BITS Pilani
Pilani Campus

Advance Computer Networks (CS G525)

Virendra S Shekhawat
Department of Computer Science and Information Systems

innovate achieve lead

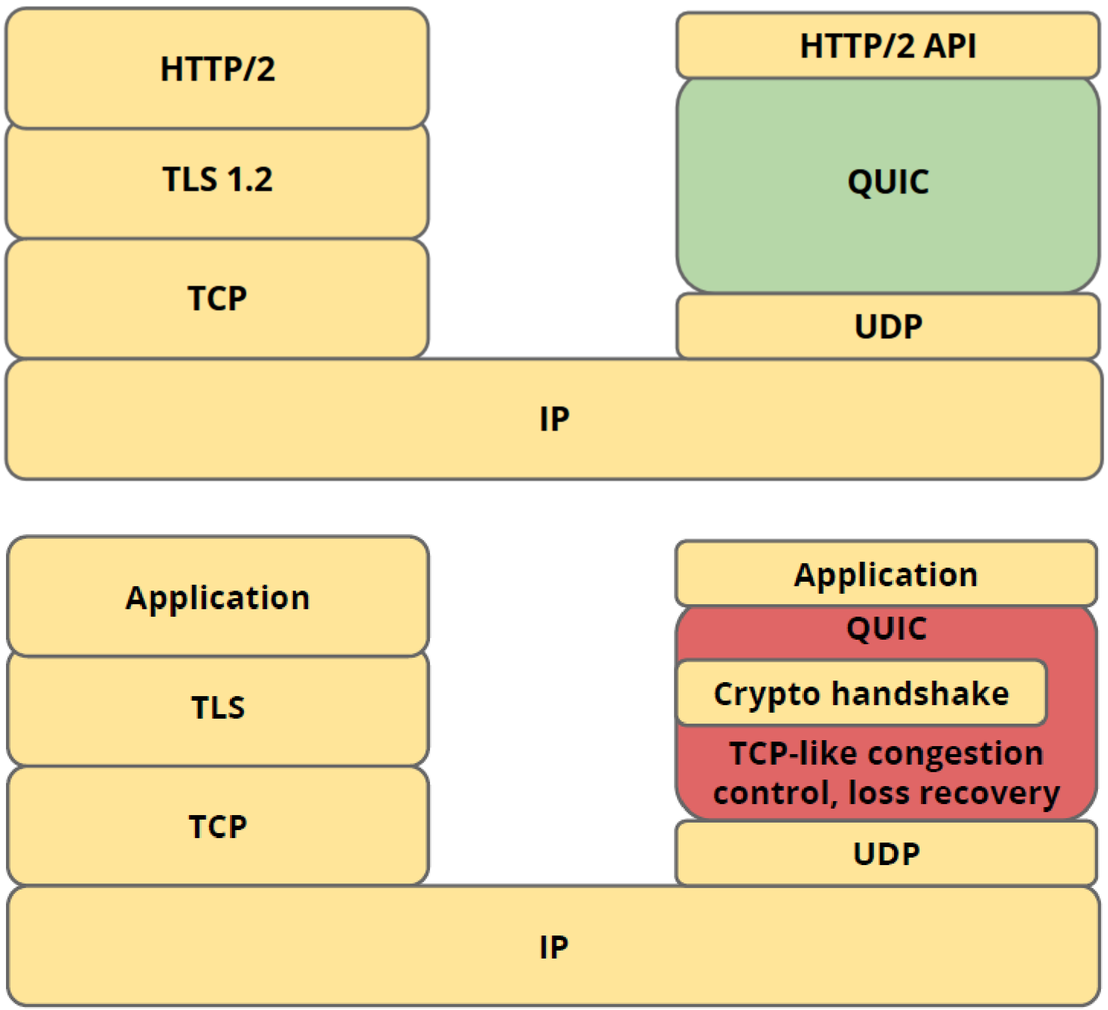
BITS Pilani
Pilani Campus

First Semester 2018-2019
Slide_Deck_M3_2

QUIC

- **Q**uick **U**DP **I**nternet **C**onnections
  - A new transport protocol for the internet, developed by Google.
  - Similar to TCP+TLS+HTTP2, but implemented on top of UDP
  - Faster connection establishment than TLS/TCP
  - Deals better with packet loss than TCP
  - Has Stream-level and Connection-level Flow Control
- Reference
  - The QUIC Transport Protocol [Adam 2017]

Where does it fit?



HTTP/2 vs HTTP/1

HTTP/2	HTTP/1
Pipelining - Parallel requests made asynchronously but server responses synchronously	**Multiplexing -** Multiple asynchronous HTTP requests over single TCP connection
NA	**Server Push** – Multiple responses for a single HTTP request
Handles multiple requests (multiplexed streams) over a single TCP connection	One outstanding request per TCP connection
Allows the server to send additional cacheable information (responses) to the client	One response per HTTP request

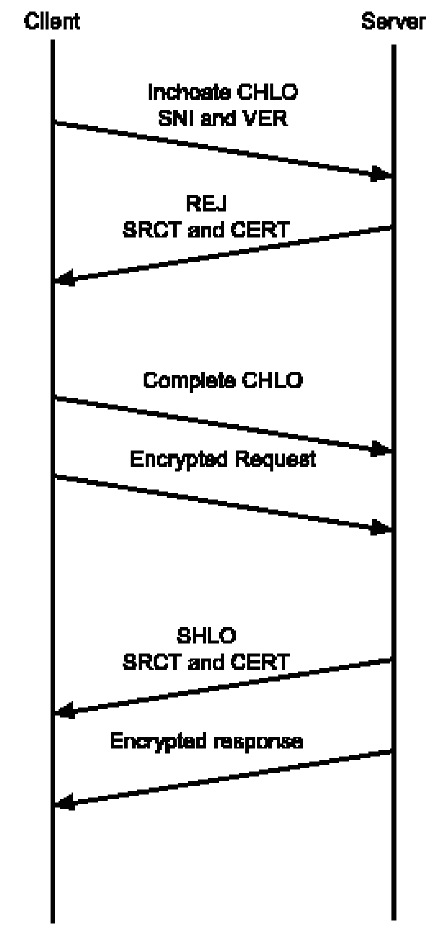
*Bi-directional sequence of text format frames sent over the HTTP/2 protocol exchanged between the server and client are known as “streams”.

QUIC Design Rationales

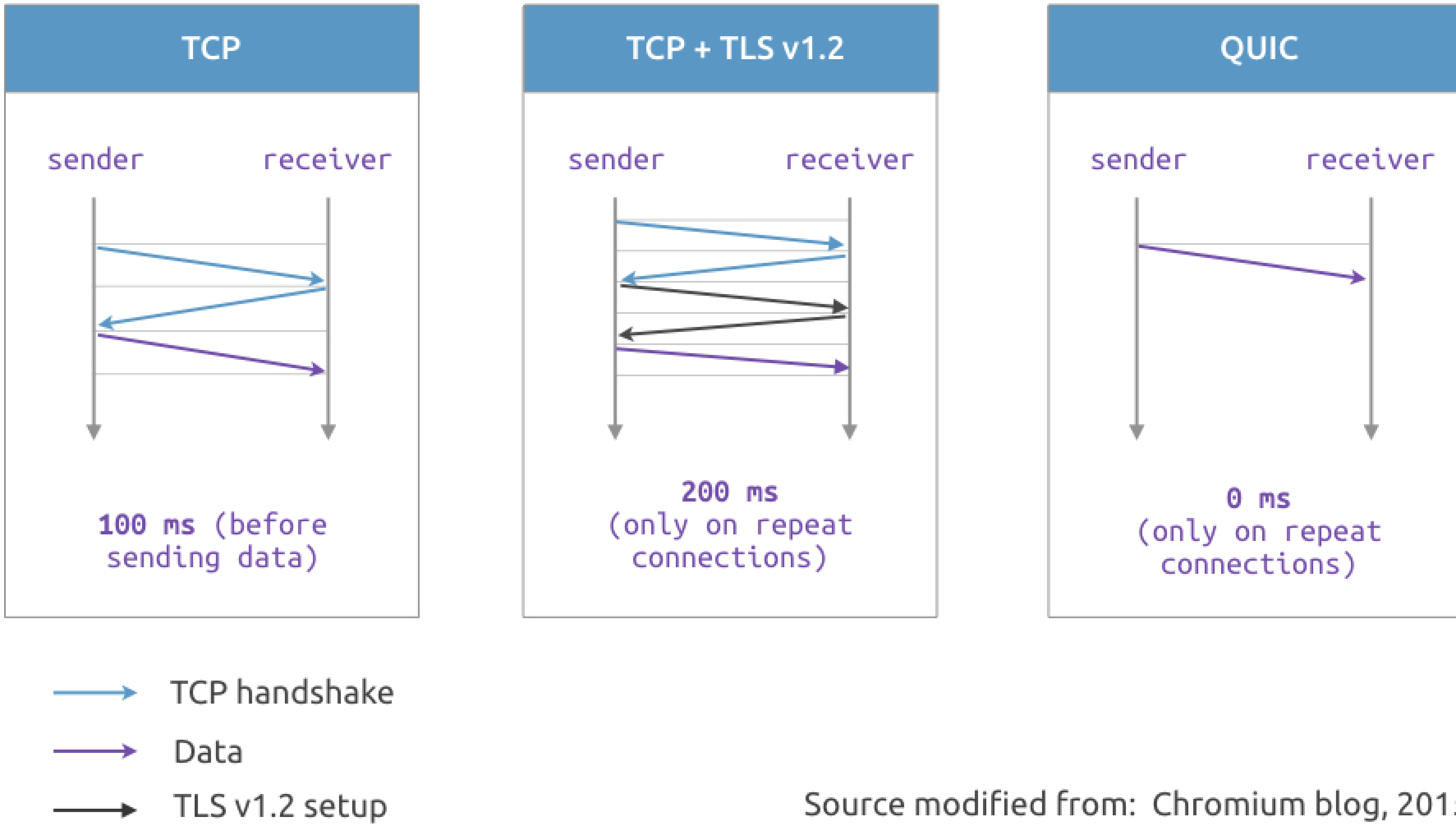
- Deployability and evolvability
- Low latency connection establishment
- Multi-streaming
- Better loss recovery and flexible congestion control
- Resilience to NAT-rebinding (Connection IDs vs. 4-tuple)
- Multipath for resilience and load sharing

First Ever Connection -1 RTT

- **First CHLO is inchoate (empty)**
  - Simply includes version and server name
- **Server responds with REJ**
  - Includes server config, certificates, etc.
  - Allows client to make forward progress
- **Second CHLO is complete**
  - Followed by initially encrypted request data
- **Server responds with SHLO**
  - Followed immediately by forward-secure encrypted response data



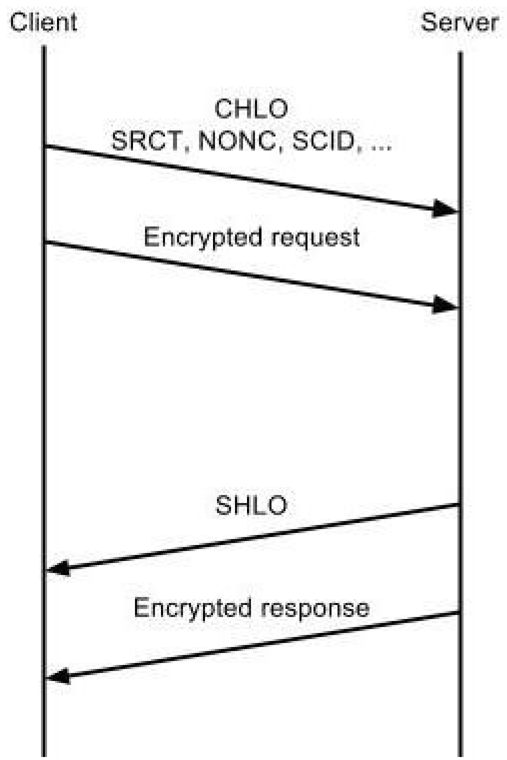
Connection Establishment



Source modified from: Chromium blog, 2015

Subsequent Connections

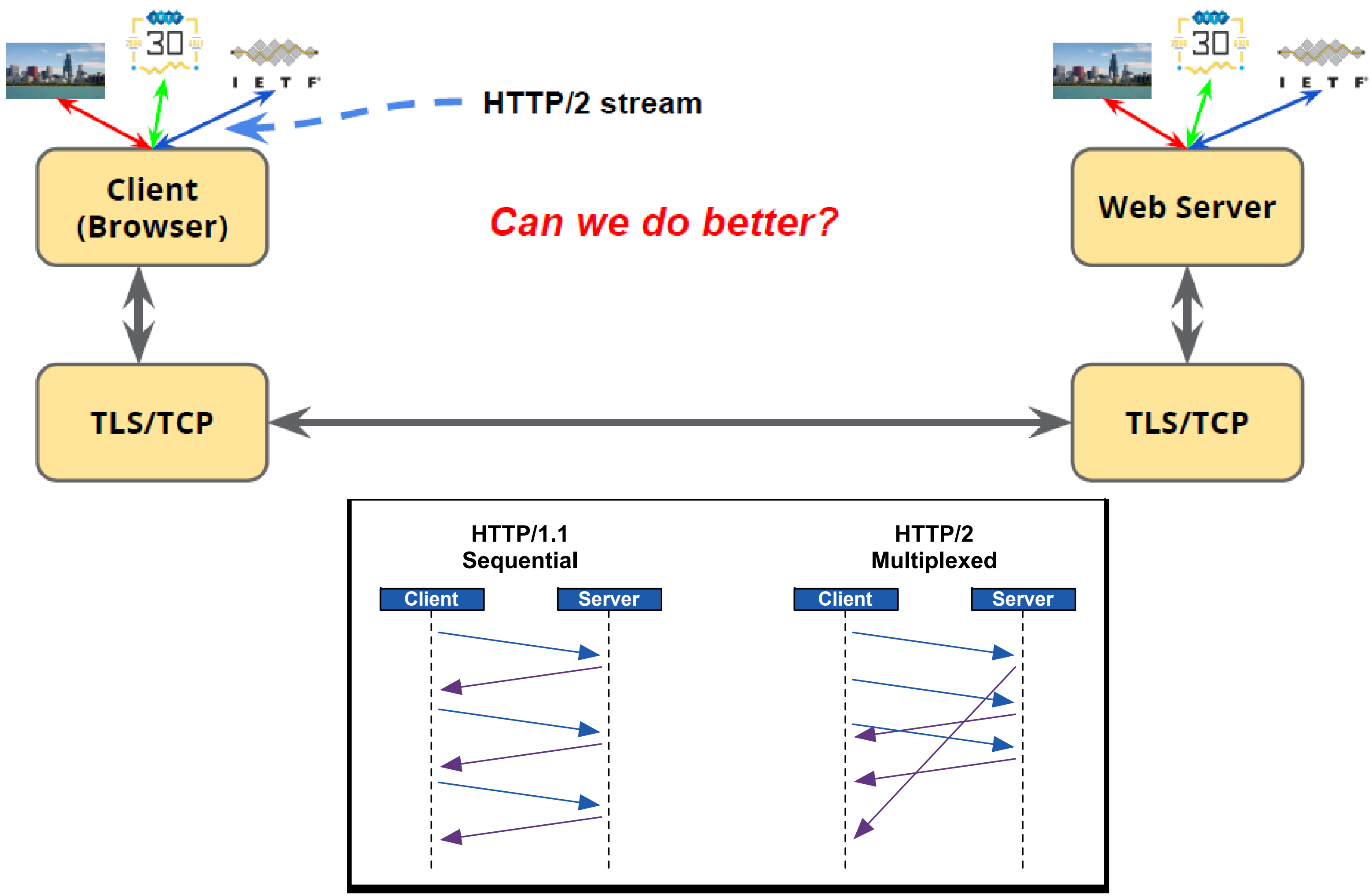
- **First CHLO is complete**
  – Based on information from previous connection
  – Followed by initially encrypted data.
- **Server responds with SHLO**
  – Followed immediately by forward-secure encrypted data



Congestion Control and Reliability

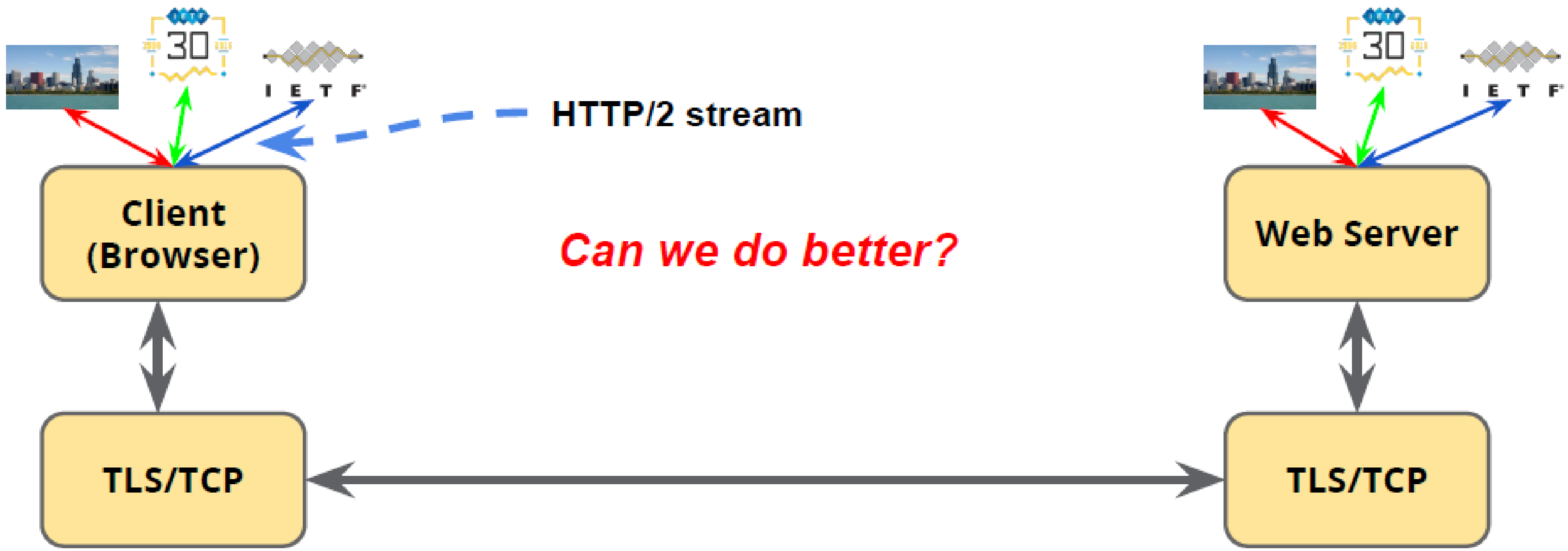
- **QUIC builds on decades of experience with TCP**
  - QUIC has pluggable congestion control
- **Retransmitted packets consume new sequence number**
  - No retransmission ambiguity
  - Prevents loss of retransmission from causing RTO
- **More verbose ACK**
  - TCP supports up to 3 SACK ranges
  - QUIC supports up to 256 NACK ranges
  - Per-packet receive times, even with delayed ACKs
- **ACK packets consume a sequence number**

Dealing with Head of Line (HoL) with HTTP/2

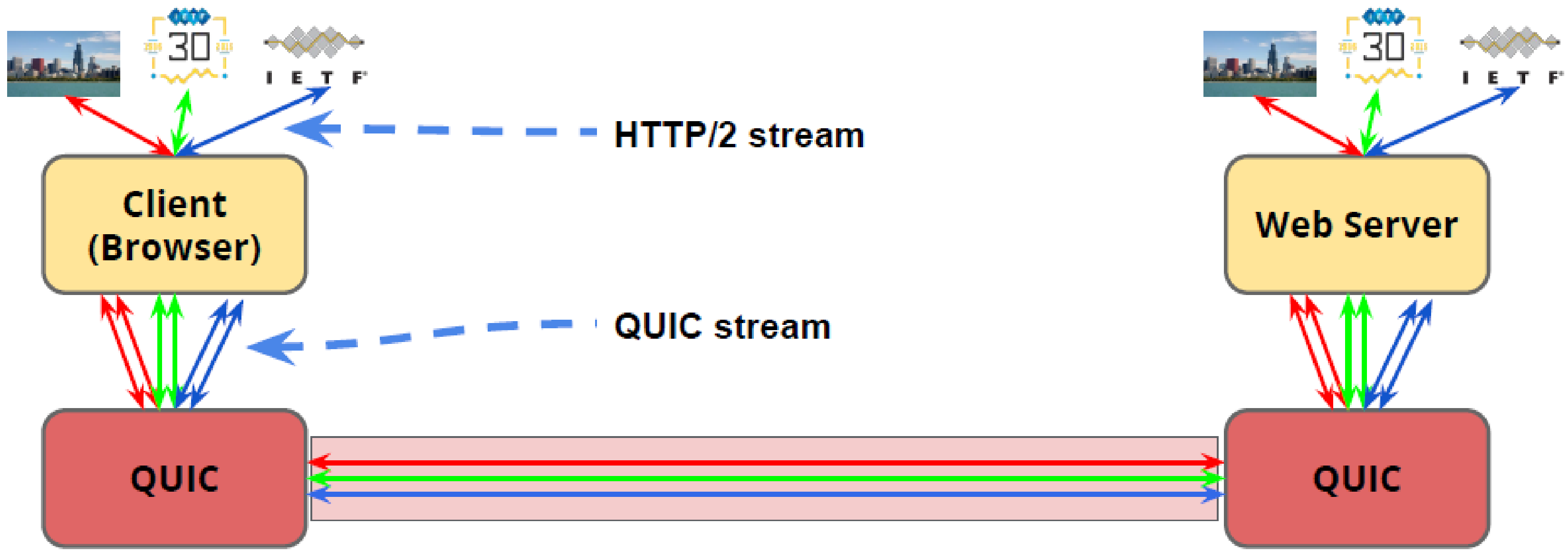


Dealing with Head of Line (HoL) with HTTP/1.1

Dealing with Head of Line (HoL) with HTTP/2



HTTP over QUIC



Structure of QUIC Packet

```
+------------+---------------------------------+
|  Flags (8) |  Connection ID (64) (optional)  | ->
+------------+---------------------------------+

+------------------------------------+--------------------------------+
|  Version (32) (client-only, optional) |  Diversification Nonce (256)  | ->
+------------------------------------+--------------------------------+

+---------------------------+
|  Packet Number (8 - 48)   | ->
+---------------------------+

+------------+------------+     +------------+
|   Frame 1  |  Frame 2   | ... |  Frame N   |
+------------+------------+     +------------+

        Stream frame
       +------------+----------------------+------------------+
       |  Type (8)  |  Stream ID (8 - 32)  |  Offset (0 - 64) |
       +------------+----------------------+------------------+
       +-------------------------+--------------------------------+
       |  Data length (0 or 16)  |  Stream Data (data length)     |
       +-------------------------+--------------------------------+
```

QUIC Support

- Client
    - Chrome enable by default
    - Wireshark support
- Library
    – libquic / goquic
    – proto-quic
        - First release 4/1
        - Supported by Google

Further References

- IETF draft
  - http://tools.ietf.org/html/draft-tsvwg-quic-protocol-01
- www.chromium.org/quic

TCP Congestion Control

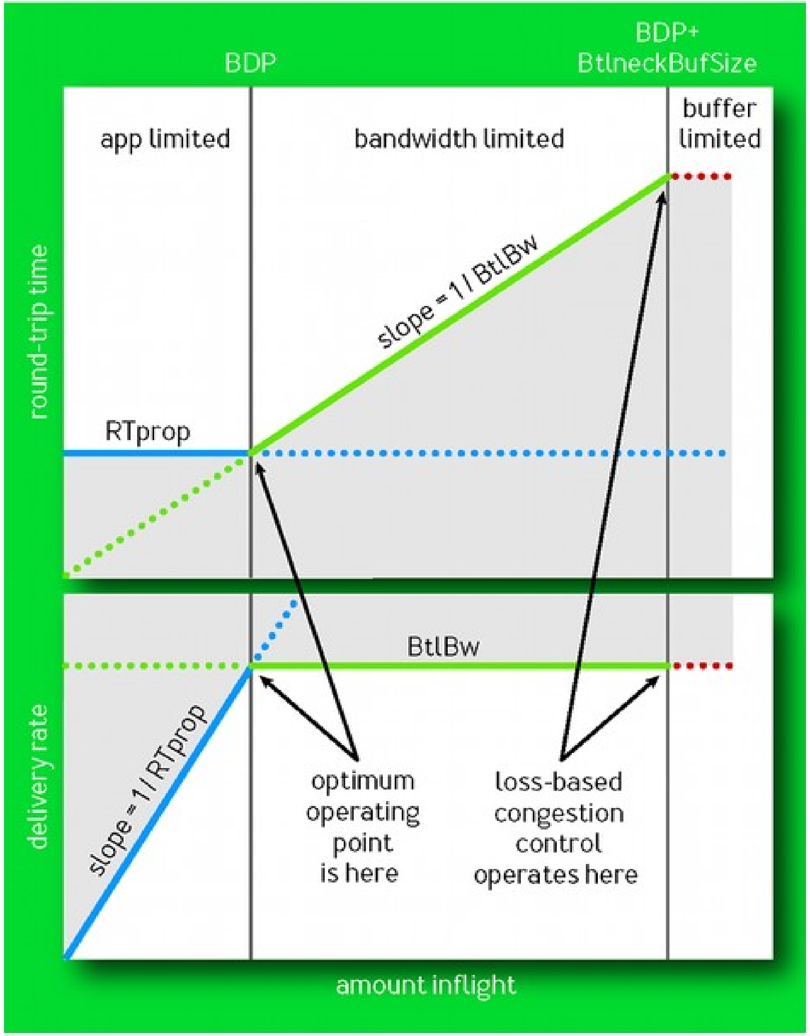
- TCP uses **loss-based congestion control** strategy
  - Poor performance with high BW links and large buffer sizes
  - Large buffers leads to long RTTs and delayed congestion notification

BBR Congestion Control

- BBR provides a "**queueless**" congestion control
- A flow should ideally have data in-flight equal to Bandwidth Delay Product (BDP)
  - *BDP = RTprop x BtlBw*
  - At this point, a connection completely saturates the bottleneck link while maintaining an empty buffer
- What is congestion…?
  - In-flight data more than BDP (for a long duration) considered as congestion
- Reference
  - Congestion Based Congestion Control- BBR [N Cardwell 2016]

BBR

- Essential Path characteristics for congestion control
  - *RTprop* and *BtlBw*
- ***RTprop*** **and** ***BtlBw*** cannot be measured simultaneously. Why?
  - Measuring *RTprop* requires operating to the left of BDP while measuring *BtlBw* requires operation to the right.
- Both parameters are independent…???



Characterizing the Bottleneck

- A connection runs with the highest throughput and lowest delay when
  a) Bottleneck packet arrival rate equals BtlBw (**rate balance**)
  b) Total data in-flight equals to BDP = *RTprop x BtlBw (**full pipe**)*
- It gives 100% utilization with no overflow
- Both conditions should meet simultaneously to ensure "**no queue**"

BtlBw Estimation [.1]

- Average delivery rate between send and ack
  - *deliveryRate = Δdelivered/Δt*
- This rate must be ≤ the bottleneck rate
  - Arrival amount is known exactly so all the uncertainty associated with Δt (Δt ≥ true arrival interval)
  - Therefore, the ratio must be ≤ the true delivery rate and upper-bounded by the bottleneck capacity
- BtlBw is given as Windowed-max of delivery rate

$$\widehat{BtlBw} = \max(deliveryRate_t) \quad \forall t \in [T - W_B, T]$$

  - where the time window W_B is typically six to ten RTTs.

BtlBw Estimation [..2]

- TCP must record the departure time of each packet to compute RTT
- BBR augments that record with the total data delivered so each ack arrival yields both an RTT and a delivery rate measurement

- Objective of BBR
  - Matching the packet flow to the delivery path

BBR Algorithm: When ack is received

- Two parts
  - When ack is received
  - When data is sent

```
function onAck(packet)
  rtt = now - packet.sendtime
  update_min_filter(RTpropFilter, rtt)
  delivered += packet.size
  delivered_time = now
  deliveryRate = (delivered - packet.delivered)
                 /(now - packet.delivered_time)
  if (deliveryRate > BtlBwFilter.currentMax
      || ! packet.app_limited)
      update_max_filter(BtlBwFilter,
                        deliveryRate)
  if (app_limited_until > 0)
      app_limited_until - = packet.size
```

BBR Algorithm: When data is sent

```
function send(packet)
    bdp = BtlBwFilter.currentMax
          * RTpropFilter.currentMin
    if (inflight >= cwnd_gain * bdp)
        // wait for ack or timeout
        return
    if (now >= nextSendTime)
        packet = nextPacketToSend()
        if (! packet)
            app_limited_until = inflight
            return
        packet.app_limited =
                (app_limited_until > 0)
        packet.sendtime = now
        packet.delivered = delivered
        packet.delivered_time = delivered_time
        ship(packet)
        nextSendTime = now + packet.size /
                (pacing_gain *
                 BtlBwFilter.currentMax)
    timerCallbackAt(send, nextSendTime)
```

Next Topic…

- Congestion Control at Routers-Queuing Algorithms
  - Fair Queuing (FQ)
  - Nagle's FQ Algorithm
  - Max-Min Fairness
  - Weighted Fair Queuing (WFQ)
  - Other Queuing Algorithms (FIFO, CSFQ, RED)
- Reading
  - Random Early Detection Gateways for Congestion Avoidance by Sally Floyd 1993

Thank You!