

HTTP3 + GO: чего ожидать?



> go, 22 мая 2024

Нина Пакшина <

TECH
Meetup #5 X



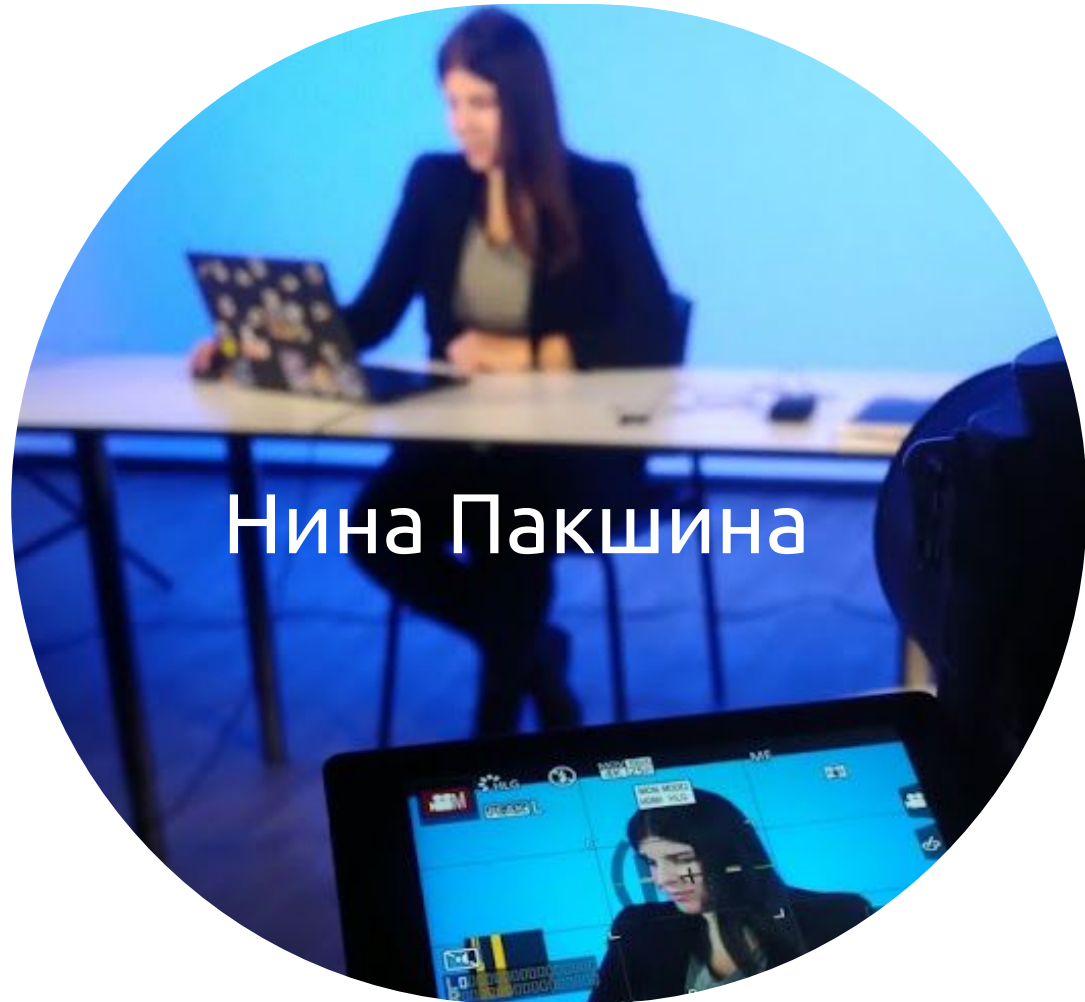
Go-разработчик

13+ лет в IT

Организатор
Московского клуба
программистов

ПЛК ИБ
Python Go

Ninucium @medium
Ninako @Habr
PakshNina @github



План доклада

Часть 1. Предубеждения

Часть 2. Зачем использовать HTTP/3?

Часть 3. Подводные камни





Часть 1. Предубеждения

HTTP/2 и HTTP/3 отличаются кардинально

Спойлер: нет

HTTP 0.9 (1991)

Сетевой уровень: IP
Транспорт: TCP
Методы: GET

HTTP 1.0 (1996)

- + Заголовки запросов
- + Коды статусов
- + Content type
- + Методы POST HEAD
- + HTTP-version поле

HTTP 1.1 (1997)

- + Заголовок Host
- + Keep-alive
- + 100 Continue
- + PUT PATCH DELETE CONNECT TRACE OPTIONS

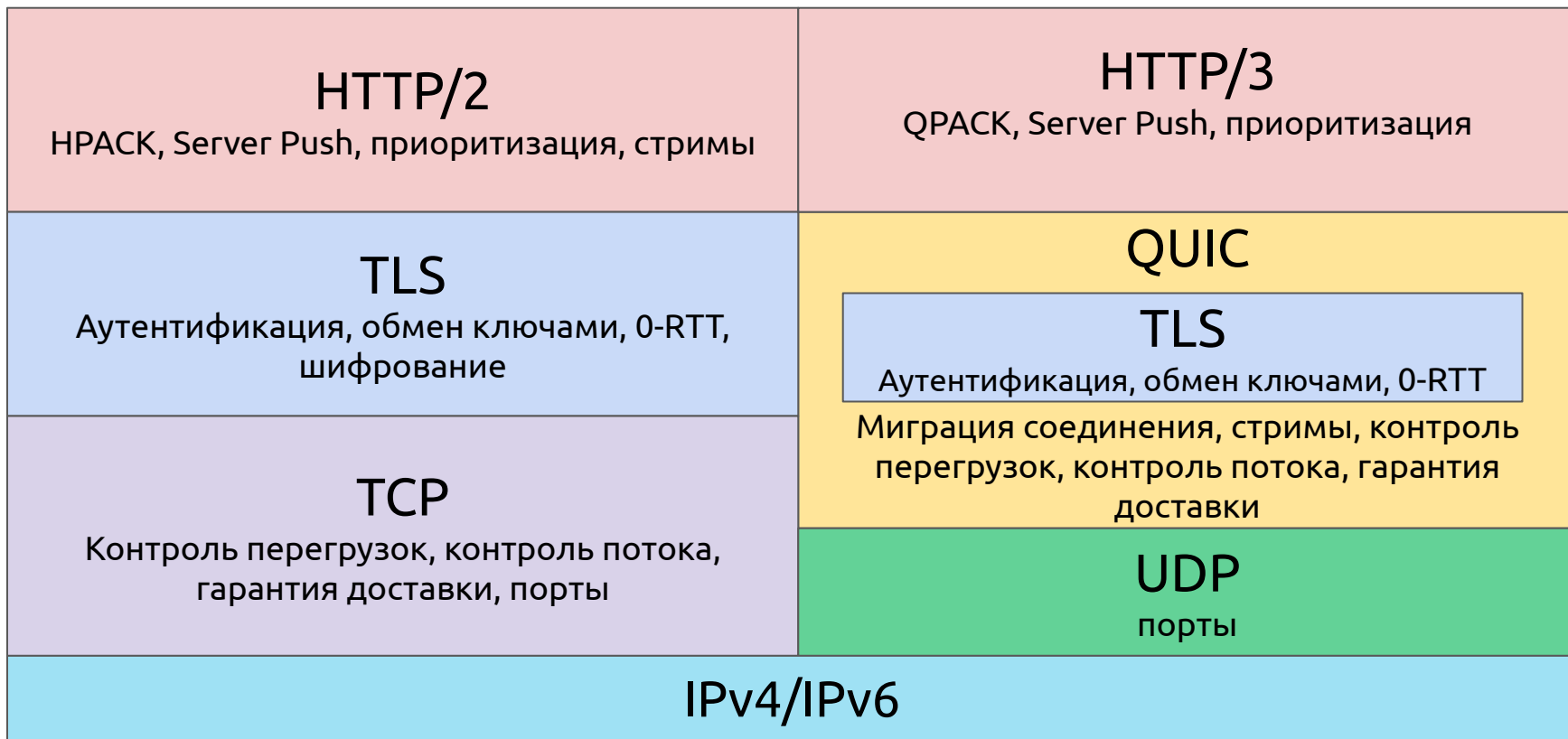
HTTP 2.0 (2015)

- + Стримы
- + Приоритизация
- + Сжатие
- + Сервер Push

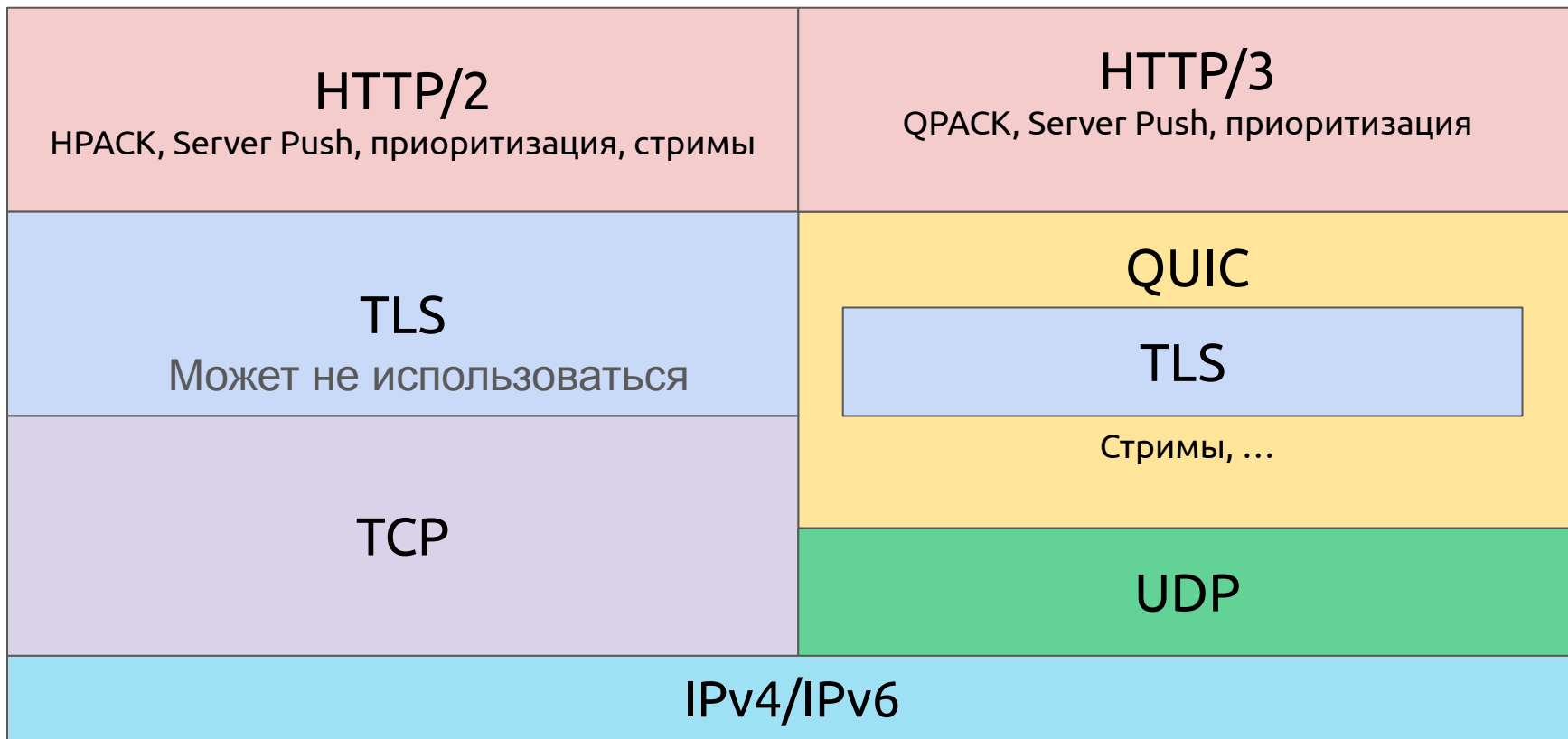
HTTP 3.0 (2020)

Транспорт:
UDP/QUIC

HTTP/3 - это HTTP/2 поверх другого транспорта



HTTP/3 - это HTTP/2 поверх другого транспорта



Реализация сервера HTTP/3 на Go

```
mux := http.NewServeMux()
mux.HandleFunc("/", mainHandle)
```

```
tlsConfig := &tls.Config{...}
quicConfig := &quic.Config{...}
```

```
srv := http3.Server{
    Addr:          addr,
    TLSConfig:    tlsConfig,
    QuicConfig:    quicConfig,
    Handler:      mux,
}
```

```
if err = srv.ListenAndServeTLS("cert.pem", "key.pem"); err != nil {
    log.Error(err)
}
```

quic-go ver 0.42!

Реализация клиента HTTP/3 на Go

```
roundTripper := &http3.RoundTripper{
    TLSClientConfig: &tls.Config{...},
    QuicConfig:      &quic.Config{...},
}

client := http.Client{
    Transport: roundTripper,
}

resp, err := client.Get("https://127.0.0.1:9999/")
...
body, err := io.ReadAll(resp.Body)
if err != nil {...}
```

HTTP/3 поверх UDP
ПОЭТОМУ ОН НЕНАДЕЖЕН?

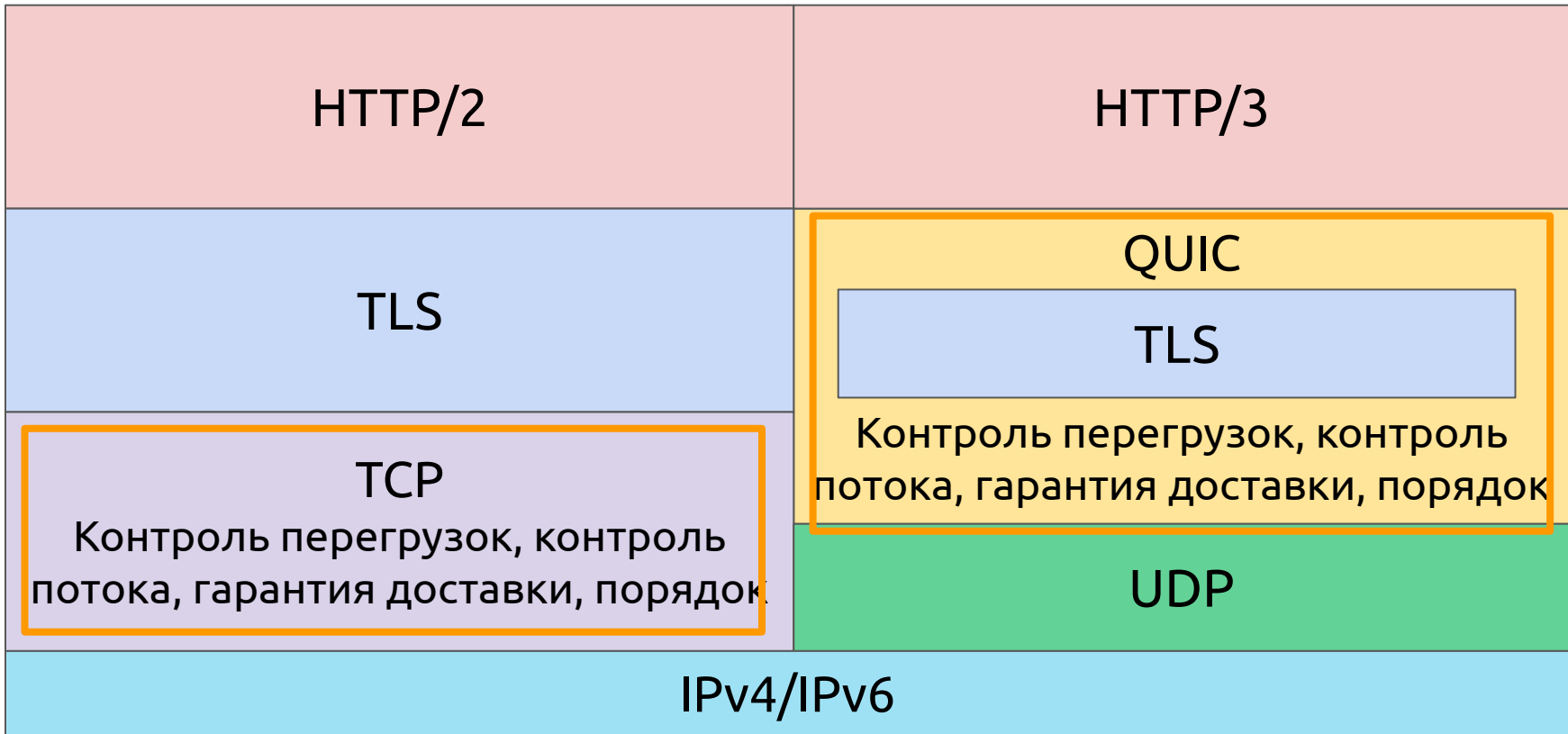
Спойлер: нет

UDP не устанавливает соединение

UDP не гарантирует доставку сообщений

UDP не гарантирует порядок сообщений

HTTP/3 ненадежный протокол?





QUIC дублирует
надежность TCP, но
с доработкой

Гарантия доставки и порядок

Подтверждение, что все пакеты были доставлены [ACKNOWLEDGEMENT](#)

Пересылка сообщений [RETRANSLATION](#)

Потерянные пакеты не передаются целиком: информация, содержащаяся в них, передается снова в новых пакетах и фреймах

Пакеты нумеруются



Подтверждение доставки

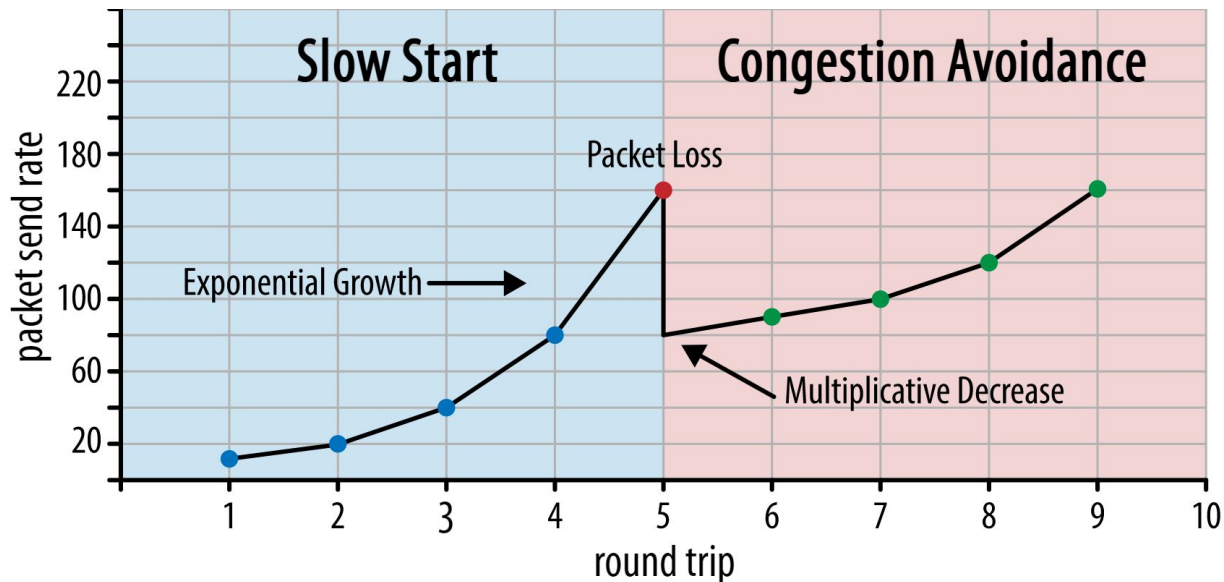
Повторяют возможности HTTP/2

- Подтверждение, что все пакеты были доставлены:
[ACKNOWLEDGEMENT](#)
- Клиент подтверждает все полученные пакеты, которые требуют подтверждения в рамках `max_ack_delay` времени



Контроль перегрузок

Поиск доступной полосы пропускания CONGESTION CONTROL





В UDP нет встроенного контроля перегрузок,
в отличие от TCP

В TCP контроль перегрузок стартует
медленно и это реализовано на уровне
транспортного протокола в ОС

В QUIC контроль перегрузок реализован
более эффективно

Контроль потока

Предельное количество данных, которое приемник готов принять на данном потоке или для всех

[FLOW CONTROL](#)

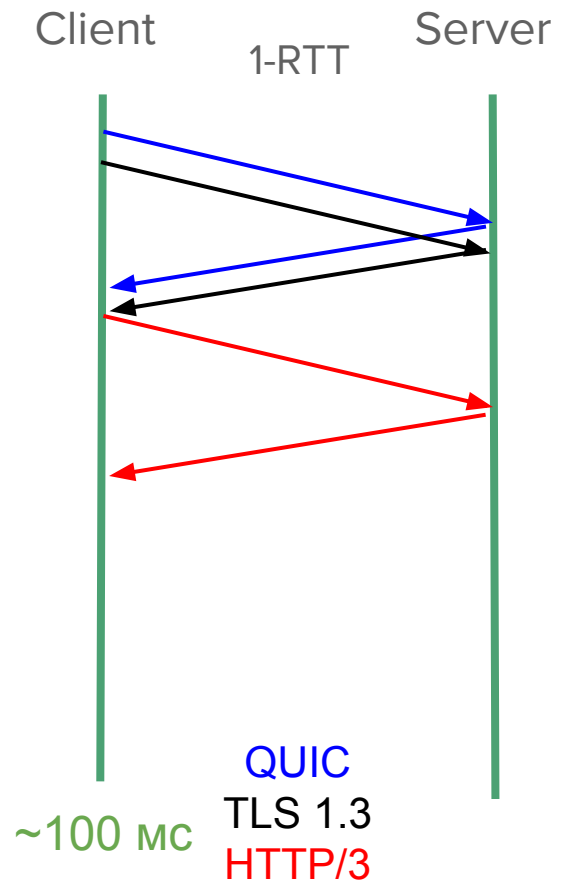
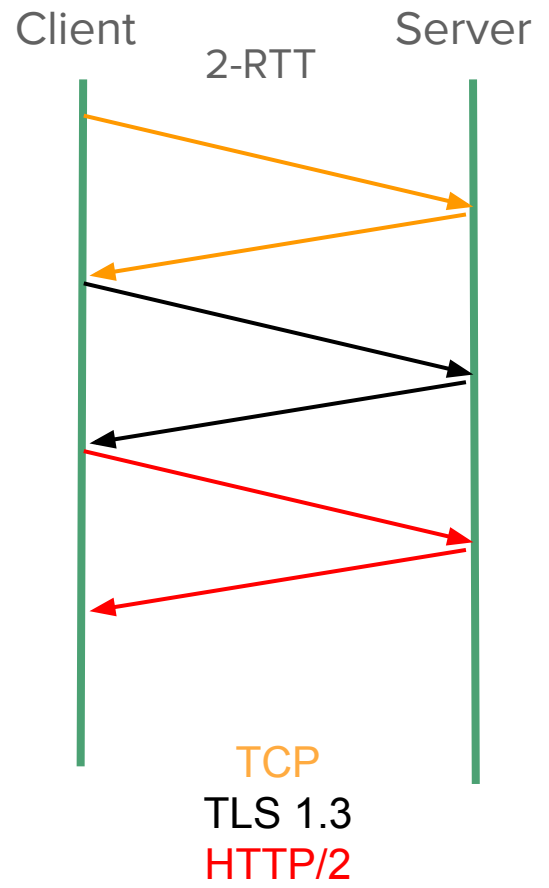
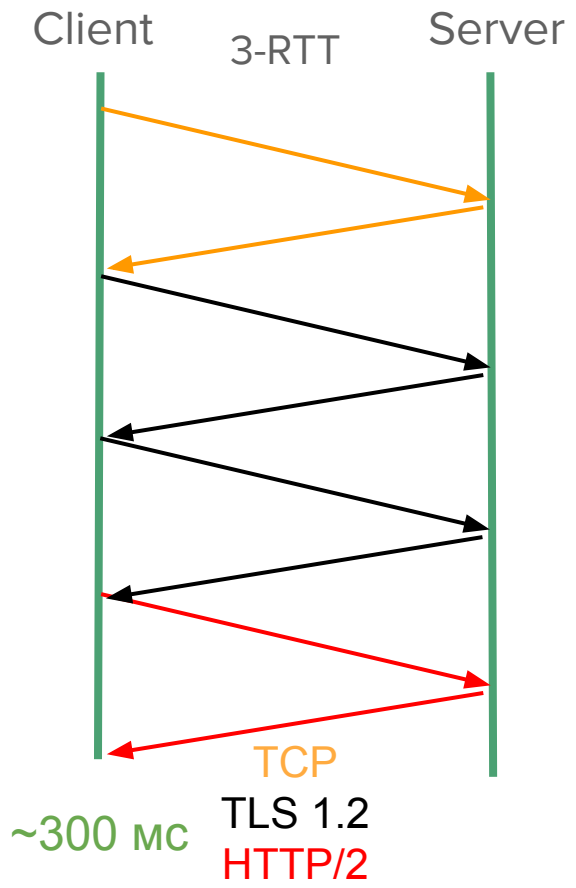


Значения по умолчанию

```
quic.Config{
    InitialStreamReceiveWindow:    (1 << 10) * 512, // 512 kb
    MaxStreamReceiveWindow:       6 * (1 << 20),   // 6Mb
    InitialConnectionReceiveWindow: (1 << 10) * 512, // 512 kb
    MaxConnectionReceiveWindow:   15 * (1 << 20),  // 15Mb
    AllowConnectionWindowIncrease: func(c quic.Connection, delta uint64) bool
    {
        return false
    },
}
```

HTTP/3 существенно
быстрее, чем HTTP/2

Спойлер: нет



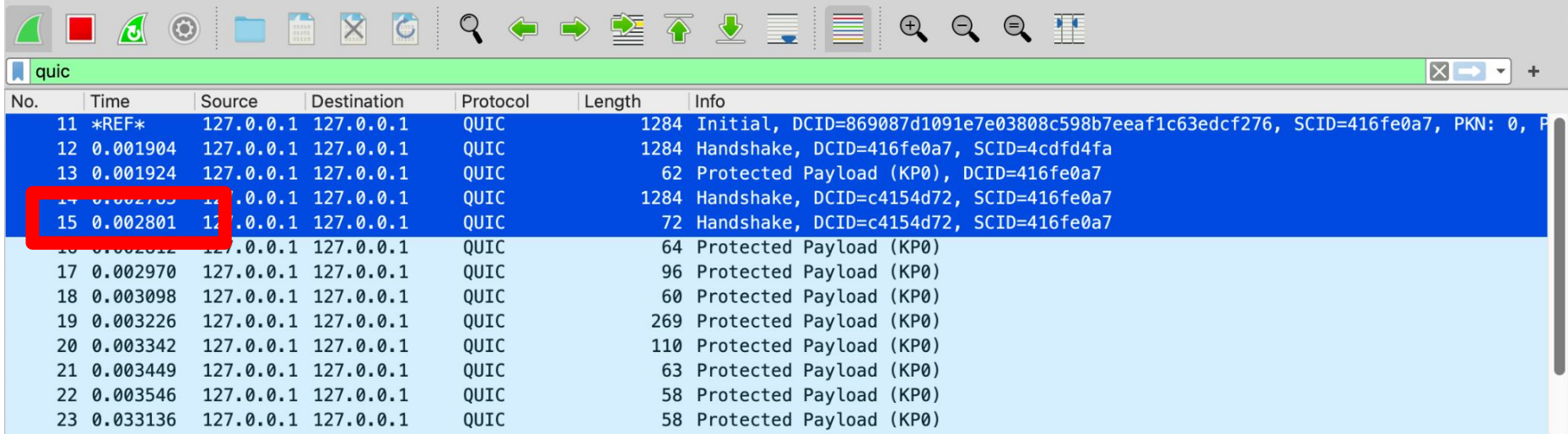
HTTP/2

Capturing from Loopback: lo0

tcp.port==9393

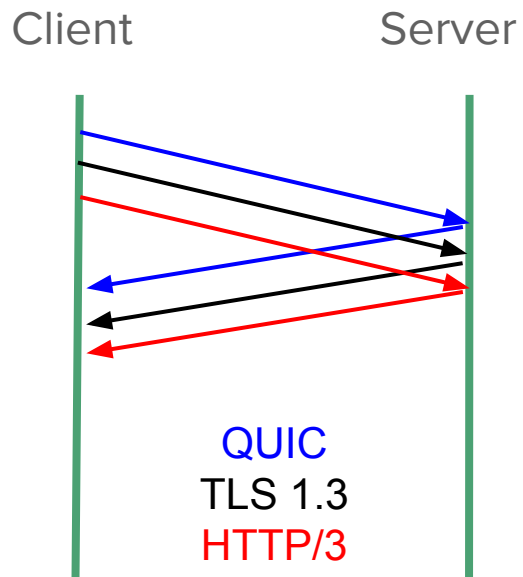
No.	Time	Source	Destination	Protocol	Length	Info	time de
15	*REF*	:::1	:::1	TCP	88	54396 → 9393 [SYN] Seq=0 Win=65535 Len=0 MSS=16324 WS=64 TSval=3100639359...	0.0
16	0.000040	:::1	:::1	TCP	64	9393 → 54396 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0	0.0
17	0.000132	127.0.0.1	127.0.0.1	TCP	68	54397 → 9393 [SYN] Seq=0 Win=65535 Len=0 MSS=16344 WS=64 TSval=2463729067...	0.0
18	0.000378	127.0.0.1	127.0.0.1	TCP	68	9393 → 54397 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=16344 WS=64 TSval...	0.0
19	0.000399	127.0.0.1	127.0.0.1	TCP	56	54397 → 9393 [ACK] Seq=1 Ack=1 Win=408256 Len=0 TSval=2463729067 TSecr=13...	0.0
20	0.000415	127.0.0.1	127.0.0.1	TCP	56	[TCP Window Update] 9393 → 54397 [ACK] Seq=1 Ack=1 Win=408256 Len=0 TSval...	0.0
21	0.000560	127.0.0.1	127.0.0.1	TLSv1.2	266	Client Hello (SNI=localhost)	0.0
22	0.000588	127.0.0.1	127.0.0.1	TCP	56	9393 → 54397 [ACK] Seq=1 Ack=211 Win=408064 Len=0 TSval=1304680373 TSecr=...	0.0
23	0.001071	127.0.0.1	127.0.0.1	TLSv1.2	709	Server Hello, Certificate, Server Key Exchange, Server Hello Done	0.0
24	0.001092	127.0.0.1	127.0.0.1	TCP	56	54397 → 9393 [ACK] Seq=211 Ack=654 Win=407616 Len=0 TSval=2463729068 TSec...	0.0
25	0.001551	127.0.0.1	127.0.0.1	TLSv1.2	149	Client Key Exchange, Change Cipher Spec, Encrypted Handshake Message	0.0
26	0.001571	127.0.0.1	127.0.0.1	TCP	56	9393 → 54397 [ACK] Seq=654 Ack=304 Win=407936 Len=0 TSval=1304680374 TSec...	0.0
27	0.001572	127.0.0.1	127.0.0.1	TLSv1.2	107	Change Cipher Spec, Encrypted Handshake Message	0.0
28	0.001693	127.0.0.1	127.0.0.1	TCP	56	54397 → 9393 [ACK] Seq=304 Ack=705 Win=407552 Len=0 TSval=2463729068 TSec...	0.0
29	0.001719	127.0.0.1	127.0.0.1	TLSv1.2	124	Application Data	0.0
30	0.001739	127.0.0.1	127.0.0.1	TCP	56	54397 → 9393 [ACK] Seq=304 Ack=773 Win=407488 Len=0 TSval=2463729068 TSec...	0.0
31	0.001758	127.0.0.1	127.0.0.1	TLSv1.2	149	Application Data	0.0
32	0.001777	127.0.0.1	127.0.0.1	TCP	56	9393 → 54397 [ACK] Seq=773 Ack=397 Win=407872 Len=0 TSval=1304680374 TSec...	0.0
33	0.001819	127.0.0.1	127.0.0.1	TLSv1.2	98	Application Data	0.0

HTTP/3



No.	Time	Source	Destination	Protocol	Length	Info
11	*REF*	127.0.0.1	127.0.0.1	QUIC	1284	Initial, DCID=869087d1091e7e03808c598b7eeaf1c63edcf276, SCID=416fe0a7, PKN: 0, P
12	0.001904	127.0.0.1	127.0.0.1	QUIC	1284	Handshake, DCID=416fe0a7, SCID=4cdfd4fa
13	0.001924	127.0.0.1	127.0.0.1	QUIC	62	Protected Payload (KP0), DCID=416fe0a7
14	0.002785	127.0.0.1	127.0.0.1	QUIC	1284	Handshake, DCID=c4154d72, SCID=416fe0a7
15	0.002801	127.0.0.1	127.0.0.1	QUIC	72	Handshake, DCID=c4154d72, SCID=416fe0a7
16	0.002812	127.0.0.1	127.0.0.1	QUIC	64	Protected Payload (KP0)
17	0.002970	127.0.0.1	127.0.0.1	QUIC	96	Protected Payload (KP0)
18	0.003098	127.0.0.1	127.0.0.1	QUIC	60	Protected Payload (KP0)
19	0.003226	127.0.0.1	127.0.0.1	QUIC	269	Protected Payload (KP0)
20	0.003342	127.0.0.1	127.0.0.1	QUIC	110	Protected Payload (KP0)
21	0.003449	127.0.0.1	127.0.0.1	QUIC	63	Protected Payload (KP0)
22	0.003546	127.0.0.1	127.0.0.1	QUIC	58	Protected Payload (KP0)
23	0.033136	127.0.0.1	127.0.0.1	QUIC	58	Protected Payload (KP0)

0-RTT



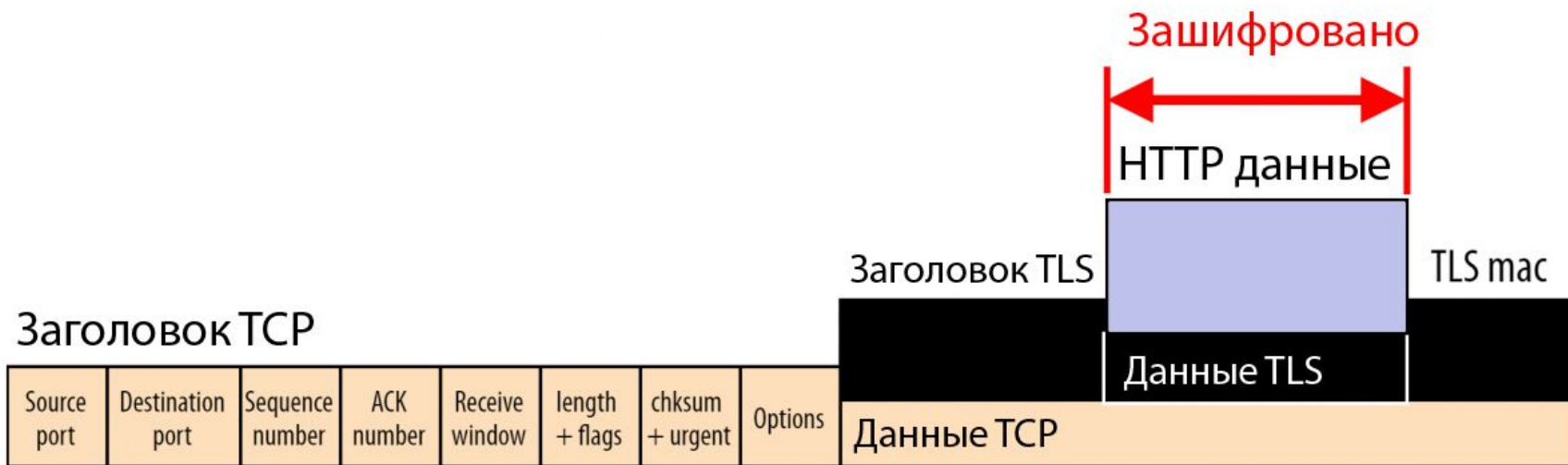
Для сервера

```
quic.Config{  
    Allow0RTT: true,  
}
```

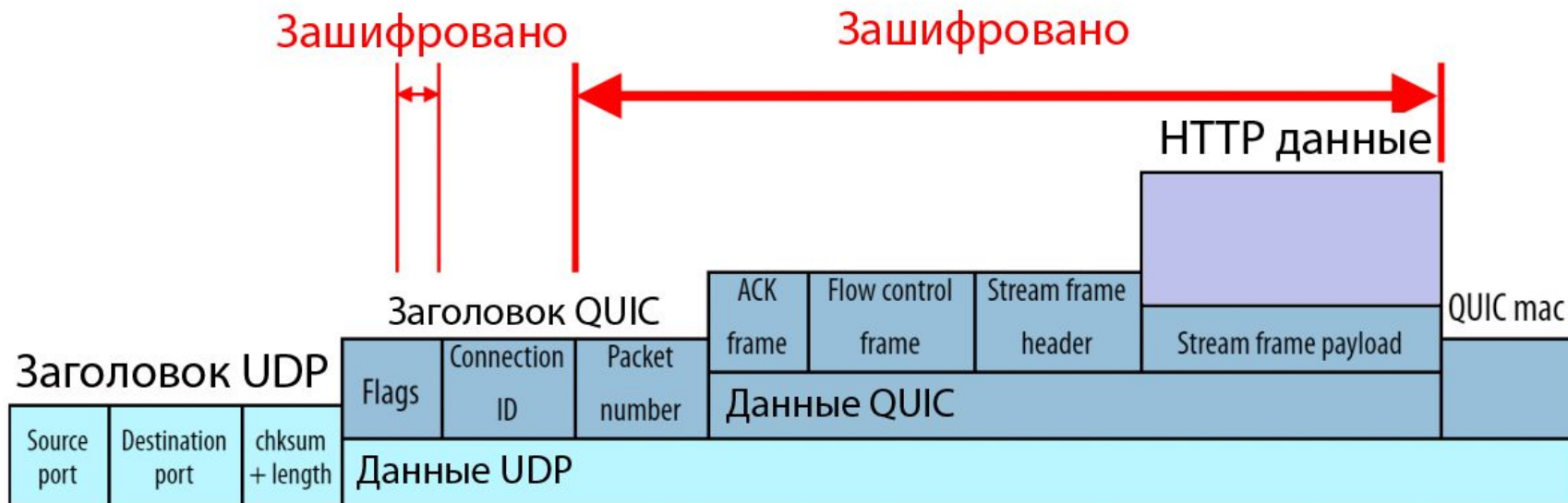
HTTP/3 безопаснее HTTP/2

Спойлер: в каком-то смысле...

TLS реализован на уровень выше, чем TCP, поэтому шифруется только TCP payload



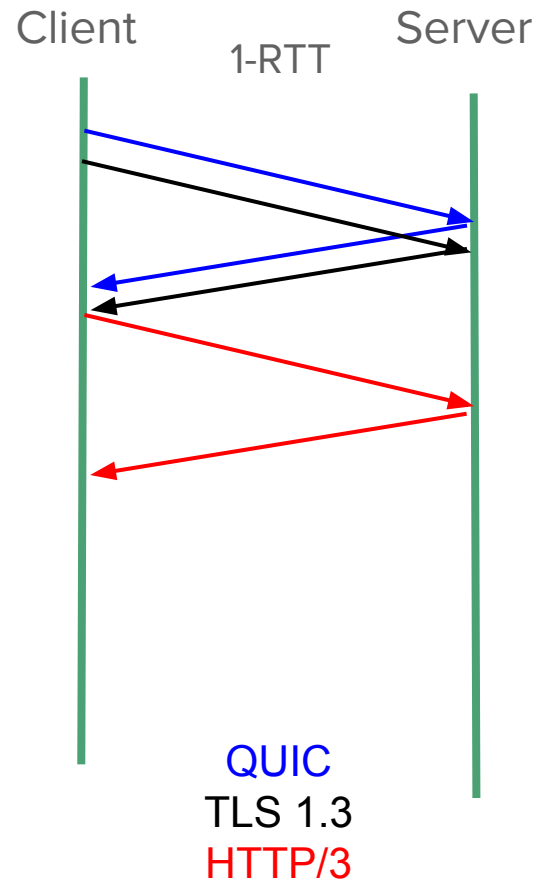
В QUIC TLS интегрирован, поэтому шифруется все, кроме части технической информации QUIC



Шифрование данных

Крипто-соединение и рукопожатие: [CRYPTO HANDSHAKE](#)

QUIC комбинирует крипто и транспортное рукопожатие, чтобы минимизировать задержку в установлении соединения



В QUIC TLS 1.3 по умолчанию

Perfect Forward Secrecy (PFS) - защита сессионных ключей в долгосрочной перспективе, даже если закрытый ключ был скомпрометирован

Меньше шифров, но они безопаснее (не поддерживает MD5 и SHA-1)

Исправлены уязвимости, лучше защита от атак и производительность

HTTP/3 отличается от HTTP/2? Не сильно

HTTP/3 надежный? Да!

HTTP/3 быстрее? Не особо

HTTP/3 безопаснее? В целом да

Часть 2. Зачем использовать HTTP/3?



Стримы

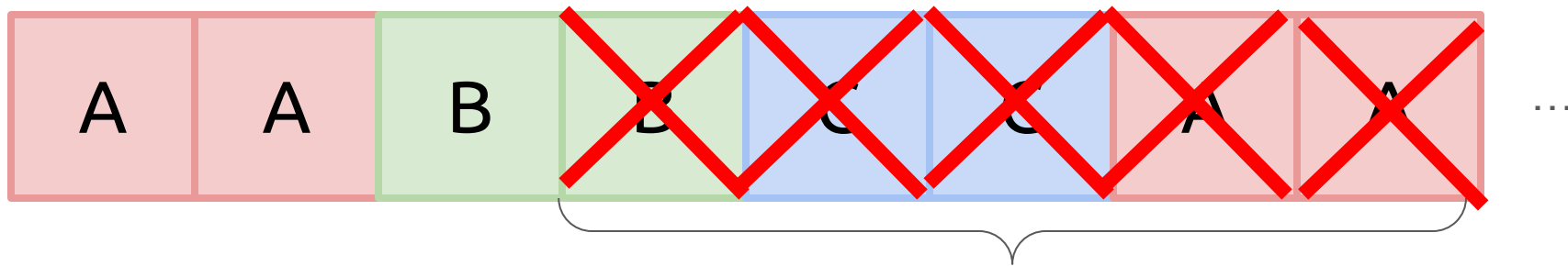


HTTP/2 стримы

Short-live — существуют в рамках одного запроса

Воспринимаются как единый байтовый поток

В случае потеря пакета — блокируются все остальные стримы (Head-Of-Line)



Все ожидает своей очереди,
пока пакет ретранслируется

Запись в стрим HTTP/2 сервером

```
func handleStream(w http.ResponseWriter, r *http.Request) {  
    for i := 0; i < 5; i++ {  
        message := fmt.Sprintf("That's stream %d\n", i)  
        err := w.Write([]byte(message))  
        if err != nil {...}  
    }  
}
```

Чтение клиентом HTTP/2 данных из стрима

```
resp, err := client.Get(url)
if err != nil {...}
defer resp.Body.Close()
```

```
body, err := io.ReadAll(resp.Body)
if err != nil {...}
```

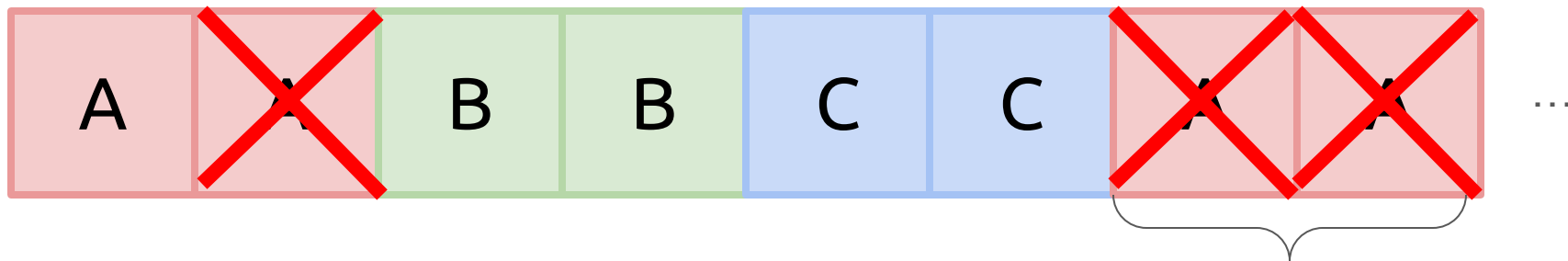
HTTP/3 стримы

Реализованы на уровне QUIC

Могут жить вне запросов

Мультиплексирование - несколько стримов может быть открыто в рамках одного QUIC соединения

Каждый стрим независим — нет Head-Of-Line



Нет Head-Of-Line блокировки

Только один стрим страдает от потери данных

Типы стримов

Два последних бита ID	Тип стрима
id = 0, 4, 8	Инициирован клиентом, двунаправленный
id = 2, 6, 10	Инициирован клиентом, однонаправленный
id = 1, 5, 9	Инициирован сервером, двунаправленный
id = 3, 7, 11	Инициирован сервером, однонаправленный

Ограничение количества стримов

Максимально возможное количество стримов 2^{60}

Сервер и клиент может ограничить количество открытых двусторонних и односторонних стримов

Сервер и клиент могут ограничить окно потока на один стрим и на все соединения

```
quicConfig := &quic.Config{
    MaxIncomingStreams:      2,
    MaxIncomingUniStreams:  2,
}
```

Открытие двустороннего стрима клиентом HTTP/3

```
roundTripper := &http3.RoundTripper{
    TLSClientConfig: &tls.Config{...},
    QuicConfig:      &quic.Config{...},
}

req, err := http.NewRequest("GET", "https://e.com/str", nil) if err != nil {...}
roundTripOpt := http3.RoundTripOpt{
    DontCloseRequestStream: true
}

resp, err := roundTripper.RoundTripOpt(req, roundTripOpt)
...

stream := resp.Body.(http3.HTTPStreamer).HTTPStream()
if _, err := stream.Write(bytes); err != nil {...}
<-stream.Context().Done()
```


Зачем читать RFC?

This means that the client's first request occurs on QUIC stream 0, with subsequent requests on streams 4, 8, and so on. In order to permit these streams to open, an HTTP/3 server SHOULD configure non-zero minimum values for the number of permitted streams and the initial stream flow-control window. So as to not unnecessarily limit parallelism, at least 100 request streams SHOULD be permitted at a time.

HTTP/3 does not use server-initiated bidirectional streams, though an extension could define a use for these streams. Clients MUST treat receipt of a server-initiated bidirectional stream as a connection error of type H3_STREAM_CREATION_ERROR unless such an extension has been negotiated.

6.2. Unidirectional Streams

Получение данных на сервере

```
func Str(w http.ResponseWriter, req *http.Request) {  
    w.WriteHeader(200)  
    w.(http.Flusher).Flush()  
  
    stream := req.Body.(http3.HTTPStreamer).HTTPStream()  
  
    data := make([]byte, lengthOfData)  
    if _, err = stream.Read(data); err != nil {...}  
    ...  
    stream.CancelRead(quick.StreamErrorCode(quick.NoError))  
}
```

Управление стримами

`CancelRead(StreamErrorCode)`

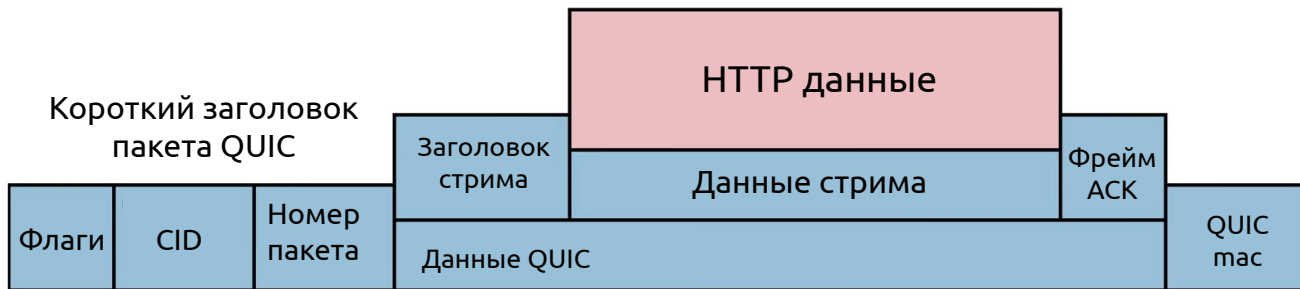
`SetReadDeadline(time.Time) error`

`CancelWrite(StreamErrorCode)`

`SetWriteDeadline(t time.Time) error`

`Context() context.Context`

Стрим - фрейм внутри пакета QUIC



Тип фреймов	DATA	0x00	Datagram	0x30, 0x31
	HEADER	0x01	Webtransport	0x41
	SETTINGS	0x04		

Типы кадры

```
switch t {
case 0x0:
    return &dataFrame{Length: l}, nil
case 0x1:
    return &headersFrame{Length: l}, nil
case 0x4:
    return parseSettingsFrame(r, l)
case 0x3: // CANCEL_PUSH
case 0x5: // PUSH_PROMISE
case 0x7: // GOAWAY
case 0xd: // MAX_PUSH_ID
}
```

Сервер

```
stream.CancelRead(quic.StreamErrorCode(quic.NoError))
```

Клиент

```
<-stream.Context().Done()
```

Datagrams

Негарантированная, неупорядоченная доставка ([RFC](#))

Клиент

```
http3.RoundTripper{
    ...
    EnableDatagrams: true,
    QuicConfig: &quic.Config{
        ...
        EnableDatagrams: true,
    },
},
```

Сервер

```
http3.Server{
    ...
    EnableDatagrams: true,
    QuicConfig: &quic.Config{
        ...
        EnableDatagrams: true,
    },
},
```

Frame types: 0x30, 0x31

Webtransport over HTTP/3

[RFC](#), июль 2022

- Гарантированная упорядоченная доставка с помощью одно/двух направленных стримов (альтернатива WebSocket)
- Негарантированная неупорядоченная доставка с помощью Datagrams
- Резервированный тип стрима

Code: 0x41

Frame Type: WEBTRANSPORT_STREAM



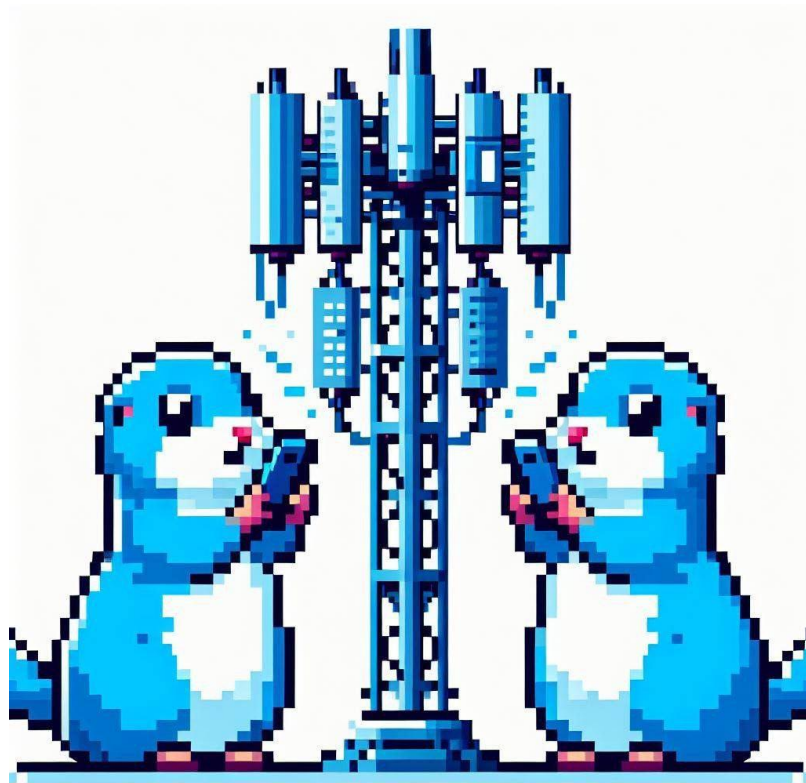
Собственные типы фреймов

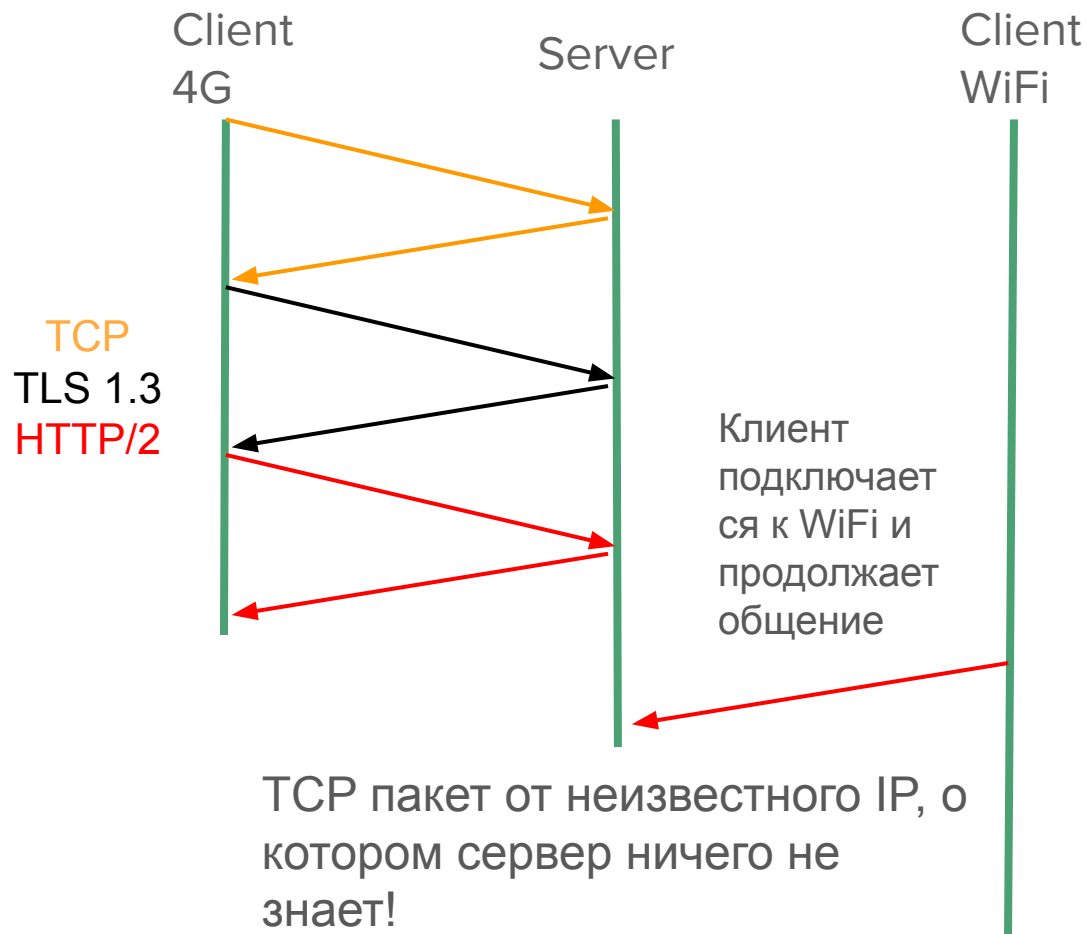
```
roundTripper := &http3.RoundTripper{
    ...
    UniStreamHijacker: UniStreamHijacker,
    StreamHijacker:    StreamHijacker,
}
```

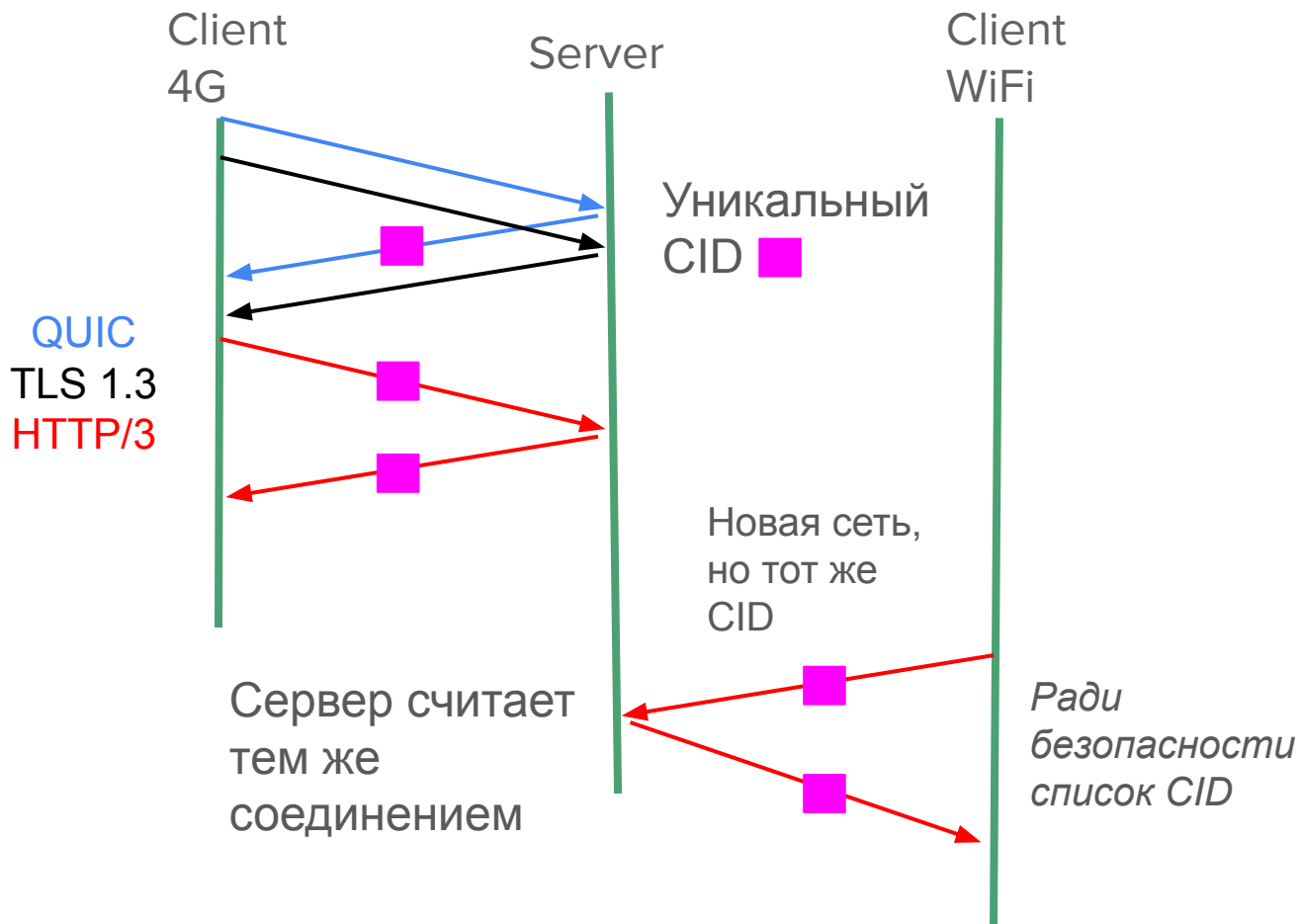
UniStreamHijacker func(StreamType, quic.Connection,
quic.ReceiveStream, error) (hijacked bool)

StreamHijacker func(FrameType, quic.Connection, quic.Stream,
error) (hijacked bool, err error)

Миграция соединения







Реализация QUIC



TCP — сложно дорабатывать протокол

Реализован на уровне системных библиотек ОС

Были попытки доработать:

TCP Fast Open - без рукопожатия

MultiPath TCP - использовать и WiFi и 4G, чтобы увеличить пропускную способность



HTTP/3 / QUIC / UDP — проще развивать

Реализован на уровне библиотек прикладных программ

Почти весь кадр QUIC шифруется: достаточно обновить конечные устройства, промежуточные устройства не смогут проанализировать изменения

Открыт к изменениям и доработкам, например, нет жестких требований к алгоритму контролю перегрузки или приоритизации стримов

Часть 3. Подводные камни

HTTP/3 в Go

Не устоявшийся стандарт

Не полностью безопасен — существуют атаки

Отсутствие нормальной документации и примеров использования

Трафик HTTP/3 шифруется на транспортном уровне, поэтому его сложно анализировать

В Go библиотеке реализован алгоритм CUBIC, но используется классический алгоритм Reno

```
// NewCubicSender makes a new cubic sender
func NewCubicSender(
    clock Clock,
    rttStats *utils.RTTStats,
    initialMaxDatagramSize protocol.ByteCount,
    reno bool,
    tracer *logging.ConnectionTracer,
) *cubicSender {
```

НО...

```
congestion := congestion.NewCubicSender(
    congestion.DefaultClock{},
    rttStats,
    initialMaxDatagramSize,
    reno: true, // use Reno
    tracer,
)
```



[QUIC-GO](#)

Новые релизы могут ломать старый код!

```
roundTripper := &http3.RoundTripper{
    TLSClientConfig: &tlsConfig,
    QuicConfig:      &quic.Config{},
    QUICConfig: &quic.Config{},
}
```

Минимальная реализация сжатия заголовков QPACK

QPACK



This is a minimal QPACK ([RFC 9204](#)) implementation in Go. It is minimal in the sense that it doesn't use the dynamic table at all, but just the static table and (Huffman encoded) string literals. Wherever possible, it reuses code from the [HPACK implementation in the Go standard library](#).

It is interoperable with other QPACK implemetations (both encoders and decoders), however it won't achieve a high compression efficiency.

ИТОГ

HTTP version by requests share over time (Multiple browsers, Worldwide)

Worldwide - 2022-05-26 00:00:00 to 2023-04-30 01:00:00 (UTC)

