QUIC tracing

Victor Vasiliev

IETF 104, tsvarea

Why collect network traces?

1. Transport stack developers: debugging implementation
   - Debugging congestion control
   - Debugging flow control
   - Debugging code that sends ACKs
2. Network operators: debugging network itself
   - Diagnosing network problems by examining individual flows
   - Monitoring network by analyzing traces in aggregate
3. Server operators: debugging applications
   - Application performance may be affected by transport issues
   - This includes both failures specific to a network, and general issues with the way application uses the network

What is a trace?

A trace is a transcript of events that occur during a connection.

Various kinds of events can be recorded at transport level:

1. Connection-level events (open, closed, migrated, stalled)
2. Packet-level events (packet sent, lost, acked, retransmitted)
3. Stream-level events (stream opened, closed)

Since our focus here is congestion control and loss recovery, we will focus on **packet-level events**.

Where do traces come from?

Three options:

1. Taken at (or very close) the sender
   - Congestion control
   - Loss detection
2. Taken at the receiver
   - Flow control
   - ACK policy
3. Taken at a midpoint on the network path
   - Has relevant information, but seldom as useful as previous two

In this presentation, all traces are from the sender’s perspective.

What’s in a trace: TCP edition

Usually, a TCP trace is just a recording of TCP packets as seen on the wire, together with time recorded.

What’s in a trace: TCP edition (text)

```
$ tcpdump -r upload.pcap
20:33:50.292870 IP alice.example.42938 > bob.example.443: Flags [S], seq 712278647, win 29200, options [mss 1460,sackOK,TS val 2472050221 ecr 0,nop,wscale 7], length 0
20:33:50.300518 IP bob.example.443 > alice.example.42938: Flags [S.], seq 679305247, ack 712278648, win 60192, options [mss 1380,sackOK,TS val 564023105 ecr 2472050221,nop,wscale 8],
length 0
20:33:50.300582 IP alice.example.42938 > bob.example.443: Flags [.], ack 1, win 229, options [nop,nop,TS val 2472050229 ecr 564023105], length 0
20:33:50.300953 IP alice.example.42938 > bob.example.443: Flags [P.], seq 1:519, ack 1, win 229, options [nop,nop,TS val 2472050229 ecr 564023105], length 518
20:33:50.307999 IP bob.example.443 > alice.example.42938: Flags [.], ack 519, win 240, options [nop,nop,TS val 564023112 ecr 2472050229], length 0
20:33:50.308236 IP bob.example.443 > alice.example.42938: Flags [P.], seq 1:148, ack 519, win 240, options [nop,nop,TS val 564023112 ecr 2472050229], length 147
20:33:50.308256 IP alice.example.42938 > bob.example.443: Flags [.], ack 148, win 237, options [nop,nop,TS val 2472050236 ecr 564023112], length 0
20:33:50.308489 IP alice.example.42938 > bob.example.443: Flags [P.], seq 519:570, ack 148, win 237, options [nop,nop,TS val 2472050236 ecr 564023112], length 51
20:33:50.309804 IP alice.example.42938 > bob.example.443: Flags [P.], seq 570:1113, ack 148, win 237, options [nop,nop,TS val 2472050238 ecr 564023112], length 543
20:33:50.316842 IP bob.example.443 > alice.example.42938: Flags [.], ack 1113, win 244, options [nop,nop,TS val 564023121 ecr 2472050236], length 0
20:33:50.389196 IP bob.example.443 > alice.example.42938: Flags [P.], seq 148:404, ack 1113, win 244, options [nop,nop,TS val 564023193 ecr 2472050236], length 256
20:33:50.390874 IP alice.example.42938 > bob.example.443: Flags [P.], seq 1113:1517, ack 404, win 245, options [nop,nop,TS val 2472050319 ecr 564023193], length 404
20:33:50.402236 IP bob.example.443 > alice.example.42938: Flags [.], ack 1517, win 248, options [nop,nop,TS val 564023206 ecr 2472050319], length 0
20:33:50.464570 IP bob.example.443 > alice.example.42938: Flags [P.], seq 404:544, ack 1517, win 248, options [nop,nop,TS val 564023269 ecr 2472050319], length 140
20:33:50.465948 IP alice.example.42938 > bob.example.443: Flags [P.], seq 1517:1573, ack 544, win 254, options [nop,nop,TS val 2472050394 ecr 564023269], length 56
20:33:50.472943 IP bob.example.443 > alice.example.42938: Flags [.], ack 1573, win 248, options [nop,nop,TS val 564023277 ecr 2472050394], length 0
20:33:50.480014 IP alice.example.42938 > bob.example.443: Flags [.], seq 1573:4309, ack 544, win 254, options [nop,nop,TS val 2472050408 ecr 564023277], length 2736
20:33:50.480105 IP alice.example.42938 > bob.example.443: Flags [.], seq 4309:7045, ack 544, win 254, options [nop,nop,TS val 2472050408 ecr 564023277], length 2736
20:33:50.480116 IP alice.example.42938 > bob.example.443: Flags [.], seq 7045:9781, ack 544, win 254, options [nop,nop,TS val 2472050408 ecr 564023277], length 2736
20:33:50.480170 IP alice.example.42938 > bob.example.443: Flags [.], seq 9781:12517, ack 544, win 254, options [nop,nop,TS val 2472050408 ecr 564023277], length 2736
20:33:50.480341 IP alice.example.42938 > bob.example.443: Flags [.], seq 12517:15253, ack 544, win 254, options [nop,nop,TS val 2472050408 ecr 564023277], length 2736
20:33:50.487056 IP bob.example.443 > alice.example.42938: Flags [.], ack 2941, win 259, options [nop,nop,TS val 564023291 ecr 2472050408], length 0
20:33:50.487089 IP alice.example.42938 > bob.example.443: Flags [.], seq 15253:17989, ack 544, win 254, options [nop,nop,TS val 2472050415 ecr 564023291], length 2736
20:33:50.487101 IP bob.example.443 > alice.example.42938: Flags [.], ack 4309, win 270, options [nop,nop,TS val 564023291 ecr 2472050408], length 0
20:33:50.487115 IP alice.example.42938 > bob.example.443: Flags [.], seq 17989:20725, ack 544, win 254, options [nop,nop,TS val 2472050415 ecr 564023291], length 2736
20:33:50.487126 IP bob.example.443 > alice.example.42938: Flags [.], ack 5677, win 280, options [nop,nop,TS val 564023291 ecr 2472050408], length 0
20:33:50.487133 IP bob.example.443 > alice.example.42938: Flags [.], ack 7045, win 291, options [nop,nop,TS val 564023291 ecr 2472050408], length 0
20:33:50.487145 IP alice.example.42938 > bob.example.443: Flags [.], seq 20725:24829, ack 544, win 254, options [nop,nop,TS val 2472050415 ecr 564023291], length 4104
20:33:50.487155 IP bob.example.443 > alice.example.42938: Flags [.], ack 8413, win 302, options [nop,nop,TS val 564023291 ecr 2472050408], length 0
20:33:50.487175 IP bob.example.443 > alice.example.42938: Flags [.], ack 9781, win 312, options [nop,nop,TS val 564023291 ecr 2472050408], length 0
[...]
```

What’s in a trace: TCP edition

Many graphs to draw:

- Throughput graph
- Goodput graph
- RTT graph
- Loss rate
- rwin graph
- Time-sequence graph

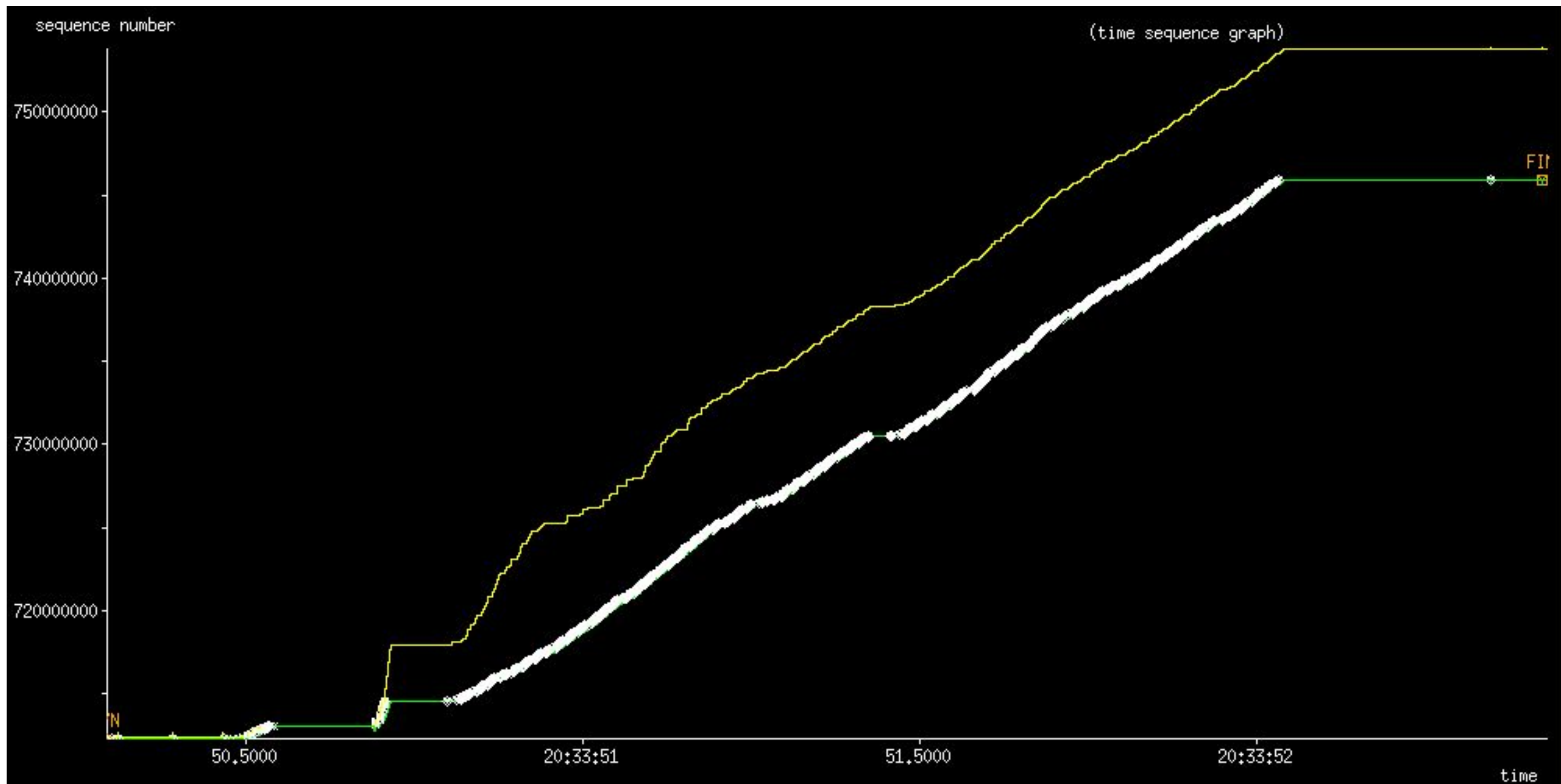
What's in a trace: TCP edition

Many graphs to draw:

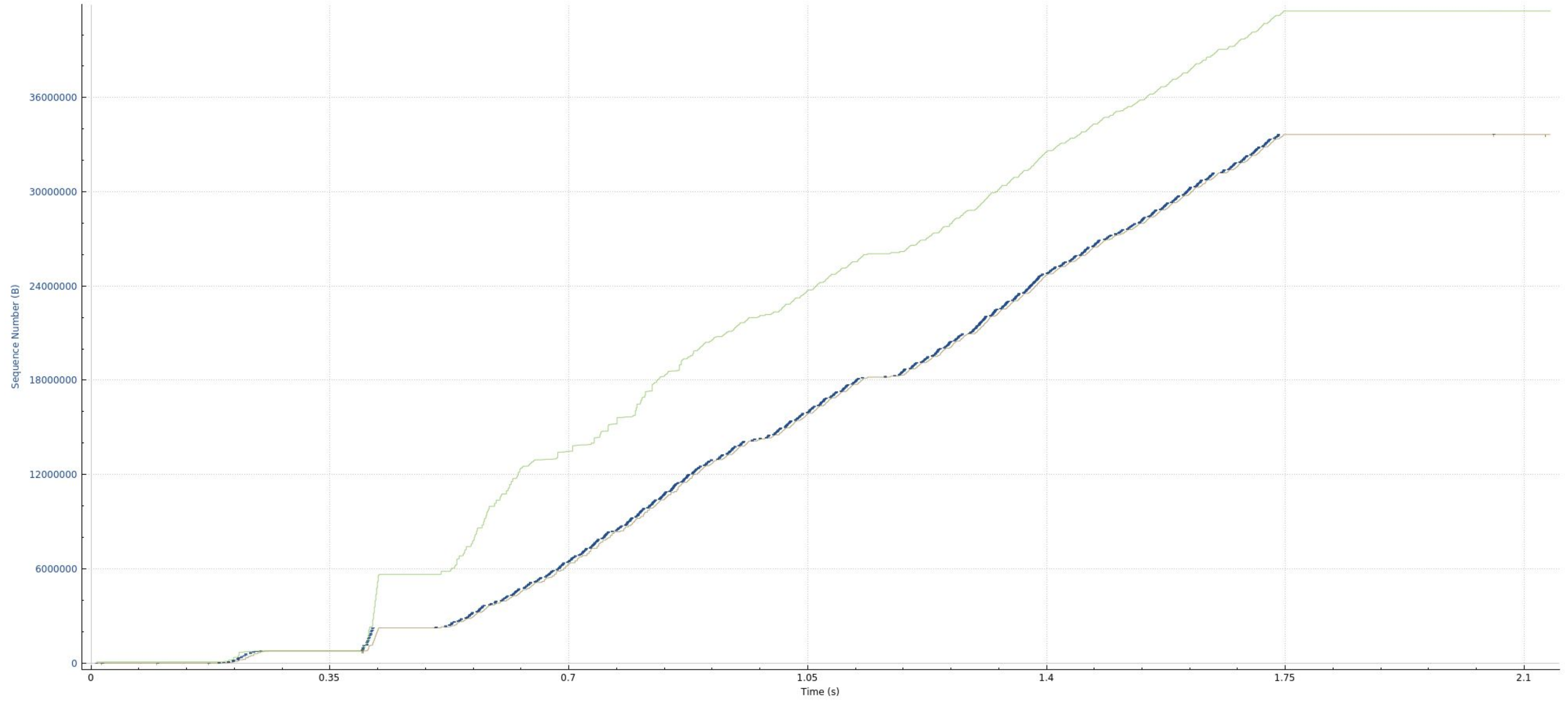
- Throughput graph
- Goodput graph
- RTT graph
- Loss rate
- rwin graph
- Time-sequence graph ← **actually has everything**

Time-sequence plot

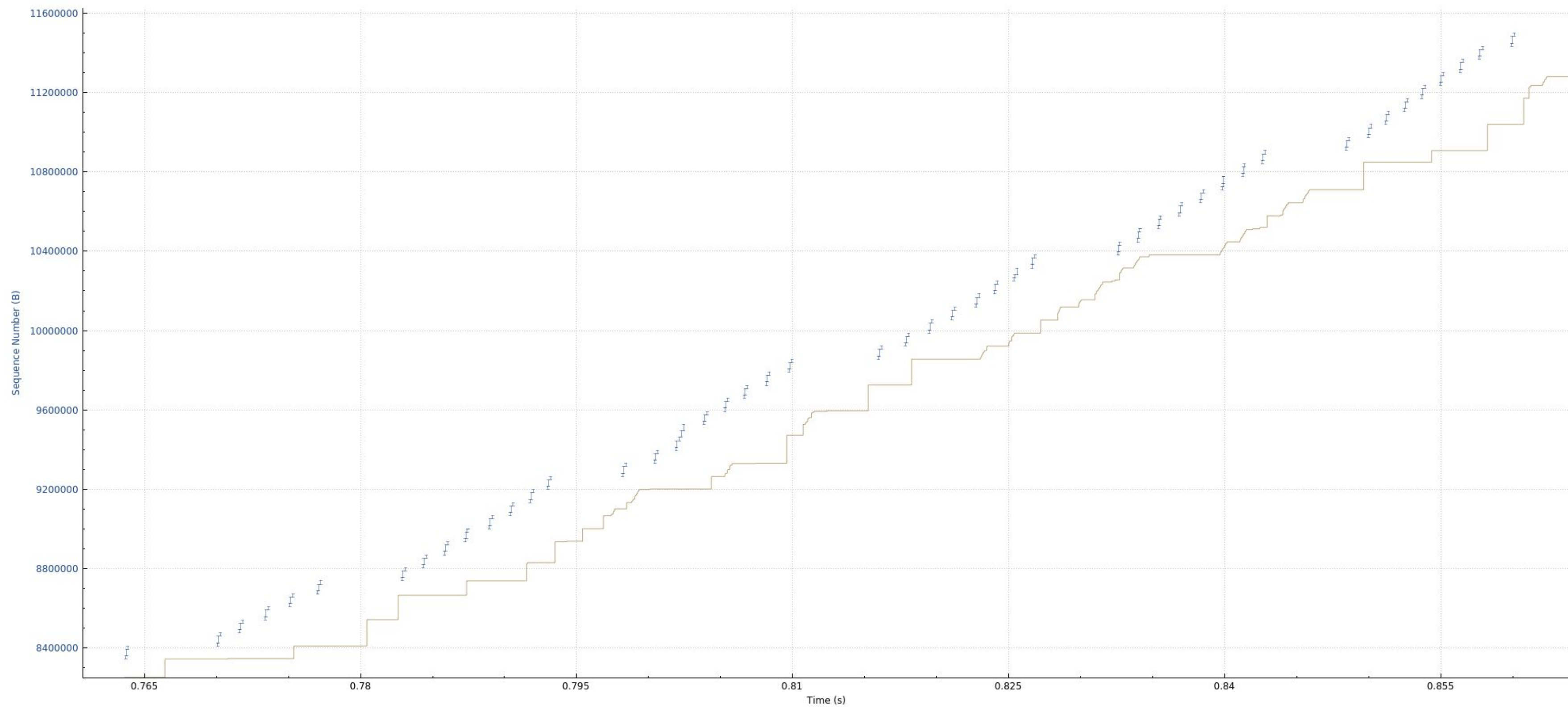
- X axis is time (usually from the first event)
- Y axis is TCP sequence number (usually adjusted to start with zero)
- Each sent packet is marked with a vertical line
- A solid line represents the graph of the last sequence number fully acked
- SACK representations can vary greatly from tool to tool
- Sometimes, a line representing rwin is drawn.



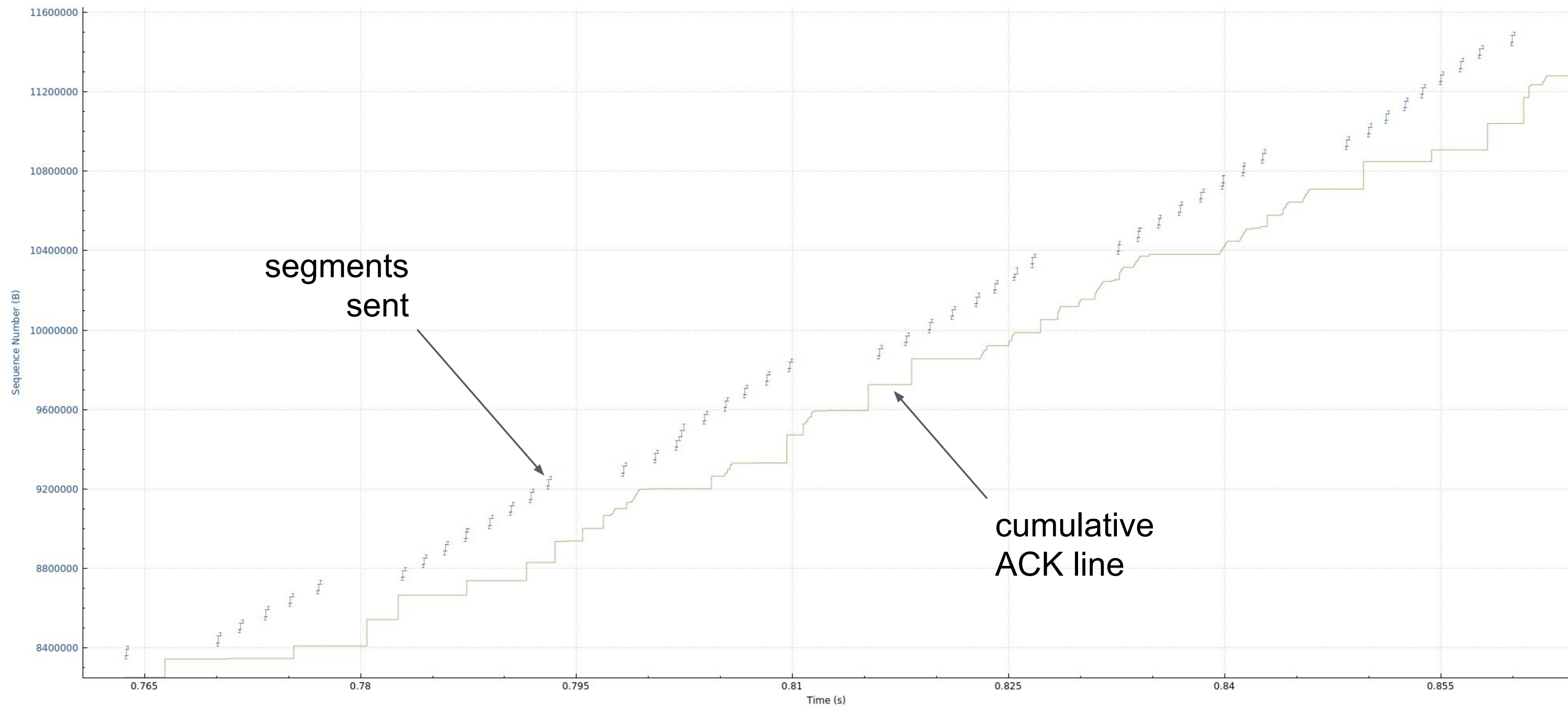
xplot



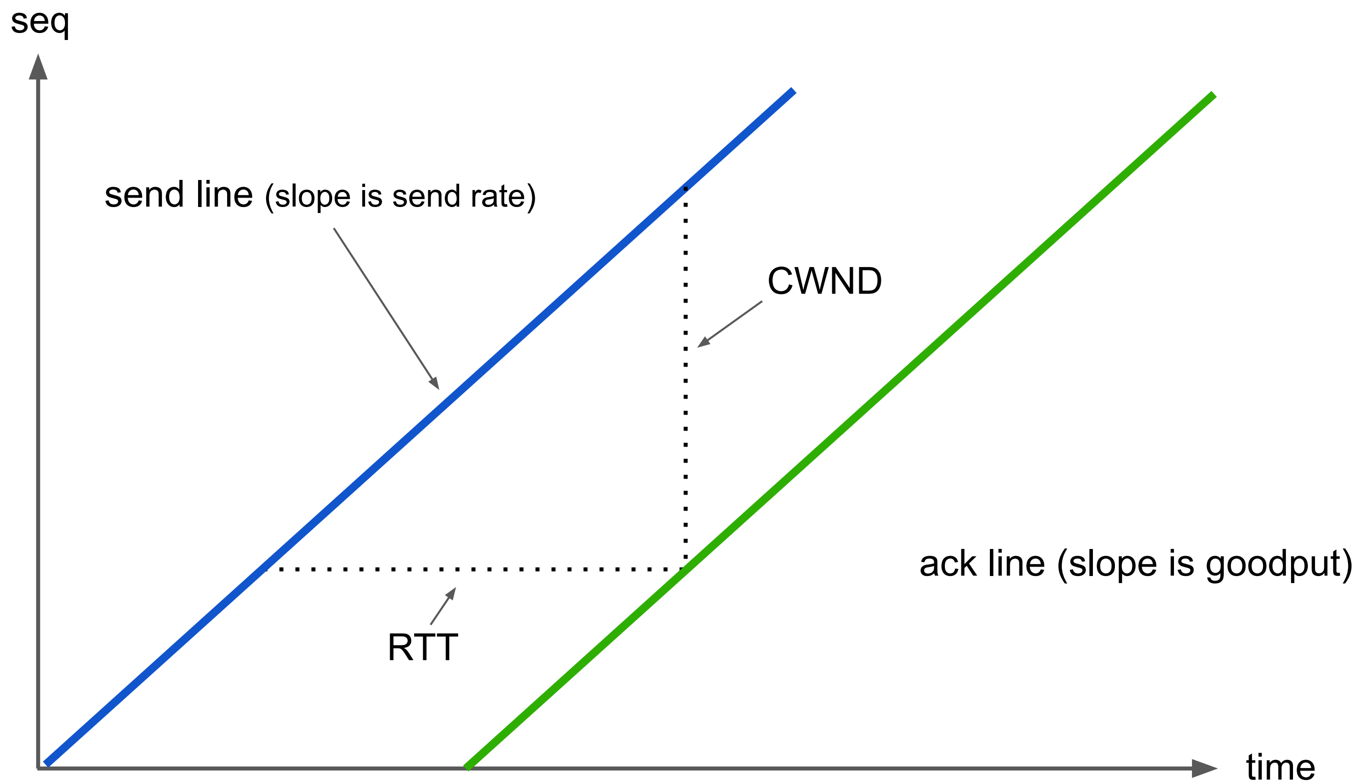
Wireshark



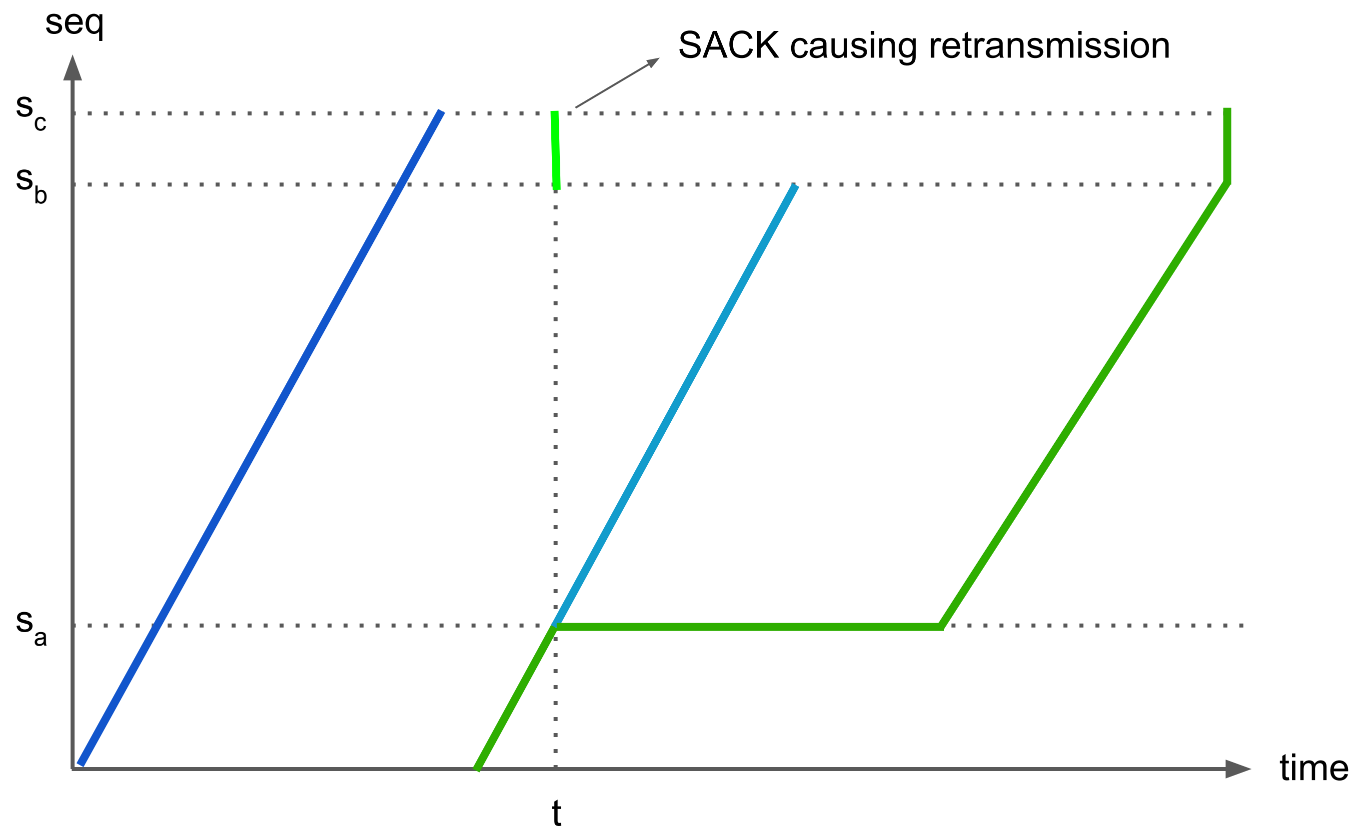
Wireshark (zoomed in)



Wireshark (zoomed in)



Idealized time-sequence plot of a TCP connection



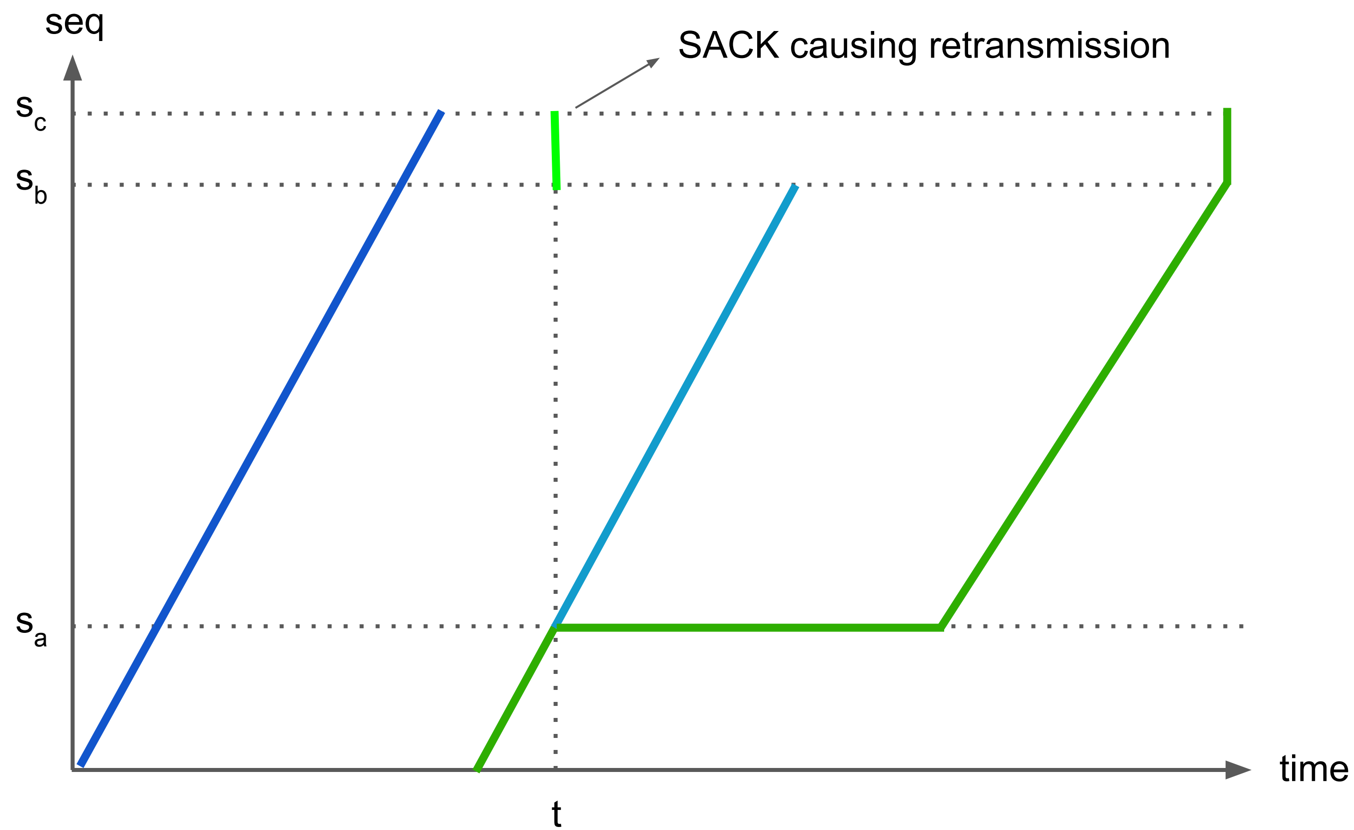
Time-sequence plot with losses: SACK for s_b:s_c causes retransmission of s_a:s_b

What’s in a trace: QUIC edition

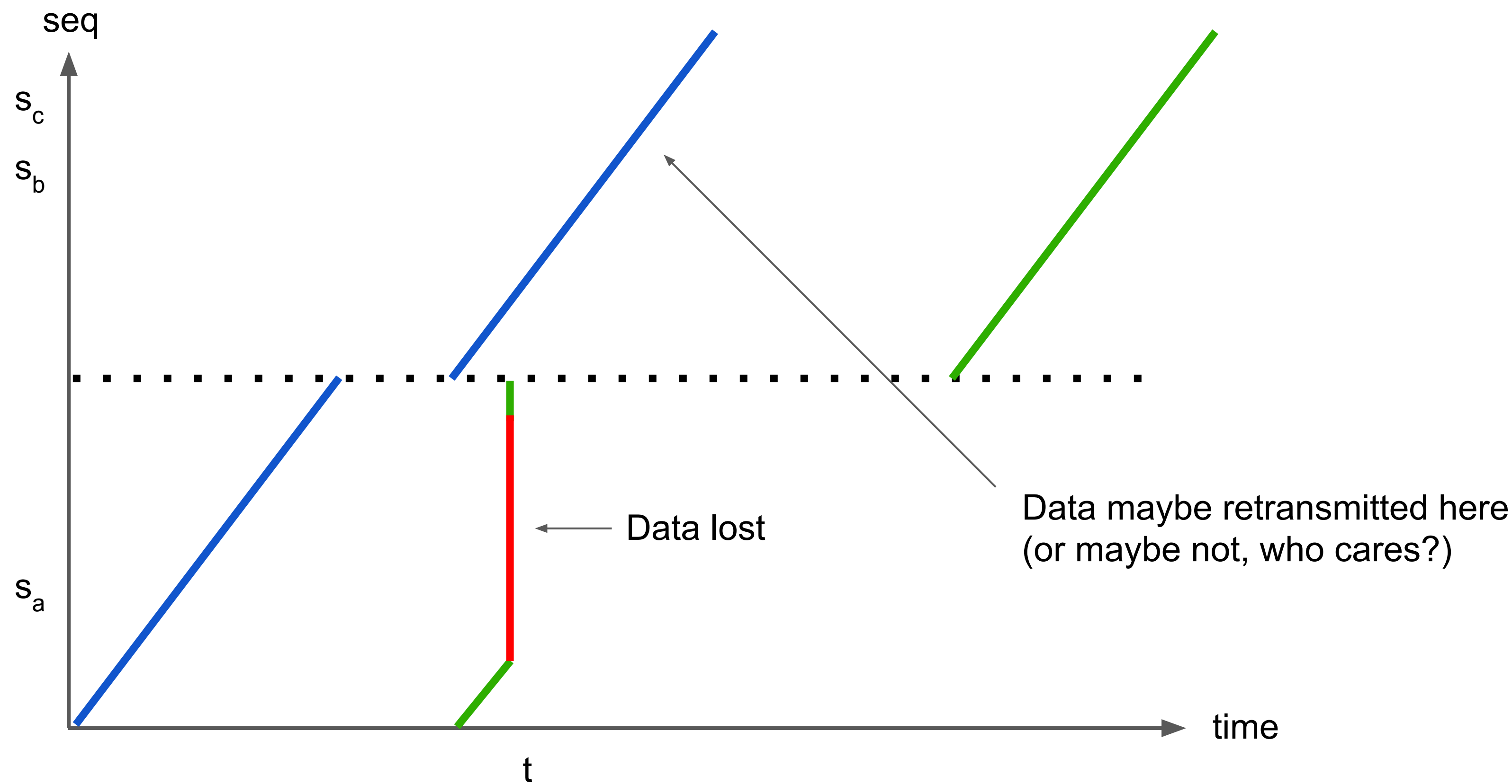
- QUIC packets do not have a sequence number
- How do we replace them?
- General idea: keep a counter of all bytes sent.
- Provides same graph as TCP would if there are no losses

What’s in a trace: QUIC edition

- Byte counter provides a TCP-like time sequence plot when there are no losses.
- Two approaches to showing a retransmission.
- Approach #1: map retransmissions to original stream data
  - Matches TCP worldview closely
  - Doesn’t match what QUIC does; retransmissions are of stream data, not of individual packets
  - If some data is never retransmitted, TCP renderers get confused
- Approach #2: assume retransmissions are just new data
  - Matches what QUIC does closely
  - Doesn’t look anything like TCP



Approach #1 (repeated diagram)



Approach #2

Rich data logging

- In traditional TCP tracing flow, the trace is derived from the TCP headers on the wire
- For QUIC, this is still possible (provided decryption keys are available), but suboptimal
- Logging happens within the endpoint
- The endpoint can export the exact values it makes decisions about sending

Rich data logging: variables to log

- Congestion window
- RTT measurement (min RTT, average RTT, etc)
- ssthresh
- Frames sent in the packet
- Flow control windows
- Bytes in flight
- Pacing rate

Rich data logging: success stories

Friday, September 25, 2015

Thanks Google for Open Source TCP Fix!

The Google transport networking crew (QUIC, TCP, etc..) deserve a shout out for identifying and fixing a nearly decade old Linux kernel TCP bug that I think will have an outsized impact on performance and efficiency for the Internet.

Their patch addresses a problem with cubic congestion control, which is the default algorithm on many Linux distributions. The problem can be roughly summarized as the controller mistakenly characterizing the lack of congestion reports over a quiescent period as positive evidence that the network is not congested and therefore it should send at a faster rate when sending resumes. When put like this, its obvious that an endpoint that is not moving any traffic cannot use the lack of errors as information in its feedback loop.

Firefox Networking @mcmanusducksong

View my complete profile

Blog Archive

- ▶ 2018 (1)
- ▶ 2017 (1)
- ▶ 2016 (1)
- ▼ 2015 (5)

Success story: CUBIC Quiescence bug, discovered thanks to CWND logging

Success stories: QUIC BBRv1

- BBR has rich internal state (10+ variables)
- During the development of QUIC implementation, we actively exported a lot of those variables per packet
- BBR relies heavily on app-limited markings, which are unobvious from the trace
- TCP BBRv1 also exported BBR mode for all development simulations