



Инфраструктура вычислений в биоинформатике

Лекция 3. Компьютерные сети. Стек протоколов TCP/IP. Прикладной и транспортный уровни.

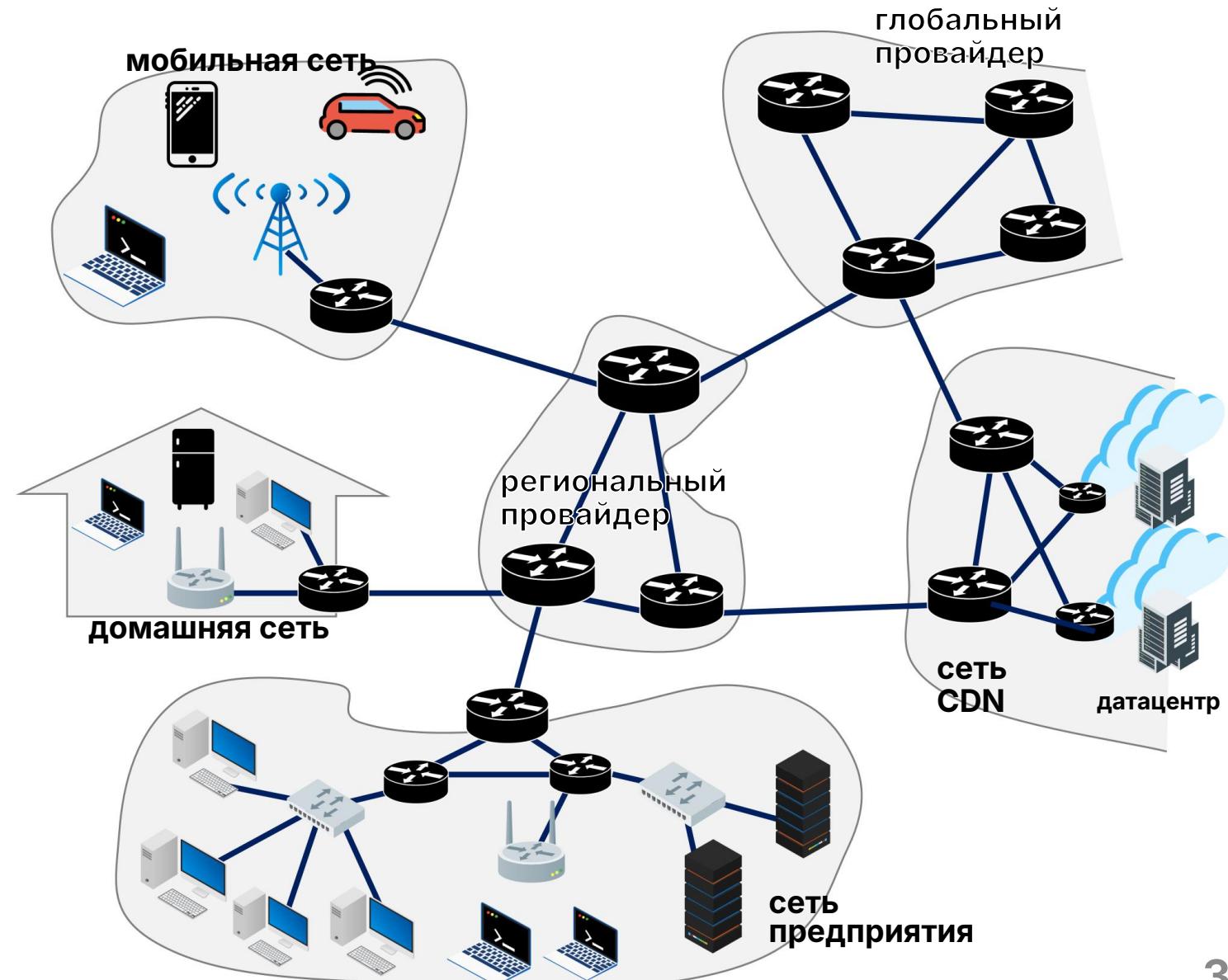
Что такое интернет?

Интернет - сеть сетей (межсеть)

Состоит из **конечных устройств** (хосты, или периферия) и **ядра сети** (маршрутизаторы, коммутаторы).

Задача: эффективно передавать данные между разными устройствами.

Как это реализуется?
Протоколы - интернет как сервисы!

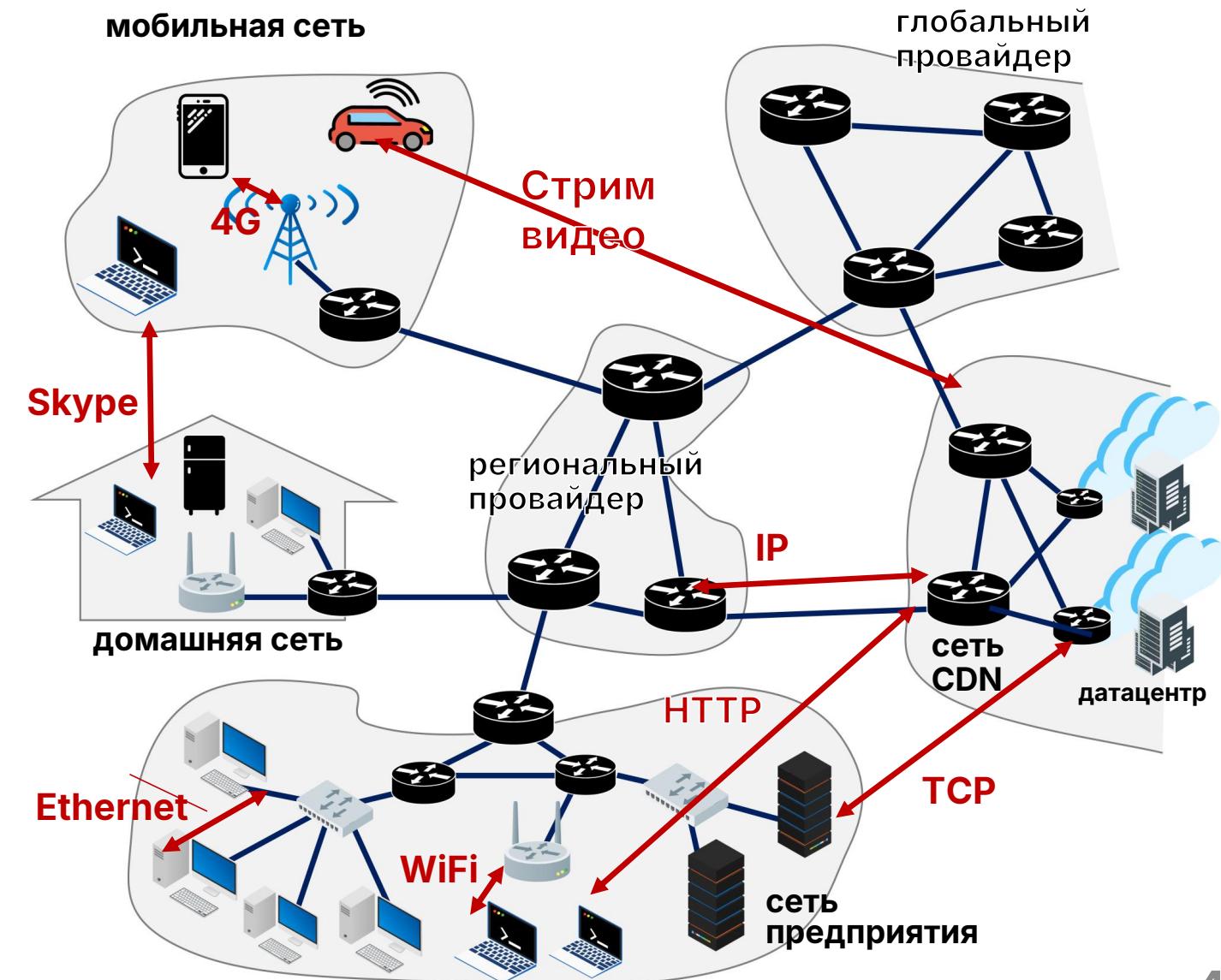


Интернет - сеть взаимодействия

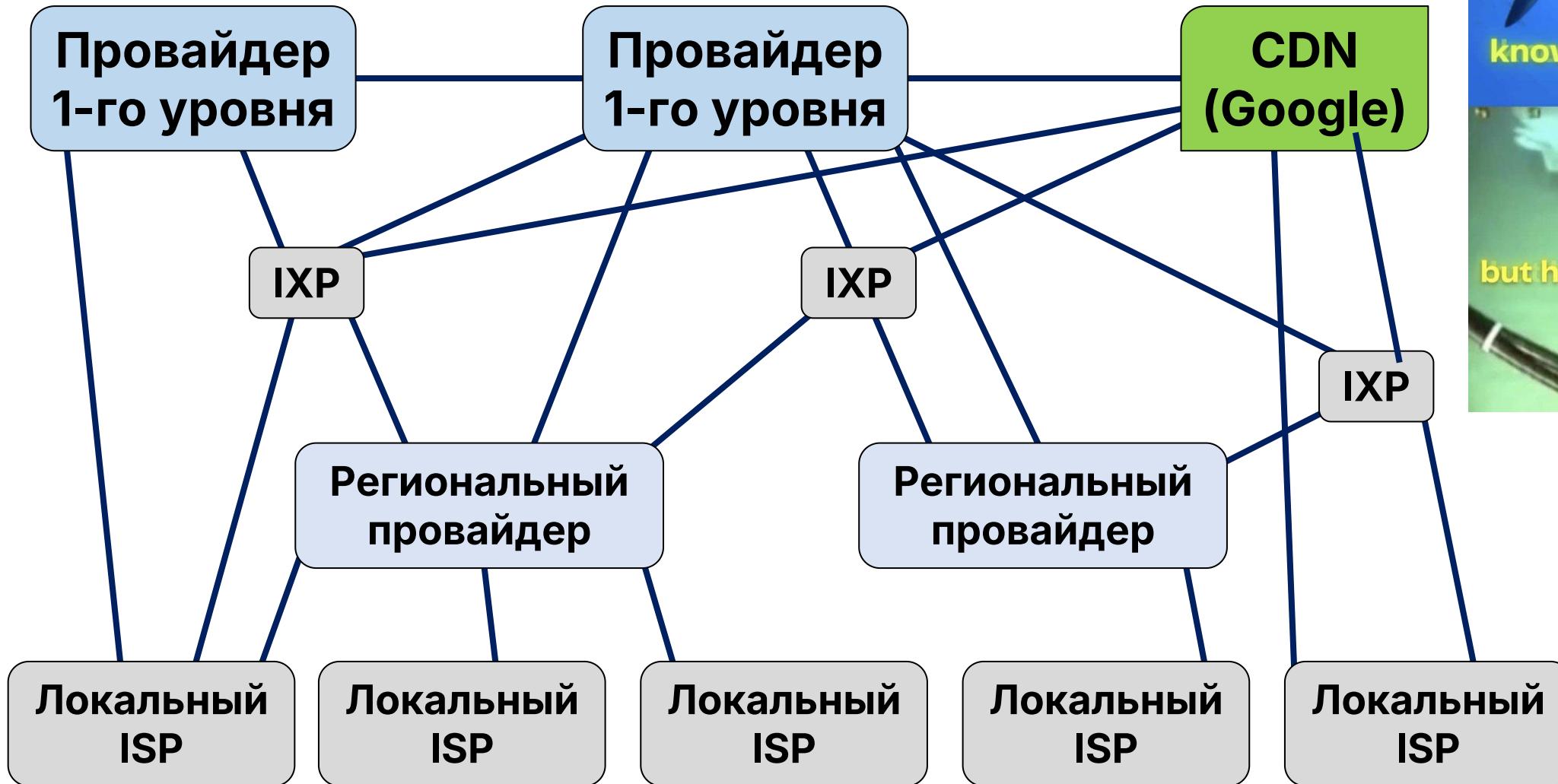
Связанные между собой сети взаимодействуют по особым правилам - протоколам (HTTP, TCP, IP, ...).

Интернет - black-box
разработчика, который предоставляет сервисы для распределённых приложений.

Мораль как и везде - не пытайтесь переизобрести велосипед, пользуйтесь изобретённым.

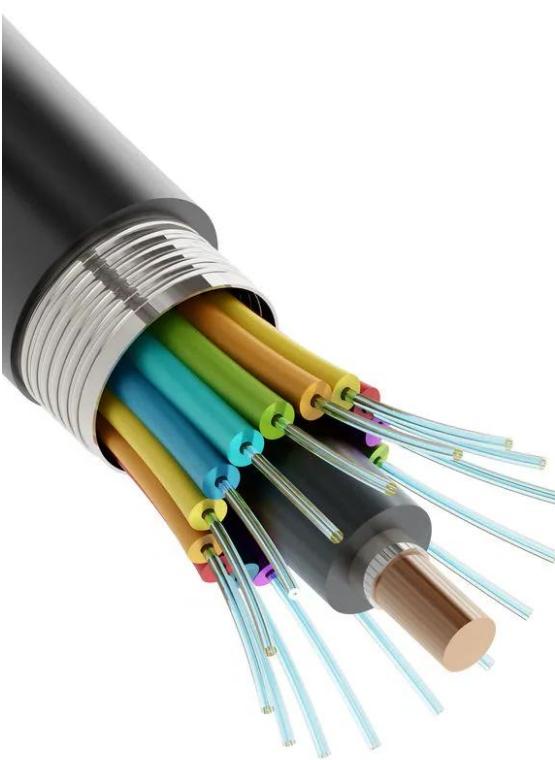


Глобальный интернет

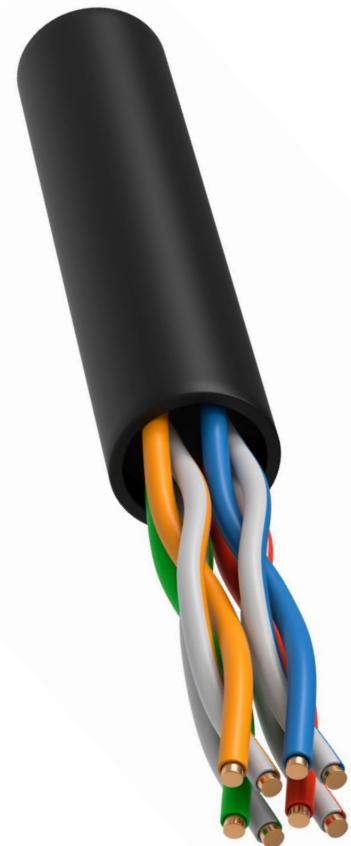


*Среды передачи

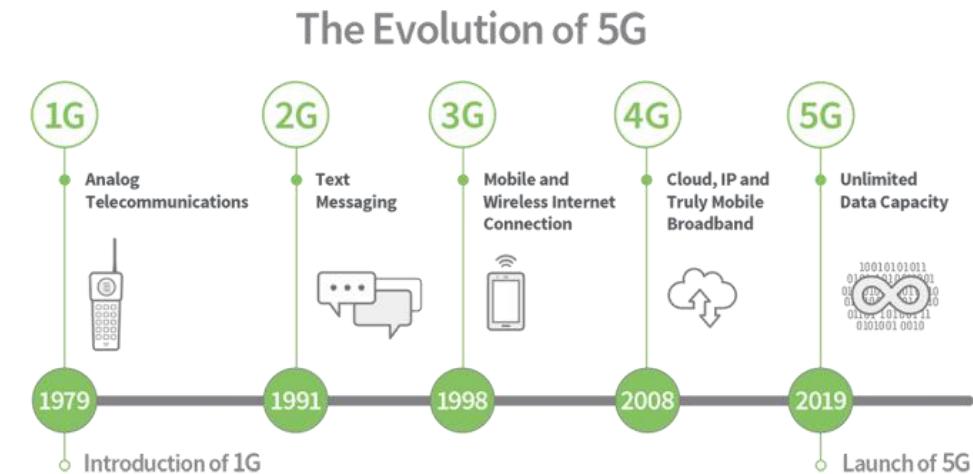
Передача на физическом уровне:



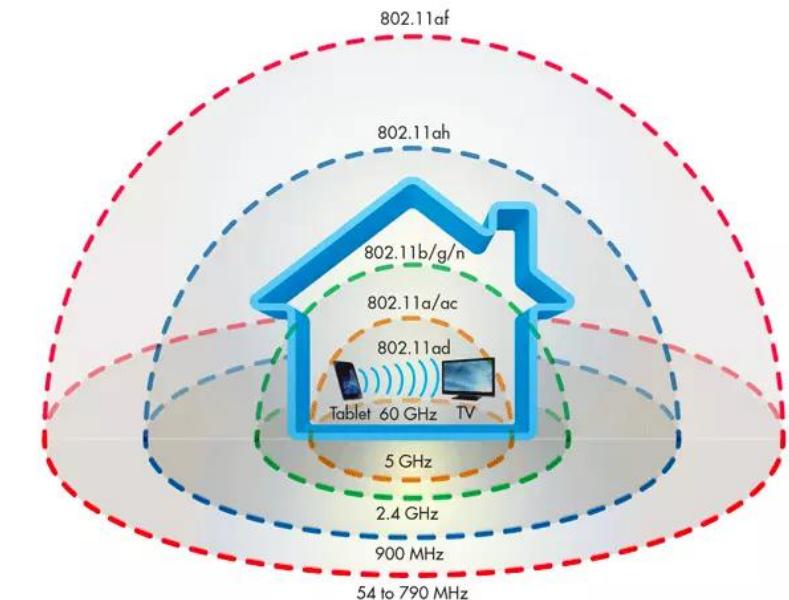
коаксиальное оптоволокно
(лазерные вспышки + электричество)



медная витая пара
(DSL, вместе с телефонией)



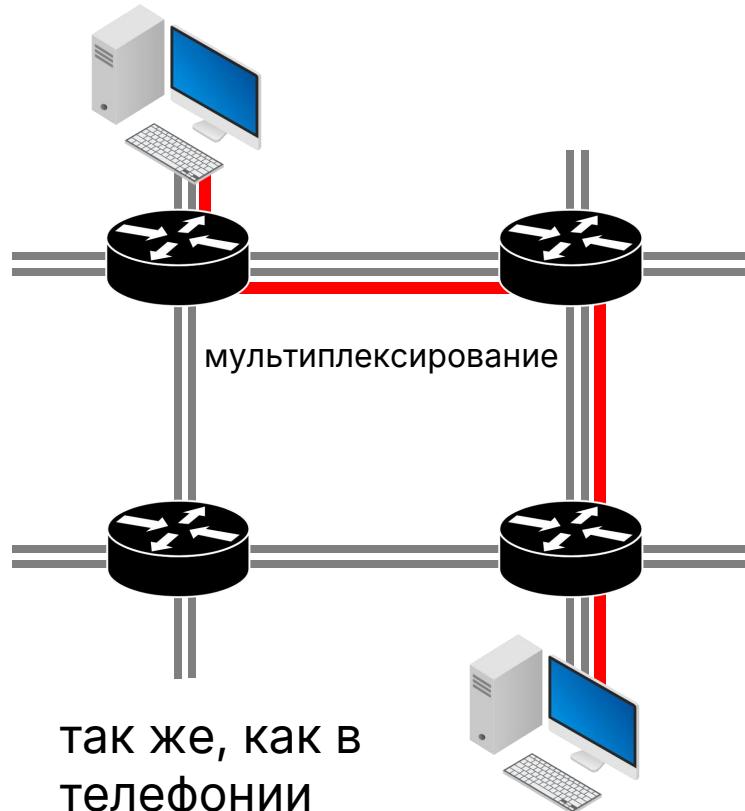
радиочастотные сигналы: WiFi и сотовая связь



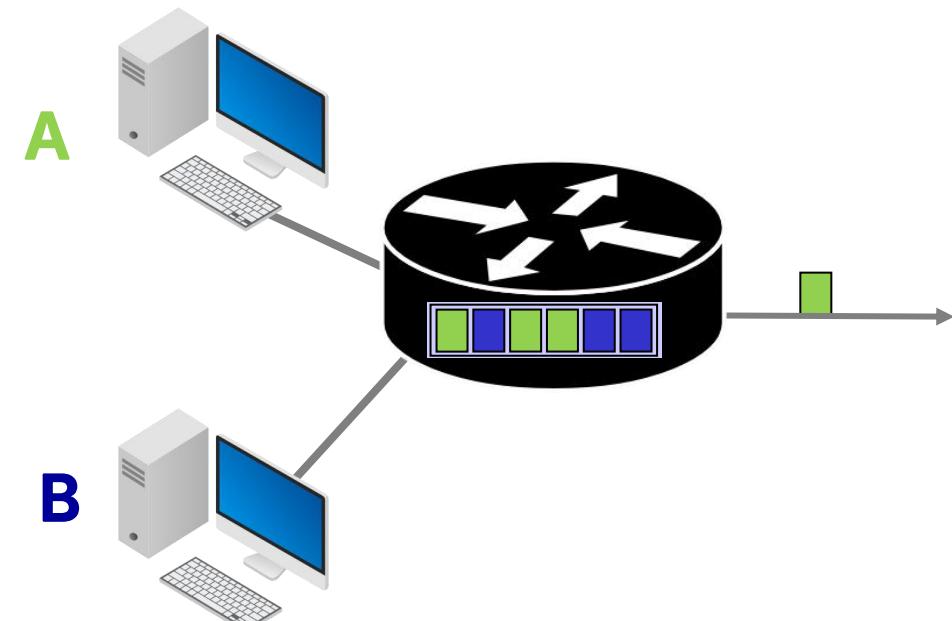
Передача данных

Файлы надо сегментировать (почему?). Как работать роутеру?

Канальная передача данных



Пакетная передача данных



Почему пакетная передача: пример

Пусть есть несущая 1 Gb/s.

Каждый пользователь при активности даёт 100 Mb/s.

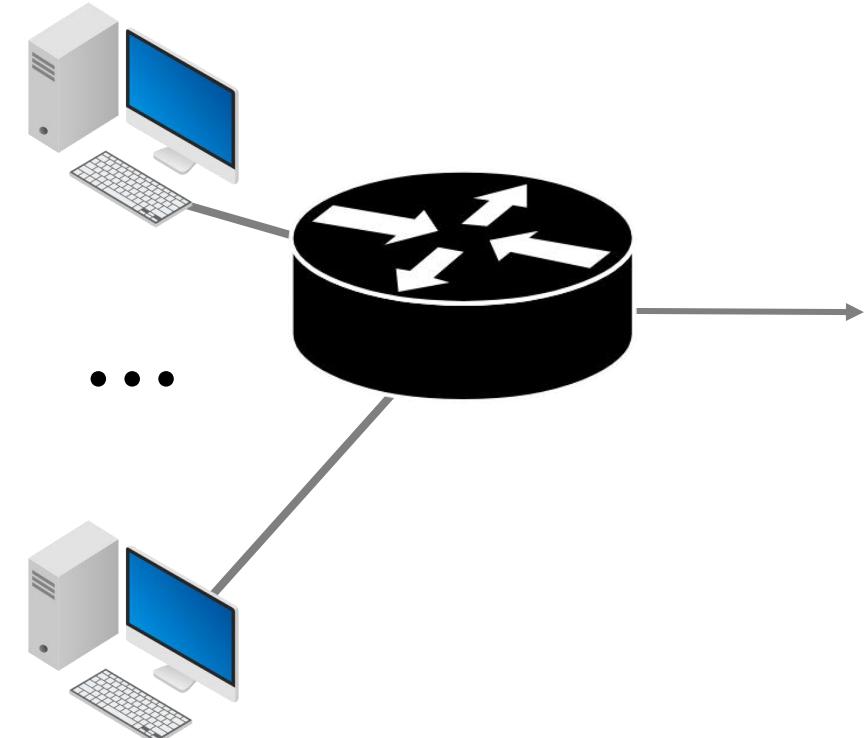
Каждый активен с вероятностью 10%

Канальная передача: 10 пользователей

Пакетная: с какой вероятностью больше 10 юзеров активны?

Вероятность задержки достигает 0.5 при ~ 96 пользователях!

При 35 пользователях вероятность задержки 0.0004

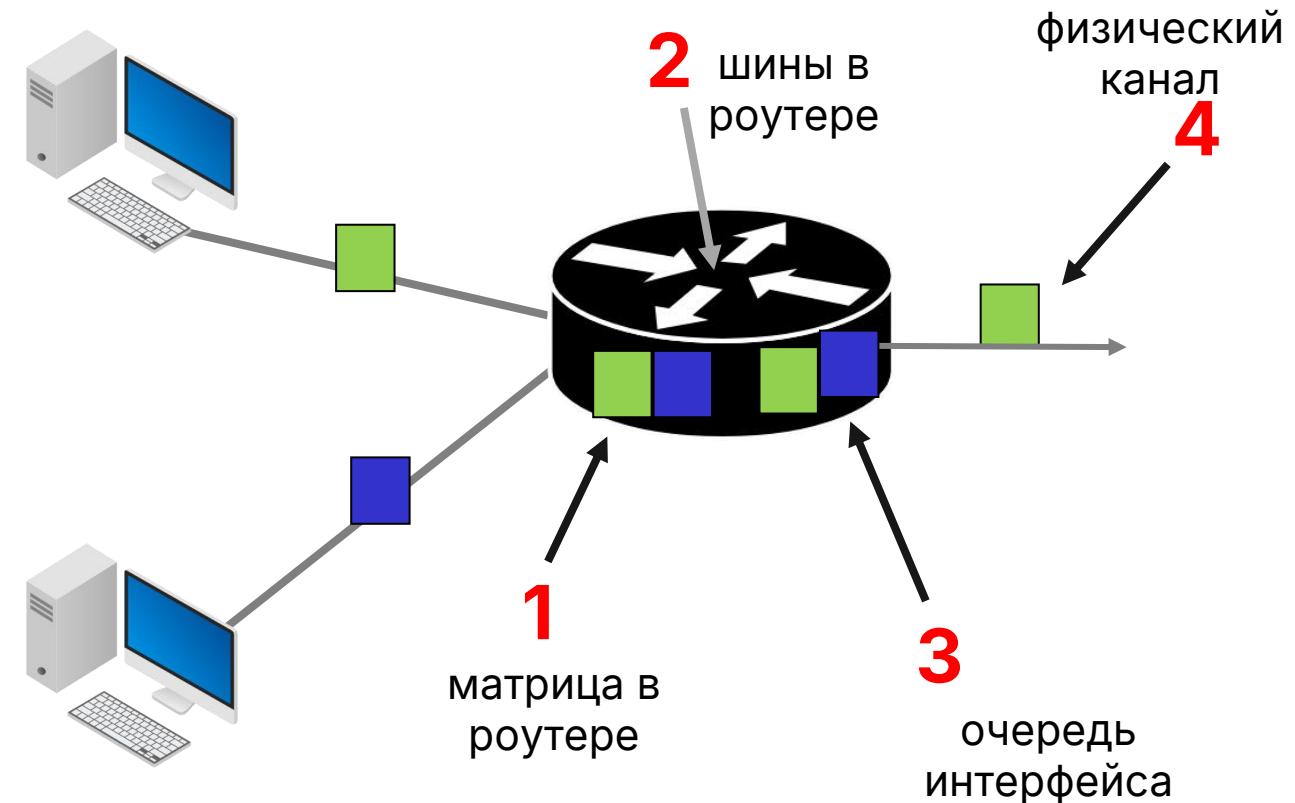


Источники потери данных

Потеря данных происходит из-за разных скоростей обработки.

Задержка состоит из:

1. Времени определения выходного интерфейса
2. Времени передачи на интерфейс
3. Времени ожидания в очереди
4. Времени передачи по каналу



traceroute: просмотр задержки

```
~$ traceroute -T -p 443 altius.org
traceroute to altius.org (198.49.23.144), 30 hops max, 60 byte packets
 1 gw2.bioinf.fbb.msu.ru (192.168.111.1)  0.121 ms  0.123 ms  0.135 ms
 2 172.16.0.15 (172.16.0.15)  0.282 ms  0.364 ms  0.365 ms
 3 93.180.1.2 (93.180.1.2)  0.690 ms  0.881 ms  0.733 ms
 4 188.44.33.37 (188.44.33.37)  0.917 ms  0.971 ms  1.157 ms
 5 188.44.33.2 (188.44.33.2)  0.650 ms  0.657 ms  0.633 ms
 6 93.180.0.190 (93.180.0.190)  0.990 ms  0.764 ms  0.767 ms
 7 654.ae0.gw5.m9.msk.niks.su (194.190.254.117)  1.121 ms  0.992 ms  1.063 ms
 8 engine.sobko.sovintel.ru (62.231.31.101)  1.885 ms  1.409 ms  1.517 ms
 9 * * *
10 * * *
11 a2-21-124-51.deploy.static.akamaitechnologies.com (2.21.124.51)  63.529 ms
63.336 ms  60.361 ms
12 a2-19-6-24.deploy.static.akamaitechnologies.com (2.19.6.24)  56.253
13 * * *
14 198.49.23.144 (198.49.23.144)  55.150 ms  51.681 ms  55.589 ms
15 198.49.23.144 (198.49.23.144)  53.415 ms  51.647 ms  66.765 ms
```



Сервер → Gateway → МГУ → IXP РФ → РТК → Европейский Akamai → ... → Сервер

Протокол: взаимодействие устройств

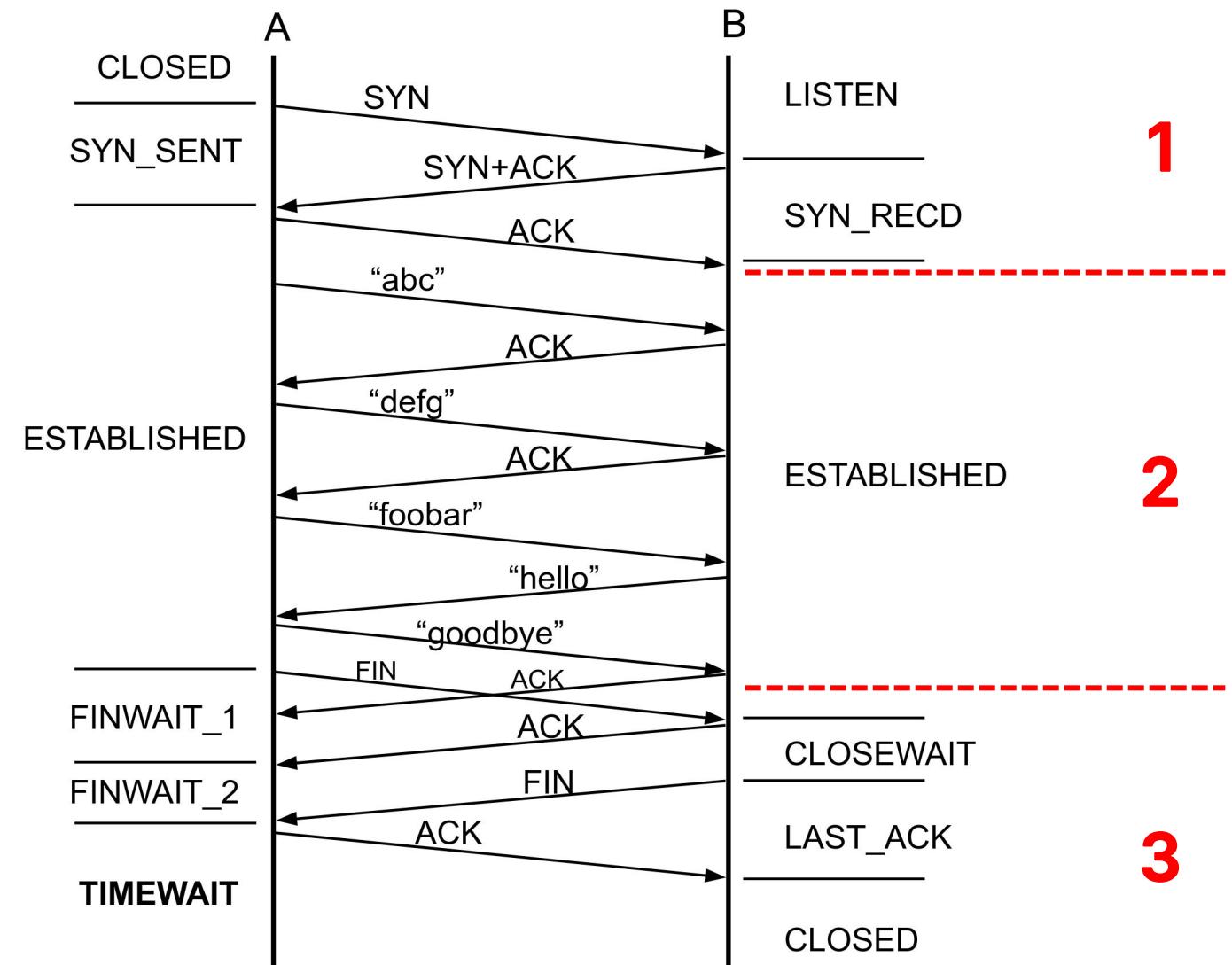
Как эффективно
реализовать общение
между компьютерами?

Так же, как между людьми!

2-3 этапа:

1. Приветствие
2. Собственно общение
3. Прощание
(опционально)

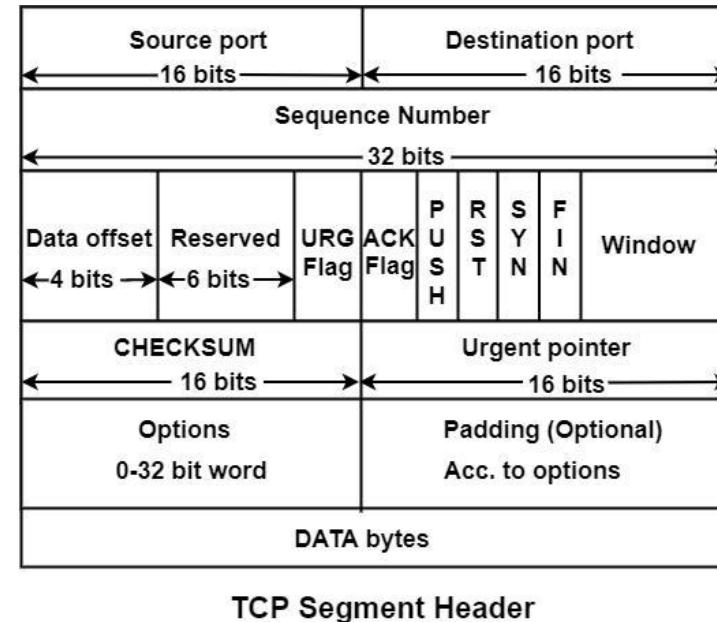
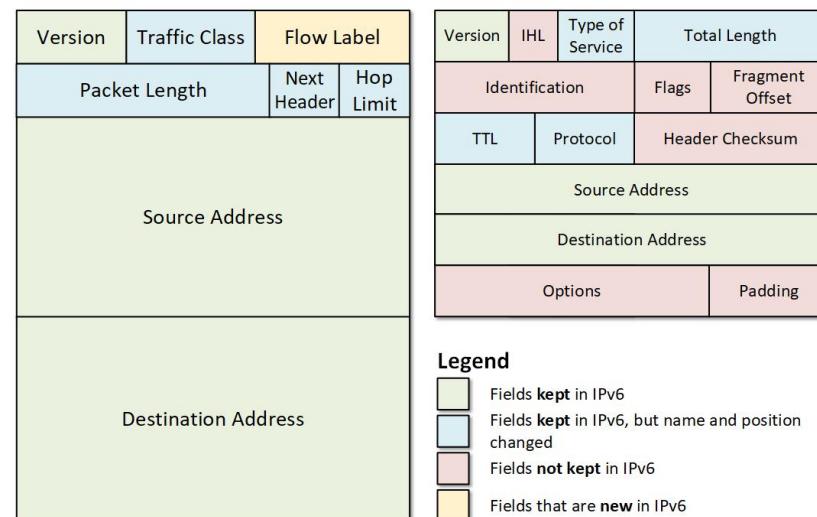
Так работают “хорошие” протоколы.
Не все протоколы такие. “Best-effort”
- лучшее описание интернета



Пример: TCP

Протокол: язык общения

Всё это возможно благодаря специальной структуре пакетов протокола. У каждого протокола она своя – позволяет разный функционал.



TCP Segment Header



UDP Segment Header

Как правильно реализовать протоколы?

Стек протоколов

Каждый пишет
свой сетевой код

неизбежно :(

несовместимость,
ошибки, огромные
лишние затраты

Решение: стек протоколов.

Каждый уровень реализует свои сервисы,
пользуясь нижележащими. Модульная
структура даёт независимость и гибкость

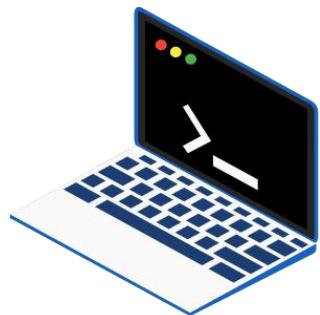
Аналогия с авиаперелётами:
изменение работы стойки регистрации
не влияет на то, как у самолёта
занимается двигатель

7	APPLICATION
6	PRESENTATION
5	SESSION
4	TRANSPORT
3	NETWORK
2	DATA LINK
1	PHYSICAL

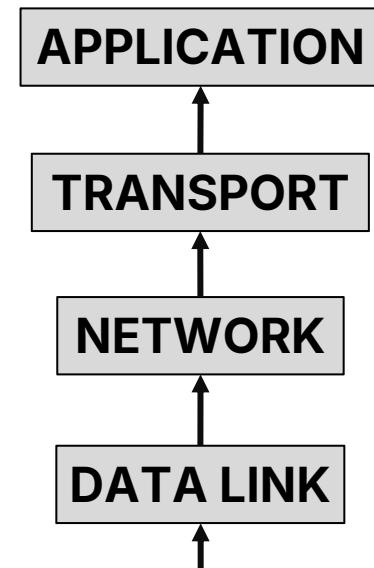
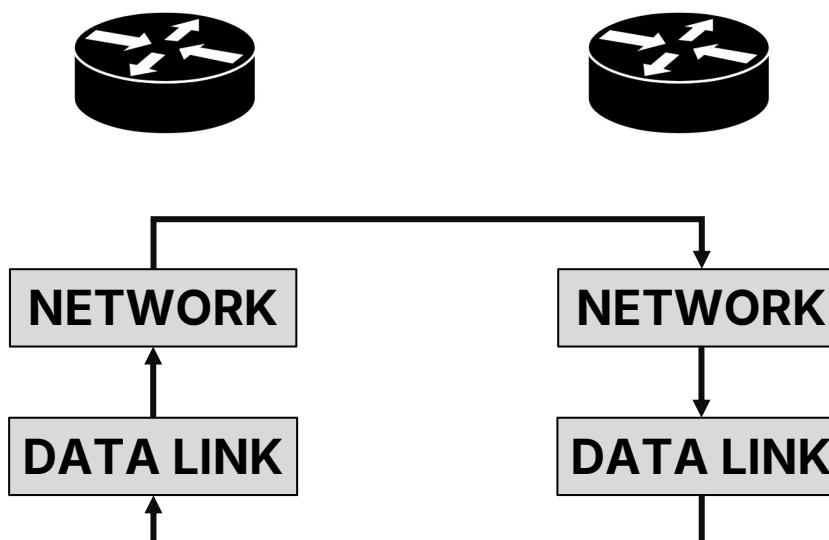
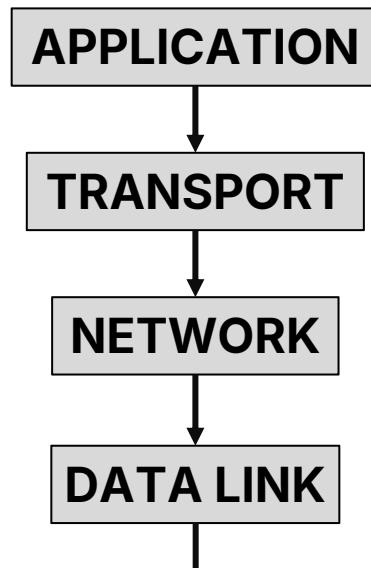
Сетевая модель протоколов OSI

Реальный стек протоколов: TCP/IP

TCP-over-IP: то, как идеал реализован на практике.

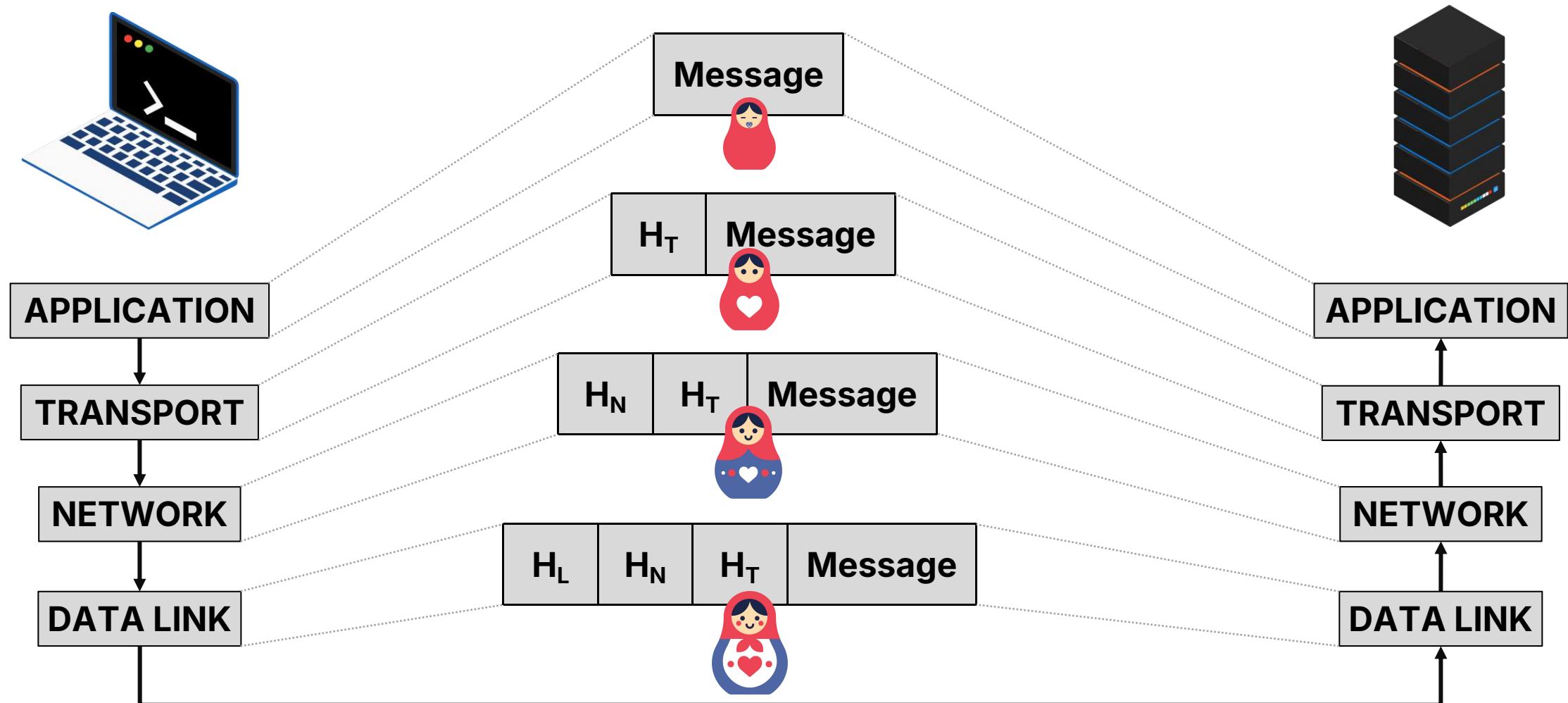


Presentation и Session
реализованы в прикладном
уровне, physical входит в
канальный уровень



Инкапсуляция

Большая часть передаваемого пакета - заголовки протоколов.



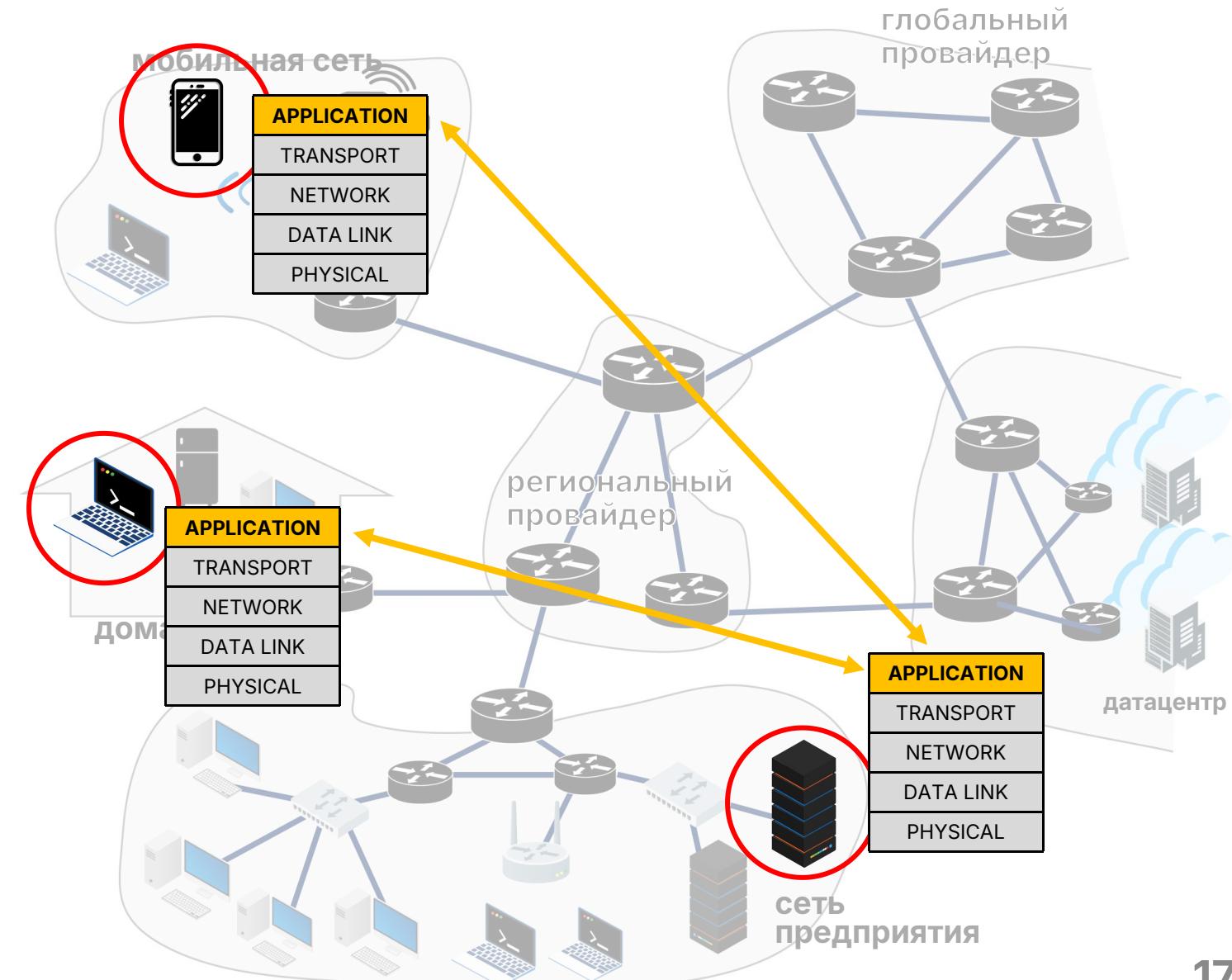


NETFLIX

Прикладной уровень

- Работает только на хостах, общающихся по сети (веб-сервер и браузер)
- Парадигма **клиент-сервер**:
 - **сервер** - always-on хост с постоянным адресом
 - **клиент** - любое устройство, кроме сервера, не вз-т друг с другом
- Бывает P2P, не рассматриваем
- Уровень коммуникации **ПРОЦЕССОВ ОС**

Протоколы: **HTTP, FTP, SMTP, DNS**



Какие службы требуются от транспортного слоя?

- **Целостность данных** (опциональна, например, для звонков) - TCP & UDP протоколы
- **Своевременная доставка** (обязательна для звонков) - TCP
- Определённая пропускная способность канала (обязательна, например, для онлайн-игр) - TCP
- **Безопасность** - TLS

Приложение	Прикладной протокол	Транспортный протокол
Передача файлов	FTP	TCP
e-mail	SMTP	TCP
Веб-документы	HTTP	TCP
Телефония	SIP, HTTP, DASH	TCP or UDP
Стриминг	HTTP	TCP
Онлайн-игры	WOW, FPS	UDP or TCP

Что определяет прикладной протокол?

- Типы сообщений (request / response)
- Синтаксис и семантика сообщений
- Правила отправки и получения сообщений

Идея в вынесении всего сложного за пределы ядра сети.

**Примеры открытых
протоколов (RFC):**



Проприетарные протоколы:



HTTP

HyperText Transfer Protocol

Работает через одно или несколько соединений TCP (персистентность HTTP).

НЕ хранит состояния обмена (stateless, историю не хранит)

1. Открытие TCP соединения
2. Передача файла
3. Закрытие соединения

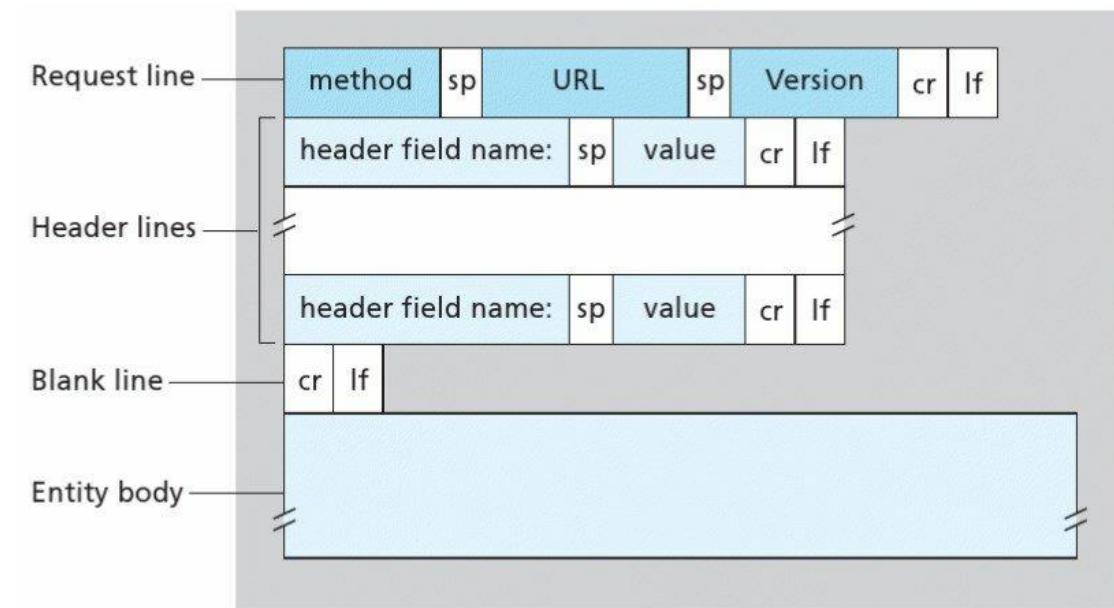
Иногда используется прокси-кэш сервер для более быстрого доступа.

2 типа сообщений: request и response.

Пример GET-request'a через библиотеку `requests`

```
GET http://kodomo.fbb.msu.ru/  
User-Agent: python-requests/2.32.3  
Accept-Encoding: gzip, deflate, br, zstd  
Accept: */*  
Connection: keep-alive
```

Поле **if-modified-since** - кэш браузера



HTTP

HyperText Transfer Protocol

Статус-коды:

HTTP Status Codes



2 типа сообщений: request и response.

Пример response с kodomo.fbb.msu.ru

```
HTTP/1.1 200 OK
Date: Sat, 03 Jan 2026 12:07:51 GMT
Server: Apache/2.4.62 (Debian)
Vary: Cookie,User-Agent,Accept-Language,Accept-Encoding
Content-Encoding: gzip
Keep-Alive: timeout=5, max=99
Connection: Keep-Alive
Transfer-Encoding: chunked
Content-Type: text/html; charset=utf-8
```

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
...
```

HTTP request-сообщения (вспоминаем)

GET

Отправка
данных на
сервер для
получения
конкретной
информации

POST

Для страниц со
вводом
(пароли,
формы)

PUT

Загрузка
нового
файла на
сервер

DELETE

Удаление
объекта с
конкретным URI

HEAD

Получить хедеры
взаимодействия

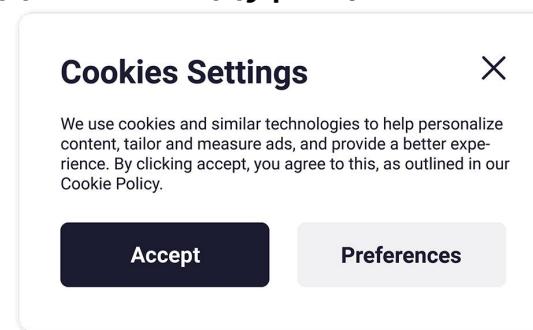
Элементы CRUD-
парадигмы:
Create-Read-
Update-Delete

Сохранение состояния HTTP: cookies

Что делать, если хочется всё же хранить состояние?

Сервер с БД будет отправлять пользователю cookie-идентификатор, который клиент **нужно будет отправлять** обратно в cookie-файл.

Куки-файлы позволяют функционал **рекомендаций, аутентификации** пользователя, и **многое другое**. Однако стоит вопрос приватности, поэтому на каждом сайте вы видите эти дурацкие баннеры.



CLI для HTTP

netcat: рабочий для TCP/UDP, но в домашке может не сработать из-за невозможности работы с TLS.

```
printf "GET / HTTP/1.1\r\nHost: kodomo.fbb.msu.ru\r\n\r\n" | nc kodomo.fbb.msu.ru 443
HTTP/1.1 400 Bad Request
Date: Sat, 03 Jan 2026 12:30:05 GMT
Server: Apache/2.4.62 (Debian)
Content-Length: 445
Connection: close
Content-Type: text/html; charset=iso-8859-1
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>400 Bad Request</title>
</head><body>
<h1>Bad Request</h1>
<p>Your browser sent a request that this server could not understand.<br />
Reason: You're speaking plain HTTP to an SSL-enabled server port.<br />
Instead use the HTTPS scheme to access this URL, please.<br />
</p>
<hr>
<address>Apache/2.4.62 (Debian) Server at kodomo.fbb.msu.ru Port 80</address>
</body></html>
```

CLI для HTTP

OpenSSL:

```
openssl s_client -connect kodomo.fbb.msu.ru:443 -servername kodomo.fbb.msu.ru -quiet
GET /wiki/ HTTP/1.1\r\nHost: kodomo.fbb.msu.ru\r\nConnection: close\r\n\r\n
Connecting to 93.180.63.127
depth=2 C=US, O=Internet Security Research Group, CN=ISRG Root X1
verify return:1
depth=1 C=US, O=Let's Encrypt, CN=R13
verify return:1
depth=0 CN=kodomo.fbb.msu.ru
verify return:1
HTTP/1.1 200 OK
Date: Sat, 03 Jan 2026 12:26:25 GMT
Server: Apache/2.4.62 (Debian)
Vary: Cookie,User-Agent,Accept-Language,Accept-Encoding
Connection: close
Transfer-Encoding: chunked
Content-Type: text/html; charset=utf-8

643
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/strict.dtd">
<html>
```

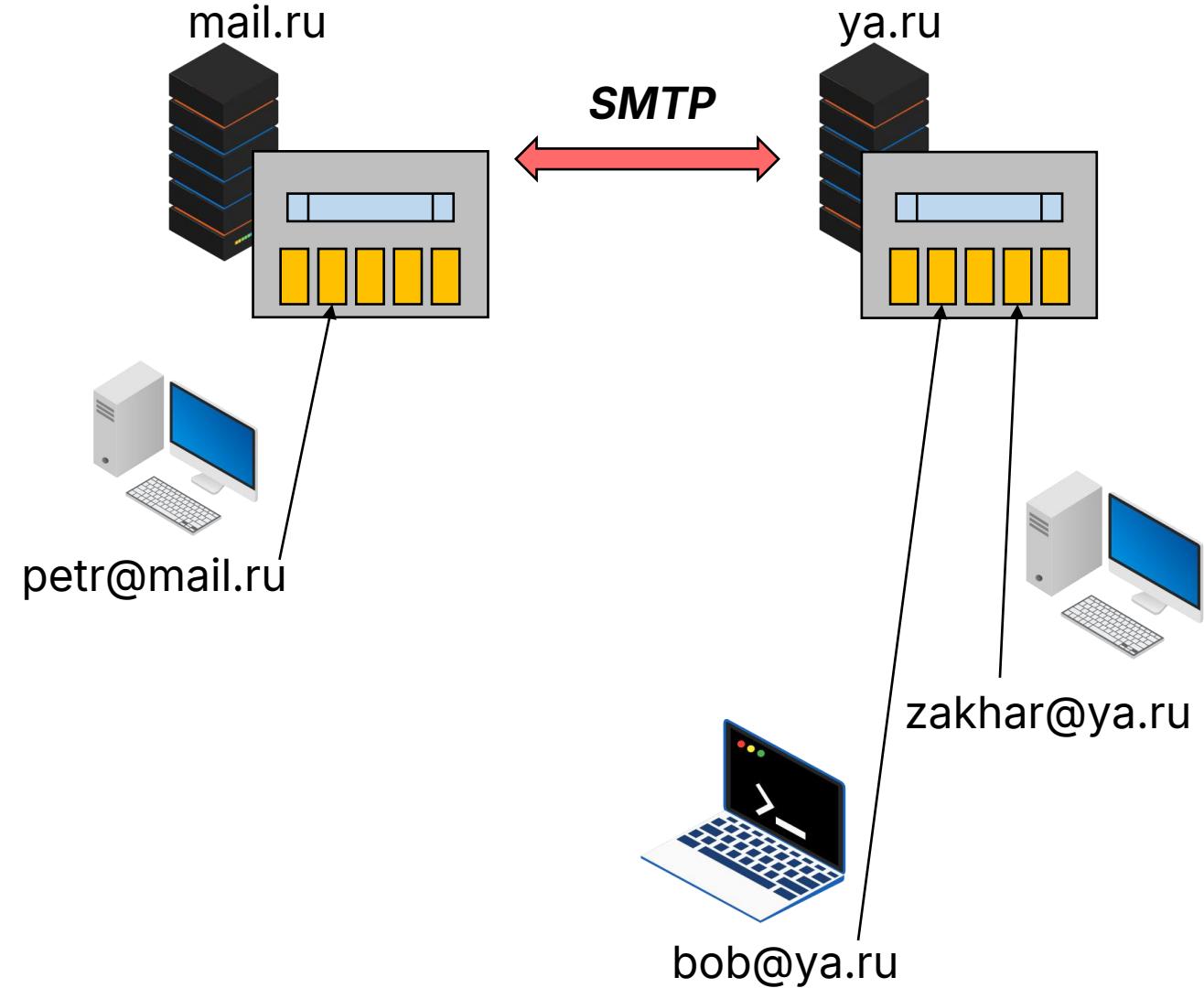
SMTP: e-mail

Ещё пример прикладного протокола: Simple Mail Transfer Protocol.

Три участника: отправитель, получатель и почтовые серверы.

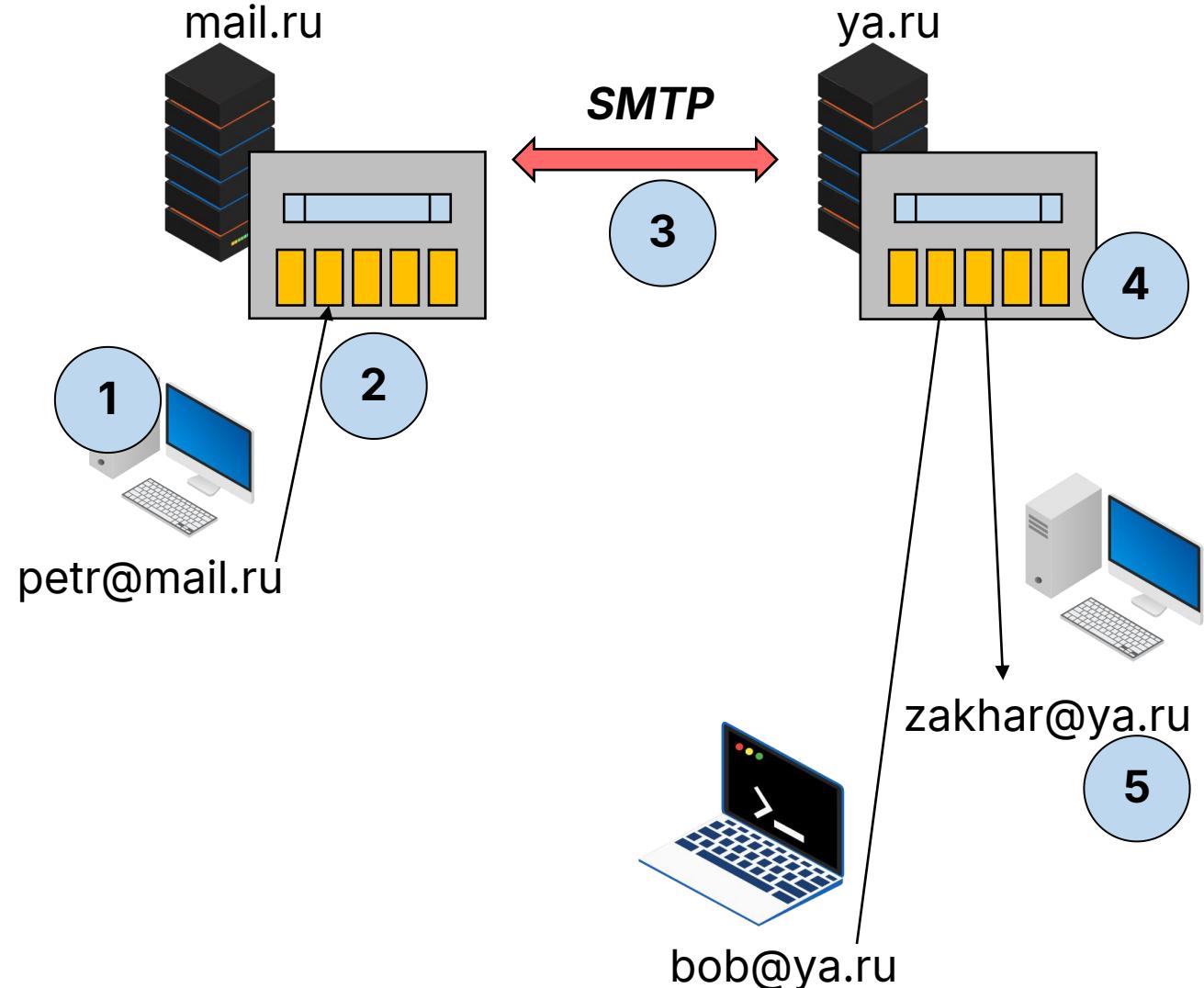
Сам протокол между почтовыми серверами.

1. Юзер кладёт с приложения письмо на свой сервер.
2. Клиент-серверный контакт между двумя серверами
3. Другой юзер считывает со своего сервера свои новые письма.



SMTP: e-mail

1. Пётр пишет письмо на адрес **zakhar@ya.ru**.
2. Приложение на ПК отправляет письмо в очередь на SMTP **mail.ru**
3. TCP-соединение между **mail.ru** и **ya.ru** и отправка сообщения Петра
4. **ya.ru** кладёт письмо в ящик **zakhar@ya.ru**
5. Захар читает своё письмо с помощью приложения на своем ПК (другой протокол! **IMAP**)



Пример SMTP-общения серверов

```
S: 220 ya.ru  
C: HELO mail.ru  
S: 250 Hello mail.ru, pleased to meet you  
C: MAIL FROM: <petr@mail.ru>  
S: 250 petr@mail.ru... Sender ok  
C: RCPT TO: <zakhar@ya.ru>  
S: 250 zakhar@ya.ru ... Recipient ok  
C: DATA  
S: 354 Enter mail, end with "." on a line by itself  
C: Hello Zakhar,  
C: MGIIMO finished?  
C:  
S: 250 Message accepted for delivery  
C: QUIT  
S: 221 ya.ru closing connection
```

Сообщения SMTP
после установления
TCP-соединения

Передача сообщения



В отличие от HTTP, происходит push данных, а не pull.
Много объектов. Есть статус-коды, оба в ASCII.

DNS

Критически важный протокол прикладного уровня: **Domain Name System**. Перевод адреса в IP для нижележащих по стеку протоколов.

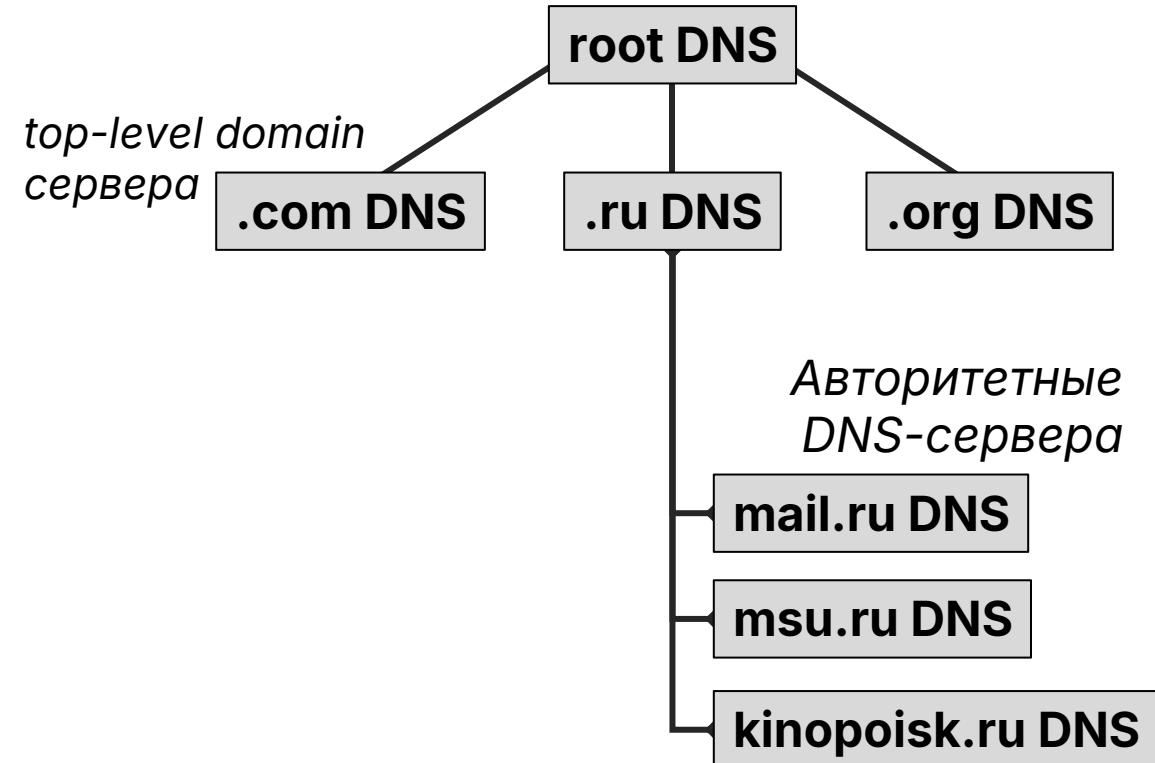
У каждого сервера есть 32-битный IP-адрес и "имя".

Фактически, **DNS** - это **иерархическая распределённая база данных** "имя - IP".

DNS предоставляет перевод, aliasing и перераспределение нагрузки между серверами

root DNS (13 шт) управляет **ICANN**
TLD DNS управляет реестрами DNS

Иерархия DNS-серверов:



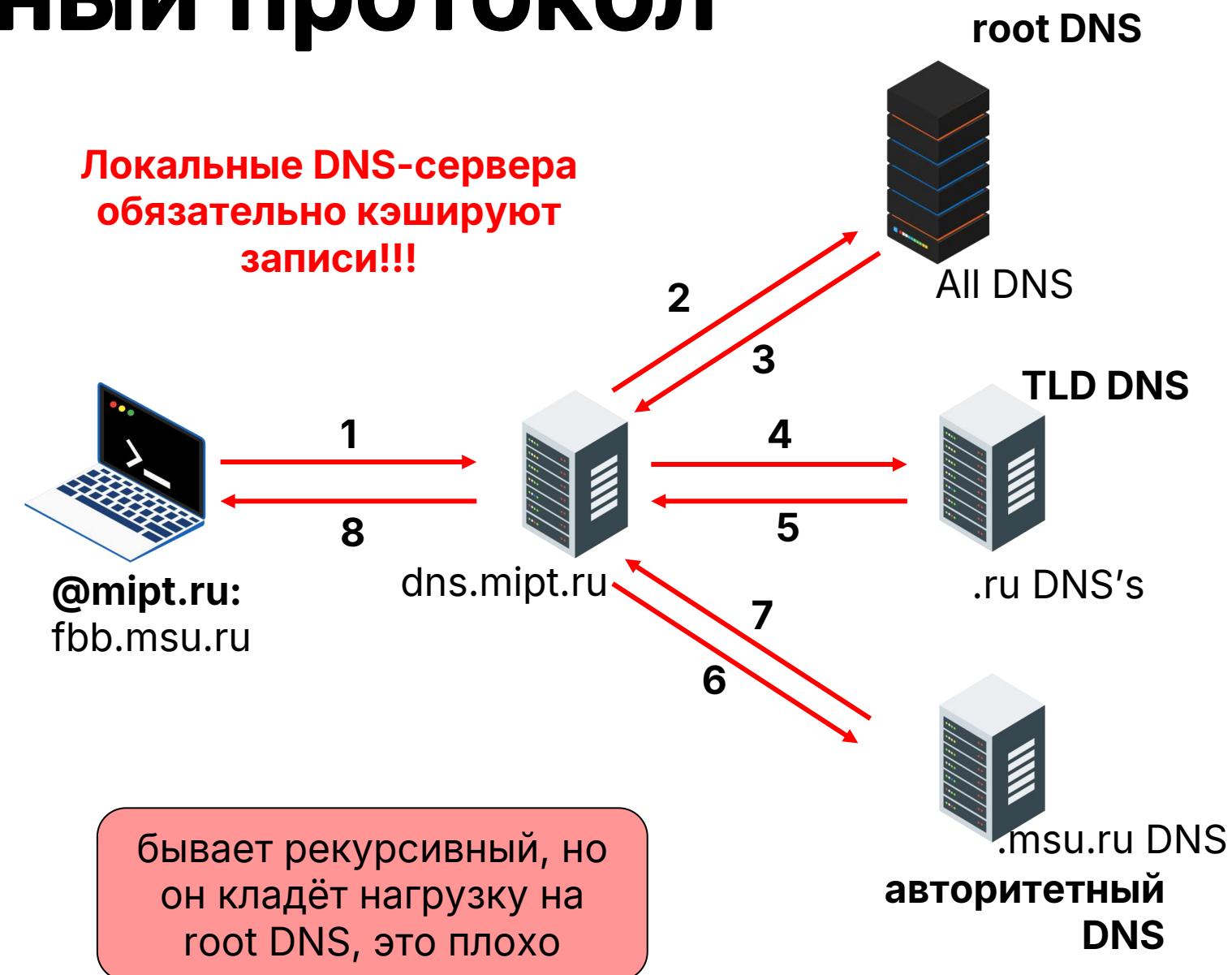
Посмотреть на свой локальный DNS-сервер: **resolvectl**. Обязательно вспомнить, когда будет docker!

DNS - итеративный протокол

Если в локальном кэше DNS-сервера не нашлось записи, нужно обходить весь стек серверов сверху вниз.

Пример:

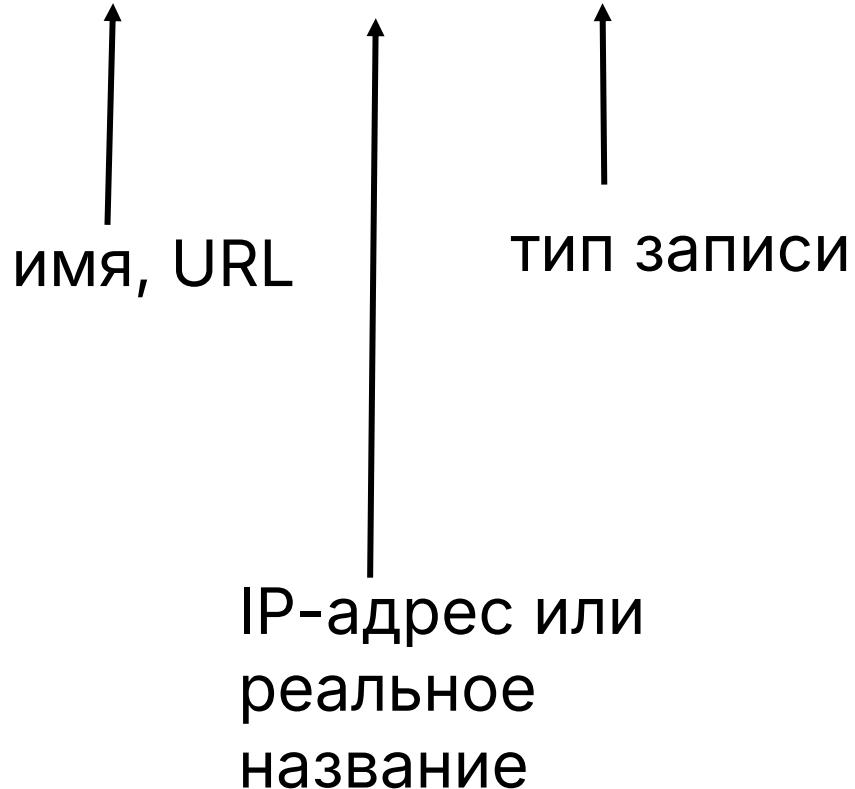
1. Из МФТИ хотят увидеть сайт ФББ. Запрос в локальный DNS.
2. В локальном DNS нет ФББ, смотрим в root DNS.
- 3-4. root направляет в .ru DNS.
- 5-6. .ru DNS направляет в .msu.ru DNS.
- 7-8. В .msu.ru DNS нашли fbb.msu.ru!



DNS-записи

DNS хранит БД вида:

(name, value, type, time-to-live)



время жизни
в кэше

Примеры:

(bioinf.fbb.msu.ru, 93.180.63.229, A, TTL)

обычная запись

(www.ibm.com,

outer-global-dual.ibmcom-tls12.edgekey.net,

CNAME,

TTL)

алиас сложного имени

(ibm.com,dns2.p05.nsone.net,NS,TTL)

DNS-сервер

Просмотр записей: `dig URL type`

Сокеты: идентификация процессов

Любая ОС требует знания о том, от какого к какому процессу нужно передавать информацию. Адрес процесса в интернете - IP-адрес хоста и порт на машине.

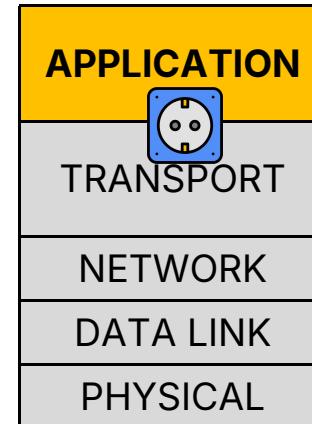
Эта комбинация - сокет. Через сокеты процессы отправляют и получают пакеты.

IP-адрес - адрес устройства в интернете

Порт - число-индекс в компьютере, по которому определяется сокет
У одного **процесса** м.б. несколько **сокетов** (вкладки браузера)

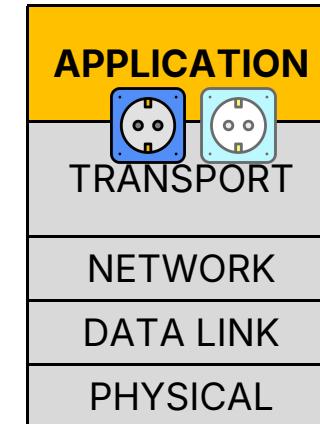


192.168.1.1



порт
51243

порт
443
80

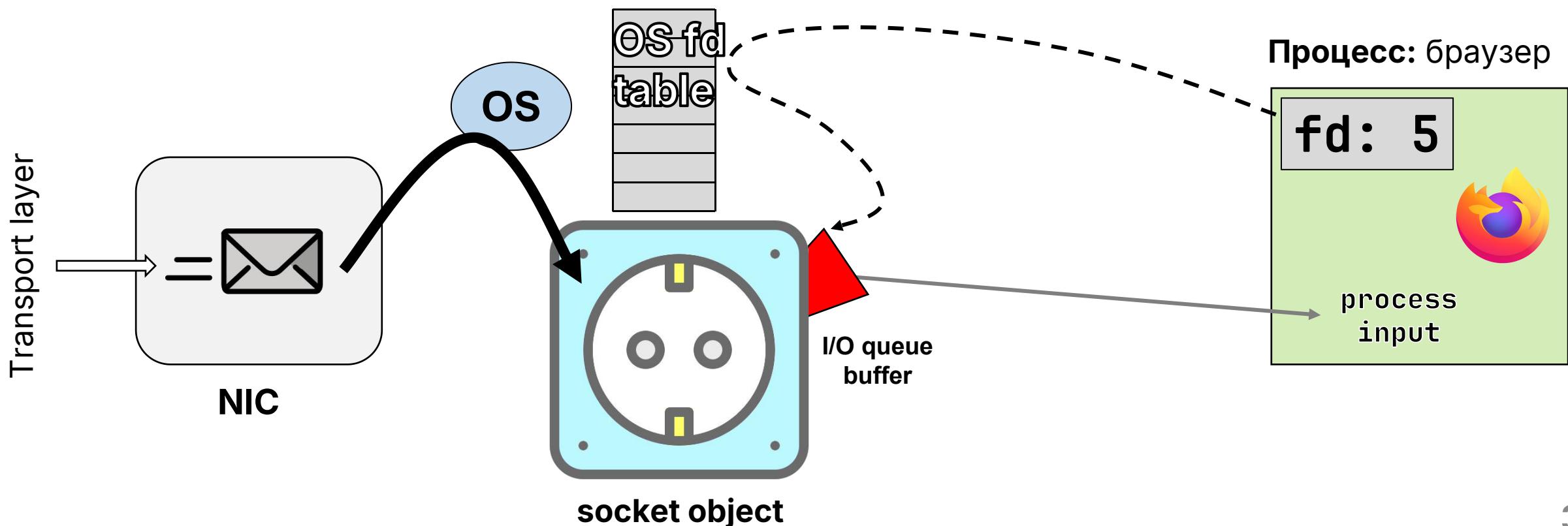


233.86.34.17



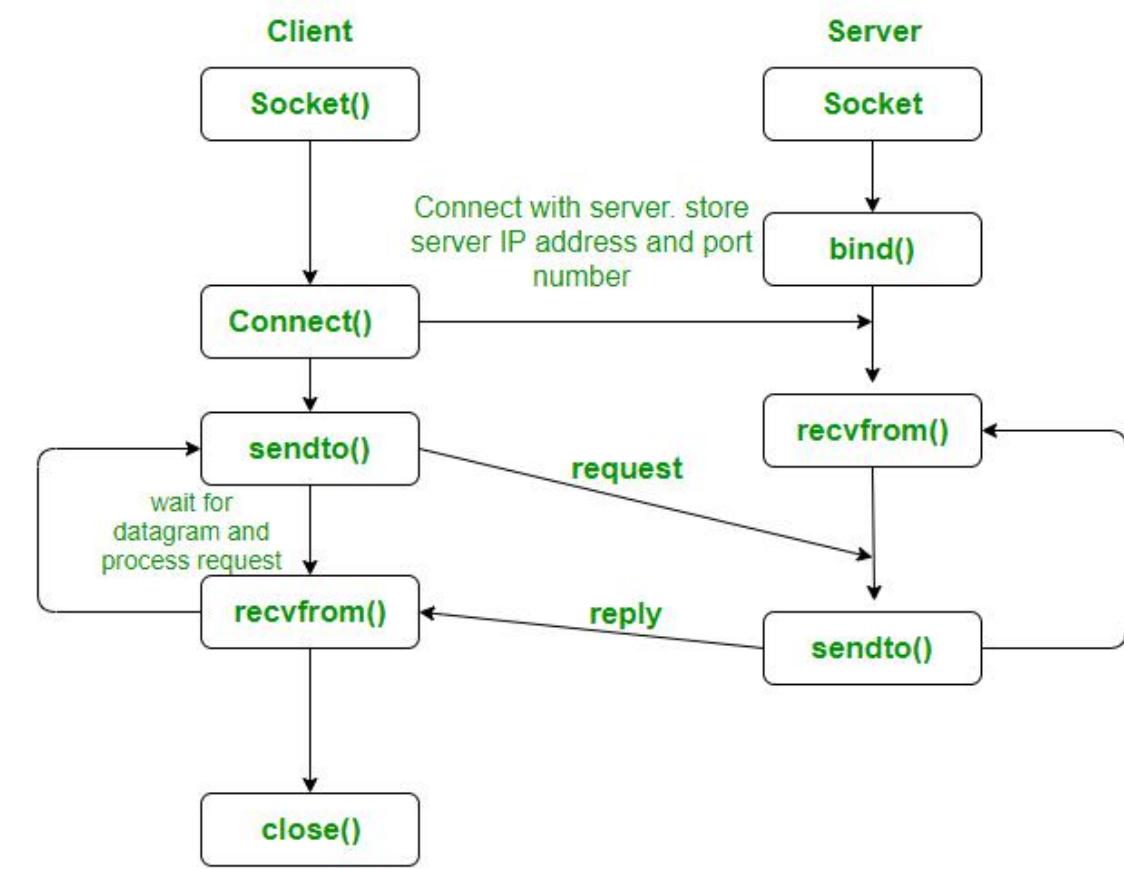
Сокет, ОС, ФД и процесс

Пакет поступает на карту сетевого интерфейса, где ОС распределяет его на конкретный сокет, определяемый через пару (IP, порт). Нужный процесс приложения имеет ФД, ссылающийся на сокет, чтобы читать оттуда данные!



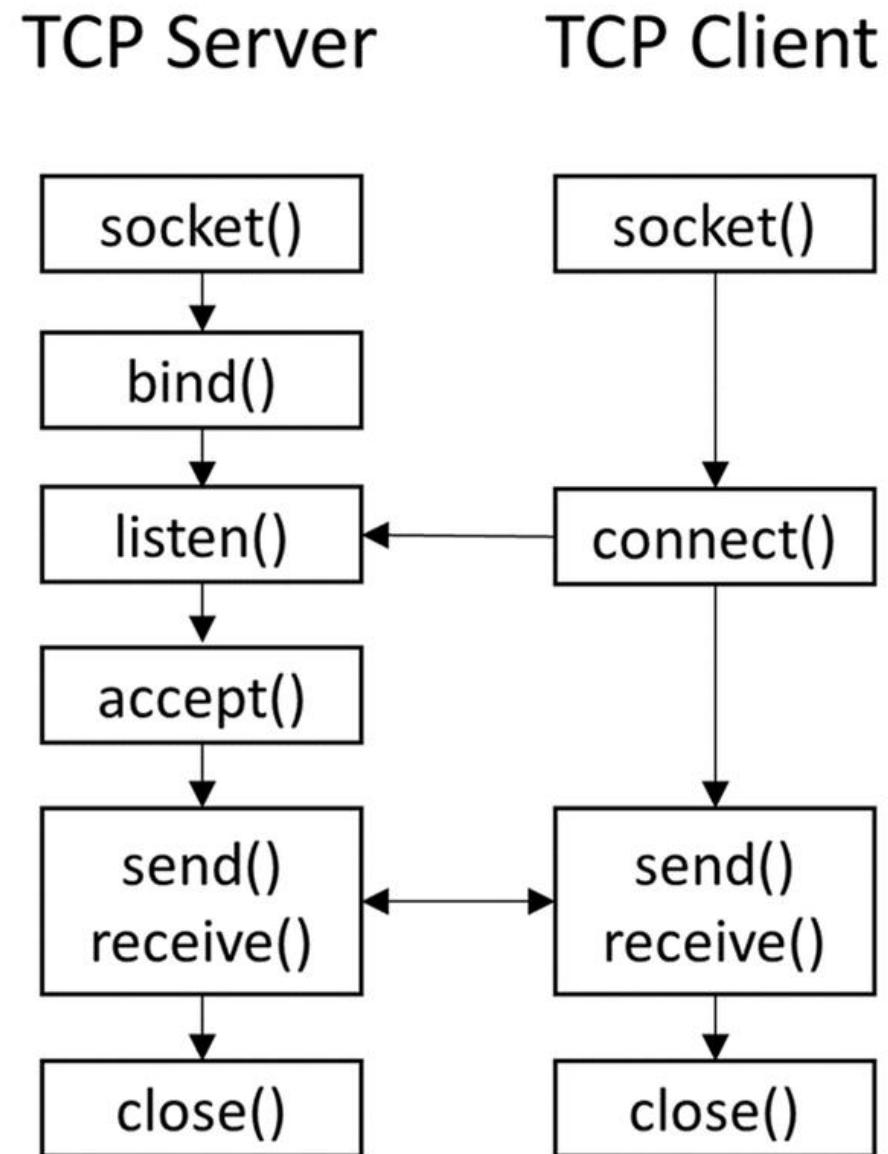
Сокет TCP и сокет UDP

- UDP - протокол, не требующий соединения, ему не важно, есть сервер или нет. Протокол отправляет данные по адресу (IP, порт), более ничего не обрабатывая.
- TCP - протокол, ориентированный на постоянное соединение. Есть исходный сокет встречи на известном порте. При соединении создаётся новый сокет, по которому с конкретным клиентом идёт взаимодействие.



Сокет TCP и сокет UDP

- UDP - протокол, не требующий соединения, ему не важно, есть сервер или нет. Протокол отправляет данные по адресу (IP, порт), более ничего не обрабатывая.
- **TCP – протокол, ориентированный на постоянное соединение. Есть исходный сокет встречи на известном порте. При соединении создаётся новый сокет, по которому с конкретным клиентом идёт взаимодействие.**



Написание сокетов TCP: сервер

сокет интернета с протоколом TCP!

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET, SOCK_STREAM)
serverSocket.bind(('',serverPort))
serverSocket.listen(1)
print('The server is ready to receive')
while True:
    connectionSocket, addr = serverSocket.accept()
    sentence = connectionSocket.recv(1024).decode()
    capitalizedSentence = sentence.upper()
    connectionSocket.send(capitalizedSentence.encode())
    connectionSocket.close()
```

привязка и слушание
запросов в сокете встречи

создание сокета соединения и
получение данных от клиента

возврат обработки и
закрытие соединения

proto-eventloop: реализация always-on хоста.

Написание сокетов TCP: клиент

сокет интернета с протоколом TCP!

```
from socket import *
serverName = 'servername'
serverPort = 12000
clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect((serverName, serverPort))
sentence = input('Input lowercase sentence: ')
clientSocket.send(sentence.encode())
modifiedSentence = clientSocket.recv(1024)
print ('From Server:', modifiedSentence.decode())
clientSocket.close()
```

соединение с сервером и
ввод данных

отправка данных на сервер

получение и
декодирование ответа
сервера

В ДЗ: разобраться с UDP-сокетами
(они проще), первично разобраться
в async-сервере + `.settimeout()`

Стандартные порты протоколов

- 80 - HTTP
- 443 - HTTPS over TLS/SSL
- 22 - SSH
- 53 - DNS
- 25 - SMTP
- 20/21 - FTP
- 67/68 - DHCP

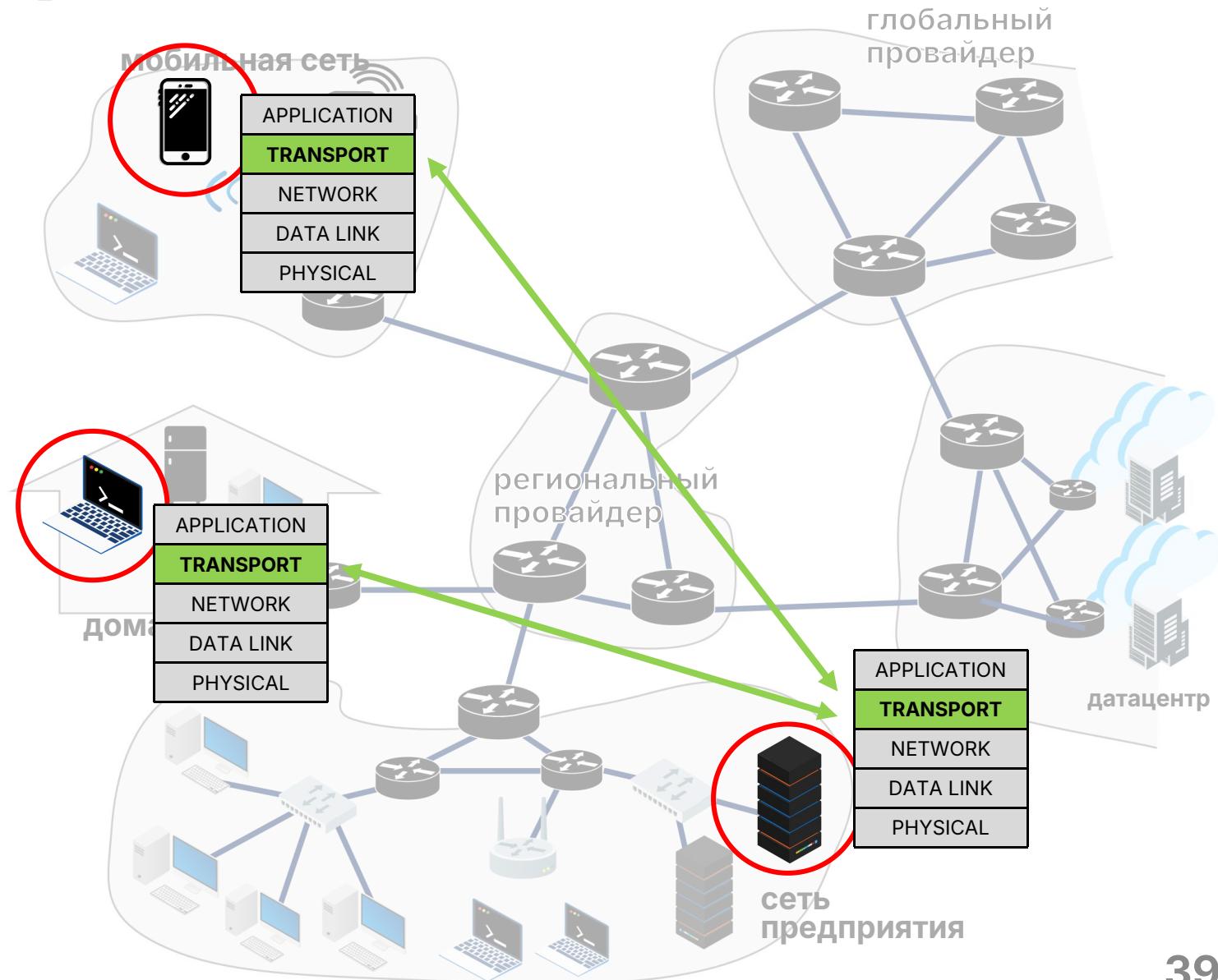


У стандартного UNIX ПК 65536 (2^{16}) портов. Из них:

- 0-1023 под привилегированные процессы,
- 1024-49151 под специальные приложения (e.g., PostgreSQL на порте 5432),
- остальные – временные (эфемерные) порты, свободные всем.

Транспортный уровень

- Сегментация сообщений в пакеты
- (де)мультиплексирование
- КОММУНИКАЦИЯ ХОСТОВ И ПРОЦЕССОВ



Transport vs. network layer

Службы транспортного уровня определяют, в какой процесс отправить данные, исходя из данных порта.

Службы сетевого уровня определяют, в какой хост отправить данные.

Аналогия: почта - сетевой уровень, ваш сосед по комнате, вручающий вам ваше письмо, - транспортный.



Два протокола: TCP и UDP

Transmission Control Protocol

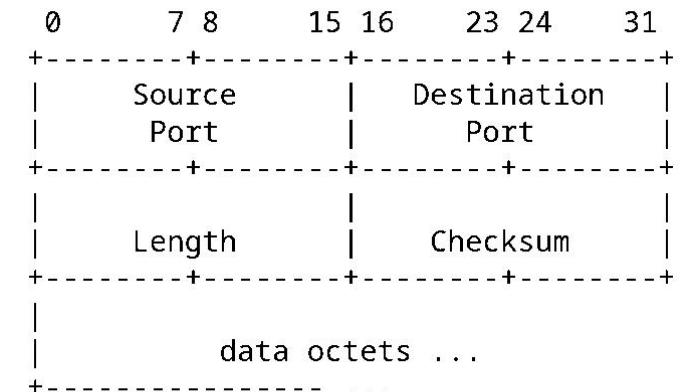
- надёжная доставка
- контроль перегрузок
- контроль потока
- ориентирован на установление долгосрочного соединения

User DProtocol

- ненадёжная доставка
- доставка без гарантии порядка пакетов
- “best-effort”

DNS, SNMP, HTTP/3

Ни один из протоколов не
гарантирует времени
конкретного времени задержки
или ширины канала!



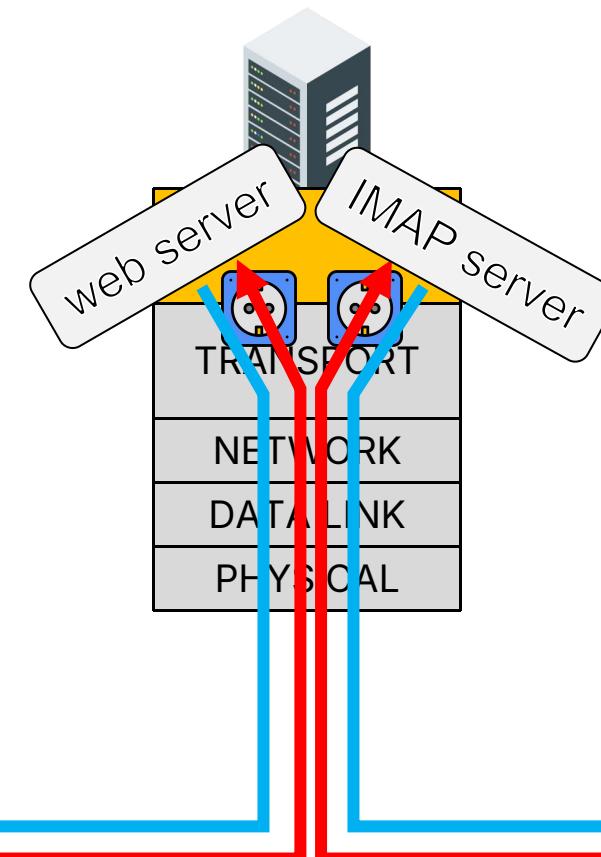
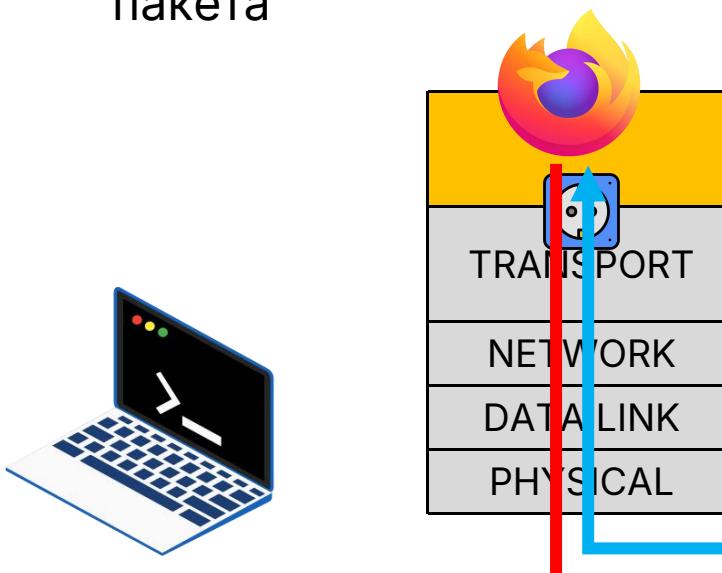
User Datagram Header Format

Мультиплексирование

Протоколы транспортного уровня осуществляют слияние и разделение потока данных в хостах от и в разные процессы по паре (IP, port) - по сокету.

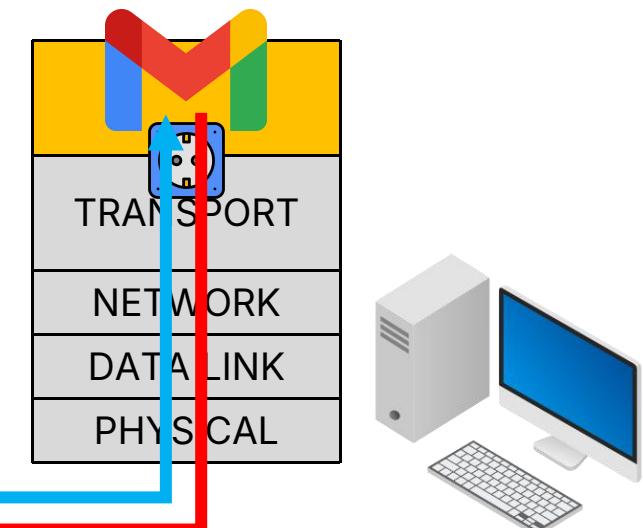
Мультиплекс:

Собрать данные со всех сокетов и создать заголовок пакета



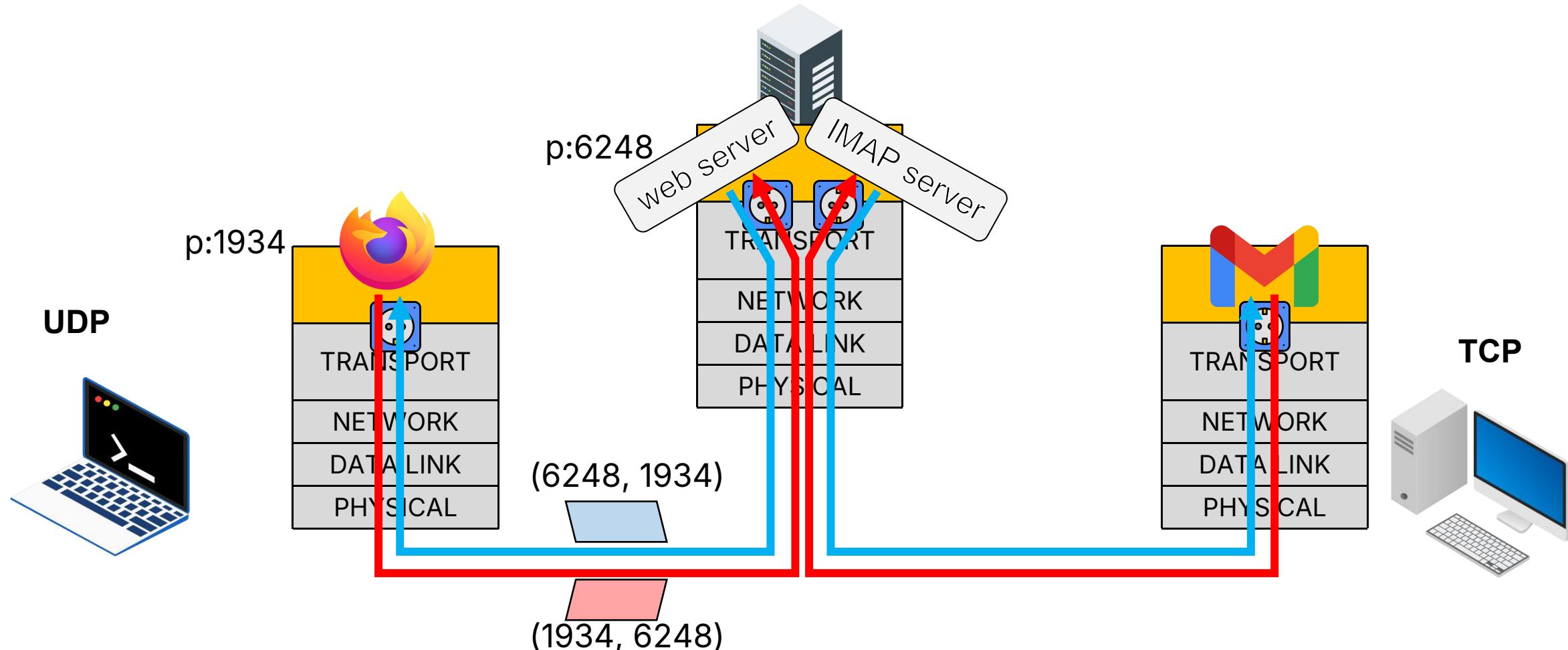
Демультиплекс:

используя заголовок доставить к нужным сокетам данные



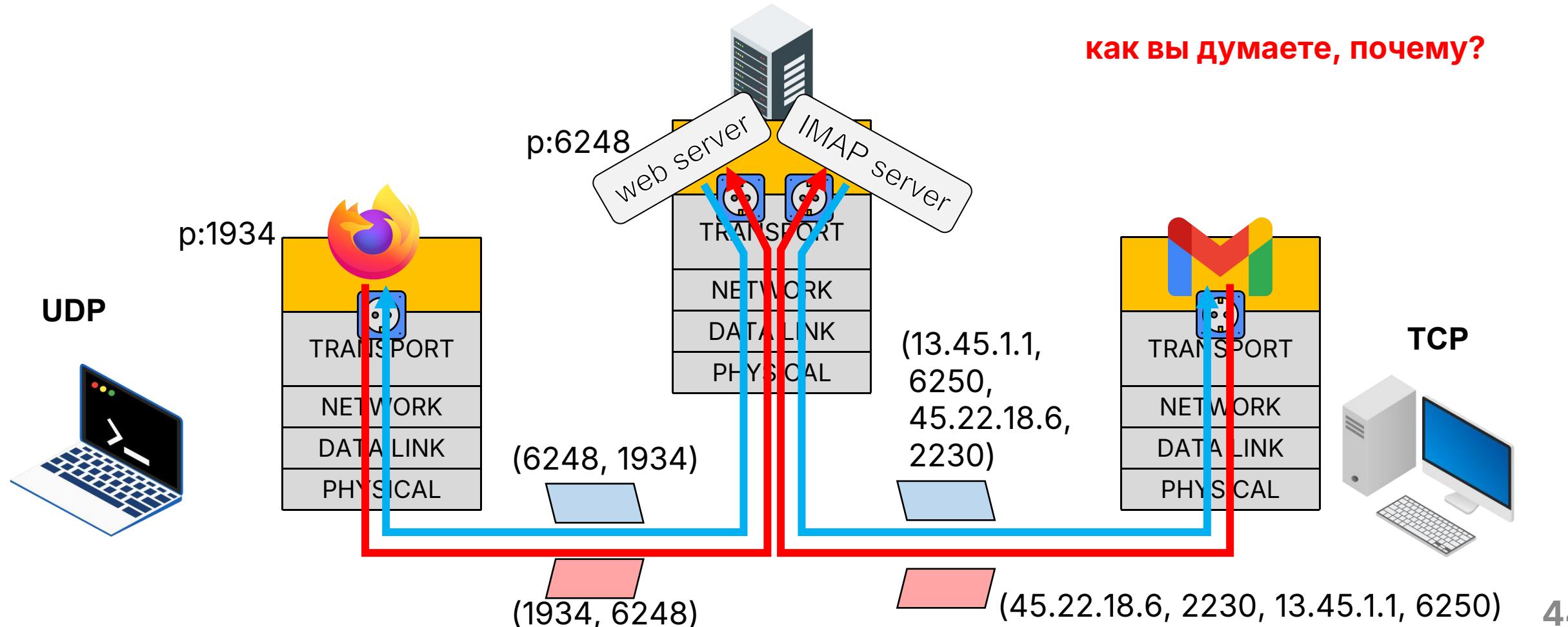
Мультиплексирование

В UDP для взаимного обмена требуется только пара (src_port, dest_port) для (де)мультиплексирования



Мультиплексирование

В TCP для взаимного обмена требуется четверка
(src_IP, src_port, dest_IP, dest_port) для (де)мультиплексирования



Как выглядит надёжная передача?

надёжная
односторонняя
передача

=

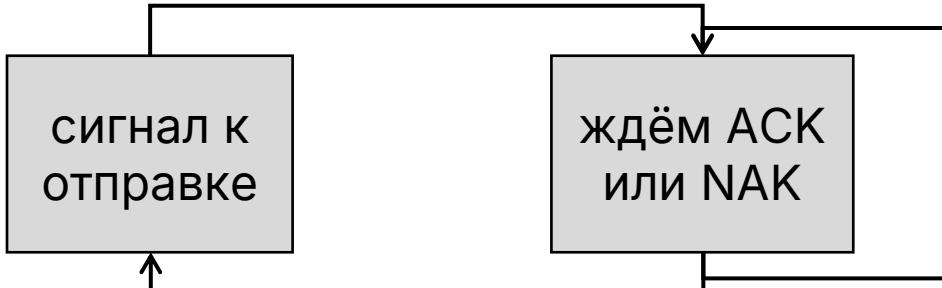
ненадёжная
двусторонняя
передача

Если нет потери пакетов:

Отправка:

ACK - пакет в порядке, NAK - повреждён

```
snpkt=make_pkt(data,checksum)  
send(sndpkt)
```



ACK:pass

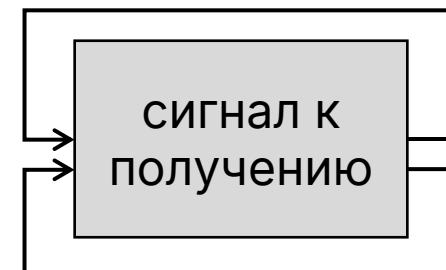
Что если ACK/NAK повреждён?

Отправитель ещё раз отправит пакет, а ACK-NAK нужно нумеровать

Получение:

контрольная сумма и возврат ACK/NAK

```
good checksum:  
extract_deliver(pkt)  
send(ACK)
```



```
bad checksum:  
send(NAK)
```

Модификация: ломаные ACK/NAK

Отправка:

```
    sndpkt=make_pkt(0,data,checksum)  
    send(sndpkt)
```

сигнал к
отправке

ждём ACK
или NAK 0

NAK/corrupt:
send(sndpkt)

ACK 1:
pass

ACK 0: pass

ждём ACK
или NAK 1

сигнал к
отправке

NAK/corrupt:
send(sndpkt)

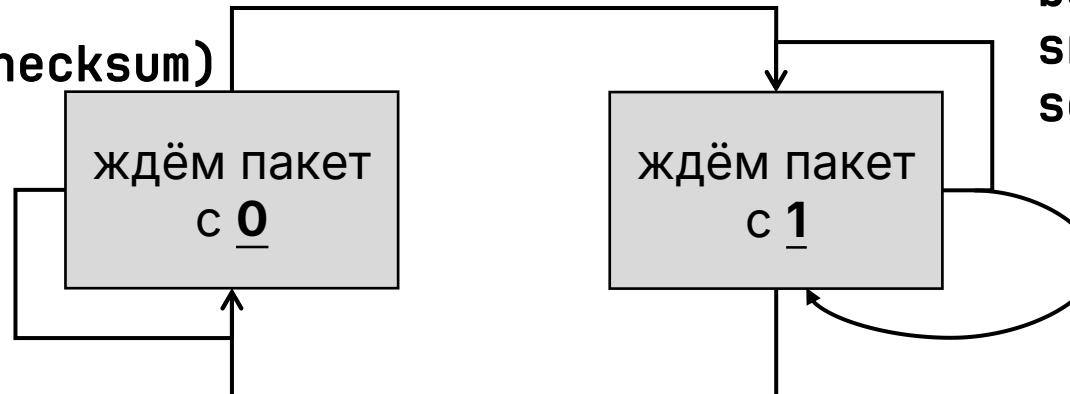
```
    sndpkt=make_pkt(1,data,checksum)  
    send(sndpkt)
```

Модификация: ломаные ACK/NAK

Получение:

```
good checksum + 0:  
extract_deliver(pkt)  
sndpkt=make_pkt(0,ACK,checksum)  
send(sndpkt)
```

```
good checksum + 1:  
sndpkt=  
make_pkt(1,ACK,checksum)  
send(sndpkt)
```



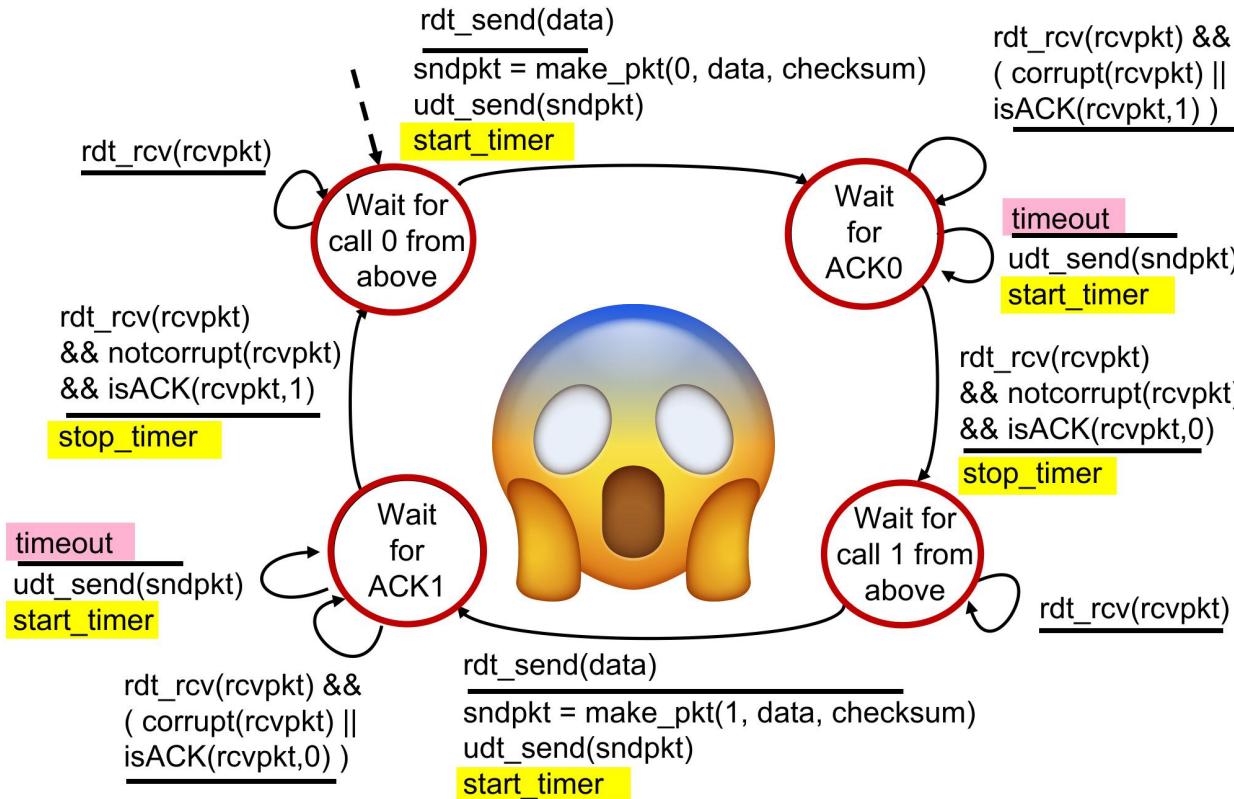
bad checksum:
sndpkt=make_pkt(0,NAK,checksum)
send(sndpkt)

```
good checksum + 0:  
sndpkt=make_pkt(0,ACK,checksum)  
send(sndpkt)
```

```
good checksum + 1:  
extract_deliver(pkt)  
sndpkt=make_pkt(1,ACK,checksum)  
send(sndpkt)
```

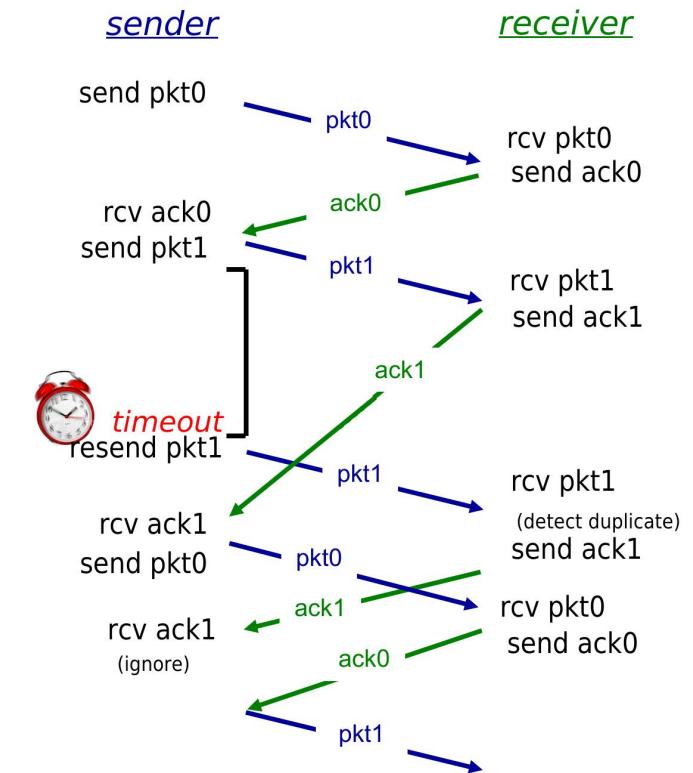
Как выглядит надёжная передача?

Самое близкое к TCP: потеря пакетов - таймауты без ACK/NAK. Проверка дубликатов.



Включает дополнительные протоколы типа Go-Back-N, Selective Repeat, и т.д.

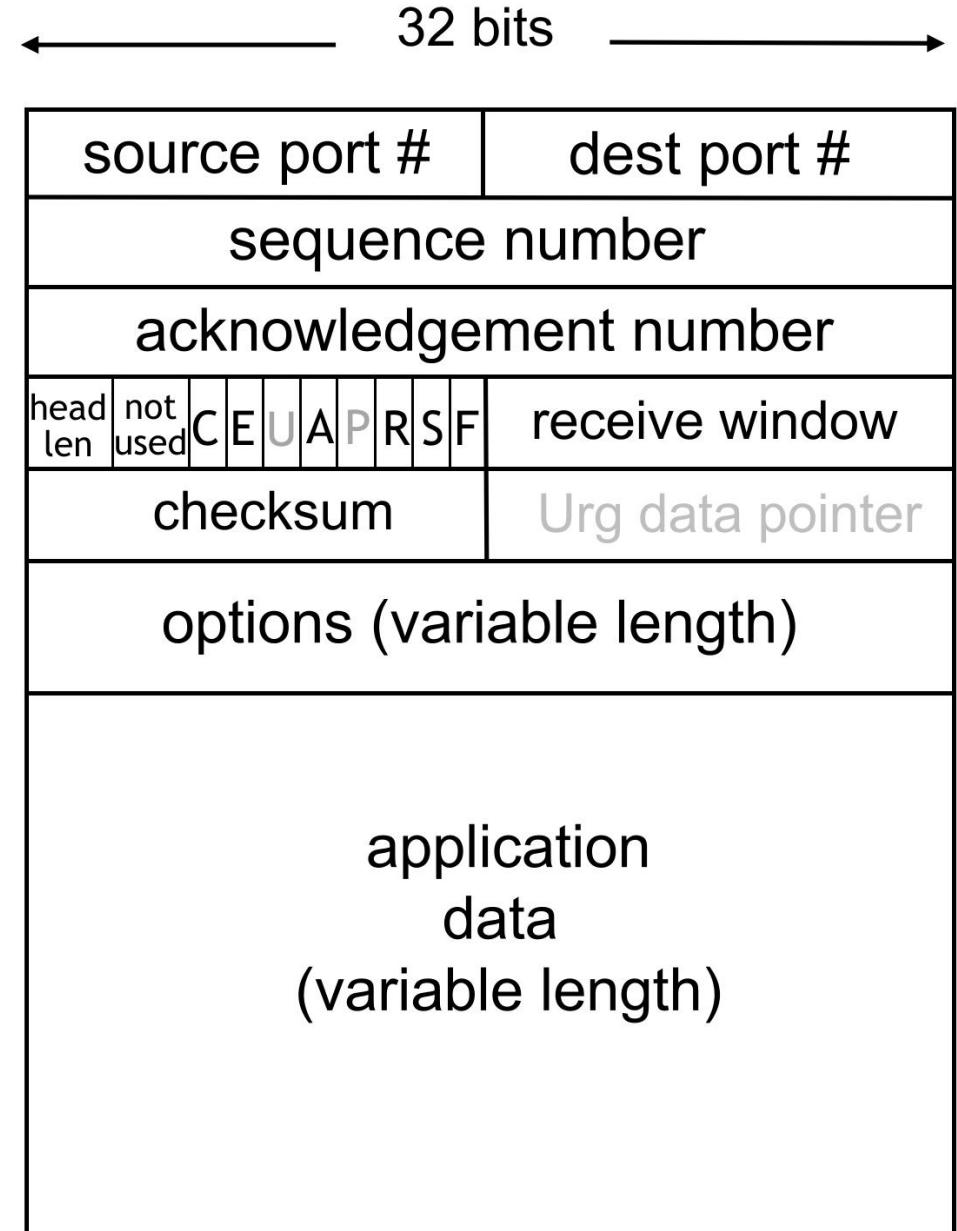
Уже сильно более сложно.
Необязательно, но можно разобраться



Итог: TCP

Механизмы обеспечения надёжной передачи данных:

- Контрольные суммы
- ACK / NAK сегменты
- Повторные отправки
- Нумерация пакетов для избегания дубликатов
- Таймауты



TCP сегмент

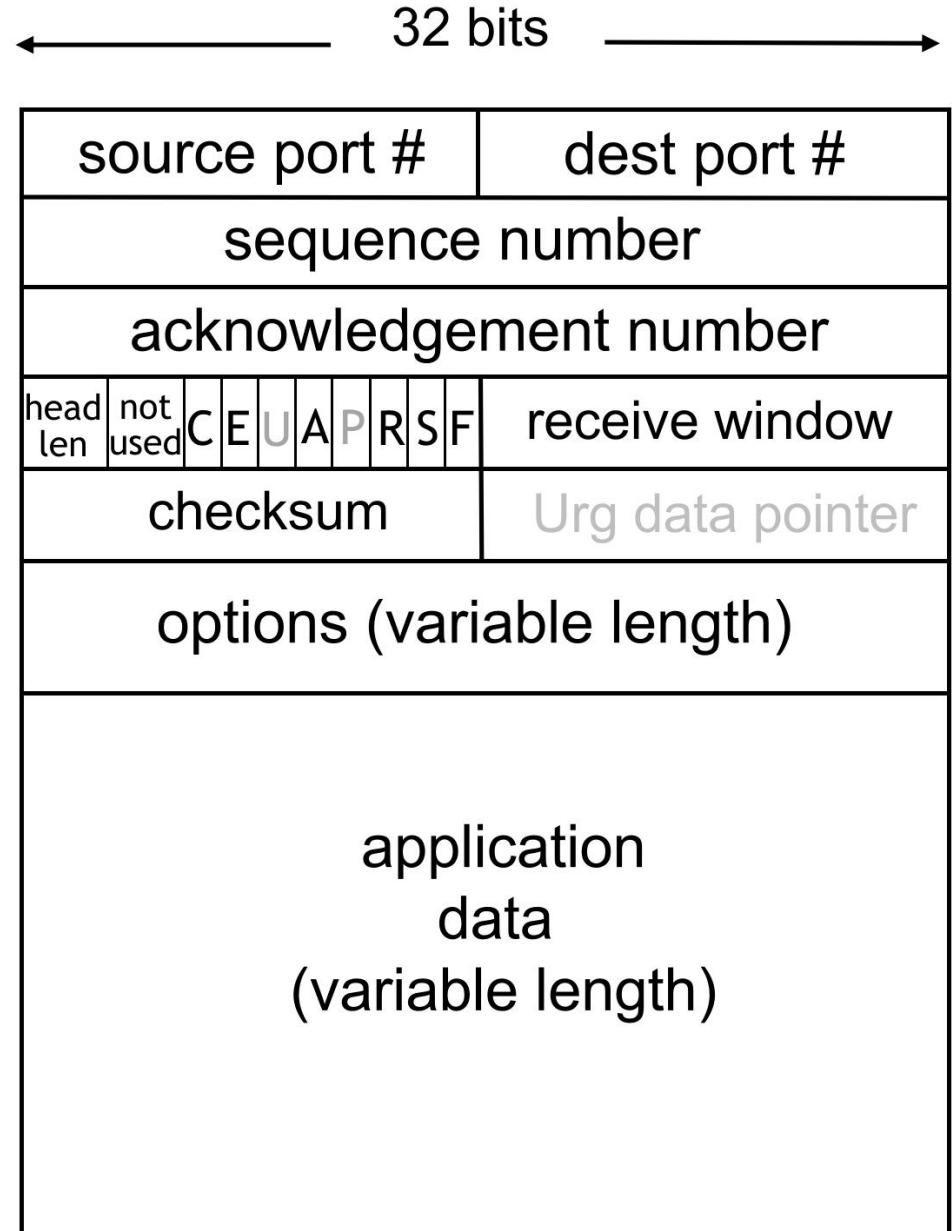
A - ACK / NAK

R/S/F (RST/SYN/FIN) -
технические биты начала и
конца соединения

C/E - нагрузка на сеть

Receive window - размер
пакетов на принимающей
сторону

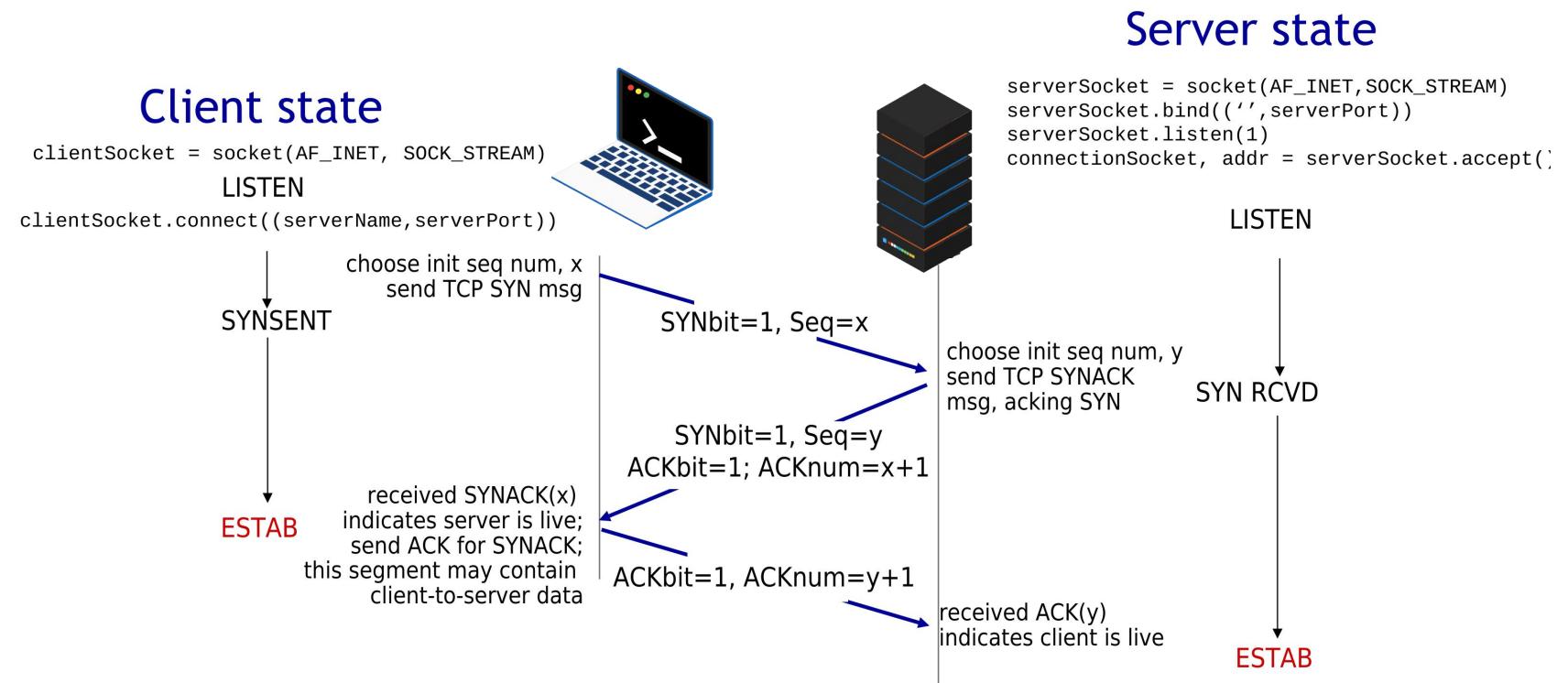
Header length - длина заголовка,
где начало данных



Механизм TCP

1. Тройное рукопожатие
2. Передача данных
3. Окончание соединения

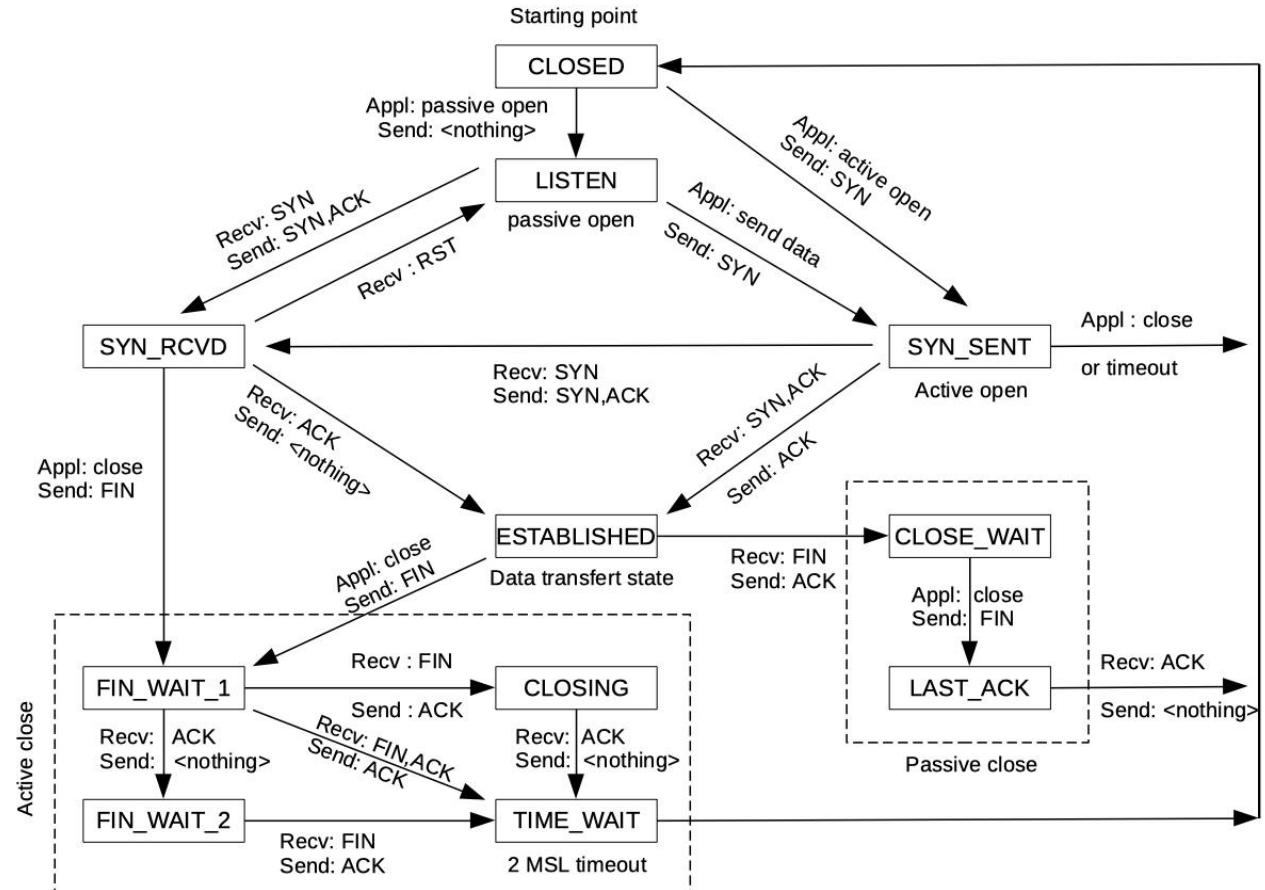
Передача битов SYN, SYNACK,
ACK



Механизм TCP

1. Тройное рукопожатие
2. Передача данных
3. Окончание соединения

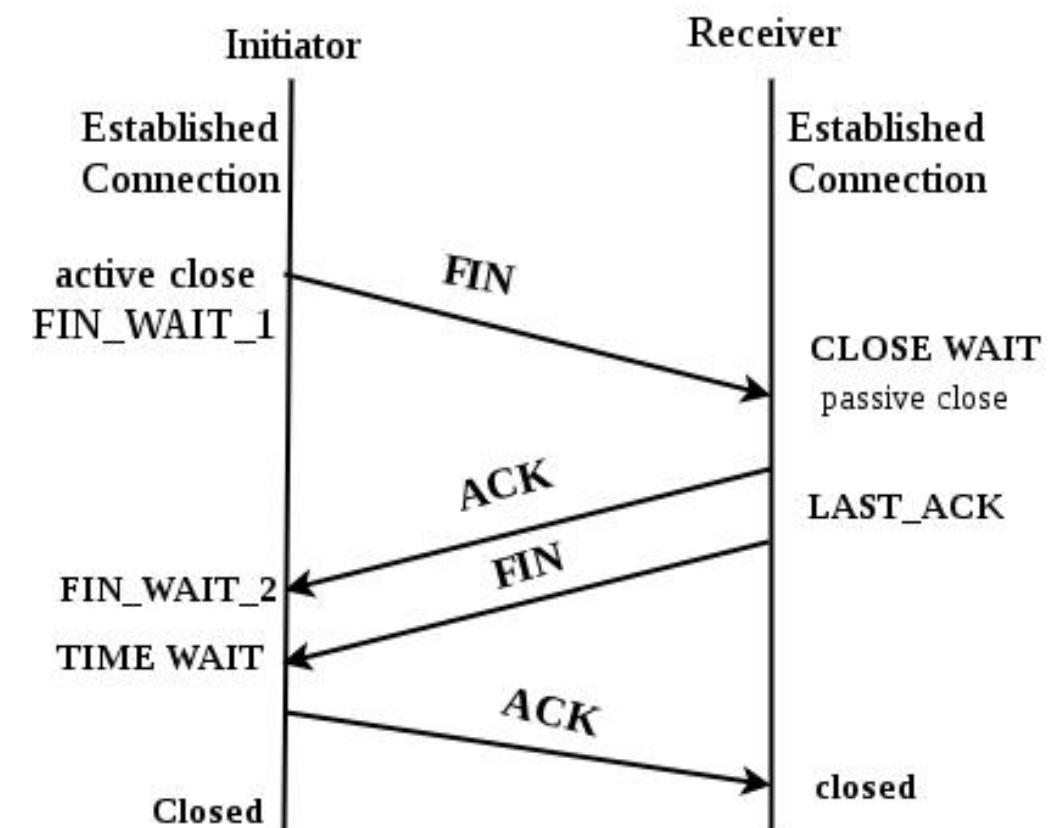
Огромная
страшная
диаграмма
состояний



Механизм TCP

1. Тройное рукопожатие
2. Передача данных
3. **Окончание соединения**
отправка бита FIN

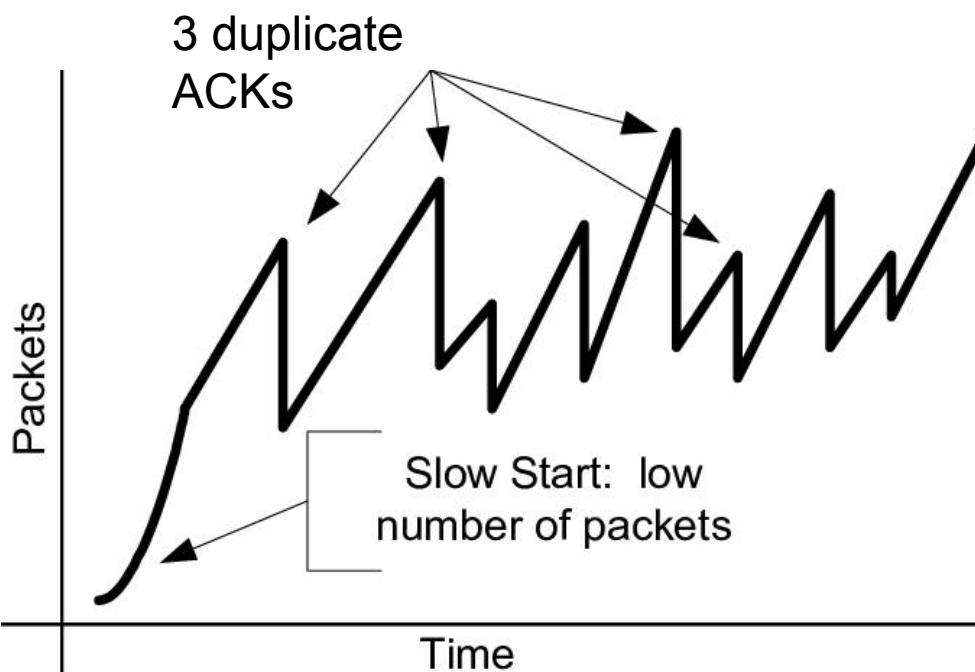
Сейчас HTTP/3 использует QUIC -
UDP с плюшками TCP



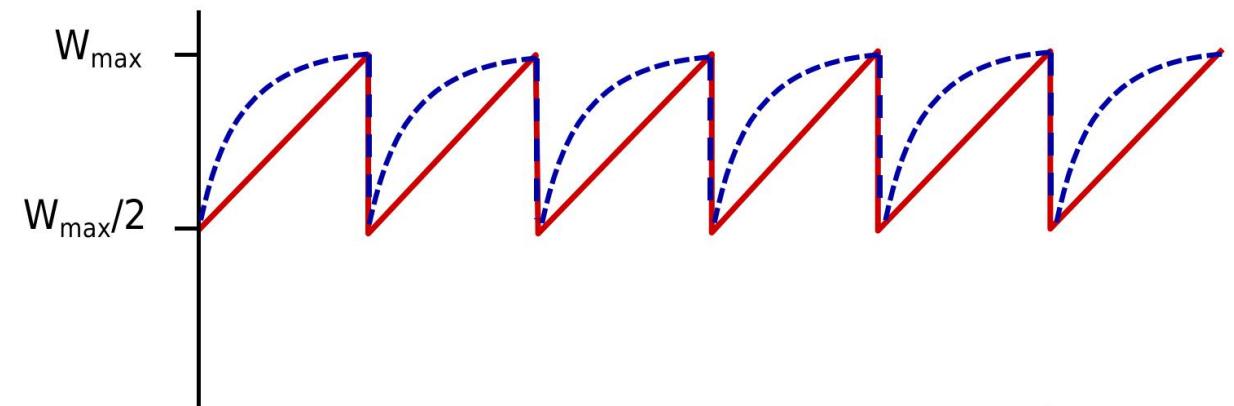
Congestion control. AIMD.

TCP выводит нагрузку на сеть из того, сколько пакетов теряется, и какой таймаут.

Отправитель увеличивает скорость отправки до момента, пока не наблюдается перегрузка. После перегрузки резко снижает: **Additive Increase, Multiplicative Decrease**

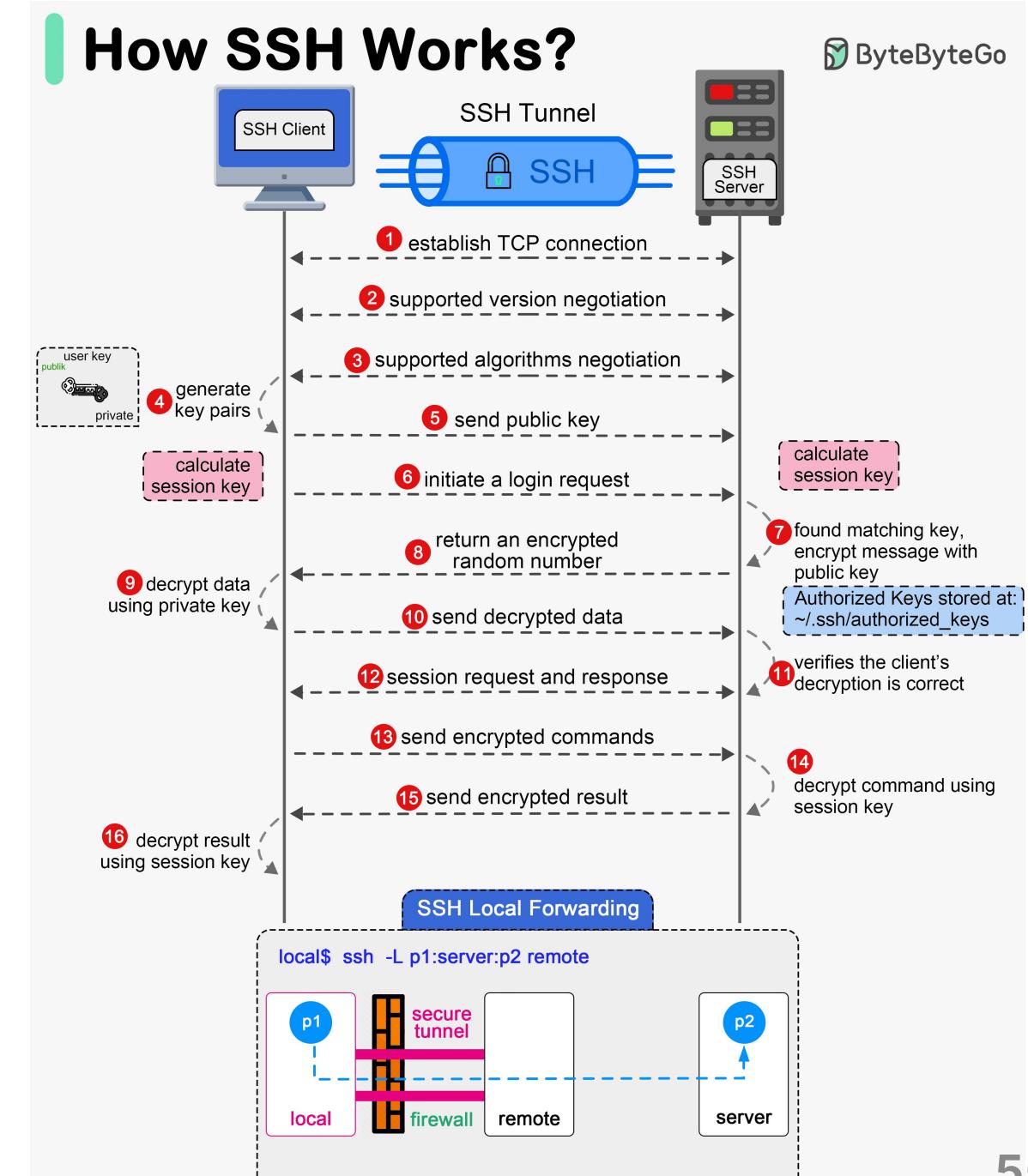


Более того, AIMD ведёт ширину канала к "честной"



ssh

Удалённый доступ = TCP
соединение с выводом букв.
Поверх добавляется слой
TLS+SSL - безопасная версия
TCP.



Пример: jupyter на сервере

```
user@remote.server:~$ jupyter-lab --no-browser --port=4559
```

```
user@local-PC:~$ ssh -L 8888:localhost:4559 user@remote.server
```

Открывается локальный порт 8888 на local-PC и все данные пересылаются через SSH на порт 4559 на удалённом сервере remote.server



видит веб-сервер
на localhost:8888

пересыпает SSH с 8888
на remote.server:4559

TCP-взаимодействие сервера
Jupyter и SSH демона