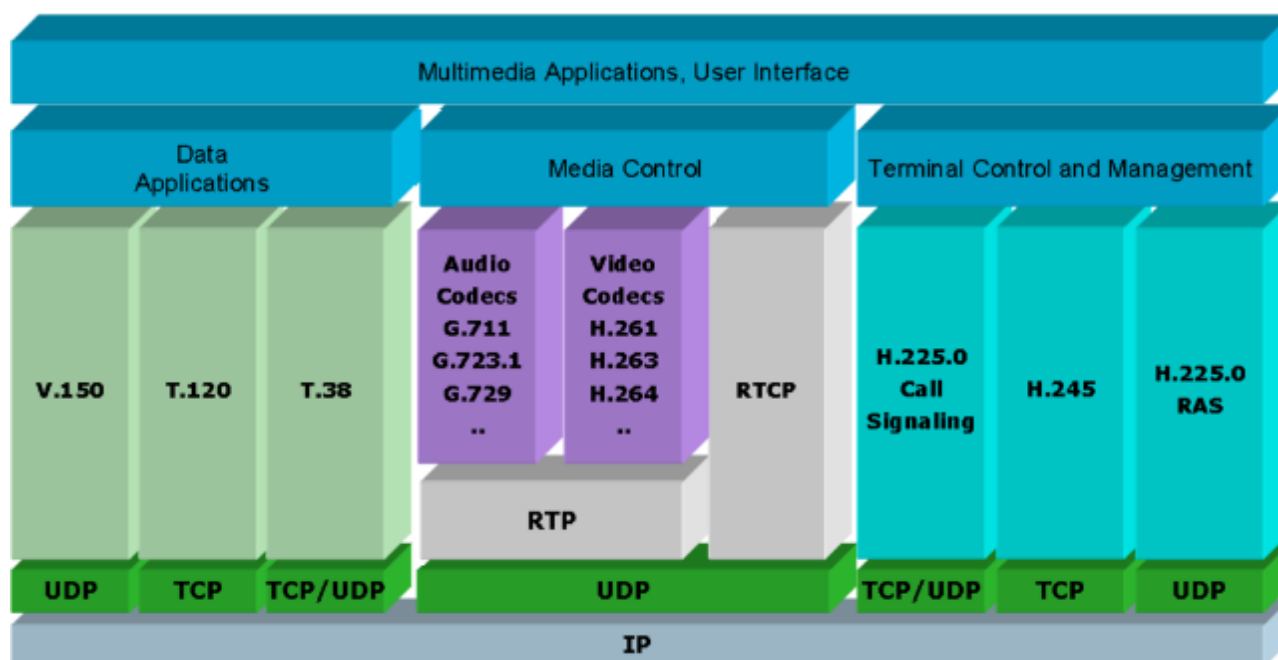
- Bzip2 — сжатие, основанное на свободном формате bzip2 (поддерживается в lighttpd);
- Lzma — сжатие, основанное на «сыром» LZMA (доступно в Opera 20 и в elinks);
- Peerdist — Microsoft Peer Content Caching and Retrieval;
- Sdch — Google Shared Dictionary Compression для HTTP, основанный на VCDIFF (RFC 3284);
- Xpress — протокол сжатия Microsoft, используемый Windows 8 и более поздними версиями для обновлений приложений Windows Store. Основанное на LZ77 сжатие может использовать алгоритм Хаффмана;

- Xz — сжатие содержимого на основе LZMA2, поддерживаемое неофициальным патчем Firefox и полностью реализованное в mget с 2013 года;

Не менее интересно работает и сжатие потоковых данных (как правило, с потерями). Рассмотрим стек H.323.

Стек H.323

Здесь имеются сигнальные протоколы (H.225.0, сюда же можно отнести конкурирующий протокол SIP), RTCP (Real-Time Control Protocol), которые можно отнести к сеансовому уровню модели OSI/ISO. Протокол RTP (Real-time Transport Protocol) работает поверх UDP в качестве транспорта для потокового вещания (не путать с RTMP — Real Time Messaging Protocol — от Adobe) и используется для вебинаров, в т. ч. в Clickmeeting, который, к тому же, работает поверх TCP. На уровне представления тогда будут находиться кодеки: аудио (G.711.1, G.723.1, G.729) и видео (H.261, H.263, H.264).



Кодеки позволяют сжать голос и видео, в том числе и при очень небольших скоростях передачи.

MIDI

MIDI (англ. Musical Instrument Digital Interface — цифровой интерфейс музыкальных инструментов) — стандарт, описывающий физический интерфейс, кодирование музыкальных данных (инструменты, ноты, эффекты, нажатия клавиш), формат музыкального файла, используемый в синтезаторах, секвенсорах, для подключения к компьютеру MIDI-клавиатур и работы с ними, в музыкальных редакторах (Steinberg Cubase, Cakewalk). Стандарт во многом ориентируется на профессиональную музыку и ориентирован на стандартную музыкальную нотацию. Существуют и альтернативы, такие как трекерная музыка (форматы MOD, XM для редактора-трекера FastTracker 2, IT для Impulse Tracker;

ModPlug Tracker понимает все указанные форматы). Трекеры так же позволяют использовать семплы музыкальных инструментов, как и MIDI, но написание музыки в трекерах имеет другую концепцию, нежели академическая композиция, и больше похоже на программирование.

XDR

XDR (External Data Representation) — международный стандарт передачи данных в Интернете. Он позволяет упаковывать данные не зависящим от архитектуры способом — таким образом, данные могут передаваться между разнородными компьютерными системами.

- Преобразование из локального представления в XDR называется кодированием.
- Преобразование из XDR в локальное представление называется декодированием.
- XDR выполнен как портативная (переносная) библиотека функций между различными операционными системами и также не зависит от транспортного уровня.

Используется в Sun RPC/ONC (Open Network Computing Remote Procedure Call — система удаленного вызова процедур) и NetCDF (Network Common Data Form — формат файлов для обмена научными данными).

SSL (Secure Socket Layer)

SSL — протокол шифрования, который обеспечивает безопасное соединение между клиентом и сервером. Протокол SSL был разработан фирмой Netscape достаточно давно. Версия 1.0 не была обнародована. Версия 2.0 была выпущена в феврале 1995 года, но имела много недостатков в плане безопасности, из-за которых и разработали SSL версии 3.0.

(По материалам

<https://blog.jenrom.com/2014/09/07/internet-fundamentals-osi-модель-уровень-представления/>)

TLS (Transport Layer Security)

TLS — протокол шифрования, обеспечивающий защищенную передачу данных между узлами в сети Интернет. Он является следующим поколением протокола SSL.

На данный момент есть три версии протокола TLS: 1.0, 1.1 и 1.2. Они имеют внутренние идентификаторы версии 3.1, 3.2 и 3.3 соответственно, поэтому иногда называются SSL 3.1, SSL 3.2 и SSL 3.3.

TLS и SSL используют асимметричную криптографию для аутентификации и симметричное шифрование для передачи данных.

В <https://blog.jenrom.com/2014/09/07/internet-fundamentals-osi-модель-уровень-представления/>

отмечается, что основная работа шифрования данных TLS и SSL проходит на 6 уровне модели OSI (уровень представления), а аутентификация — на 5 уровне модели OSI (сеансовый уровень).

Сеансовый уровень (Session Layer)

В ГОСТ ЭМВОС: Взаимосвязь открытых систем | ГОСТ Р ИСО/МЭК 7498-1-99 дается следующее определение сеансового уровня:

7.3 Сеансовый уровень

7.3.1 Определения

7.3.1.1 Административное управление полномочием — средство услуг сеансового уровня, которое позволяет взаимодействующим логическим объектам уровня представления явно управлять очередностью выполнения ими определенных функций управления.

7.3.1.2 Дуплексный режим — режим взаимодействия, при котором оба логических объекта уровня представления данных могут одновременно передавать и принимать нормальные данные.

7.3.1.3 Полудуплексный режим — режим взаимодействия, при котором в данном сеансе только одному из двух взаимодействующих логических объектов уровня представления данных позволено передавать нормальные данные.

7.3.1.4 Синхронизация соединения сеансового уровня — средство услуг сеансового уровня, позволяющее логическим объектам уровня представления определять и идентифицировать точки синхронизации, выполнять сброс соединения сеансового уровня в некоторое определенное состояние, а также согласовывать точку повторной синхронизации.

7.3.2 Назначение

7.3.2.1 Сеансовый уровень предназначен для обеспечения средств, необходимых взаимодействующим логическим объектам уровня представления данных, организации и синхронизации диалога и административного управления обменом данными между ними. С этой целью сеансовый уровень предоставляет услуги по установлению соединения сеансового уровня между двумя логическими объектами уровня представления данных, а также услуги по поддержанию упорядоченного обмена данными при взаимодействии и услуги по освобождению соединения в соответствии с установленной процедурой.

7.3.2.2 Только сеансовый уровень для обмена данными в режиме без установления соединения обеспечивает преобразование адресов транспортного уровня в адреса сеансового уровня.

7.3.2.3 Соединение сеансового уровня создается по запросу логического объекта уровня представления через ПДУ сеансового уровня. Соединение сеансового уровня существует до тех пор, пока оно не будет освобождено логическими объектами уровня представления или логическими объектами сеансового уровня.

7.3.2.4 Иницирующий логический объект уровня представления данных задает логический объект получателя уровня представления данных с помощью адреса на сеансовом уровне. Однако в общем случае между адресами на сеансовом и транспортном уровнях существует соответствие типа «несколько к одному». Это не подразумевает мультиплексирования соединений сеансового уровня в соединения транспортного уровня, а означает, что во время установления соединения сеансового уровня по запросу на его установление, поступающему по данному транспортному соединению, потенциальными получателями этого запроса являются несколько логических

объектов уровня представления. Однако при необходимости между адресами на сеансовом и транспортном уровнях может существовать соответствие типа «один к одному».

Фактически речь идет о том, что протокол сеансового уровня решает задачи установки соединения, управлением соединения, разрывом соединения.

Согласно странице в Википедии https://en.wikipedia.org/wiki/Session_layer, сеансовый уровень решает задачи.

- аутентификации;
- авторизации;
- восстановления сеанса (установки контрольных точек и восстановления).

Там же отмечается, что, хотя в модели TCP/IP и нет аналога сеансового уровня модели OSI/ISO, управление сеансом производится транспортными протоколами (TCP и SCTP) либо непосредственно прикладными протоколами (такими как QUIC, работающий поверх UDP).

На сеансовом уровне может работать межсетевой экран Netfilter/iptables в Linux.

Механизм определения состояний (state machine, connection tracking) — система трассировки соединений, важная часть netfilter, при помощи которой реализуется межсетевой экран на сеансовом уровне (**stateful firewall**). Система позволяет определить, к какому соединению или сеансу принадлежит пакет.

<https://ru.wikipedia.org/wiki/Netfilter>

В настоящее время это осуществляется модулем conntrack.

В Linux просмотреть атрибуты активных подключений можно в файле /proc/net/nf_conntrack (или /proc/net/ip_conntrack) в procfs.

К шлюзам сеансового уровня относится технология NAT. (В Linux NAT реализуется в составе Netfilter/Iptables.)

(См. <https://www.osp.ru/lan/1999/09/134421>)

Разберем механизм обеспечения безопасности TLS для прикладных приложений.

TLS на примере HTTPS, сертификаты

Изначально многие прикладные протоколы были небезопасными. Безопасность была добавлена потом с помощью протокола TLS, позволяющего обеспечить аутентификацию и шифрование. Изначально небезопасный HTTP, использующий порт 80/TCP, с помощью инкапсуляции в TLS «превратился» в протокол HTTPS, использующий порт 443/TCP.

Альтернативный способ — STARTLS, используемый в SMTP. Сначала открывается нешифрованное соединение, в котором производится обмен (приветствие сервера HELO или EHLO, отображение доступных расширений), после чего производится запрос STARTLS, и зашифрованный сеанс начинается заново (снова приветствие и т. д.).

Сложнее с FTP. Изначально это не зашифрованный протокол. Туннелировать с помощью SSH его сложно (из-за использования минимум двух потоков, одного для управления и другого/других для данных). Такой механизм получил название SSH+FTP. Кроме того, путаницы добавил протокол SFTP, не являющийся ни FTP, ни SSH+FTP, но расшифровывается как SSH File Transfer Protocol. Это отдельный протокол, являющийся частью SSH и использующий SSH как транспорт (и заодно механизм управления сеансом).

Оригинальный FTP с применением TLS может использоваться двумя способами: как FTPS (FTP/SSL), используя порты 990/TCP для управления соединением и 989/TCP для передачи данных, или с помощью STARTLS, используя традиционные 20 и 21 порты.

Путаницы добавляет и то, что современные FTP-клиенты (FAR, Total Commander, Filezilla, Core FTP) одновременно реализуют и SFTP клиент.

Внимание! Многие провайдеры до сих пор предлагают закидывать файлы на сервер по нешифрованному FTP. В таком виде данные не защищены от перехвата, подмены. Посмотрите через Wireshark, как передается логин и пароль при использовании нешифрованного FTP. Никогда не используйте незащищенный FTP для передачи файлов на сервер.

Но есть ситуации, когда нешифрованный FTP допустим — например, для размещения архивов/репозиториях для скачивания. Впрочем, механизмы аутентификации (например, в репозиториях Debian/Ubuntu) все равно используются — как внешний по отношению к протоколу инструмент.

В принципе, то же касается и HTTP. Если веб-сервер содержит открытую публичную информацию и не получает от пользователя файлы и/или пароли, он может использовать незащищенный HTTP. Если же сервер требует от пользователя ввода конфиденциальных данных (в т. ч. пароля) или работы с файлами, уже требуется шифрование. То есть нужно использовать HTTPS. Впрочем, поисковые машины вроде Google могут пессимизировать в поисковой выдаче HTTP-серверы в пользу HTTPS-серверов независимо от того, требуется ли на сервере ввод данных или он только предоставляет файлы для публичного скачивания.

Протокол TLS предоставляет услуги шифрования и аутентификации. Шифрование бывает симметричное и асимметричное. В симметричном шифровании один и тот же ключ используется и для шифрования, и для расшифровки.

Пусть задано следующее.

- f — функция шифрования;

- ключ — симметричный ключ;
- исходный_текст — открытый, незашифрованный текст;
- шифровка — зашифрованный текст.

Тогда получается.

- $f(\text{Ключ, Текст}) = \text{Шифровка}$.
- $f(\text{Ключ, Шифровка}) = \text{Текст}$.

Простейшим примером симметричного шифрования является операция XOR (исключающее сложение по модулю 2).

a	b	$a \oplus b$
0	0	0
0	1	1
1	0	1
1	1	0

Для A и B (A — исходное сообщение, B — ключ, Z — шифрованное сообщение):

$$Z = A \oplus B,$$

$$A = Z \oplus B.$$

Это легко проверить по таблице или вывести из алгебраических свойство операции XOR:

$$Z \oplus B = (A \oplus B) \oplus B = A$$

Сама по себе операция XOR не годится для шифрования в силу простоты, но имеются более сложные алгоритмы шифрования с ее использованием.

В качестве примеров алгоритмов симметричного шифрования можно указать DES (data encryption standard) и AES (advanced encryption standard).

Одним из первых алгоритмов, использующих закрытые ключи, стал **алгоритм Диффи — Хеллмана**, позволяющий двум абонентам получить общий секретный симметричный ключ при работе по не защищенному от прослушивания каналу. Оба абонента используют свои закрытые ключи и, обмениваясь результатами арифметических операций, получают общий симметричный ключ. Недостаток алгоритма заключался в том, что он работал в случае, если канал был прослушиваемый, но данные не изменялись. В случае, если злоумышленник мог подменить данные, достоинства алгоритма сводились на нет. Такой тип атаки называется MITM (Man In The Middle). Таковым злоумышленником может оказаться провайдер, прокси-сервер, сниффер, любой человек, который

имеет доступ к каналу в силу должностных полномочий или несанкционированно получивший к нему доступ (и установивший соответствующее оборудование).

Задача работы через канал, в котором данные могут быть не только прослушаны, но и подменены, была решена алгоритмом асимметричного шифрования **RSA**, названного по имени авторов (Rivest, Shamir и Adleman).

RSA — алгоритм асимметричного шифрования. Вместо общего симметричного ключа используется пара асимметричных ключей, публичный и приватный. То, что зашифровано одним, может быть расшифровано парным ключом. При этом публичный ключ распространяется корреспондентам, а приватный остается только у выпустившего ключ. Публикация приватного ключа — ошибка, требующая перевыпуск ключа и сертификатов.

Асимметричные алгоритмы могут решить одновременно только одну задачу из двух:

1. Шифрование трафика. Если шифруется публичным ключом, расшифруется приватным. Зашифровать может любой, а прочитать только владелец публичного ключа. Трафик недоступен для прослушивания, но подтвердить подлинность отправителя невозможно.
2. Аутентификация — проверка подлинности абонента. Отправитель шифрует приватным ключом, а получатели расшифровывают публичным. Если они смогли расшифровать, значит, отправить мог только отправитель, чей публичный ключ у них есть, но расшифровать такой трафик может любой.

Так как на практике требуется решение обеих задач, применяется третья сторона — удостоверяющий центр. Публичные ключи удостоверяющих центров импортируются в браузеры производителями браузеров, и удостоверяющая способность таких центров (CA) не вызывает сомнений.

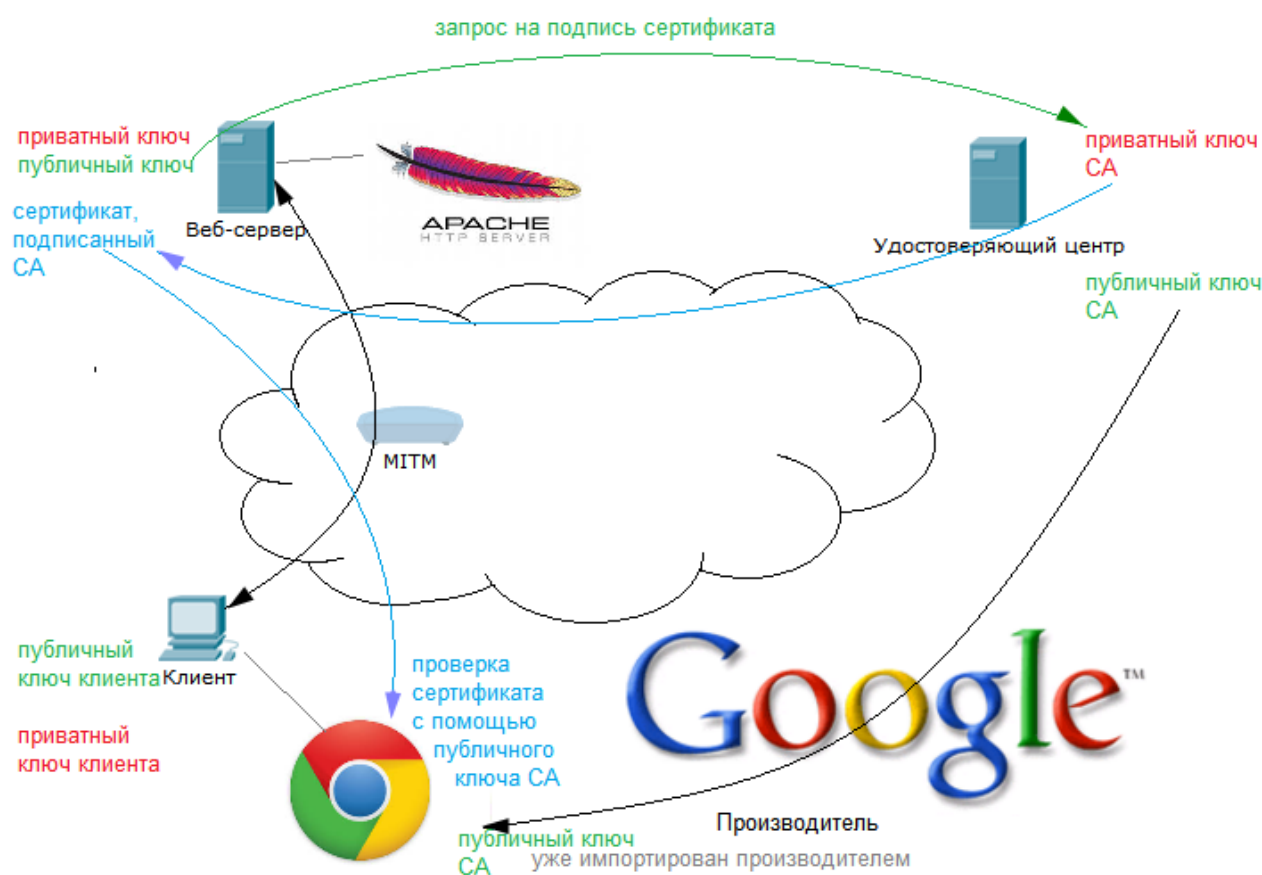
Владелец же сайта выпускает приватный и публичный ключи и подписывает публичный ключ у удостоверяющего центра. Для этого центру сертификации передается запрос на подпись по стандарту X.509, а в ответ приходит сертификат. Так браузер может проверить, является ли сайт тем, за что себя выдает.

Сертификат — публичный ключ и дополнительная информация, хеш от которой зашифрован приватным ключом удостоверяющего центра. Подпись ключа — услуга, предоставляемая удостоверяющими центрами. Можно подписать сайт самостоятельно сгенерированным удостоверяющим центром, но браузеры будут выводить сообщение о том, что валидность сайта проверить невозможно. Но для тестирования вариант самоподписанного сертификата подойдет.

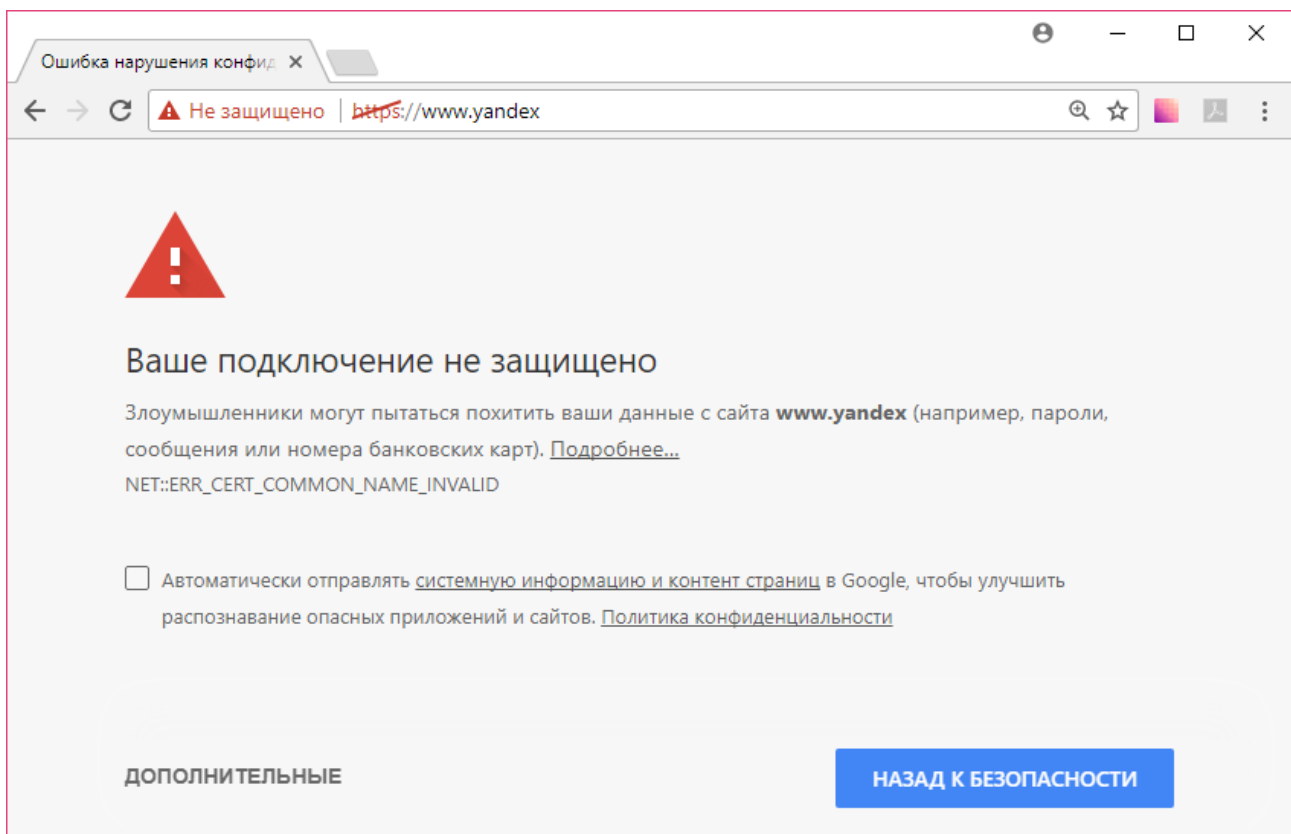
TLS — новый протокол безопасности, основанный на SSL (тем не менее TLSv1 тоже уже успел устареть, это надо учитывать). Его идея состоит в использовании алгоритма RSA, сложность взлома которого определяется алгебраической сложностью задачи нахождения простых сомножителей натурального числа. (На практике в TLS применяется алгоритм RSA для аутентификации и алгоритм Диффи — Хеллмана для получения общего симметричного ключа, используемого для шифрования.)

Интересно, что с протоколом TLS произошло что и с HTTP (добавление заголовка Host). Изначально не было возможности указать в открытом виде домен, для которого выдавался сертификат, поэтому единственным способом использовать несколько доменов было получение для каждого отдельного IP-адреса. Затем был добавлен механизм SNI (Server Name Identification), позволяющий передать доменное имя в открытом виде.

На практике часто, говоря об SSL, подразумевают TLS. Тем не менее SSL v3 и TLS v1 уже признаны небезопасными, это надо иметь в виду при настройке сервисов, использующих шифрование.



Если сертификат будет подменен или из-за ошибки конфигурирования не будет соответствовать домену, браузер выдаст ошибку.



Такая же ошибка будет выводиться и для самоподписанных (вместо удостоверяющего центра) сертификатов.

К https-серверу можно подключиться и писать сырые HTTP-запросы аналогично тому, как мы с вами делали с Telnet. Для этого понадобится утилита openssl в Linux.

```
$ openssl s_client -connect vk.com:443
```

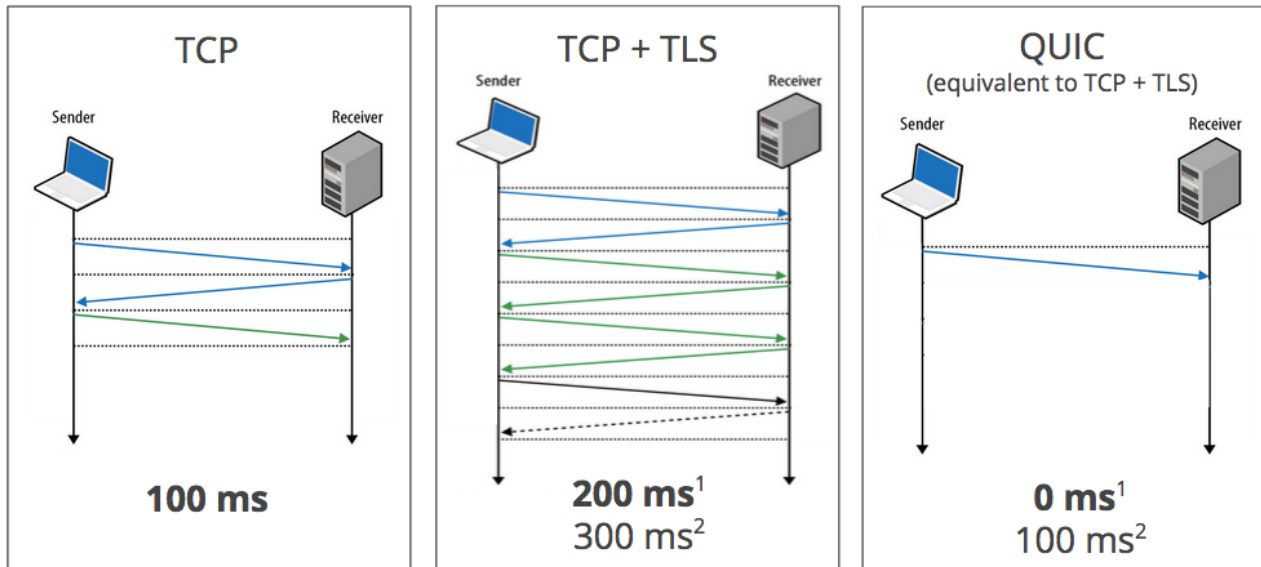
После получения информации о сессии и сертификате можно писать традиционный GET.

QUIC как альтернатива HTTP+TLS+TCP

Протокол HTTP появился в 1991 году в CERN (вместе с HTML и URL). В 2012 году в Google стали разрабатывать протокол SPDY, но прекратили его разработку, отдав наработки в HTTP/2. Это уже двоичный протокол, обладающий рядом новых возможностей. Интересно, что Google продолжила разработку альтернативы HTTP. Новый протокол получил название QUIC. Он использует на верхнем уровне HTTP/2 API-прослойку и сам решает задачи управления соединением, шифрования и сжатия, заменяя TLS+TCP.

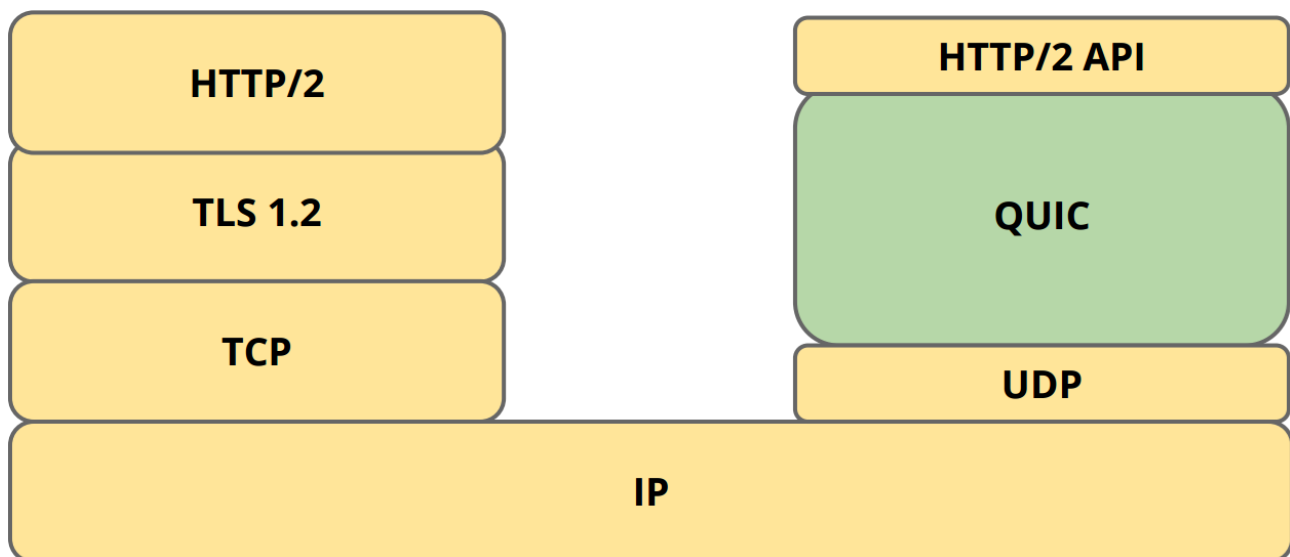
На рисунке ниже видно, что TLS в установлении зашифрованного соединения подобен TCP, а при их совместном использовании получается значительный overhead. Протокол QUIC («быстрый») использует UDP-дейтаграммы и избавлен от необходимости долгих «рукопожатий», что позволяет быстро открыть соединение.

Zero RTT Connection Establishment



1. Repeat connection
2. Never talked to server before

<https://www.opennet.ru/opennews/art.shtml?num=42063>



<https://habrahabr.ru/company/infopulse/blog/315172/>

Протокол QUIC можно наблюдать в Wireshark, подключаясь с помощью Google Chrome к сервисам Google (например, к YouTube).

Есть мнение, что наработки протокола QUIC могут послужить основой для дальнейшей разработки протокола TCP.

Примеры протоколов прикладного уровня

Прикладной уровень решает задачи передачи данных между приложениями через сеть. В стеке TCP/IP прикладные протоколы непосредственно реализуют сами приложения, получая доступ к услугам транспортного и сетевого уровня с использованием механизма сокетов.

Основные задачи, решаемые на прикладном уровне:

- передача запросов от клиента к серверу;
- передача ответов от сервера к клиенту;
- шифрование данных и идентификация абонента;
- обеспечение работы сетевых служб.

Прикладные протоколы могут использовать один из транспортных протоколов (TCP для гарантированной, UDP для негарантированной доставки) или даже оба в зависимости от задач (DNS использует UDP для DNS-запросов, TCP — для передачи файлов зон и DNSSEC). Прикладные протоколы могут как использовать стандартные средства для управления сеансом (TCP) и реализации безопасности (TLS), так и реализовать их самостоятельно (протокол QUIC, использующий HTTP/2 API на верхнем уровне и самостоятельно реализующий управление соединением и безопасность, используя UDP в качестве транспорта).

Многие прикладные протоколы изначально небезопасны. Ряд небезопасных протоколов был приспособлен для работы в безопасном режиме благодаря механизму TLS (HTTPS, FTPS), STARTTLS (SMTP, XMPP) либо использованию самостоятельных решений (SSH и его надстройки).

Мы разберем в качестве примера устройство протоколов SMTP и HTTP.

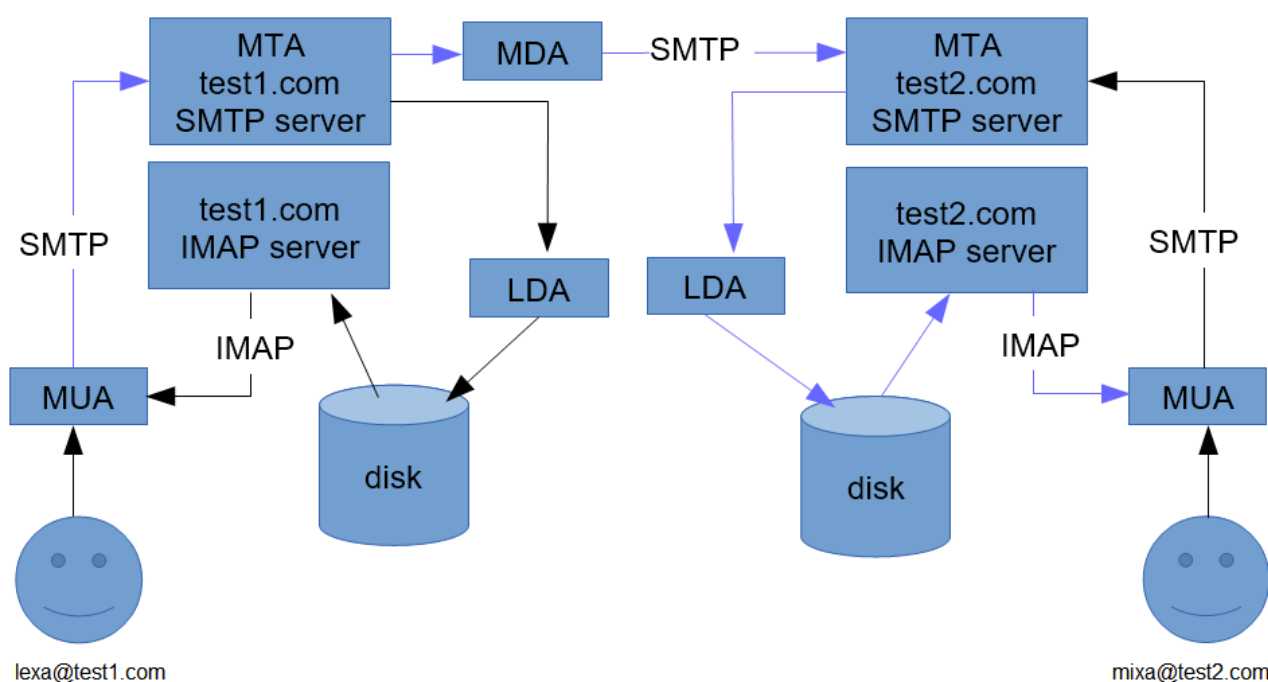
Сервисы электронной почты: SMTP/POP3/IMAP

Процесс доставки электронной почты

В процессе доставки электронное сообщение обычно проходит через несколько узлов электронной почты и ряд программ, выполняющих разные функции. Опишем главных участников процесса доставки в общепринятой для электронной почты терминологии.

- MTA (Mail Transfer Agent) — программа, которая принимает, маршрутизирует и отправляет другим серверам электронную почту. Как правило, для этого используется протокол SMTP. В качестве MTA в Linux используют такие программы, как postfix, exim. Исторически использовался sendmail.

- MDA (Mail Delivery Agent) — программа, которую вызывает MTA для действительной доставки сообщения, например по протоколу SMTP, на другую систему.
- LDA (Local Delivery Agent) — разновидность MDA для локальной доставки писем. LDA помещает письмо в почтовый ящик пользователя. Иногда MTA и LDA не разделяют, называя все программы доставки MDA. В качестве примера LDA может привести Dovecot, который умеет принимать от MTA письма по протоколу SMTP, сохранять на диск для последующей отдачи пользователю по протоколу PO3/IMAP4.
- MUA (Mail User Agent) — почтовая программа пользователя, например Outlook или Mozilla Thunderbird.



Рассмотрим процесс доставки сообщения, которое пользователь `lexa@test1.com` отправил пользователю `mixa@test2.com`.

1. Пользователь `lexa@test1.com` в своей почтовой программе нажимает кнопку «Отправить», после чего почтовая программа (MUA) соединяется с локальным почтовым сервером `test1.com` по протоколу SMTP и выполняет передачу сообщения. Далее обработка сообщения выполняется MTA на сервере `test1.com`.
2. MTA на `test1.com` сверяется с локальной конфигурацией и выясняет, что он не принимает локально письма для домена `test2.com`.
3. Поэтому MTA на `test1.com` выполняет запрос к DNS и получает MX-записи для домена `test2.com`. Почтовым сервером с наивысшим приоритетом для домена `test2.com` является сервер `test2.com`.
4. MTA на `test1.com` вызывает SMTP MDA для доставки сообщения на сервер `test2.com` по протоколу SMTP.

5. MTA на test2.com, получив сообщение, сверяется с локальной конфигурацией и выясняет, что он принимает локально письма для домена test2.com.
6. Пользователь misha@test2.com в своей почтовой программе нажимает кнопку «Получить почту», после чего его почтовая программа (MUA) обращается к серверу test2.com по протоколу IMAP или POP3 и загружает доставленное ему сообщение от пользователя lexa@test1.com.

Удобство использования MX-записей в том, что веб-сервер, выдающий страницу по адресу test.com, и почтовый сервер, принимающий письма на имейл email@test.com, могут не совпадать. Более того, можно настроить так, чтобы почту для домена обслуживали Яндекс.Почта или Gmail.

Протокол SMTP

Протокол SMTP, как мы уже видели, используется при передаче сообщений от MUA к MTA или между MTA на различных узлах электронной почты. SMTP представляет собой текстовый протокол, когда клиент с помощью определенного набора команд задает параметры отправляемого сообщения, а сервер подтверждает его прием или выдает сообщения об ошибках. Ниже представлен пример SMTP-сессии.

```
root@ubutubu:~# telnet localhost 25 Trying ::1...
Trying 127.0.0.1...
Connected to localhost. Escape character is '^]'.
220 ubutubu.example.com ESMTP Sendmail 8.14.4/8.14.4/Debian-4.1ubuntu1; Thu, 10
Jul 2014 14:06:50
+0400; (No UCE/UBE) logging access from: localhost(OK)-localhost [127.0.0.1]
ehlo localhost
250-ubutubu.example.com Hello localhost [127.0.0.1], pleased to meet you
250-ENHANCEDSTATUSCODES
250-PIPELINING
250-EXPN 250-VERB
250-8BITMIME
250-SIZE 250-DSN
250-ETRN
250-AUTH DIGEST-MD5 CRAM-MD5
250-DELIVERBY 250 HELP
mail from: <oga@ubutubu.example.com>
250 2.1.0 <oga@ubutubu.example.com>... Sender ok
vkrpt to: <oavdeev@prog-school.ru>
250 2.1.5 <oavdeev@prog-school.ru>... Recipient ok data
354 Enter mail, end with "." on a line by itself
Subject: Test test test mail
Test
1 2 3 .
250 2.0.0 s6AA6oi5004854 Message accepted for delivery quit
221 2.0.0 ubutubu.example.com closing connection
Connection closed by foreign host.
```

В данном примере соединение с SMTP-сервером было установлено с помощью команды telnet, и все команды вводились вручную. Первой командой SMTP-клиент представляется серверу: EHLO localhost. Сервер может выполнить на этом этапе ряд проверок: проверить корректность доменного

имени после EHLO, проверить, имеет ли имя DNS-записи типа A или MX и так далее. Уже на этом этапе можно ограничить часть спама, идущего через почтовый сервер. В нашем случае проверки прошли успешно, сервер представился сам и выдал список поддерживаемых расширенных команд протокола ESMTP. Далее клиент с помощью команды MAIL FROM сообщает серверу email, который будет использоваться сервером для возврата сообщения в случае неудачной доставки. Команда RCPT TO устанавливает получателя данного сообщения, команда может быть использована повторно, если письмо имеет несколько получателей (по одной команде на каждого получателя). Сообщения доставляются именно по значению RCPT TO, а не по полю To: служебного заголовка письма, в котором может быть вообще что угодно. После ввода команд MAIL FROM и RCPT TO сервер выполняет проверку корректности адресов и подтверждает проверку сообщениями Recipient ok и Sender ok. На данном этапе сервер проверяет возможность доставки сообщения с данным отправителем данному получателю. Как правило, один из них должен принадлежать почтовому домену, обслуживаемому данным почтовым сервером, или иметь IP в доверенной сети. Если это не так, возможно, кто-то пытается использовать почтовый сервер для отправки спама.

Это обеспечивается запросом PTR-записи для обратного разрешения домена для IP-адреса отправителя, использования расширения SPF (Sender Policy Framework), позволяющего указать для домена, от имени которого отправляется письмо, список серверов, имеющих право отправлять почту. SPF также базируется на системе DNS, используя записи вида TXT (позволяющие сопоставить доменному имени произвольное текстовое описание).

Пример SPF-записи, которая должна быть отправлена при использовании Яндекс.почты для домена на DNS-сервер, обслуживающий его зону:

```
@      TXT      v=spf1 redirect=_spf.yandex.ru
```

Пример, когда требуется указать еще отдельные серверы, которые тоже могут отправлять письма:

```
@      TXT      v=spf1 ip4:1.2.3.4 ip4:1.2.3.5 ip4:1.2.3.6 include:_spf.yandex.ru  
~all
```

При проверке писем сервер также может проверить IP-адрес отправителя в черных списках.

Одной из реализаций черных списков является DNSBL, который представляет собой DNS-сервер. Чтобы проверить, заблокирован ли IP-адрес каким-либо DNSBL-списком, надо указать проверяемый IP в нотации DNS PTR (в обратном порядке октетов) и добавить имя домена DNSBL-сервера. Если ответ будет получен, то данный адрес заблокирован (находится в списке).

```
$ host -tA 71.60.206.220.bl.spamcop.net  
220.206.60.71.bl.spamcop.net has address 127.0.0.2
```

Полученный IP-адрес может оказаться любым, важен лишь факт его наличия (отсутствия) в ответе на запрос. Именно поэтому сам IP можно использовать для описания, скажем, типа источника, по которому искомый адрес был добавлен в список. К примеру, 127.0.0.1 — открытые релеи, 127.0.0.2 — источники portscan'ов и т. п. По той же причине полезно использовать **host** с опцией **-t any**, в ответ на которую вы можете получить дополнительный комментарий в поле типа «текст» (TXT RR).

```
$ host -t any 116.47.136.151.vote.drbl.sandy.ru
151.136.47.116.vote.drbl.sandy.ru descriptive text "070715:Spam in progress"
151.136.47.116.vote.drbl.sandy.ru has address 127.0.0.2
```

Пример адреса, которого нет в DNSBL-списке:

```
$ host -tA 72.205.229.181.bl.spamcop.net
Host 181.229.205.72.bl.spamcop.net not found: 3(NXDOMAIN)
```

Если почтовый сервер допускает отправку через него чужой почты, такой сервер называют **open relay**, и обычно через некоторое время его адрес попадает в списки блокировок, и другие серверы перестают принимать от него почту.

После того, как со стороны клиента указана команда DATA, происходит непосредственно ввод сообщения. Сообщение состоит из двух частей: служебных заголовков и тела письма. В нашем примере из заголовков указан только Subject: с темой письма. Если как в нашем случае поля служебных заголовков From: и To: не указаны, MTA самостоятельно формирует эти заголовки на основе введенных ранее клиентом команд MAIL FROM и RCPT TO (как правило, To: в этом случае не показывается в письме).

Кроме того в служебных заголовках отмечается весь маршрут, который проходит сообщение. На каждом узле, через который проходит сообщение, добавляется заголовок Received, в котором указывается, с какого IP поступило сообщение, каким сервером оно было принято, дата/время приема и получатель сообщения. Служебные сообщения отделяются от тела сообщения пустой строкой.

Традиционно считается, что сообщение может проходить через узлы, которые поддерживают только семибитную кодировку, поэтому тело и заголовки, в которых встречается кириллица, перед отправкой кодируются в семибитной кодировке base64. Эта кодировка прозрачна для почтовых клиентов, поэтому пользователи сразу получают перекодированный текст и не замечают процесса перекодировки. Раньше вместо base64 для бинарных вложений в письмах использовалась программа uuencode, переводившая бинарные вложения в текст; на стороне получателя программа uudecode восстанавливала исходный файл.

Признаком окончания сообщения служит строка, состоящая из одной точки. Встретив такую строчку, MTA говорит, что сообщение принято для дальнейшей передачи. Затем для окончания сессии клиент использует команду quit.

Дополнительно необходимо заметить, что в настоящее время используются расширения SMTP протокола, которые поддерживают аутентификацию (команда AUTH LOGIN) и шифрование сессии (команда STARTTLS). Эти возможности обычно используют, чтобы защитить соединение между клиентом и сервером от перехвата злоумышленником.

Пример письма, отправленного через sendmail из виртуальной машины описанным выше способом:

```
Received: from mxfront8j.mail.yandex.net ([127.0.0.1])
  by mxfront8j.mail.yandex.net with LMTP id A9txhLMh
  for <siblis@yandex.ua>; Mon, 24 Apr 2017 21:47:00 +0300
Received: from 37.70.32.95.dsl-dynamic.vsi.ru (37.70.32.95.dsl-dynamic.vsi.ru
[95.32.70.37])
  by mxfront8j.mail.yandex.net (nsmtp/Yandex) with ESMTPS id
u58pcgfyYV-kxd4gq85;
  Mon, 24 Apr 2017 21:47:00 +0300
  (using TLSv1.2 with cipher ECDHE-RSA-AES128-GCM-SHA256 (128/128 bits))
  (Client certificate not present)
X-Yandex-Front: mxfront8j.mail.yandex.net
X-Yandex-TimeMark: 1493059620
To: undisclosed-recipients;;
X-Yandex-Spam: 1
Received: from localhost (localhost [127.0.0.1])
  by ubuntu (8.14.4/8.14.4/Debian-4.1ubuntu1) with ESMTTP id v3OIbGSO023496
  for <siblis@yandex.ua>; Mon, 24 Apr 2017 11:46:17 -0700
Date: Mon, 24 Apr 2017 11:46:17 -0700
From: medvedev@kremlin.ru
Message-Id: <201704241846.v3OIbGSO023496@ubuntu>
Subject: Good weather
Return-Path: medvedev@kremlin.ru
X-Yandex-Forward: 6cd83209625f7ffcf9c2f47a0f822f8c

Something
something mail
```

Письмо попало в папку Входящие. Но уже второе письмо с большой долей вероятности уйдет в спам.

Один из заголовков, который можно наблюдать в таком случае:

```
X-Yandex-Spam: 4
```

Самостоятельно ответьте на вопрос, почему.

Еще один инструмент для подтверждения подписей отправителя — DKIM.

Технология DomainKeys Identified Mail (DKIM) объединяет несколько существующих методов антифишинга и антиспама с целью повышения качества классификации и идентификации легитимной электронной почты. Вместо традиционного IP-адреса для определения отправителя сообщения DKIM добавляет в него цифровую подпись, связанную с именем домена организации. Подпись автоматически проверяется на стороне получателя, после чего для определения репутации отправителя применяются «белые списки» и «черные списки».

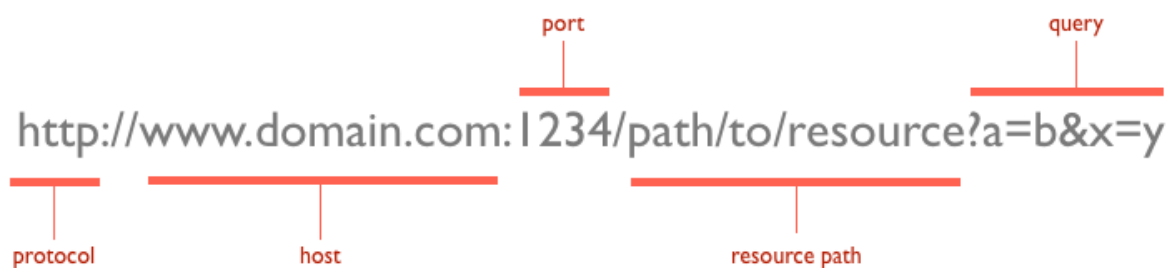
В технологии DomainKeys для аутентификации отправителей используются доменные имена. DomainKeys использует существующую систему доменных имен (DNS) для передачи открытых ключей шифрования.

Основные понятия HTTP

HyperText Transfer Protocol (протокол передачи гипертекста) — протокол прикладного уровня, осуществляющий передачу структурированных данных в формате HTML. Также протокол позволяет передавать произвольные данные (документы/картинки/видео/музыку).

Технология является клиент-серверной и использует:

- веб-сервер;
- браузер/клиентское приложение.



Методы HTTP

GET: получить доступ к существующему ресурсу. В URL перечислена вся необходимая информация, чтобы сервер смог найти и вернуть в качестве ответа искомый ресурс.

POST: используется для создания нового ресурса. POST-запрос обычно содержит в себе всю нужную информацию для создания нового ресурса.

PUT: обновить текущий ресурс. PUT-запрос содержит обновляемые данные.

DELETE: служит для удаления существующего ресурса.

Эти методы самые популярные и чаще всего используются различными инструментами и фреймворками. В некоторых случаях PUT- и DELETE-запросы заменяются POST-запросом, в содержании которого указано действие, которое нужно совершить с ресурсом: создать, обновить или удалить.

Также HTTP поддерживает и другие методы.

HEAD: аналогичен GET. Разница в том, что при данном виде запроса не передается сообщение. Сервер получает только заголовки. Используется, к примеру, для того, чтобы определить, был ли изменен ресурс.

TRACE: во время передачи запрос проходит через множество точек доступа и прокси серверов, каждый из которых вносит свою информацию: IP, DNS. С помощью этого метода можно увидеть всю промежуточную информацию.

OPTIONS: используется для определения возможностей сервера, его параметров и конфигурации для конкретного ресурса.

Коды состояния

В ответ на запрос от клиента сервер отправляет ответ, который содержит в том числе и код состояния. Коды состояния представляют собой десятичные трехзначные числа и в зависимости от первой цифры делятся на пять групп.

1xx: информационные сообщения.

Набор этих кодов был введен в HTTP/1.1. Сервер может отправить запрос вида: Expect: 100-continue, что означает, что клиент еще отправляет оставшуюся часть запроса. Клиенты, работающие с HTTP/1.0, игнорируют данные заголовки.

2xx: сообщения об успехе.

Если клиент получил код из серии 2xx, то запрос успешно выполнен. При GET-запросе сервер отправляет ответ в теле сообщения. Самый распространенный вариант — это 200 OK. Также существуют и другие возможные ответы:

- 201 Created: (в ответ на PUT и POST) запрашиваемый ресурс создан. Заголовок Location должен вернуть адрес созданного ресурса.
- 202 Accepted: запрос принят, но может не содержать ресурс в ответе. Это полезно для асинхронных запросов на стороне сервера. Сервер определяет, отправить ресурс или нет.
- 204 No Content: в теле ответа нет сообщения.
- 205 Reset Content: указание серверу о сбросе представления документа.
- 206 Partial Content: ответ содержит только часть контента. В дополнительных заголовках определяется общая длина контента и другая информация.

3xx: перенаправление.

Своеобразное сообщение клиенту о необходимости совершить еще одно действие. Самый распространенный вариант применения: перенаправить клиент на другой адрес.

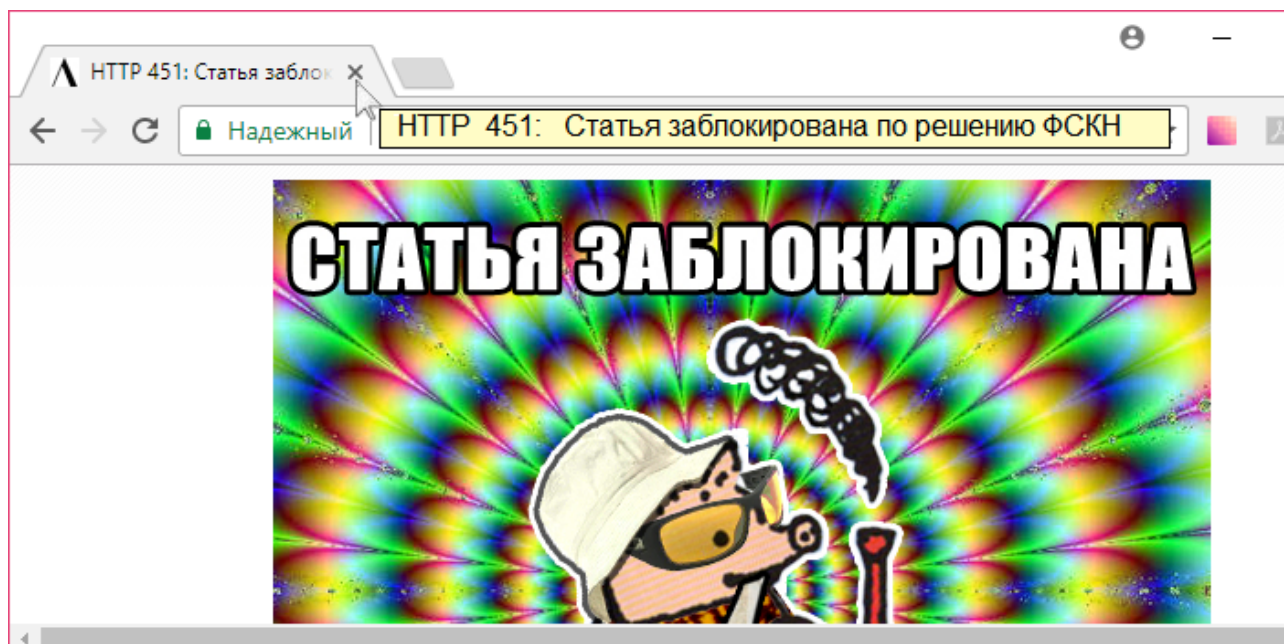
- 301 Moved Permanently: ресурс теперь можно найти по другому URL-адресу.
- 303 See Other: ресурс временно можно найти по другому URL-адресу. Заголовок Location содержит временный URL.

- 304 Not Modified: сервер определяет, что ресурс не был изменен и клиенту нужно задействовать закешированную версию ответа. Для проверки идентичности информации используется ETag (хеш сущности — Entity Tag).

4xx: клиентские ошибки.

Данный класс сообщений используется сервером, если он решил, что запрос был отправлен с ошибкой. Наиболее распространенный код — 404 Not Found. Это означает, что ресурс не найден на сервере. Другие возможные коды:

- 400 Bad Request: запрос был сформирован неверно.
- 401 Unauthorized: для совершения запроса нужна аутентификация. Информация передается через заголовок Authorization.
- 403 Forbidden: сервер не открыл доступ к ресурсу. (Пользователь не ввел верные имя пользователя и пароль в случае base- или digest-авторизации, либо ресурс доступен только с определенных IP-адресов — например, локально).
- 405 Method Not Allowed: для того, чтобы получить доступ к ресурсу был задействован неверный HTTP-метод.
- 409 Conflict: сервер не может до конца обработать запрос, так как пытается изменить более новую версию ресурса. Это часто происходит при PUT-запросах.
- 451 Unavailable For Legal Reasons — ресурс заблокирован Роскомнадзором. Название намекает на роман Рэя Брэдбери «451 градус по Фаренгейту» (там пожарные книги сжигали).



5xx: ошибки сервера.

Ряд кодов, которые используются для определения ошибки сервера при обработке запроса. Самый распространенный — 500 Internal Server Error. Возможная причина — ошибка в файле .htaccess.

Другие варианты:

- 501 Not Implemented: сервер не поддерживает запрашиваемую функциональность.
- 502 Bad Gateway: вы обратились к реверс-прокси (например, nginx; обычно такие машины называют фронтендом), но он так и не получил данные от бэкенда (например, Apache; истек таймаут).
- 503 Service Unavailable: это может случиться, если на сервере произошла ошибка или он перегружен. Обычно в этом случае сервер не отвечает, а время, данное на ответ, истекает.

Заголовки HTTP

Заголовки HTTP (англ. HTTP Headers) — это строки в HTTP-сообщении, содержащие разделенную двоеточием пару «имя — значение». Заголовки должны отделяться от тела сообщения хотя бы одной пустой строкой.

Все заголовки разделяются на четыре основных группы:

- General Headers (рус. основные заголовки) — должны включаться в любое сообщение клиента и сервера;
- Request Headers (рус. заголовки запроса) — используются только в запросах клиента;
- Response Headers (рус. заголовки ответа) — только для ответов от сервера;
- Entity Headers (рус. заголовки сущности) — сопровождают каждую сущность сообщения.

Именно в таком порядке рекомендуется посылать заголовки получателю.

Заголовки в HTML

Язык разметки HTML позволяет задавать необходимые значения заголовков HTTP внутри `<head>` с помощью тега `<meta>`. При этом название заголовка указывается в атрибуте `http-equiv`, а значение — в `content`. Почти всегда выставляется значение заголовка `Content-Type` с указанием кодировки, чтобы избежать проблем с отображением текста браузером.

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf8">
<meta http-equiv="Content-Language" content="ru">
```

User Agent

User Agent — это клиентское приложение, использующее определенный сетевой протокол. Термин обычно используется для приложений, осуществляющих доступ к веб-сайтам, таких как браузеры, поисковые роботы (и другие «пауки»), а также для устройств, таких как мобильные телефоны.

Веб-сайты для мобильных телефонов часто вынуждены жестко полагаться на определение User-Agent, так как браузеры на разных мобильных телефонах слишком различны. Поэтому

мобильные веб-порталы обычно генерируют разные страницы в зависимости от модели мобильного телефона. Эти различия могут быть как небольшими (изменение размера изображений специально для меньших экранов), так и весьма существенными (формат WML вместо XHTML).

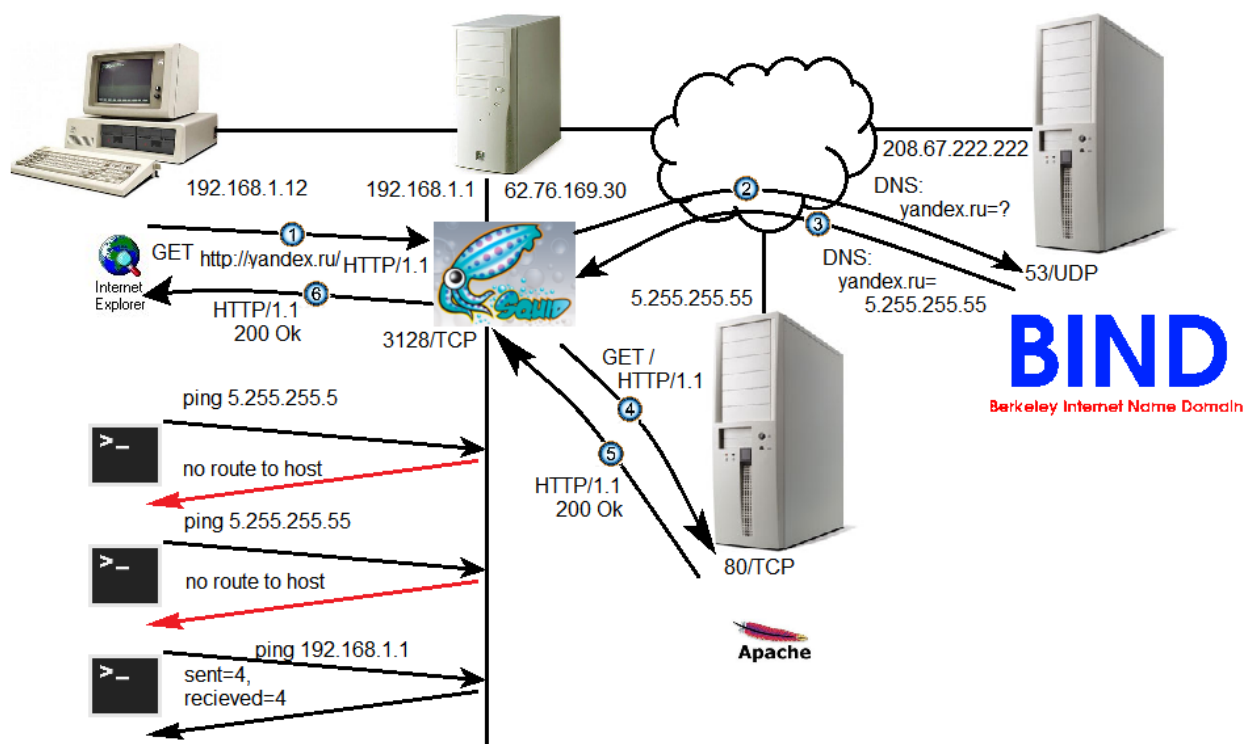
Строка User-agent также используется веб-мастерами для предотвращения индексирования «поисковыми пауками» некоторых страниц сайта, например, когда индексирование определенных страниц не имеет смысла или конкретный «паук» создает большую нагрузку на сервер. Веб-мастер может использовать специальный файл robots.txt для рекомендаций «пауку» или просто настроить веб-сайт так, чтобы тот не отдавал «пауку» эти страницы.

HTTP-прокси

Кроме веб-серверов и user-agent'ов существует еще класс программ, которые могут выступать и в качестве клиентов, и в качестве серверов. Речь идет о HTTP-прокси.

Технология HTTP-прокси старше, чем NAT. Она позволяла предоставить доступ в Интернет узлам, у которых есть сеть, но нет доступа в Интернет. Нет пингов, не работает FTP и «асечка», а странички в браузере открываются. Как же так? В настройках TCP/IP таких узлов не было адреса шлюза по умолчанию (сейчас Windows определяет такие подключения как «Без доступа в Интернет», но вот в настройках браузера был пункт «Прокси-сервер», в котором были указаны IP-адрес, номер порта, логин и пароль. IP-адрес использовался локальный, а сам компьютер, на котором был установлен прокси-сервер, имел доступ и в Интернет, и в локальную сеть. Но маршрутизация между ними не была включена).

Когда клиент хотел получить доступ в Интернет, он не обращался к удаленной машине, а устанавливал TCP-сессию на адрес прокси-сервера и у него просил все то, что мог бы попросить у настоящего сервера, будь у него подключение к Интернету. Прокси-сервер получал запрос и от своего имени обращался к веб-серверу, а получив ответ, возвращал его клиенту. Страницу он мог закешировать и, если страница не менялась, отдать ее еще потом несколько раз разным клиентам.



В случае HTTP-прокси веб-сервер устанавливает соединение (1) на адрес прокси-сервера (192.168.1.1:3128) и запрашивает ресурс yandex.ru. Как он это делает? В строке запроса GET для прокси будет находиться не адрес ресурса, а полностью URL:

```
GET http://yandex.ru/ HTTP/1.1
```

Если у прокси этот ресурс уже есть в кеше, он может сразу отдать его. Если нет, то, скорее всего, прокси запросит значение доменного имени yandex.ru (2) у DNS-сервера (208.67.222.222). DNS-сервер (3) ответит, что yandex.ru это 5.255.255.55. Прокси-сервер установит соединение на 5.255.255.55:80 и сделает HTTP-запрос (4):

```
GET / HTTP/1.1
```

Веб-сервер отправить веб-содержимое на адрес 62.76.169.30 (5).

Прокси-сервер может закешировать страницу, а также передаст уже в рамках сессии установленной на 192.168.1.1, с 192.168.1.12 в адрес клиента 192.168.1.12 веб-страницу.

Как видно, проху — это практически Man In The Middle.

А как же работает прокси, если работа идет с HTTPS? В этом случае клиент обращается к серверу с просьбой установить соединение с неким ресурсом. Прокси-сервер создаст тоннель и внутрь уже заглянуть не сможет. Для этого есть метод CONNECT:

```
CONNECT http://yandex.ru/ HTTP/1.1
```

А уже внутри сам клиент спросит:

```
GET / HTTP/1.1
```

Впрочем, с точки зрения клиента, TCP-соединение все равно установлено на адрес 192.168.1.1.

Но любые попытки пробраться в Интернет, кроме HTTP (и HTTP напрямую), успеха иметь не будут. Нет маршрута на сетевом уровне.

Когда-то (до NAT) это был единственный способ выпустить пользователей компьютеров с локальными адресами в Интернет и ускорить загрузку ресурсов. Скорости были небольшими (а то и вообще dial-up), контент был статический, JavaScript использовался для рисования снежинок и листиков на странице. Сейчас кое-где прокси-серверы используются как способ контроля: Веб 2.0 существенно ограничивает возможности кеширования, так как контент динамичный, а благодаря асинхронным запросам еще и собирается на ходу. Кеширование таких данных смысла не имеет. Скорости тоже уже большие. Из полезных вещей остаются возможность аутентификации, логирования и возможность ограничить скорость. С другой стороны, требуется распределение нагрузки и кеширования на стороне веб-сервисов (такой прокси называется обратный прокси, или реверс-прокси).

Выделяют следующие виды прокси:

- непрозрачный прокси: браузер знает, что работает через прокси, запросы отправляются на прокси;
- прозрачный прокси: трафик завернут на соответствующую машину, клиент не знает, что работает через прокси;
- реверс-прокси (обратный прокси).

Пример ПО прокси — squid (может работать как непрозрачный, прозрачный (только для HTTP), обратный прокси), nginx (последний — скорее веб-сервер).

Непрозрачный прокси не сможет обрабатывать защищенный трафик (сертификаты как раз направлены на борьбу с атакой MITM). Теоретически возможна подмена сертификата, но браузер будет об этом информировать.

Также прокси-сервер может использоваться для отладки. В этом случае он будет запущен локально (127.0.0.1). Так, Charles proху (программа платная, но есть бесплатный пробный период, доступна для Windows и MAC) или Burp Suite позволяют анализировать HTTP-трафик и заголовки, подменять сертификаты. Это чем-то может быть похоже на Wireshark, но только на прикладном уровне для HTTP.

SPDY и HTTP/2

Есть протоколы, логически продолжающие развитие протокола HTTP (а кое-где и предлагающие ему альтернативу). Один из таких протоколов мы уже упоминали — это QUIC.

В 2015 была утверждена новая версия HTTP — HTTP/2. В настоящее время HTTP/2 уже поддерживается популярными веб-серверами — Apache и nginx. Первоначально он развивался как протокол SPDY от Google. Протокол SPDY (скоростной) — это реализованный Google в 2009 проект по ускорению работы HTTP. Для того, чтобы протокол можно было использовать, протокол должен поддерживаться как сервером, так и клиентом. Как правило, с браузером Google Chrome не возникает никаких проблем при работе по новым протоколам.



В отличие от текстового протокола HTTP, протокол HTTP/2 был построен как двоичный протокол. По сравнению с HTTP/1.1 были изменены способы разбиения данных на фрагменты и их передачи между сервером и клиентом.

В HTTP/2 сервер имеет право послать то содержимое, которое еще не было запрошено клиентом. Это позволит серверу сразу выслать дополнительные файлы, которые потребуются браузеру для отображения страниц без необходимости анализа браузером основной страницы и запрашивания необходимых дополнений. Имеется механизм push (в этом плане HTTP/2 конкурирует с другим протоколом, WebSocket, который также позволяет используя 80 или 443 порт отправлять клиенту сообщения без их запроса. Такой механизм может быть использован для реализации веб-чатов и push-уведомлений).

Также часть улучшений получена (в первом черновике HTTP/2, который представлял собой копию спецификации SPDY) за счет мультиплексирования запросов и ответов для преодоления проблемы «head-of-line blocking» протоколов HTTP 1; сжатия передаваемых заголовков и введения явной приоритизации запросов.

Примеры API

Интернет-инфраструктура во многом заточена под WWW. 80 и 443 порты закрыты в последнюю очередь на файрволлах, HTTP-трафик проходит через самое дремучее оборудование. Иногда HTTP даже используется для туннелирования не HTTP-трафика (rtmpt). Поэтому возникает мысль: а почему бы вместо разработки нового протокола не воспользоваться готовым? HTTP поддерживается

практически везде, работает через NAT и прокси, имеются механизмы аутентификации и шифрования благодаря использованию TLS. Поэтому часто для взаимодействия ПО (клиент-сервер и т. д.) используют HTTP в качестве транспорта. Такой подход нарушает последовательность вложения уровней модель OSI/ISO (но мы уже знаем, что эта последовательность никогда и не соблюдается), иногда с точки зрения OSI такие протоколы относят к сеансовому уровню. По большей части это протоколы RPC (Remote Procedure Call).

SOAP (Simple Object Access Protocol — простой протокол доступа к объектам)

Один из наиболее известных вариантов — SOAP. Представляет собой библиотеку, которая инкапсулирует в HTTP XML-заголовки. Использование SOAP поверх HTTP — скорее сложившаяся практика, чем стандарт: он может быть вложен и в другой прикладной протокол, например в SMTP.

Удобство использования протокола SOAP: подключил библиотеку, и готово.

Недостатки: избыточность и нечитаемость XML, большой объем сообщений, игнорирование достоинств HTTP (методов, кодов состояний).

Пример SOAP-запроса (вкладывается в HTTP):

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <getProductDetails xmlns="http://warehouse.example.com/ws">
      <productID>12345</productID>
    </getProductDetails>
  </soap:Body>
</soap:Envelope>
```

Пример SOAP-ответа:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <getProductDetailsResponse xmlns="http://warehouse.example.com/ws">
      <getProductDetailsResult>
        <productID>12345</productID>
        <productName>Стакан граненый</productName>
        <description>Стакан граненый. 250 мл.</description>
        <price>9.95</price>
        <currency>
          <code>840</code>
          <alpha3>USD</alpha3>
          <sign>$</sign>
          <name>US dollar</name>
          <accuracy>2</accuracy>
        </currency>
      </getProductDetailsResult>
    </getProductDetailsResponse>
  </soap:Body>
</soap:Envelope>
```



```
        </currency>
        <inStock>true</inStock>
    </getProductDetailsResult>
</getProductDetailsResponse>
</soap:Body>
</soap:Envelope>
```

По материалам <https://ru.wikipedia.org/wiki/SOAP>.

С другой стороны, если в SOAP используется громоздкий XML, почему бы не изобрести свой формат обмена с лаконичностью и эстетическим удовлетворением? Сказано — сделано. Так появился XML-RPC.

XML-RPC

Название протокола расшифровывается как «удаленный вызов процедур через XML» и говорит само за себя.

Пример запроса:

```
<?xml version="1.0"?>
<methodCall>
  <methodName>examples.getStateName</methodName>
  <params>
    <param>
      <value><i0>41</i0></value>
    </param>
  </params>
</methodCall>
```

Пример ответа:

```
<?xml version="1.0"?>
<methodResponse>
  <params>
    <param>
      <value><string>South Dakota</string></value>
    </param>
  </params>
</methodResponse>
```

По материалам <https://ru.wikipedia.org/wiki/XML-RPC>.

Это уже лучше, но все равно слишком громоздко. Почему бы не упростить ещё — например, использовать вместо XML — JSON?.

JSON-RPC

Название протокола переводится как «удаленный вызов процедур через JSON».

Пример запроса:

```
{"jsonrpc": "2.0", "method": "subtract", "params": [23, 42], "id": 2}
```

Пример ответа:

```
{"jsonrpc": "2.0", "result": -19, "id": 2}
```

По материалам <https://ru.wikipedia.org/wiki/JSON-RPC>

Вот это уже гораздо лучше, но все равно не то. Дело в том, HTTP по-прежнему используется всего лишь как транспорт, а его широкие возможности, такие как методы, коды состояний, заголовки (которые сами прекрасно годятся для построения API), не используются. Так, если мы запрашиваем некий ресурс и в JSON получаем ответ «не найден», код возврата HTTP будет все равно 200 Ok, что очевидно неправильно с точки зрения философии HTTP и неинформативно с точки зрения разработчика.

REST API

Representational State Transfer (REST) — «передача состояния представления», альтернатива SOAP и RPC.

В отличие от всего вышеперечисленного это не протокол и не библиотека — это философия, согласно которой сами методы HTTP, коды состояний, запросы будут нести смысловую нагрузку. Соответственно, мы будем использовать GET для того, чтобы получить информацию, PUT — отправить, DELETE — удалить, в адресе ресурса мы укажем объект, с которым хотим работать, плюс у нас еще есть заголовки и возможность получать JSON с данными в ответе.

Хороший пример REST API — API Яндекс.Диска. Документация по API доступна по адресу: <https://tech.yandex.ru/disk/api/concepts/about-docpage/>.

Запросы можно сразу оттестировать на тестовом полигоне: <https://tech.yandex.ru/disk/poligon/>.

Подключаемся по HTTPS на cloud-api.yandex.net:443:

```
GET /v1/disk HTTP/1.1
```

Получаем 200 Ok и тело HTTP-ответа:

```
{
```

```
"max_file_size": 10737418240,
"total_space": 35433480192,
"trash_size": 0,
"is_paid": false,
"used_space": 26186303125,
"system_folders": {
  "odnoklassniki": "disk:/Социальные сети/Одноклассники",
  "google": "disk:/Социальные сети/Google+",
  "instagram": "disk:/Социальные сети/Instagram",
  "vkontakte": "disk:/Социальные сети/ВКонтакте",
  "mailru": "disk:/Социальные сети/Мой Мир",
  "downloads": "disk:/Загрузки/",
  "applications": "disk:/Приложения",
  "facebook": "disk:/Социальные сети/Facebook",
  "social": "disk:/Социальные сети/",
  "screenshots": "disk:/Скриншоты/",
  "photostream": "disk:/Фотокамера/"
},
"revision": 1503705721964162
}
```

Удаляем файл test:

```
DELETE /v1/disk/trash/resources?path=test HTTP/1.1
```

Ответ 404 Not Found, тело:

```
/v1/disk/trash/resources?path=tes
```

Помимо REST API в описании API имеется WebDAV API для подключения диска в Linux и Windows как сетевого диска и API для уведомлений вашей ПО на основе XMPP.

WebDAV

Протокол WebDAV описан по адресу: <https://tech.yandex.ru/disk/doc/dg/concepts/about-docpage/>.

Посмотрим, какие новые методы есть в WebDav по сравнению с HTTP.

Пример: приложение создает каталог /b/ внутри каталога /a/, находящегося в корневом каталоге Диска:

```
MKCOL /a/b/ HTTP/1.1
Host: webdav.yandex.ru
Accept: */*
Authorization: OAuth 023747ff12123cdhfhy773457
```

Если каталог /a/ существует и каталог /b/ был создан успешно, возвращается следующий ответ:

```
HTTP/1.1 201 Created
Content-Length: 0
```

По материалам <https://tech.yandex.ru/disk/doc/dg/reference/move-docpage/>

Пример: файл lion.png копируется из папки pictures в папку animals:

```
COPY /pictures/lion.png HTTP/1.1
Host: webdav.yandex.ru
Accept: */*
Authorization: OAuth 023747ff12123cdhfy773457
Destination: /animals/lion.png
Overwrite: F
```

Заголовок Overwrite можно задать, чтобы запретить перезапись уже существующего файла с таким именем. Значение Т по умолчанию разрешает перезапись, значение F — запрещает. Если в каталоге /animals/ уже есть файл lion.png, то запрос из примера не будет выполнен.

Если копирование прошло успешно, возвращается следующий ответ:

```
HTTP/1.1 201 Created
Content-Length: 0
```

По материалам <https://tech.yandex.ru/disk/doc/dg/reference/move-docpage/>

Практическое задание

Можно выполнить задание № 1, или задание № 2, или оба. Задание № 3 — по желанию.

Задание номер 1

1. Проанализируйте заголовки своих входящих писем.
2. Опишите, как письма отправлены и как обработаны.
3. Зафиксируйте IPv6 (при наличии).

Задание номер 2

1. Выберите несколько ресурсов, доступных по HTTP.
2. Попробуйте подключиться к ним и вручную сформировать запрос.
3. Подключитесь к порту 80, используя ресурс, доступный по HTTPS.

Отчет о выполненном практическом задании сдается в формате .docx или .pdf. Можно включать скриншоты, но документ должен содержать описание выполненных вами действий, введенные или обнаруженные вами заголовки с расшифровкой и ваши выводы.

Если домашних заданий 1 и 2 недостаточно, можно выполнить усложненное задание по желанию.

Задание номер 3 **

1. Выберите ресурс, предоставляющий API (многие такие ресурсы предоставляют возможность тестировать запросы на специальной странице).
2. Потренируйтесь, выполняя запросы к REST API.

Литература и ресурсы для дальнейшего изучения

1. <http://rfc.com.ru/> — репозиторий стандартов RFC на русском языке.
2. <http://xgu.ru/> — энциклопедия по сетям и ОС.
3. <https://version6.ru/> — ресурс по IPv6.
4. <https://www.restapitutorial.com/> — tutorial по разработке REST API.
5. <https://swagger.io/> — популярный фреймворк для разработки и документации REST API.
6. <https://ru.wikipedia.org/wiki/WebSocket> — о вебсокетах в Википедии.
7. <http://websocket.org/> — о веб-сокетах.
8. <https://www.websocket.org/echo.html> — эхо-сервер (веб-сокеты).
9. <https://ru.wikipedia.org/wiki/Slither.io> — HTML5-игра, использующая веб-сокеты.

Дополнительная литература

1. <https://tools.ietf.org/html/rfc2616>.
2. <https://www.restapitutorial.com/>.
3. <https://technet.microsoft.com/en-us/library/cc959885.aspx>.
4. Таненбаум Э., Уэзеролл Д. Компьютерные сети. 5-е изд. — СПб.: Питер, 2012. — 960 с. (Глава 7.)

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. <https://tech.yandex.ru/disk/doc/dg/reference/move-docpage/>.
2. <https://ru.wikipedia.org/wiki/JSON-RPC>.
3. <https://ru.wikipedia.org/wiki/SOAP>.
4. <https://ru.wikipedia.org/wiki/DNSBL>.
5. https://ru.wikipedia.org/wiki/DomainKeys_Identified_Mail.