MindWorks UK

Technology Transfer Partners

http://

HTTP/3
a plain language description

(many words, best read quietly)

Joe Doupnik
Prof (ret.) Univ of Oxford
jrd@netlab1.net
jdoupnik@microfocus.com
MindworksUK and Micro Focus

MICRO FOCUS

Rational for a new version of HTTP

- “Go fast, and push”, with capital F and P
- Allow client IP numbers to vary during a session, to accommodate mobile users
- Speed up TLS negotiations (and require TLS)
- Refine nuances about network congestion handling

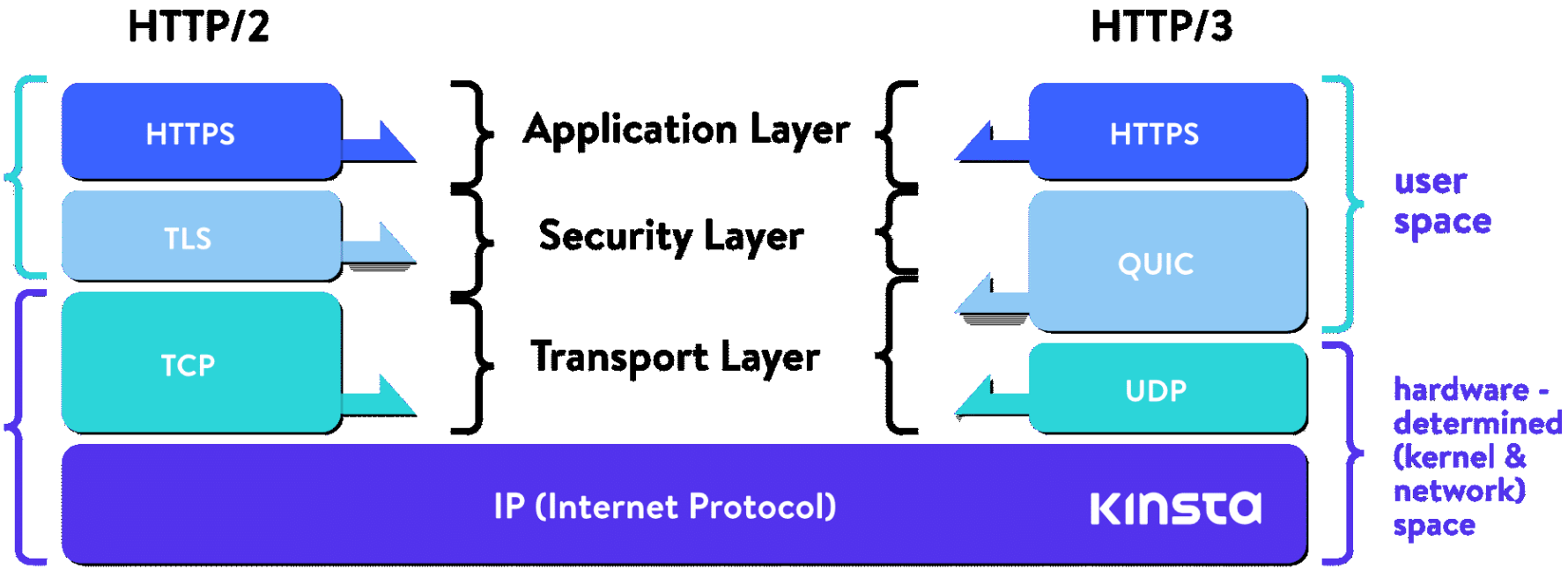
Technical details underlaying these matters are complicated, but some aspects need to be understood by us in the field

What is evident to us are a) web pages are becoming more complex and filled with goodness knows what, and b) vendors wish to push more “stuff” at us

Google’s HTTP/3-QUIC summary doc is:

https://docs.google.com/presentation/d/15e1bLKYeN56GL1oTJSF9OZiUsI-rcxisLo9dEyDkWQs/edit#slide=id.g99041b54d_0_0

HTTP/2 & HTTP/3 big picture



HTTP/2 introduces multiplexed data streams over TCP

HTTP/3+QUIC uses UDP, absorbs multiplexing plus TLS v1.3, may employ a new congestion control algorithm (BBR)

Speed: what is the problem?

- Time to load a web page is mainly controlled by three factors
  - Propagation delay (measured as round trip time RTT)
  - Number and size of individual fetches required
  - Network overload (network congestion, meaning lost packets)
- Propagation delay is often dealt with by employing Content Delivery Networks (CDNs) located regionally.
- Fetch count is up to authors. Restraint and education are needed. This is a major factor of load time. HTTP/2 and /3 assist with multiplexing and server pushing.
- Network overload stalls delivery and requires clever heuristics to recover, and at the same time not be a bad net neighbour.
- TCP today uses the Cubic congestion control algorithm.
- HTTP/3 uses QUIC which emulates parts of TCP mechanisms. It is considering use of BBR (**B**ottleneck **B**andwidth and **R**ound-trip propagation time) congestion control algorithm.

HTTP/3 major changes

- *Multiplexing*, adapted and revised from HTTP/2, to reduce waiting time of sequentially queued request-response cycles.
- Change from TCP to *UDP* in an attempt to circumvent/bypass pauses from “head of line” delays. This operates in conjunction with multiplexing to be semi-independent parallel connections.
- Considering a *new congestion control algorithm*, BBR, to refine exploitation of network capacity. Implementation details in QUIC are sketchy at best, so don’t depend upon this.
- *Replicate* some needed and proven TCP congestion heuristics to avoid Internet collapse, but do the work per multiplexed stream in user-space rather than in the kernel’s TCP/IP stack.

See https://tools.ietf.org/html/draft-tsvwg-quic-loss-recovery-01 for the current TCP-like mechanisms.

Background information: TCP details

A review about how the Internet works today

- We note that speed is set by *delays* from both propagation and processing, plus dealing with *packet loss* due to overfilled buffers (traffic congestion) in relaying routers.
- Pacing transmission (be polite) to avoid such losses is a requirement of TCP. It requires continuous learning of what works or not. Thus send some data, acquire returned ACKs, adjust the clever plan, repeat.

- HTTP/3 replaces TCP with UDP (send & forget) but the packet loss and learning problem persists.

Congestion: TCP's throttle (UDP has none)

Basic rules lead to three limits on TCP about how many bytes are allowed to be in-flight (sent but not yet acknowledged) at any moment:

- The amount available to send
- The receiver's available free space
- Estimated network capacity, as the *congestion window* **cwin**

Network capacity must be observed by the sender, and heuristics are employed to constantly calculate **cwin**. Its is basically controlled by packet loss, thus **cwin** is a loss based estimator.

Note that packet loss is almost all in routers (buffer limitations).

The receiver may briefly delay an ACK to cover multiple arrivals.

The receiver often delays while processing input or fetching yet another javascript file from somewhere.

HTTP/3 wants to use waiting time to send more data.

HTTP/1.0 & 1.1 sequencing

In original HTTP/1.0 each web request opens a new connection and closes it after the response. Requests form an orderly queue. The frequent open/close part is expensive and slow.

HTTP/1.1 introduced header ***keep-alive*** to reduce the open/close frequency by letting successive requests reuse the same connection, yet request/responses still form an orderly queue.

If a response were slow to arrive then the queue may wait and wait for it, twiddling its collective thumbs. ***Head of line blocking***.

Apache main body (SLES top of /etc/apache2/default-server.conf)

```
KeepAlive on                 (default is on)
KeepAliveTimeout 180         (default is 5 seconds)
MaxKeepAliveRequests 50      (default is 100)
```

Multiplexing in HTTP/2

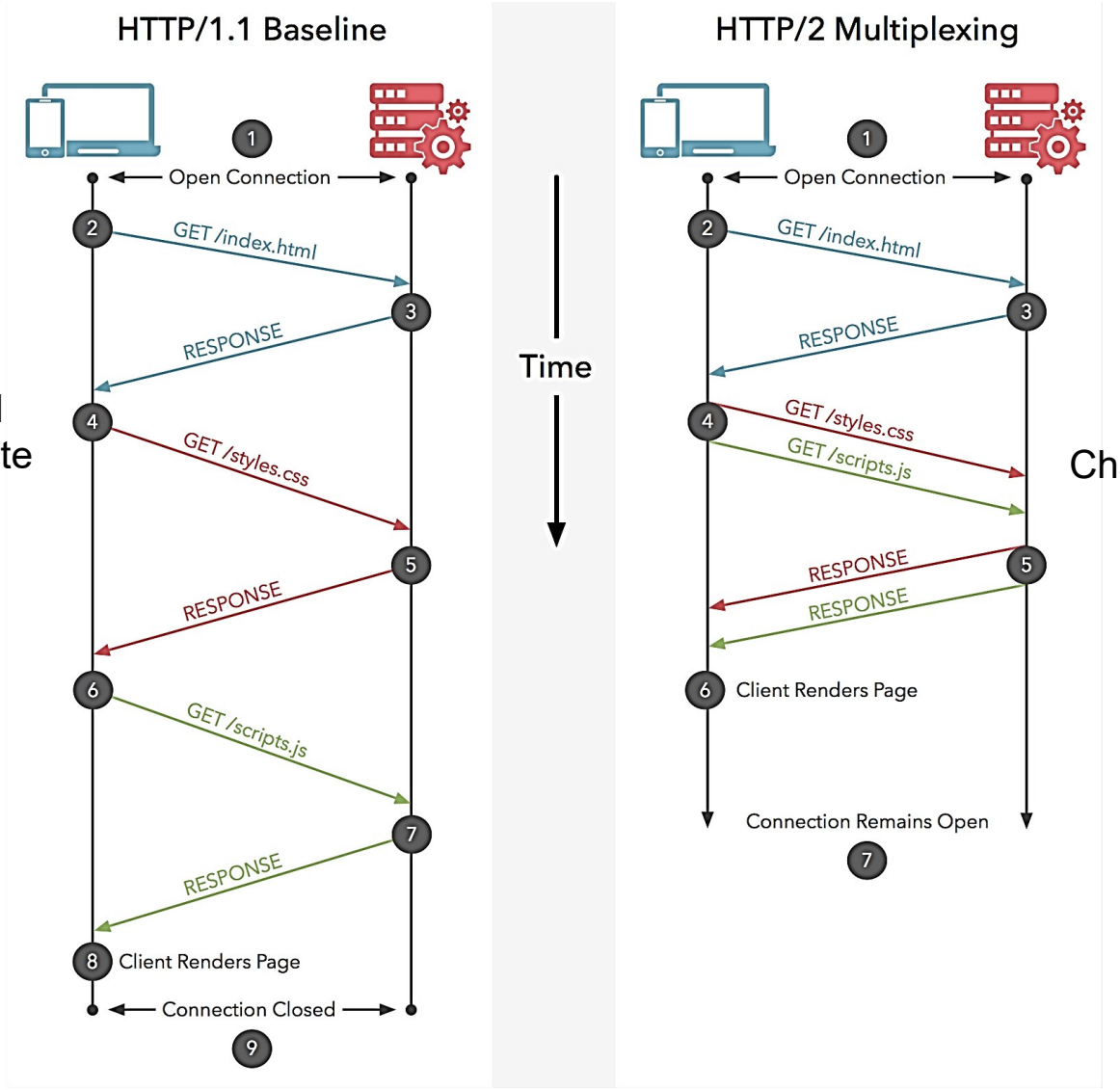
HTTP/2 introduces a multiplexed channel

- Requests can be sent one after another over that single channel without waiting, and responses arrive later, aka: ***client greed***.
- Servers can push data without requests: ***server push.***

- The technique is number each request (and its response). Numbering reunites a response with its request.
- The numbers are called ***streams*** (invisible to applications).
- The queue is nearly a mob, bypassing slow request-response pairs. Applications may need to restore proper order. *H2Push* has many controls to impose some order on server sends.

See “New Protocols HTTP/2 and TLS v1.3” on https://netlab1.net

A pictorial summary of HTTP 1.1 and 2



MindWorks UK

Wireshark: Firefox to Apache (HTTP/2)

No.	Time	Source	Destination	Protocol	Length	Info
34	09:37:13.596277	leap.netlab1.net	desktop.netlab1.net	TCP	66	443 → 1805 [SYN, ACK] Seq=0 Ack=1 Win=2920…
35	09:37:13.596347	desktop.netlab1.net	leap.netlab1.net	TCP	54	1805 → 443 [ACK] Seq=1 Ack=1 Win=65700 Len…
36	09:37:13.601076	desktop.netlab1.net	leap.netlab1.net	TLSv1.2	571	Client Hello
37	09:37:13.601324	leap.netlab1.net	desktop.netlab1.net	TCP	60	443 → 1805 [ACK] Seq=1 Ack=518 Win=30336 L…
38	09:37:13.601995	leap.netlab1.net	desktop.netlab1.net	TLSv1.2	1514	Server Hello
39	09:37:13.602216	leap.netlab1.net	desktop.netlab1.net	TCP	1514	443 → 1805 [ACK] Seq=1461 Ack=518 Win=3033…
40	09:37:13.602217	leap.netlab1.net	desktop.netlab1.net	TLSv1.2	1230	Certificate [TCP segment of a reassembled …
41	09:37:13.602259	desktop.netlab1.net	leap.netlab1.net	TCP	54	1805 → 443 [ACK] Seq=518 Ack=4097 Win=6570…
42	09:37:13.604139	leap.netlab1.net	desktop.netlab1.net	TCP	1514	443 → 1805 [ACK] Seq=4097 Ack=518 Win=3033…
43	09:37:13.604140	leap.netlab1.net	desktop.netlab1.net	TLSv1.2	643	Certificate Status, Server Key Exchange, S…
44	09:37:13.604187	desktop.netlab1.net	leap.netlab1.net	TCP	54	1805 → 443 [ACK] Seq=518 Ack=6146 Win=6570…
45	09:37:13.612414	desktop.netlab1.net	leap.netlab1.net	TLSv1.2	147	Client Key Exchange, Change Cipher Spec, F…
46	09:37:13.612888	leap.netlab1.net	desktop.netlab1.net	TLSv1.2	105	Change Cipher Spec, Finished
47	09:37:13.613061	desktop.netlab1.net	leap.netlab1.net	HTTP2	231	Magic, SETTINGS[0], WINDOW_UPDATE[0], PRIO…
48	09:37:13.613088	leap.netlab1.net	desktop.netlab1.net	HTTP2	111	SETTINGS[0], WINDOW_UPDATE[0]
49	09:37:13.613109	desktop.netlab1.net	leap.netlab1.net	HTTP2	304	HEADERS[15]: GET /, WINDOW_UPDATE[15]
50	09:37:13.613290	leap.netlab1.net	desktop.netlab1.net	TCP	60	443 → 1805 [ACK] Seq=6254 Ack=1038 Win=325…
51	09:37:13.613455	desktop.netlab1.net	leap.netlab1.net	HTTP2	92	SETTINGS[0]
52	09:37:13.613663	leap.netlab1.net	desktop.netlab1.net	HTTP2	92	SETTINGS[0]
53	09:37:13.613851	leap.netlab1.net	desktop.netlab1.net	HTTP2	1514	HEADERS[15]: 200 OK
54	09:37:13.613853	leap.netlab1.net	desktop.netlab1.net	HTTP2	1514	DATA[15][SSL segment of a reassembled PDU]…
55	09:37:13.613867	desktop.netlab1.net	leap.netlab1.net	TCP	54	1805 → 443 [ACK] Seq=1076 Ack=9212 Win=657…
56	09:37:13.614057	leap.netlab1.net	desktop.netlab1.net	HTTP2	1514	DATA[15][SSL segment of a reassembled PDU]…
57	09:37:13.614058	leap.netlab1.net	desktop.netlab1.net	HTTP2	990	DATA[15][SSL segment of a reassembled PDU]
58	09:37:13.614059	leap.netlab1.net	desktop.netlab1.net	HTTP2	1514	DATA[15][SSL segment of a reassembled PDU]
59	09:37:13.614061	leap.netlab1.net	desktop.netlab1.net	HTTP2	498	DATA[15], DATA[15] (text/html)
60	09:37:13.614075	desktop.netlab1.net	leap.netlab1.net	TCP	54	1805 → 443 [ACK] Seq=1076 Ack=13512 Win=65
61	09:37:13.653319	desktop.netlab1.net	leap.netlab1.net	HTTP2	165	HEADERS[17]: GET /ttp_logo_horizontal-02-m…
62	09:37:13.653521	desktop.netlab1.net	leap.netlab1.net	HTTP2	143	HEADERS[19]: GET /oxford-skyline-smallest.…
63	09:37:13.653702	desktop.netlab1.net	leap.netlab1.net	HTTP2	144	HEADERS[21]: GET /meetings.css, WINDOW_UPD…
64	09:37:13.653909	leap.netlab1.net	desktop.netlab1.net	TCP	60	443 → 1805 [ACK] Seq=13512 Ack=1276 Win=32…
65	09:37:13.654253	leap.netlab1.net	desktop.netlab1.net	HTTP2	1514	HEADERS[17]: 200 OK, HEADERS[19]: 200 OK

TLS handshake

HTTP/2 “the menu please”

The menu arrives

Charge!
Client greed

HTTP/2 Server Push (want fries with that?)

“8.2. Server Push

HTTP/2 allows a server to pre-emptively send (or "**push**") responses (along with corresponding "**promised**" requests) to a client in association with a previous client-initiated request. This can be useful when the server knows the client will need to have those responses available in order to fully process the response to the original request.”

“A client can request that server push be disabled, though this is negotiated for each hop independently. The

SETTINGS_ENABLE_PUSH

setting can be set to 0 to indicate that server push is disabled.”

From: RFC 7540

Server Push in Apache, Link Header

If the connection supports PUSH, these two resources will be sent to the client. As a web developer, you may set these headers either directly in your application response or you configure the server via

```
<Location /xxx.html>
    Header add Link "</xxx.css>;rel=preload"
    Header add Link "</xxx.js>;rel=preload"
</Location>
```

If you want to use `preload` links without triggering a PUSH, you can use the `nopush` parameter, as in

```
Link </xxx.css>;rel=preload;nopush
```

or you may disable PUSHes for your server entirely with the directive

```
H2Push Off
```

And there is more:

https://httpd.apache.org/docs/2.4/howto/http2.html

See RFC 8288 about Link in HTML docs

Multiplexing: a realistic comparison, UK to …

Country	h1 time	h2 time	% reduction
Argentina	4.78	3.57	34.0%
New Zealand	5.19	4.05	28.2%
Japan	4.70	3.73	26.2%
China (Beijing)	5.40	4.64	16.5%
South Africa	4.19	3.80	10.2%
United Kingdom	0.94	0.92	2.7%

Time is in seconds

"The page tested (signout) is very typical of BBC web pages so I'd expect the results to be similar across other pages from BBC Online."

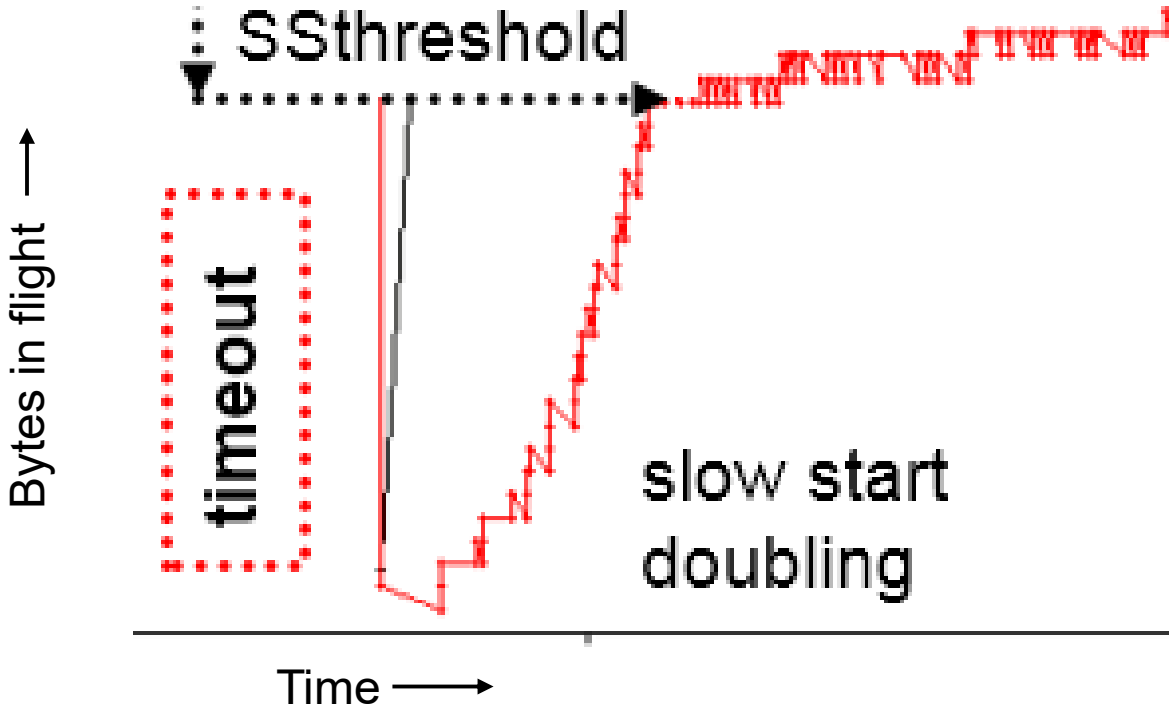
From https://medium.com/bbc-design-engineering/http-2-is-easy-just-turn-it-on-34baad2d1fb1

Limitations: page complexity, buffer sizes, round trip time, network capacity

Hint on reading the next plots

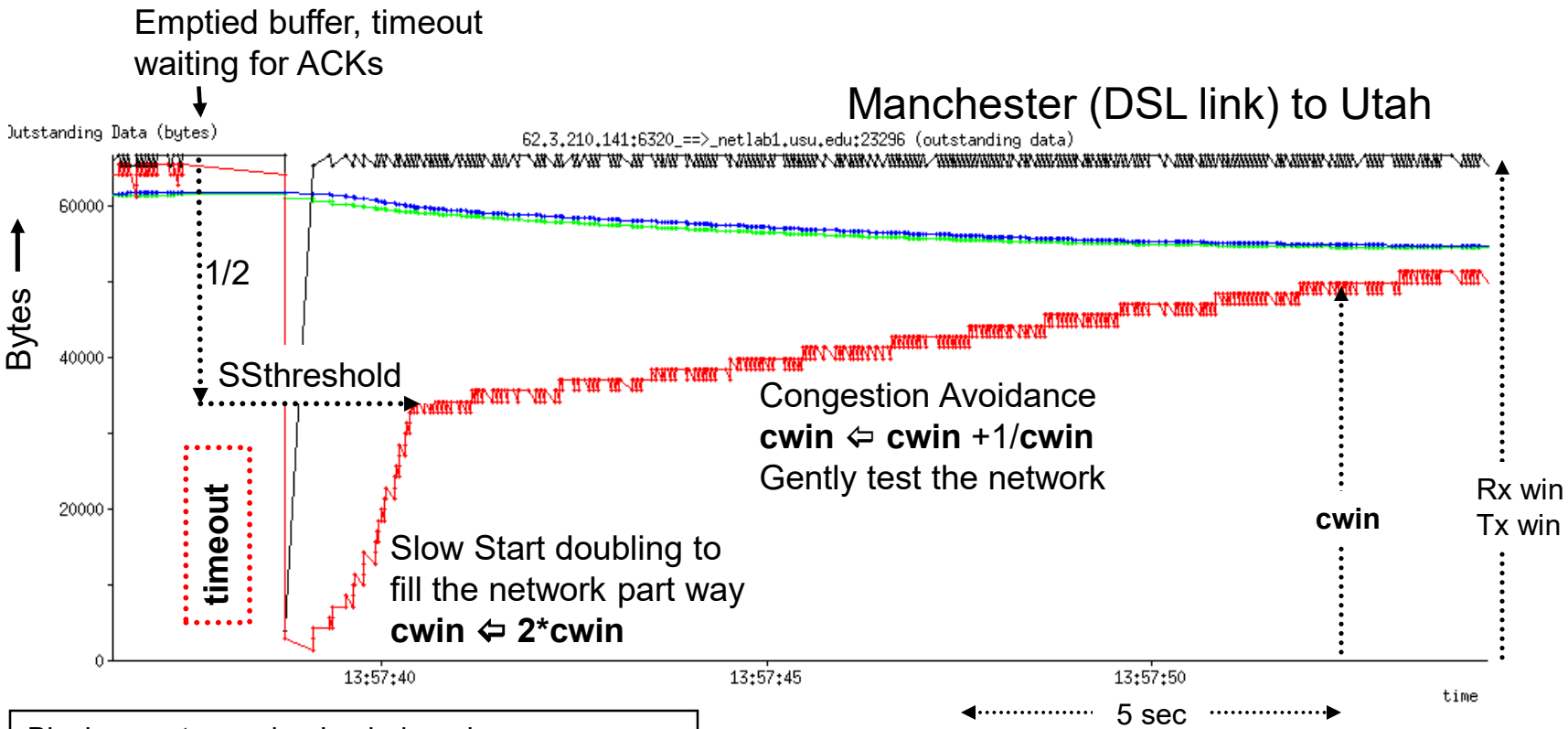
In these pictures we focus on the red line: *the number of bytes allowed to be in flight*, **cwin**. This is the sender's viewpoint.

Note the small up/down tick marks, going up when sending a new packet, down for ACK reception (saying a packet has left the network, thus freeing space on the network).



TCP classical Van Jacobson plot

Detail of a loss recovery interval, illustrating slow start and 1/**cwin** steps
Example starts with packet loss after the initial full 64KB window burst



Black: remote receiver's window size
Red: bytes in flight (sent minus ACK'd)
Blue: simple average bytes in flight
Green: long time weighted average bytes in flight

Duration at a step is set by waiting for ACKs

Made about 14 years ago, before the **cwin** Cubic variation in Linux

TCP heuristics, explaining the picture

- In the beginning the transmitter does not know the receiver's or network's capacity, so it sets **cwin** to be infinite. Blast away…
- The diagram shows this initial blast, followed by a timeout. The bundle had not yet reached the receiver. No surprise.
- Upon loss/timeout a safe amount, known as the "*slow start threshold*", is set as half the previous try and **cwin** starts at 1 packet.
- Learning then begins in earnest: send one packet, wait for its ACK, then replace it with a new packet plus an extra one, which means **cwin** is adjusted to send 1, then 2, 4, 8 and so on to get the ACK clock ticking quickly and fill the network. "*Slow Start*"
- At the "*slow start threshold*" level change tactics to allow **cwin** to grow linearly as **cwin** = **cwin** + 1/**cwin**, to gently sense network capacity. "*Congestion Avoidance*"

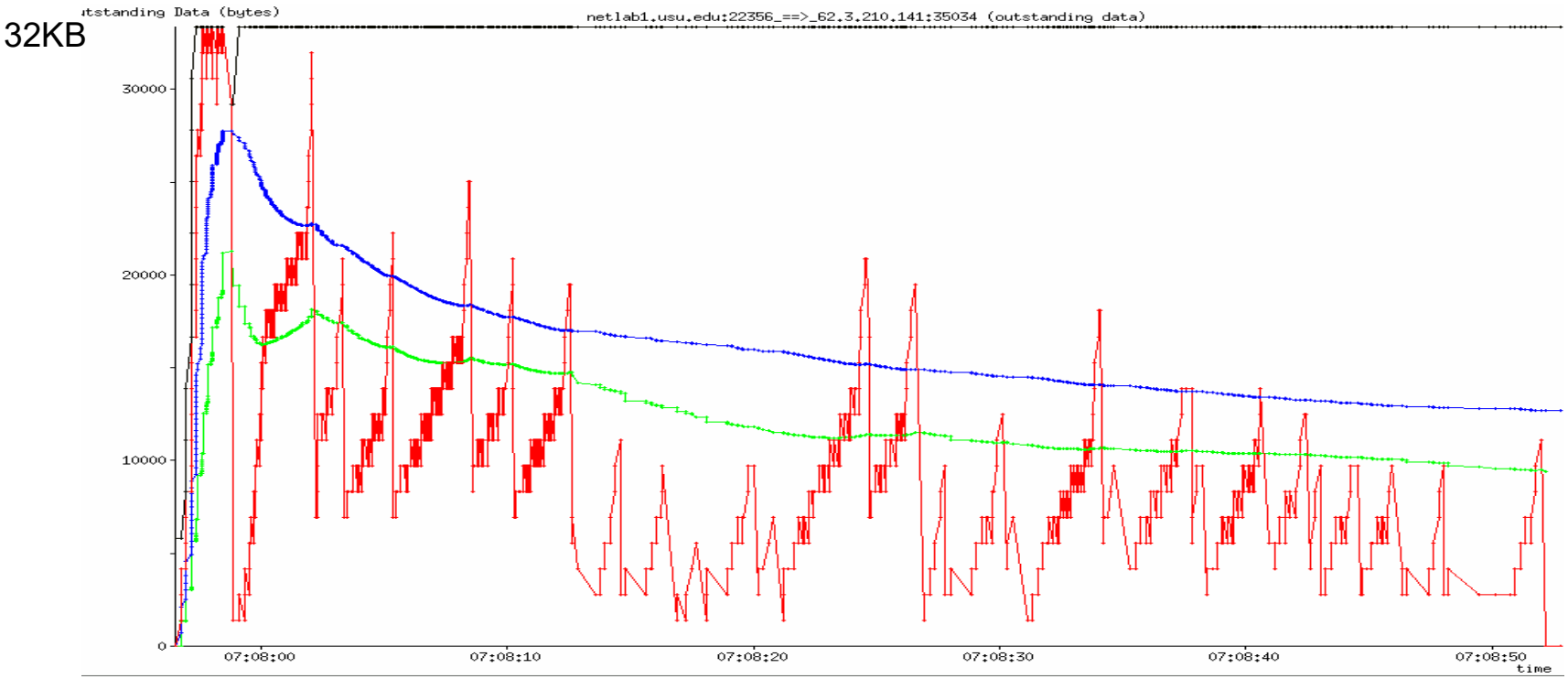
Terms: *Slow start, slow start threshold, congestion avoidance*

TCP heuristics, continued

- Viewing matters over a longer time span in the next picture shows the learning cycle to repeat over and over, adapting to network capacity of the moment.
- Backing away from packet loss must be exponential to avoid system traffic oscillations or wedging. BBR folks seem to ignore this. Halving the last try to set the SSthreshold is exponential.
- The *slow start* phase itself is exponential, repeated doubling, and thus is quick in time to reach the *SSthreshold* value.
- Spikes in the pictures are Fast Recovery bursts to fill gaps indicated by ACKs stuck at a particular sequence number (at a hole) but must be sent because more packets have arrived.
- The *congestion avoidance* phase increases **cwin** slowly while carrying most of the traffic.

Utah to Manchester DSL

Network congestion causes packet loss, restart & relearning



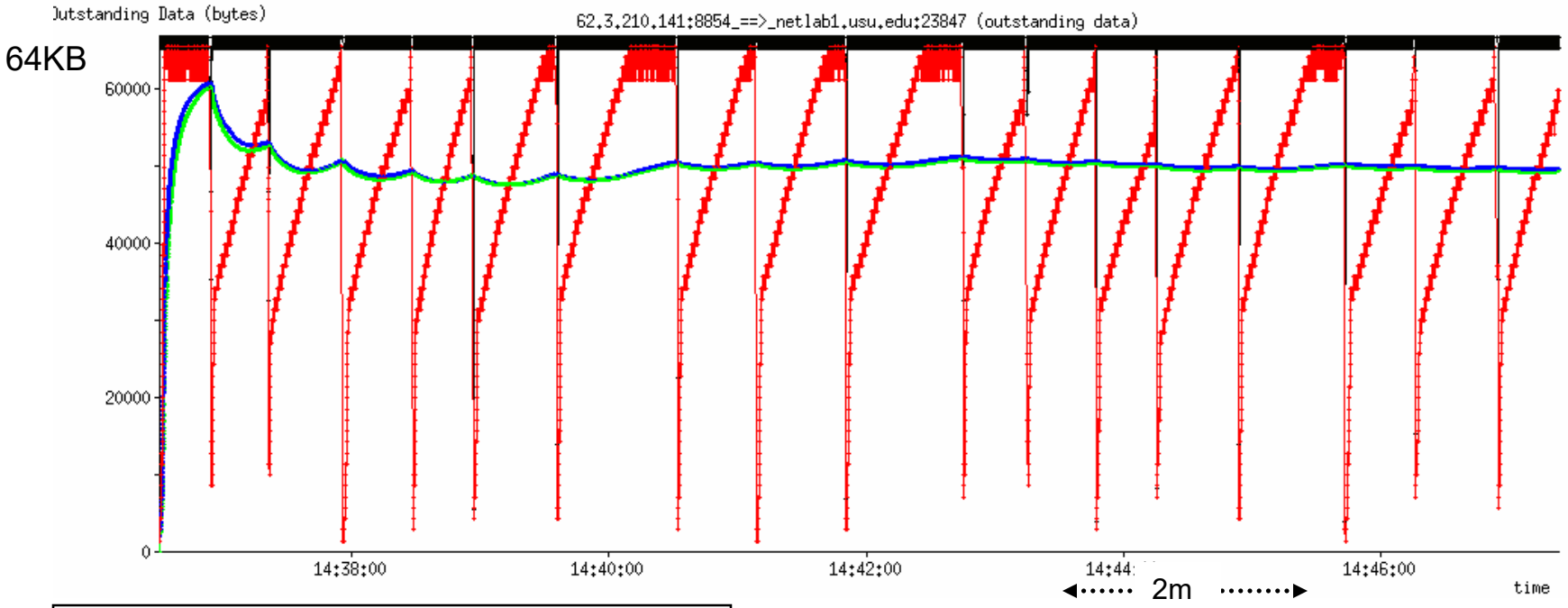
◄····· 20s ·····►

Black: remote receiver's window size
Red: bytes in flight (sent minus ACK'd)
Blue: simple average bytes in flight
Green: long time weighted average bytes in flight

About 30KBps, declining, RTT 200ms
"Slow start" really is not slow

Man-Utah, sending from slow side

Going outward (DSL "Upload") is much less congested
Frequent packet loss and end-point buffer limits are evident

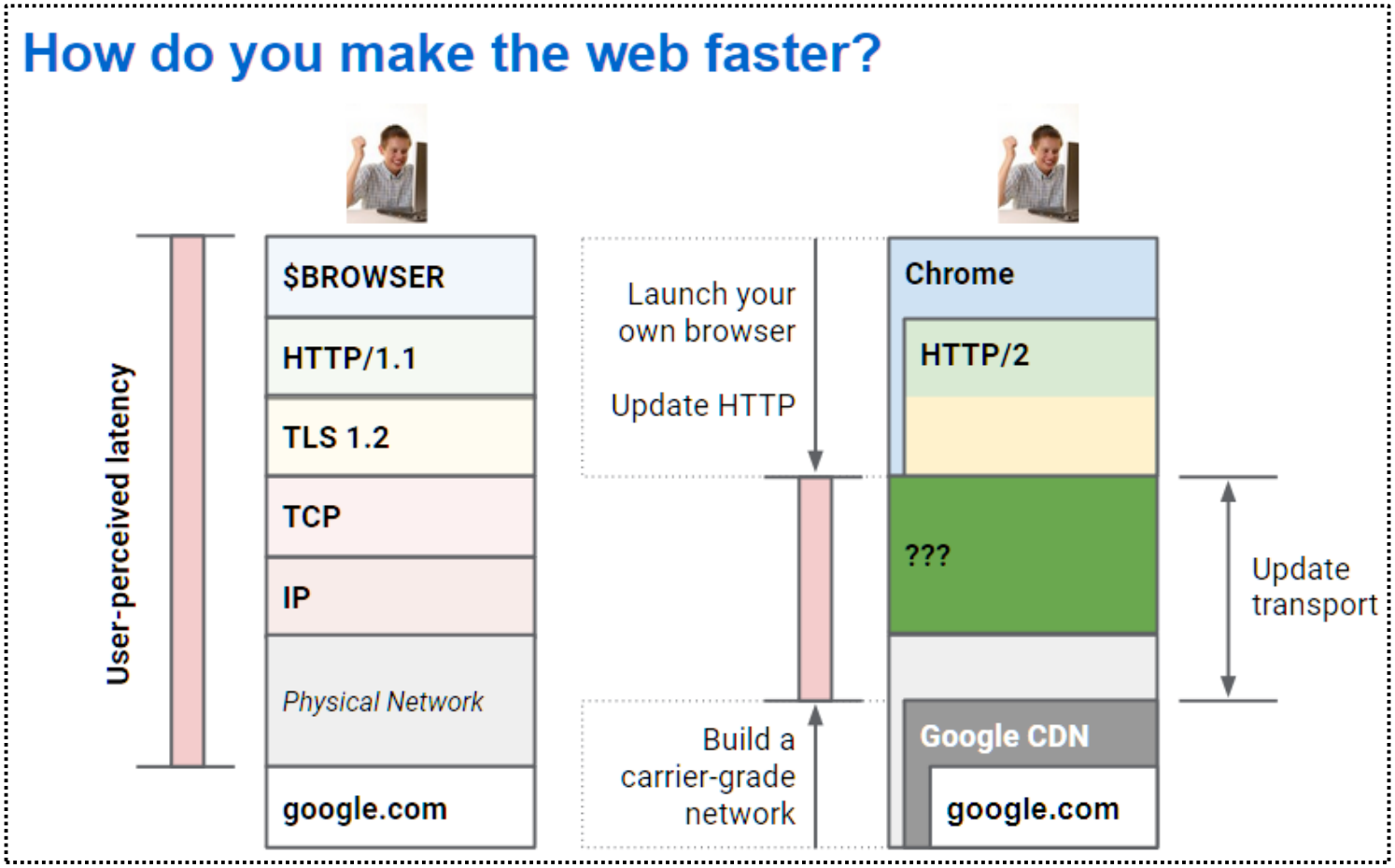


Black: remote receiver's window size
Red: bytes in flight (sent minus ACK'd)
Blue: simple average bytes in flight
Green: long time weighted average bytes in flight

Ten minutes of steady transferring, 56KB/sec. Nearly all time is in congestion avoidance mode

Now on to HTTP/3

A coup d'état, Google style



https://docs.google.com/presentation/d/15e1bLKYeN56GL1oTJSF9OZiUsIrcxisLo9dEyDkWQs/edit#slide=id.g31fceeb3a_8_238

which is an overview of QUIC

QUIC: BBR v1 congestion control algorithm

QUIC's proposed limit on network usage is not **loss-based** but instead is **rate-based**, with the rate computed from a complicated set of measurements of round trip time, queue lengths and many frequent probes.

Sophisticated heuristics try to guess why things may be slower than anticipated and then control further sending. Losses are ignored, for gosh sakes.

A major study has shown that when QUIC BBR v1 is mixed with today's loss based heuristics (TCP's Reno or Cubic algorithms) then it is not fair. A single QUIC connection may consume up to 40% of a network all by itself when faced with regular competition.

See: "Modelling BBR's Interactions with Loss-Based Congestion Control"
http://www.justinesherry.com/papers/ware-imc2019.pdf

HTTP/3 BBR propaganda, remain vigilant

"These problems result from a design choice made when TCP congestion control was created in the 1980s—interpreting packet loss as "congestion."13 This equivalence was true at the time but was because of technology limitations, not first principles. As NICs (network interface controllers) evolved from Mbps to Gbps and memory chips from KB to GB, the relationship between packet loss and congestion became more tenuous.

Today TCP's loss-based congestion control—even with the current best of breed, CUBIC11—is the primary cause of these problems. When bottleneck buffers are large, loss-based congestion control keeps them full, causing buffer bloat. When bottleneck buffers are small, loss-based congestion control misinterprets loss as a signal of congestion, leading to low throughput. Fixing these problems requires an alternative to loss-based congestion control. Finding this alternative requires an understanding of where and how network congestion originates."

https://ai.google/research/pubs/pub45646
The full doc is:
https://queue.acm.org/detail.cfm?id=3022184

Buffer backlog is not a fault.
Loss is from congestion. Duh!
This quote is dubious at best.

HTTP BBR vs the Internet. Round 2

BBR v2

A Model-based Congestion Control

Neal Cardwell, Yuchung Cheng,

Soheil Hassas Yeganeh, Ian Swett, Victor Vasiliev,

Priyaranjan Jha, Yousuk Seung, Matt Mathis

Van Jacobson

https://groups.google.com/d/forum/bbr-dev

IETF 104: Prague, Mar 2019

Google

1

https://datatracker.ietf.org/meeting/104/materials/slides-104-iccrg-an-update-on-bbr-00

BBR v2 re-discovers packet loss

Unfairness

Issues with initial (v1) version of BBR

- Low throughput for Reno/CUBIC flows sharing a bottleneck with bulk BBR flows
- Loss-agnostic; high packet loss rates if bottleneck queue < 1.5*BDP
- ECN-agnostic
- Low throughput for paths with high degrees of aggregation (e.g. wifi)
- Throughput variation due to low cwnd in PROBE_RTT

BBR v2 tackles all of these...

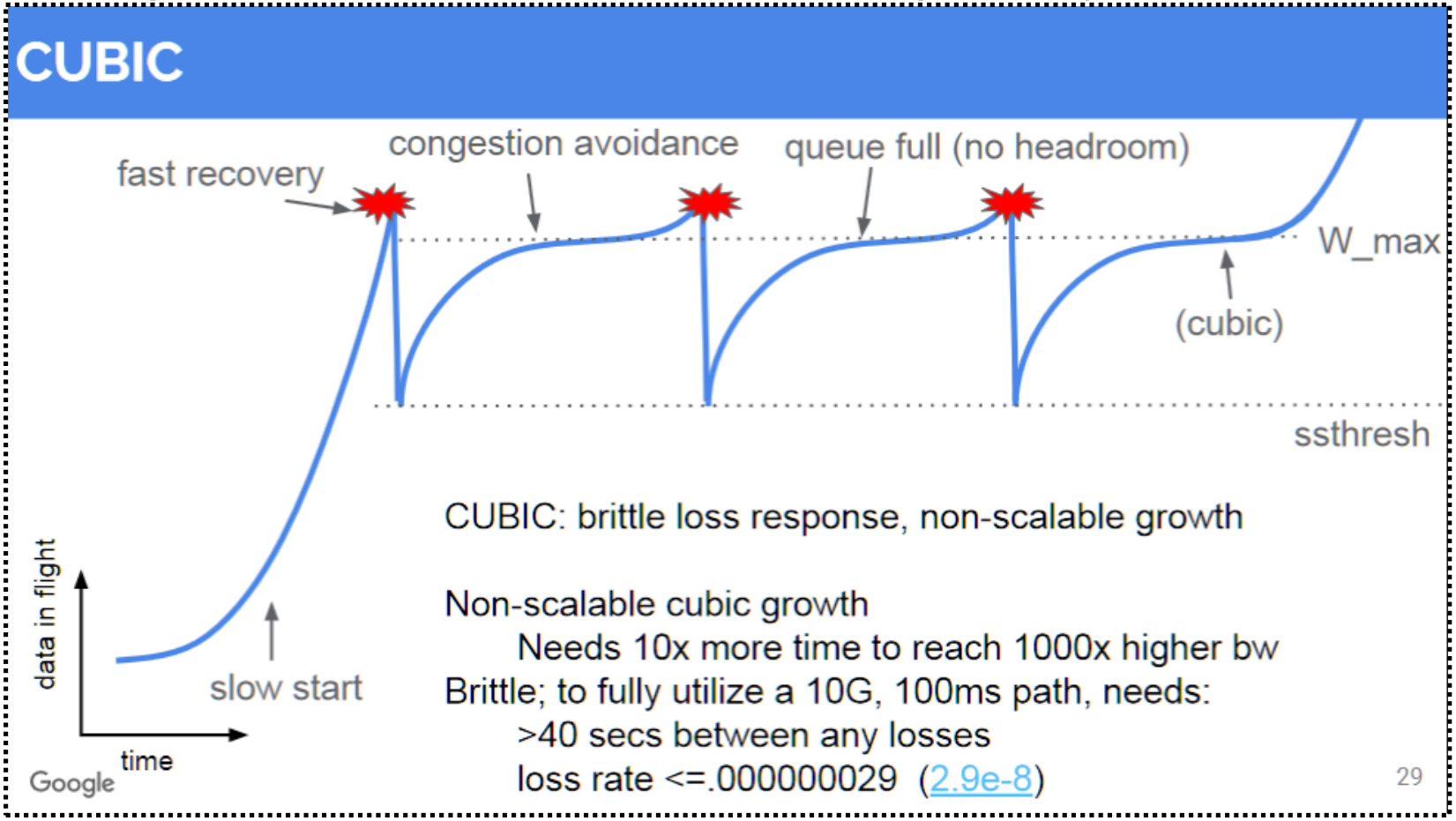
BBR v2 improvements: using packet loss as a signal

- BBR v1: agnostic to loss, susceptible to high loss rates if bottleneck queue < 1.5*BDP
- BBR v2: uses loss as an explicit signal, with an explicit target loss rate ceiling [IETF 102]

https://datatracker.ietf.org/meeting/104/materials/slides-104-iccrg-an-update-on-bbr-00

BDP is bandwidth delay product (i.e., max network path capacity)

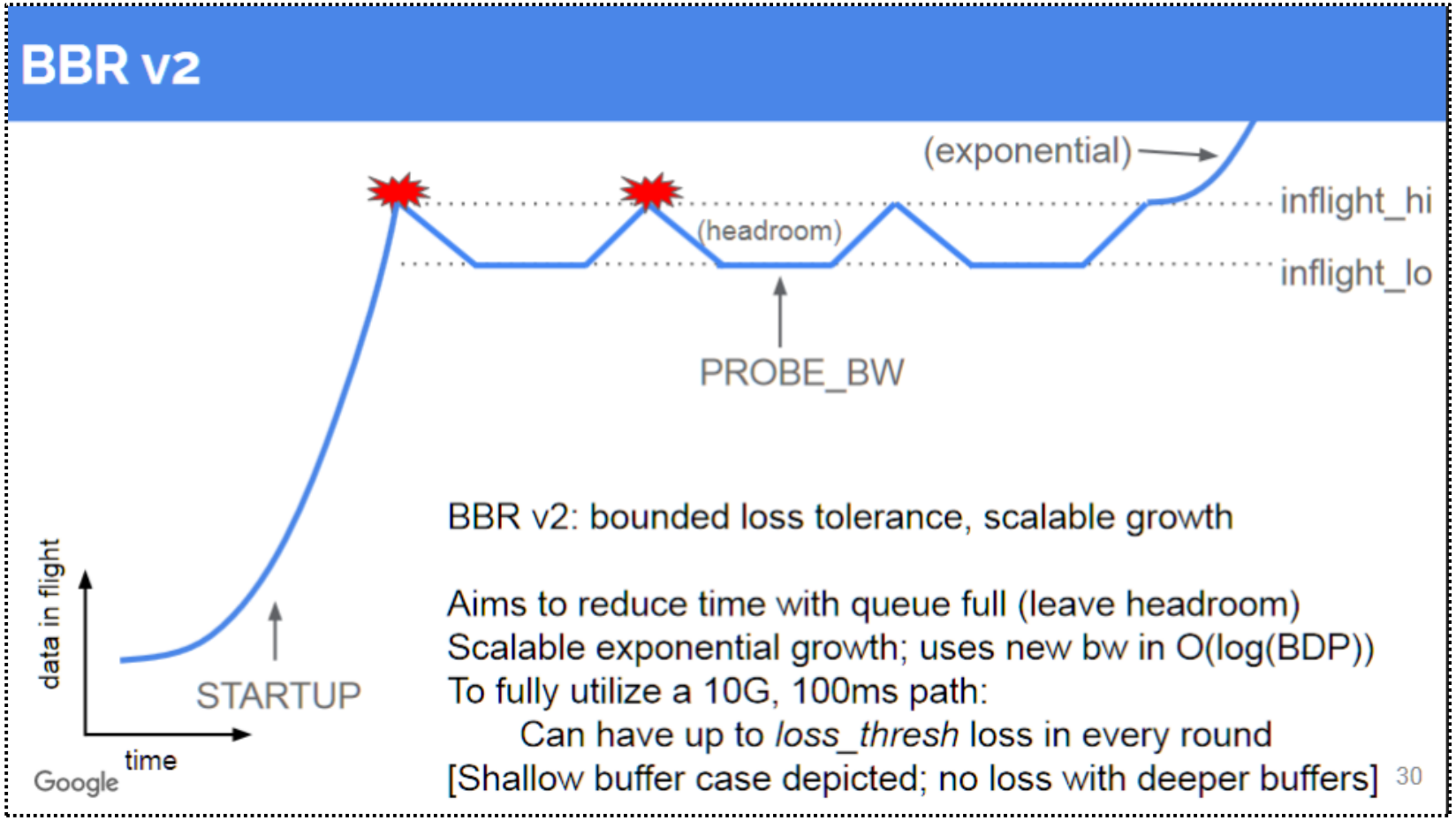
Congestion control, Cubic (today's network)



From: https://datatracker.ietf.org/meeting/104/materials/slides-104-iccrg-an-update-on-bbr-00
See also: https://www.cs.princeton.edu/courses/archive/fall16/cos561/papers/Cubic08.pdf

Be wary of over characterization by the artist; see my real data plots.

Congestion control, BBR v2 nuances



https://datatracker.ietf.org/meeting/104/materials/slides-104-iccrg-an-update-on-bbr-00

Yes, the details are complicated. But it might be a step ahead.

Sensitivity to packet loss, an experiment

Throughput Under Loss

To compare BBR to Cubic, I set up a pre-release build of LSWS 5.4.2 and conducted a few experiments downloading a static file from the web server using the LiteSpeed QUIC client. The bottleneck was set up on the client side using netem and ifb.

Rate (MBit/sec)	Delay (ms)	Loss (%)	Cubic (sec)	BBR (sec)
20	25	0	4.5	4.5
20	25	0.5	6.7	4.6
20	25	1	8.6	4.6
20	25	2	14.3	4.7
20	25	3	17.6	4.7

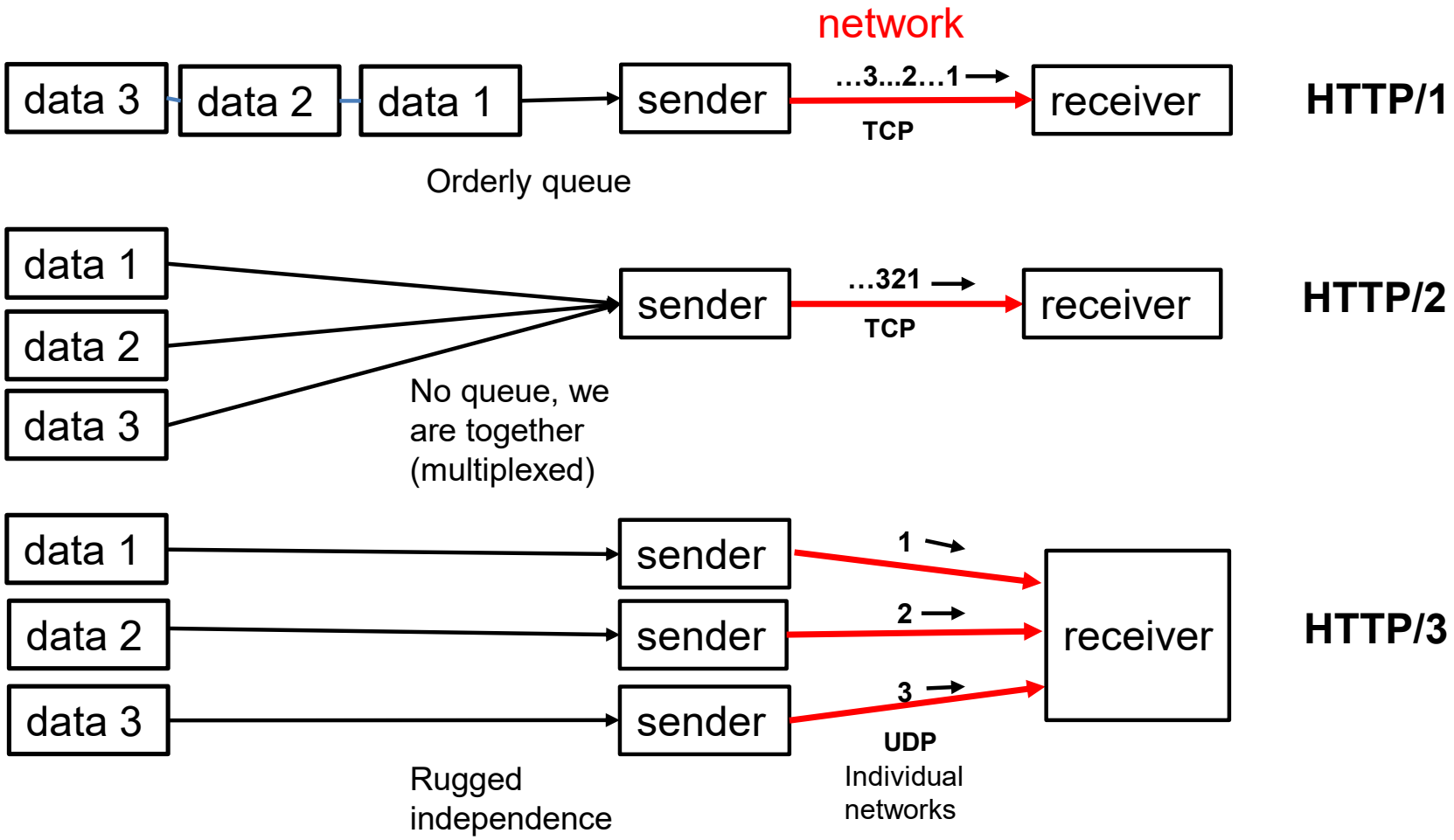
Table 1: Time to transfer 10 MB file, in seconds

It is evident that Cubic is very sensitive to packet loss, while BBR is not.

Beware: BBR v1 is overly aggressive

https://blog.litespeedtech.com/2019/10/28/bbr-congestion-control-quic-http-3/

HTTP data flow conceptual designs



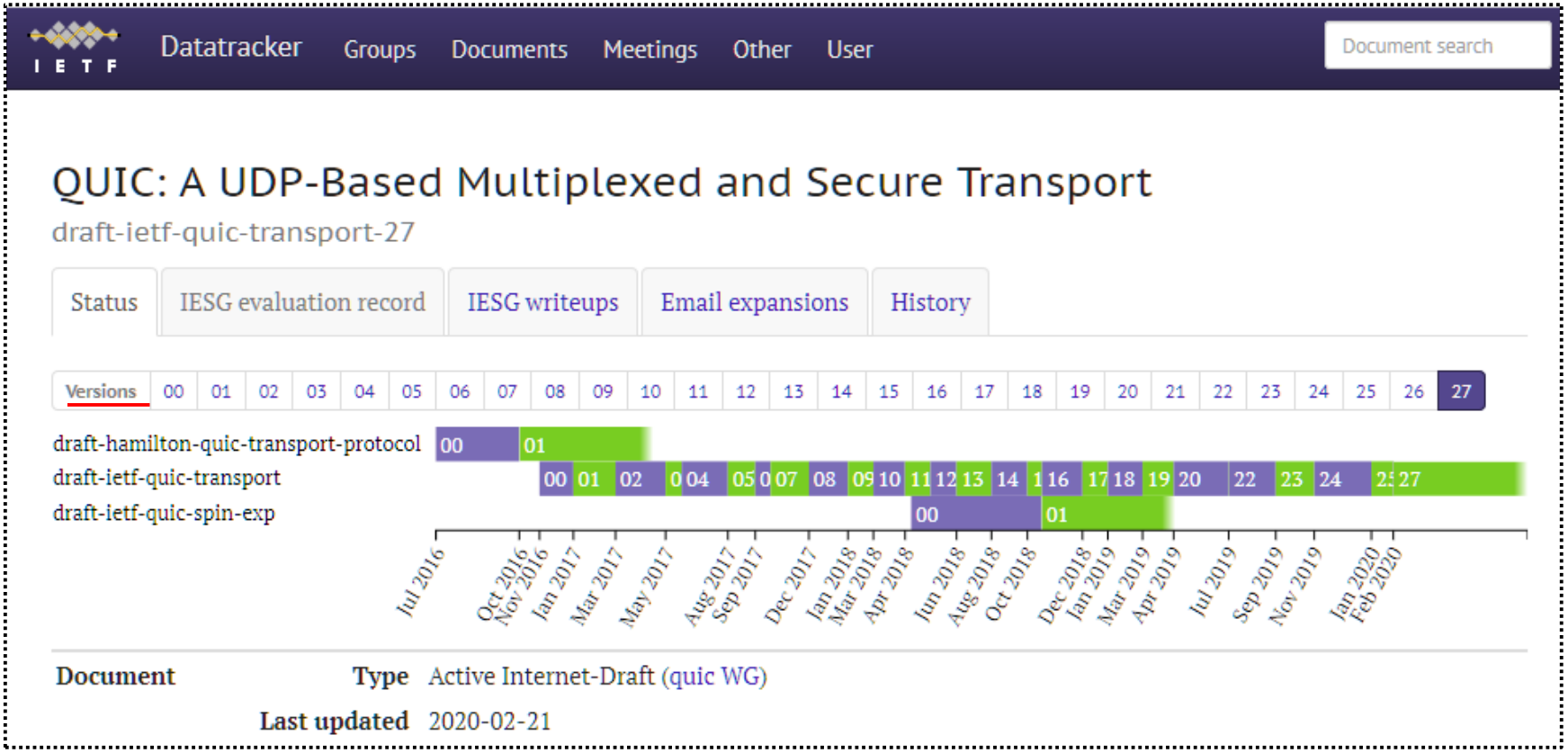
HTTP/3 tries to change the game

- HTTP/1 and /2 use TCP for reliable sequenced delivery.
- TCP can observe network congestion and adjust its sending rate to accommodate gracefully.

- HTTP/3 uses UDP to go fast and avoid TCP “Head of Line” blocking by using more channels to bypass the waiting channel (ignores congestion blocks == a dangerous driver).
- UDP ignores reliable, sequenced delivery, network congestion. It has no idea about the network, thus it does not slow down due to congestion. User-space code is needed to tame this behaviour.
- HTTP/3 reluctantly tries to add back some TCP features, but in it’s own peculiar ways which have yet to be well proven.

HTTP/3 specs change monthly, in much detail, over several years, no end in sight

https://datatracker.ietf.org/doc/draft-ietf-quic-transport/



I wonder if they know the term “overdraft”?

A few notables about QUIC HTTP/3

- Minimum IP packet length of 1200 bytes, to help reduce UDP DoS amplification attacks, and to probe for path MTU.
- Session, source and destination identifiers are carried within each UDP QUIC packet, allowing for changing IP numbers during a session. This uses an elaborate set of rules about handling of identifiers. Such handing is at the QUIC level (user-space), per stream.

- BBR class congestion control is proposed which is more aggressive than TCP/CUBIC about missing packets and assumes that losses occur by more than congestion (router queue overflow) and such losses can be accommodated with quick replacements if the algorithms guess correctly that congestion is not involved. This involves much speculation, not comforting.

From draft 25:

2. HTTP/3 Protocol Overview

HTTP/3 provides a transport for HTTP semantics using the QUIC transport protocol and an internal framing layer similar to HTTP/2.

Once a client knows that an HTTP/3 server exists at a certain endpoint, it opens a QUIC connection. QUIC provides protocol negotiation, stream-based multiplexing, and flow control. An HTTP/3 endpoint can be discovered using HTTP Alternative Services; this process is described in greater detail in Section 3.2.

Within each stream, the basic unit of HTTP/3 communication is a frame (Section 7.2). Each frame type serves a different purpose. For example, HEADERS and DATA frames form the basis of HTTP requests and responses (Section 4.1).

Multiplexing of requests is performed using the QUIC stream abstraction, described in Section 2 of [QUIC-TRANSPORT]. Each request and response consumes a single QUIC stream. Streams are independent of each other, so one stream that is blocked or suffers packet loss does not prevent progress on other streams.

"Congestion? We have heard of that, haven't we guys. Guys? Guys??"

Server says it can do HTTP/3, in two-steps

3.2. Discovering an HTTP/3 Endpoint

An HTTP origin advertises the availability of an equivalent HTTP/3 endpoint via the Alt-Svc HTTP response header field or the HTTP/2 ALTSVC frame ([ALTSVC]), using the ALPN token defined in Section 3.3.

For example, an origin could indicate in an HTTP response that HTTP/3

was available on UDP port 50781 at the same hostname by including the following header field:

```
Alt-Svc: h3=":50781"
```

On receipt of an Alt-Svc record indicating HTTP/3 support, a client MAY attempt to establish a QUIC connection to the indicated host and port and, if successful, send HTTP requests using the mapping described in this document.

Connectivity problems (e.g. firewall blocking UDP) can result in QUIC connection establishment failure, in which case the client SHOULD continue using the existing connection or try another alternative endpoint offered by the origin.

Servers MAY serve HTTP/3 on any UDP port, since an alternative always includes an explicit port.

From https://tools.ietf.org/html/draft-ietf-quic-http-25#page-8

Step one, capture the flag

The ALPN flag: where to go next

Step 2, try there

That is a new Port number, UDP, second web stack

HTTP/3 documentation snippets

4.4. Server Push

Server push is an interaction mode introduced in HTTP/2 [HTTP2] which permits a server to push a request-response exchange to a client in anticipation of the client making the indicated request. This trades off network usage against a potential latency gain. HTTP/3 server push is similar to what is described in HTTP/2 [HTTP2], but uses different mechanisms.

6. Stream Mapping and Usage

A QUIC stream provides reliable in-order delivery of bytes, but makes no guarantees about order of delivery with regard to bytes on other streams. On the wire, data is framed into QUIC STREAM frames, but this framing is invisible to the HTTP framing layer. The transport layer buffers and orders received QUIC STREAM frames, exposing the data contained within as a reliable byte stream to the application. Although QUIC permits out-of-order delivery within a stream, HTTP/3 does not make use of this feature.

QUIC, its words about congestion

6.2. Slow Start

> QUIC begins every connection in slow start and exits slow start upon loss or upon increase in the ECN-CE counter. QUIC re-enters slow start anytime the congestion window is less than ssthresh, which only occurs after persistent congestion is declared. While in slow start, QUIC increases the congestion window by the number of bytes acknowledged when each acknowledgment is processed.

Fine. That is the TCP way.

6.3. Congestion Avoidance

> Slow start exits to congestion avoidance. Congestion avoidance in NewReno uses an additive increase multiplicative decrease (AIMD) approach that increases the congestion window by one maximum packet size per congestion window acknowledged. When a loss is detected, NewReno halves the congestion window and sets the slow start threshold to the new congestion window.

Explicit **C**ongestion **N**otification: bits in IP&TCP set by a router or end points, only if they wish to.

6.4. Recovery Period

> Recovery is a period of time beginning with detection of a lost packet or an increase in the ECN-CE counter. Because QUIC does not retransmit packets, it defines the end of recovery as a packet sent after the start of recovery being acknowledged. This is slightly different from TCP's definition of recovery, which ends when the lost packet that started recovery is acknowledged.
>
> The recovery period limits congestion window reduction to once per round trip. During recovery, the congestion window remains unchanged irrespective of new losses or increases in the ECN-CE counter.

reluctant adjuster

https://tools.ietf.org/html/draft-ietf-quic-recovery-24#section-6.2

Explicit Congestion Notification ECN: friendly advice from downstream boxes

- ECN requires specific support at both the Internet layer and the transport layer for the following reasons:
- In TCP/IP, routers operate within the Internet layer, while the transmission rate is handled by the endpoints at the transport layer.
- Congestion may be handled only by the transmitter, but since it is known to have happened only after a packet was sent, there must be an echo of the congestion indication by the receiver to the transmitter.
- Without ECN, congestion indication echo is achieved indirectly by the detection of lost packets. With ECN, the congestion is indicated by setting the ECN field within an IP packet to CE and is echoed back by the receiver to the transmitter by setting proper bits in the header of the transport protocol. For example, when using TCP, the congestion indication is echoed back by setting the ECE bit.

Use of ECN has been found to be detrimental to performance on highly congested networks when using AQM algorithms that never drop packets.[8] Modern AQM implementations avoid this pitfall by dropping rather than marking packets at very high load.

From https://en.wikipedia.org/wiki/Explicit_Congestion_Notification

AQM is active queue management, control of overflow in a middle box

What about SSL/TLS details?

From https://www.feistyduck.com/bulletproof-tls-newsletter/:

"OpenSSL developers published a blog post discussing QUIC and a possible future API for it. The OpenSSL developers believe that QUIC is not stable enough to already get a stable API and therefore will likely not be provided in the upcoming version 3.0.0."

From https://www.openssl.org/blog/blog/2020/02/17/QUIC-and-OpenSSL/:

"Judging from IETF's datatracker at the moment of writing, QUIC is still at a point in its development where it is difficult to predict what stability to expect. Based on that, and our recent experience with the TLSv1.3 implementation, we consider there to be a high risk that the IETF process will not have reached sufficient maturity by the time that we need to freeze the OpenSSL 3.0 APIs when we release beta1 in June of this year."

Report card for HTTP/3+QUIC

+ HTTP/2 style multiplexing to reduce waiting on request-responses.
+ Possibly, maybe, perhaps, refine congestion control algorithms, but that is uncertain and absence could lead to traffic unfairness.
+ Allow client IP numbers to vary during a session, but also be a firewall concern.
+ Requires TLS (v1.3), improves TLS negotiation performance.

- UDP rather than TCP. Places much complicated code in user space of both clients and servers, requires new Port and firewall rules.
- Fall back to TCP if no QUIC UDP service, complicates offering web services (duplicate web server stacks at each end).
- Bypass network congestion indications in an effort to Go Go Go Fast.
- A strong focus on going fast, very little about the consequences.
- Net gain of performance over HTTP/2 is small, plus traffic fairness.

HTTP/3 is still very much a work in <need of> progress

MindWorks Inc. Ltd
210 Burnley Road
Weir
Bacup
OL13 8QE UK

Telephone: +44 (0) 170 687 1900
Fax: +44 (0) 170 687 8203
Web: www.mindworksuk.com
Email: training@mindworksuk.com