Rapid Start - インターネット高速化への挑戦、そして Ruby とAIのハッピーな分業

Kazuho Oku, Fastly

自己紹介

- オープンソースHTTPサーバ「H2O」主開発者
  - Fastlyが使用
  - HTTP/1, 2, 3, TLS/1.3, QUIC等を実装
  - mruby (Rack) によるカスタマイズが可能

自己紹介

- オープンソースHTTPサーバ「H2O」主開発者
  - Fastlyが使用
  - HTTP/1, 2, 3, TLS/1.3, QUIC等を実装
  - mruby (Rack) によるカスタマイズが可能
- ホビープログラマとして：
  - rat (IPv4 NATのruby実装)

自己紹介

- オープンソースHTTPサーバ「H2O」主開発者
  - Fastlyが使用
  - HTTP/1, 2, 3, TLS/1.3, QUIC等を実装
  - mruby (Rack) によるカスタマイズが可能
- ホビープログラマとして：
  - rat (IPv4 NATのruby実装)
- RFCの共著者として：
  - RFC 8297 (HTTP 103 Early Hints)
  - RFC 9218 (HTTP Extensible Priorities)
  - RFC 9849 (TLS Encrypted Client Hello)

目次

- Rapid Start
  - H2O/Fastlyで使用中の新しい輻輳制御スタートアップアルゴリズム
- jrf - ruby + AI で開発したログ解析ツール

Rapid Start

接続確立にかかる時間

- TCP+TLS/1.2:
  - full handshake: 3 RT
  - resumption: 2 RT

※RT = ラウンドトリップ (電文の往復)の回数

接続確立にかかる時間

- TCP+TLS/1.2:
  - full handshake: 3 RT
  - resumption: 2 RT
- TCP+TLS/1.3:
  - full handshake: 2 RT
  - resumption: 1 RT

※RT = ラウンドトリップ (電文の往復)の回数

接続確立にかかる時間

- TCP+TLS/1.2:
  - full handshake: 3 RT
  - resumption: 2 RT
- TCP+TLS/1.3:
  - full handshake: 2 RT
  - resumption: 1 RT
- QUIC:
  - full handshake: 1 RT
  - resumption: 0 RT

※RT = ラウンドトリップ (電文の往復)の回数

接続確立にかかる時間

- TCP+TLS/1.2:
  - full handshake: 3 RT
  - resumption: 2 RT
- TCP+TLS/1.3:
  - full handshake: 2 RT
  - resumption: 1 RT
- QUIC:
  - full handshake: 1 RT
  - resumption: 0 RT

HTTP/3

※RT = ラウンドトリップ (電文の往復)の回数

HTTP: 遅延削減の取り組み

- HTTP/3で接続遅延は極小化
  - full handshake: 1 RT
  - resumption: 0 RT

HTTP: 遅延削減の取り組み

- HTTP/3で接続遅延は極小化
  - full handshake: 1 RT
  - resumption: 0 RT
- したがって、HTTPレスポンスの Time To First Byte (TTFB) は:
  - full handshake: 2 RT
  - resumption: 1 RT

HTTP: 遅延削減の取り組み

- HTTP/3で接続遅延は極小化
  - full handshake: 1 RT
  - resumption: 0 RT
- したがって、HTTPレスポンスの Time To First Byte (TTFB) は:
  - full handshake: 2 RT
  - resumption: 1 RT
- では Time To Last Byte (TTLB) は?
  - TTLBは通常、TTFBとスロースタートの合計

スロースタート

- 輻輳制御の第一フェーズ：
  - 接続で利用可能なバンド幅が不明な際に利用
  - 利用可能なバンド幅を迅速に見極めるため

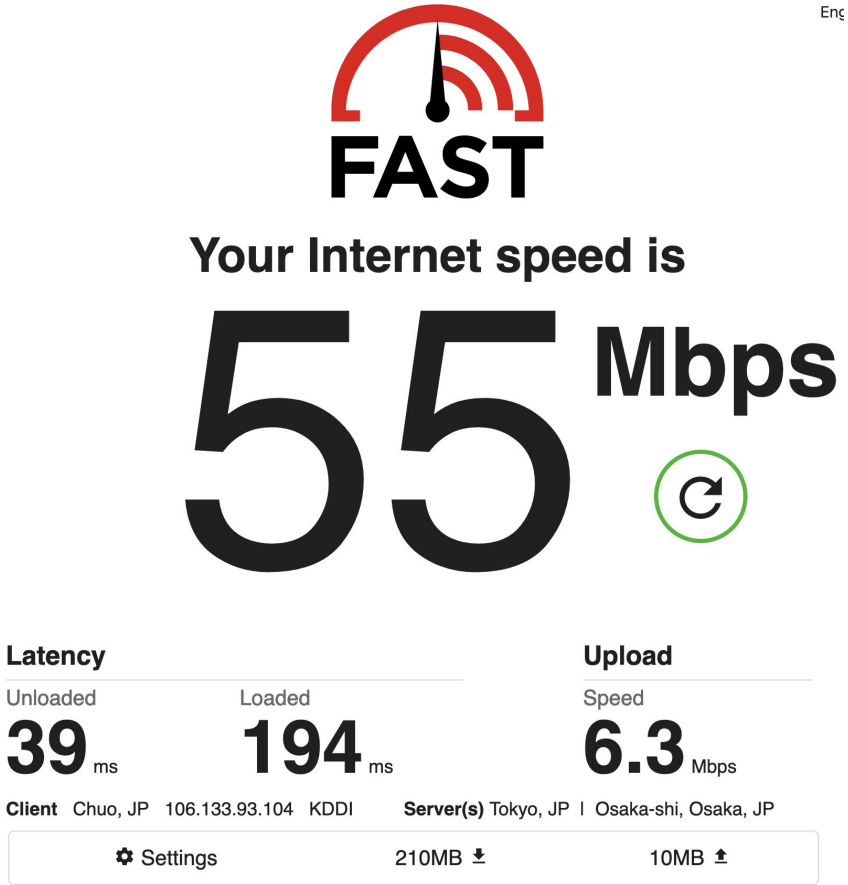
スロースタート

- 輻輳制御の第一フェーズ：
  - 接続で利用可能なバンド幅が不明な際に利用
  - 利用可能なバンド幅を迅速に見極めるため
- まず、IW個のパケットを送出：
  - IW = 10 (RFC), 30 (世の中の実際)
  - ACK受信するたびに 2倍のパケットを追加で送出

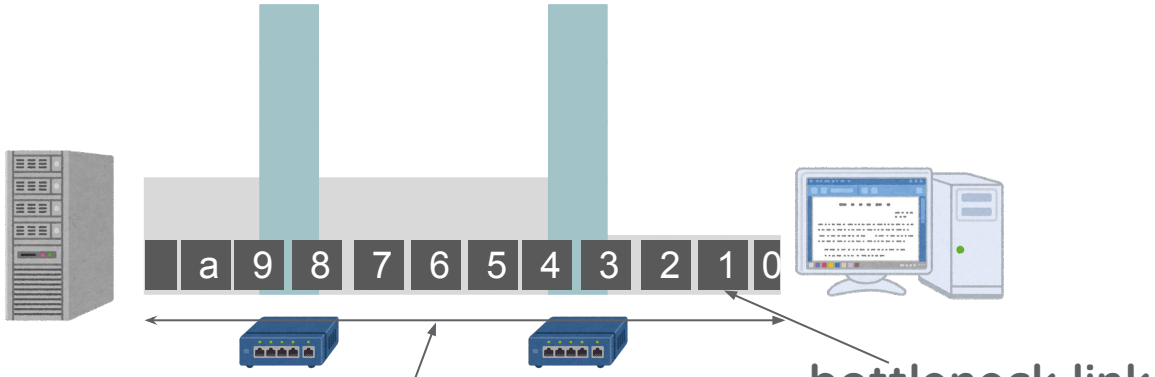
スロースタート

- 輻輳制御の第一フェーズ：
  - 接続で利用可能なバンド幅が不明な際に利用
  - 利用可能なバンド幅を迅速に見極めるため
- まず、IW個のパケットを送出：
  - IW = 10 (RFC), 30 (世の中の実際)
  - ACK受信するたびに2倍のパケットを追加で送出
- パケロスすると(つまりネットワークが溢れると)失ったパケットを再送する「リカバリ」を行い、輻輳制御の第二フェーズである「輻輳回避」に遷移

スロースタートと BDP (5G回線)

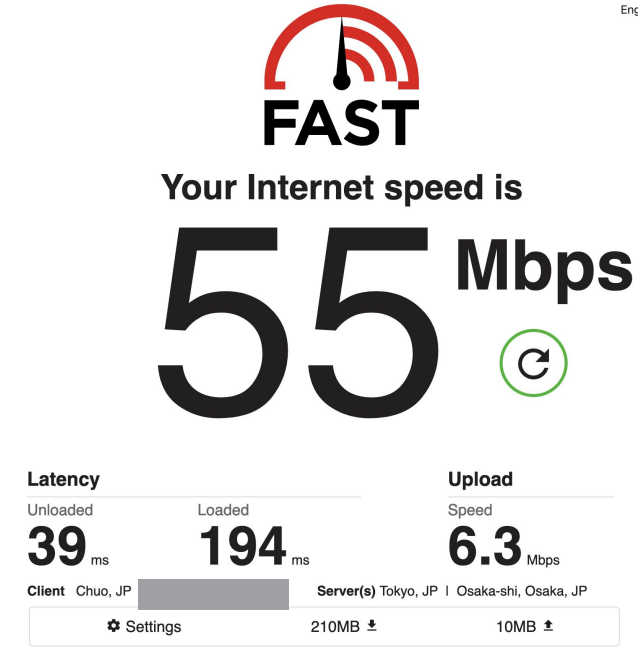


スロースタートと BDP (5G回線)

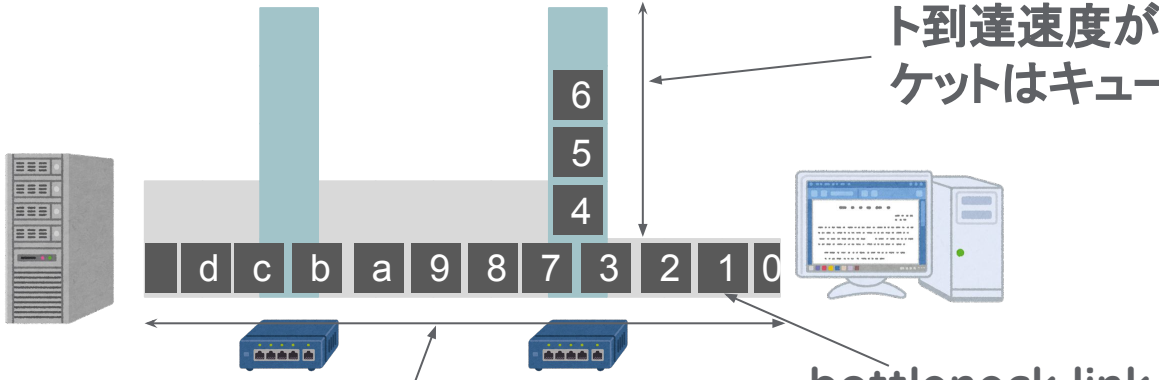


Idle BDP: ボトルネックを使い切るために必要なパケットの数

※BDP (Bandwidth Delay Product): 帯域幅遅延積



スロースタートと BDP (5G回線)

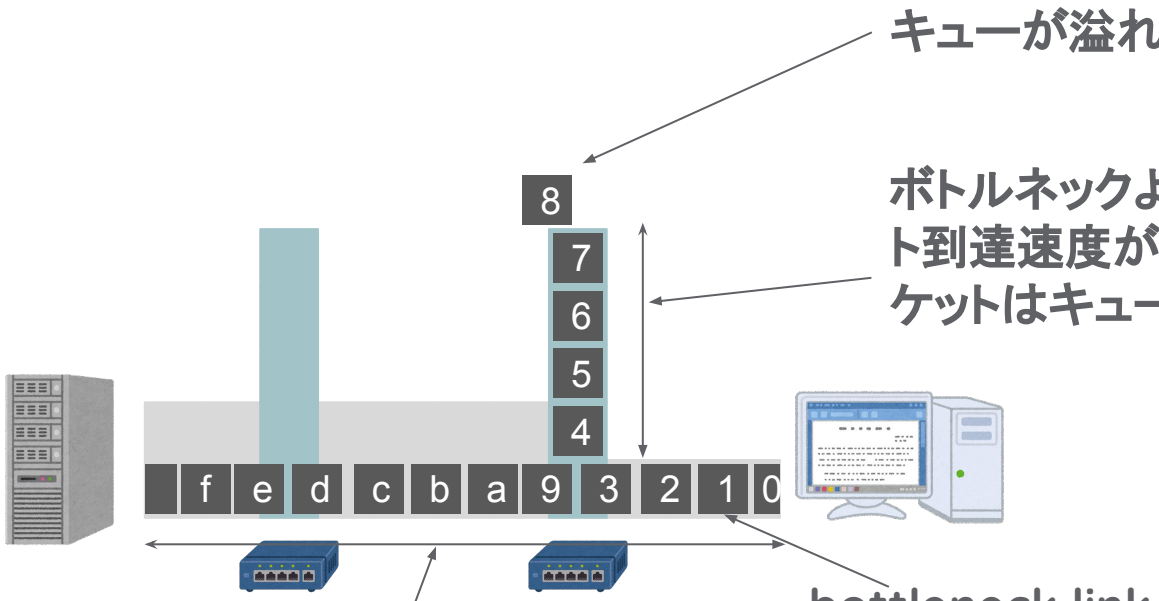


ボトルネックよりもパケット到達速度が速いと、パケットはキューに滞留

Idle BDP: ボトルネックを使い切るために必要なパケットの数

※BDP (Bandwidth Delay Product): 帯域幅遅延積

スロースタートと BDP (5G回線)

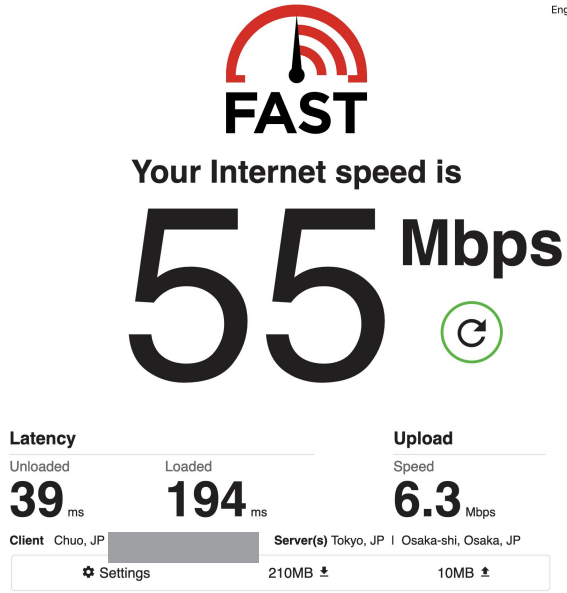


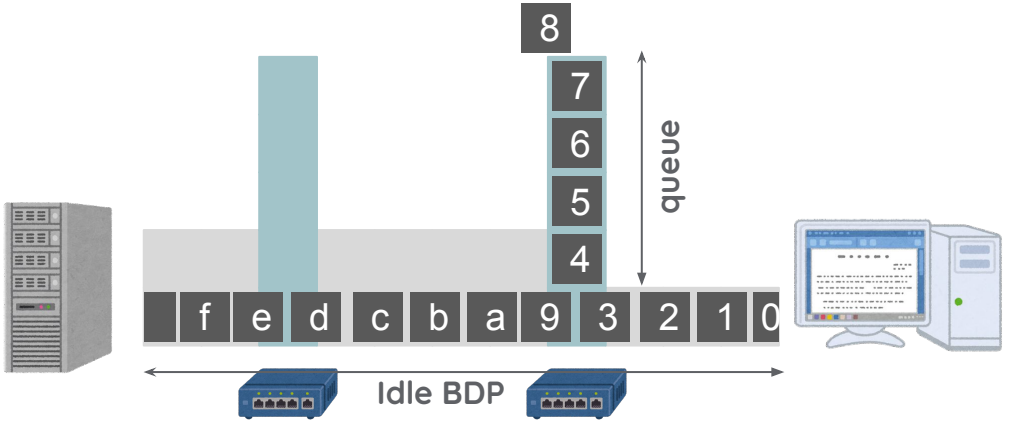
Idle BDP: ボトルネックを使い切るために必要なパケットの数

※BDP (Bandwidth Delay Product): 帯域幅遅延積

スロースタートと BDP (5G)

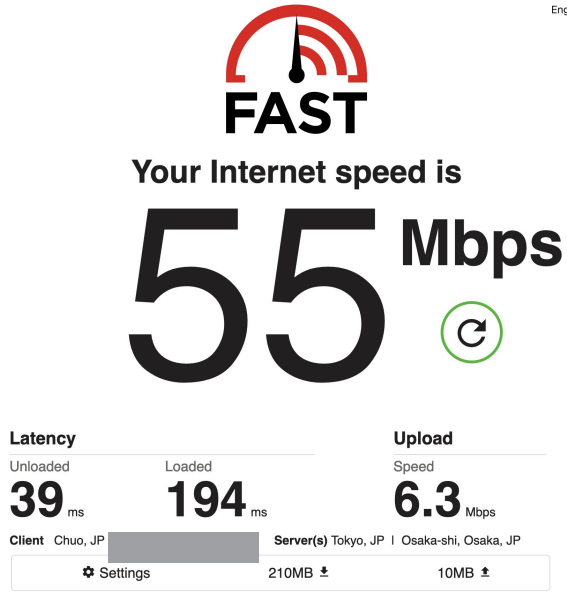
- Idle BDP = 55Mb/s * 0.039s

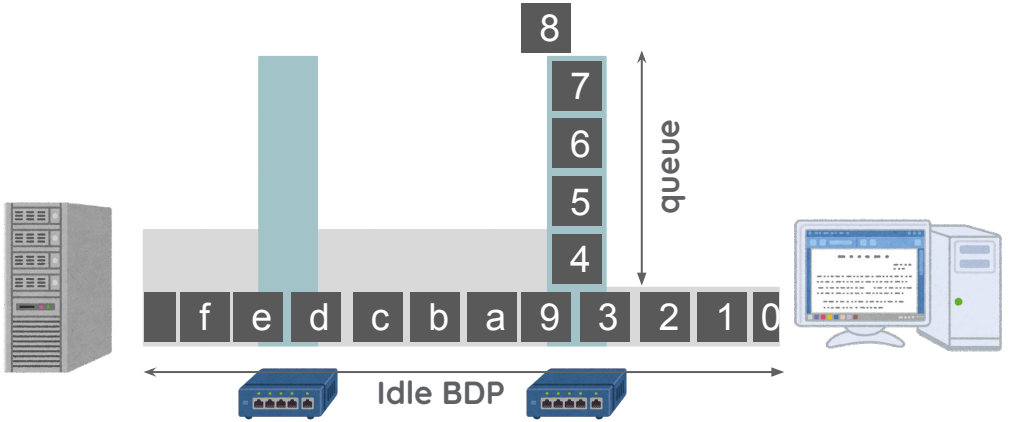




スロースタートと BDP (5G)

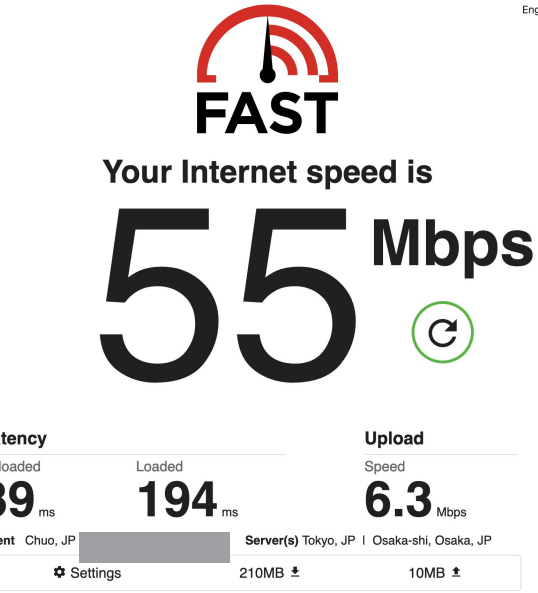
- Idle BDP = 55Mb/s * 0.039s
  - ≒ 2.15Mb ≒ 268KB
  - ≒ 209 packets

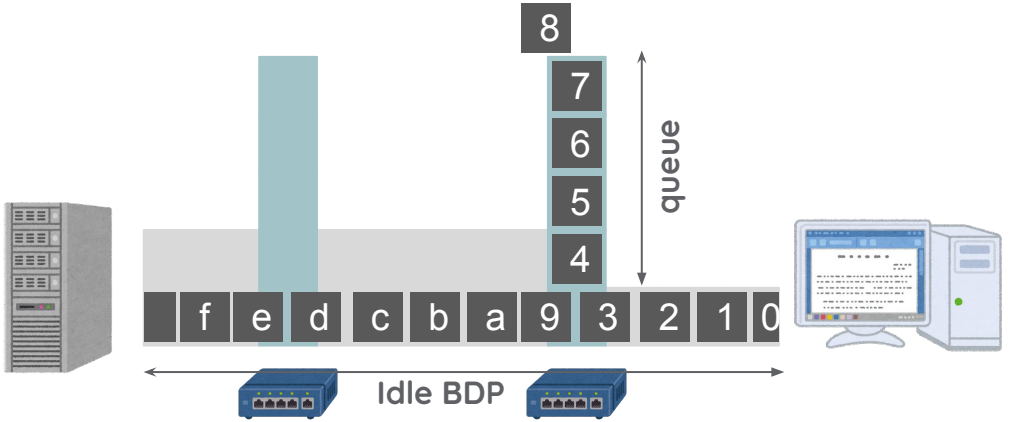




スロースタートと BDP (5G)

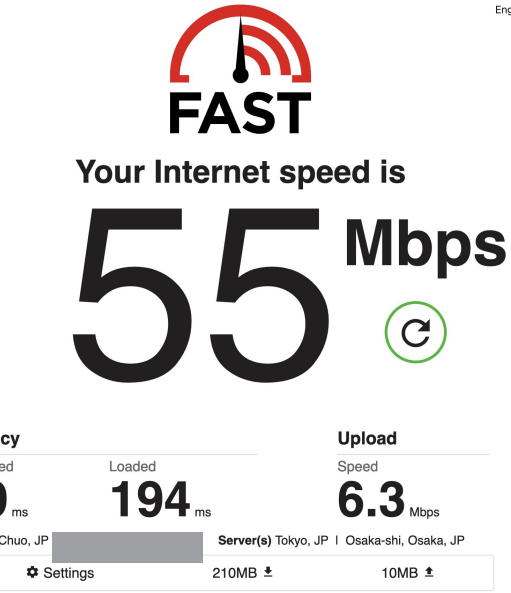
- Idle BDP = 55Mb/s * 0.039s
  - ≒ 2.15Mb ≒ 268KB
  - ≒ 209 packets
- スロースタート :

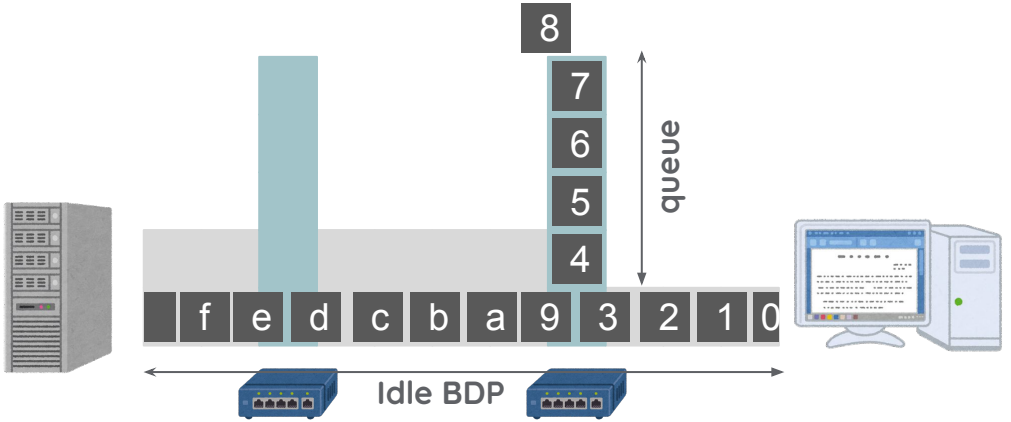




スロースタートと BDP (5G)

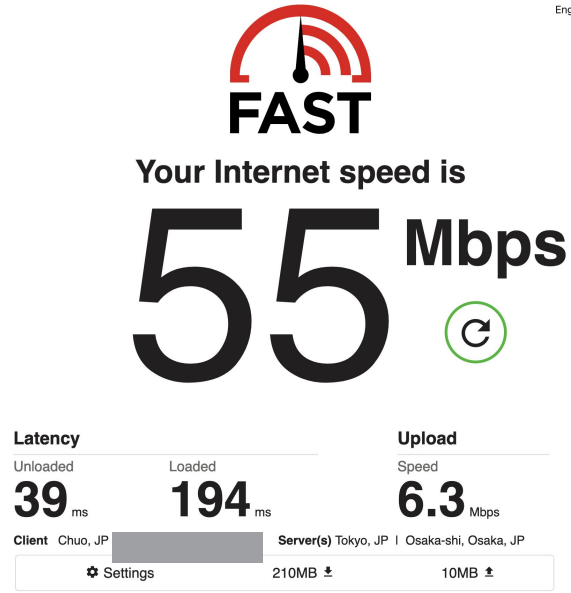
- Idle BDP = 55Mb/s * 0.039s
  - ≒ 2.15Mb ≒ 268KB
  - ≒ 209 packets
- スロースタート :
  - 1RT: 30 packets

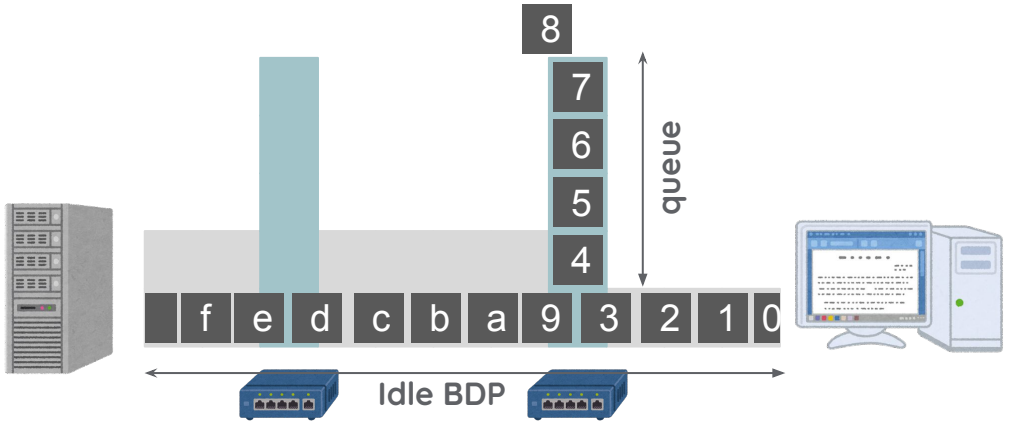




スロースタートと BDP (5G)

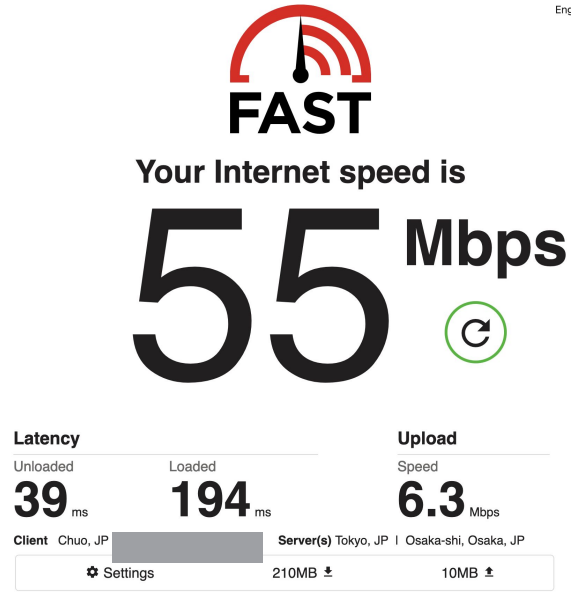
- Idle BDP = 55Mb/s * 0.039s
  - ≒ 2.15Mb ≒ 268KB
  - ≒ 209 packets
- スロースタート :
  - 1RT: 30 packets
  - 2RT: 60 packets

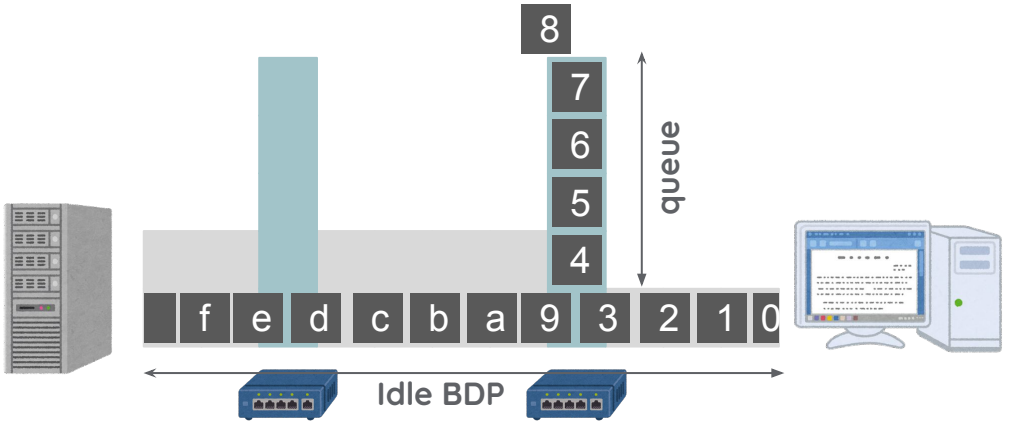




スロースタートと BDP (5G)

- Idle BDP = 55Mb/s * 0.039s
  - ≒ 2.15Mb ≒ 268KB
  - ≒ 209 packets
- スロースタート :
  - 1RT: 30 packets
  - 2RT: 60 packets
  - 3RT: 120 packets

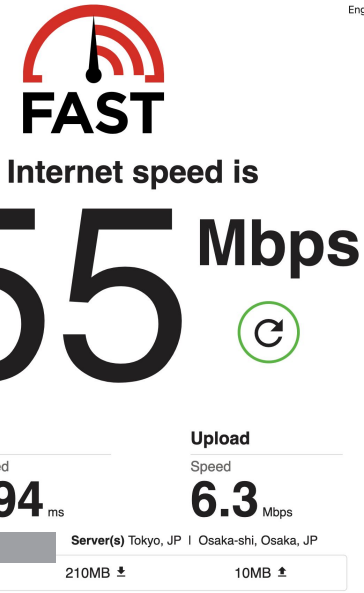


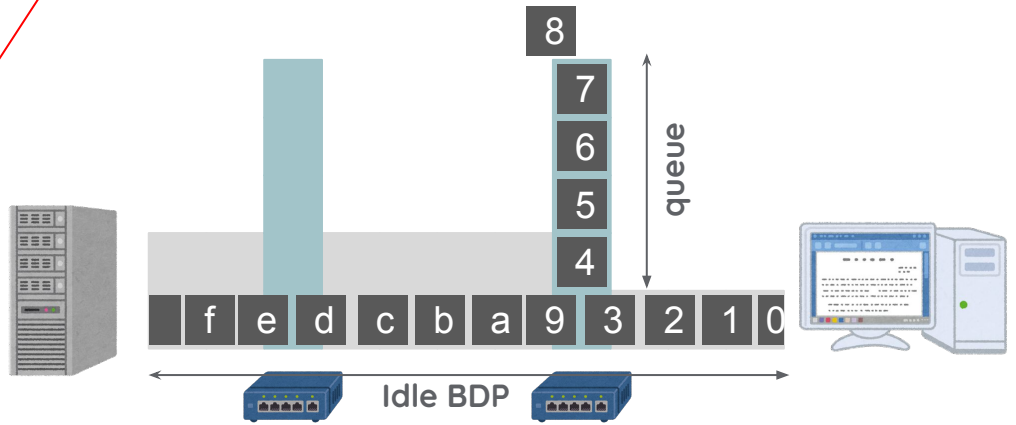


スロースタートと BDP (5G)

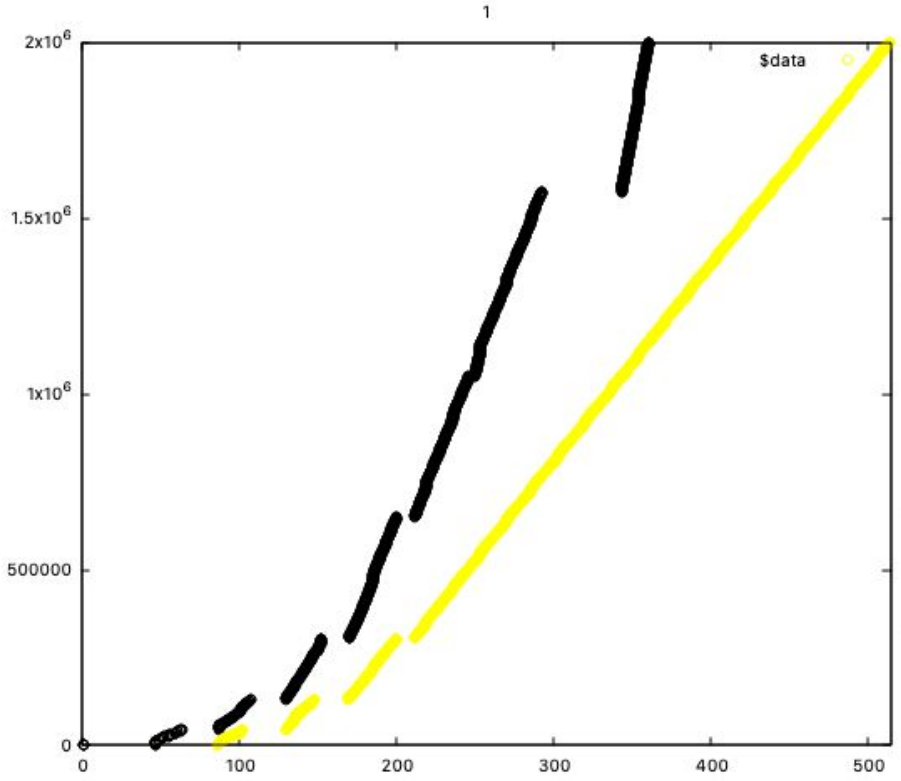
- Idle BDP = 55Mb/s * 0.039s
  - ≒ 2.15Mb ≒ 268KB
  - ≒ 209 packets
- スロースタート :
  - 1RT: 30 packets
  - 2RT: 60 packets
  - 3RT: 120 packets
  - 4RT: 240 packets

ようやくボトルネック飽和





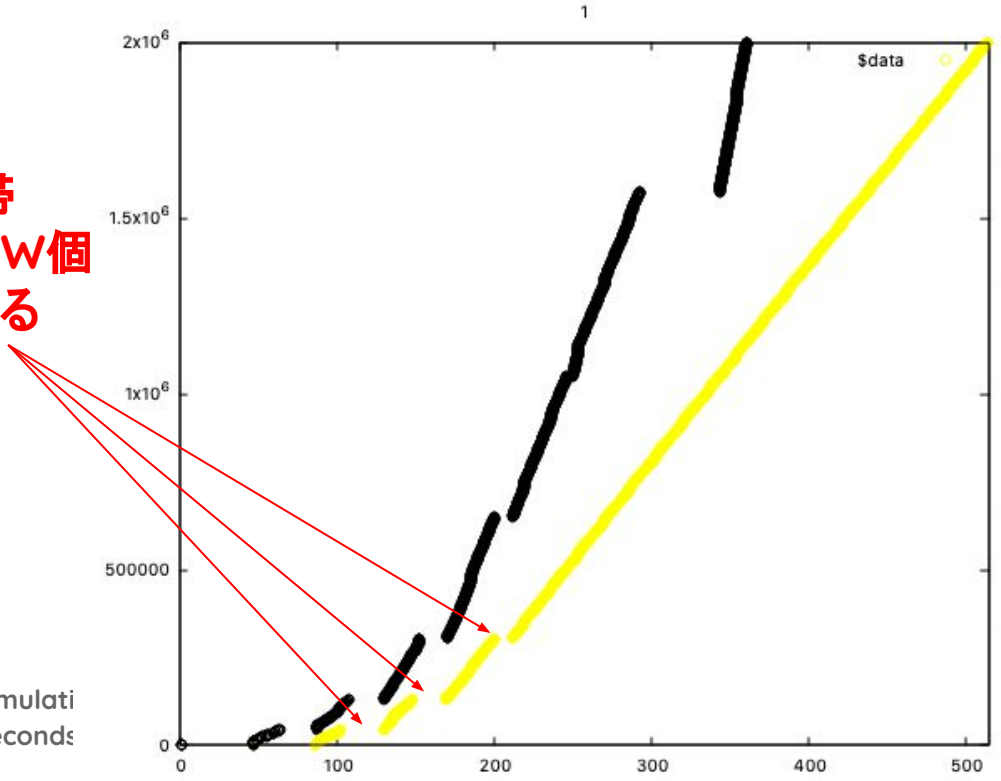
シミュレータによる可視化 (5G)



Vertical axis: bytes sent / acked (cumulati
Horizontal axis: time elapsed (milliseconds
Black dot: packet sent
Yellow dot: ack received (reflects when the receiver received packets)

シミュレータによる可視化 (5G)

何も受信しない時間帯
※送信側が0.5RTでIW個のパケット送信を終えるため

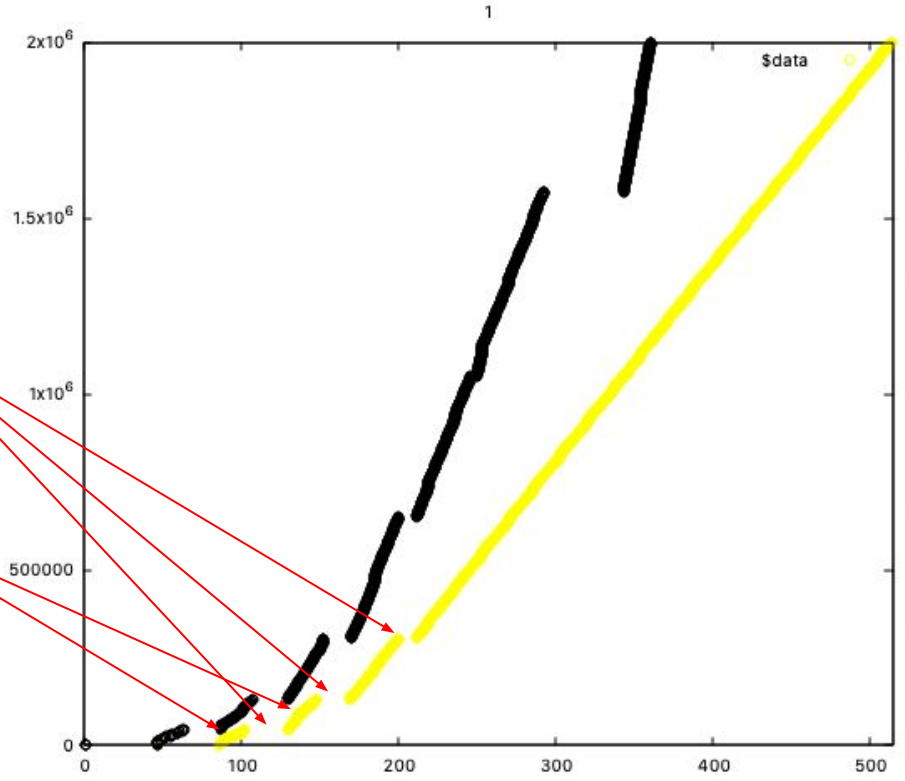


Vertical axis: bytes sent / acked (cumulati
Horizontal axis: time elapsed (milliseconds
Black dot: packet sent
Yellow dot: ack received (reflects when the receiver received packets)

シミュレータによる可視化 (5G)

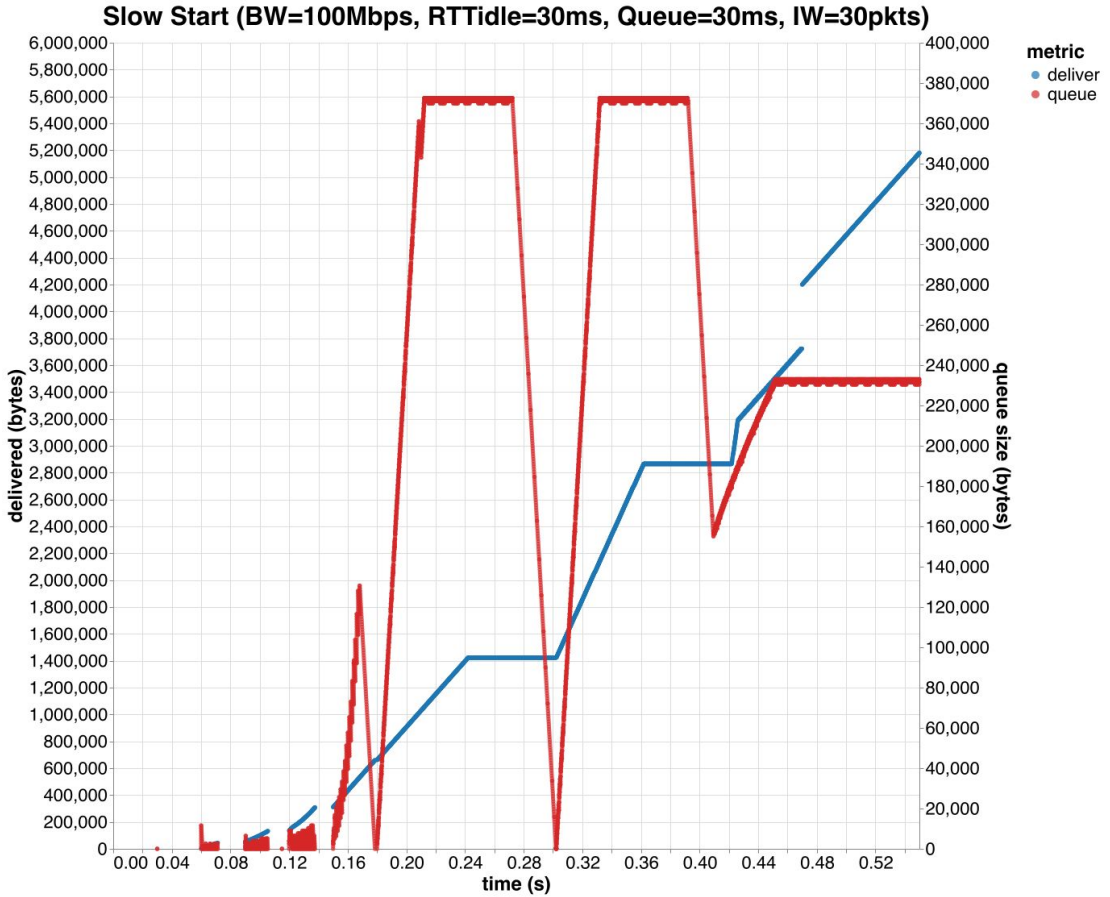
何も受信しない時間帯
※送信側が0.5RTでIW個のパケット送信を終えるため

不十分な送信速度

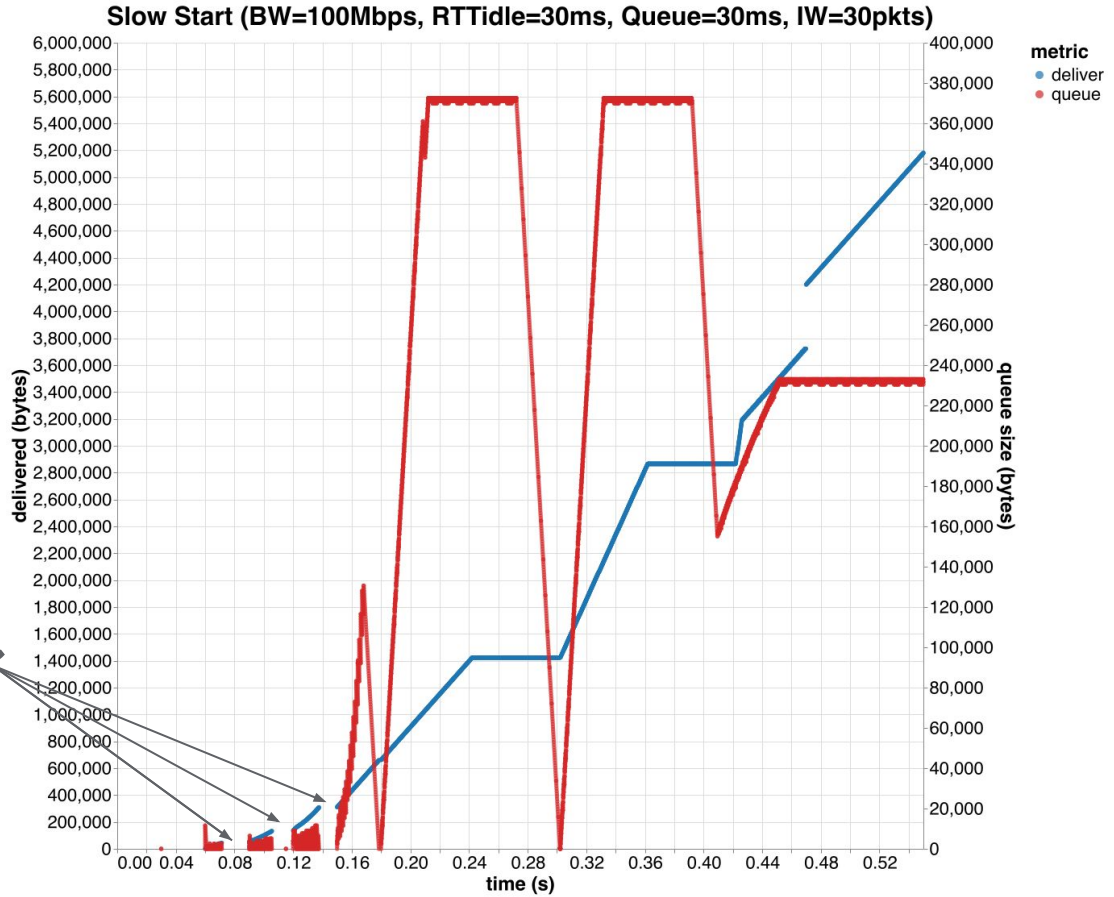


Vertical axis: bytes sent / acked (cumulati
Horizontal axis: time elapsed (milliseconds
Black dot: packet sent
Yellow dot: ack received (reflects when the receiver received packets)

キュー滞留とパケロス（VDSL回線）



キュー滞留とパケロス（VDSL回線）

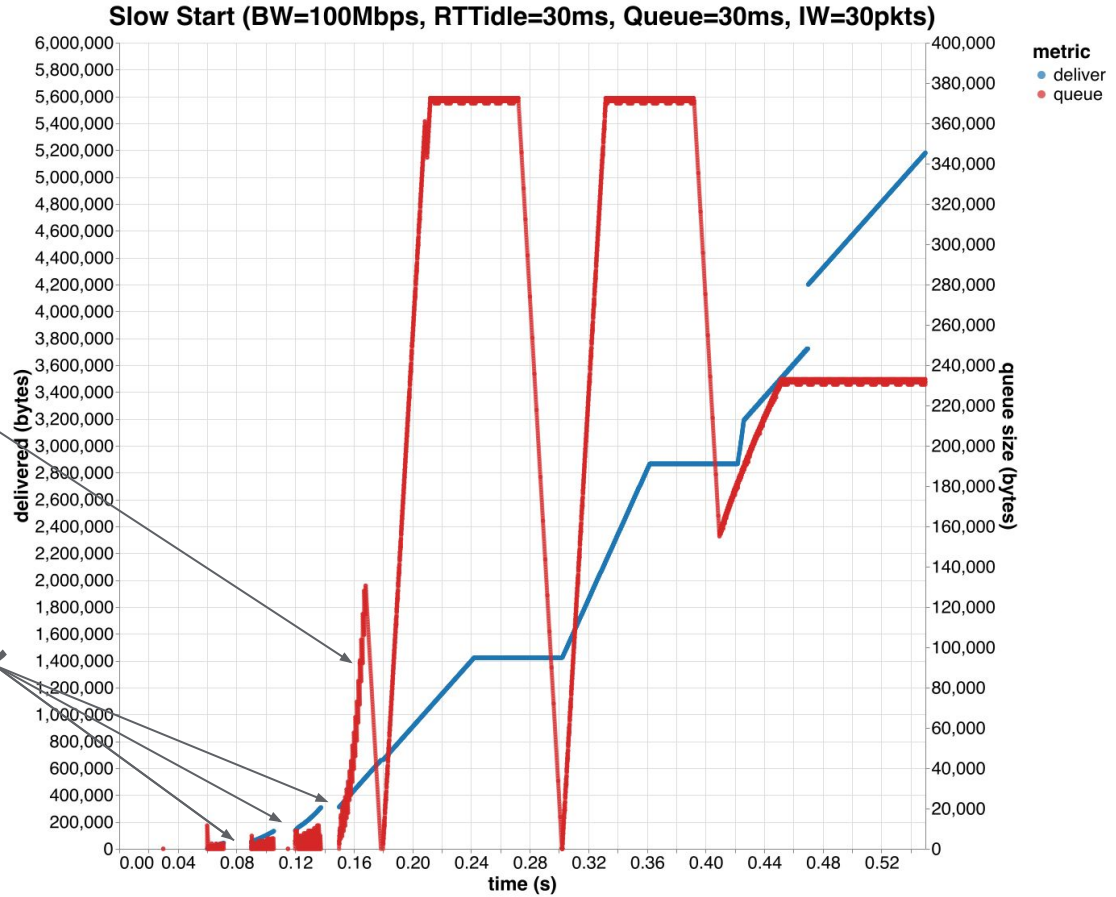


なにも受信しない時間

キュー滞留とパケロス（VDSL回線）

バーストによる
キュー滞留

なにも受信し
ない時間

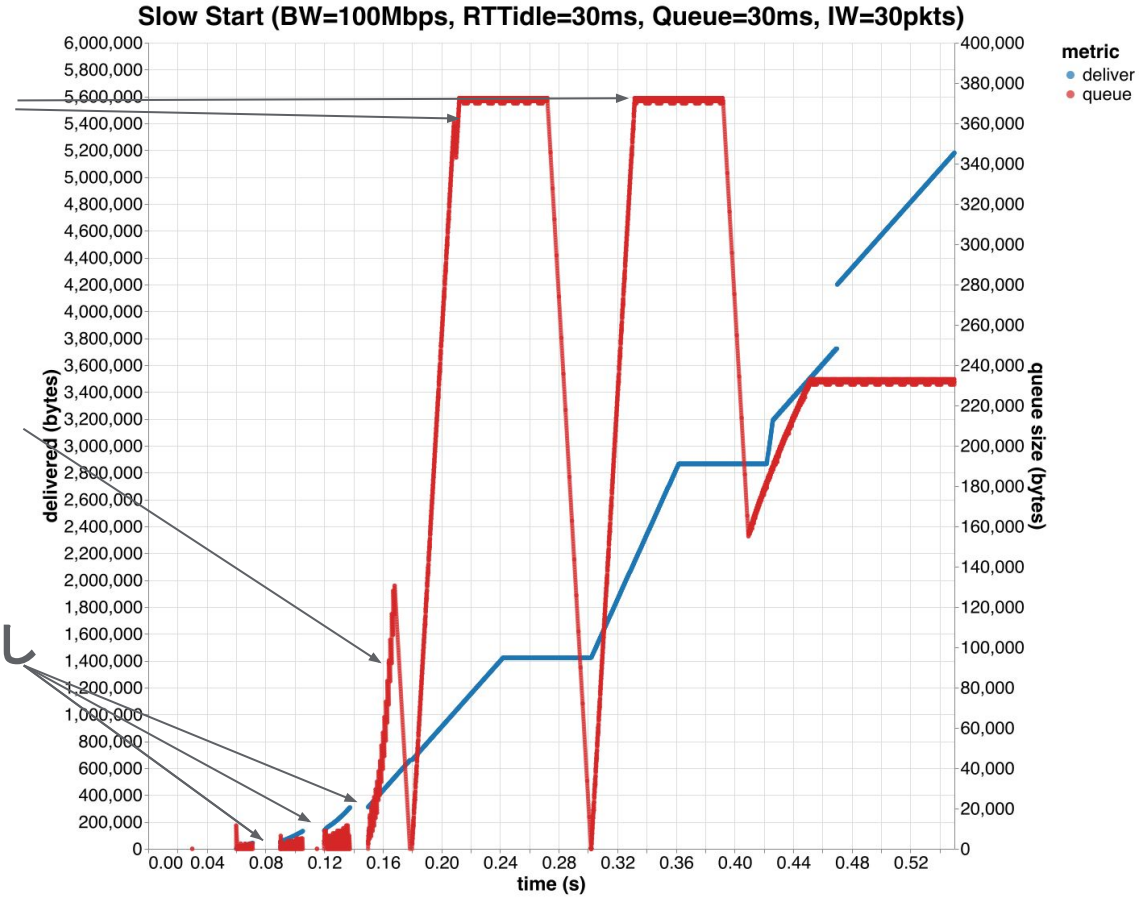


キュー滞留とパケロス（VDSL回線）

キュー溢れによるパケロス（リカバリが必要に）

バーストによるキュー滞留

なにも受信しない時間

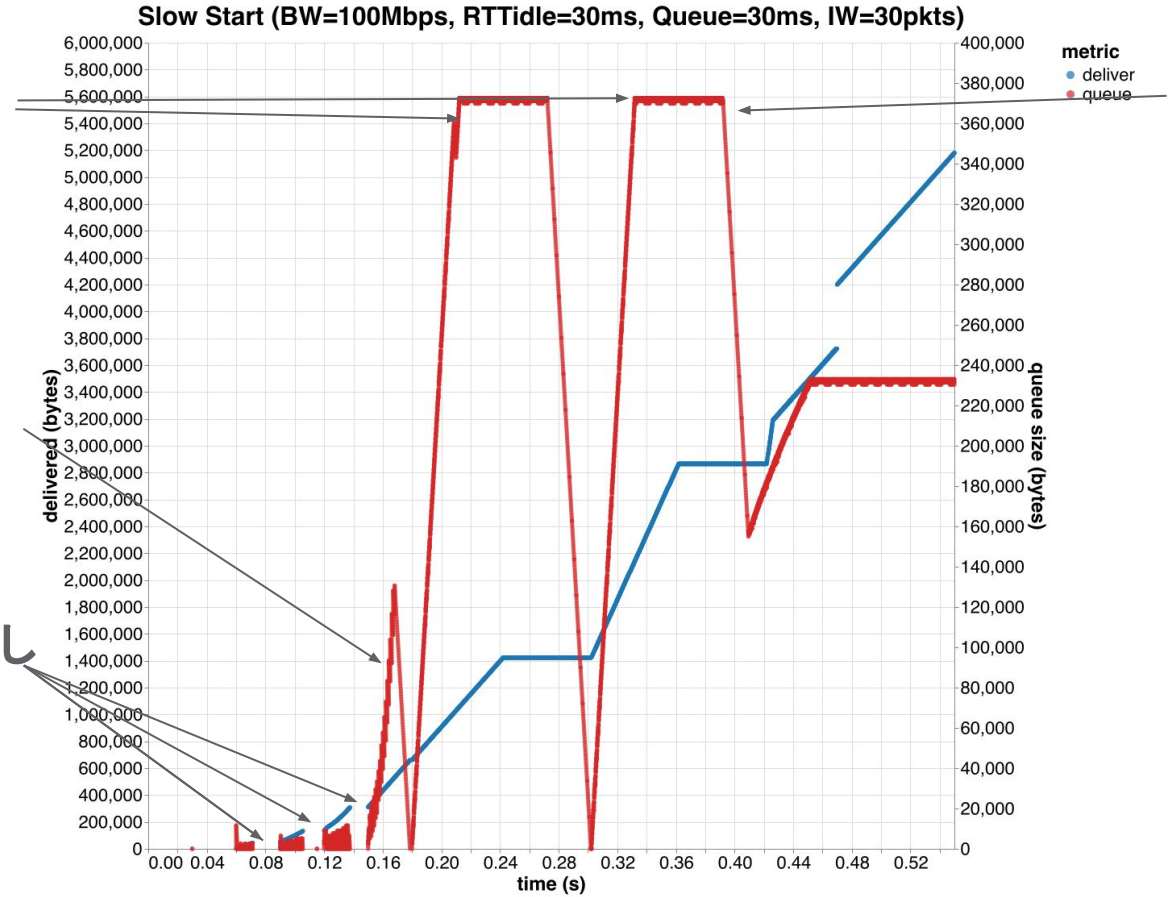


キュー滞留とパケロス（VDSL回線）

キュー溢れによるパケロス（リカバリが必要に）

バーストによるキュー滞留

なにも受信しない時間



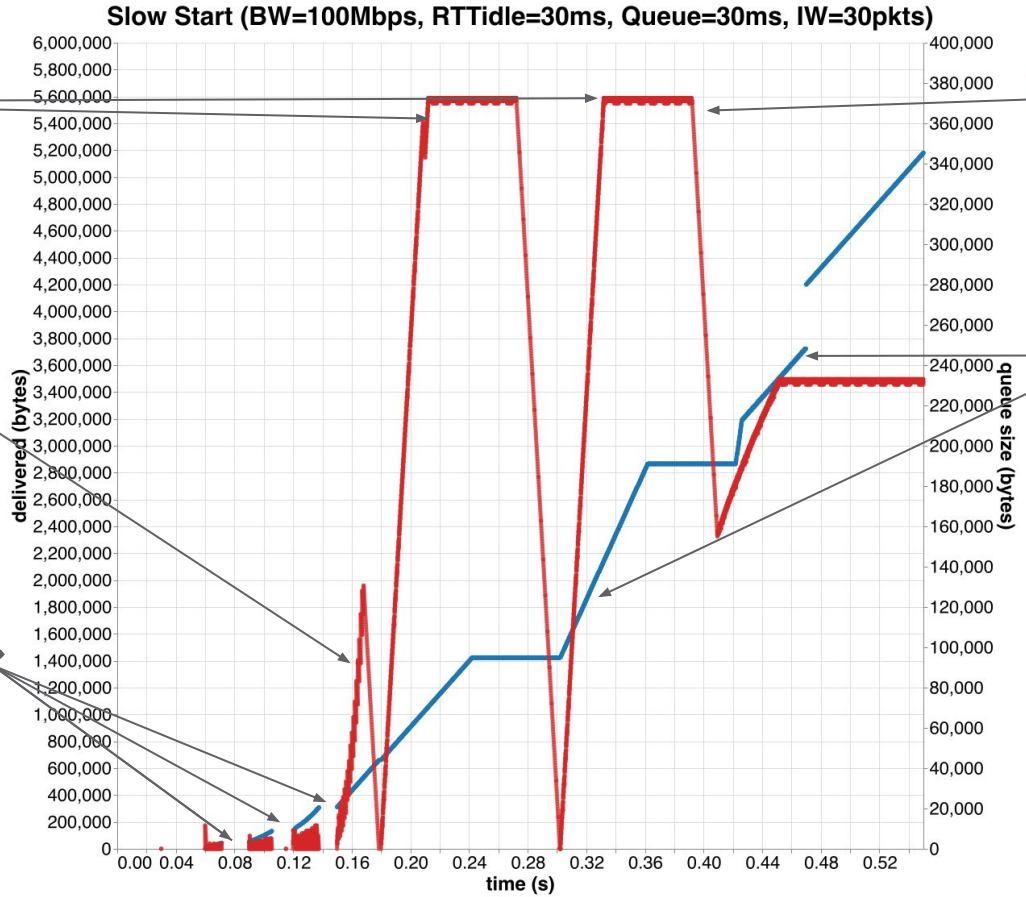
2回目のパケロス（リカバリ）が直後に再発生

キュー滞留とパケロス（VDSL回線）

キュー溢れによるパケロス（リカバリが必要に）

バーストによるキュー滞留

なにも受信しない時間



2回目のパケロス（リカバリ）が直後に再発生

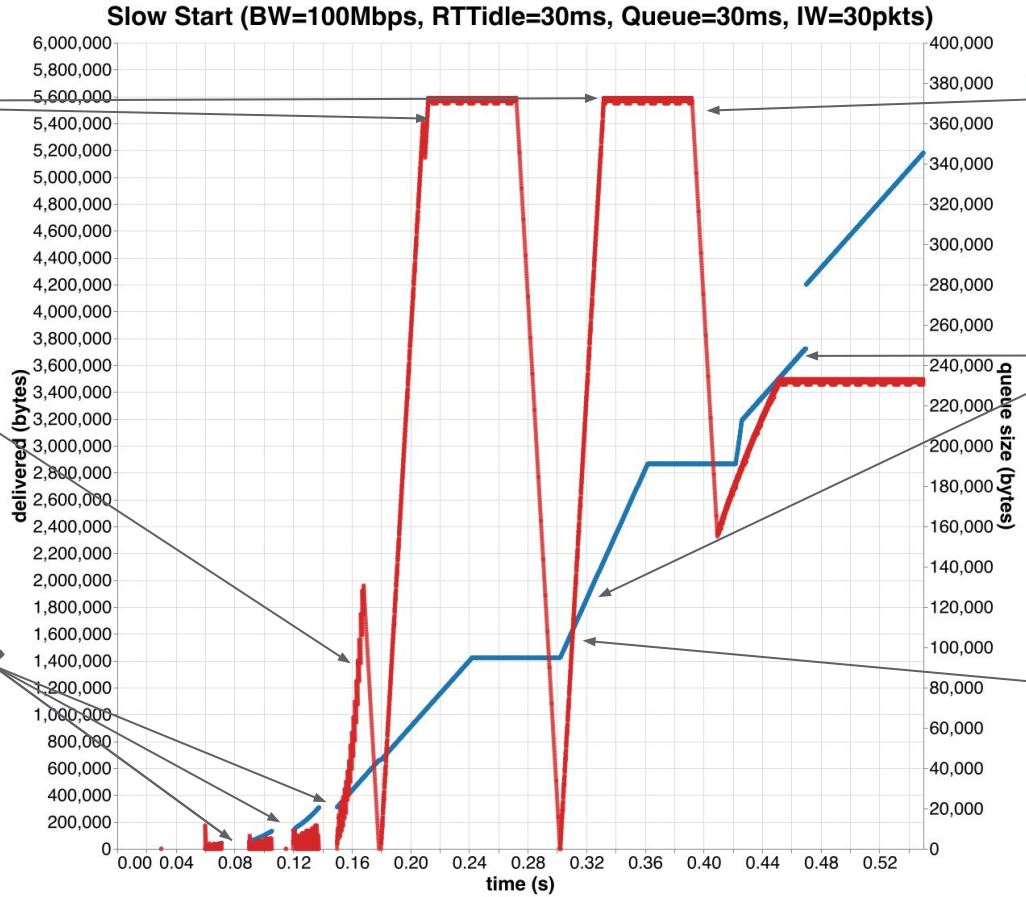
リカバリ完了まで受信したデータは使えない

キュー滞留とパケロス（VDSL回線）

キュー溢れによるパケロス（リカバリが必要に）

バーストによるキュー滞留

なにも受信しない時間



2回目のパケロス（リカバリ）が直後に再発生

リカバリ完了まで受信したデータは使えない

バースト後に無送信時間が長すぎることによるキュー容量低下→ボトルネック不飽和の危険

輻輳制御：理想のスタートアップ

- 利用可能なバンド幅をできるだけ早く使い切る
  - 30パケットより大きな初期送信ウィンドウ　(IW)
  - RTごとに2倍を超える送信量増加

輻輳制御：理想のスタートアップ

- 利用可能なバンド幅をできるだけ早く使い切る
  - 30パケットより大きな初期送信ウィンドウ　(IW)
  - RTごとに2倍を超える送信量増加
- パケロスの悪影響を極小化
  - 短いレスポンスでのパケロス影響を避けるため、最初のパケロスをできる限り遅らせる
  - リカバリの発生回数を減らす
  - リカバリのたびに再送が必要になるパケット数を極小化

輻輳制御：理想のスタートアップ

- 利用可能なバンド幅をできるだけ早く使い切る
  - 30パケットより大きな初期送信ウィンドウ　(IW)
  - RTごとに2倍を超える送信量増加
- パケロスの悪影響を極小化
  - 短いレスポンスでのパケロス影響を避けるため、最初のパケロスをできる限り遅らせる
  - リカバリの発生回数を減らす
  - リカバリのたびに再送が必要になるパケット数を極小化
- バースト送信による、バンド幅飽和以前のタイミングでのパケロスを回避

Rapid Start

- Fastlyが提案 (RFC草案第一版 : 2025年11月)

	Slow Start	Rapid Start
初期送信	0.5 RTTでIWパケット	1 RTTで2 * IWパケット
送信量増加	RTT毎に2倍	RTT毎に3倍 (キュー滞留観測したら 2倍)
リカバリ	CWND *= 0.5	パケロス率に応じた CWNDの調整

※CWND (輻輳制御ウィンドウ): full BDP (idle BDP + キュー容量)の推定値

Rapid Start: 初期送信

- Slow Start:
  - IW個のパケットを 0.5 RTTで送信
- Rapid Start:
  - 2x IWパケットを1 RTTで送信

Rapid Start: 初期送信

- Slow Start:
  - IW個のパケットを 0.5 RTTで送信
- Rapid Start:
  - 2x IWパケットを1 RTTで送信
  - 懸念点: キュー溢れによるパケロスの可能性
    - だが送信速度自体はスロースタートと同じ

Rapid Start: CWND増加

- Slow Start: 2x
- Rapid Start:
  - queue_buildup?[注] ? 3x : 2x

注: 推奨閾値 : *rtt_floor > min(rtt_min + 4ms, rtt_min * 1.1)*, ただし *rtt_floor*は直近1RTTに観測された最小の RTT

Rapid Start: CWND増加

- Slow Start: 2x
- Rapid Start:
  - queue_buildup?[注] ? 3x : 2x
    - キュー滞留は送信がボトルネックより速いから
    - 滞留観測時の増加速度低減はキュー溢れによるパケロスを遅らせる効果

注: 推奨閾値 : *rtt_floor > min(rtt_min + 4ms, rtt_min * 1.1)*, ただし *rtt_floor*は直近1RTTに観測された最小の RTT

スロースタートのリカバリ問題

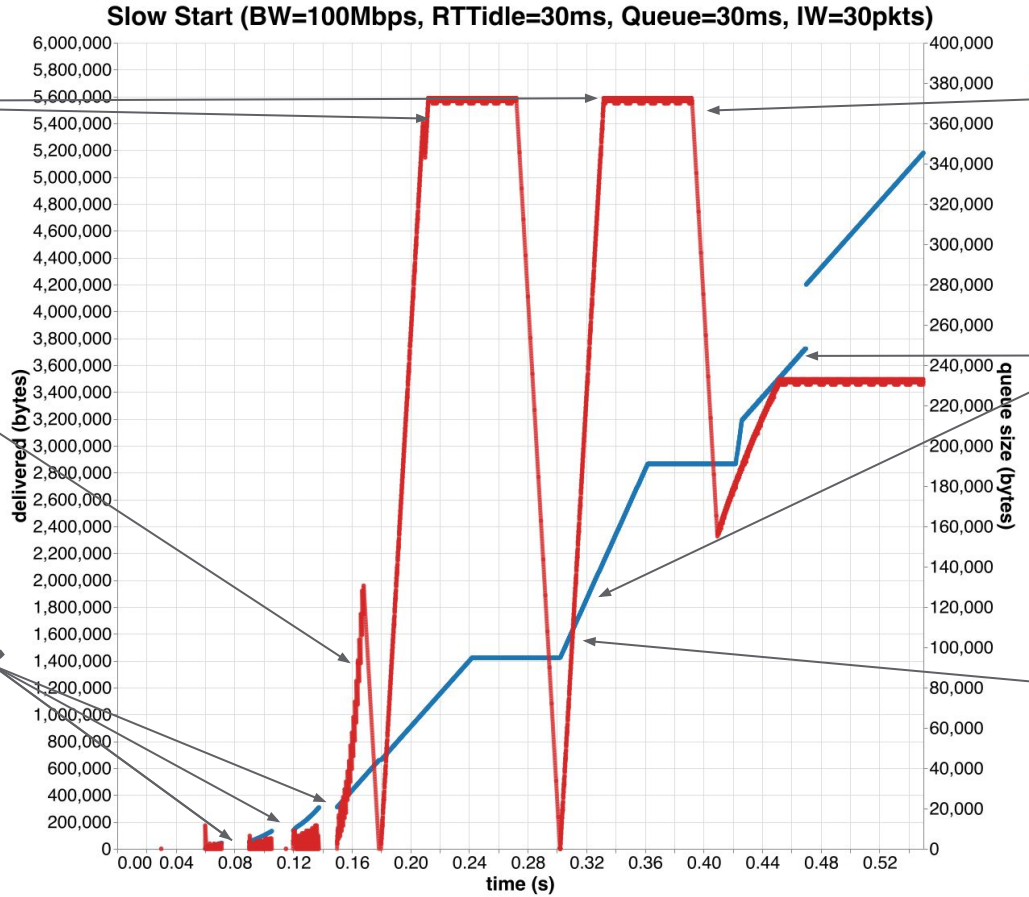
- スロースタートでは 2回目のリカバリが 1回目の直後に発生

復習: キュー滞留とパケロス（VDSL）

キュー溢れによるパケロス（リカバリが必要に）

バーストによるキュー滞留

なにも受信しない時間



2回目のパケロス（リカバリ）が直後に再発生

リカバリ完了まで受信したデータは使えない

バースト後に無送信時間が長すぎることによるキュー容量低下→ボトルネック不飽和の危険

スロースタートのリカバリ問題

- スロースタートでは 2回目のリカバリが 1回目の直後に発生
- なぜなら:

スロースタートのリカバリ問題

- スロースタートでは 2回目のリカバリが 1回目の直後に発生
- なぜなら:
  - パケロスが観測されるのはキューが溢れてから 1 RT経過後

スロースタートのリカバリ問題

- スロースタートでは 2回目のリカバリが 1回目の直後に発生
- なぜなら:
  - パケロスが観測されるのはキューが溢れてから 1 RT経過後
  - スロースタートは RT毎にCWNDを2倍、つまりパケロス観測時のCWNDはfull BDPの約2倍

スロースタートのリカバリ問題

- スロースタートでは 2回目のリカバリが 1回目の直後に発生
- なぜなら:
  - パケロスが観測されるのはキューが溢れてから 1 RT経過後
  - スロースタートは RT毎にCWNDを2倍、つまりパケロス観測時のCWNDはfull BDPの約2倍
  - スロースタートはパケロス観測時に CWNDを½、つまりfull BDPに更新するが、それはつまり、キューが満杯になる量

スロースタートのリカバリ問題

- スロースタートでは 2回目のリカバリが 1回目の直後に発生
- なぜなら:
  - パケロスが観測されるのはキューが溢れてから 1 RT経過後
  - スロースタートは RT毎にCWNDを2倍、つまりパケロス観測時のCWNDはfull BDPの約2倍
  - スロースタートはパケロス観測時に CWNDを½、つまりfull BDPに更新するが、それはつまり、キューが満杯になる量
  - すれすれの値に更新するから、すぐ溢れる

スロースタートのリカバリ問題

- スロースタートでは 2回目のリカバリが 1回目の直後に発生
- なぜなら:
  - パケロスが観測されるのはキューが溢れてから 1 RT経過後
  - スロースタートは RT毎にCWNDを2倍、つまりパケロス観測時のCWNDはfull BDPの約2倍
  - スロースタートはパケロス観測時に CWNDを½、つまりfull BDPに更新するが、それはつまり、キューが満杯になる量
  - すれすれの値に更新するから、すぐ溢れる
- CWNDを¼に更新すると一見良さそうだが ...

スロースタートのリカバリ問題

- スロースタートでは 2回目のリカバリが 1回目の直後に発生
- なぜなら:
  - パケロスが観測されるのはキューが溢れてから 1 RT経過後
  - スロースタートは RT毎にCWNDを2倍、つまりパケロス観測時のCWNDはfull BDPの約2倍
  - スロースタートはパケロス観測時に CWNDを½、つまりfull BDPに更新するが、それはつまり、キューが満杯になる量
  - すれすれの値に更新するから、すぐ溢れる
- CWNDを¼に更新すると一見良さそうだが ...
  - ¾ RTTに渡り無送信が続いてしまい、キューが空に、さらにはパケットがボトルネックを流れない無駄な時間が発生

Rapid Start: リカバリ

- ACKを受信あるいはパケロスを観測するたびに CWNDを**徐々に**削減し、リカバリ完了時の CWNDを

  $CWND_{リカバリ完了時} = 0.5 * ACKされた量$

  に調整

Rapid Start: リカバリ

- ACKを受信あるいはパケロスを観測するたびに CWNDを**徐々に**削減し、リカバリ完了時の CWNDを

  $CWND_{リカバリ完了時} = 0.5 * ACKされた量$

  に調整
  - 理由: リカバリ中の1 RTで観測されるACK量≈full BDPだから

Rapid Start: リカバリ

- ACKを受信あるいはパケロスを観測するたびに CWNDを徐々に削減し、リカバリ完了時の CWNDを

  $CWND_{リカバリ完了時} = 0.5 * ACKされた量$

  に調整
  - 理由: リカバリ中の1 RTで観測されるACK量≈full BDPだから
- 利点:
  - 増加速度に依存しない手法
  - CWNDを徐々に削減するため、無送信にともなうキューレベルの過剰低下が発生しない

Rapid Start: リカバリ

- リカバリ開始時 :

  *cwnd *= 5/6*

- ACK受信時:

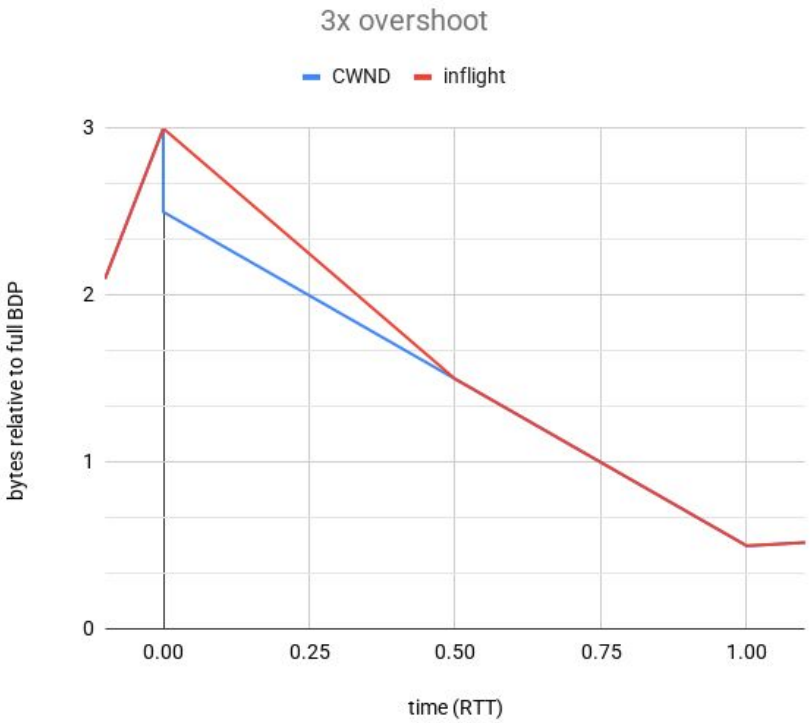
  *cwnd -= 1/3 * bytes_newly_acked*

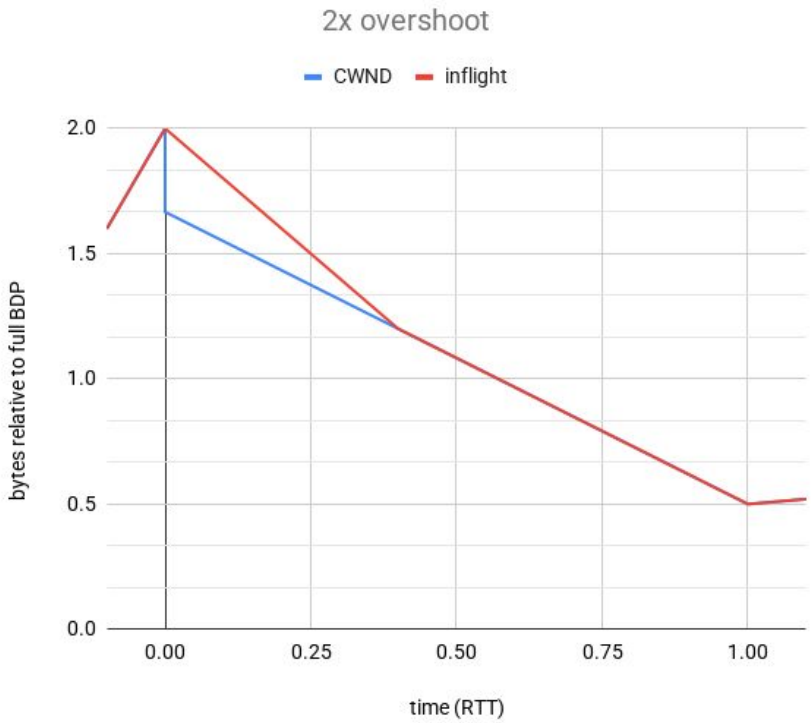
- パケロス確認時 :

  *cwnd -= 5/6 * bytes_newly_lost*

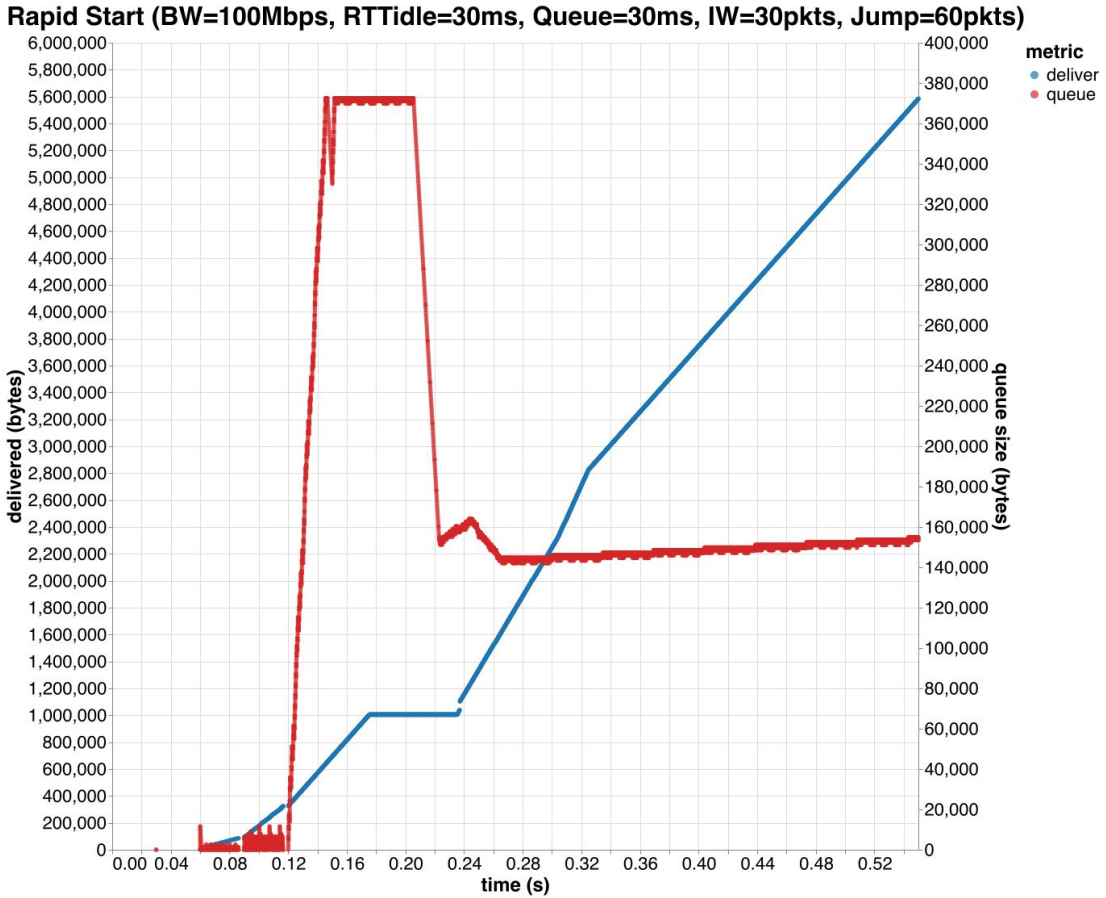
定数の導出については draft-kazuho-ietf-rapid-start-02 参照

Rapid Start: リカバリ

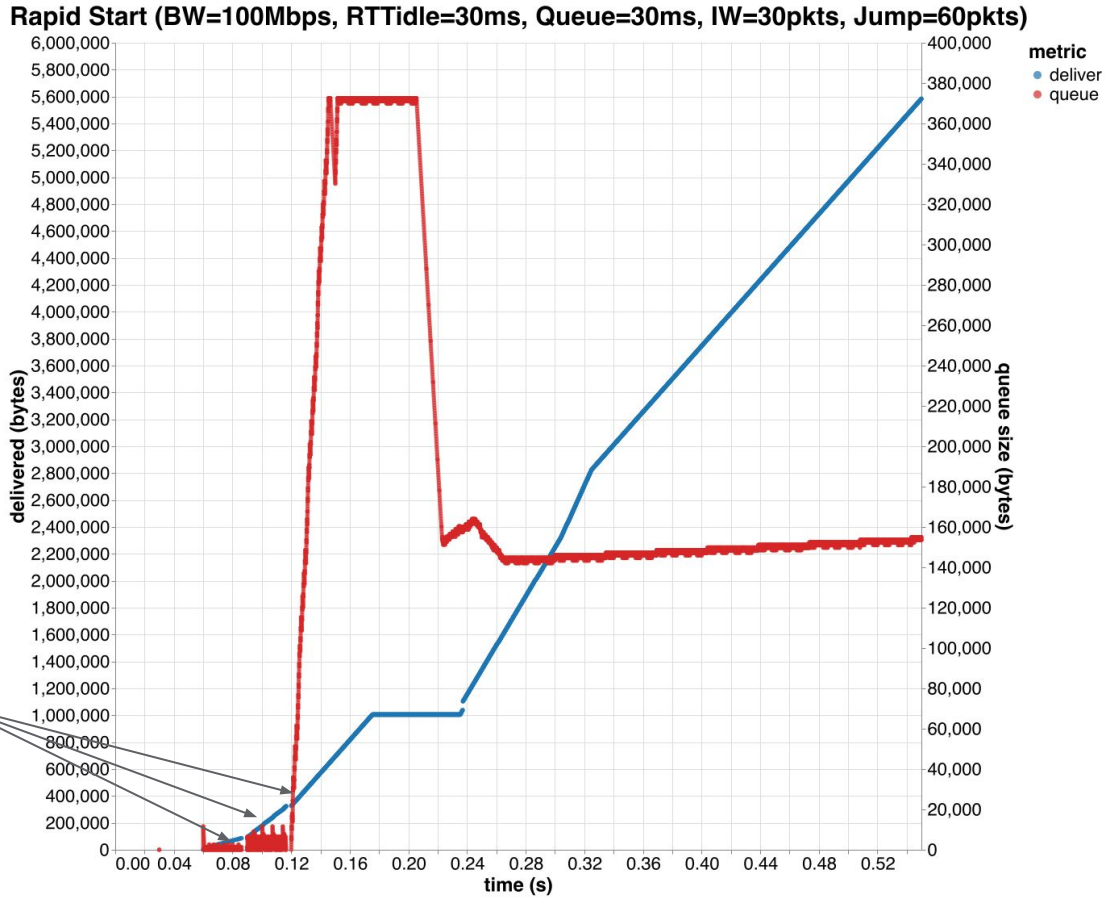




Rapid Startシミュレーション (VDSL)



Rapid Startシミュレーション (VDSL)

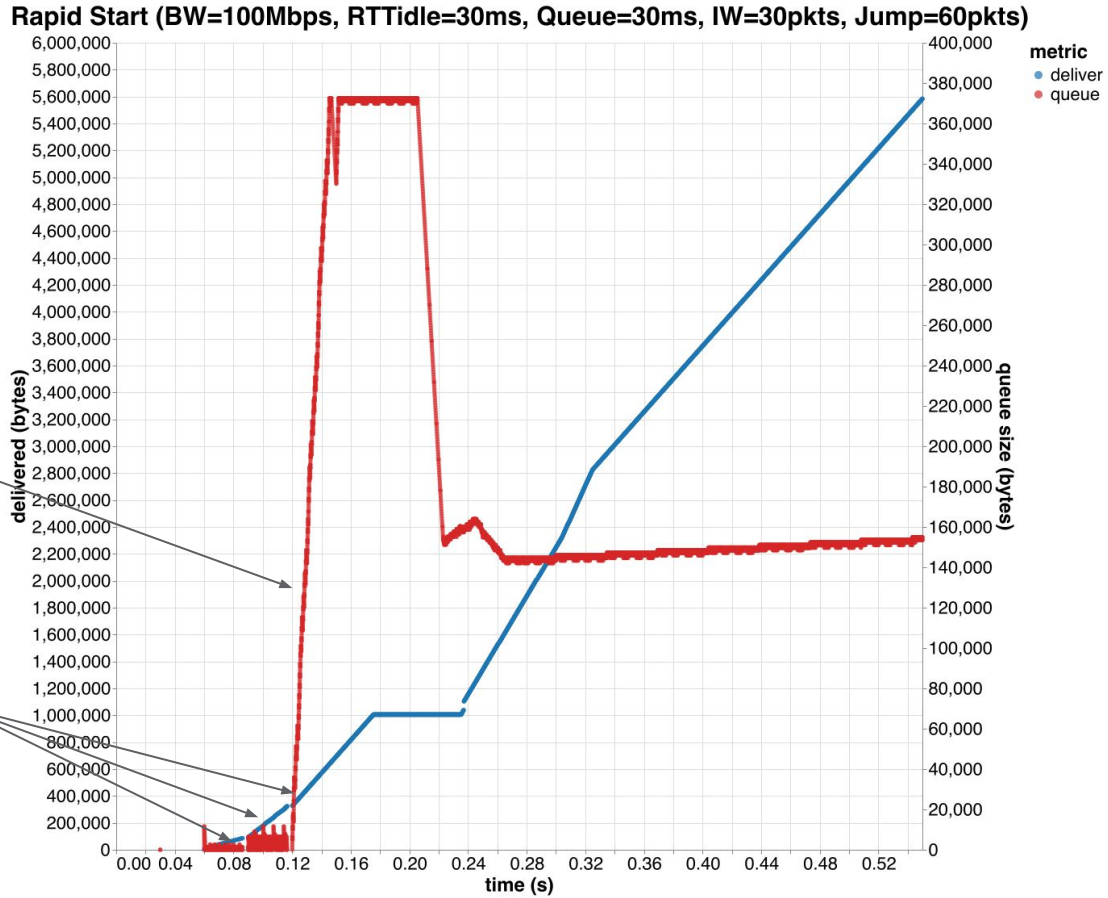


なにも受信しない時間はない

Rapid Startシミュレーション (VDSL)

パケットがキューに滞留開始

なにも受信しない時間はない

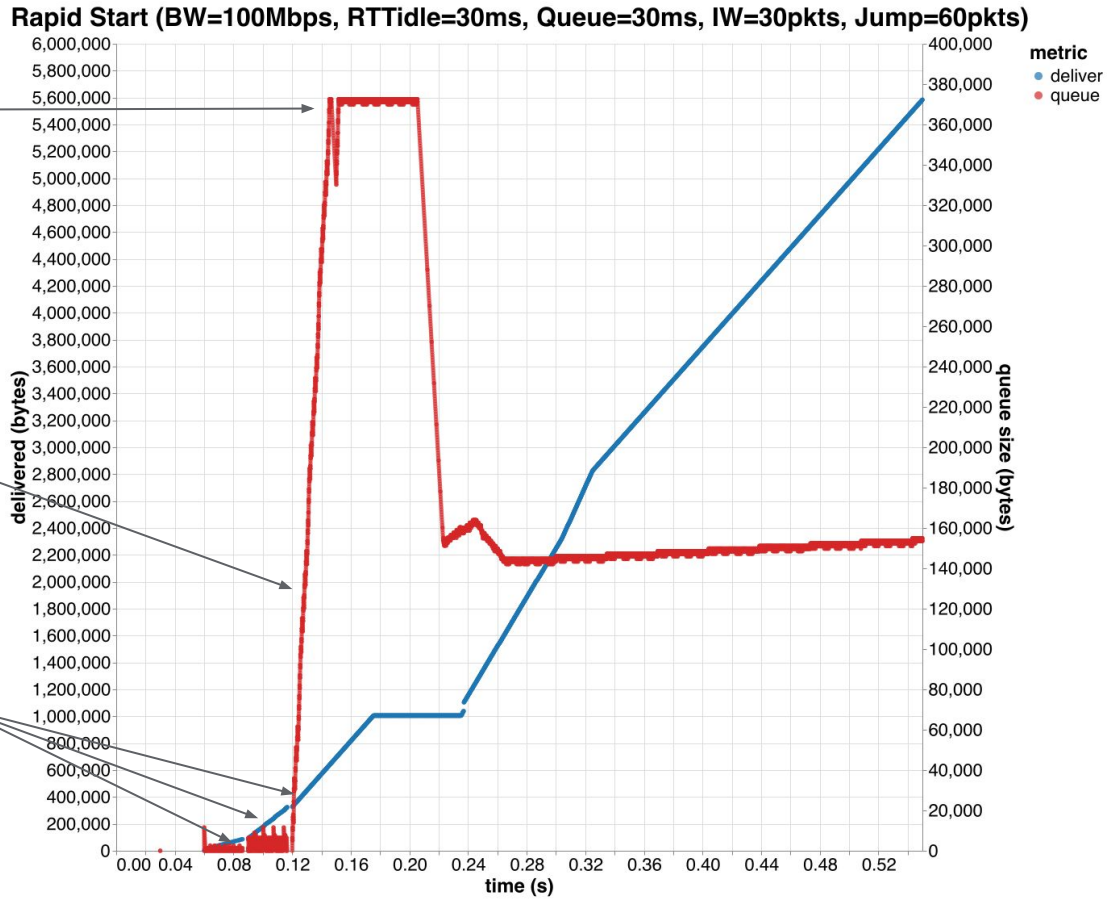


Rapid Startシミュレーション (VDSL)

パケロス

パケットがキューに滞留開始

なにも受信しない時間はない

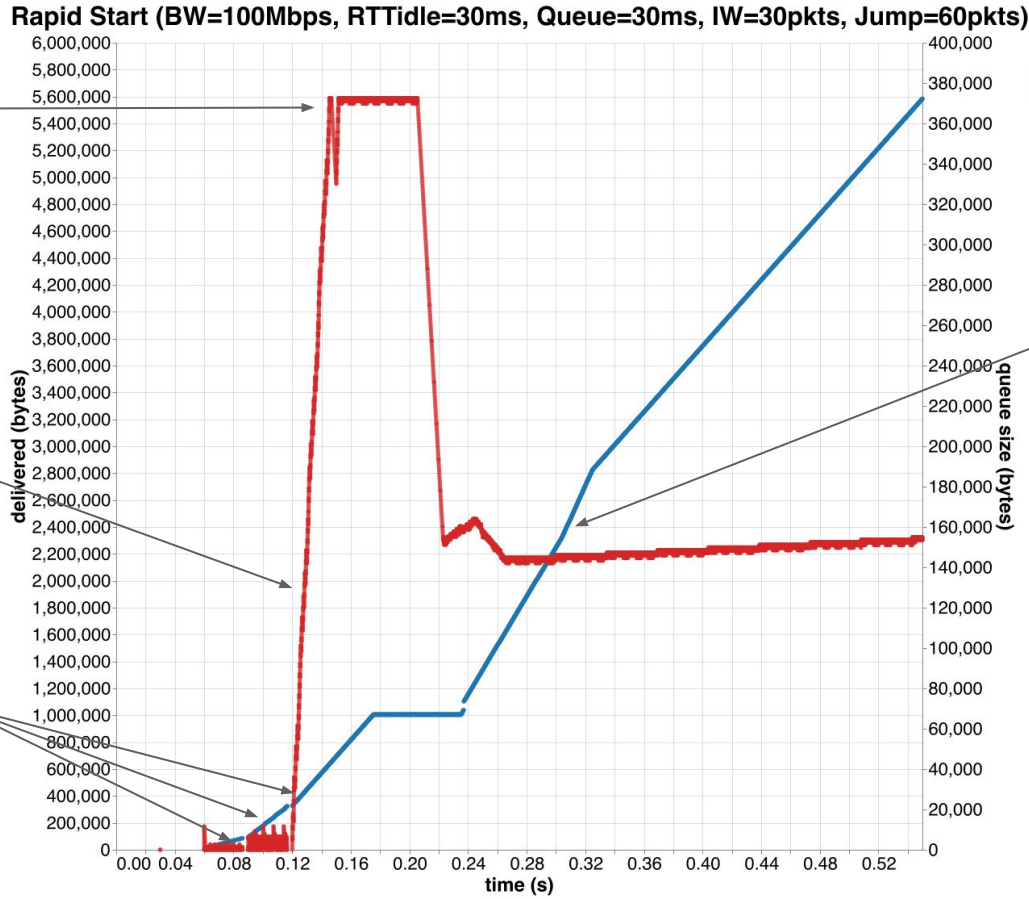


Rapid Startシミュレーション (VDSL)

パケロス

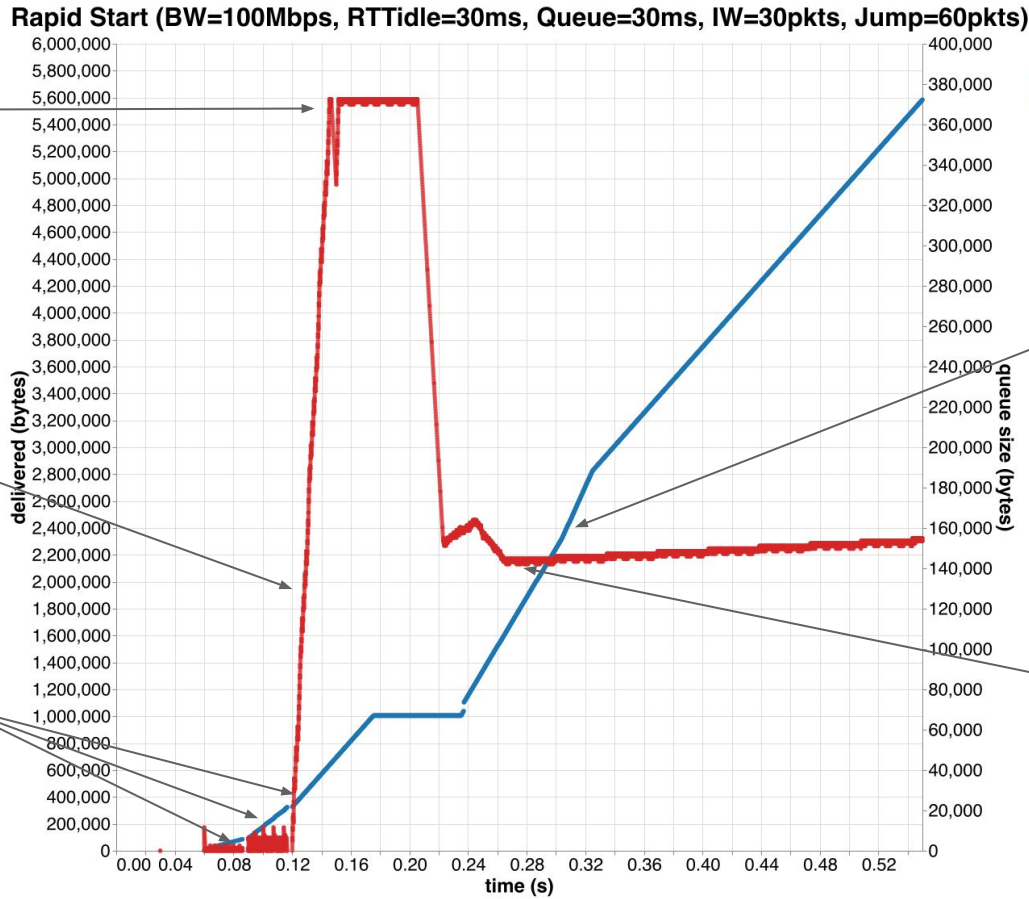
パケットがキューに滞留開始

なにも受信しない時間はない



リカバリは1回だけ、ただしfull BDPを+2倍超過して送信したため回復に時間を要する

Rapid Startシミュレーション (VDSL)



パケロス

パケットがキューに滞留開始

なにも受信しない時間はない

リカバリは1回だけ、ただしfull BDPを+2倍超過して送信したため回復に時間を要する

キューを適切なレベルに調整

実環境での評価

実環境での評価：4グループに分割

	初期送信	増加	リカバリ
対称群 (slow start)	30 pkts / 0.5 RTT	2x	CWND *= 0.5
jumpstart	60 pkts / 1 RTT	2x	CWND *= 0.5
rapid-wo-jump	30 pkts / 0.5 RTT	3x / 2x	パケロス率に応じたCWND調整
rapidstart	60 pkts / 1 RTT		

- HTTP/3接続を4グループに分割
- 接続の最初のレスポンスが、200KB以上のキャッシュ済オブジェクトであった場合、同オブジェクトの全てのバイトが ACKされるまでの時間を TTLBとして保存
- 約1週間に渡り、世界の 7拠点 (東・東南アジア、東西ヨーロッパ、アフリカ、南北アメリカ) で測定

TTLBとともに保存されるデータ例

{"module":"h2o","type":"h3s_stream0_ttlb","tid":397502,"time":1773026516280,"conn_id":1907798,"method":"GET",**"content_length":226578,"ttlb":364**,"num-packets.received":24,"num-packets.decryption-failed":0,**"num-packets.sent":191,"num-packets.lost":0**,"num-packets.lost-time-threshold":0,"num-packets.ack-received":191,"num-packets.late-acked":0,"num-packets.initial-received":2,"num-packets.zero-rtt-received":0,"num-packets.handshake-received":2,"num-packets.initial-sent":1,"num-packets.zero-rtt-sent":0,"num-packets.handshake-sent":4,"num-packets.received-out-of-order":0,"num-packets.received-ecn-ect0":0,"num-packets.received-ecn-ect1":0,"num-packets.received-ecn-ce":0,"num-packets.acked-ecn-ect0":0,"num-packets.acked-ecn-ect1":0,"num-packets.acked-ecn-ce":0,"num-packets.sent-promoted-paths":0,"num-packets.ack-received-promoted-paths":0,"num-packets.max-delayed":0,"num-packets.delayed-used":0,"num-bytes.received":4737,"num-bytes.sent":236943,"num-bytes.lost":0,"num-bytes.ack-received":236895,"num-bytes.stream-data-sent":231728,"num-bytes.stream-data-resent":226,"num-frames-received.padding":3259,"num-frames-received.ping":1,"num-frames-received.ack":19,"num-frames-received.reset_stream":0,"num-frames-received.stop_sending":0,"num-frames-received.crypto":2,"num-frames-received.new_token":0,"num-frames-received.stream":2,"num-frames-received.max_data":0,"num-frames-received.max_stream_data":0,"num-frames-received.max_streams_bidi":0,"num-frames-received.max_streams_uni":0,"num-frames-received.data_blocked":0,"num-frames-received.stream_data_blocked":0,"num-frames-received.streams_blocked":0,"num-frames-received.new_connection_id":0,"num-frames-received.retire_connection_id":0,"num-frames-received.path_challenge":0,"num-frames-received.path_response":0,"num-frames-received.transport_close":0,"num-frames-received.application_close":0,"num-frames-received.handshake_done":0,"num-frames-received.datagram":0,"num-frames-received.ack_frequency":0,"num-frames-received.immediate_ack":0,"num-frames-sent.padding":0,"num-frames-sent.ping":1,"num-frames-sent.ack":3,"num-frames-sent.reset_stream":0,"num-frames-sent.stop_sending":0,"num-frames-sent.crypto":7,"num-frames-sent.new_token":2,"num-frames-sent.stream":188,"num-frames-sent.max_data":0,"num-frames-sent.max_stream_data":0,"num-frames-sent.max_streams_bidi":0,"num-frames-sent.max_streams_uni":0,"num-frames-sent.data_blocked":0,"num-frames-sent.stream_data_blocked":0,"num-frames-sent.streams_blocked":0,"num-frames-sent.new_connection_id":6,"num-frames-sent.retire_connection_id":0,"num-frames-sent.path_challenge":0,"num-frames-sent.path_response":0,"num-frames-sent.transport_close":0,"num-frames-sent.application_close":0,"num-frames-sent.handshake_done":1,"num-frames-sent.datagram":0,"num-frames-sent.ack_frequency":0,"num-frames-sent.immediate_ack":0,"num-paths.created":0,"num-paths.validated":0,"num-paths.validation-failed":0,"num-paths.migration-elicited":0,"num-paths.promoted":0,"num-paths.closed-no-dcid":0,"num-paths.ecn-validated":0,"num-paths.ecn-failed":1,"num-ptos":1,"num-handshake-timeouts":0,"num-initial-handshake-exceeded":0,"num-jumpstart-applicable":1,"quic.jumpstart.applicable":1,**"num-rapid-start":0**,"num-paced":1,"num-respected-app-limited":0,"handshake-confirmed-msec":369,"jumpstart.prev-rate":0,"jumpstart.prev-rtt":0,"jumpstart.new-rtt":106,**"jumpstart.cwnd":0**,"quic.jumpstart.time-to-idle":647,"token-sent.at":0,"token-sent.rate":579889,"token-sent.rtt":67,**"rtt.minimum":66,"rtt.smoothed":81,"rtt.variance":19,"rtt.latest":75**,"loss-thresholds.use-packet-based":1,"loss-thresholds.time-based-percentile":128,"cc.cwnd":273280,"cc.ssthresh":4294967295,"cc.cwnd-initial":44160,"cc.cwnd-exiting-slow-start":0,"cc.exit-slow-start-at":9223372036854775807,"cc.cwnd-exiting-jumpstart":0,"cc.cwnd-minimum":4294967295,"cc.cwnd-maximum":273280,"cc.num-loss-episodes":0,"cc.num-ecn-loss-episodes":0,"delivery-rate.latest":210149,"delivery-rate.smoothed":739154,"delivery-rate.stdev":1078961,"num-sentmap-packets-largest":89}

データ解析

- データセットのサイズ：
  - 2000万行のLDJSON: 80GB (gzip後に3.7GB)
- さまざまなクエリを実行できるツールが必要 ：

データ解析

- データセットのサイズ：
  - 2000万行のLDJSON: 80GB (gzip後に3.7GB)
- さまざまなクエリを実行できるツールが必要 ：
  - jqが当然の解だが ...

jqの問題

- クエリ言語がわかりにくい

jqの問題

- クエリ言語がわかりにくい
- 遅い

jqの問題

- クエリ言語がわかりにくい
- 遅い
- 巨大なLDJSONの処理が苦手
  - 例: `| min` を実行すると全データをバッファ
  - ログ解析で求められるのは、ほとんどが map-reduce 的な処理
  - だから、バッファせずにストリーム処理できるはずなのに…

例: rtt.minimumの統計

```
jq -s '{
  "min": (map(."rtt.minimum") | min),
  "max": (map(."rtt.minimum") | max),
  "avg":
    (map(."rtt.minimum") | add / length),
}'
```

例: rtt.minimumの統計

```
jq -s '{
  "min": (map(."rtt.minimum") | min),
  "max": (map(."rtt.minimum") | max),
  "avg":
    (map(."rtt.minimum") | add / length),
}'
```

-sは全データをバッファして処理、つまりLDJSONがRAM容量より大きいと固まる

例: rtt.minimumの統計

```
jq -n '
  reduce inputs as $o (
    {min: null, max: null, sum: 0, n: 0};
    ($o."rtt.minimum") as $x | {
      min: (if .min == null or $x < .min then $x else .min end),
      max: (if .max == null or $x > .max then $x else .max end),
      sum: (.sum + $x),
      n: (.n + 1),
    }
  ) | {
    min,
    max,
    avg: (.sum / .n),
  }'
```

-nをつけるとストリーム処理になるが、min/max/avgといった統計演算のロジックの手書きが必要に

じゃあrubyで書くか…

- ストリーム処理を書くのは簡単だし
- JSONパーサは速いし
- しかもJIT-compiledされる

じゃあrubyで書くか…

- ストリーム処理を書くのは簡単だし
- JSONパーサは速いし
- しかもJIT-compiledされる
- しかし、一方で…
  - ワンライナーにはとても収まらない
  - 結果、大量のオプションをもつスクリプトになる
    - 保守が大変

じゃあrubyで書くか…

- ストリーム処理を書くのは簡単だし
- JSONパーサは速いし
- しかもJIT-compiledされる
- しかし、一方で…
  - ワンライナーにはとても収まらない
  - 結果、大量のオプションをもつスクリプトになる
    - 保守が大変
- AIに書かせてもいいが…
  - 試行錯誤する様々なクエリが正しく実装されているか、どうやって検証するの？

jq的な何かをrubyで再実装

- SQLに近いクエリ文法を ruby DSLで実現

jq的な何かをrubyで再実装

- SQLに近いクエリ文法を ruby DSLで実現
- クエリをevalでrubyコードにコンパイル
  - rubyのJITインタプリタやライブラリを直に利用

jq的な何かをrubyで再実装

- SQLに近いクエリ文法を ruby DSLで実現
- クエリをevalでrubyコードにコンパイル
  - rubyのJITインタプリタやライブラリを直に利用
- NDJSONのストリーム処理

jrf

```
jrf '{
  "min" => min(_["rtt.minimum"]),
  "max" => max(_["rtt.minimum"]),
  "avg" => average(_["rtt.minimum"]),
}'
```

jrf

```
# フィルタしてから抽出
jrf 'select(_["x"] > 10) >> _["foo"]'

# 集約演算
jrf 'select(_["item"] == "Apple") >> sum(_["count"])'
jrf 'percentile(_["ttlb"], 0.50)'

# キー毎にまとめてから集約演算
jrf 'group_by(_["item"]) {
  |row| sum(row["count"] * row["price"])
}'
```

jrf

- クエリ文法：ステージを `>>` で連結
  - 各ステージは rubyのブロック
- フィルタ：
  - select(expr)
- 射影：
  - _["foo"]
- 集約演算：
  - min(expr), max(expr), sum(expr), …
  - reduce(initial) { 任意のrubyコード }

jrf - 内部構造

各ステージの式をメソッドに
変換して実行

```
class Stage
  def initialize(block, src : nil)
    ...
    @ctx = Class.new(RowContext) do
      define_method(:__jrf_expr__, &block)
    end
  end
end

# 初期化:
Stage.new(eval("proc { #{stage[:src]} }", ...))
```

jrf - 自動並列化

- 一般的なログ解析では：
  - 条件抽出と変換を行ったあとに集約演算
  - ログは複数のファイル

```
jrf -P 10 'filter >> transform >> filter >> transform >> reduce'
```

jrf - 自動並列化

- 一般的なログ解析では：
  - 条件抽出と変換を行ったあとに集約演算
  - ログは複数のファイル
- よって、以下の処理をファイル毎に並列実行することが可能　：
  - 条件抽出と変換が続く間の全てのステージ

```
jrf -P 10 'filter >> transform >> filter >> transform >> reduce'
```

jrf - 自動並列化

- 一般的なログ解析では：
  - 条件抽出と変換を行ったあとに集約演算
  - ログは複数のファイル
- よって、以下の処理をファイル毎に並列実行することが可能　：
  - 条件抽出と変換が続く間の全てのステージ
  - 一部の集約演算（e.g., min, max, sum）
    - 各スレッドで中間結果を計算したあと、全スレッドの値をまとめて最終結果を計算

```
jrf -P 10 'filter >> transform >> filter >> transform >> reduce'
```

jrf - 自動並列化

- jrf 内部では以下のように実装：
  1. 最初の JSONオブジェクトを先頭ステージから順に流してみて、どこまで並列化できるか調査
  2. forkを呼んで、ファイル毎にプロセスを起動し並列処理
  3. 各プロセスは結果を NDJSONとしてpipeに書き出し
  4. メインプロセスは pipeを読んで集約し、後続の並列化できなかったステージ群を実行

jrf - 性能測定

	950MB (single file)		81.4GB (29 files)	
	min(rtt.minimum)	TTLB percentile delta	min(rtt.minimum)	TTLB percentile delta
jq -s				
jq -n				
jrf				
jrf -P 10				

all units in seconds

jrf - 性能測定

- min:
  - `jq -s 'map(."rtt.minimum") | min'`
  - ```
    jq -n 'reduce inputs."rtt.minimum" as $x
       (null;
        if . == null or $x < . then $x else . end)'
    ```
  - `jrf 'min(_["rtt.minimum"])'`
  - `jrf -P 10 'min(_["rtt.minimum"])'`
    ```

jrf - 性能測定

- TTLB percentile delta:
  - ```
    jq -n '
      include "helpers";
      0.1 as $step | reduce inputs as $row (
        {"baseline": [], "jumpstart": [],
          "rapid-no-jump": [], "rapidstart": []};
        if ($row | base_cond(200000; 400000))
          then .[$row | group_name] += [$row.ttlb]
          else .
          End
        ) | with_entries(.value |= percentiles($step))
        | .baseline as $baseline
        | with_entries(select(.key != "baseline"))
        | with_entries(
          .value |= [range(0; length) as $i | (.[$i] / $baseline[$i] - 1)]
        )'
    ```
    ```

jrf - 性能測定

- TTLB percentile delta:
  - ```
    jrf 'select(base_cond(_, 200000, 400000))
      >> [group_name(_), _["ttlb"]]
      >> group_by(_[0]) {
        percentile(_[1], $perc ||= 0.05.step(0.95, 0.1))
      } >> map_values{|arr|
        arr.zip(_["baseline"]).map {|v,bv| v.to_f / bv - 1 }
      } >> _.reject{|k| k == "baseline"}'
    ```
  - ```
    jrf -P 10 '...(same as above)...'
    ```

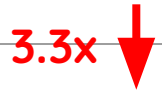
jrf - 性能測定

	950MB (single file)		81.4GB (29 files)	
	min(rtt.minimum)	TTLB percentile delta	min(rtt.minimum)	TTLB percentile delta
jq -s	7.93	8.52	out of memory	
jq -n	7.45	13.59	667.44	> 1800
jrf	2.29	2.39	226.80	240.92
jrf -P 10	2.27	2.38	31.41	31.69

all units in seconds

jrf - 性能測定

	950MB (single file)		81.4GB (29 files)	
	min(rtt.minimum)	TTLB percentile delta	min(rtt.minimum)	TTLB percentile delta
jq -s	7.93	8.52	out of memory	
jq -n	7.45	13.59	667.44	> 1800
jrf	2.29	2.39	226.80	240.92
jrf -P 10	2.27	2.38	31.41	31.69



all units in seconds

jrf - 性能測定

	950MB (single file)		81.4GB (29 files)	
	min(rtt.minimum)	TTLB percentile delta	min(rtt.minimum)	TTLB percentile delta
jq -s	7.93	8.52	out of memory	
jq -n	7.45	13.59	667.44	> 1800
jrf	2.29	2.39	226.80	240.92
jrf -P 10	2.27	2.38	31.41	31.69

3.3x

3.6x

21x

> 50x

2.6GB/s

all units in seconds

jrf - バイブコーディング

- 99.9%のコードはCodexとClaude Codeが記述
  - AIは、性能を低下させるような変更を入れようとしがち
  - 人間による設計レビューで構造の健全性を確保

jrf - バイブコーディング

- 99.9%のコードはCodexとClaude Codeが記述
  - AIは、性能を低下させるような変更を入れようとしがち
  - 人間による設計レビューで構造の健全性を確保
- 生産性と正確性が、ともに向上 ：
  - AIが処理系 (jrf) とテストスイートを実装
    - AIの馬力で、十分なカバレッジを確保

jrf - バイブコーディング

- 99.9%のコードはCodexとClaude Codeが記述
  - AIは、性能を低下させるような変更を入れようとしがち
  - 人間による設計レビューで構造の健全性を確保
- 生産性と正確性が、ともに向上 ：
  - AIが処理系 (jrf) とテストスイートを実装
    - AIの馬力で、十分なカバレッジを確保
  - クエリは人間とAIが記述
    - クエリDSLは宣言的なので分かりやすく、バグが入りにくい

jrf - 複雑なクエリの書き方

- 複雑なクエリは、チートシートを睨んで考える　...

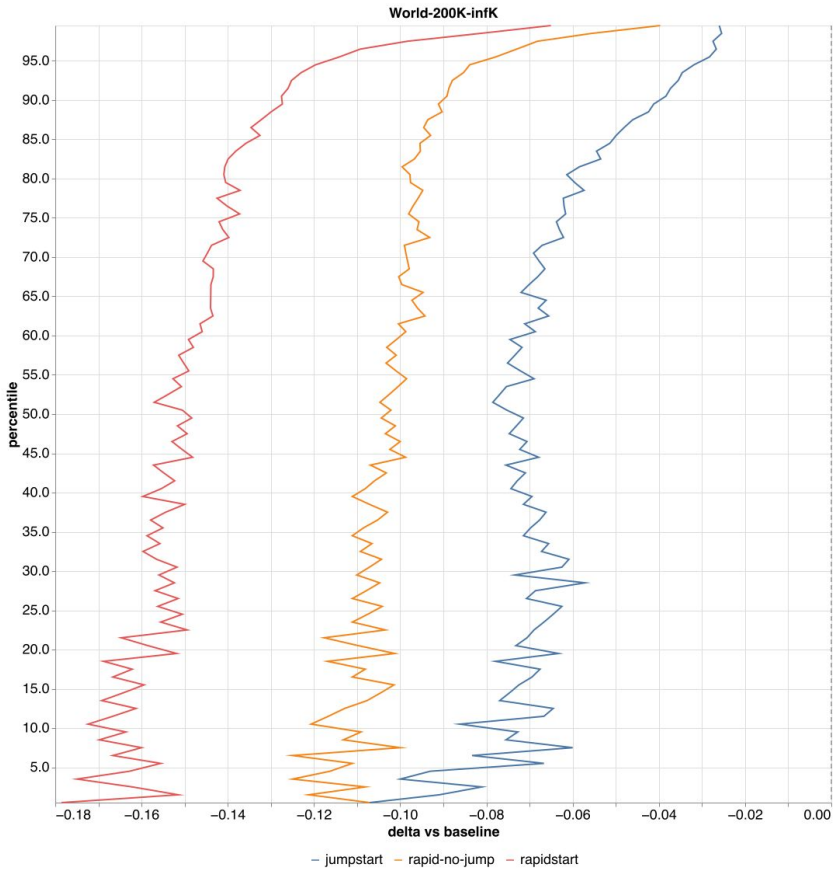
jrf - 複雑なクエリの書き方

- 複雑なクエリは、チートシートを睨んで考える　... のではなくて、AIに相談
- AIはjrfの実装を読み（rubyだから短い！）、実際に動作検証したクエリを教えてくれる
- 教えてくれるクエリは、もちろん宣言的な DSLだから理解しやすい

Rapid Start - 実環境での評価

TTLB削減: 全世界

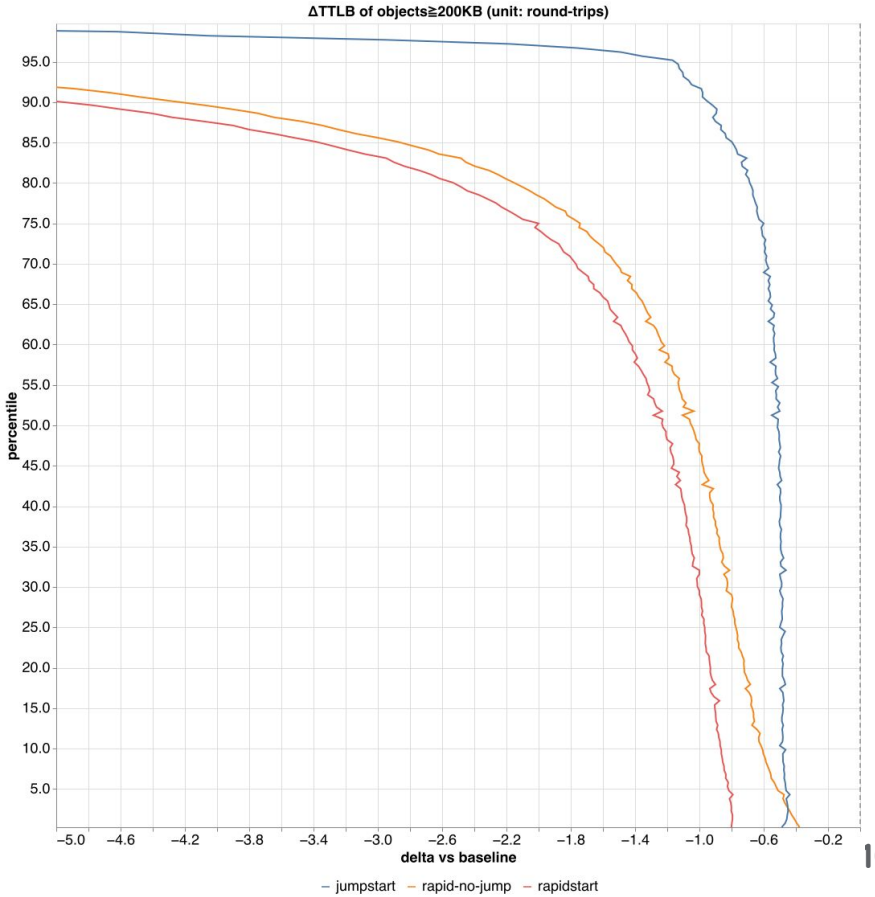
- 全拠点集計
- レスポンス ≥ 200KB
- Rapid StartによりTTLBを平均14.7%削減



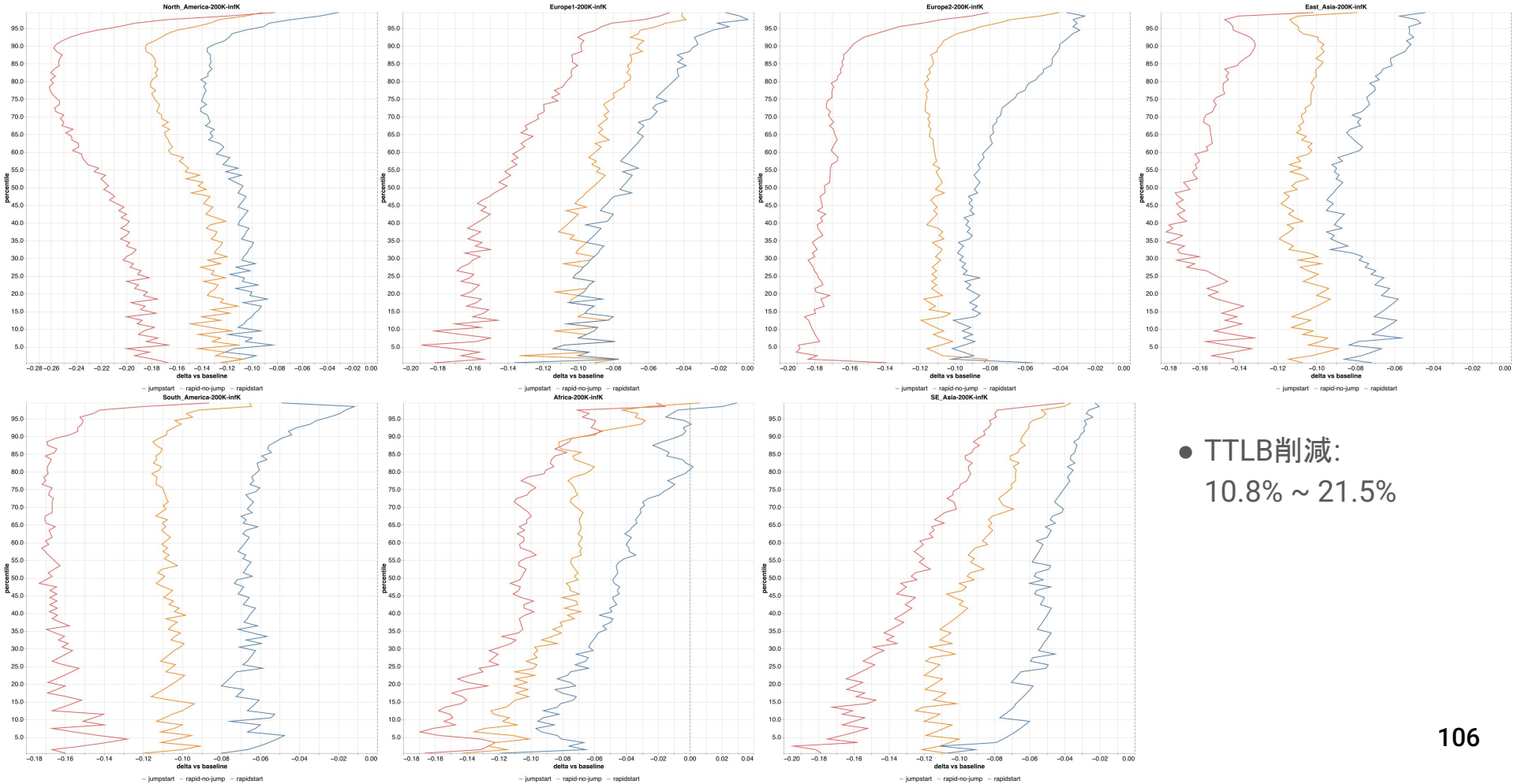
注: 鋸歯状なのはミリ秒単位の測定結果を用いているため

TTLB削減: 全世界 (RTT換算)

- 約70%の接続で、1 RTT以上のTTLB削減

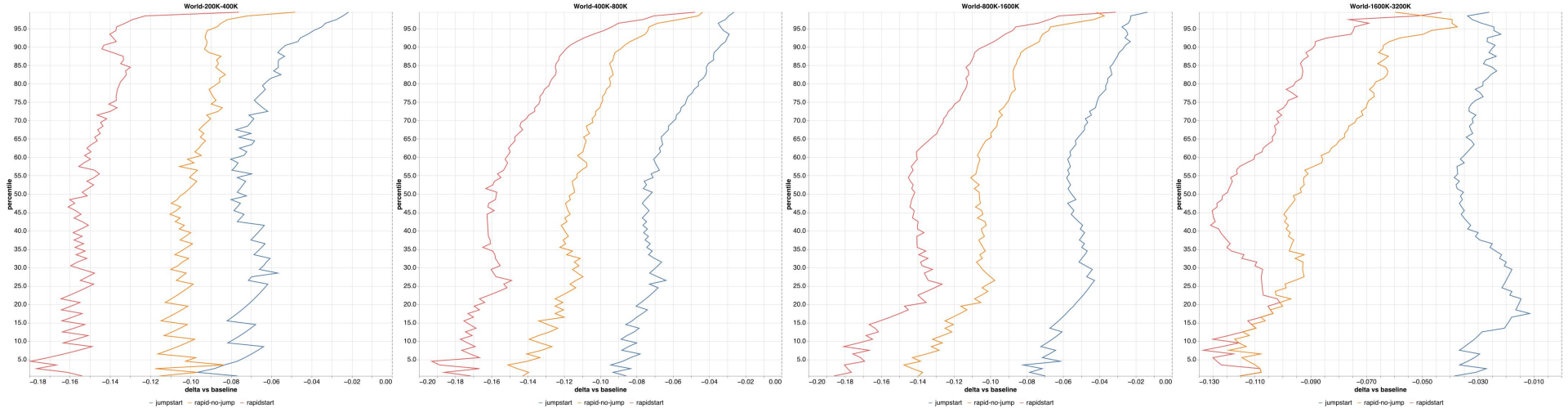


TTLB削減: 拠点毎



- TTLB削減:
  10.8% ~ 21.5%

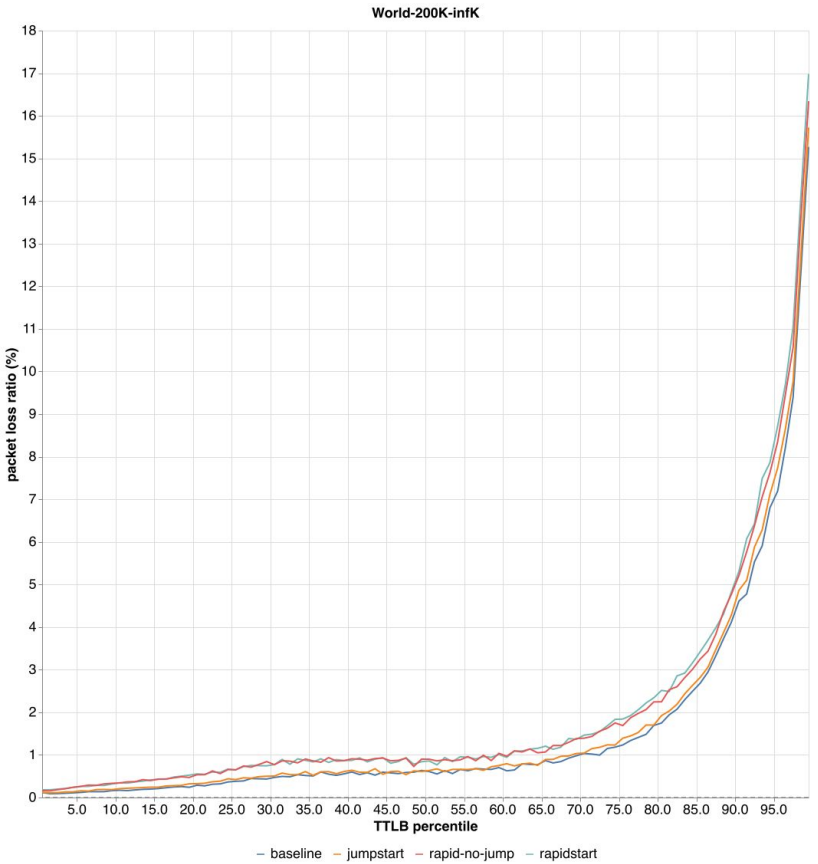
TTLB削減: レスポンスサイズ別



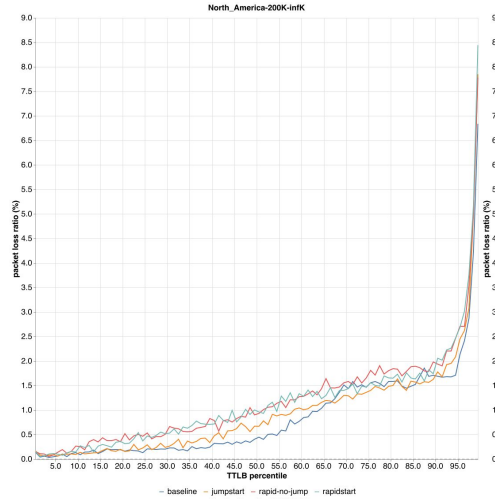
- サイズで分割して全世界集計:
  200KB - 400KB / 400KB - 800KB / 800KB - 1.6MB / 1.6MB - 3.2MB
- TTLB削減: 10.6% (1.6MB - 3.2MB) ~ 14.9% (200KB - 400KB)

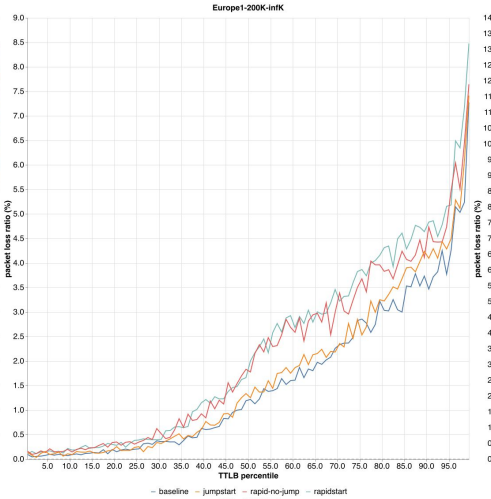
パケロス率 : 全世界

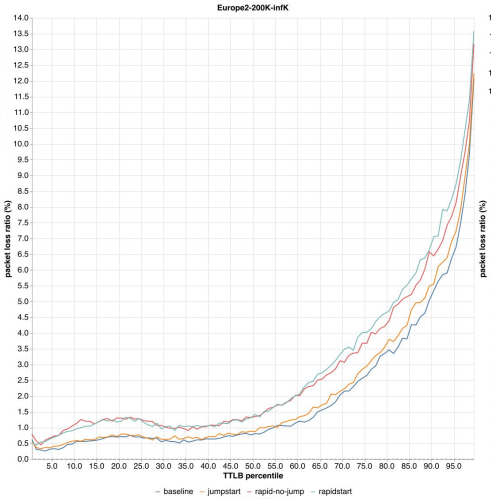
	slow start (baseline)	jumpstart	rapid-no-jump	rapidstart
avg.	1.52%	1.61%	1.92%	1.98%
P50	0.62%	0.62%	0.90%	0.85%
P90	4.36%	4.57%	4.99%	5.06%
P99	13.80%	14.22%	14.97%	15.55%

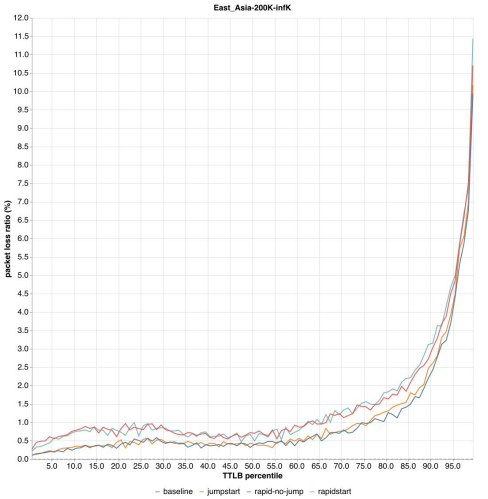


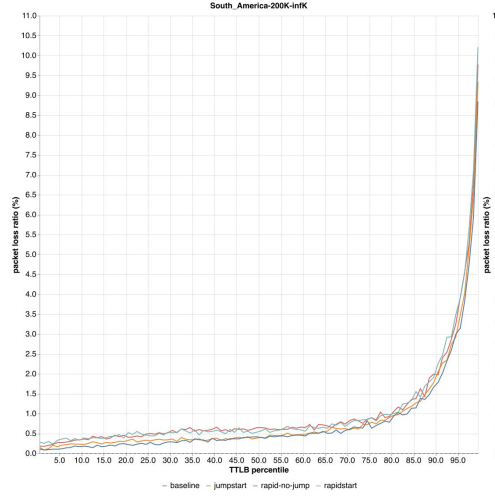
パケロス率 : 拠点毎

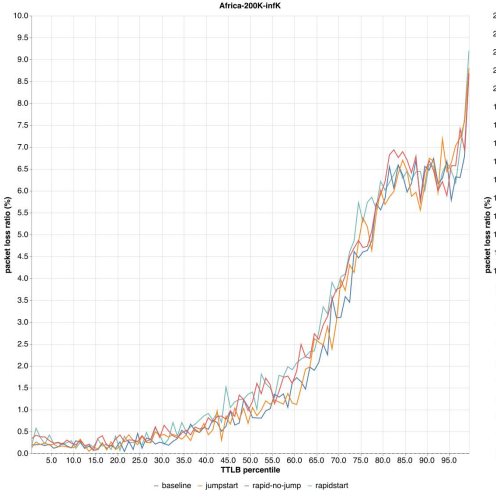


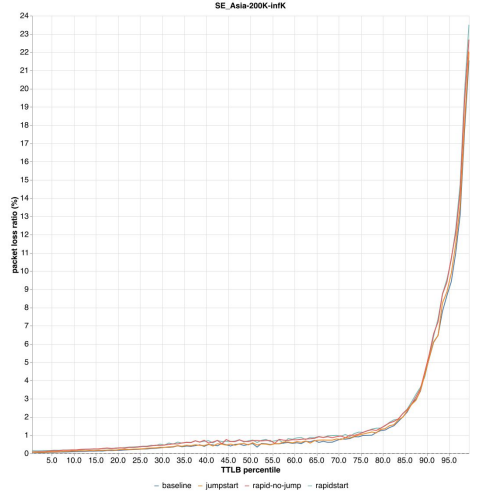








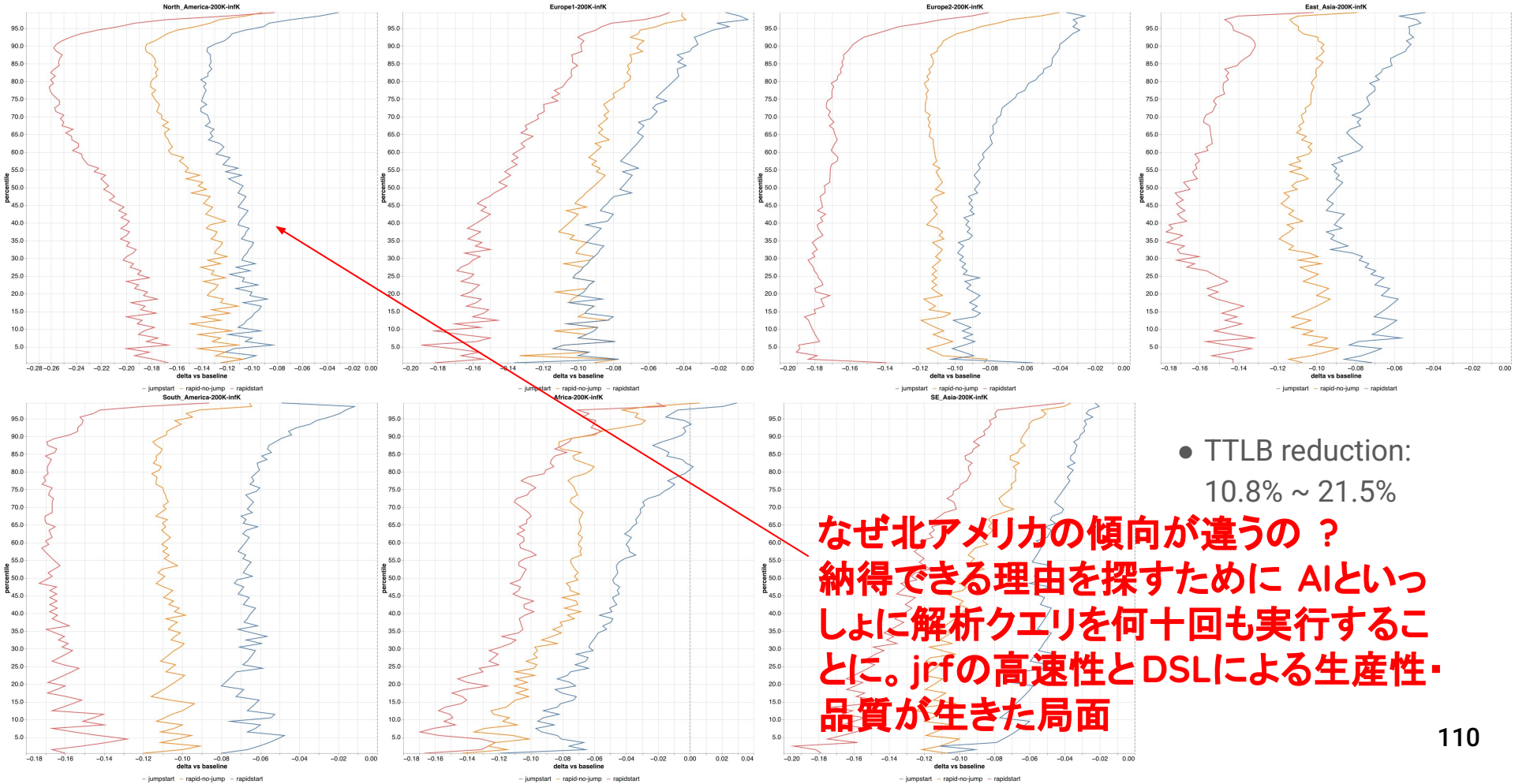




P99でのパケロスが最高の拠点におけるパケロス:

- slow start: 19.60%
- jumpstart: 20.05%
- rapid-no-jump: 20.99%
- rapid start: 21.65%

TTLB削減: 拠点毎



- TTLB reduction: 10.8% ~ 21.5%

なぜ北アメリカの傾向が違うの ？
納得できる理由を探すために AIといっしょに解析クエリを何十回も実行することに。jrfの高速性とDSLによる生産性・品質が生きた局面

まとめ

jrf - 高速なログ解析ツール

- ログ解析には、直感的なクエリ言語を実行する高速な処理系が望ましい：
  - さまざまなクエリを簡単に実行
  - 前処理が不要
    - つまり、ログデータベース等のインフラが不要

jrf - 高速なログ解析ツール

- ログ解析には、直感的なクエリ言語を実行する高速な処理系が望ましい：
  - さまざまなクエリを簡単に実行
  - 前処理が不要
    - つまり、ログデータベース等のインフラが不要
- jrfは、この条件をみたす NDJSONクエリプログラム
    - SQLライクでrubyで拡張可能な DSL
    - jqより20倍高速
      - 10コアCPUで2.6GB/秒

AIによるDSL処理系開発

- 従来: 処理系開発は大変だから ...
  - 既存の処理系にあわせたプログラムを記述
    - jq - JSON用DSLなのにログ解析だと遅くて複雑
    - ruby直書き - 速いが複雑になりすぎる
  - 皆と同じ言語じゃないと、困った時に調べられない

AIによるDSL処理系開発

- 従来: 処理系開発は大変だから ...
  - 既存の処理系にあわせたプログラムを記述
    - jq - JSON用DSLなのにログ解析だと遅くて複雑
    - ruby直書き - 速いが複雑になりすぎる
  - 皆と同じ言語じゃないと、困った時に調べられない
- AI時代: 処理系開発はAIの得意な作業
  - テストスイートもガンガン書かせて品質担保が容易
  - DSLの書き方も、困ったらAIに聞けばいい
  - ワークロードにあわせた独自 DSL処理系をAIに実装させることで、人間の生産性が向上

DSL処理系実装言語としての Ruby

- RubyはDSL処理系を記述するのにむいた言語 :
  - DSLと親和性が高い柔軟な構文
  - 高速なインタプリタ基盤
    - DSLをevalしてまとめて実行
      - さらには JITも

DSL処理系実装言語としての Ruby

- RubyはDSL処理系を記述するのにむいた言語 :
  - DSLと親和性が高い柔軟な構文
  - 高速なインタプリタ基盤
    - DSLをevalしてまとめて実行
      - さらには JITも
- JSONパーサなどライブラリも、インタプリタ基盤にあわせて最適化されている

DSL処理系実装言語としての Ruby

- RubyはDSL処理系を記述するのにむいた言語 :
  - DSLと親和性が高い柔軟な構文
  - 高速なインタプリタ基盤
    - DSLをevalしてまとめて実行
      - さらには JITも
- JSONパーサなどライブラリも、インタプリタ基盤にあわせて最適化されている
- コンパイル言語で自前の DSL処理系を実装したもの (jq) より、はるかに効率的

Rapid Start

- TLS/1.3 と QUIC は TTFB を、それぞれ 1 RTT 削減

Rapid Start

- TLS/1.3 と QUIC は TTFB を、それぞれ 1 RTT 削減
- 次の最適化ターゲットは TTLB:

Rapid Start

- TLS/1.3 と QUIC は TTFB を、それぞれ 1 RTT 削減
- 次の最適化ターゲットは TTLB:
  - Rapid Start は TTLB[注] を
    - 14.7% 削減
    - RTT 換算で 70% が 1 RTT 以上の効果

注: 200KB以上のレスポンスを送信する場合の世界平均

Rapid Start

- TLS/1.3 と QUIC は TTFB を、それぞれ 1 RTT 削減
- 次の最適化ターゲットは TTLB:
  - Rapid Start は TTLB[注] を
    - 14.7% 削減
    - RTT 換算で 70% が 1 RTT 以上の効果
  - Ruby + AI による DSL (jrf) がそのために必要だった

注: 200KB以上のレスポンスを送信する場合の世界平均

Rapid Start

- TLS/1.3 と QUIC は TTFB を、それぞれ 1 RTT 削減
- 次の最適化ターゲットは TTLB:
  - Rapid Start は TTLB[注] を
    - 14.7% 削減
    - RTT 換算で 70% が 1 RTT 以上の効果
  - Ruby + AI による DSL (jrf) がそのために必要だった

Ruby + AI is making the Web faster!

注: 200KB以上のレスポンスを送信する場合の世界平均