Advanced Computer Networking (ACN)

IN2097 – WiSe 2024–2025

Prof. Dr.-Ing. Georg Carle, Sebastian Gallenmüller

Christian Dietze, Max Helm, Benedikt Jaeger,
Marcel Kempf, Jihye Kim, Patrick Sattler

Chair of Network Architectures and Services
School of Computation, Information, and Technology
Technical University of Munich

Transport Layer Protocols

- TCP
    - Basics
    - Flow Control
    - Congestion Control
- UDP
- SCTP
- QUIC
    - QUIC Features
    - IETF QUIC
    - Analysis
    - Applications
- Bibliography

Basics

What is TCP?

- Short for Transmission Control Protocol
- Defined in RFC 793 [1] and many more RFCs
- Connection-oriented service
- In-sequence delivery of byte stream → Stream-oriented
- Reliability properties
  - Bit error detection
  - TSDU (Transport Service Data Unit) loss detection and retransmission
- Provides Flow Control (sender will not overwhelm receiver)
- Provides Congestion Control (sender will not overwhelm network)

When is it used?

- HTTP
- FTP
- SMTP / POP3 / IMAP
- SSL / TLS
- SSH, BGP, Backups, . . .

Properties

Point-to-Point

- One sender, one receiver

Reliable

- Everything that was sent will be received at some point in time

In-Order

- Sending order is receiving order

Stream-oriented

- Data is one continuous stream, no message boundaries
- Example:
  - `send("Hello"); send("World");`
  - `recv()`
    "HelloWorld", "Hello", "", "Hell"
  - not possible:
    "World", "HeWorld"

Connection-oriented

- Handshakes, holding state on both sides, tear-downs
- 3-way handshake
  SYN → SYN / ACK → ACK
  FIN → FIN / ACK → ACK

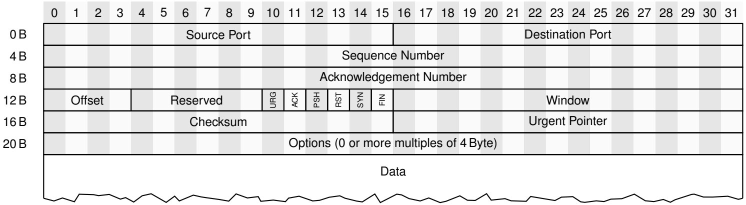
Flow controlled

- Sender can only send as much as the receiver can utilize

Congestion controlled

- Sender throttles bandwidth to not overwhelm network

Basics
TCP Header



0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

0 B Source Port | Destination Port

4 B Sequence Number

8 B Acknowledgement Number

12 B Offset | Reserved | URG | ACK | PSH | RST | SYN | FIN | Window

16 B Checksum | Urgent Pointer

20 B Options (0 or more multiples of 4 Byte)

Data

- Source Port: Identifier for sending application
- Destination Port: Identifier for receiving application
- Sequence Number: Identifier for segment, byte sequence number of first byte of the segment
- Acknowledgment Number: Next expected sequence number
- Offset: Offset to start of payload (header length including options)
- Reserved: Reserved for future use
- Window: Size of receiver window (buffer size of receiver), used for Flow Control
- Flags:
  - URG: Urgent pointer is set
  - ACK: Acknowledgment is set
  - PSH: Push, OS should not buffer
  - RST: Reset connection (instant termination)
  - SYN: Synchronize using handshake packets
  - FIN: Finish, start to tear-down connection
- Checksum: 16 bit one's complement of the one's complement sum of all 16-bit words in TCP pseudo-header and payload
- Urgent Pointer: Points to urgent data
- Options: Optional extensions

Basics

Basic Operation

Sender:

- Sends segments
- Expects acknowledgments

How does the sender know if a segment is lost?

- ACK is not received within a certain time interval
  - Timeout occurs
  - Sender retransmits segment
  - Takes at least timeout value + 1 RTT until lost segment is acknowledged
- Multiple duplicate ACKs arrive
  - Can be caused by reordering or packet loss
  - Segments arrived out-of-order, or at least one segment was lost
  - 3 duplicate ACKs $\Rightarrow$ Sender performs Fast Retransmit
  - Remark: 3 duplicate ACKs $\neq$ 3 lost segments
  - One or more lost segments always cause duplicate ACKs as long as there are subsequent segments

TCP Round Trip and Timeout

How to set the TCP timeout value?

- longer than RTT, but RTT varies
- too short: premature timeout
    - unnecessary retransmissions (Spurious Retransmission)
- too long: slow reaction to segment loss

How to estimate RTT

TCP Round Trip and Timeout

How to set the TCP timeout value?

- longer than RTT, but RTT varies
- too short: premature timeout
  - unnecessary retransmissions (Spurious Retransmission)
- too long: slow reaction to segment loss

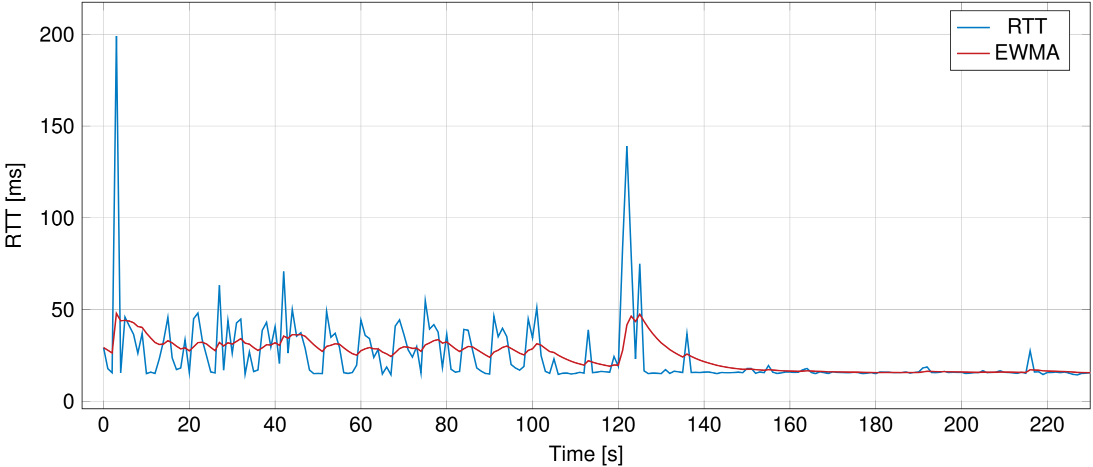
How to estimate RTT

- SampleRTT: measured time from segment transmission until ACK receipt
  - ignore retransmissions
- SampleRTT will vary, want estimated RTT “smoother”
  - average several recent measurements, not just current SampleRTT

$$\mathit{EstimatedRTT} = (1 - \alpha) \cdot \mathit{EstimatedRTT} + \alpha \cdot \mathit{SampleRTT}$$

- Called: Exponential weighted moving average (EWMA)
- influence of past sample decreases
- more weight on recent samples than on older samples
- typical value: $\alpha = 0.125$

Exponential weighted moving average (EWMA)



Basics

TCP Options

Maximum Segment Size Option (RFC 6691)

- Announce MSS during handshake: accept no segments larger than MSS
- Can be completely independently in each direction
- MSS counts only data octets in the segment, it does not count the TCP / IP header.

TCP Timestamp Option (RFC 7323)

- Prevents ambiguity of ACKs (is the ACK from the original packet or the retransmission?)
- Sender and receiver have a (virtual) "timestamp clock"
- Append two "timestamps" to each sent TCP segment:
  - Timestamp Value (TSVal): current "timestamp" when the packet is sent
  - Timestamp Echo Reply (TSecr): latest TSVal received before sending the packet
- On receive compute: TSecr − current timestamp

Selective Acknowledgment Options (RFC 2018)

- TCP only provides feedback about the next expected segment
- What if each second segment gets lost? → many RTTs to retransmit everything
- Goal: provide more information about received / missing segments

TCP Window Scale Option (RFC 7323)

- 2 B windows size field → at most about 65 kB receive buffer
- Scale the announced window by a factor (shift the window)

Flow Control

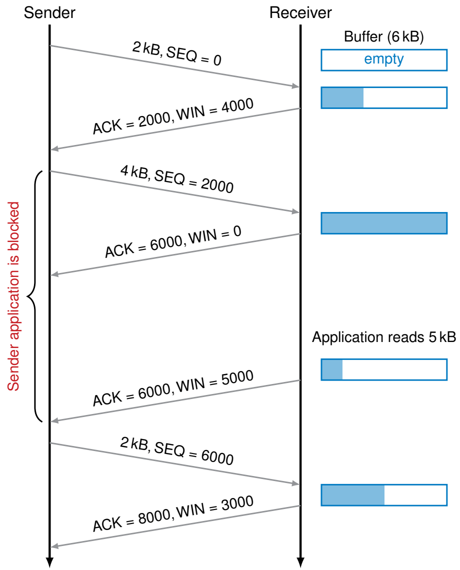
What is Flow Control?

- Receiver may be a resource limited device
- OS kernel buffers segments for applications to process
- Buffer is of limited (maybe small) size
- If the buffer is full: incoming segments are dropped
- Flow Control sets a maximum of data the sender is allowed to send
- Implemented by using the window field in the header
- Used in order to avoid overwhelming of receiver buffer

Flow Control

What is Flow Control?

- Receiver may be a resource limited device
- OS kernel buffers segments for applications to process
- Buffer is of limited (maybe small) size
- If the buffer is full: incoming segments are dropped
- Flow Control sets a maximum of data the sender is allowed to send
- Implemented by using the window field in the header
- Used in order to avoid overwhelming of receiver buffer



Congestion Control

Principles of Congestion Control

Definition:

- Informally: “Too many sources sending too much data too fast for the network to handle”
- Different from flow control (which handles overload at the recipient)

Manifestations:

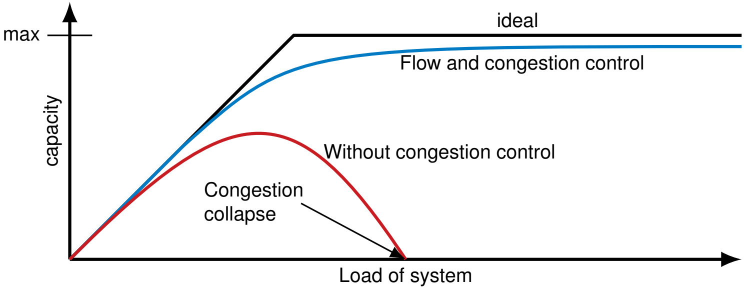
- Lost packets (buffer overflow at routers)
- Long delays (router buffer queues fill up)

Congestion Control

Goals and problems

What do we hope for?

- Reasonable behavior in case of high load of network
- Without controlling amount of outgoing data, capacity may drop dramatically because of congestion collapse
- Fair resource sharing
- Criteria: effective, simple, robust, end-host driven

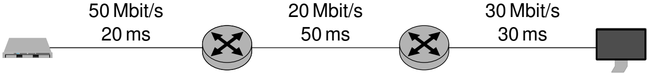


Congestion Control

Operation Point – Terms and Definitions

Every path can be described with two parameters:

- Round-trip propagation delay: RTprop = $\sum_i$ RTprop$_i$ with RTprop$_i$ being the delay of link i

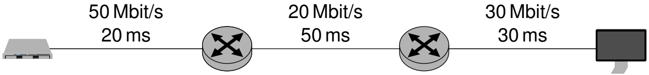


Operation Point – Terms and Definitions

Every path can be described with two parameters:

- Round-trip propagation delay: RTprop = $\sum_i$ RTprop$_i$ with RTprop$_i$ being the delay of link i
- Bottleneck bandwidth: BtlBw = min(BtlBw$_i$) with BtlBw$_i$ being the bandwidth of link i

$$RTprop = 20\text{ ms} + 50\text{ ms} + 30\text{ ms} = 100\text{ ms}$$



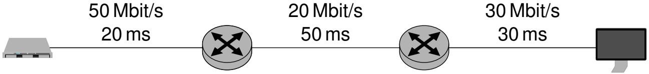
Congestion Control

Operation Point – Terms and Definitions

Every path can be described with two parameters:

- Round-trip propagation delay: RTprop = $\sum_i$ RTprop$_i$ with RTprop$_i$ being the delay of link i
- Bottleneck bandwidth: BtlBw = min(BtlBw$_i$) with BtlBw$_i$ being the bandwidth of link i
- BtlneckBufSize: buffer size at the bottleneck link
- Amount inflight: data which is sent but not acknowledged

$$
\begin{aligned}
RTprop &= 20\,\text{ms} + 50\,\text{ms} + 30\,\text{ms} = 100\,\text{ms} \\
BtlBw &= min(\,50\,\text{Mbit}/\text{s}, 20\,\text{Mbit}/\text{s}, 30\,\text{Mbit}/\text{s}\,) = 20\,\text{Mbit}/\text{s} \\
BDP &= 100\,\text{ms} \cdot 20\,\text{Mbit}/\text{s} = 2000\,\text{kbit}
\end{aligned}
$$

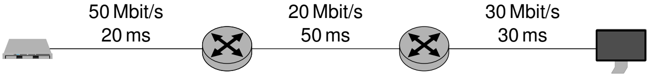


Operation Point – Terms and Definitions

Every path can be described with two parameters:

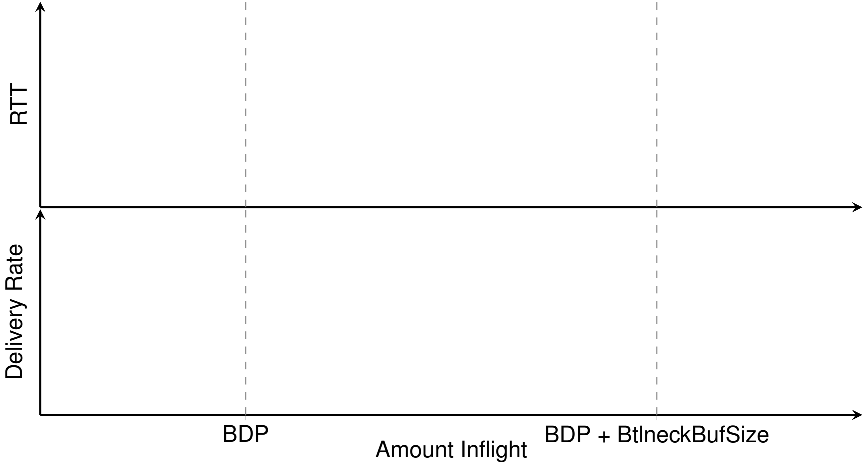
- Round-trip propagation delay: RTprop = $\sum_i$ RTprop$_i$ with RTprop$_i$ being the delay of link *i*
- Bottleneck bandwidth: BtlBw = *min*(BtlBw$_i$) with BtlBw$_i$ being the bandwidth of link *i*
- BtlneckBufSize: buffer size at the bottleneck link
- Amount inflight: data which is sent but not acknowledged
- Bandwidth-delay product: BDP = RTprop · BtlBw
  "How much data can fit on a link with bandwidth BtlBw and propagation delay RTprop"

$$\begin{aligned} RTprop &= 20\,\text{ms} + 50\,\text{ms} + 30\,\text{ms} = 100\,\text{ms} \\ BtlBw &= min(\,50\,\text{Mbit}/\text{s}, 20\,\text{Mbit}/\text{s}, 30\,\text{Mbit}/\text{s}\,) = 20\,\text{Mbit}/\text{s} \\ BDP &= 100\,\text{ms} \cdot 20\,\text{Mbit}/\text{s} = 2000\,\text{kbit} \end{aligned}$$

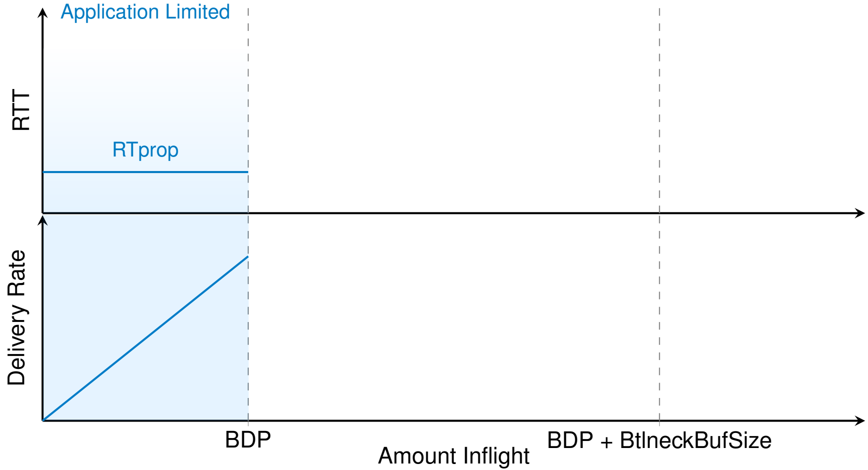


Congestion Control

Operation Point

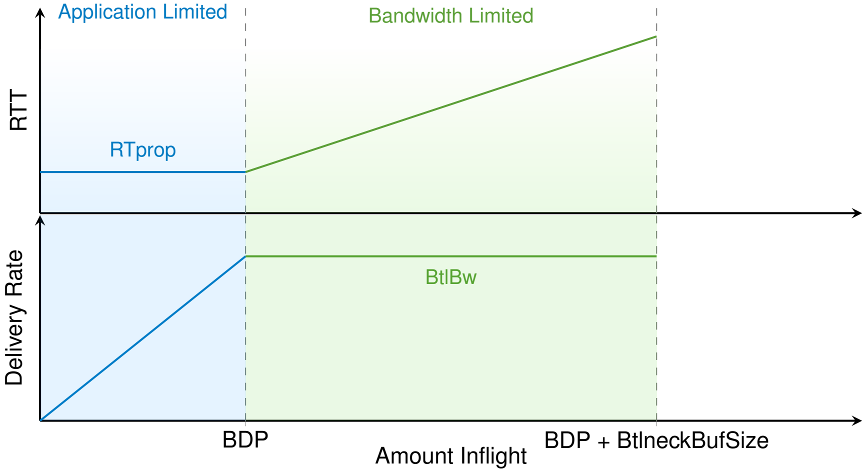


Congestion Control
Operation Point



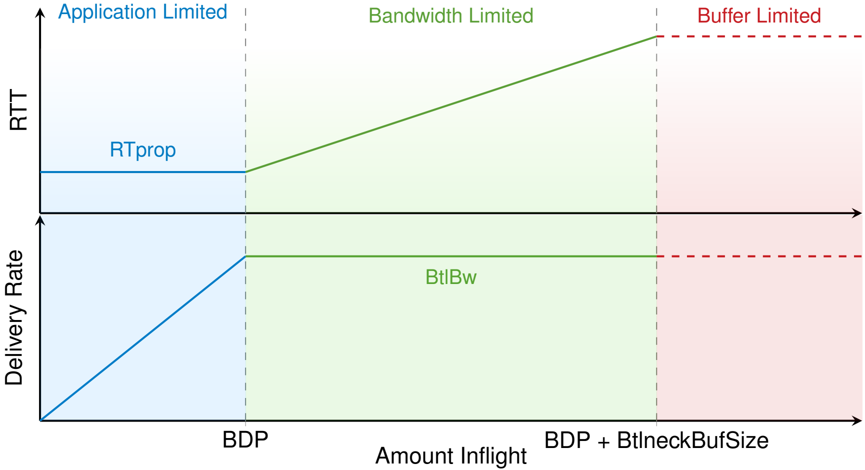
Congestion Control

Operation Point



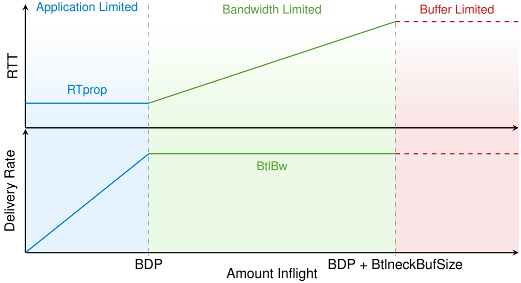
Congestion Control

Operation Point



Congestion Control
Operation Point – Summary

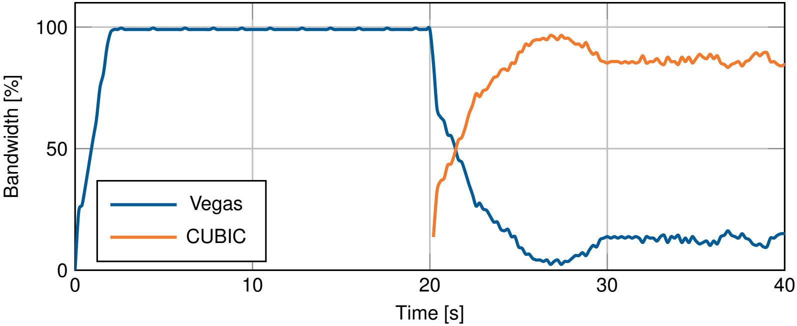
- Application Limited if Inflight $<$ BDP
  - link underutilization
  - low latency
- Bandwidth Limited if BDP $<$ Inflight $<$ BDP + BtlneckBufSize
  - full link utilization
  - buffer starts filling
- Buffer Limited if BDP + BtlneckBufSize $<$ Inflight
  - packet loss leads to lower goodput (retransmission consume bandwidth)
  - unpredictable latency due to retransmissions



Is TCP Fair?

Problem:

- Multiple TCP flows use the same path
- Do all of them get an equal share of the bandwidth?
- Multiple different congestion control algorithms may be used!



Congestion Control

Measuring Fairness

Metrics often used for assessing numerically the fairness between n flows with x_i the bandwidth of flow i:

- The product measure:

$$\prod_i x_i$$

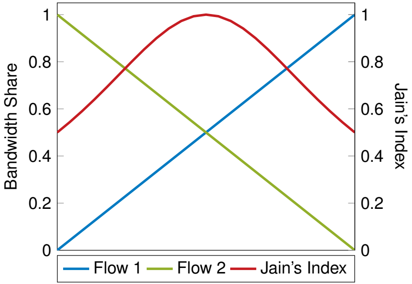
- Epsilon-fairness: A rate allocation is defined as epsilon-fair if

$$\frac{\min_i x_i}{\max_i x_i} \geq 1 - \epsilon$$

- Jain's fairness index:

$$\frac{\left(\sum_i x_i\right)^2}{n \cdot \sum_i x_i^2} \qquad \in \left[\frac{1}{n}, 1\right]$$

- Returns a value between 0 and 1
- Scale free
- Arbitrary number of flows
- Is $\frac{k}{n}$ if there are k flows are perfectly fair while the other $n - k$ shares are 0



Congestion Control Algorithms

How does TCP regulate the sending rate?

- Only have a well-defined number of bytes (/ segments) in the network, which have not yet been acknowledged
- This number of bytes is called the Congestion Window

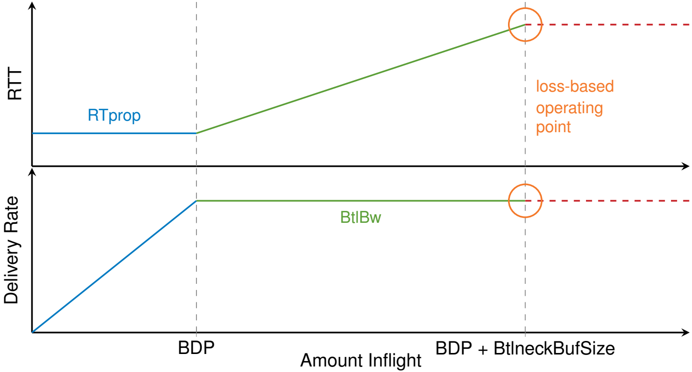
How to process available information to modify the congestion window size?

- There are a lot of algorithms
- Different classes:
  - loss-based
  - delay-based
  - model-based
  - hybrid approach
  - . . .
- Most popular / interesting:
  - TCP Tahoe (slow start, congestion avoidance)
  - TCP Reno (fast retransmit, fast recovery, today: TCP New Reno)
  - TCP Vegas
  - TCP Cubic (current default in the Linux kernel)
  - TCP BBR (new, proposed by Google)

Loss-based Congestion Control

How do loss-based algorithms detect congestion?

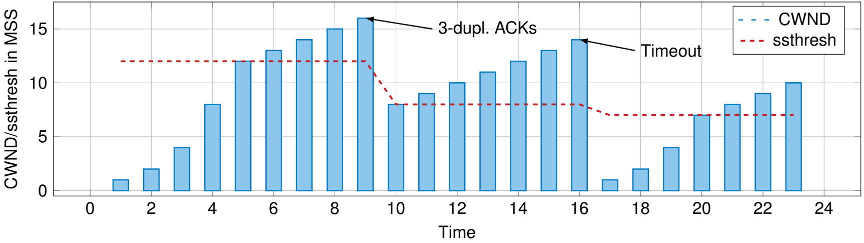
- Assumption: Packet loss only happens due to congestion
- No packet loss → increase congestion window
- Packet loss → decrease congestion window
- Advantages: robust, reliable, efficient
- Disadvantages: buffers are kept full → high latency, performance drop on lossy links
- Examples: Reno, Bic, Cubic



TCP Reno
Basics of the Algorithm

Theory behind Reno:

- Every packet loss is induced by a network overload
  - Therefore, TCP senders should reduce data rate
  - However, think about lossy links!
- AIMD strategy: Additive Increase Multiplicative Decrease
- Two modes of operation:
  - Slow Start: Exponential growth of congestion window
  - Congestion Avoidance: Linear growth of congestion window



TCP Reno

Algorithm

Variables:

- CWND: Congestion Window, limits the amount of inflight data
- ssthresh: Slow Start threshold

Slow Start:

- For every acknowledged MSS, increase CWND by 1 MSS
- Use this mode, if CWND $<$ ssthresh

Congestion Avoidance:

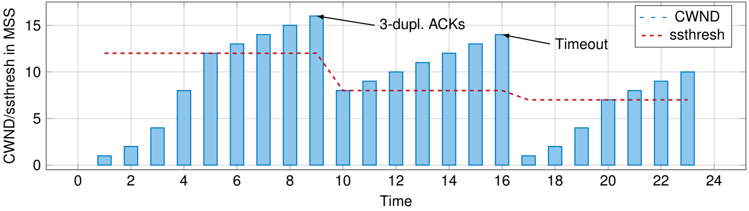
- For every acknowledged MSS, increase CWND by 1/CWND effectively increase CWND by 1 MSS each RTT → additive increase
- Use this mode, if CWND $\geq$ ssthresh

Reception of 3 duplicated acknowledgments:

- Set ssthresh to CWND/2
- Set CWND to ssthresh (Fast Recovery) → multiplicative decrease

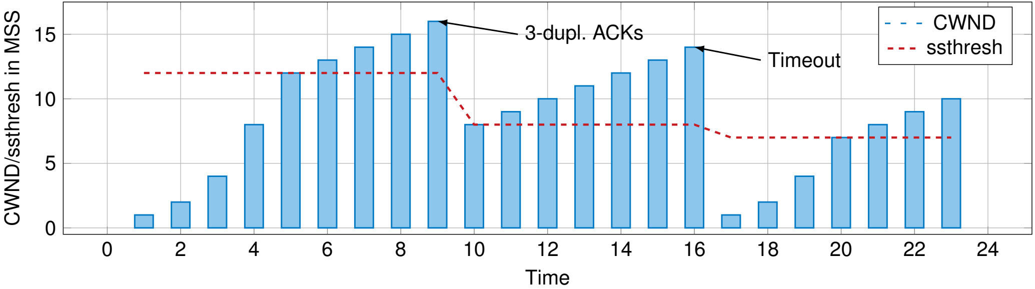
Acknowledgement timeout:

- Set ssthresh to CWND/2
- Set CWND to 1 MSS
- Restart with Slow Start



Problems of TCP Reno

- Low performance on lossy links
- Buffers are filled
- Increase depends on the RTT → slow growth on long distance links
- Has problems fully utilizing large BDP links



TCP Cubic
Basics of the Algorithm

Theory behind Cubic:

- Published in 2008 [2]
- RFC since February 2018 [3] (informational) and August 2023 [4] (standards track)
- Default congestion control algorithm in Linux since kernel 2.6.19 (Nov. 2006)
- Same as Reno: Packet losses are considered to indicate a network overload
- But: Scaling should be different
- Maximum usable bandwidth is estimated
- That bandwidth should be used, and if nothing is lost, higher bandwidth is explored

TCP Cubic
Formulas

- W_{cubic}: Congestion Window according to TCP Cubic
- W_{max}: window size at which last packet loss occurred
- t: time since the last packet loss
- β: window decrease constant for multiplicative decrease of window
- C: Cubic parameter

$$W_{cubic}(t) = C \cdot (t - K)^3 + W_{max} \tag{1}$$

$$K = \sqrt[3]{W_{max} \cdot (1 - \beta_{cubic})/C} \tag{2}$$

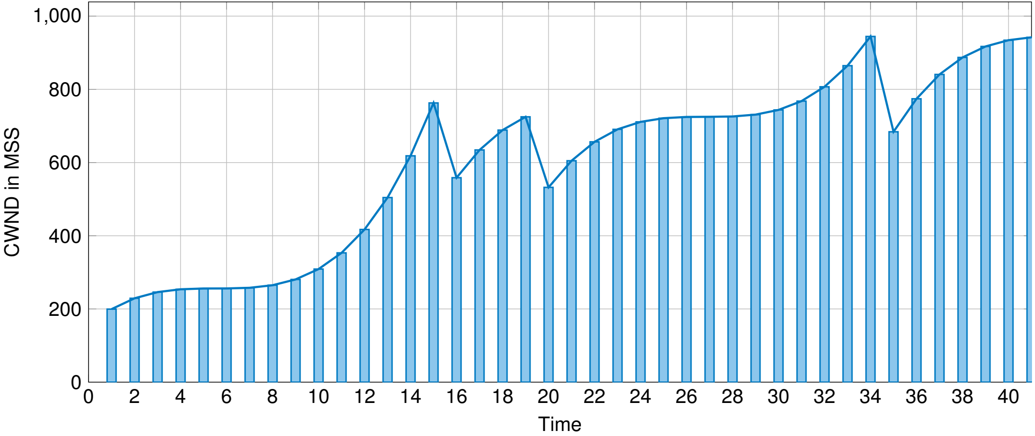
$$C = 0.4 \tag{3}$$

$$\beta = 0.7 \tag{4}$$

Things to note:

- Congestion window is not halved for every packet loss ($\beta = 0.7$)
- Congestion window growth is modeled after a cubic function with plateau W_{cubic}
- Converges fast (concave growth) towards the bandwidth of the last packet loss W_{cubic} (estimated network maximum)
- If this is fine, higher bandwidth is explored (convex growth)

TCP Cubic
Example



C and β_{cubic} were tweaked for demonstrative purposes.

TCP Cubic
Result

Advantages

- CWND growth is independent of the RTT
- Scalable to high BDP networks
- More resilient against single stochastic packet loss than Reno

Disadvatages

- Buffers are filled faster (cubic growth function)
- Buffers are kept full (reduced only by 30 % after packet loss)

Delay-based Congestion Control

Basics

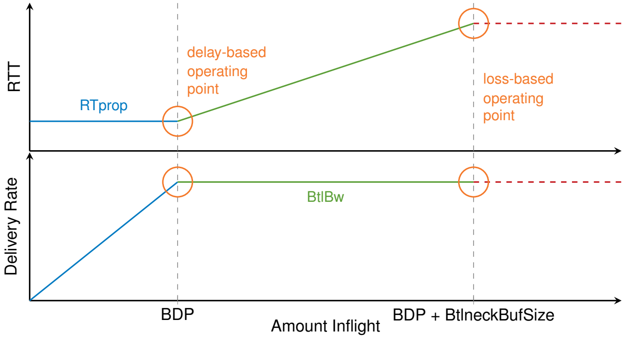
- Use delay to detect congestion
- Increase in RTT → a buffer is filling up somewhere

Advantages

- Less restrained by random packet loss
- Early congestion detection
- High throughput with low latency

Disadvantages

- One loss-based flow cancels all advantages
- Poor performance against loss-based flows



TCP Vegas
Basics of the Algorithm

Theory behind Vegas:

- Presented in 1994 [5]
- Reno relies on losses to detect network congestion
- At that point something already has gone wrong
- Vegas tries to detect that congestion is about to happen, and then reduces data rate
- RTTs are continuously measured
- RTT increases due to queuing effects
- If RTT increases, the network is considered to become more congested
- If RTT decreases, not all available bandwidth may be used
- AIAD strategy: Addititive Increase Addititive Decrease

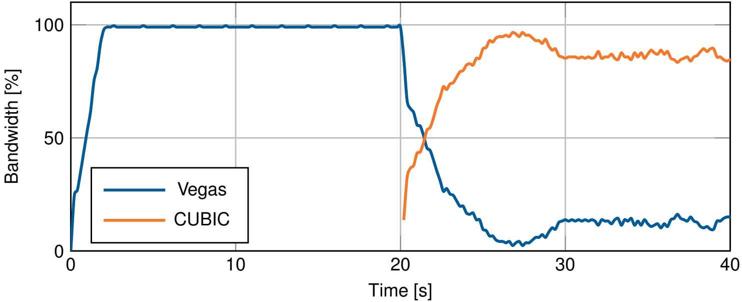
Formula:

$$\Delta = \text{CWND} \cdot \frac{\text{RTT} - \text{RTT}_{\text{min}}}{\text{RTT}}$$

Each RTT:

- If $\Delta > \beta$ (Linux: 4): window size is decreased by 1 MSS
- If $\Delta < \alpha$ (Linux: 2): window size is increased by 1 MSS
- If $\alpha < \Delta < \beta$: Steady state → no modifications

TCP Vegas
Delay-based vs Loss-based



- Why use delay-based algorithms at all?
  - background applications like downloading updates
  - e.g. LEDBAT (Low Extra Delay Background Transport) [6]
- Use hybrid approach for better performance when competing with loss-based algorithms
- For example TCP Illinois:
  - Packet-loss prescribes if CWND is increased or decreased
  - Delay determines the quantity of the change
    - low delay: faster increase, slower decrease
    - high delay: slower increase, larger decrease

TCP BBR
Basics of the Algorithm

Theory behind BBR:

- Presented by Google in 2016 [7]
- BBR: Bottleneck Bandwidth and RTT
- Aims for the same operation point as delay-based algorithms
- Maximum bandwidth is determined by a single bottleneck
- RTT increases due to queuing

Usage

- Available in Linux since kernel v4.9
- Used on Google's and YouTube's servers
- Used in Google's B4 backbone network
- *"BBR's throughput is consistently 2 to 25 times greater than CUBIC's"* [7]
- *"BBR yielded 4 percent higher network throughput [...] BBR also keeps network queues shorter, reducing round-trip time by 33 percent"*[1]

[1] https://cloudplatform.googleblog.com/2017/07/TCP-BBR-congestion-control-comes-to-GCP-your-Internet-just-got-faster.html

TCP BBR
Theory: ACK-clocking and Pacing

ACK-clocking

- Used by Reno, Cubic, Vegas, . . .
- CWND limits the inflight datd but sending rate is not limited
- Arrival rate of ACKs determines the sending rate
- Traffic bursts can create queues even if link is not utilized
- Slow Start, retransmissions, ACK compression can cause bursts

Pacing

- Goal: evenly space the transmission the packets of a window across an entire RTT
- Linux Kernel $<$ 4.13: requires FQ queuing discipline on outgoing interface
  ```
  tc qdisc add dev eth0 root fq pacing
  ```
- Linux Kernel $\geq$ 4.13: pacing implemented in the Kernel

TCP BBR
In practice

Goals

- Keep 1 BDP of data inflight → full link utilization and no queuing delay
- Send with the bottleneck bandwidth → no queue can build up

Implementation

- Continuously monitors the network to find the minimal RTT and maximum bandwidth
- Problem: theses parameters cannot be measured at once
    - RTprop can only be measured if the buffers are empty
    - BtlBw can only be measured while the link is fully utilized and a queue starts growing
    - Solution: alternating measurements
- Use filters to record those values against sliding windows
- Requires pacing to match sending rate to the bottleneck bandwidth

Internal BBR values:

- RTprop
- BtlBw
- PacingGain
- WindowGain

Four phases:

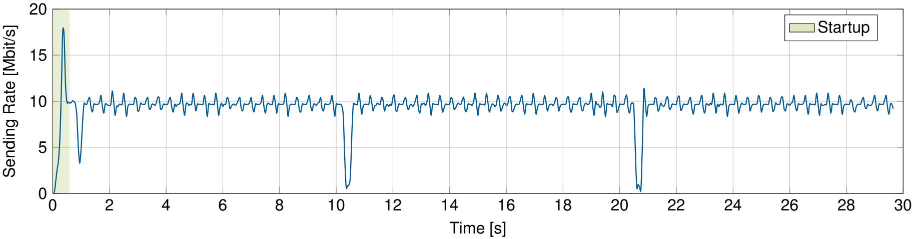
- Startup
- Drain
- Probe Bandwidth
- Probe Round-Trip Time

TCP BBR

Startup and Drain

Startup

- Similar to Slow Start → double sending rate each RTT
- Sending Rate = BtlBw · 2.8853 $\left(\frac{2}{\ln 2} \approx 2.8853\right)$
- Stop after three consecutive RTTs with less than 25 % in delivery rate increase
- Finds BtlBw in $\log_2(\mathrm{BDP})$ RTTs
- Can create queue up to 2 BDP



TCP BBR

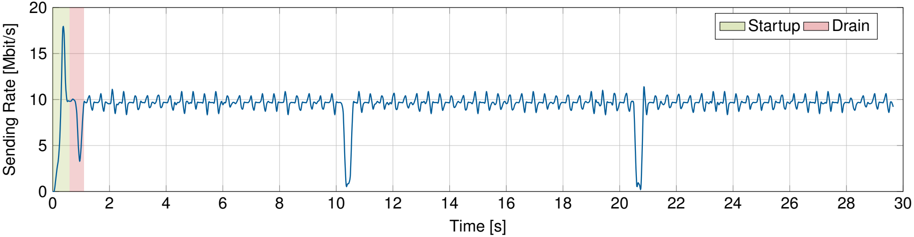
Startup and Drain

Startup

- Similar to Slow Start → double sending rate each RTT
- Sending Rate = BtlBw · 2.8853 $\left(\frac{2}{\ln 2} \approx 2.8853\right)$
- Stop after three consecutive RTTs with less than 25 % in delivery rate increase
- Finds BtlBw in $\log_2(\text{BDP})$ RTTs
- Can create queue up to 2 BDP

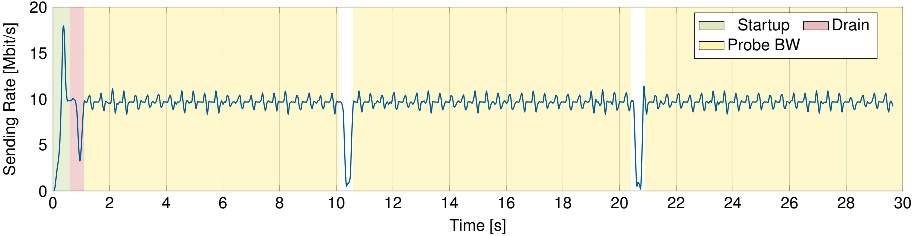
Drain

- Goal: Remove the queue created during Startup
- Sending Rate = BtlBw · 0.3465 $\left(\frac{\ln 2}{2} \approx 0.3465\right)$
- Leave Drain when data in flight matches estimated BDP



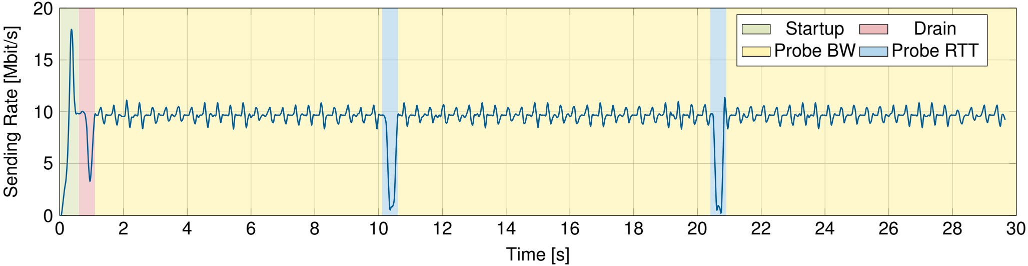
TCP BBR
Probe Bandwidth

- Periodically probe for more bandwidth
- BtlBw is estimated using a max filter of length about ten estimated RTTs
- Sending Rate = BtlBw · PacingGain, with PacingGain in `[1.25, 0.75, 1, 1, 1, 1, 1, 1]`
- Each step takes about one RTT
- If no bandwidth is available: sending rate is reduced afterwards to remove queue
- If bandwidth is available: BtlBw is updated and thus sending rate increases



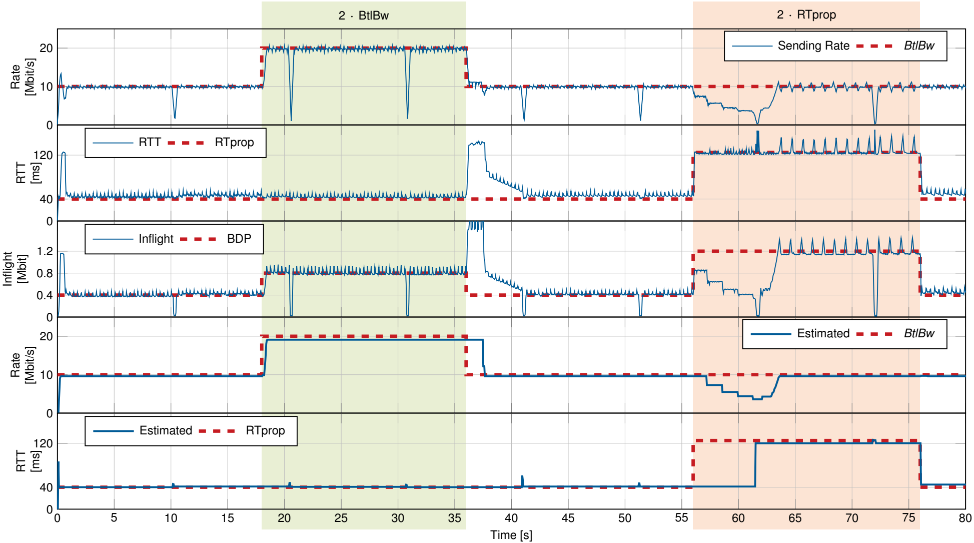
TCP BBR
Probe RTT

- RTprop is estimated using a min filter of length 10 s
- If no new RTprop value is measured during this interval BBR enters Probe RTT
  → ≈ 10 s interval between two Probe RTT phases
- To ensure that all queues are empty, BBR reduces inflight to 4 segments for 200 ms + RTT
- Problem: low delivery rates during Probe RTT → performance drop
- Multiple BBR flows have to synchronize their Probe RTT phases to reach fairness



TCP BBR

BBR Single Flow



TCP BBR
Strengths of BBR

- Robustness against random packet loss
- Low delay
- High bandwidth usage
- Close to the optimal operation point
- Does not starve when competing with other algorithms

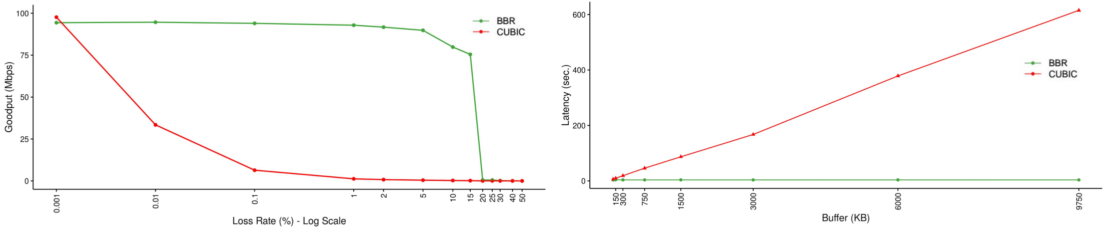


Figure 1: Figures from [7].

TCP BBR
Problems with BBR [8], [9]

- Numerous BBR flows fail to keep the buffer empty
  - Flows probe alternating for more bandwidth
  - Sum of the bandwidth estimations is larger than actual bandwidth
  - Flows create a persistent queue of size ≈ 1.5 BDP
- High number of retransmissions in networks with shallow buffers
  - If the buffer is smaller than the persistent queue → packet loss
  - BBR does not react to it
  - With small (shallow) buffers BBR can generate 20 % retransmissions
- RTT unfairness
  - BBR flows with larger RTT receive larger bandwidth shares than flows with lower RTT
  - With Reno and Cubic flows with lower RTT are favored

But:

- First version already shows promising results
- Still under active development: `https://groups.google.com/d/forum/bbr-dev`
- BBR v2 already announced

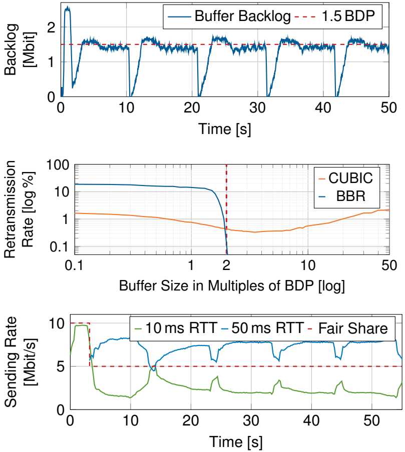


Figure 2: Source [8]

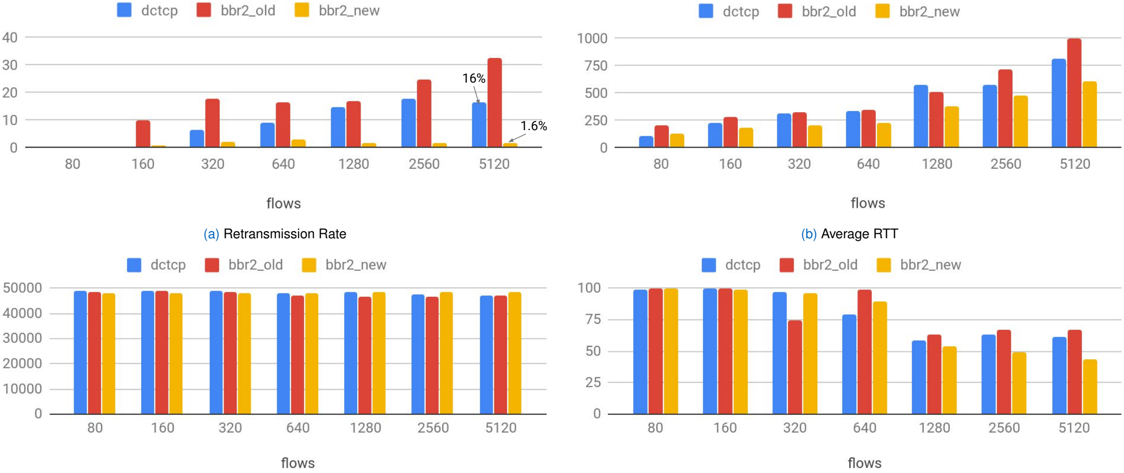
TCP BBR
BBR v2

BBR developers regularly present updates on BBR v2 on IETF meetings

Features:

- During Probe RTT, reduce cwnd to 50 % instead of 4 packets
- Consider detected packet loss for the model
- Incoporate protocol features like ECN
- Handle problems with ACK-aggregation
- Better coexistence with Reno / CUBIC
- Leave space for new entering flows

TCP BBR
BBR v2



(a) Retransmission Rate

(b) Average RTT

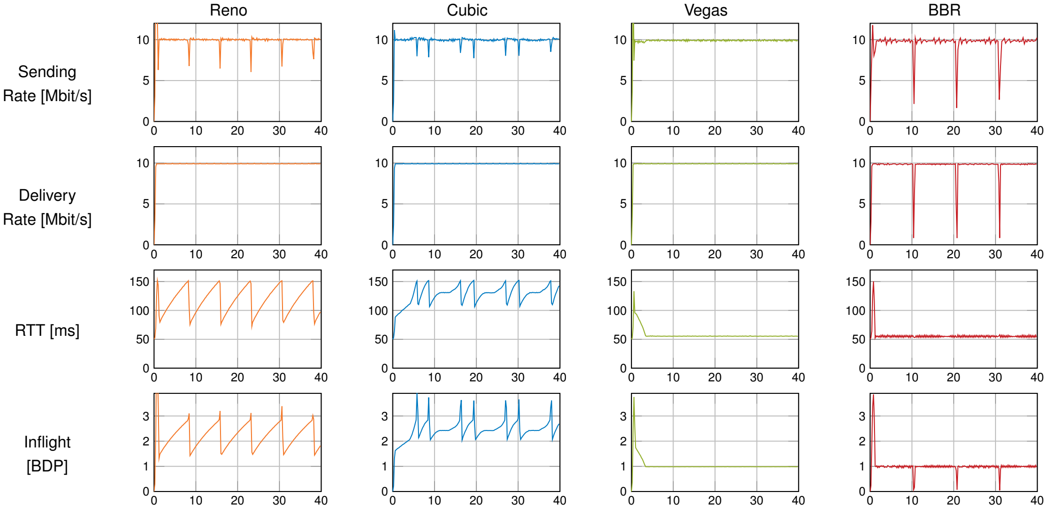
(c) Throughput

(d) Fairness (Jain’s Index)

Figure 3: Figures from [10]

Summary Congestion Control

Single flow, 10 Mbit/s bandwidth, 50 ms RTT, 2 BDP buffer size, flows run for 40 s, x-axis is time in seconds



Try it yourself: `https://gitlab.lrz.de/tcp-bbr/measurement-framework`

TCP Congestion Control and Linux

Loaded congestion control

-
  ```
  $ sysctl net.ipv4.tcp_congestion_control
  net.ipv4.tcp_congestion_control = cubic
  ```
- `cubic` (since version 2.6.19 - Nov. 2006)

Available congestion control

-
  ```
  $ sysctl net.ipv4.tcp_available_congestion_control
  net.ipv4.tcp_available_congestion_control = cubic reno
  ```

Implemented congestion control

-
  ```
  $ ls /lib/modules/‘uname -r‘/kernel/net/ipv4/ | grep tcp
  ```

Load BBR module

-
  ```
  modprobe tcp_bbr
  ```

-
  ```
  $ sysctl net.ipv4.tcp_available_congestion_control
  net.ipv4.tcp_available_congestion_control = cubic reno bbr
  ```

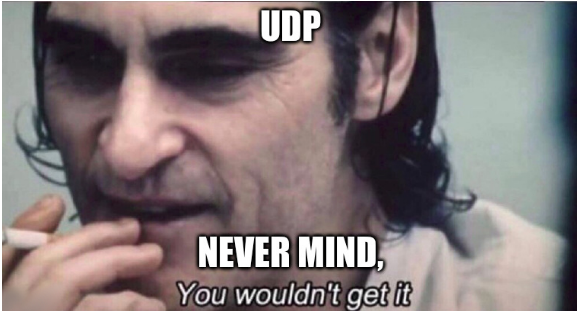
Set algorithm

-
  ```
  $ sysctl -w net.ipv4.tcp_congestion_control=bbr
  ```
- For BBR: Don’t forget to enable pacing for your interface if you have Kernel < 4.13
  ```
  $ tc qdisc add dev eth0 root fq pacing
  ```

UDP
User Datagram Protocol

What is UDP?

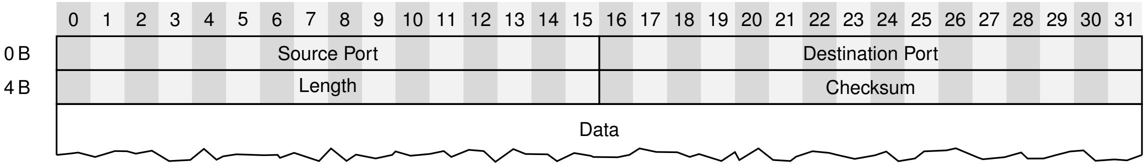
- Short for User Datagram Protocol
- Defined in RFC 768 [11] (only 3 pages long!)
- A connectionless transport protocol
- Bit error detection
- No flow or congestion control
- No reordering, loss detection or recovery
- Lightweight
- Easy to implement



When is it used?

- DNS queries
- Voice-over-IP (VoIP)
- Game server-client / client-client communication
- NTP (Network Time Protocol)
- . . .

UDP
UDP Header



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0 B	Source Port																Destination Port															
4 B	Length																Checksum															
	Data																															

- Source Port: Which application of the sending host sent the datagram
- Destination Port: Which application of the receiving host should receive the datagram
- Length: Length of the datagram (L4-PDU)
- Checksum: Checksum over IP pseudo header, UDP header, data

UDP
Idea behind UDP

What does it achieve?

- Multiplexing of multiple communication instances between two hosts
- Not much else – that is the point

Why use UDP, if it does not do much?

- Thin layer above IPv4 / IPv6
- Application retains a lot of control
- Suitable for time sensitive applications
  ⇒ no transport layer mechanims that may impair timing properties
- Occasional loss of datagrams tolerated or done by application
- Re-ordering of datagrams tolerated or done by application

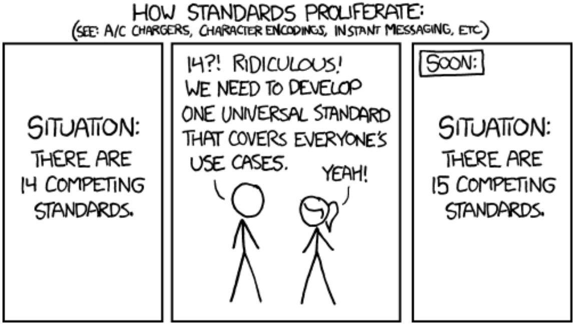
Example: Real-time video conferencing

- One frame is lost during transit: Nobody notices anyways
- If strict ordering was applied, retransmission would be needed
  - Delays the video for a whole RTT
  - Noticeable stuttering of the video

SCTP
Stream Control Transmission Protocol

Goals:

- Proposed in RFC 2960 (October 2000)
- Extended in RFC 4960 (September 2007)
- "Reliable transport protocol operating on top of a connectionless packet network such as IP"
- Combines advantages of TCP and UDP
- Multiple streams
- Supports of multi-homing


HOW STANDARDS PROLIFERATE:
(SEE: A/C CHARGERS, CHARACTER ENCODINGS, INSTANT MESSAGING, ETC.)
SITUATION: THERE ARE 14 COMPETING STANDARDS.
14?! RIDICULOUS! WE NEED TO DEVELOP ONE UNIVERSAL STANDARD THAT COVERS EVERYONE'S USE CASES.
YEAH!
SOON:
SITUATION: THERE ARE 15 COMPETING STANDARDS.

Sounds good, so why don't we use it everywhere?

- TCP was already established as the default transport layer protocol (network ossification)
- Poor support in operating systems and applications
- Many middleboxes (e.g. firewalls, NAT) do not work with SCTP → packets are discarded

QUIC
Introduction

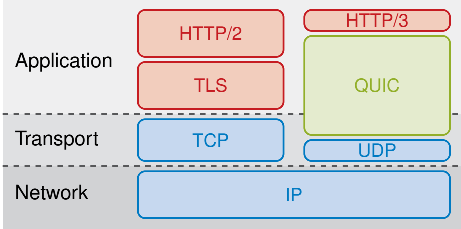


Figure 4: QUIC protocol stack adapted from [12]

What is QUIC?

- Originally Quick UDP Internet Connections, but not an acronym
- Developed around 2012 by Google, deployed in Google Chrome and Chromium [12], [13]
- A substitute for the TCP/TLS protocol stack, based on UDP
- 2016 - 2021 standardization by the IETF

Motivation and Goals

- Decrease handshake delay
- Get rid of head-of-line blocking
- Faster development cycles
- Middlebox resistance
- IP mobility

QUIC Features

Connection ID

- Used instead of the 5-tuple as identifier
- This allows to change IP and port

Stream Multiplexing

- Multiple streams within a connection
- Each stream provides a reliable bidirectional bytestream
- QUIC packet contains several frames
- QUIC packet can carry stream frames from multiple streams

Different Frame Types

- Control frames
- Data and acknowledgement frames

Flow Control

- Stream flow control
- Connection flow control

Congestion Control

- Currently Cubic
- BBR implementation in progress

Different Packet Types

- Version Negotiation Packet
- Initial Packet
- Retry Packet
- Handshake Packet

Encryption and Authentication

- Packets are always protected using TLS 1.3

Loss Detection and Re-ordering

- Retransmissions have different packet numbers → use Stream Offset for in order delivery
- More elaborated acknowledgement mechanism including selective and negative ACKs (SACKs and NACKs)

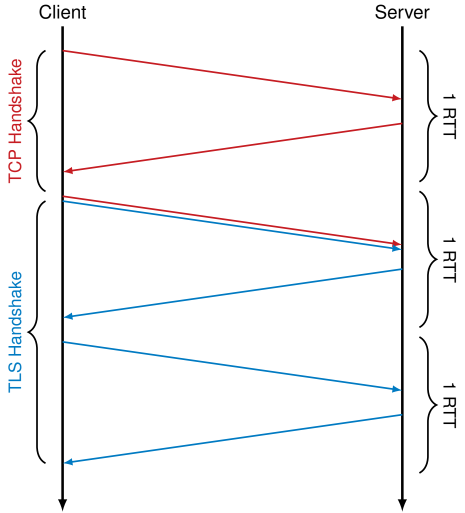
QUIC Features
Decrease Handshake Delay

Problem:

- TCP does a 3-way handshake
- TLS does at least 3-way handshake (or more...)
- Results in a lot of RTTs before data transmission
- Can in part be decreased using TCP Fast Open (but not widely deployed)

Solution:

- Introduce a 0-RTT and a 1-RTT handshake
- Merge the TCP and TLS component into one protocol
- Reuse old connections
- Client saves information about the server



QUIC Features
Get Rid of Head-of-Line Blocking

Problem:

- If one TCP segment is lost in transit, everything after that has to wait for delivery until successful retransmission (in-order property)
- Frequent goal: multiplexing multiple data streams over one TCP connection
- Example: Two videos get transmitted over one TCP connection
  - Server sends videos interleaved
  - One packet containing a part of video #1 gets lost
  - Following parts of video #2 cannot be processed, although they may already be present
  - Result: Video #2 has unnecessary quality impairments

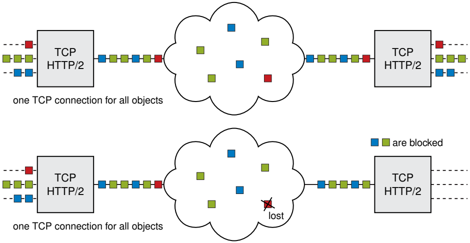

TCP
HTTP/2
one TCP connection for all objects
TCP
HTTP/2
🟦 🟩 are blocked
TCP
HTTP/2
one TCP connection for all objects
lost
TCP
HTTP/2

Figure 5: Adopted from QUIC: Next generation multiplexed transport over UDP

QUIC Features
Get Rid of Head-of-Line Blocking

Problem:

- If one TCP segment is lost in transit, everything after that has to wait for delivery until successful retransmission (in-order property)
- Frequent goal: multiplexing multiple data streams over one TCP connection
- Example: Two videos get transmitted over one TCP connection
  - Server sends videos interleaved
  - One packet containing a part of video #1 gets lost
  - Following parts of video #2 cannot be processed, although they may already be present
  - Result: Video #2 has unnecessary quality impairments

Solution:

- Protocol is aware of multiple streams
- Retransmission is done at stream-level, not connection-level

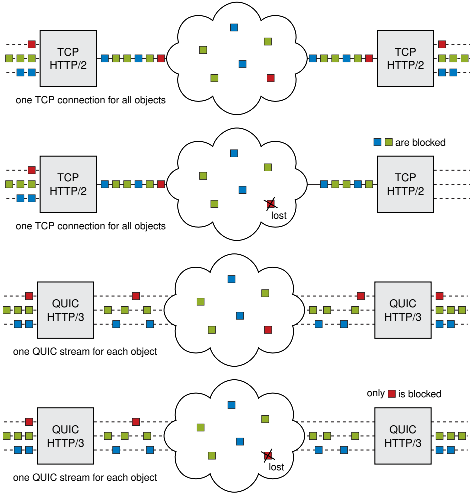

TCP HTTP/2
TCP HTTP/2
one TCP connection for all objects
🟦 🟩 are blocked
TCP HTTP/2
TCP HTTP/2
lost
one TCP connection for all objects
QUIC HTTP/3
QUIC HTTP/3
one QUIC stream for each object
only 🟥 is blocked
QUIC HTTP/3
QUIC HTTP/3
lost
one QUIC stream for each object

Figure 5: Adopted from QUIC: Next generation multiplexed transport over UDP

QUIC Features
Faster Development Cycles

Problem:

- TCP is implemented in the kernel
- ⇒ slow deployment of new mechanisms
  - Devices often don't get updated to newer kernel
  - Getting modifications of kernel protocol mechanisms is a slow process
  - Involves a lot of testing with a lot of different applications
  - Running big-scale experiments with TCP is very difficult

Solution:

- QUIC is based on UDP, and implemented in user space
- The kernel is not involved in the protocol itself
- Experiments with new protocol mechanisms are straightforward,
  as long as user-space is controlled by the application vendor

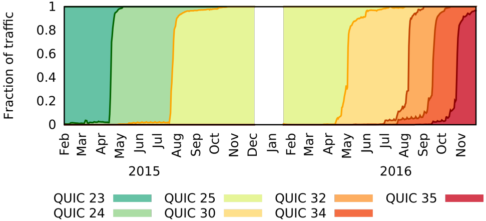


Figure 6: (Source: [12])

Note: In Dec 2015 Google disabled QUIC due to a vulnerability in the client code

QUIC Features
Middlebox Resistance

Why use UDP? Why not implement a new layer 4 protocol?
Problem:

- Middleboxes such as firewalls, “optimizers”, etc. exist
- In many cases, they make things worse
- May lead to obscure behaviour
- Get produced by a variety of different vendors/manufacturers
- Getting along with middleboxes is like herding cats

Solution by QUIC:

- Encrypt data stream transported by UDP
- ⇒ protocol headers above are not accessible to middleboxes
- TCP-like “optimizers” are not possible due to encryption

QUIC Features
IP Mobility

Problem:

- TCP connections are identified by the 5-tuple
- Client IP address may change during the connection
- DSL connection gets re-established after 24h
- Mobile clients move from one network to another
- NAT entry might expire → port changes

Solution:

- Do not use the 5-tuple as connection identifier
- QUIC identifies connections by a Connection ID
- Last client IP address to send a valid packet for a given Connection ID is the current IP address of the client

In Practice

- Google Chrome: `chrome://flags/` → Experimental QUIC protocol → enabled
- QUIC is deployed for example on `google.com` and `youtube.com`

- There exist multiple implementations in different programming languages

Name	Language	Version	Link
aioquic	Python	v1	https://github.com/aiortc/aioquic
lsquic	C	v1, v2	https://github.com/litespeedtech/lsquic
quic-go	Go	v1, v2	https://github.com/quic-go/quic-go
quiche	Rust	v1	https://github.com/cloudflare/quiche
. . .			

- Not all implementations are compatible to each other

Standardization
IETF

- QUIC standardization since July 2016 by the Internet Engineering Task Force (IETF)
- Standardization finished with the release of RFC 9000 in May 2021 (after 34 drafts)
- `https://datatracker.ietf.org/wg/quic/documents/`
- 5 key goals:
  - 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
  - Providing always-secure transport, using TLS 1.3 by default

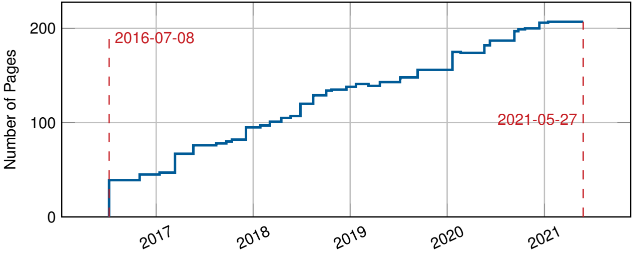
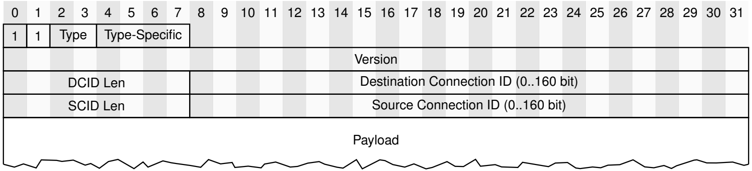


Figure 7: Number of pages in the IETF QUIC draft / RFC.

IETF QUIC
Packet Format

Long Header

- Only used for `Initial`, `0-RTT`, `Handshake`, and `Retry` packets



0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

1 | 1 | Type | Type-Specific

Version

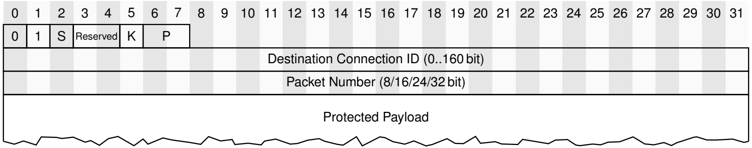
DCID Len | Destination Connection ID (0..160 bit)

SCID Len | Source Connection ID (0..160 bit)

Payload

Short Header

- Designed for minimal overhead
- Used after a connection is established



0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

0 | 1 | S | Reserved | K | P

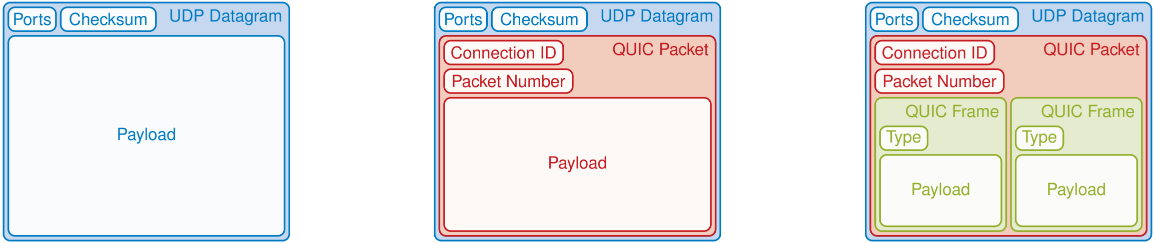
Destination Connection ID (0..160 bit)

Packet Number (8/16/24/32 bit)

Protected Payload

IETF QUIC

Packet Format



QUIC packet:

- A complete processable unit of QUIC that can be encapsulated in a UDP datagram
- Multiple QUIC packets can be encapsulated in a single UDP datagram
- Connection ID used to get connection, packet number to decrypt payload

QUIC frame:

- Types: `PADDING`, `PING`, `ACK`, `STREAM`, ...
- Some frame types are only allowed in certain packet types, e.g. at connection start / end

Packet number:

- Integer in the range 0 to $2^{62} - 1$
- Used in determining the cryptographic nonce for packet protection
- Different packet number spaces for initial packets, handshake packets, and application packets
- Start at packet number 0 and must be increased by at least 1 for subsequent packets

IETF QUIC
Packet Format

Different QUIC packet types:

- `Initial` and `Handshake`: carries the first CRYPTO frames and ACKs sent by the client and server to perform key exchange
- `0-RTT`: used to carry "early" data from the client to the server as part of the first flight, prior to handshake completion, e.g. `HTTP` request
- `1-RTT`: used with the short header once 1-RTT keys are available

Different QUIC frame types:

`PADDING`, `PING`, `ACK`, `STREAM`, `...`

Variable Length Integer Encoding:

- ensures that smaller integer values need fewer bytes to encode
- the two most significant bits of the first byte encode the $\log_2$ of the integer encoding length in bytes

2 bit	Length	Usable Bits	Range
00	1	6	0 - 63
01	2	14	0 - 16383
10	4	30	0 - 1073741823
11	8	62	0 - 4611686018427387903

Security

Security Goals:

- Confidentiality (only encrypted data transfer)
- Authentication (server is authenticated, client optionally)
- Integrity (message authentication code)

IETF QUIC
Security

Security Goals:

- Confidentiality (only encrypted data transfer)
- Authentication (server is authenticated, client optionally)
- Integrity (message authentication code)

TLS 1.3

- TLS (Transport Layer Security) 1.3 specified in RFC 8446
- Faster handshakes than previous TLS versions, also 0-RTT
- Removes several outdated / insecure cipher suites
- Only supports AEAD algorithms

IETF QUIC
Security

Security Goals:

- Confidentiality (only encrypted data transfer)
- Authentication (server is authenticated, client optionally)
- Integrity (message authentication code)

TLS 1.3

- TLS (Transport Layer Security) 1.3 specified in RFC 8446
- Faster handshakes than previous TLS versions, also 0-RTT
- Removes several outdated / insecure cipher suites
- Only supports AEAD algorithms

AEAD

- Authenticated encryption with additional data
- Encrypt and compute message authentication code (MAC) simultaneously
- Plaintext P, ciphertext C, associated data A, nonce N, key k
- Encrypt: $C = f(k, N, A, P)$
- Decrypt: $P = f(k, N, A, C)$, should return an error if integrity check fails

IETF QUIC

Packet Protection

Cryprography:

- Shared secret *S*, plaintext *P*, ciphertext *C*
- Derived keys from *S* using key derivation function:
  - *key*
  - *iv* (initialization vector)
  - *hp* (header protection)
- Number used once (nonce) *N* to prevent replay attacks, derived from the packet number

IETF QUIC

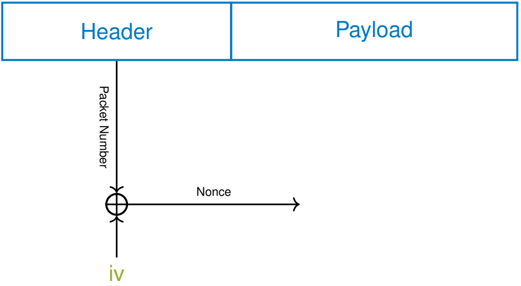
Packet Protection

Cryprography:

- Shared secret *S*, plaintext *P*, ciphertext *C*
- Derived keys from *S* using key derivation function:
  - *key*
  - *iv* (initialization vector)
  - *hp* (header protection)
- Number used once (nonce) *N* to prevent replay attacks, derived from the packet number

Encrypt:

1. Compute packet nonce *N*



IETF QUIC

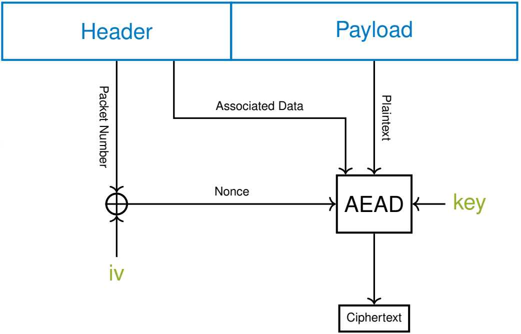
Packet Protection

Cryprography:

- Shared secret S, plaintext P, ciphertext C
- Derived keys from S using key derivation function:
  - *key*
  - *iv* (initialization vector)
  - *hp* (header protection)
- Number used once (nonce) N to prevent replay attacks, derived from the packet number

Encrypt:

1. Compute packet nonce N
2. Compute $C = AEAD(key, N, \text{associated data}, P)$



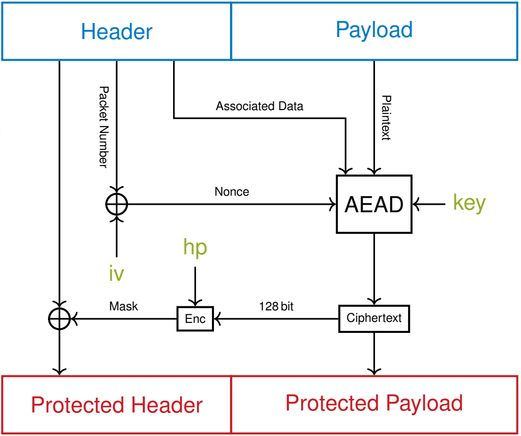
Packet Protection

Cryprography:

- Shared secret *S*, plaintext *P*, ciphertext *C*
- Derived keys from *S* using key derivation function:
  - *key*
  - *iv* (initialization vector)
  - *hp* (header protection)
- Number used once (nonce) *N* to prevent replay attacks, derived from the packet number

Encrypt:

1. Compute packet nonce *N*
2. Compute $C = AEAD(key, N, \text{associated data}, P)$
3. Add header protection
   - Encrypt certain 128 bit of *C* with *hp* key
   - Mask so that only some header fields are protected (e.g. packet number)
   - XOR with original header



IETF QUIC

Packet Protection

Cryprography:

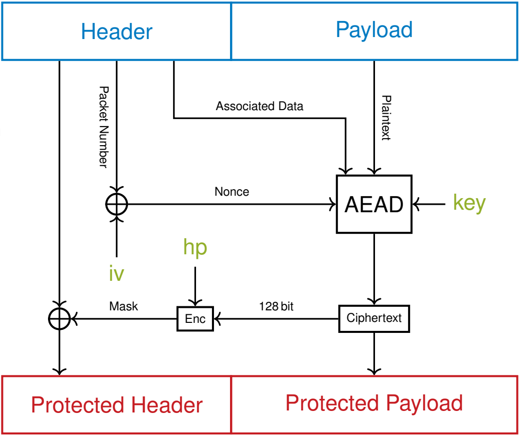
- Shared secret *S*, plaintext *P*, ciphertext *C*
- Derived keys from *S* using key derivation function:
  - *key*
  - *iv* (initialization vector)
  - *hp* (header protection)
- Number used once (nonce) *N* to prevent replay attacks, derived from the packet number

Encrypt:

1. Compute packet nonce *N*
2. Compute $C = AEAD(key, N, \text{associated data}, P)$
3. Add header protection
   - Encrypt certain 128 bit of *C* with *hp* key
   - Mask so that only some header fields are protected (e.g. packet number)
   - XOR with original header

Decrypt:

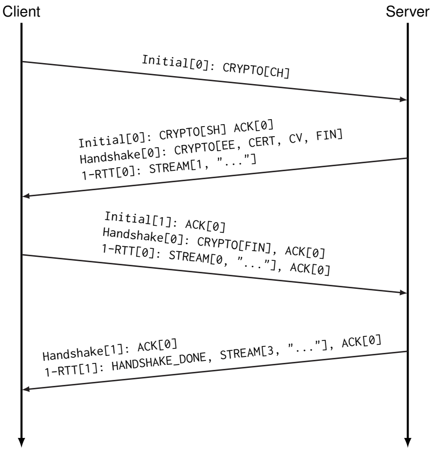
1. Remove header protection
2. Compute packet nonce *N*
3. Compute $P = AEAD(key, N, \text{associated data}, C)$



IETF QUIC
Handshake

- Combined Transport and cryptographic handshake (current version uses TLS 1.3)
- Authenticated key exchange
  - Server is always authenticated (e.g. certificate)
  - Client is optionally authenticated
- Authenticated exchange of values for transport parameters
  - E.g. `max_idle_timeout`, `max_udp_payload_size`, `initial_max_data`, `...`
- Negotiating Connection IDs

IETF QUIC
Handshake Example



Client

Server

Initial[0]: CRYPTO[CH]

Initial[0]: CRYPTO[SH] ACK[0]
Handshake[0]: CRYPTO[EE, CERT, CV, FIN]
1-RTT[0]: STREAM[1, "..."]

Initial[1]: ACK[0]
Handshake[0]: CRYPTO[FIN], ACK[0]
1-RTT[0]: STREAM[0, "..."], ACK[0]

Handshake[1]: ACK[0]
1-RTT[1]: HANDSHAKE_DONE, STREAM[3, "..."], ACK[0]

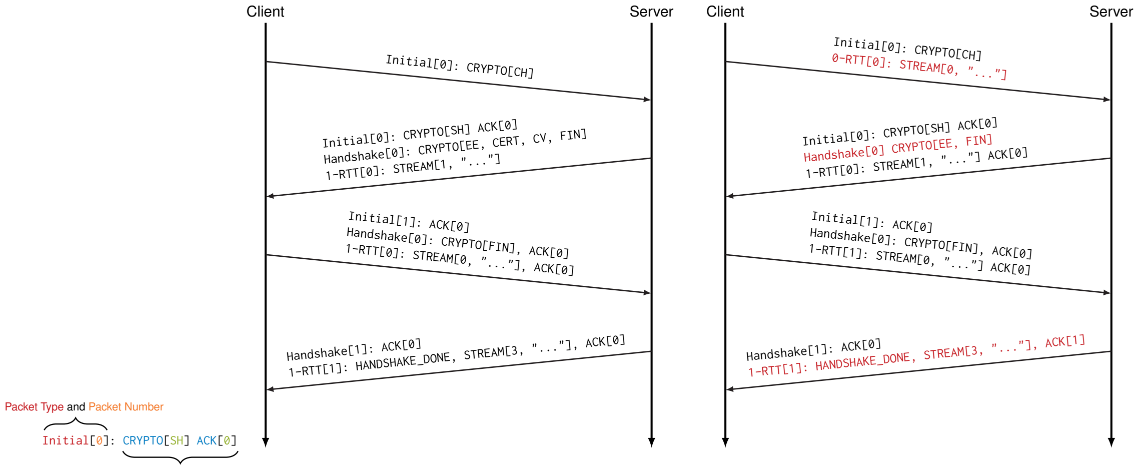
Packet Type and Packet Number

`Initial[0]: CRYPTO[SH] ACK[0]`

Frame Types and Content

(a) 1-RTT Handshake

IETF QUIC
Handshake Example



Client

Server

Initial[0]: CRYPTO[CH]

Initial[0]: CRYPTO[SH] ACK[0]
Handshake[0]: CRYPTO[EE, CERT, CV, FIN]
1-RTT[0]: STREAM[1, "..."]

Initial[1]: ACK[0]
Handshake[0]: CRYPTO[FIN], ACK[0]
1-RTT[0]: STREAM[0, "..."], ACK[0]

Handshake[1]: ACK[0]
1-RTT[1]: HANDSHAKE_DONE, STREAM[3, "..."], ACK[0]

(a) 1-RTT Handshake

Client

Server

Initial[0]: CRYPTO[CH]
0-RTT[0]: STREAM[0, "..."]

Initial[0]: CRYPTO[SH] ACK[0]
Handshake[0] CRYPTO[EE, FIN]
1-RTT[0]: STREAM[1, "..."] ACK[0]

Initial[1]: ACK[0]
Handshake[0]: CRYPTO[FIN], ACK[0]
1-RTT[1]: STREAM[0, "..."] ACK[0]

Handshake[1]: ACK[0]
1-RTT[1]: HANDSHAKE_DONE, STREAM[3, "..."], ACK[1]

(b) 0-RTT Handshake

Packet Type and Packet Number

Initial[0]: CRYPTO[SH] ACK[0]

Frame Types and Content

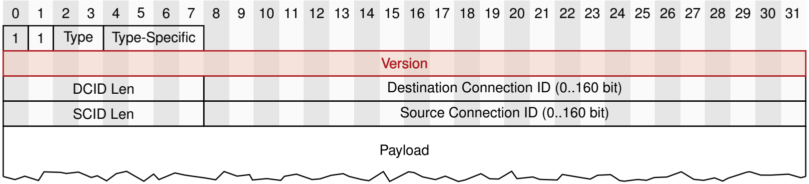
IETF QUIC
Version Negotiation

- QUIC versions are identified using a 32-bit unsigned number
- Version `0x0000 0000` is reserved to represent version negotiation
- The version of QUIC v1 is identified by the number `0x0000 0001`
- Public known versions of different vendors:
  `https://github.com/quicwg/base-drafts/wiki/QUIC-Versions`

Version	Owner
`0x0000 0001`	IETF (QUIC v1)
`0x5130 xxxx`	Google
`0xface b00x`	Facebook
`0xabcd 000x`	Microsoft
`0xf0f0 f0fx`	ETH Zürich
`0xf123 f0cx`	Mozilla
...	...

Procedure:

- Client sends used version in the long header
- If the version is not supported by the server it replies with a `Version Negotiation` packet listing all supported versions (its own version field is set to `0x0000 0000`)
- The client can pick a supported version



IETF QUIC
Streams and Acknowledgements

Streams:

- Lightweight, ordered byte-stream abstraction
- Bidirectional or unidirectional
- Stream frames can open, carry data for, or close a stream
- Unique stream ID (62-bit integer), two bits used to identify initiator and if bi- or unidirectional
- Multiple streams are sent interleaved, streams can be prioritized (avoidance of head-of-line blocking)

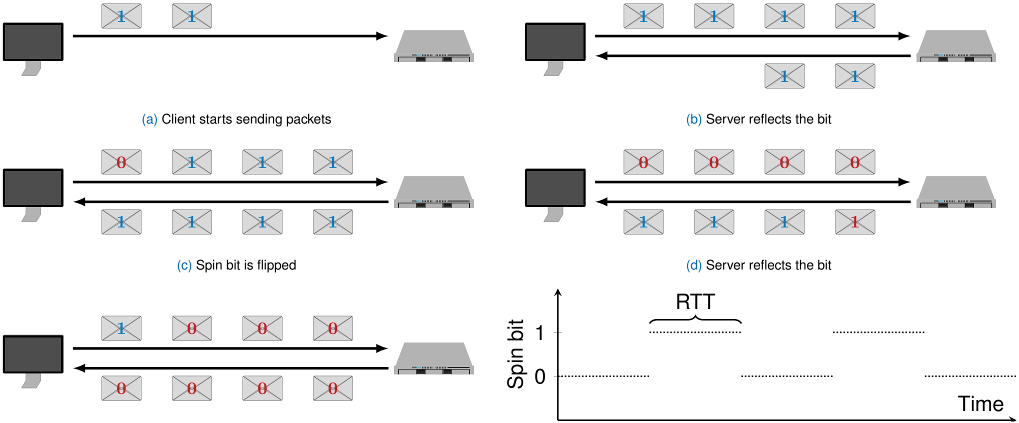
Acknowledgements:

- Packet numbers are acknowledged, after all frames have been processed
- Tries to send ACK frames as often as possible to improve loss and congestion response
- Trade-off between load generation and short response times
- ACK frame contains multiple ACK ranges

Analysis

Spin Bit

- Most of the QUIC PDU is encrypted, which makes passive monitoring impossible
  - e.g. for TCP SEQ / ACK pairs and timestamp options are observable
- Spin bit introduces the possibility to passively measure the connection’s RTT



(a) Client starts sending packets

(b) Server reflects the bit

(c) Spin bit is flipped

(d) Server reflects the bit

(e) Spin bit is flipped again

(f) Spin bit as seen by an observer

Analysis
qlog and qvis [14]

- IETF drafts gives guidelines for implementing the QUIC protocol
- Implementations widely differ due to different developers / languages
  - Packets on the wire are encrypted (requires session keys to analyze)
  - Internal QUIC state / events cannot be analyzed only with packet traces

➔ Tool to analyze, compare and verify implementations is needed

qlog

- Based on JSON
- (timestamp, event type, event specific data)

qvis

- Browser interface to visualize qlog files
- Different diagram types: sequence diagram, congestion diagram, . . .
  - sequence diagram
  - congestion diagram
  - multiplexing diagram
  - packetization diagram
- Try it: `https://qvis.quictools.info`

Applications

Unreliable Datagram Extension (RFC 9221)

- Encrypted and congestion controlled but not flow controlled and reliable (retransmitted)
- QUIC datagrams can share a connection with reliable QUIC streams

→ Only one handshake, one congestion controller, one encryption context, . . .

MASQUE

- **M**ultiplexed **A**pplication **S**ubstrate over **QU**IC **E**ncryption
- Protocol group under standardization by the MASQUE working group
- Proxying of UDP- and IP-based traffic over HTTP

Applications

Multipath Extension for QUIC

- Simultaneous usage of multiple paths for a single connection
- Extension not yet standardized

HTTP/3 (RFC 9114)

- Next version of HTTP is standardized using QUIC as underlying protocol
- Distribute different transactions (request / response pairs) to individual streams

→ Fixes HoL-blocking problem of HTTP/2

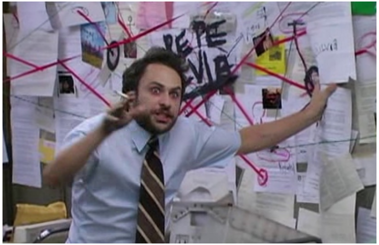
QUIC Version 2 (RFC 9369)

- Version field value: `0x6b3343cf` (first four bytes of the `sha256sum` of "QUICv2 version number")
- Further prevent network ossification

QUIC – Conclusion

- Still relatively new protocol
- Higher CPU costs as TCP / TLS, but optimization is ongoing
  - UDP interface is still far less optimized than TCP
  - QUIC encrypts packets twice (header and payload), each packet has to be encrypted individually
- Deploying networking protocols in user space
  - faster and easier development cycles
  - bypass problems like head-of-line blocking
- "Layering enables modularity but often at the cost of performance" [12]
- Achieve lower latency with 0-RTT handshake
- HTTP/3 is standardized using QUIC instead of TCP

“In other words, QUIC is as simple as the modern internet demands, which is not very simple in absolute terms.” [2]



[2] https://www.fastly.com/blog/maturing-of-quic

QUIC

[1] DARPA, TRANSMISSION CONTROL PROTOCOL, https://tools.ietf.org/html/rfc793, 1981.

[2] S. Ha, I. Rhee, and L. Xu, “CUBIC: a new TCP-friendly high-speed TCP variant,” ACM SIGOPS operating systems review, vol. 42, no. 5, pp. 64–74, 2008.

[3] L. Xu, A. Zimmermann, L. Eggert, I. Rhee, R. Scheffenegger, and S. Ha, “CUBIC for Fast Long-Distance Networks,”, 2018.

[4] L. Xu, S. Ha, I. Rhee, V. Goel, and L. Eggert, “CUBIC for Fast and Long-Distance Networks,” RFC Editor, RFC 9438, 2023.

[5] L. S. Brakmo, S. W. O’Malley, and L. L. Peterson, “TCP Vegas: New Techniques for Congestion Detection and Avoidance,” in Proceedings of the Conference on Communications Architectures, Protocols and Applications, 1994. [Online]. Available: http://doi.acm.org/10.1145/190314.190317.

[6] S. Shalunov, G. Hazel, J. Iyengar, and M. Kuehlewind, “Low extra delay background transport (ledbat),” RFC 6817, 2012. [Online]. Available: http://www.rfc-editor.org/rfc/rfc6817.txt.

[7] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson, “BBR: Congestion-Based Congestion Control,” ACM Queue, 2016. [Online]. Available: http://queue.acm.org/detail.cfm?id=3022184.

[8] D. Scholz, B. Jaeger, L. Schwaighofer, D. Raumer, F. Geyer, and G. Carle, “Towards a Deeper Understanding of TCP BBR Congestion Control,” in IFIP Networking 2018, Zurich, Switzerland, May 2018.

[9] M. Hock, R. Bless, and M. Zitterbart, “Experimental evaluation of BBR congestion control,” in 2017 IEEE 25th International Conference on Network Protocols (ICNP), IEEE, 2017, pp. 1–10.

QUIC

[10] N. Cardwell, Y. Cheng, et al., “BBR v2: A Model-based Congestion Control Performance Optimizations,” IETF 106, 2019, Presentation Slides. [Online]. Available: https://datatracker.ietf.org/meeting/106/materials/slides-106-iccrg-update-on-bbrv2.

[11] J. Postel, User Datagram Protocol, https://tools.ietf.org/html/rfc768, 1990.

[12] A. Langley, A. Riddoch, A. Wilk, et al., “The QUIC Transport Protocol: Design and Internet-Scale Deployment,” in Proceedings of the Conference of the ACM Special Interest Group on Data Communication, 2017. [Online]. Available: http://doi.acm.org/10.1145/3098822.3098842.

[13] E. J. Iyengar and E. M. Thomson, QUIC: A UDP-Based Multiplexed and Secure Transport, https://tools.ietf.org/html/draft-ietf-quic-transport-05, 2017.

[14] R. Marx, M. Piraux, P. Quax, and W. Lamotte, “Debugging QUIC and HTTP/3 with Qlog and Qvis,” in Proceedings of the Applied Networking Research Workshop, ser. ANRW ’20, Virtual Event, Spain: Association for Computing Machinery, 2020, 58–66, ISBN: 9781450380393. DOI: 10.1145/3404868.3406663. [Online]. Available: https://doi.org/10.1145/3404868.3406663.