QUIC+HTTP/3

と、これからの HTTP

奥 一穂

Nov. 2025

自己紹介

- 奥一穂 / senior principal OSS Engineer, Fastly
- 主開発者として:
  - H2O HTTP server / picotls / quicly
- 仕様の(共)著者として:
  - RFC 8297 (103 Early Hints)
  - RFC 9218 (Extensible Priorities)
  - to-be RFC 9849 (Encrypted Client Hello)
  - draft-ietf-quic-reliable-stream-reset
  - draft-ietf-httpbis-incremental
  - draft-ietf-scone-protocol

目次

- QUICの背景
  - TCPの進化史 / 遅延削減の重要性 / プライバシー・倍・デザイン / HTTP/2 / TLS/1.3 / さらなる高速化を目指して
- 第3世代HTTPの要件
- QUICの設計ポイント
  - Invariants, QUIC version 1, 再送制御, 硬直化抑止策
- QUICならではの最適化
- QUICとHTTPのこれから

QUICの背景

TCPの進化史

TCP/IPのレイヤ (RFC 1122)

トランスポートの機能

- 接続管理
  - 4-tupleで識別 (サバクラ双方の IP:port)
  - SYN, FIN, RST等によるステート

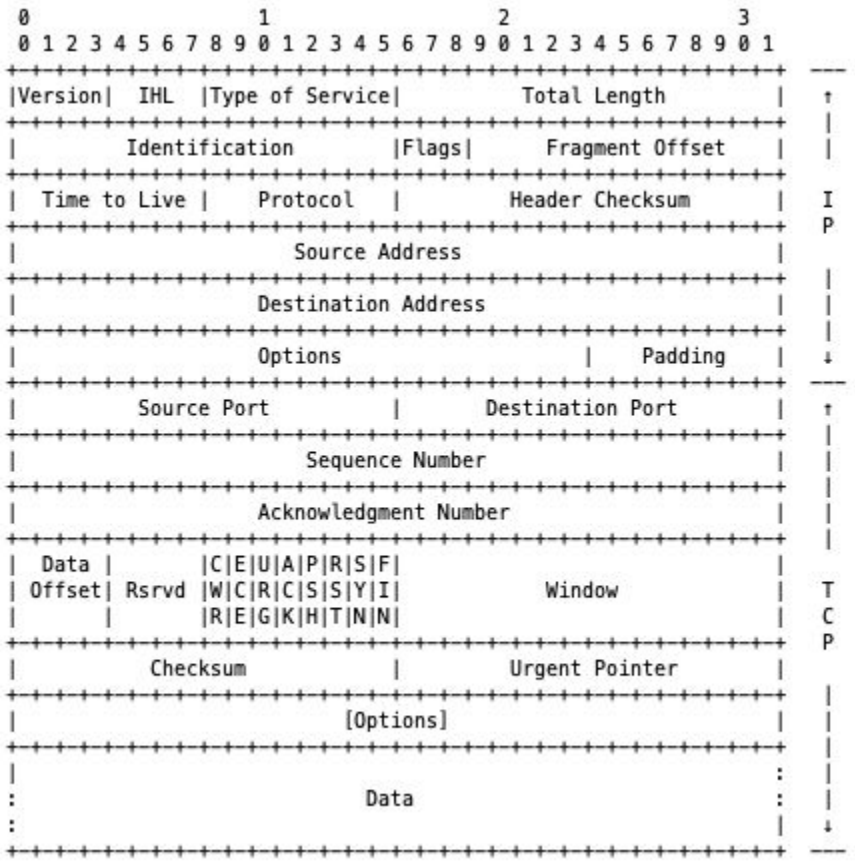
トランスポートの機能

- 接続管理
  - 4-tupleで識別 (サバクラ双方の IP:port)
  - SYN, FIN, RST等によるステート
- 信頼性
  - パケロスを検知して再送

トランスポートの機能

- 接続管理
  - 4-tupleで識別 (サバクラ双方の IP:port)
  - SYN, FIN, RST等によるステート
- 信頼性
  - パケロスを検知して再送
- 送信レート制御
  - フロー制御 - 受信側を保護
  - 輻輳制御 - 経路を保護・公平性を確保

TCP

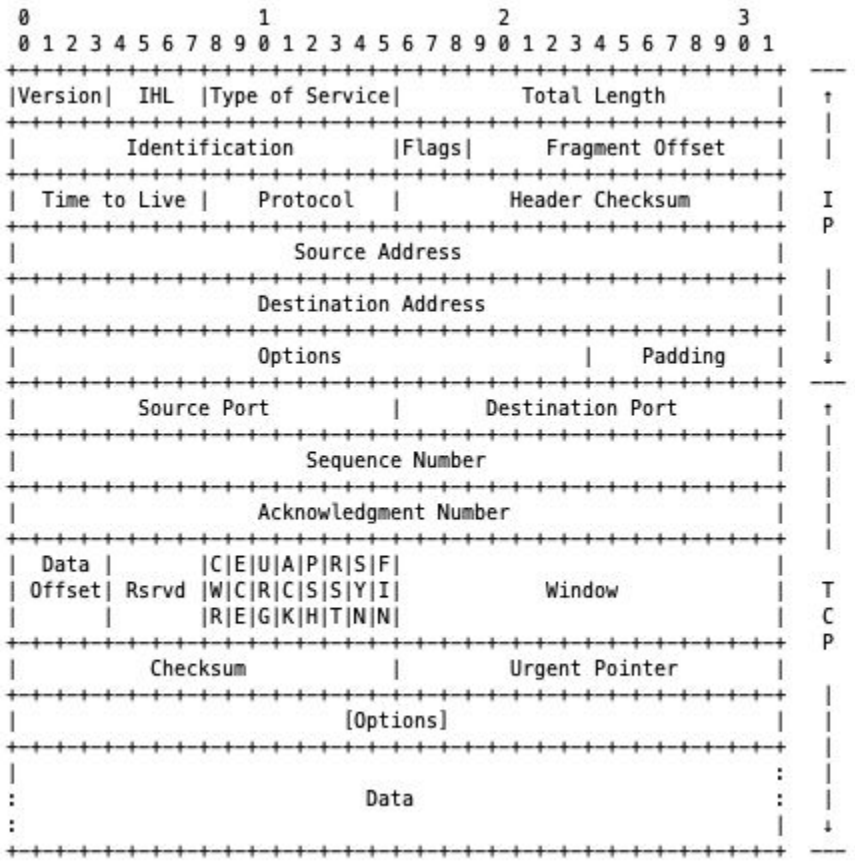

0 1 2 3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
Version
IHL
Type of Service
Total Length
Identification
Flags
Fragment Offset
Time to Live
Protocol
Header Checksum
Source Address
Destination Address
Options
Padding
IP
Source Port
Destination Port
Sequence Number
Acknowledgment Number
Data Offset
Rsrvd
C|E|U|A|P|R|S|F|
W|C|R|C|S|S|Y|I|
R|E|G|K|H|T|N|N|
Window
Checksum
Urgent Pointer
[Options]
Data
TCP

TCP

接続管理

信頼性

(ロス・破壊検知)



フロー制御

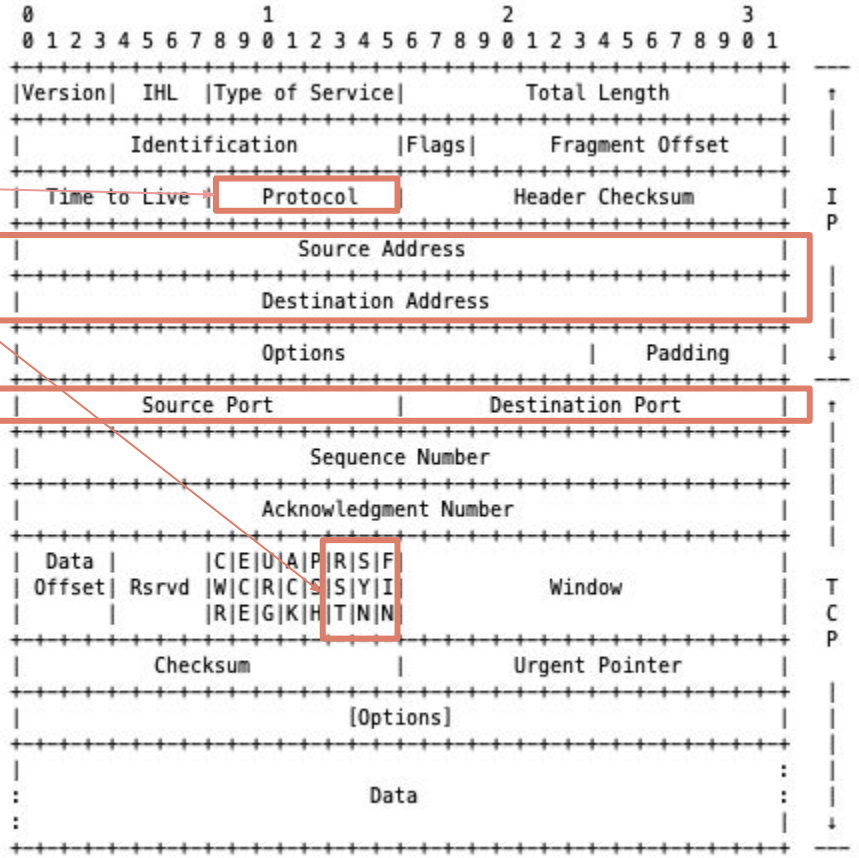
輻輳制御

TCP

接続管理

信頼性

(ロス・破壊検知)



フロー制御

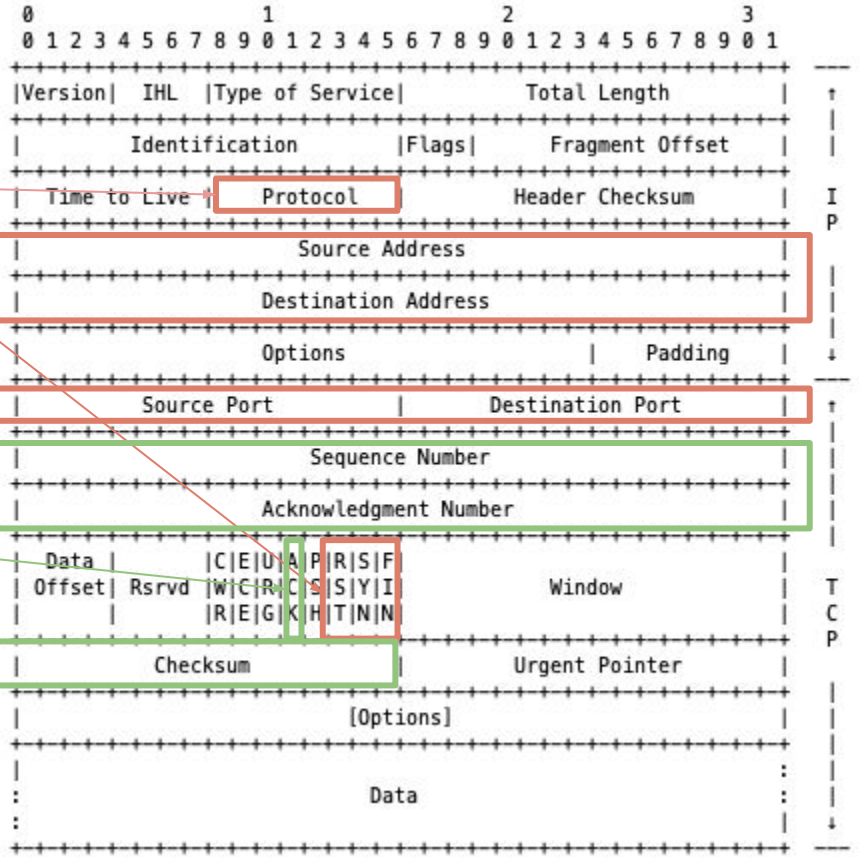
輻輳制御

TCP

接続管理

信頼性

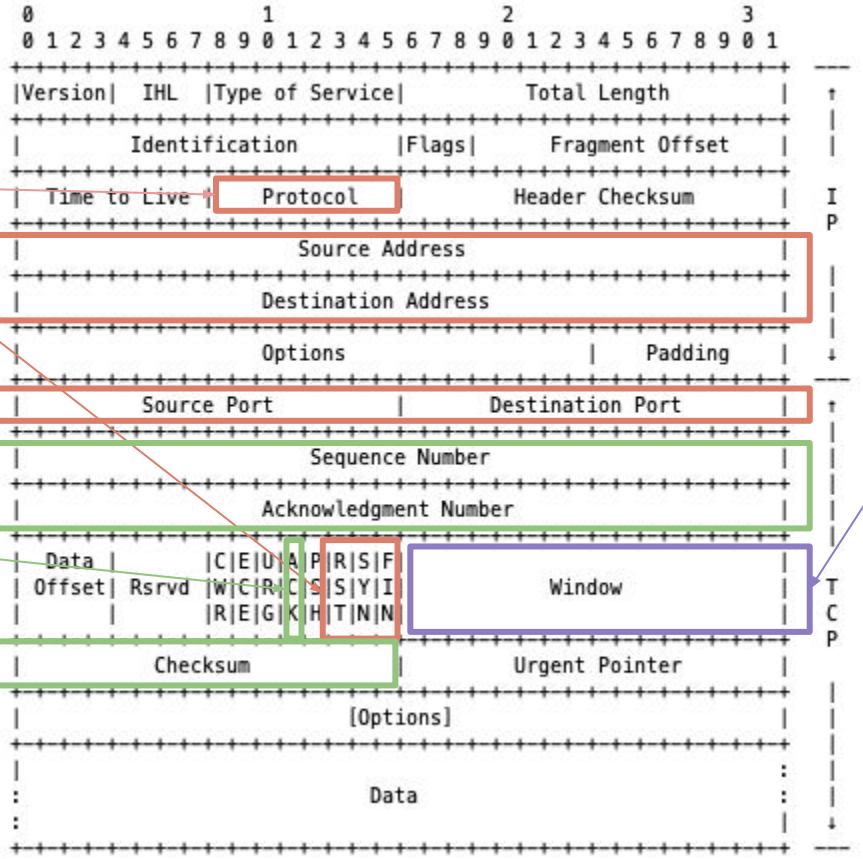
(ロス・破壊検知)



フロー制御

輻輳制御

TCP



接続管理

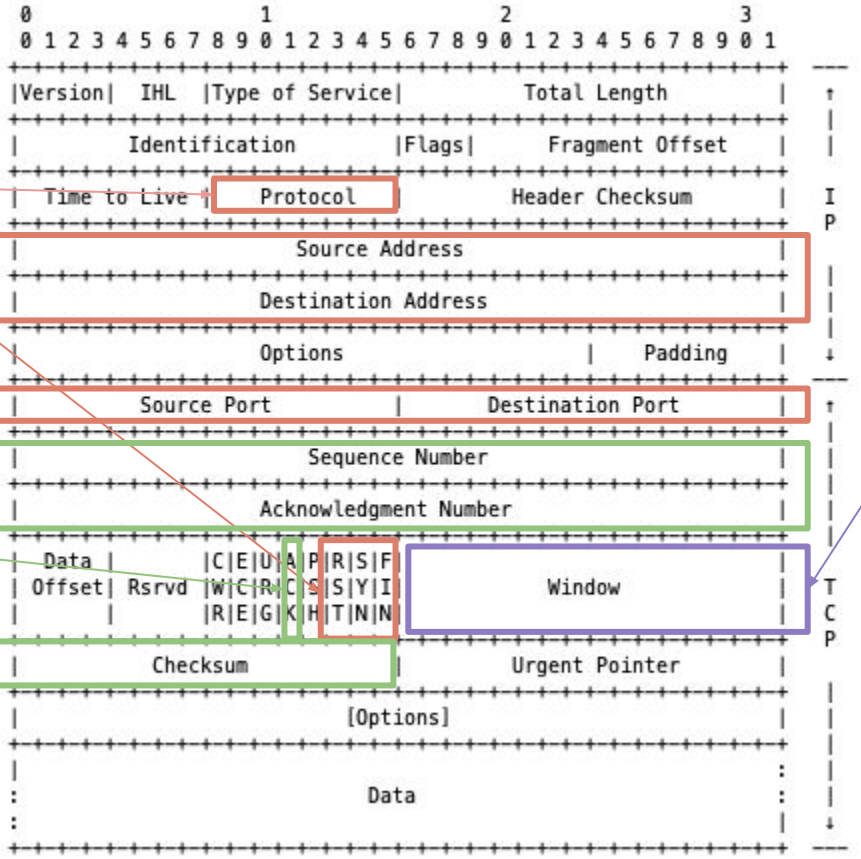
信頼性

(ロス・破壊検知)

フロー制御

輻輳制御

TCP



接続管理

信頼性

(ロス・破壊検知)

フロー制御

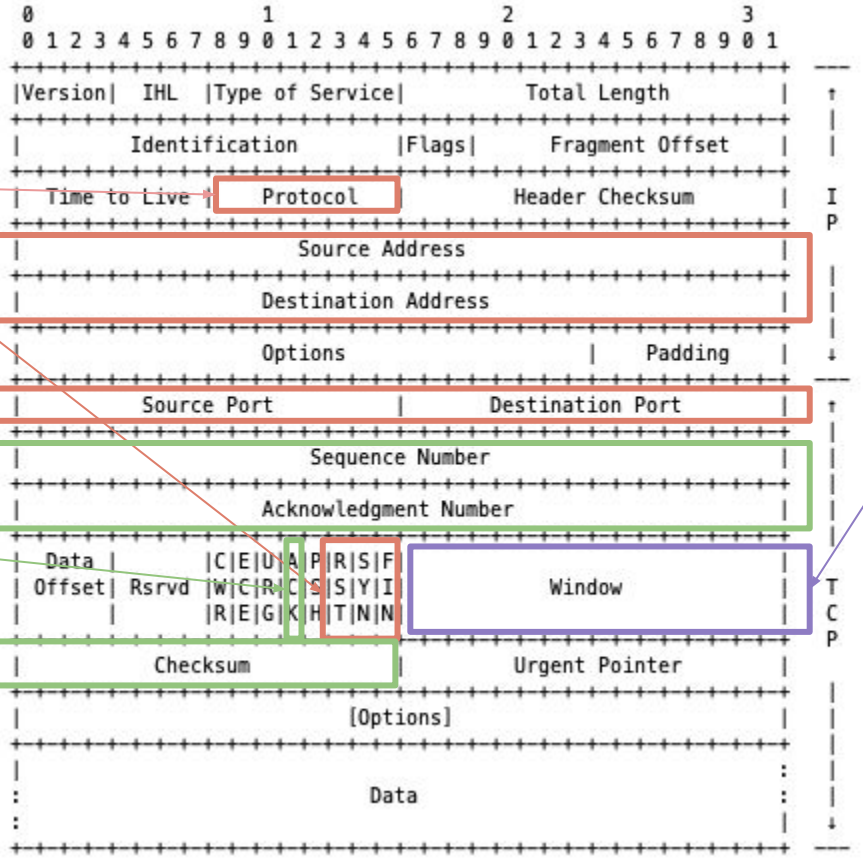
輻輳制御

TCP

接続管理

信頼性

(ロス・破壊検知)



フロー制御

輻輳制御

???

初期のTCP

- 送信レート制限はフロー制御のみ
  - 受信側が最大64KBのウィンドウ告知

初期のTCP

- 送信レート制限はフロー制御のみ
  - 受信側が最大64KBのウィンドウ告知
- 送信者は
  - ウィンドウ制限まで送信

初期のTCP

- 送信レート制限はフロー制御のみ
  - 受信側が最大64KBのウィンドウ告知
- 送信者は
  - ウィンドウ制限まで送信
  - ACK+新たなウィンドウが返ってきたら
    - 上の繰り返し

初期のTCP

- 送信レート制限はフロー制御のみ
  - 受信側が最大 64KBのウィンドウ告知
- 送信者は
  - ウィンドウ制限まで送信
  - ACK+新たなウィンドウが返ってきたら
    - 上の繰り返し
  - ACKが一定時間以内に返ってこなければ
    - 返ってこなかったところの 1パケットを再送信（RTO）

輻輳制御の誕生

- 輻輳崩壊 (1986年)

輻輳制御の誕生

- 輻輳崩壊 (1986年)
- Congestion Avoidance and Control (V. Jacobson; 1998年)
  - 輻輳制御の基本概念

輻輳制御の基本 (1)

- スロースタート
  - 目的: CWNDを高速に推定
    - CWND=Congestion Window
      - 1RTごとに送達可能なデータ量

輻輳制御の基本 (1)

- スロースタート
  - 目的: CWNDを高速に推定
    - CWND=Congestion Window
      - 1RTごとに送達可能なデータ量
  - 接続確立直後はバンド幅 (CWND)がわからないため
    - 1バイトACKされたら CWND += 2バイト
      - i.e., 1ラウンドトリップ (RT)毎にCWNDが2倍

輻輳制御の基本 (1)

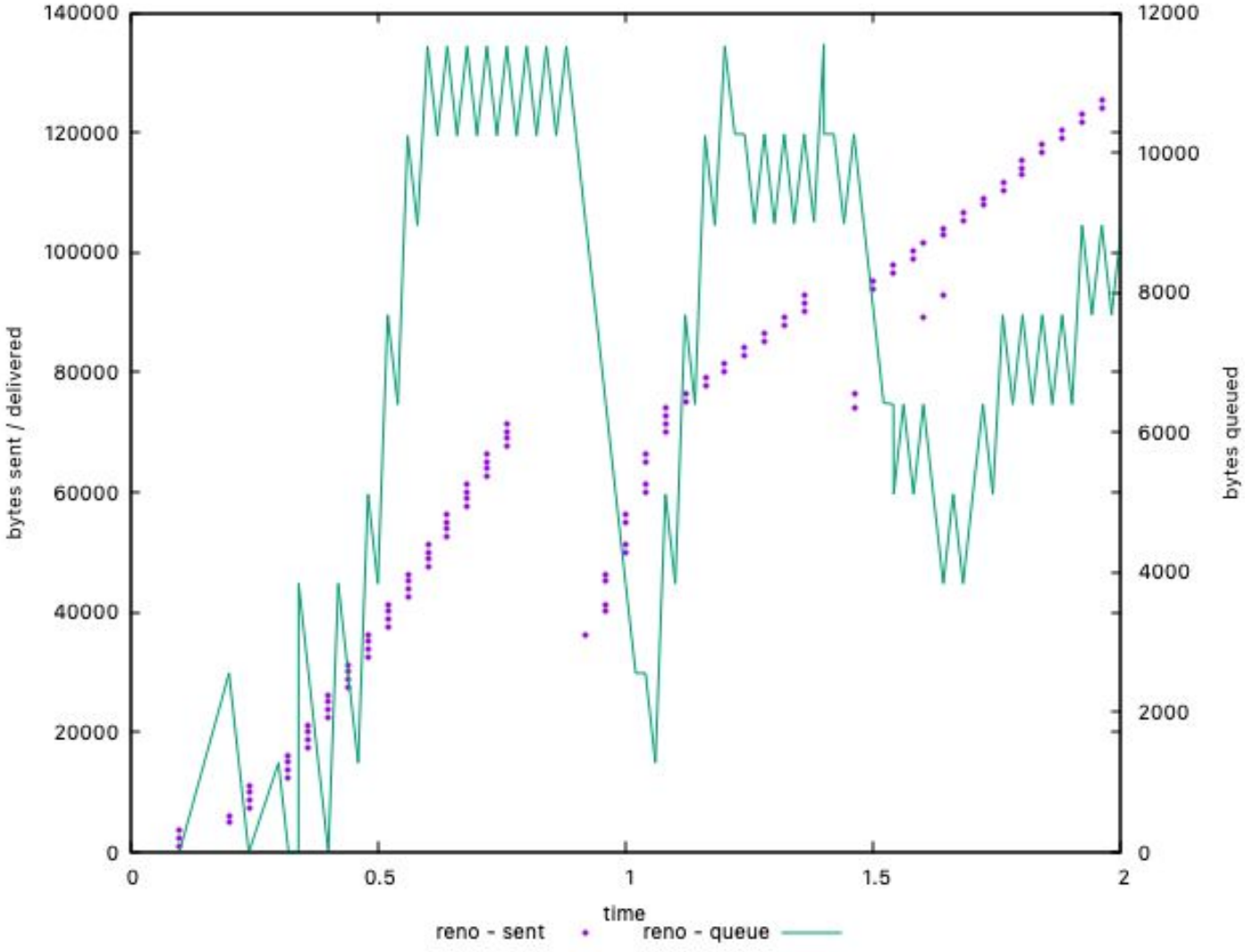
- スロースタート
  - 目的: CWNDを高速に推定
    - CWND=Congestion Window
      - 1RTごとに送達可能なデータ量
  - 接続確立直後はバンド幅 (CWND)がわからないため
    - 1バイトACKされたら CWND += 2バイト
      - i.e., 1ラウンドトリップ (RT)毎にCWNDが2倍
  - パケロスしたら輻輳回避フェーズに遷移

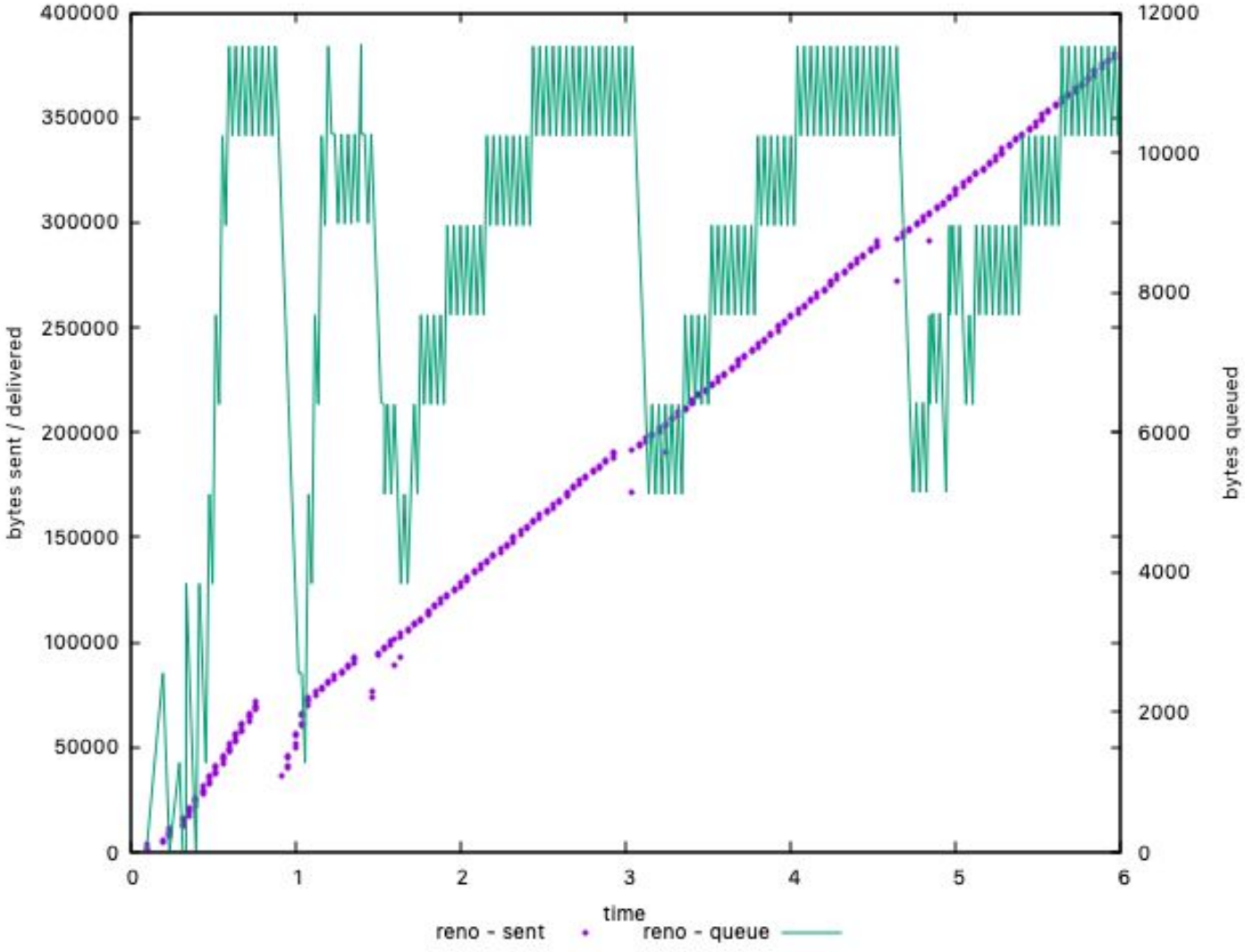
輻輳制御の基本 (2)

- 輻輳回避 (Congestion Avoidance)
  - パケロスが稀な程度にバンド幅を利用

輻輳制御の基本 (2)

- 輻輳回避 (Congestion Avoidance)
  - パケロスが稀な程度にバンド幅を利用
  - AIMD制御
    - RTごとに送信量を CWND += 1パケット
    - パケロスしたら CWND *= 0.5


140000
120000
100000
80000
60000
40000
20000
0
bytes sent / delivered
12000
10000
8000
6000
4000
2000
0
bytes queued
0
0.5
1
1.5
2
time
reno - sent
reno - queue


400000
350000
300000
250000
200000
150000
100000
50000
0
bytes sent / delivered
12000
10000
8000
6000
4000
2000
0
bytes queued
0
1
2
3
4
5
6
time
reno - sent
reno - queue

輻輳制御の基本（3）

- AIMD制御で公平性確保
  - 競合する接続の間で、ロス発生までの増分は同じ
    - RTTが同一と仮定
  - ロス発生時の減分は CWNDが大きい方が大

輻輳制御の基本（3）

- AIMD制御で公平性確保
  - 競合する接続の間で、ロス発生までの増分は同じ
    - RTTが同一と仮定
  - ロス発生時の減分は CWNDが大きい方が大

	接続A	接続B
t=0	10	70
t=10	20 → 10	80 → 40
t=35	35 → 18	65 → 33
t=59	43 → 22	57 → 29
t=85	46	54

輻輳制御の基本 (4)

- 公平性確保は輻輳回避フェーズの役割
  - スロースタートは競合フロー出現時に短期間 (1RT) パケロスを発生させるノイズ

輻輳制御の発展

- Reno - 説明した考え方を実装したもの
  - 1988 Jacobsonをコードに落としたもの
  - 今でも基本

輻輳制御の発展

- Reno - 説明した考え方を実装したもの
  - 1988 Jacobsonをコードに落としたもの
  - 今でも基本
- Cubic - 高遅延かつ高帯域で Renoの動作を修正
  - 低遅延or低帯域なら Reno同様の動作

輻輳制御の発展

- Reno - 説明した考え方を実装したもの
  - 1988 Jacobsonをコードに落としたもの
  - 今でも基本
- Cubic - 高遅延かつ高帯域で Renoの動作を修正
  - 低遅延or低帯域なら Reno同様の動作
- 遅延ベースのアプローチ（vs. 上記ロスベース）
  - Vegas
  - BBR
  - いずれもロスベースと競合する場合の公平性が課題

再送制御の高度化

- パケロス発生前提になると RTOだけでは不足

再送制御の高度化

- パケロス発生前提になると RTOだけでは不足
- Fast Retransmit / Recovery
  - 1988 Jacobson
- Selective Acknowledgment
  - RFC 1072 (1988) -> RFC 2018 (1996)
- Window Scaling & Timestamps Option
  - RFC 1072 (1988) -> RFC 1323 (1992)

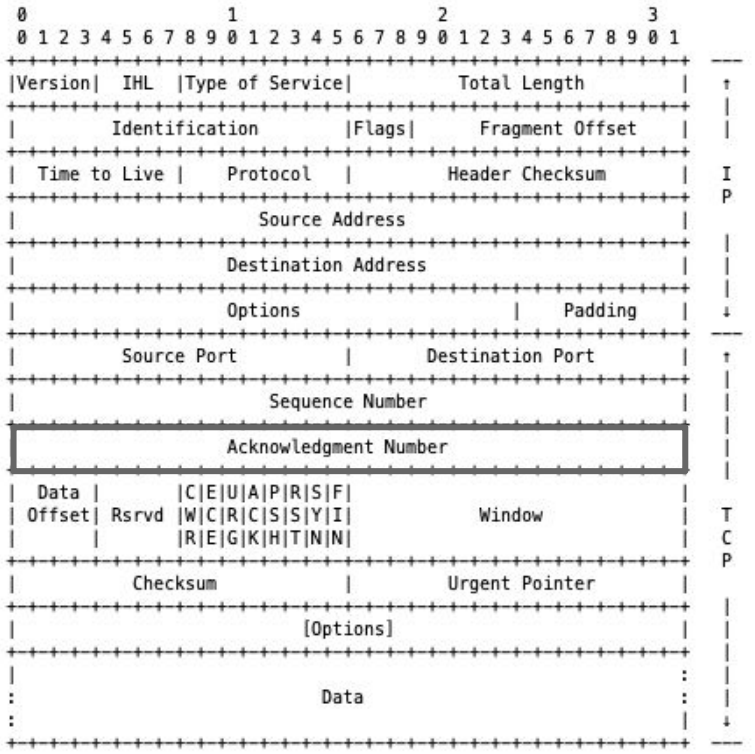
TCP Acknowledgment

- acknowledgment numberは「どこが抜けていたか」を送ることができない

1 2 3 4 5 6

↑

acknowledgment number



Fast Retransmit / Recovery

- RTOを待たずに再送したい
- 受信側: 抜けがある状態でパケット受信したら即 ACK
  - acknowledge numberは変わらない (DupACK)
- 送信側はDupACKを3つ受信したら、ロス判定 → 再送
  - acknowledge number以降全部再送しちゃうけど ...

Selective Acknowledgment

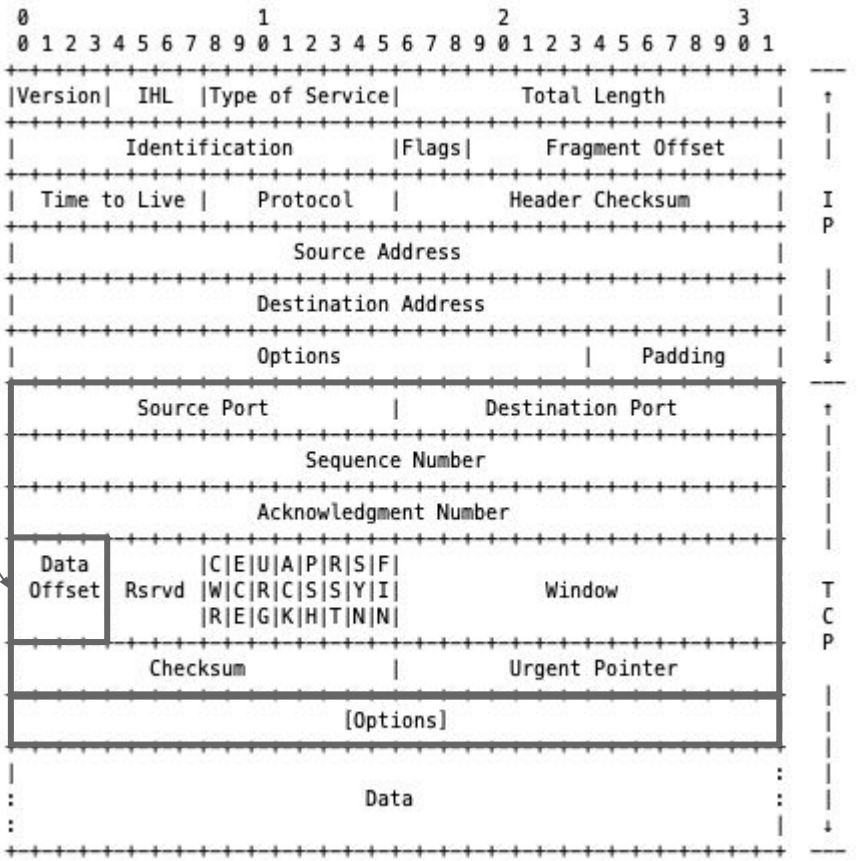
- 抜けているパケットを列挙して通知できれば、それらだけ再送したい！！

↓

TCP Optionsを使おう

TCP Options

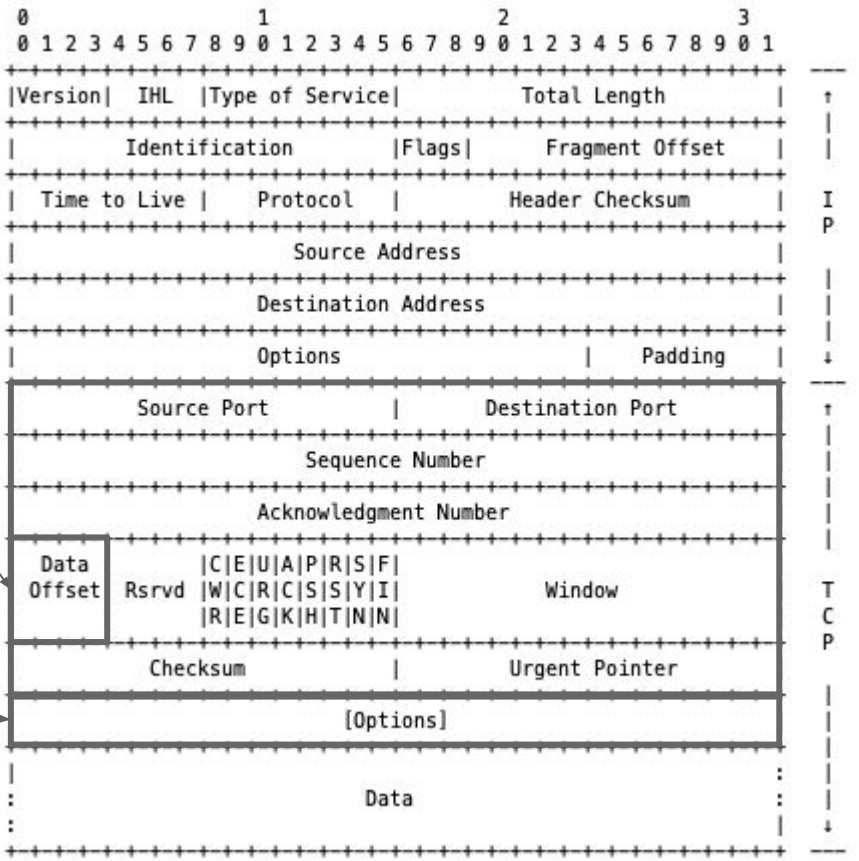
Dataの開始位置＝ Data Offset * 4 byte



TCP Options

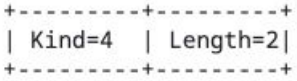
Dataの開始位置＝ Data Offset * 4 byte

TCPヘッダは最低 20バイト、
Data Offsetは4ビットフィールド
だからオプションは最大 15バイト



SACK Options

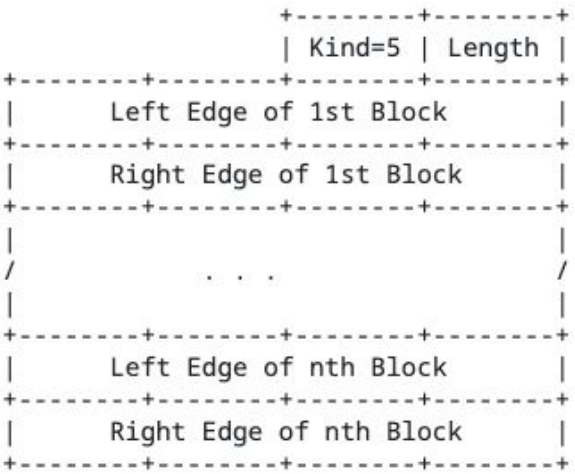
- SYNパケットにSack-Permitted Optionをつけてネゴ



SACK Options

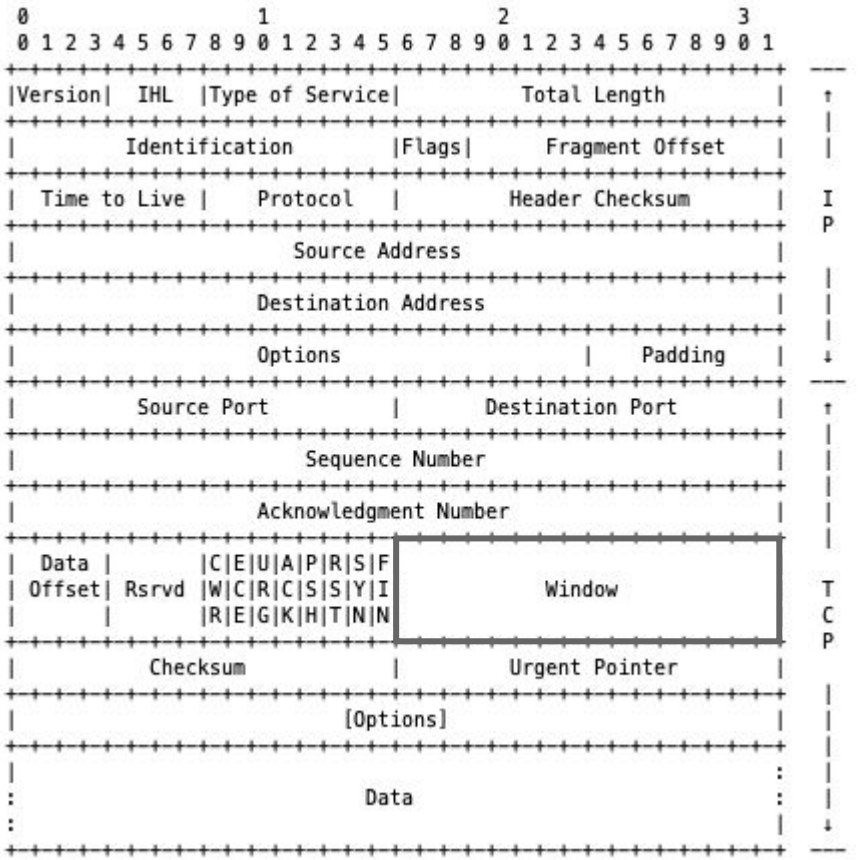
- SYNパケットにSack-Permitted Optionをつけてネゴ

- ACKパケットにSACK Optionをつけて最初の欠落以降に受信した範囲を通知
  - 最大4つの範囲を通知可能(実質3)
    cf. オプションのサイズ制限



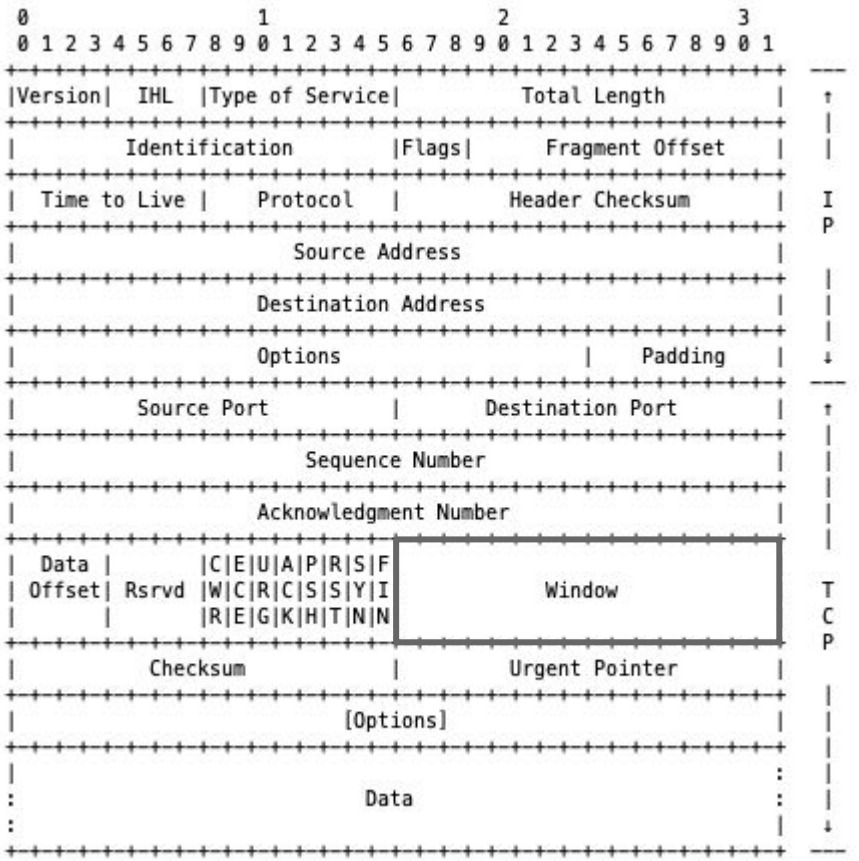
Window Scale Option

- 受信ウィンドウは 16bit、つまり65.5KB未満



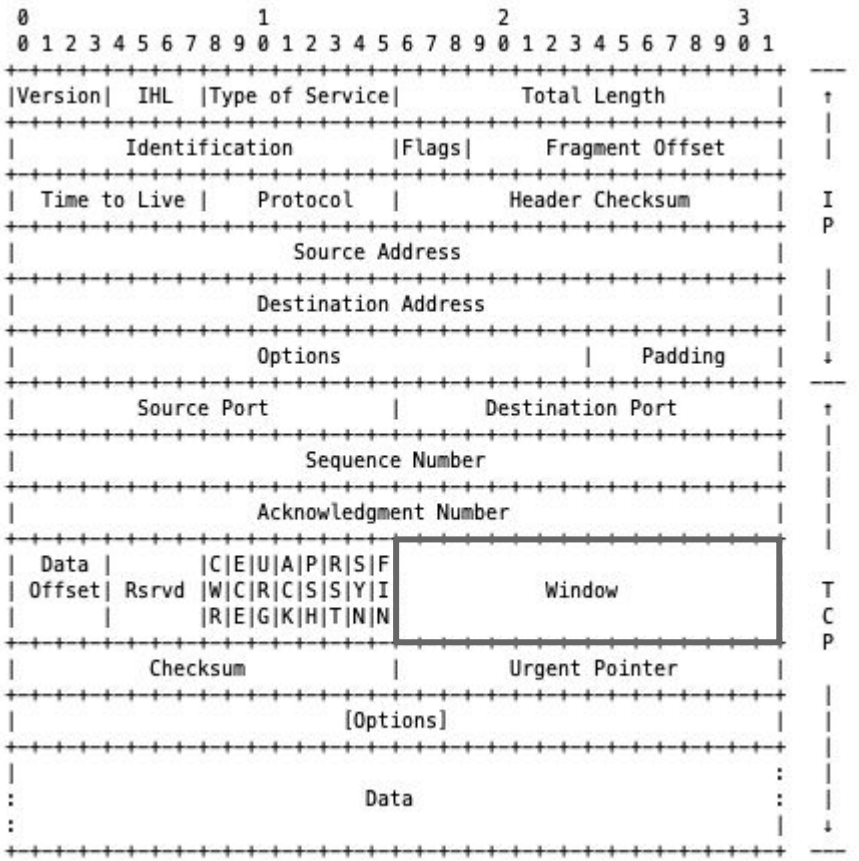
Window Scale Option

- 受信ウィンドウは 16bit、つまり65.5KB未満
- 1RTTで送信可能なデータ量=65.5KB



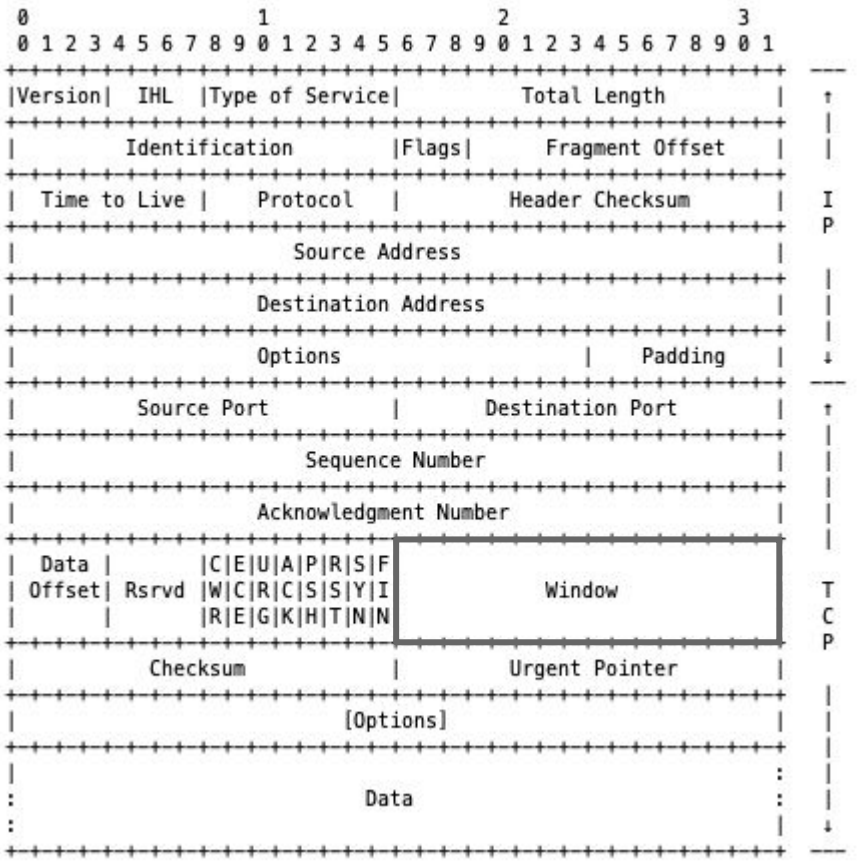
Window Scale Option

- 受信ウィンドウは 16bit、つまり65.5KB未満
- 1RTTで送信可能なデータ量=65.5KB
- RTT=200msなら328KB/sが上限



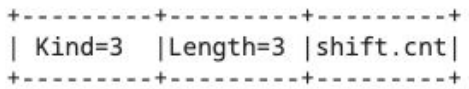
Window Scale Option

- 受信ウィンドウは 16bit、つまり65.5KB未満
- 1RTTで送信可能なデータ量=65.5KB
- RTT=200msなら328KB/sが上限
- もっと大きな受信ウィンドウを通知したい



Window Scale Option

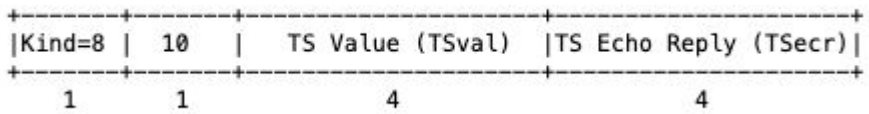
- SYNパケットにWindow Scale Optionをつけてネゴ



- 真のウィンドウサイズ = フィールドの値 << shift.cnt
  - ただしshift.cntは14以下に制限 (i.e., 1GiB)

Timestamps Option

- タイムスタンプ付きのパケット送って、送り返してもらおう、
  それ見たらRTT推定が捗る (RTO改善)



RACK-TLP

- RFC 8985 (2021) / 1988年以来の大変化
- パケット毎に送信時刻を記憶・送信時刻からの経過時間でロス判定

ロス検知	従来の再送制御	RACK-TLP
末尾以外	DupACK * 3	sRTT + ¼minRTT
末尾	RTO ≧ 0.2秒[注1]	max(2*sRTT, 1.5*sRTT + 40ms)[注2] 2回目以降: RTO利用

注1: 0.2秒はlinuxの場合。RFC 6298は1秒以上推奨
注2: linuxの場合の式。RFC 8985は2*sRTTにack-delayを足してもよいと規定

TCPの進化史 - まとめ

- 40年以上の歴史
- 元々は輻輳制御もなかった
- 1990年前後に主要な拡張が出揃う
- 近年に再送制御見直し (Rack-TLP)

遅延削減の重要性

遅延削減の重要性

- 2000年代後半に入ると、遅延削減とトラヒックや売上の相関が話題に

遅延削減の重要性

- 2000年代後半に入ると、遅延削減とトラヒックや売上の相関が話題に
- 2006:
  - 0.5秒の遅延増加で 20%減
    - Google Marrisa Mayer講演
  - 100msの遅延で売上 1%減
    - Greg Linden回想 (元Amazon従業員)

遅延削減の重要性

- 2000年代後半に入ると、遅延削減とトラヒックや売上の相関が話題に
- 2006:
  - 0.5秒の遅延増加で 20%減
    - Google Marrisa Mayer講演
  - 100msの遅延で売上 1%減
    - Greg Linden回想 (元Amazon従業員)
- 2009:
  - Bing, Microsoft, GoogleのVelocity等での発表

2009年Bing/Microsoft

Server Delays Experiment: Results

	Distinct Queries/User	Query Refinement	Revenue/User	Any Clicks	Satisfaction	Time to Click (increase in ms)
50ms	-	-	-	-	-	-
200ms	-	-	-	-0.3%	-0.4%	500
500ms	-	-0.6%	-1.2%	-1.0%	-0.9%	1200
1000ms	-0.7%	-0.9%	-2.8%	-1.9%	-1.6%	1900
2000ms	-1.8%	-2.1%	-4.3%	-4.4%	-3.8%	3100

- Means no statistically significant change

- Strong negative impacts
- Roughly linear changes with increasing delay
- Time to Click changed by roughly double the delay

https://www.scribd.com/presentation/232007755/The-User-and-Business-Impact-of-Server-Delays-Additional-Bytes-And-HTTP-Chunking-in-Web-Search-Presentation

2009年Google

Search Traffic Impact

Type of Delay	Delay (ms)	Experiment Duration (weeks)	Impact on Average Daily Searches Per User
Pre-header	50	4	Not measurable
Pre-header	100	4	-0.20%
Post-header	200	6	-0.29%
Post-header	400	6	-0.59%
Post-ads	200	4	-0.30%

- Increase in abandonment heuristic = less satisfaction
  - Abandonment heuristic measures if a user stops interacting with search engine before they find what they are looking for
- Active users (users that search more often a priori) are more sensitive

https://services.google.com/fh/files/blogs/google_delayexp.pdf

Google

プライバシー・バイ・デザイン

プライバシー・バイ・デザイン

- 元々インターネットは平文の通信
  - e.g., HTTP over TCP

プライバシー・バイ・デザイン

- 元々インターネットは平文の通信
  - e.g., HTTP over TCP
- スノーデン事件 (2013年)
  - 米国家安全保障局 (NSA)の元契約社員が、NSAが市民の通話記録やネット企業のアクセス情報などを大量に収集していると暴露

プライバシー・バイ・デザイン

- 元々インターネットは平文の通信
  - e.g., HTTP over TCP
- スノーデン事件 (2013年)
  - 米国家安全保障局 (NSA)の元契約社員が、NSAが市民の通話記録やネット企業のアクセス情報などを大量に収集していると暴露
- GDPRなど、プライバシー保護を前提とした制度設計・システム設計を求める流れ

プライバシー・バイ・デザイン

- 元々インターネットは平文の通信
  - e.g., HTTP over TCP
- スノーデン事件 (2013年)
  - 米国家安全保障局 (NSA)の元契約社員が、NSAが市民の通話記録やネット企業のアクセス情報などを大量に収集していると暴露
- GDPRなど、プライバシー保護を前提とした制度設計・システム設計を求める流れ

↓

HTTPS for everything

プロコトル標準化への影響

- RFC 7258:
  “Pervasive Monitoring Is an Attack … should be mitigated in the design of IETF protocols, where possible”

プロコトル標準化への影響

- RFC 7258:
  “Pervasive Monitoring Is an Attack … should be mitigated in the design of IETF protocols, where possible”
- RFC 7624:
  “Protocols that encrypted their payload … (e.g., TLS) still expose all the information in their network- and transport-layer headers to the attacker…”

HTTP/2

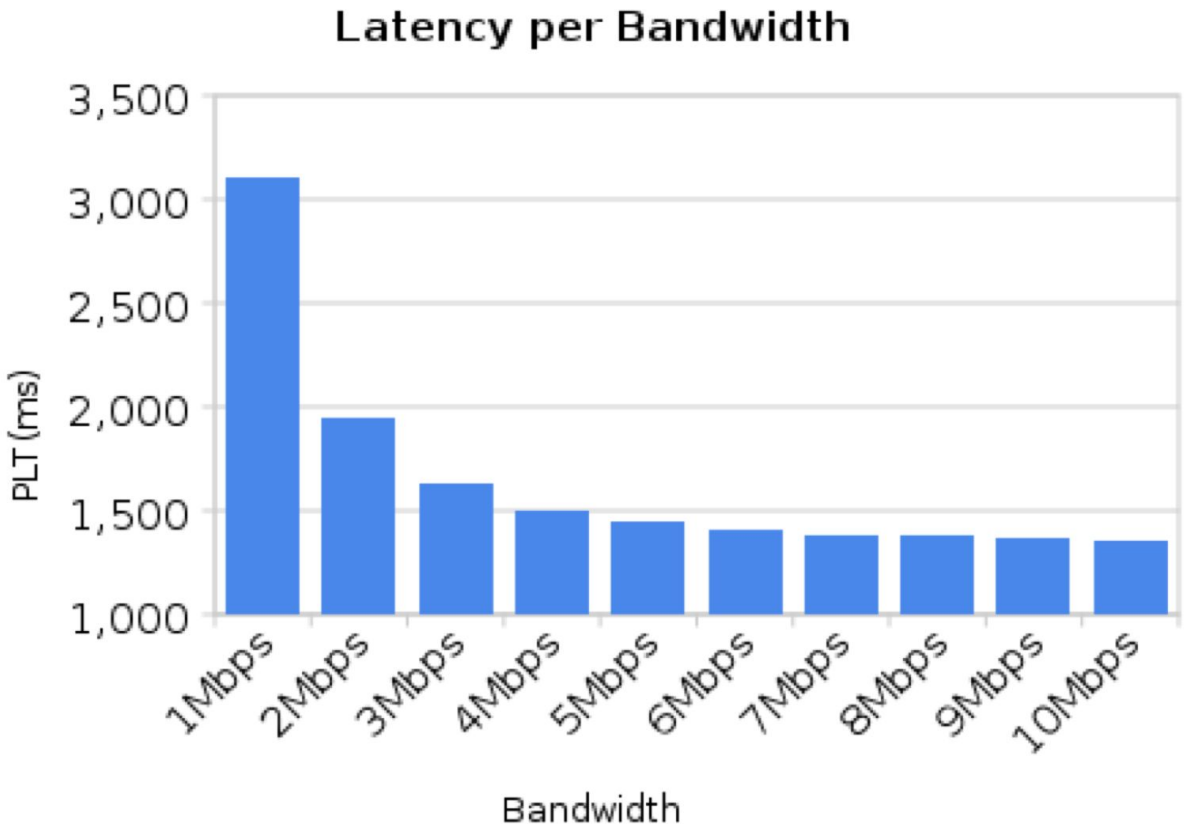
ニールセンの法則

- 「ハイエンドユーザーのネット接続は毎年　50%高速化」
  - つまり約17年で1,000倍
    - 1990年代 - ISDN ~128Kbps
    - 2010年頃 - FTTH 100Mbps~

ニールセンの法則

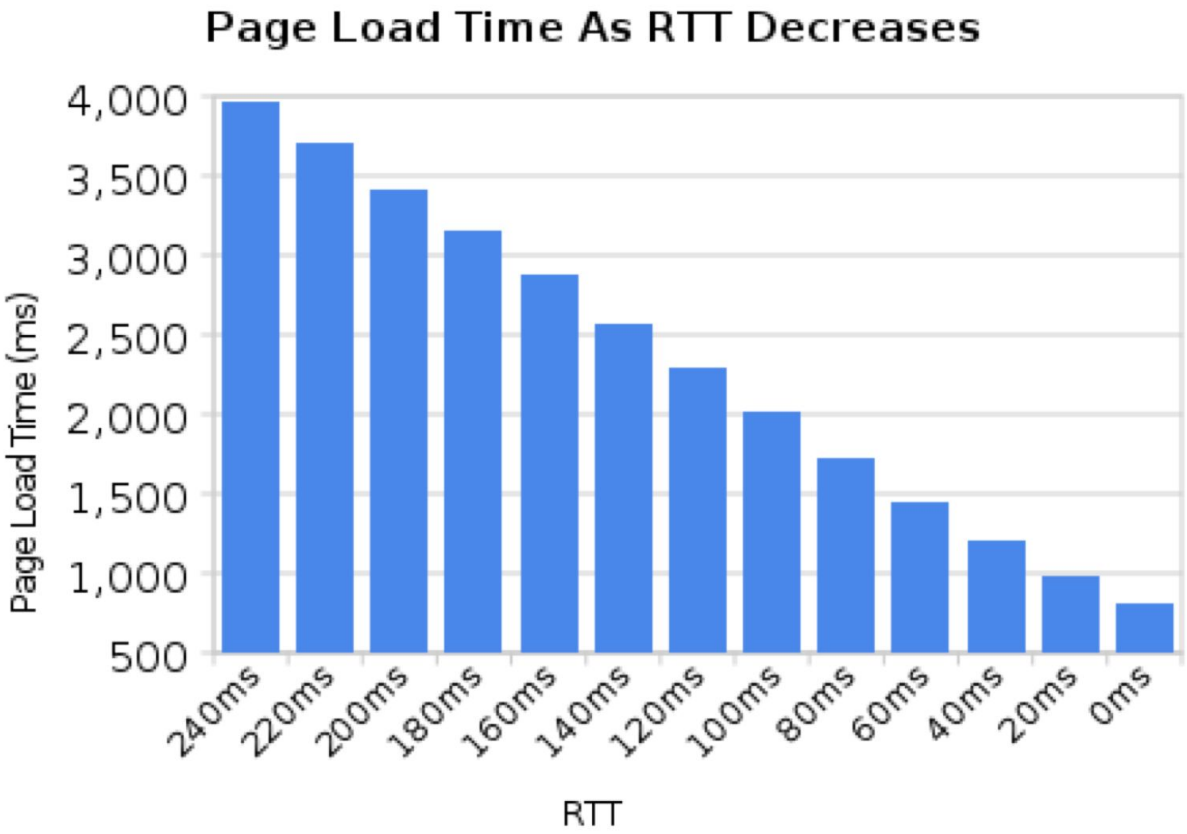
- 「ハイエンドユーザーのネット接続は毎年　50%高速化」
  - つまり約17年で1,000倍
    - 1990年代 - ISDN ~128Kbps
    - 2010年頃 - FTTH 100Mbps~
- HTTP/1.1のボトルネックがバンド幅 →レイテンシに

ロード時間はバンド幅に比例しない



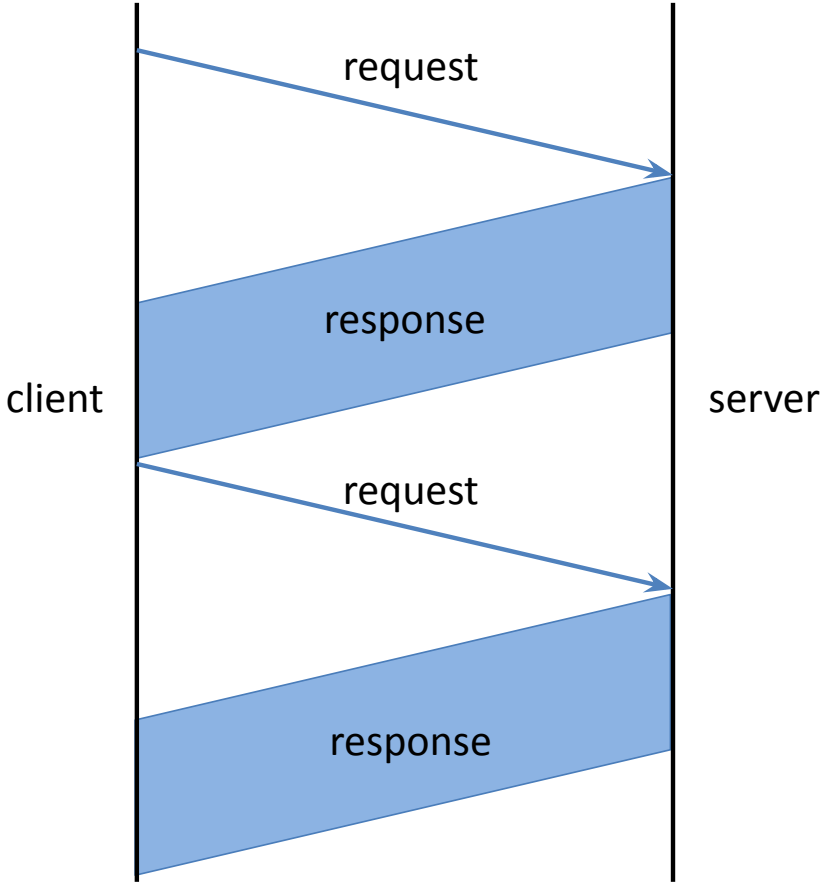
出典: More Bandwidth Doesn't Matter - 2011 Mike Belshe (Google)

ロード時間はレイテンシに比例

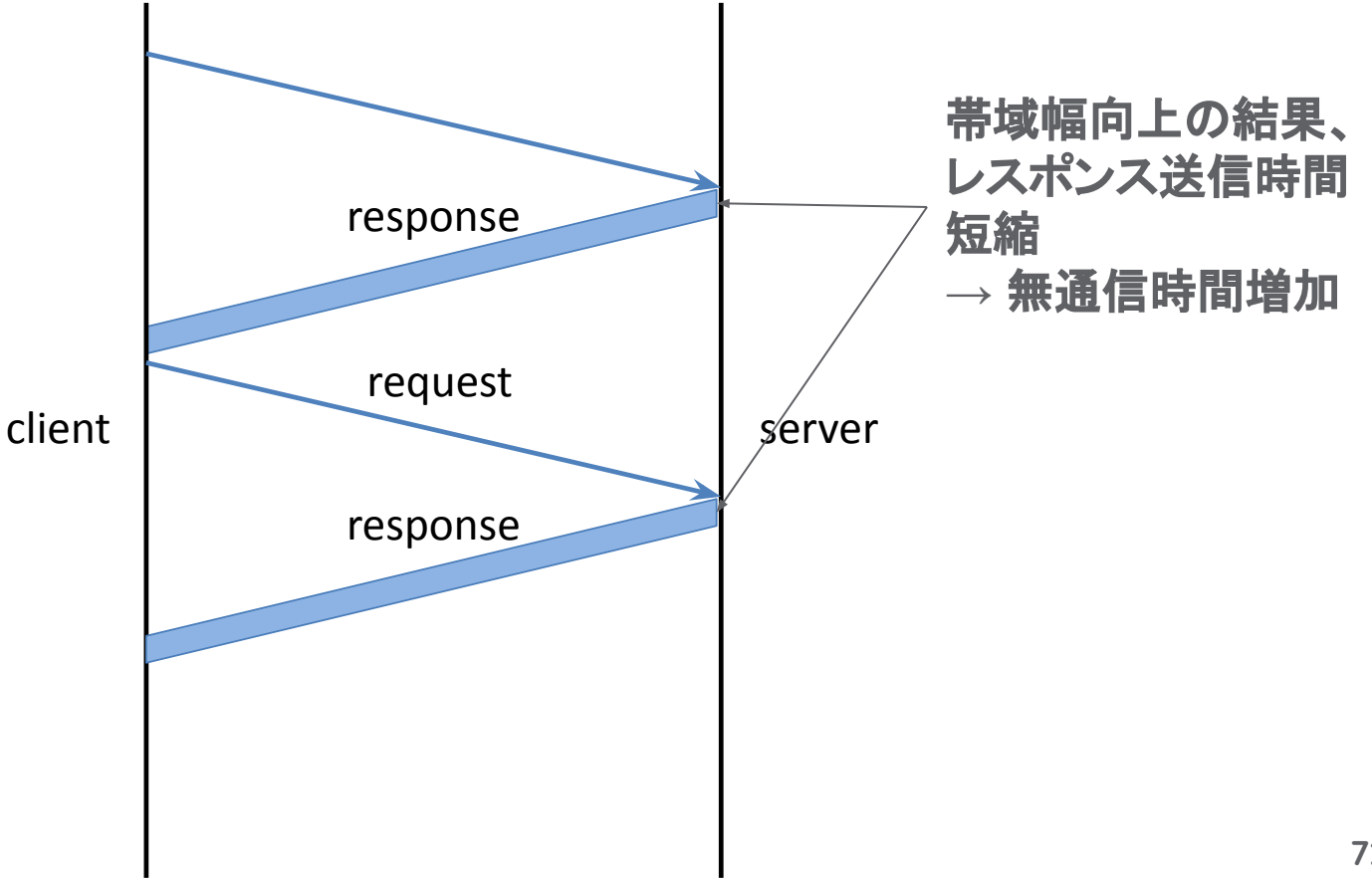


出典: More Bandwidth Doesn't Matter - 2011 Mike Belshe (Google)

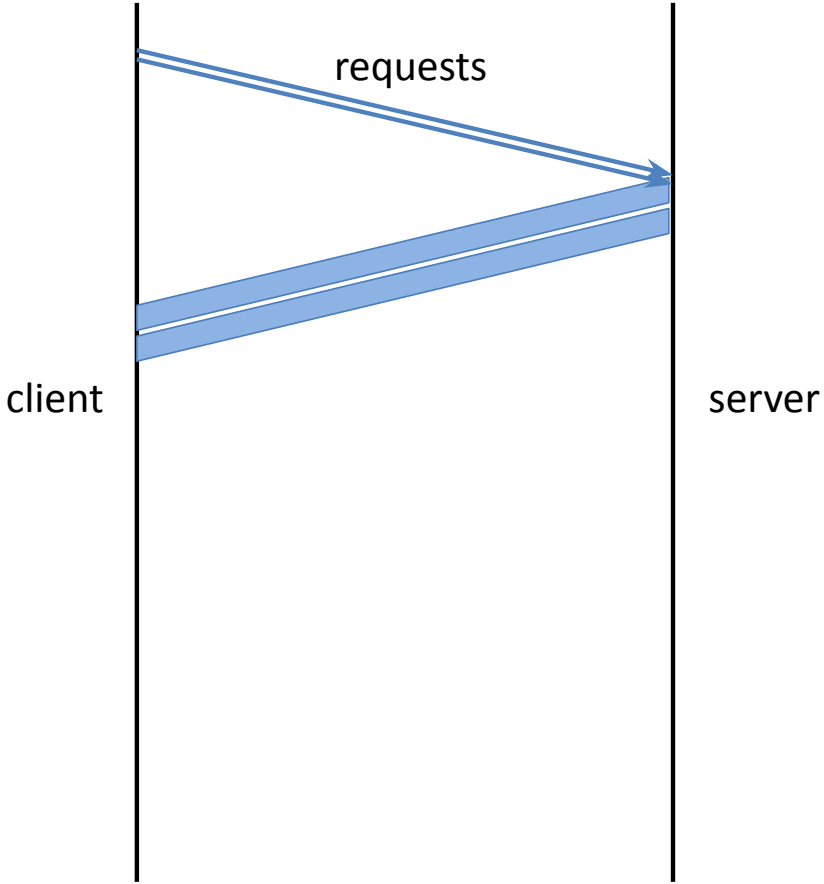
HTTP/1.1



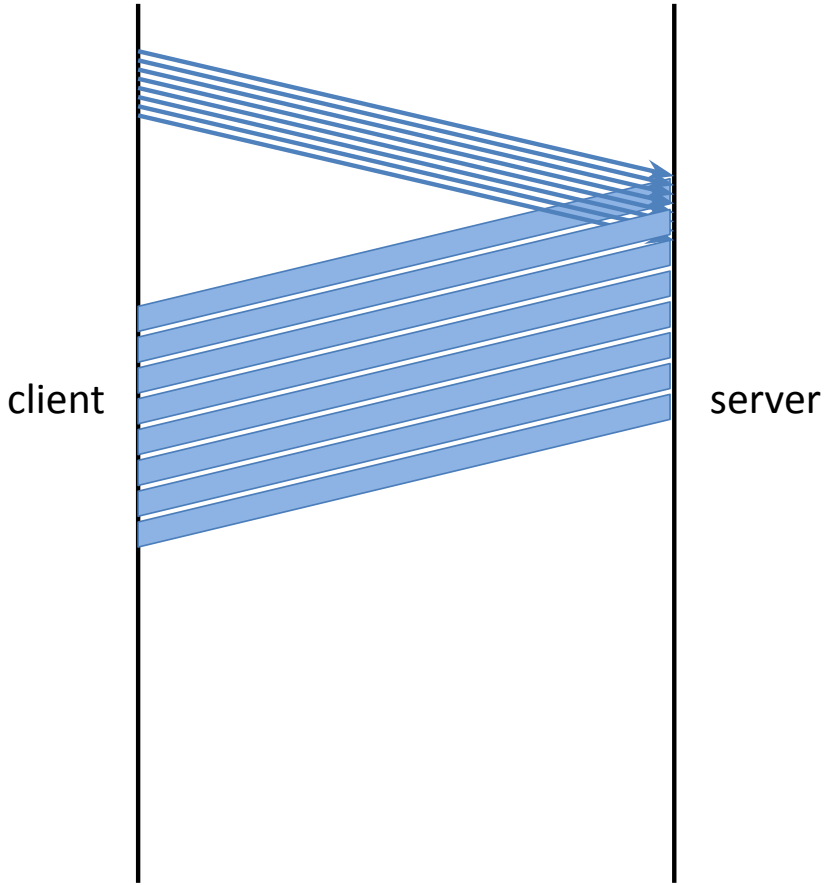
HTTP/1.1


client
server
response
request
response
帯域幅向上の結果、
レスポンス送信時間
短縮
→ 無通信時間増加

HTTP/2



HTTP/2



HTTP/2

- RFC 7540 (2015年)
  - GoogleのSPDYプロトコルを叩き台に標準化
- バイナリプロトコル
- リクエスト多重化
- ヘッダ圧縮 (HPACK: RFC 7541)
- (プッシュ)

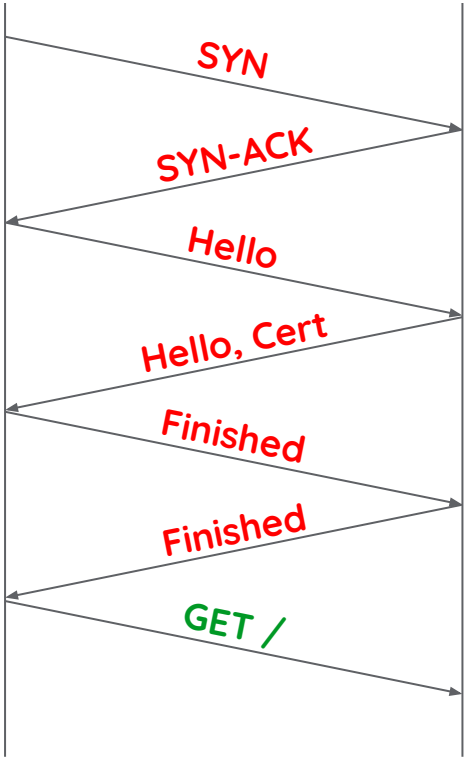
TLS/1.3

TLS/1.3

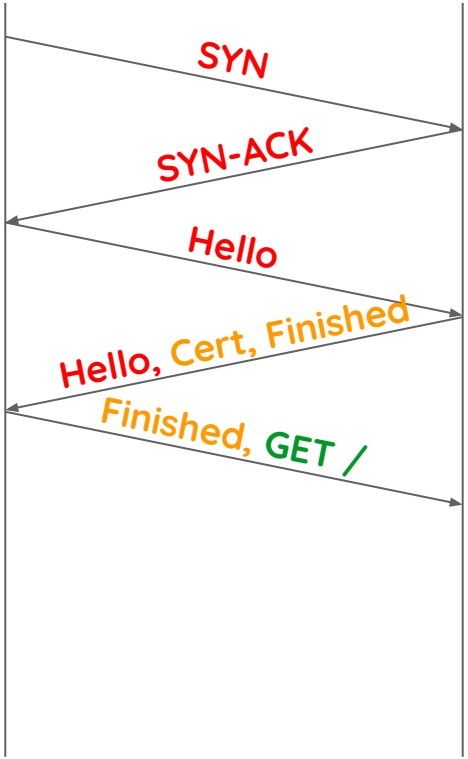
- RFC 8446 (2018年)
- TLS/1.2の問題点を解決：
  - 接続確立に2RT
  - 証明書を平文で交換
    - クライアント証明書が経路上見えるのが特に問題

TLS/1.3

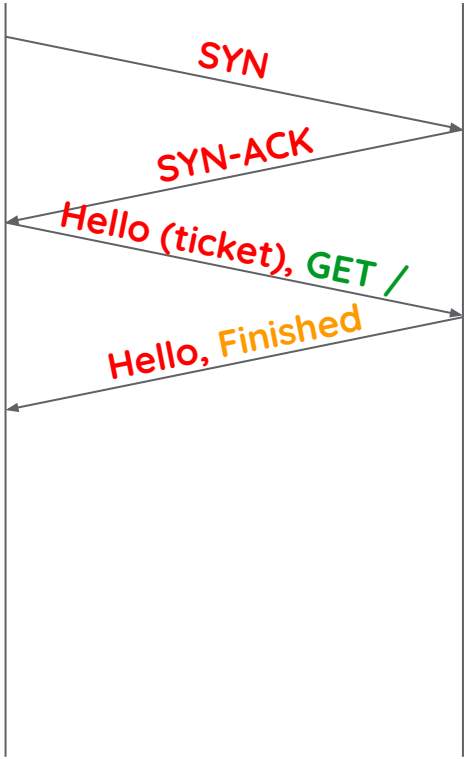
赤: 平文
橙: 匿名暗号化
緑: 暗号化(証明書に紐づいた鍵によって認証)



TLS 1.2



TLS 1.3



TLS 1.3 (0-RTT再接続)

プライバシー保護・攻撃耐性

	HTTP	HTTPS (TLS/1.2)	HTTPS (TLS/1.3)	HTTPS (TLS/1.3+ECH)
IPアドレス/ポート	×	×	×	×
送受信データ量	×	×	×	×
接続サーバ名	×	×	×	⚪
証明書	×	×	⚪	⚪
HTTP上のデータ	×	⚪	⚪	⚪
RST注入攻撃耐性	×	×	×	×

Encrypted Client Hello (ECH)

- TLSのサーバ証明書選択：
  - クライアントが送る Hello内のserver_name拡張でサーバ名を指定
  - サーバは指定されたサーバの証明書を使って応答

Encrypted Client Hello (ECH)

- TLSのサーバ証明書選択：
  - クライアントが送る Hello内のserver_name拡張でサーバ名を指定
  - サーバは指定されたサーバの証明書を使って応答
- server_name拡張は平文なので経路上で監視可能

Encrypted Client Hello (ECH)

- TLSのサーバ証明書選択：
  - クライアントが送る Hello内のserver_name拡張でサーバ名を指定
  - サーバは指定されたサーバの証明書を使って応答
- server_name拡張は平文なので経路上で監視可能
- Encrypted Client Hello:
  - DNSでサーバの IPアドレスと同時に公開鍵を返し

Encrypted Client Hello (ECH)

- TLSのサーバ証明書選択：
  - クライアントが送る Hello内のserver_name拡張でサーバ名を指定
  - サーバは指定されたサーバの証明書を使って応答
- server_name拡張は平文なので経路上で監視可能
- Encrypted Client Hello:
  - DNSでサーバの IPアドレスと同時に公開鍵を返し
  - クライアントはその鍵で server_nameを暗号化

Encrypted Client Hello (ECH)

- TLSのサーバ証明書選択：
  - クライアントが送る Hello内のserver_name拡張でサーバ名を指定
  - サーバは指定されたサーバの証明書を使って応答
- server_name拡張は平文なので経路上で監視可能
- Encrypted Client Hello:
  - DNSでサーバの IPアドレスと同時に公開鍵を返し
  - クライアントはその鍵で server_nameを暗号化
  - RFC 9849として公開間近

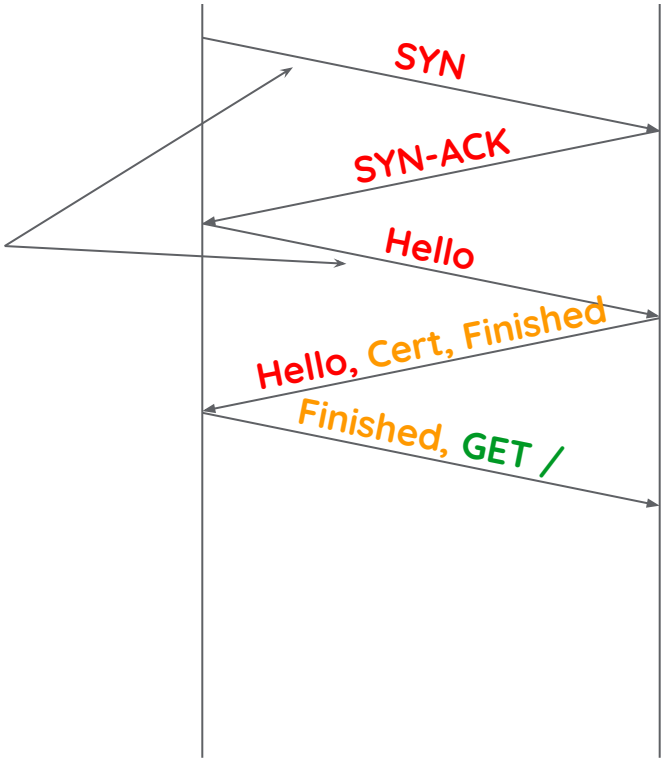
さらなる高速化を目指して

さらなる高速化を目指して

- 接続RT削減 - TCP FastOpen
- ヘッドオブラインブロッキング解消　- Minion

接続RT削減

TCPとTLSの
ハンドシェイク同
時にやりたい



TLS 1.3

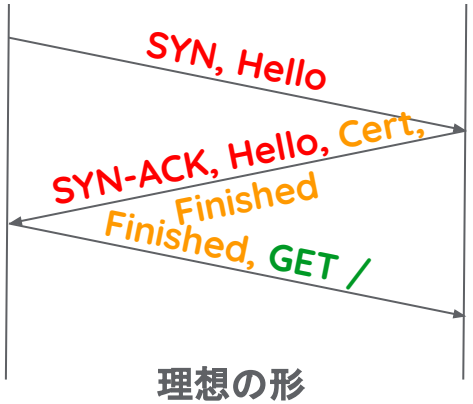
TCP FastOpen

- TCPとTLSのハンドシェイクを同時にやりたい
- TCP FastOpen:
  - SYNにデータを載せる拡張
    - 保守的な設計で再接続時だけ

理想の形

TCP FastOpen

- TCPとTLSのハンドシェイクを同時にやりたい
- TCP FastOpen:
  - SYNにデータを載せる拡張
    - 保守的な設計で再接続時だけ
- 2011: Internet-Draft



理想の形

TCP FastOpen

- TCPとTLSのハンドシェイクを同時にやりたい
- TCP FastOpen:
  - SYNにデータを載せる拡張
    - 保守的な設計で再接続時だけ
- 2011: Internet-Draft
- 2014: RFC 7413

理想の形

TCP FastOpen

- TCPとTLSのハンドシェイクを同時にやりたい
- TCP FastOpen:
  - SYNにデータを載せる拡張
    - 保守的な設計で再接続時だけ
- 2011: Internet-Draft
- 2014: RFC 7413
- 2017: Apple「20%くらい失敗するわ」

理想の形

https://archive.nanog.org/sites/default/files/Paasch_Network_Support.pdf

TCP FastOpen失敗の理由

- クライアントの SYN+dataをブロック
  - 中継装置※がSYNにデータがついてる可能性を考えていない

理想の形

※中継装置 = ファイアウォールだったりステートフル NATだったりIDS(攻撃検知システム)だったり
https://archive.nanog.org/sites/default/files/Paasch_Network_Support.pdf

TCP FastOpen失敗の理由

- クライアントの SYN+dataをブロック
  - 中継装置※がSYNにデータがついてる可能性を考えていない
- サーバの SYN-ACKをブロック
  - SYN-ACKのackno = SYNのseqno + データサイズ
  - 中継装置はackno = SYNのseqno以外無視

理想の形

※中継装置 = ファイアウォールだったりステートフル NATだったりIDS(攻撃検知システム)だったり
https://archive.nanog.org/sites/default/files/Paasch_Network_Support.pdf

TCP FastOpen失敗の理由

- クライアントの SYN+dataをブロック
  - 中継装置※がSYNにデータがついてる可能性を考えていない
- サーバの SYN-ACKをブロック
  - SYN-ACKのackno = SYNのseqno + データサイズ
  - 中継装置はackno = SYNのseqno以外無視
- 攻撃判定してクライアント IPまるごとブロックする IDS

理想の形

※中継装置 = ファイアウォールだったりステートフル NATだったりIDS(攻撃検知システム)だったり
https://archive.nanog.org/sites/default/files/Paasch_Network_Support.pdf

TCPを「理解」する中継装置

- ステートフル NAT
- ファイアウォール
- IDS(攻撃検知システム)

TCPを「理解」する中継装置

- ステートフル NAT
- ファイアウォール
- IDS(攻撃検知システム)
- Performance Enhancing Proxy (PEP)

TCPを「理解」する中継装置

- ステートフル NAT
- ファイアウォール
- IDS(攻撃検知システム)
- Performance Enhancing Proxy (PEP)
  - split PEP (terminating proxy)
  - non-split PEP

Split PEP

- 乗客のTCP接続を機内のプロキシで終端
- 衛星区間は独自プロトコルで地上ゲートウェイへ中継

Non-split PEP

- TCP接続を分断しないまま「最適化」

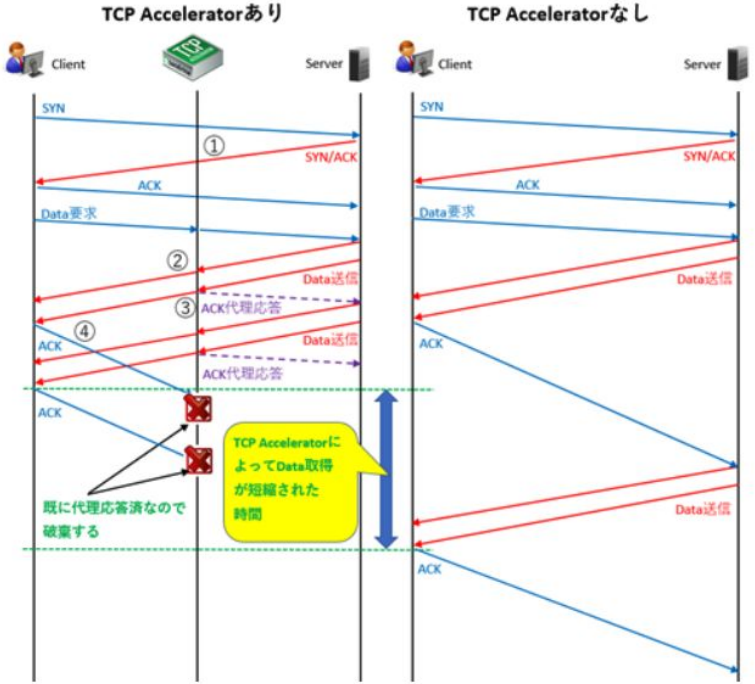


図7. TCP Acceleratorシーケンス例 1

https://www.netone.co.jp/knowledge-center/blog-column/knowledge_takumi_143/

TCPの「硬直化」

- これらの装置は、それぞれに目的がある
  - ステートフル NAT
  - ファイアウォール
  - IDS(攻撃検知システム)
  - Performance Enhancing Proxy (PEP)

TCPの「硬直化」

- これらの装置は、それぞれに目的がある
  - ステートフル NAT
  - ファイアウォール
  - IDS(攻撃検知システム)
  - Performance Enhancing Proxy (PEP)
- だが、TCPの「現在の挙動」に依存

TCPの「硬直化」

- これらの装置は、それぞれに目的がある
  - ステートフル NAT
  - ファイアウォール
  - IDS(攻撃検知システム)
  - Performance Enhancing Proxy (PEP)
- だが、TCPの「現在の挙動」に依存
  - TCP FastOpenのような単純な拡張でさえ通すのが難しい

ヘッドオブラインブロッキング

- HTTP/2でマルチ「ストリーム」化
- ストリーム 1がパケロスしてもストリーム 2は使えるはず
- TCPスタックが順番にしか読ませてくれない

Minion (2011)

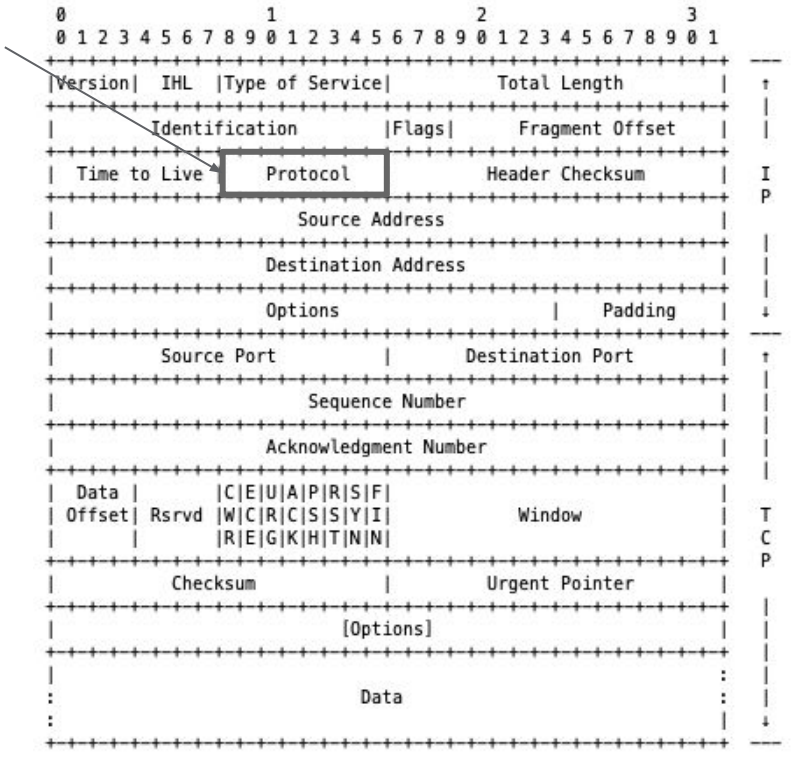
- TCPスタックに順序無視した読み書き APIを追加
  - SO_UNORDERED_SEND / RECV
- ブロックは全て「 0…非ゼロ…0」の形にエンコード
  - 抜けがあっても 0を探せばブロックが見つかる

Minion (2011)

- 問題:
  - 複雑
  - 標準化して、各 OSが対応するまで性能向上しない
    - TCP FastOpenでさえ標準化まで 3年、実験まで 6年
  - OS対応しても、中継装置が「理解」してると速度向上しない

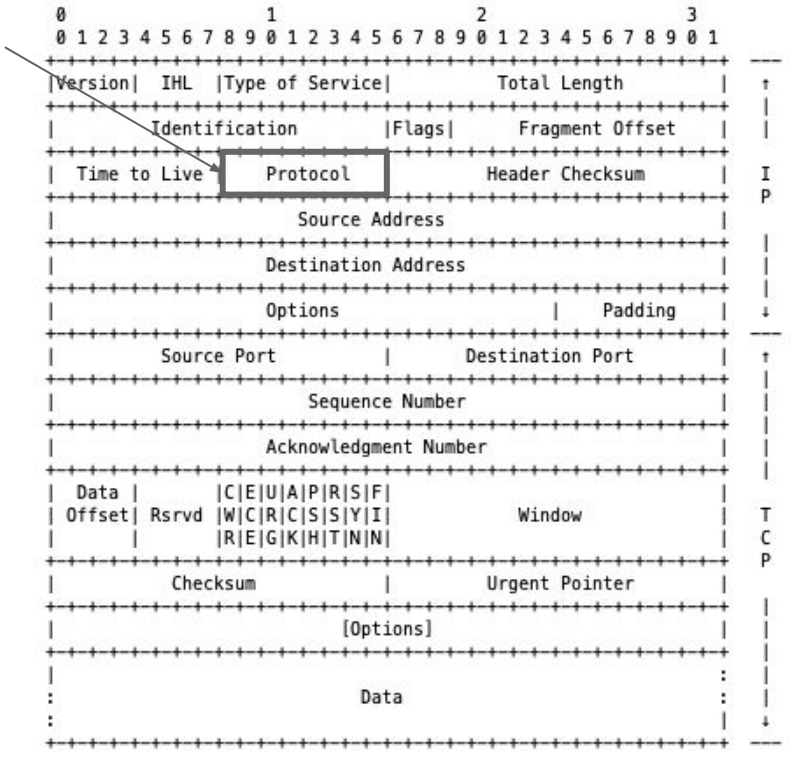
SCTP

- マルチストリーム対応
- protocol=132 (tcp=6, udp=17)



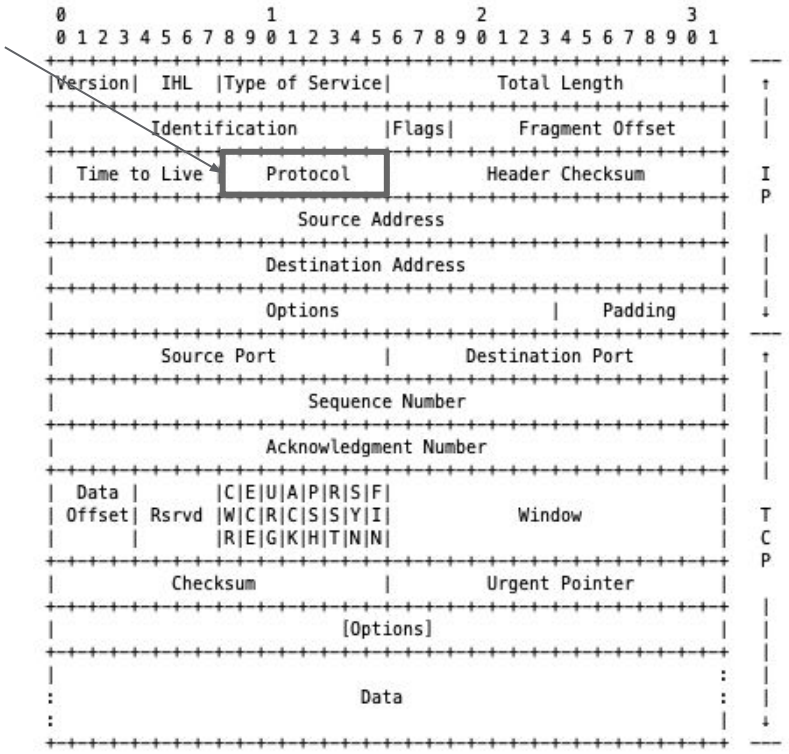
SCTP

- マルチストリーム対応
- protocol=132 (tcp=6, udp=17)
- 132はNAT等通らないので
  UDP上でもSCTP利用できる



SCTP

- マルチストリーム対応
- protocol=132 (tcp=6, udp=17)
- 132はNAT等通らないので
  UDP上でもSCTP利用できる
- 現実には暗号化するので
  DTLS (UDP上で動くTLS) 上で
  使用



SCTPの問題

- 接続確立に 2RT
  - DTLSのフルハンドシェイクに 1RT
  - SCTPのハンドシェイクに 1RT

SCTPの問題

- 接続確立に 2RT
  - DTLSのフルハンドシェイクに 1RT
  - SCTPのハンドシェイクに 1RT
- メッセージ間の interleaveがない
  - 大きなHTTPレスポンス送信中に小さなレスポンスを割り込ませることができない
    - HTTP/2でもできるのに ...

SCTPの問題

- 接続確立に 2RT
  - DTLSのフルハンドシェイクに 1RT
  - SCTPのハンドシェイクに 1RT
- メッセージ間の interleaveがない
  - 大きなHTTPレスポンス送信中に小さなレスポンスを割り込ませることができない
    - HTTP/2でもできるのに ...
  - RFC 8260で後に解決

第3世代HTTPの要件

性能要件

- 1RTで接続確立
  - トランスポートと TLSハンドシェイクを同時実行

性能要件

- 1RTで接続確立
  - トランスポートと TLSハンドシェイクを同時実行
- HTTPリクエスト間で
  - ヘッドオブラインブロッキングがない
    - インターリーブできることを含む

性能要件

- 1RTで接続確立
  - トランスポートと TLSハンドシェイクを同時実行
- HTTPリクエスト間で
  - ヘッドオブラインブロッキングがない
    - インターリーブできることを含む

↓

この時点で新プロトコル確定

実装基盤

- TCPの2つの問題:
  - 中継装置による硬直化
  - カーネル実装であるがゆえに歩みが遅い

実装基盤

- TCPの2つの問題:
  - 中継装置による硬直化
  - カーネル実装であるがゆえに歩みが遅い
- TCP/UDP以外はNAT/ファイアウォールを越えられない

実装基盤

- TCPの2つの問題:
  - 中継装置による硬直化
  - カーネル実装であるがゆえに歩みが遅い
- TCP/UDP以外はNAT/ファイアウォールを越えられない

↓

この時点でUDP上で実装確定

実装基盤

- TCPの2つの問題:
  - 中継装置による硬直化
  - カーネル実装であるがゆえに歩みが遅い
- TCP/UDP以外はNAT/ファイアウォールを越えられない

↓

この時点でUDP上で実装確定

残された唯一の選択肢。ここの硬直化を許したらゲームオーバー

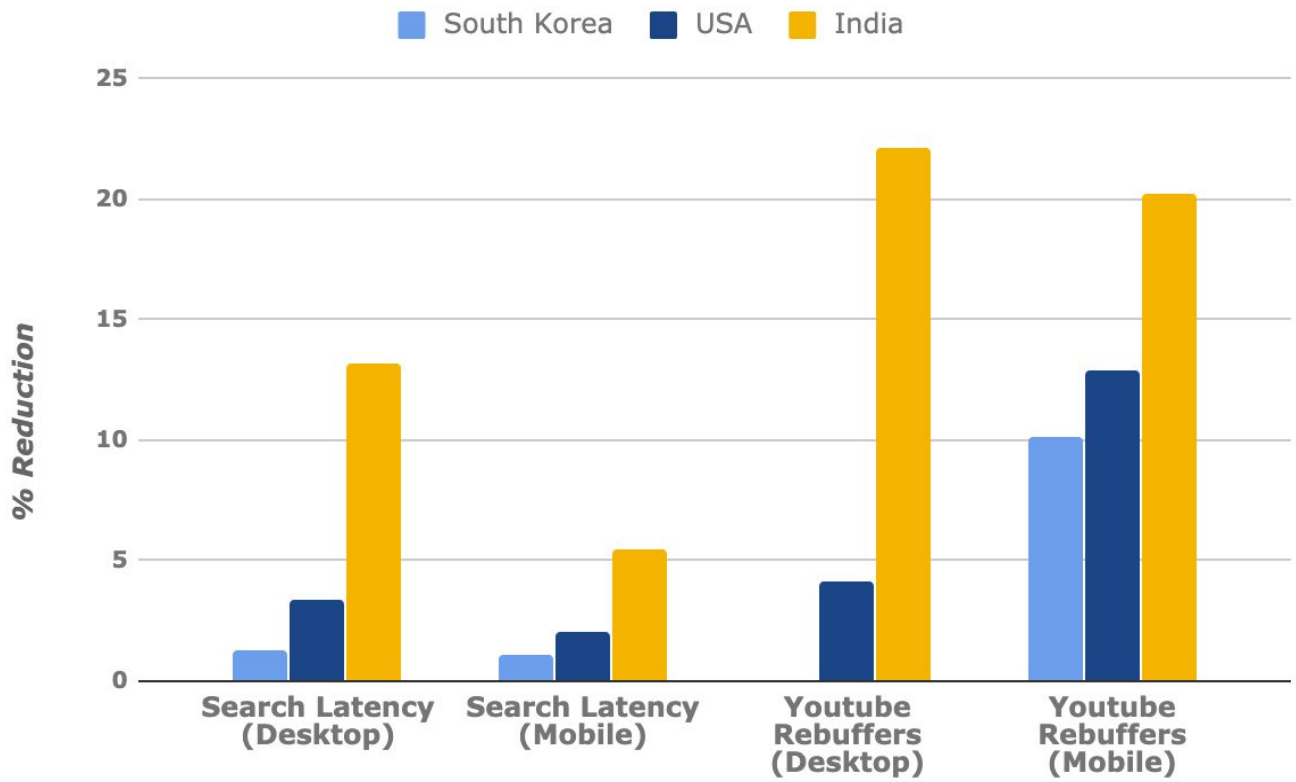
Google QUICからIETF QUICへ

- Chrome上に「QUIC」という独自プロトコルを実装
  - 性能要件と実装基盤は前述のとおり

Google QUICからIETF QUICへ

- Chrome上に「QUIC」という独自プロトコルを実装
  - 性能要件と実装基盤は前述のとおり
  - 暗号化は独自方式

Google QUIC (2017)



https://research.google.com/pubs/archive/46403.pdf

Google QUICからIETF QUICへ

- Chrome上に「QUIC」という独自プロトコルを実装
  - 性能要件と実装基盤は前述のとおり
  - 暗号化は独自方式
- 実績をもとに、IETFに標準化を持ちかけ

Google QUICからIETF QUICへ

- Chrome上に「QUIC」という独自プロトコルを実装
  - 性能要件と実装基盤は前述のとおり
  - 暗号化は独自方式
- 実績をもとに、IETFに標準化を持ちかけ
- 2016年10月「QUIC Working Group」誕生

Key Goals for QUIC

- Minimizing connection establishment and overall transport latency for applications, starting with HTTP/2;
- Providing multiplexing without head-of-line blocking;
- Requiring only changes to path endpoints to enable deployment;
- Enabling multipath and forward error correction extensions; and
- Providing always-secure transport, using TLS 1.3 by default.

https://datatracker.ietf.org/doc/charter-ietf-quic/01/

QUICの設計ポイント

QUICとバージョン

- TCP制定以来約40年ぶりの再設計(を狙う)
- 硬直化を2度と起こさない

QUICとバージョン

- TCP制定以来約40年ぶりの再設計(を狙う)
- 硬直化を2度と起こさない
- バージョン番号の導入
  - Invariants - バージョンが変わっても不変な要素
  - version 1 - 最初に作るQUIC

Invariants

Invariants - long header packet

```
Long Header Packet {
  Header Form (1) = 1,
  Version-Specific Bits (7),
  Version (32),
  Destination Connection ID Length (8),
  Destination Connection ID (0..2040),
  Source Connection ID Length (8),
  Source Connection ID (0..2040),
  Version-Specific Data (..),
}
```

先頭バイトの MSBは常に1

バージョン番号
（long header packetは接続確立・ステートレス動作用なので、バージョン番号がある）

送信先 Connection ID(0～255バイト)

送信元 Connection ID(0～255バイト)

Invariants - short header packet

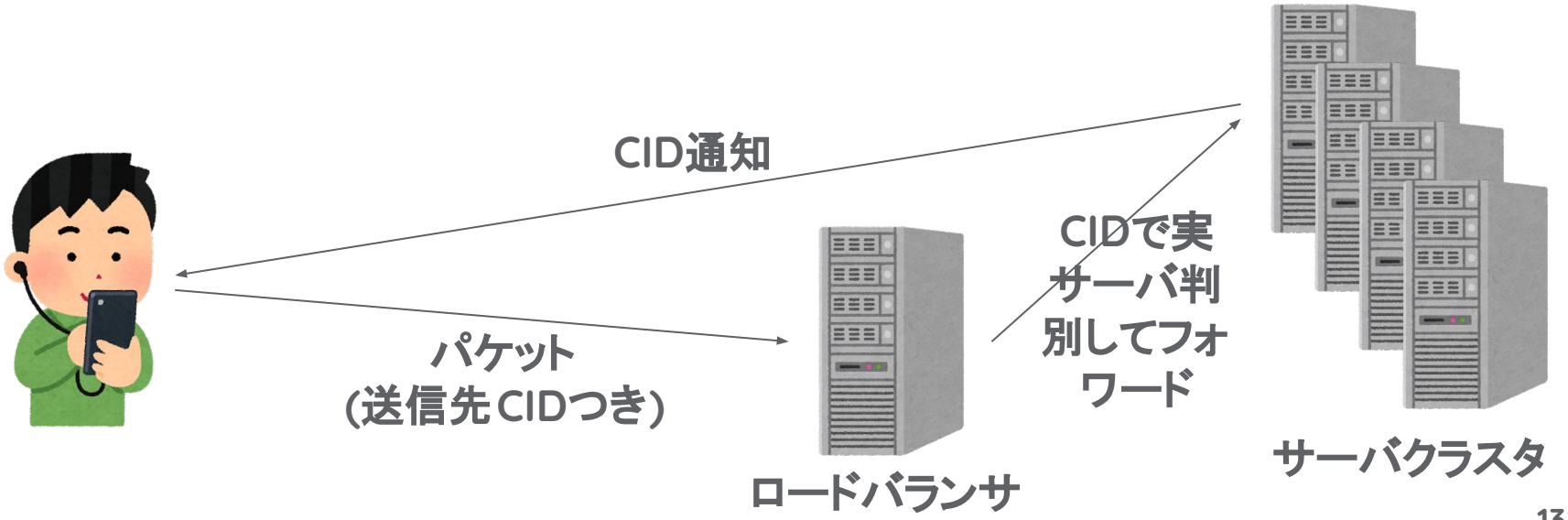
```
Short Header Packet {
  Header Form (1) = 0,
  Version-Specific Bits (7),
  Destination Connection ID (0..2040),
  Version-Specific Data (..),
}
```

先頭バイトの MSBは常に0

送信先 Connection ID(0〜255バイト)（Lengthフィールドはない：接続確立後に使う前提なので、受信側はConnection IDの長さを知っているはず）

Invariants

- CIDがあるから同一ポート使い回し可能（枯渇の心配なし）
- バージョン非依存でロードバランサ実現
- クライアントの IPアドレス/ポートが変わっても通信持続



Invariants - Version Negotiation

```
Version Negotiation Packet {
  Header Form (1) = 1,
  Unused (7),
  Version (32) = 0,
  Destination Connection ID (0..2040),
  Supported Version (32) ...,
}
```

long header packet!

version=0がバージョンネゴシエーションパケット
（未知のバージョンの long header packetへの応答で使用）

対応バージョンの一覧

Invariants

- とにかく中継機器が見ていいのはこれだけ
  - あとはQUICのバージョンごとに変わるので

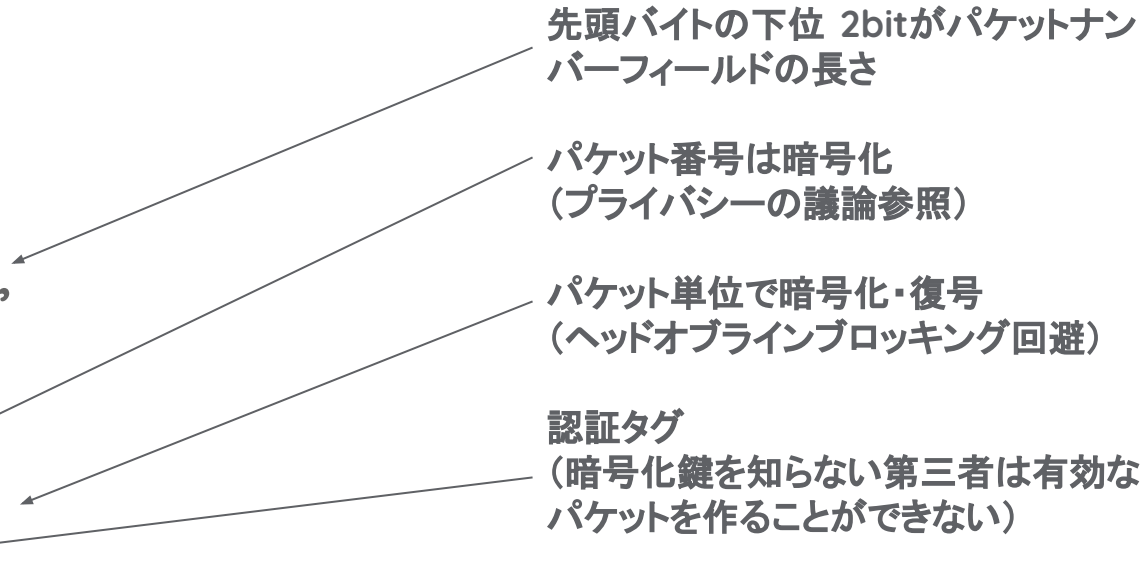
Invariants

- とにかく中継機器が見ていいのはこれだけ
  - あとはQUICのバージョンごとに変わるので
- 見ていいフィールド
  - Version
  - Destination / Source Connection ID

QUIC version 1

パケット暗号化

```
QUICv1 Packet {
  Header Form (1),
  … (5),
  Packet Number Bits (2),
  …
  Packet Number (8..32),
  Encrypted Payload (…),
  AEAD Tag (…),
}
```



パケット暗号化

1. ペイロードを共通鍵とパケット番号で
   AEAD暗号化しAEADタグ出力
2. パケット番号フィールドから 4バイト後
   （暗号化されたペイロードの先頭付近）
   から16バイト抽出
3. これをブロック暗号で暗号化してマスク生成
4. このマスクをパケット番号やヘッダ等に XOR

```
QUICv1 Packet {
  Header Form (1),
  … (5),
  Packet Number Bits (2),
  …
  Packet Number (8..32),
  Encrypted Payload (…),
  AEAD Tag (…),
}
```

パケット復号

1. パケット番号フィールドから 4バイト後（暗号化されたペイロードの先頭付近）から16バイト抽出
2. これをブロック暗号で暗号化してマスク生成
3. このマスクをパケット番号やヘッダ等に XOR
4. パケット番号が復号できたので、これを使ってペイロードをAEAD復号

```
QUICv1 Packet {
  Header Form (1),
  … (5),
  Packet Number Bits (2),
  …
  Packet Number (8..32),
  Encrypted Payload (…),
  AEAD Tag (…),
}
```

パケットヘッダ暗号化の理由

- プライバシー対策
  - ハンドオーバー (IPアドレス変化)した際、パケット番号を利用したトラッキングを防ぐため等
- 硬直化対策
  - パケット番号の順序を利用する中継機器や IDSの出現を防ぐ
    - パケットが逆順で届いてもストリームが別なら利用可能だし…

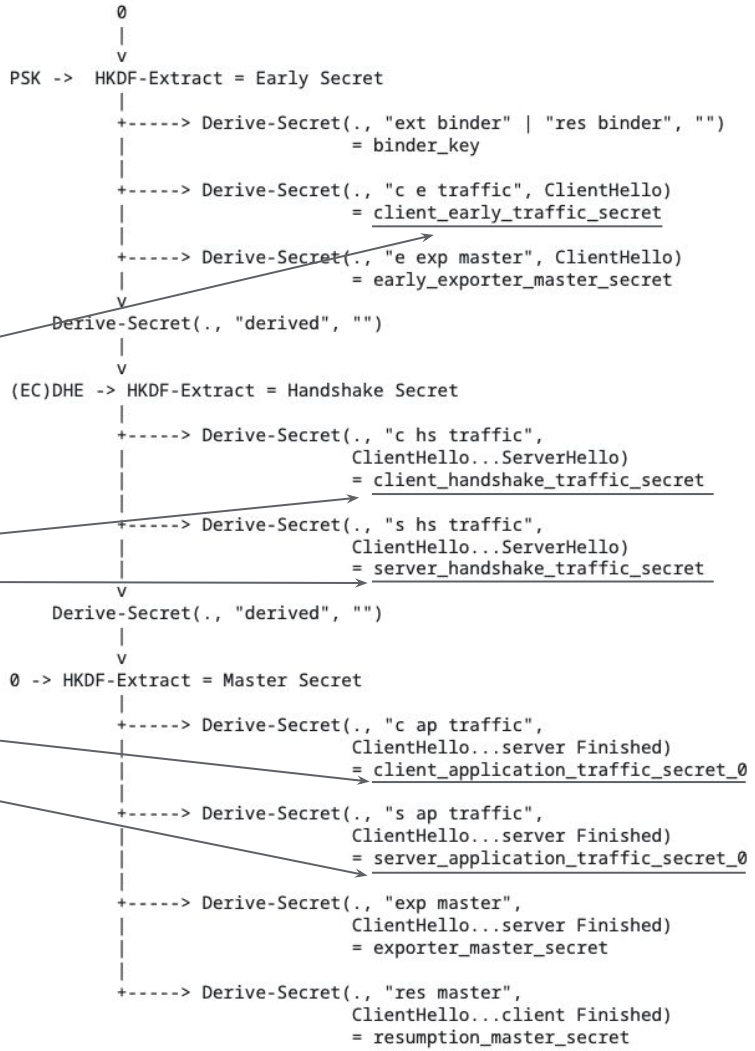
パケットの種類とパケット番号空間

Retry以外の 4種類のパケットはハンドシェイク～データ　通信で利用
Retryは過負荷時等にハンドシェイク開始の遅延に利用

	種類	番号空間
long header	Initial	Initial
	0-RTT	Application Data
	Handshake	Handshake
	Retry	N/A
short header	1-RTT	Application Data

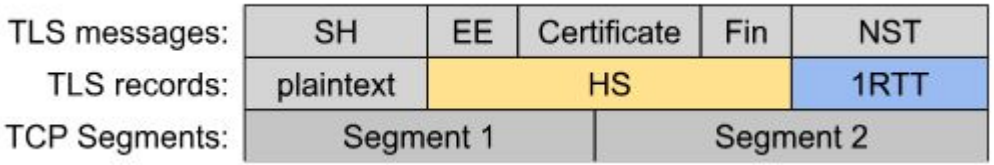
パケットの種類

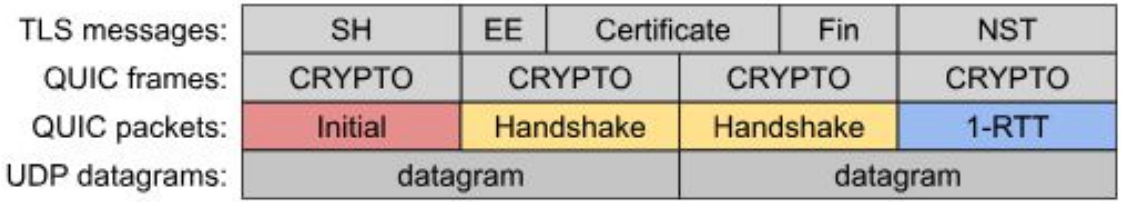
- Retryを除くと4種類
- TLS/1.3の暗号鍵生成に呼応
  - Initial
  - 0-RTT
  - Handshake
  - 1-RTT
- Initialの暗号鍵は固定シード +CIDから生成



TLS/1.3は2階層のプロトコル

	機能	最大サイズ
ハンドシェイクメッセージ	鍵交換・相手の認証	16MB
レコード	暗号化	16KB





パケットの種類とパケット番号空間

Retry以外の4種類のパケットはハンドシェイク～データ　通信で利用（それぞれ TLSの暗号鍵生成に呼応）
Retryは過負荷時等にハンドシェイク開始の遅延に利用

	種類	番号空間
long header	Initial	Initial
	0-RTT	Application Data
	Handshake	Handshake
	Retry	N/A
short header	1-RTT	Application Data

暗号鍵ごとにパケット番号空間を分けることで、第三者も知りうる鍵で暗号化された InitialのACKがハンドシェイクやデータ通信に影響を与える可能性を防ぐ

0-RTTと1-RTTはどちらも保護されたアプリケーションデータなので共有（0-RTTは別接続としてリプレイされるのが差、接続内では同等で良い）

v1 long header packet

```
v1 Long Header Packets {
  Header Form (1) = 1,
  Fixed Bit (1) = 1,
  Long Packet Type (2),
  Reserved (2),
  Packet Number Length (2),
  Version (32) = 1,
  Destination Connection ID Length (8),
  Destination Connection ID (0..160),
  Source Connection ID Length (8),
  Source Connection ID (0..160),
  Token Length (i),
  Token (..),
  Length (i),
  Packet Number (8..32),
  Encrypted Payload (..),
  AEAD Tag (..),
}
```

※Retry以外のフォーマット

パケットタイプ：0=Initial, 1=0-RTT, 2=Handshake, 3=Retry

v1のCIDは最大20バイト

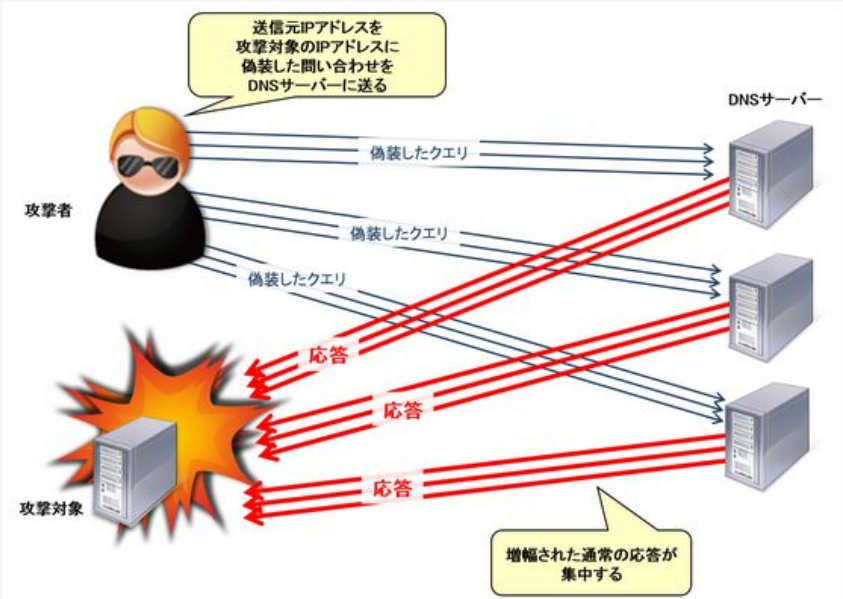
Initialのみ。後述

ハンドシェイク中は小さいパケットが多い→長さフィールドをつけて、複数のQUIC v1パケットをひとつの UDPデータグラムに格納

アンプ攻撃とは？

DNSリフレクター攻撃（DNSアンプ攻撃）

DNSを利用したDoS攻撃の一つです。送信元IPアドレスを偽った通常のDNS問い合わせをDNSサーバーに送ることでDNSサーバーが応答パケットを攻撃対象に送り、サービス不能の状態に陥らせます。

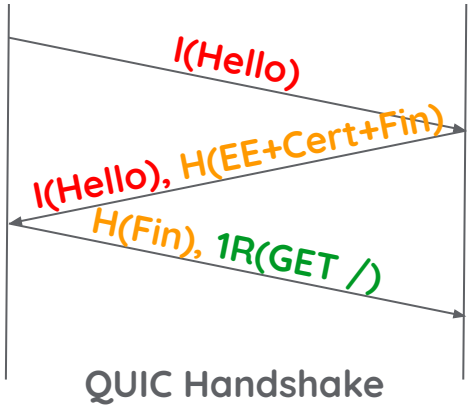


この攻撃手法では、攻撃者が送信元IPアドレスを攻撃対象のIPアドレスに偽装した問い合わせを通常のDNSサーバーに送ります。DNSサーバーは、攻撃対象に通常のDNS応答を返すため、攻撃者からみるとDNSサーバーが攻撃を反射（リフレクター（reflector）/ リフレクション（reflection））しているように見えます。DNSでは、問い合わせよりも応答の方が大きいため、小さなクエリパケットで大きな攻撃パケットを生成（攻撃を増幅）できることから「DNSアンプ攻撃（DNS Amplification Attacks）」とも呼ばれます。

https://jprs.jp/glossary/index.php?ID=0156

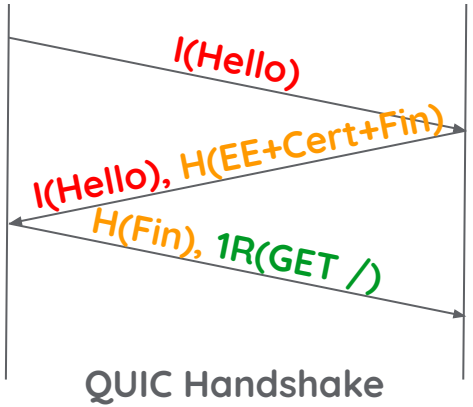
QUICとアンプ攻撃（1）

- サーバは、Initialパケット(TLS Hello)を受信すると一気にハンドシェイク完了まで応答したい
  - 証明書チェインが大きい



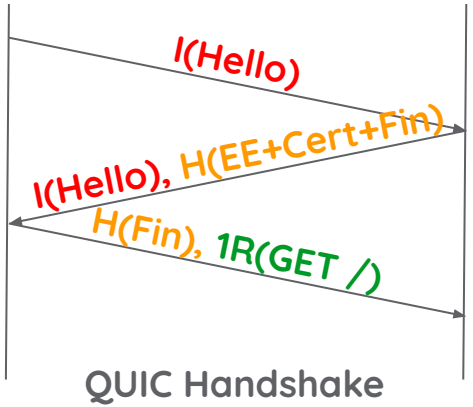
QUICとアンプ攻撃（2）

- クライアントはフルサイズ UDPパケット送信



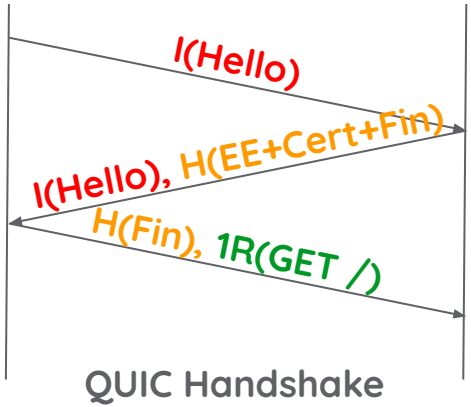
QUICとアンプ攻撃（2）

- クライアントはフルサイズ UDPパケット送信
- サーバはフルサイズ x3までに制限
  - 証明書チェインぎりぎり入る
    - gzip / bzip2圧縮すれば…
      - RFC 8879



QUICとアンプ攻撃（2）

- クライアントはフルサイズ UDPパケット送信
- サーバはフルサイズ x3までに制限
  - 証明書チェインぎりぎり入る
    - gzip / bzip2圧縮すれば…
      - RFC 8879
- サーバが受信量の 3倍以上送っていいのは、クライアントが経路上にいることを確認してから (Address Validation)



Address Validation

- クライアントはサーバの Initial(Hello)を受信すると Handshakeパケットの暗号鍵を作れる
  - クライアントの Handshakeパケットを受信 → OK!

Address Validation

- クライアントはサーバの Initial(Hello)を受信すると Handshakeパケットの暗号鍵を作れる
  - クライアントの Handshakeパケットを受信 → OK!
- 64bit以上のCIDを指定してクライアントに使用要求
  - クライアントがその CIDを使用 → OK!

Address Validation

- クライアントはサーバの Initial(Hello)を受信すると Handshakeパケットの暗号鍵を作れる
  - クライアントの Handshakeパケットを受信 → OK!
- 64bit以上のCIDを指定してクライアントに使用要求
  - クライアントがその CIDを使用 → OK!
- Retryパケットで token送信
  - クライアントがその tokenをInitialパケットに同梱 → OK!

Address Validation

- クライアントはサーバの Initial(Hello)を受信すると Handshakeパケットの暗号鍵を作れる
  - クライアントの Handshakeパケットを受信 → OK!
- 64bit以上のCIDを指定してクライアントに使用要求
  - クライアントがその CIDを使用 → OK!
- Retryパケットで token送信
  - クライアントがその tokenをInitialパケットに同梱 → OK!
- 次回接続用に tokenを送信
  - 同一アドレスから tokenをInitialパケットに同梱 → OK!

Transport Parameters (1)

- 1RTで接続確立完了するには、TLSハンドシェイクと並行してフロー制御やフィーチャーのネゴを行う必要がある

Transport Parameters (1)

- 1RTで接続確立完了するには、TLSハンドシェイクと並行してフロー制御やフィーチャーのネゴを行う必要がある
- TLS extension: quic_transport_parameters
  - クライアントは送信する TLS Helloに突っ込む
  - サーバは送信する TLS EncryptedExtensionsに

Transport Parameters (2)

- {Parameter ID, 値} の配列
- 例. フロー制御関連 :
  - initial_max_data - 接続全体のフロー制御
  - initial_max_streams_bidi - 最大双方向ストリーム数
  - initial_max_streams_uni - 最大単方向ストリーム数
  - initial_max_stream_data_(bidi|uni)_(local|remote) - ストリーム単位のフロー制御 （双方向or単方向、どちらが開くストリームか ）
- いずれも接続直後の初期値

QUIC Streams

- QUICはマルチストリーム対応
- 4種類のストリーム（ストリームIDの下位2bitで判別）

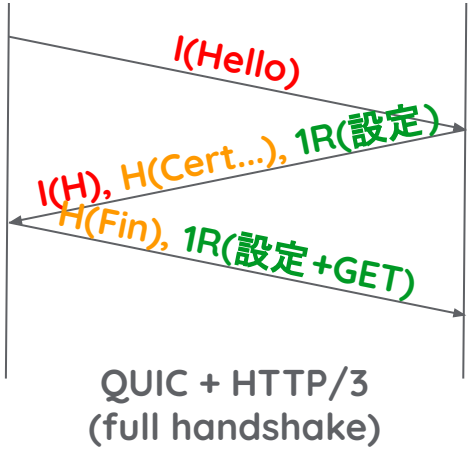
Bits	Stream Type
0x00	Client-Initiated, Bidirectional
0x01	Server-Initiated, Bidirectional
0x02	Client-Initiated, Unidirectional
0x03	Server-Initiated, Unidirectional

Table 1: Stream ID Types

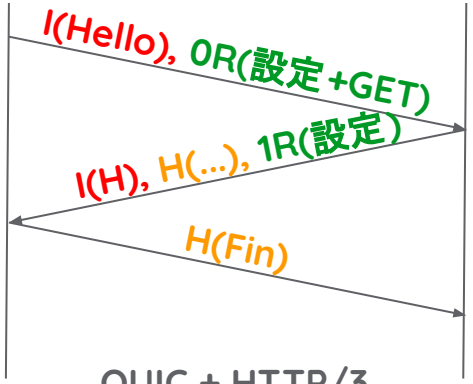
ストリーム 4種類の理由

- 接続確立後、即 HTTPの設定関連情報を投げたい
  - フルハンドシェイクならサーバが先、0-RTTならクライアントが先
  - 開く側が場合によって変わる以上、双方向ストリームで設定交換は辛い

→ どちらからでも開ける単方向必要



QUIC + HTTP/3
(full handshake)



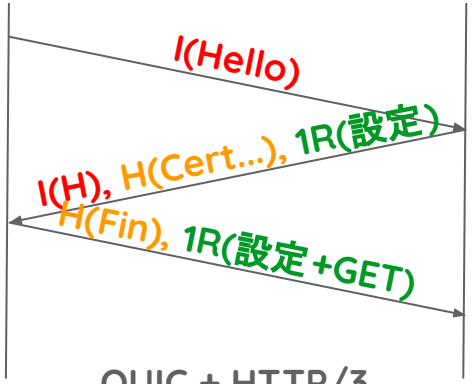
QUIC + HTTP/3
(0-RTT handshake)

ストリーム 4種類の理由

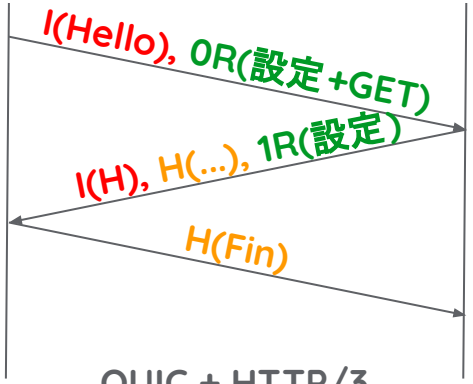
- 接続確立後、即 HTTPの設定関連情報を投げたい
  - フルハンドシェイクならサーバが先、0-RTTならクライアントが先
  - 開く側が場合によって変わる以上、双方向ストリームで設定交換は辛い

  → どちらからでも開ける単方向必要
- HTTPはリクエストレスポンス

  → 双方向あると楽



QUIC + HTTP/3
(full handshake)



QUIC + HTTP/3
(0-RTT handshake)

ストリームの閉じ方

- 単方向・双方向の違いは、開始時に双方向に開くか

ストリームの閉じ方

- 単方向・双方向の違いは、開始時に双方向に開くか
- 閉じるのは常に一方向ずつ
  - FIN - 正常終了
  - RESET_STREAM - リセット

ストリームの閉じ方

- 単方向・双方向の違いは、開始時に双方向に開くか
- 閉じるのは常に一方向ずつ
  - FIN - 正常終了
  - RESET_STREAM - リセット
- 相手に閉じるよう要求
  - STOP_SENDING

ストリームの閉じ方

- 単方向・双方向の違いは、開始時に双方向に開くか
- 閉じるのは常に一方向ずつ
  - FIN - 正常終了
  - RESET_STREAM - リセット
- 相手に閉じるよう要求
  - STOP_SENDING
- RESET_STREAM / STOP_SENDINGはアプリケーションプロトコルが定義したエラーコード添付
  - 例. H3_REQUEST_CANCELLED

フロー制御

- ストリーム本数を例に
- 初期値:initial_max_streams Transport Parameter
  - 「n本目まで開いていいよ」

フロー制御

- ストリーム本数を例に
- 初期値:initial_max_streams Transport Parameter
  - 「n本目まで開いていいよ」
- 開かれる側が MAX_STREAMSフレームを送信して *n*を更新
  - 受信側が能動的に更新しないと送信側は新たなストリームを開けない

フロー制御

- ストリーム本数を例に
- 初期値:initial_max_streams Transport Parameter
  - 「n本目まで開いていいよ」
- 開かれる側が MAX_STREAMSフレームを送信して *n*を更新
  - 受信側が能動的に更新しないと送信側は新たなストリームを開けない
  - HTTP/2: 送信者は古いのを閉じたら新しいのを開ける
    - Reset Flood攻撃:
      while (1) { 開いてリクエスト送ってリセット　}

フロー制御

- 全て同じ、絶対値を使うやりかた
  - ストリーム本数
    - 「n本目まで開いていいよ」
  - ストリーム毎の受信データ量制御
    - 「nバイト目まで送っていいよ」
  - 接続全体の受信データ量制御
    - 「nバイト目まで送っていいよ」

接続切断

- CONNECTON_CLOSEを送信
- アイドルタイムアウト
  - サバクラ両者で max_idle_timeout Transport Parameterを交換
  - 両者の min(max_idle_timeout) 秒間アイドル
    → 何も送受信せずに切断
  - 無線を起こさなくていいので省電力

さまざまな拡張が進行中

- Multipath
  - 複数IP:portで同時送受信
- delayed-ack
  - ACKの個数を減らして効率向上
- receive-ts
  - 接続開始からの経過時間共有　- 方向ごとの遅延測定
- reset-stream-at
  - 指定範囲はデータ転送しつつストリームリセット

QUICの再送制御

ACKフレーム

```
ACK Frame {
  Type (i) = 0x02..0x03,
  Largest Acknowledged (i),
  Ack Delay (i),
  Ack Range Count (i),
  First ACK Range (i),
  ACK Range (..), ...,
  [ECN Counts (..)],
}
```

暗号化ペイロード内に配置
（中継機器からは見えない）

ECNカウントを含む (3)か否(2)か

受信した最大のパケット番号

Largest受信してから ACK送信までの
待機時間
（より正確な RTT推定が可能に）

受信したパケット番号の範囲を多数カ
バーして、パケロスした部分を高精度
に通知（実装によるが 64個とか ...）

RTT推定

- TCPのTimestampオプション的なのはない
- RACK同様、パケット毎に送信時間記憶
- ACKフレームを利用
  - RTT = ACK受信時刻
    - ACK.largest_acknowledged の送信時刻
    - ACK.ack_delay
- 高精度なRTT推定
  - 再送でもパケット番号が変わるので曖昧性ゼロ
  - ACK.ack_delayフィールドの存在

ロス検知と再送タイミング

TCPの両形式のいいとこどり

	TCP		QUIC
ロス検知	従来形	RACK-TLP	
末尾以外	DupACK 3個	sRTT + ¼minRTT	欠落後3パケット受信 または sRTT * 9/8
末尾	RTO ≧ 0.2秒[注1] 指数的退行	max(2*sRTT, 1.5*sRTT + 40ms)[注2] 2回目以降: RTO利用	sRTT + 4*RTTvar + 25ms[注3] 2回目以降: 指数的退行

TCPより早期に
再送(試し撃ち)

注1: 0.2秒はlinuxの場合。RFC 6298は1秒以上推奨
注2: linuxの場合の式。RFC 8985は2*sRTTにack-delayを足してもよいと規定
注3: 25msは標準の値。max_ack_delay Transport Parameterにより調整・通知可能

輻輳制御

- QUICのRFC (9002) はTCPのNewRenoをコピー
- 実際に何を使うかは実装による
  - 輻輳制御はプロトコル関係なく置換しやすい
    - 入力が輻輳と遅延、出力が CWNDな関数なので…

QUICの硬直化抑止策

greasing

- 暗号化で中継装置の影響は排除
- 通信相手が未知の拡張で誤動作する可能性

greasing

- あらかじめ Transport Parameter IDの一部を予約
  - ID = 31 * 整数 + 27
  - これらは決して意味をもたない
- 送信側がランダムに使用
- 受信側の「未知の IDを無視する」コードが実行されるように(グリース＝潤滑剤を投入) する
- 他のプロトコルでもグリースの導入が進む
  - RFC 8701 - TLS拡張について、グリース ID予約

greasing

- 暗号化で中継装置の影響は排除
- 通信相手が未知の拡張で誤動作する可能性

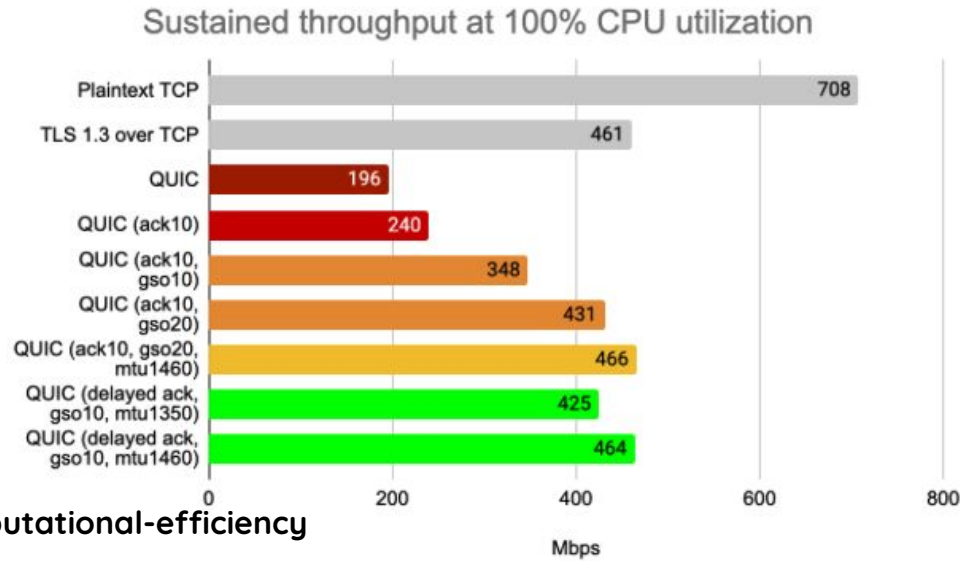
QUIC version 2

- RFC 9369
- 中継装置に見える部分について、 v1と違う定数を使用
  - version = 2
  - long header packet type: I=1, 0R=2, H=3, R=0
  - Initialパケット暗号化のソルト変更
  - 鍵生成につかうHMACのラベル変更

QUICならではの最適化

QUICって重たいんでしょ

- 一般論としては正しい
  - 小サイズ（パケット毎）の暗号化
  - カーネルではなくユーザ空間でのパケット処理
- やり方次第では結構軽い



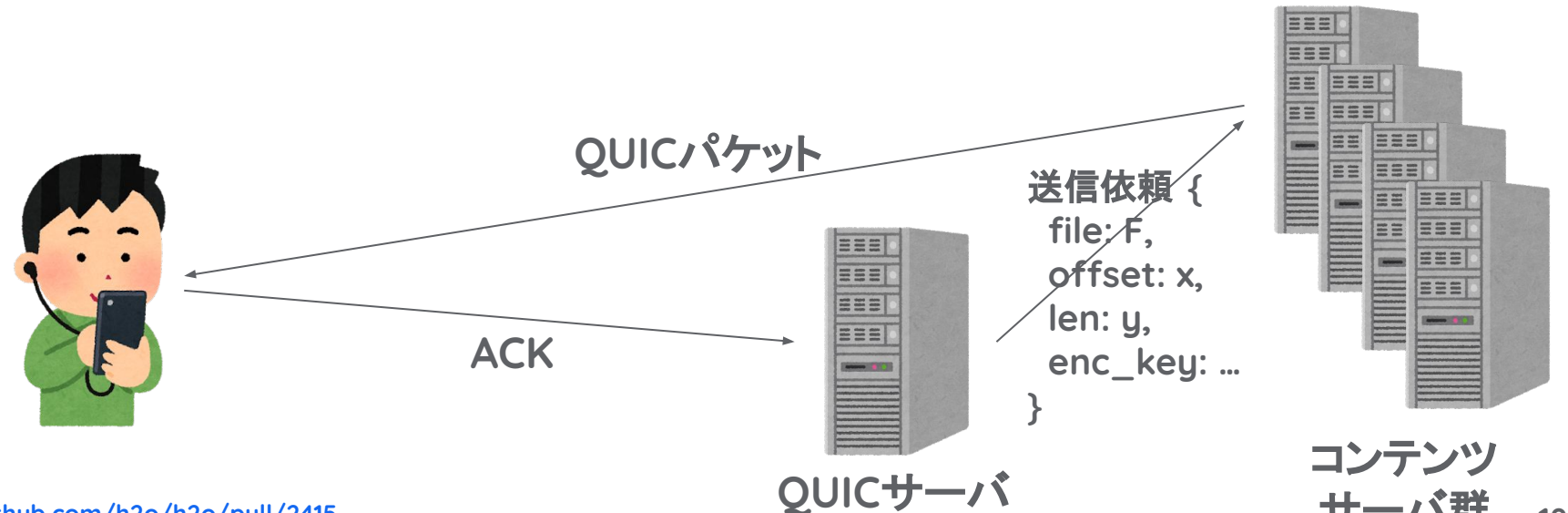
https://www.fastly.com/blog/measuring-quic-vs-tcp-computational-efficiency

送信バッファレスなコンテンツ配信

- TCPは接続毎に送信バッファをもつ
  - cleartext TCPならsendfile(2)すればゼロコピー
    - 送信バッファはファイルへの「参照」もつ
  - TLSだと送信バッファ書込み時に暗号化≒コピー
    - 送信バッファの中身を ACKが返るまで保持
- QUICはパケット生成時に暗号化
  - 送信バッファが不要
  - ファイルを読んでパケット生成 →DMAでNICに渡す
    - CPUのキャッシュ内で完結可能

Direct Server Return (DSR)

- 接続ステートは QUICサーバで管理
- データパケットの「送信」のみをコンテンツサーバが代理



https://github.com/h2o/h2o/pull/2415
https://engineering.fb.com/2022/07/06/networking-traffic/watch-metas-engineers-discuss-quic-and-tcp-innovations-for-our-network/

QUICとHTTPのこれから

QUIC / HTTP3の普及率

- 75% of Meta's internet traffic[注1] (2020年10月)
- 26-28% of origins[注2] (2024年6月)
- 20% of Firefox traffic[注3] (2025年9月)

注1: https://engineering.fb.com/2020/10/21/networking-traffic/how-facebook-is-bringing-quic-to-billions/
注2: https://almanac.httparchive.org/en/2024/http#http-version-adoption
注3: https://max-inden.de/post/fast-udp-io-in-firefox/

更なる普及を目指して

- IETF 124 “More QUIC, How?” 会議 (2025年11月)

更なる普及に向けた障害

- エンタープライズで TCPを使用せざるを得ない
  - 独自CAインストール →TCPプロキシで通信解読
- CDNベンダーがHTTP/3を勧めにくい
  - HTTP/2 vs. 3 の性能比較がサーバサイドできないため
    - HTTP/2しか話せないクライアントの影響
- QUIC実装の品質・運用上の問題
- (NATの話題は出なかった)

QUICの性能向上させる提案

- SCONE
- スロースタート高速化
  - Careful Resume / Jumpstart
  - SUSS
  - Rapid Start

SCONE

SCONE

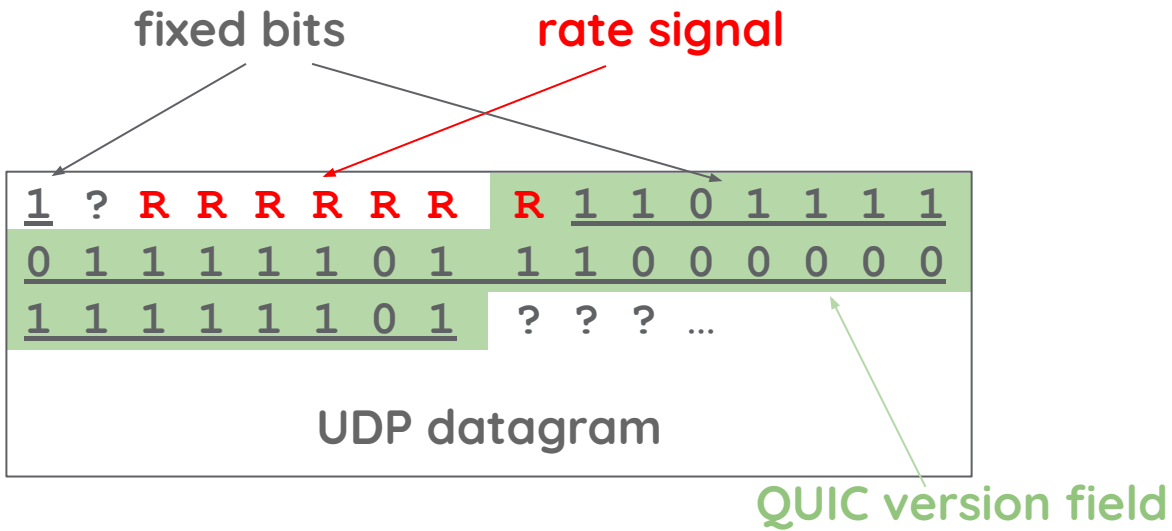
- Standard COmmunication with Network Elements
- エンドポイントとネットワーク機器が強調して、エンドポイントの帯域利用を最適化するためのプロトコル
- 2024年末に標準化開始

SCONEの背景

- 動画配信がネットワーク帯域の過半を占めるように
- ネットワーク側 :
  - IPアドレス/SNIで動画フロー検出 →スロットリング
  - 誤判定多い (false negativeに加えfalse positiveも)
  - スロットリングは無線の電波利用効率が低い
- エンドポイント側 :
  - 利用可能バンド幅を probe (小さい帯域からはじめて、どこまで大きな帯域を使えるか、徐々に拡大)
  - 問題: 動画見始めが汚い / 帯域探索時のパケロス考慮してバッファ確保が必要(遅延つらい)

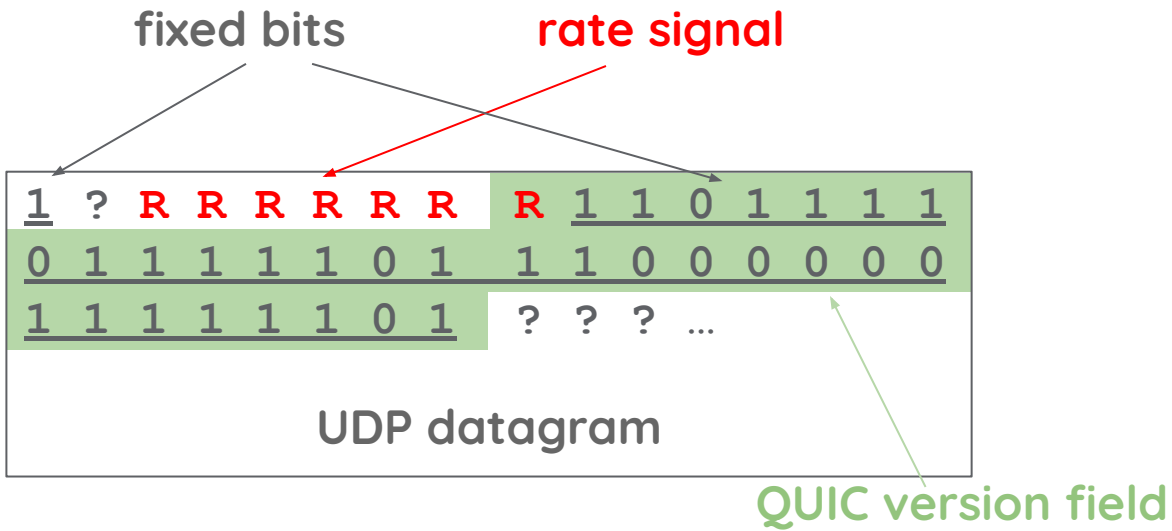
SCONEパケット

- SCONEパケット=特殊なバージョンの QUICパケット



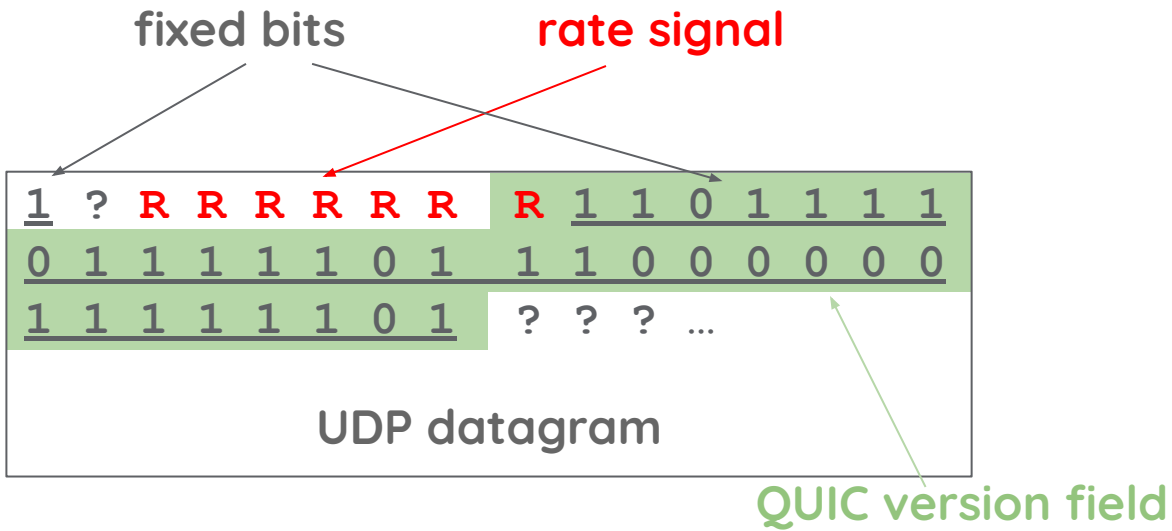
SCONEパケット

- SCONEパケット=特殊なバージョンの QUICパケット
- サーバが定期的に送信



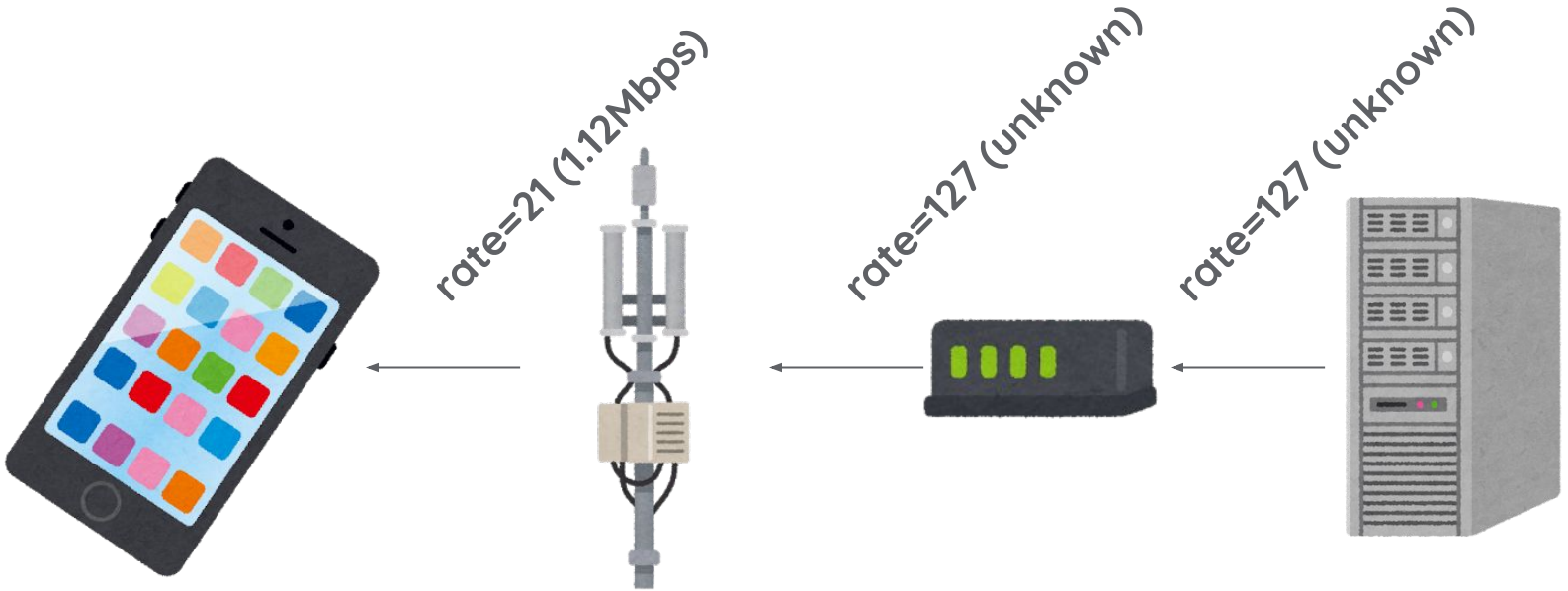
SCONEパケット

- SCONEパケット=特殊なバージョンの QUICパケット
- サーバが定期的に送信
- ネットワーク機器が rate signalを書き換え



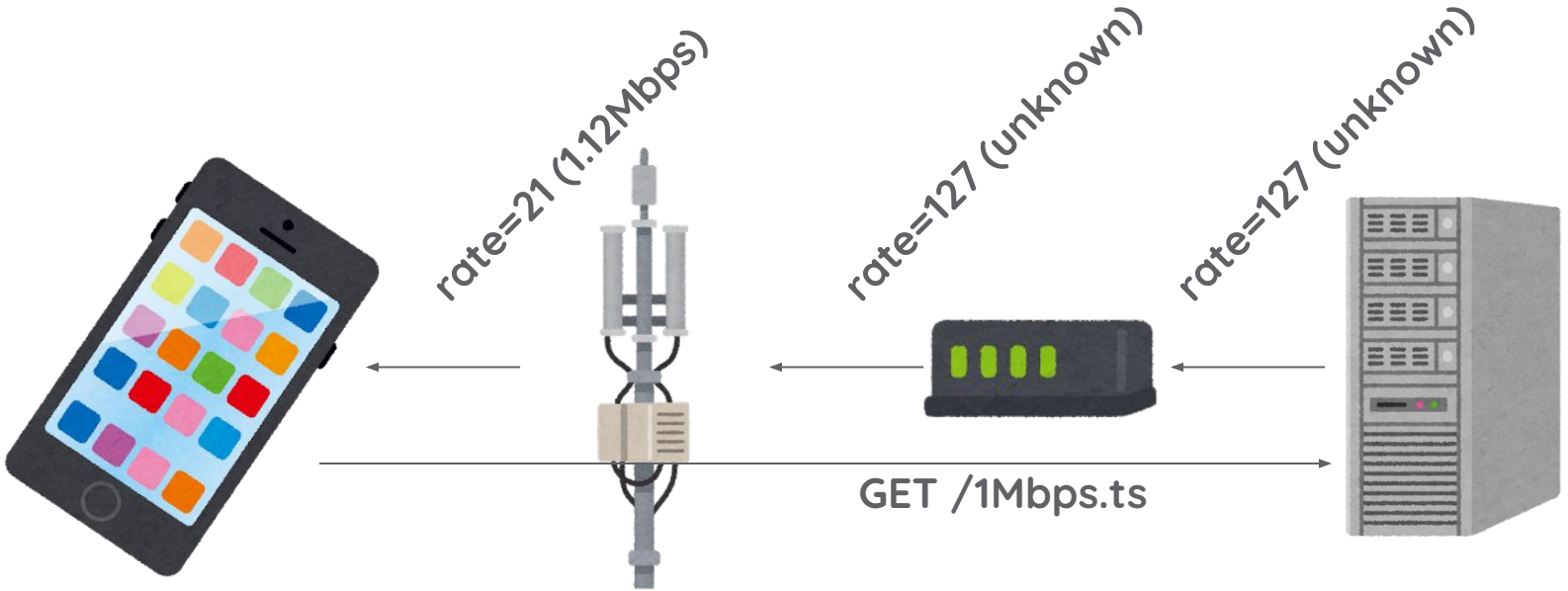
SCONE - 帯域通知

- 通知帯域を小さくしたい場合に rate signal書き換え



SCONE - 帯域通知

- 通知帯域を小さくしたい場合に rate signal書き換え
- クライアントは通知帯域以下の動画ファイルを要求



SCONE - rate signal

- 0-127 (7bit)で表現 / +20で10倍

rate signal	帯域上限
0	100 Kbps
1	112 Kbps
2	126 Kbps
…	
19	891 Kbps
20	1 Mbps
…	
39	8.91 Mbps

…

rate signal	帯域上限
100	10 Gbps
101	11.2 Gbps
…	
119	89.1 Gbps
120	100 Gbps
…	
126	199 Gbps
127	制限なし

SCONE - 判定とスロットリング

- SCONE対応QUICは最初のパケットの末尾 2バイトが C8 13
  - IPアドレスやSNIの解析が不要に
- ただし、SCONE対応≠自主的な帯域制限実施
  - 自主性のみに期待すると abuseされる
  - **約1分以上平均して** 帯域超過したらスロットリング
    - 短時間のバーストは可
    - 動画配信はセグメントごとにバースト転送 &アイドルの繰り返しに → 電波利用効率の向上

SCONE - 判定とスロットリング

- Webブラウザは全ての接続を「 SCONE対応」に
  - 動画フローかどうかはネットワークには漏洩しない
  - Webブラウズのトラヒックは瞬発的　→ スロットリングされない
- 長時間持続する通信は自主規制なければスロットリング
  - 動画かどうかは関係ない
    - cf. ソフトウェアアップデートの配信

SCONE - IETF 124

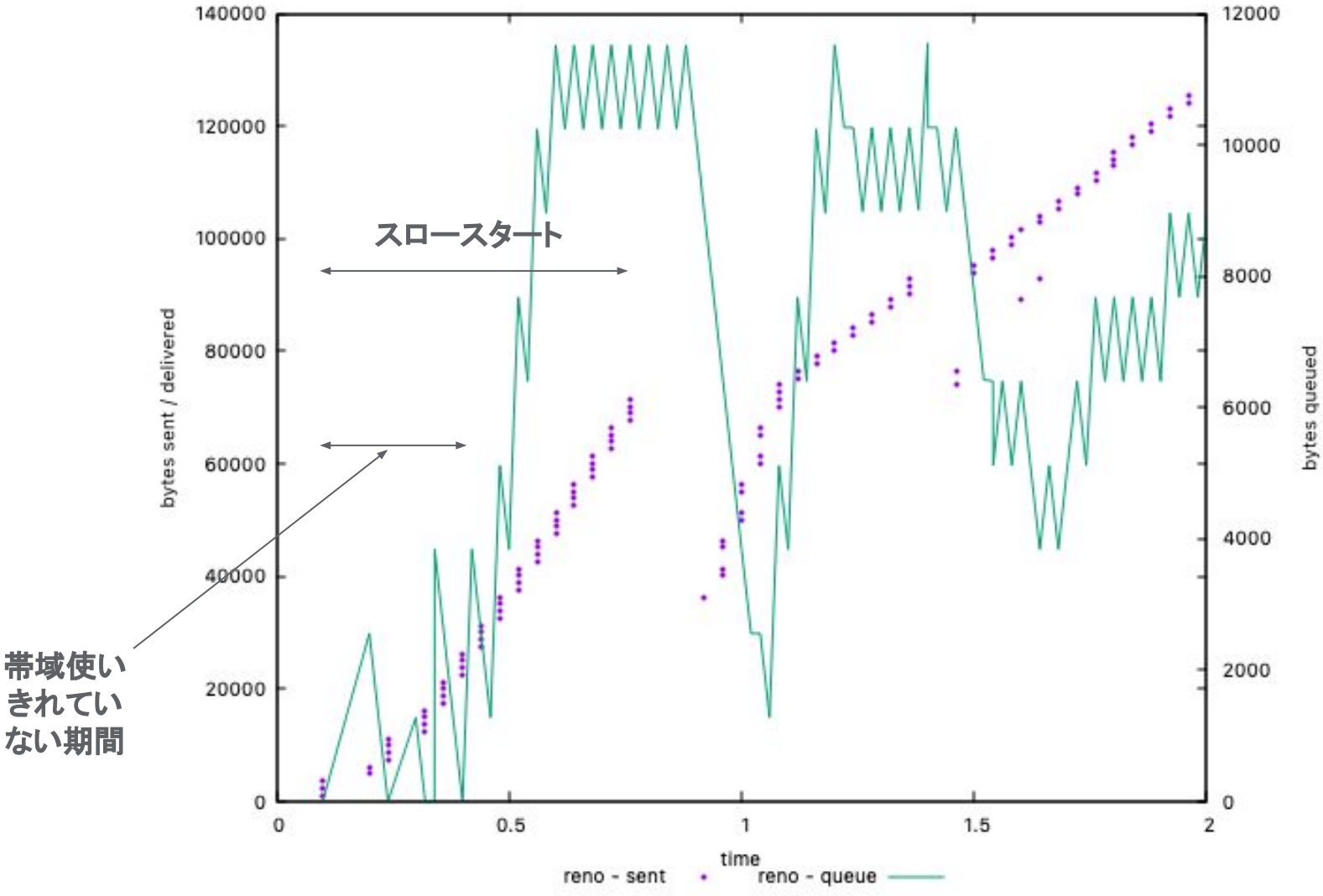
- IETF 124 (2025年11月)で接続試験やデモ
  - Ericsson - SCONE対応のポリサー , 試験アプリ
  - Meta - FacebookアプリにSCONE対応実装
- 1Mbps程度の低レートでも動画が固まらなくなることを確認

https://www.youtube.com/watch?v=XOSRb5VVkjc#t=10m50s

スロースタート高速化

スロースタート高速化

- HTTP接続の過半はスロースタートで終了
  - 長時間持続する通信ではないため


スロースタート
帯域使いきれていない期間
bytes sent / delivered
bytes queued
time
reno - sent
reno - queue
0
20000
40000
60000
80000
100000
120000
140000
2000
4000
6000
8000
10000
12000
0.5
1
1.5
2

Careful Resume

- スロースタートより優れたアルゴリズム
  - IETF TSVWGで策定中

Careful Resume

- スロースタートより優れたアルゴリズム
  - IETF TSVWGで策定中
- 前回接続時の帯域幅を覚えておいて

Careful Resume

- スロースタートより優れたアルゴリズム
  - IETF TSVWGで策定中
- 前回接続時の帯域幅を覚えておいて
- 再接続時は、いきなりその速度で投げてみる

Careful Resume

- スロースタートより優れたアルゴリズム
  - IETF TSVWGで策定中
- 前回接続時の帯域幅を覚えておいて
- 再接続時は、いきなりその速度で投げてみる
- パケロスしたら、予測過大

  → パケロスした分だけ送信速度を下げて
  輻輳回避フェーズに移行

Careful Resume

- スロースタートより優れたアルゴリズム
  - IETF TSVWGで策定中
- 前回接続時の帯域幅を覚えておいて
- 再接続時は、いきなりその速度で投げてみる
- パケロスしたら、予測過大

  → パケロスした分だけ送信速度を下げて
  輻輳回避フェーズに移行
- パケロスしなかったら

  → スロースタート継続

Careful Resume

- スロースタートより優れたアルゴリズム
  - IETF TSVWGで策定中
- 前回接続時の帯域幅を覚えておいて
- 再接続時は、いきなりその速度で投げてみる
- パケロスしたら、予測過大
  → パケロスした分だけ送信速度を下げて
  輻輳回避フェーズに移行
- パケロスしなかったら
  → スロースタート継続

予測

Careful Resume

- スロースタートより優れたアルゴリズム
  - IETF TSVWGで策定中
- 前回接続時の帯域幅を覚えておいて
- 再接続時は、いきなりその速度で投げてみる
- パケロスしたら、予測過大
  → パケロスした分だけ送信速度を下げて
  輻輳回避フェーズに移行
- パケロスしなかったら
  → スロースタート継続

予測

予測が過大でも悪影響少ない初回送信パターン

Careful Resume

- スロースタートより優れたアルゴリズム
  - IETF TSVWGで策定中
- 前回接続時の帯域幅を覚えておいて
- 再接続時は、いきなりその速度で投げてみる
- パケロスしたら、予測過大

  → パケロスした分だけ送信速度を下げて
  輻輳回避フェーズに移行
- パケロスしなかったら

  → スロースタート継続

予測が外れても悪影響少ないなら、予測不要では？

予測

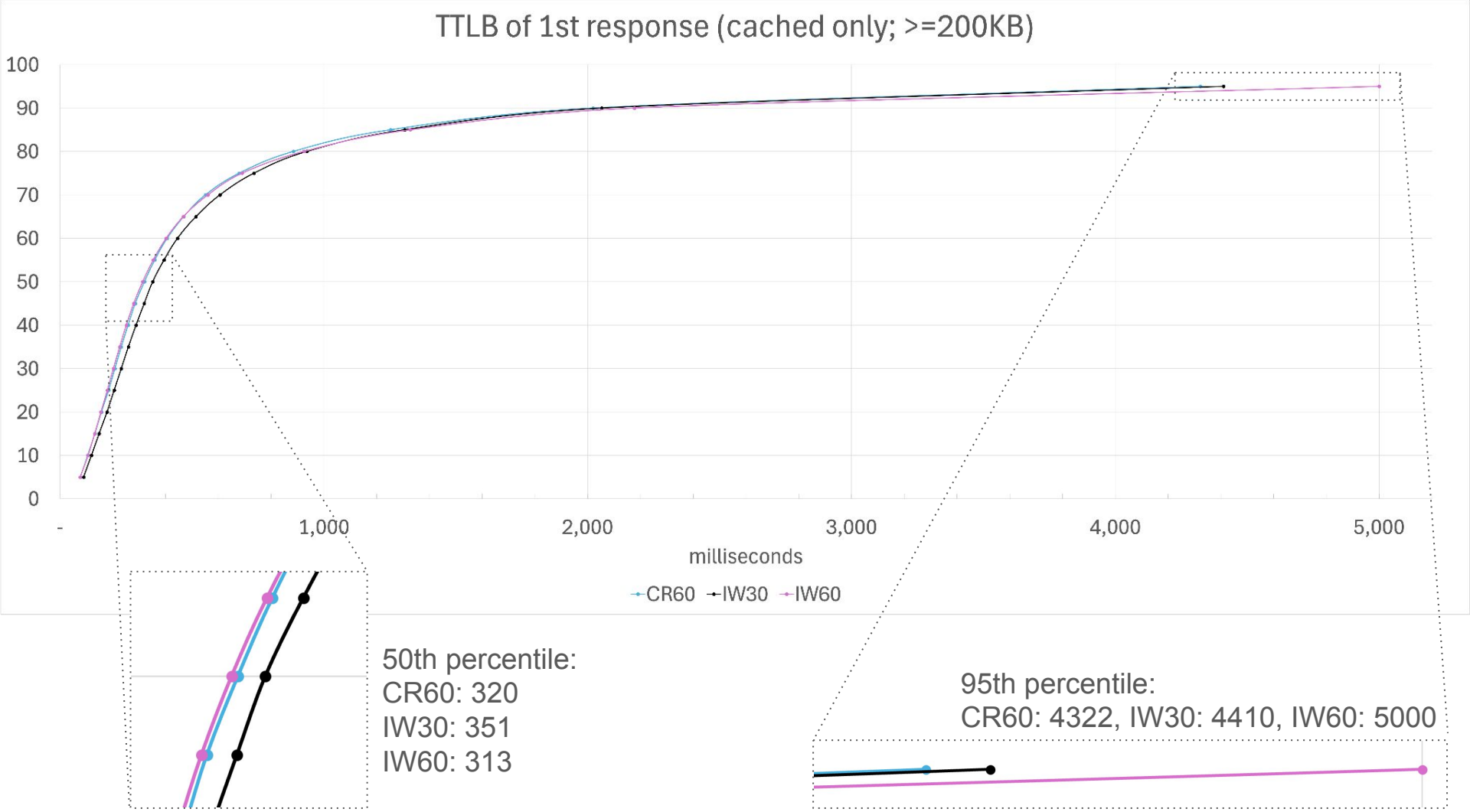
予測が過大でも悪影響少ない初回送信パターン

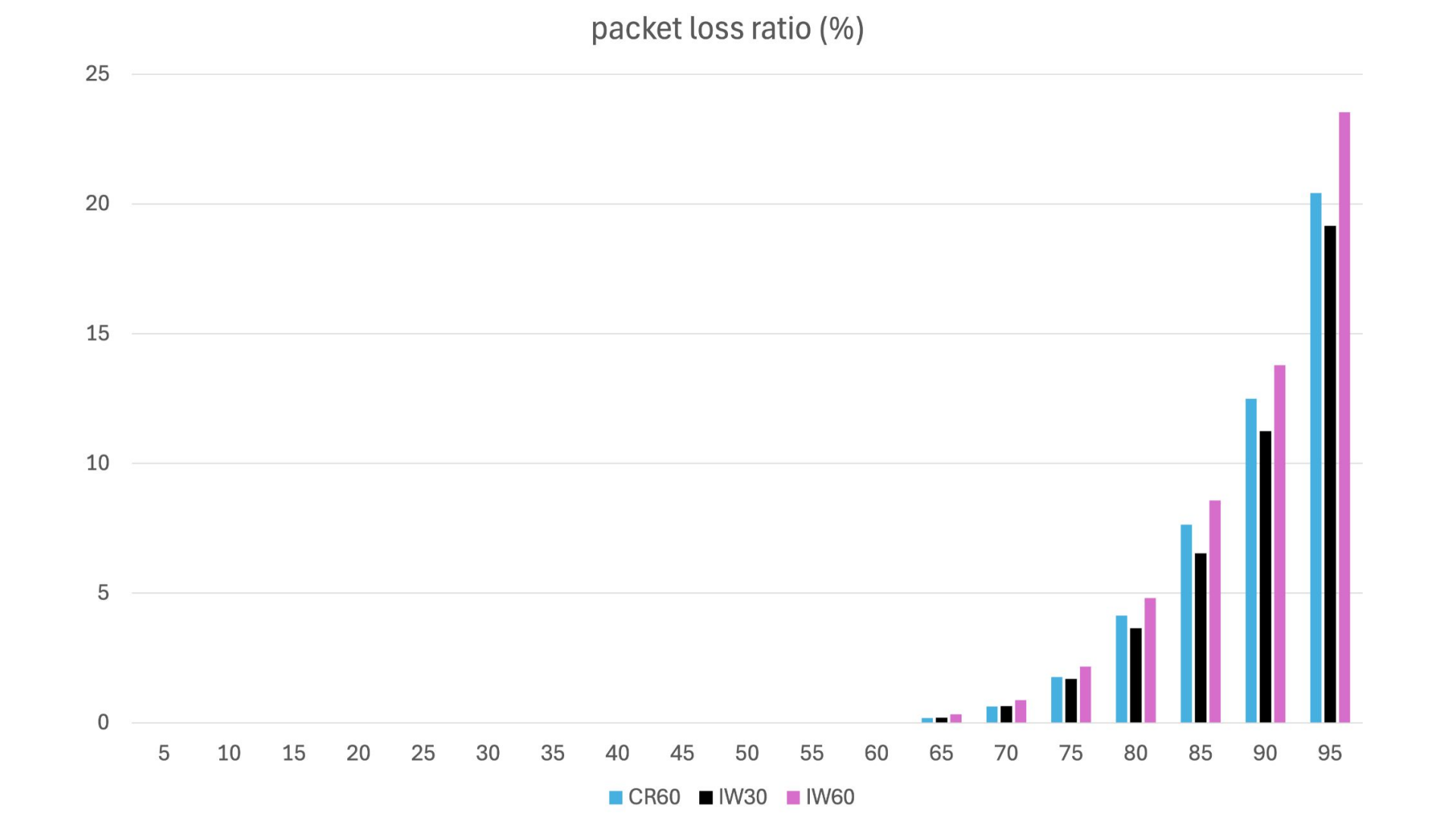
Jumpstart

- Jumpstart = Careful Resume - 帯域幅予測

Jumpstart

- Jumpstart = Careful Resume - 帯域幅予測
- FastlyのサーバでA/Bテスト
  - ヨーロッパのとある POPのうち、サーバ 1台
  - HTTP/3接続を3つの群にわけてテスト
    - CR60 - Careful Resumeウィンドウが 60パケット
    - IW30 - スロースタート初期ウィンドウが 30パケット
    - IW60 - スロースタート初期ウィンドウが 60パケット
  - 1stレスポンスが 200KB以上のcache-hitの接続を抽出
  - TTLBを比較


TTLB of 1st response (cached only; >=200KB)
100
90
80
70
60
50
40
30
20
10
0
-
1,000
2,000
3,000
4,000
5,000
milliseconds
CR60
IW30
IW60
50th percentile:
CR60: 320
IW30: 351
IW60: 313
95th percentile:
CR60: 4322, IW30: 4410, IW60: 5000


packet loss ratio (%)
25
20
15
10
5
0
5
10
15
20
25
30
35
40
45
50
55
60
65
70
75
80
85
90
95
CR60
IW30
IW60

SUSS / rapid start

- スロースタートは RTごとに送信量 2倍
  - さらに倍率を上げることはできる
- SUSS (2024)
  - パケットを一気に投げて、一度に ACKが返ればバンド幅に余裕がありそう → 4倍にする
- rapid start (2025)
  - Jumpstartの上に実装
  - RTTが大きくなり始めるまで 3倍
- どちらも20%程度のTTLB削減を報告

https://dl.acm.org/doi/10.1145/3651890.3672234
https://mailarchive.ietf.org/arch/msg/ccwg/IAIkJn2NA-tpO5P501idmjBYJi0/

まとめ

まとめ

- QUIC=メインストリーム向けトランスポート層プロトコル
  - TCP以来、約40年ぶり
  - 既に20%以上のトラヒックで利用
- 低レイテンシ
- TCPで培った再送・輻輳制御のよいところを継承・高度化
- プライバシーファーストな設計
- Invariants,暗号化とグリースにより将来性を確保
- SCONEやスロースタート高速化など、新たな取組の揺籠に