TCP Cubic
Basics of the Algorithm

Theory behind Cubic:

- Published in 2008 [2]
- RFC since February 2018 [3]
- 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

\[ W_{cubic}(t) = C \cdot (t - K)^3 + W_{max} \]  

(2)

\[ K = \frac{3 \sqrt{W_{max} \cdot (1 - \beta_{cubic})}}{C} \]  

(3)

\[ C = 0.4 \]  

(4)

\[ \beta = 0.7 \]  

(5)

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

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)
$C$ and $\beta_{\text{cubic}}$ 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

Disadvantages

• Buffers are filled faster (cubic growth function)
• Buffers are kept filled (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:

- 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: Additive Increase Additive Decrease

Formula

\[ \Delta = CWND \cdot \frac{RTT - RTT_{\text{min}}}{RTT} \]  

(6)

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) [5]
- 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 [6]
- 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" [6]
- "BBR yielded 4 percent higher network throughput […] BBR also keeps network queues shorter, reducing round-trip time by 33 percent"¹

TCP BBR

Theory: ACK-clocking and Pacing

ACK-clocking

- Used by Reno, Cubic, Vegas, ...
- CWND limits the inflight data 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 a 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 \cdot 2.8853 \approx 2.8853
- Stop after three consecutive RTTs with less than 25% in delivery rate increase
- Finds BtlBw in \log_2(BDP) RTTs
- Can create queue up to 2 BDP

Drain

- Goal: Remove the queue created during Startup
- Sending Rate = BtlBw \cdot 0.3465 \approx 0.3465
- Leave Drain when data in flight matches estimated BDP

![Graph showing Startup send rate over time](image.png)
TCP BBR

Startup and Drain

Startup

- Similar to Slow Start → double sending rate each RTT
- Sending Rate = BtlBw · 2.8853 \((\frac{2}{\ln 2} \approx 2.8853)\)
- 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 \((\frac{\ln 2}{2} \approx 0.3465)\)
- Leave Drain when data in flight matches estimated BDP

![Graph showing Startup and Drain phases](image-url)
TCP BBR

Probe Bandwidth

- Periodically probe for more bandwidth
- BtIBw is estimated using a max filter of length about ten estimated RTTs
- Sending Rate = BtIBw \cdot PacingGain, with PacingGain in \([1.25, 0.75, 1, 1, 1, 1, 1, 1, 1]\)
- Each step takes about one RTT
- If no bandwidth is available: sending rate is reduced afterwards to remove queue
- If bandwidth is available: BtIBw 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

![Graph showing sending rate over time](image)

**Legend**
- Startup
- Drain
- Probe BW
- Probe RTT

**Axes**
- Sending Rate [Mbit/s]
- Time [s]
TCP BBR

BBR Single Flow

![Graph showing the performance of TCP BBR with two BtlBw and three RTprop scenarios. The graph includes plots for sending rate, BtlBw, RTT, Inflight, and Estimated BtlBw and RTprop over time.](image)
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 2: Figures from [6].
TCP BBR

Problems with BBR [8, 7]

- 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 $\approx 1.5$ BDP

- High number of retransmissions in networks with shallow buffers
  - If the buffer is smaller than the persistent queue $\rightarrow$ 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 3: Source [7]
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
- Incorporate 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 4: Figures from [9]
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

![Graphs of Reno, Cubic, Vegas, and BBR showing sending and delivery rates, RTT, and inflight BDP over time.]

Try it yourself: [https://gitlab.lrz.de/tcp-bbr/measurement-framework](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 [10] (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

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-4 B</td>
<td>Source Port: Which application of the sending host sent the datagram</td>
</tr>
<tr>
<td>4-8 B</td>
<td>Destination Port: Which application of the receiving host should receive the datagram</td>
</tr>
<tr>
<td></td>
<td>Length: Length of the datagram (L4-PDU)</td>
</tr>
<tr>
<td></td>
<td>Checksum: Checksum over IP pseudo header, UDP header, data</td>
</tr>
</tbody>
</table>

**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 mechanisms 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