

Congestion Control in High Speed Networks: Part 3

BBR, XCP, RCP

Lecture 7

Subir Varma

BBR

Bottleneck BW and Rate

networks

1 of 34

TEXT
ONLY



BBR

Congestion-Based Congestion Control

NEAL CARDWELL
YUCHUNG CHENG
C. STEPHEN GUNN
SOHEIL HASSAS YEGANEH
VAN JACOBSON

By all accounts, today's Internet is not moving data as well as it should. Most of the world's cellular users experience delays of seconds to minutes; public Wi-Fi in airports and conference venues is often worse. Physics and climate researchers need to exchange petabytes of data with global collaborators but find their carefully engineered multi-Gbps infrastructure often delivers at only a few Mbps over intercontinental distances.⁶

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

**MEASURING
BOTTLENECK
BANDWIDTH
AND ROUND-TRIP
PROPAGATION
TIME**

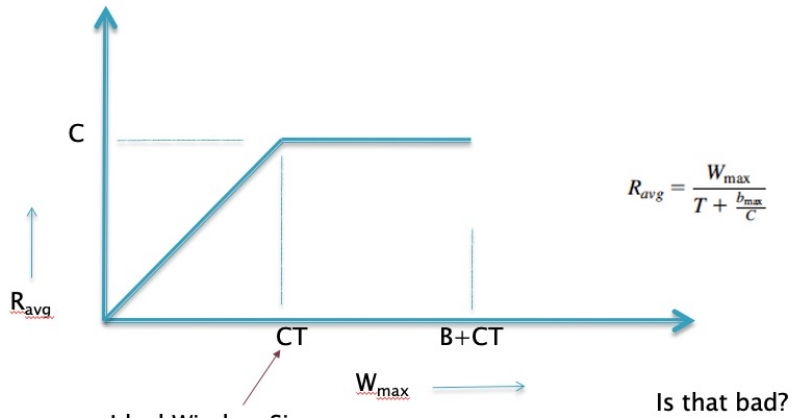
BBR

- ▶ So far we have seen the following types of congestion control algorithms:
 - Those that use packet loss as a measure of congestion (Reno, CUBIC)
 - Those that use queueing latency as a measure of congestion (Vegas, FAST)
 - Those that use a combination of loss and latency (Compound TCP, Yeah TCP)
- ▶ BBR adopts a new way: It uses the available bandwidth in the bottleneck link as a measure of congestion.
- ▶ TCP Westwood was actually the first to propose measuring the bottleneck bandwidth, and used it for setting the window size after a packet loss.
- ▶ BBR is a rate based algorithm, hence it sets its rate so as to match its estimate of the bottleneck bandwidth.
- ▶ BBR tries to provide high link utilization while avoiding the need to overload bottleneck buffers.
- ▶ BBR was proposed by Google, and is used in Google's B4 wide area backbone network, as well on Google.com and YouTube servers.

BBR Motivation

- ▶ Issue with Loss based algorithms: They fill up the buffer, which have become larger over time in routers and switches (bufferbloat). This increases latency for all flows traversing the link.
- ▶ Alternative: Use Delay based algorithms. But these algorithms are at a disadvantage when competing with Loss based algorithms.
- ▶ BBR: A Delay type algorithm that is able to hold its own against Loss based algorithms. How does it do this?
 - By using a Rate based scheme, which sets the rate based on bottleneck bw estimates, not on either Loss or Delay
- ▶ Since BBRv1 does not react to loss, what happens when it shares the link with Loss based algos?
 - BBRv1 tends to get an unfair share of the link capacity!
 - This hasn't stopped Google and some other large companies from deploying BBR. This also means that the requirement of inter-protocol fairness has become less important in recent years.

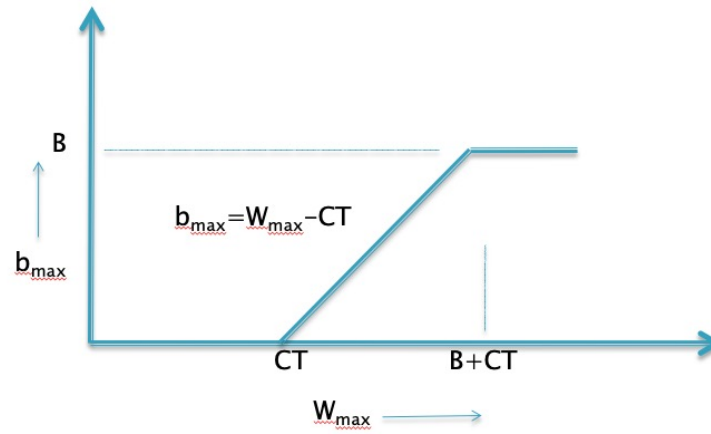
Recall from Lecture 2



Ideal Window Size

Larger Window Size causes queueing delays without adding to the throughput

Is that bad?



When W_{max} increases to $B+CT$, then $b_{max}=B$, and any increase of W_{max} beyond this value will cause the buffer to overflow.

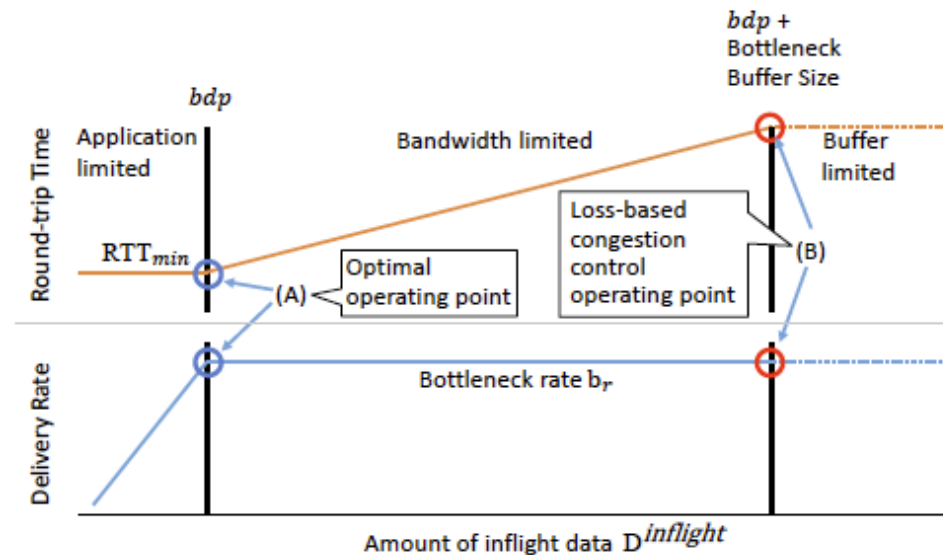
Issue with Loss Based Protocols

- ▶ The bottleneck link is fully utilized if the amount of inflight data D_{inflight} matches exactly the bandwidth delay product $\text{bdp} = b_r \text{RTT}_{\text{min}}$, where b_r is the available bottleneck data rate and RTT_{min} is the minimal round-trip time
- ▶ A fundamental difficulty of congestion control is to calculate a suitable amount of inflight data without exact knowledge of the current bdp.
 - If D_{inflight} is smaller than bdp, the bottleneck link is not fully utilized and bandwidth is wasted.
 - If D_{inflight} is larger than bdp, the bottleneck is overloaded, and any excess data is filled into a buffer at the bottleneck link or dropped if the buffer capacity is exhausted. If this overload situation persists the bottleneck becomes congested.

Issue with Loss Based Protocols

- ▶ Loss-based congestion controls (such as CUBIC TCP or TCP Reno) use packet loss as congestion signal. They tend to completely fill the available buffer capacity at a bottleneck link, since most buffers in network devices still apply a tail drop strategy. A filled buffer implies a large queuing delay that adversely affects everyone's performance on the Internet: the inflicted latency is unnecessarily high.
- ▶ This also highly impacts interactive applications (e.g., Voice-over-IP, multiplayer online games), which often have stringent requirements to keep the one way end-to-end delay below 100 ms.

BBR



- If the amount of inflight data $D_{inflight}$ is just large enough to fill the available bottleneck link capacity (i.e., $D_{inflight} = bdp$), the bottleneck link is full utilized and the queuing delay is still zero or close to zero.
- This is the optimal operating point (A), because the bottleneck link is already fully utilized at this point.
- If the amount of inflight data is increased any further, the bottleneck buffer gets filled with the excess data. The delivery rate, however, does not increase anymore. The data is not delivered any faster since the bottleneck does not serve packets any faster and the throughput stays the same for the sender: the amount of inflight data is larger, but the round-trip time increases by the corresponding amount.
- BBR tries to shift the operating point of congestion control to the left toward (A).

BBR Steady State Operation

Since BBR is Rate Based, the only way it can find out that more bandwidth is available is by sending probes

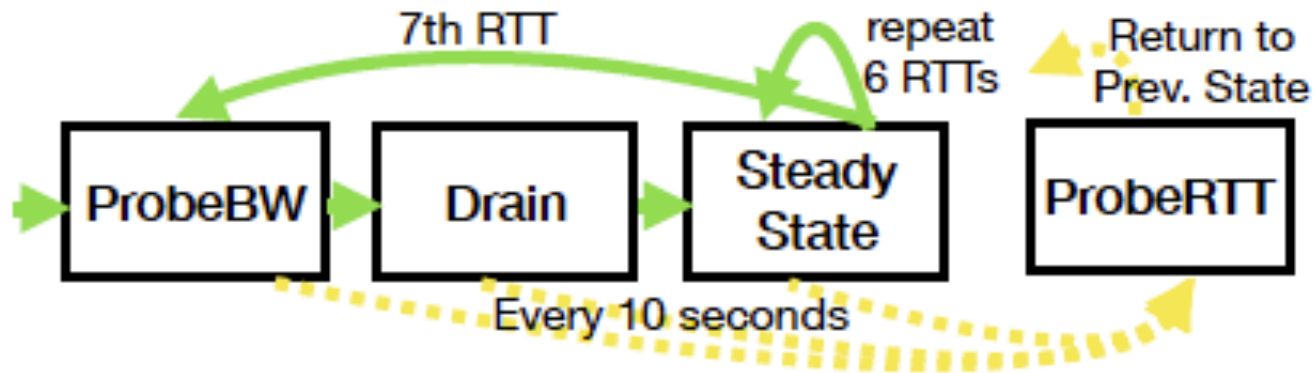


Figure 4: BBR's steady-state operation.

BBR tries to get a better estimate of RTT_{min} by draining all the queues periodically

BBR v1 Description: Single Flow Case

- ▶ BBR uses estimates for the available bottleneck data rate b_r and the minimal round-trip time RTT_{min} to calculate a path's available bdp. The estimate for b_r is based on a windowed maximum filter of the delivery rate that the receiver experiences.
- ▶ A BBR sender controls its transmission rate s_r with the help of pacing and an estimated data rate b_r , i.e., it is **rate-based**. It does not use a congestion window or ACK clocking to control the amount of inflight data, but uses an inflight data limit of $2 * bdp$.
- ▶ BBR probes for more bandwidth by increasing its sending rate s_r to $1.25s_{r0}$ for an RTT and directly reducing it again to $0.75s_{r0}$, where s_{r0} is the current estimate of the available data rate. The reduction aims at draining a potential queue that was possibly created by the higher rate.
- ▶ BBR uses a special ProbeRTT phase that **tries to drain the queue completely in order to measure RTT_{min}** . Ideally, all BBR flows enter this phase together.
- ▶ BBR is neither delay-based nor loss-based and it ignores packet loss as congestion signal. It also **does not explicitly react to congestion**, whereas congestion window-based approaches often use a multiplicative decrease strategy to reduce $D_{inflight}$.

TCP Westwood: ABE Estimate

Define the following:

t_k : Arrival time of the k^{th} ACK at the sender

d_k : Amount of data being acknowledged by the k^{th} ACK

$\Delta_k = t_k - t_{k-1}$ Time interval between the k^{th} and $(k-1)^{\text{st}}$ ACKs

$R_k = \frac{d_k}{\Delta_k}$, Instantaneous value of the connection bandwidth at the time of arrival of the k^{th} ACK

\hat{R}_k : Low-pass filtered (LPF) estimate of the connection bandwidth

τ : Cut-off frequency of the LPF

The LPF estimate of the bandwidth is then given by

$$\hat{R}_k = \frac{2\tau - \Delta_k}{2\tau + \Delta_k} \hat{R}_{k-1} + \frac{\Delta_k}{2\tau + \Delta_k} (R_k + R_{k-1}) \quad (6)$$

Low-pass filtering is necessary because congestion is caused by the low-frequency components and because of the delayed ACK option.

The filter coefficients are time varying to counteract the fact that the sampling intervals Δ_k are not constant.

Westwood+: To counteract ACK compression

Instead of computing the bandwidth R_k after every ACK is received, compute it once every RTT seconds. Hence, if D_k bytes are acknowledged during the last RTT interval Δ_k , then

$$R_k = \frac{D_k}{\Delta_k}$$

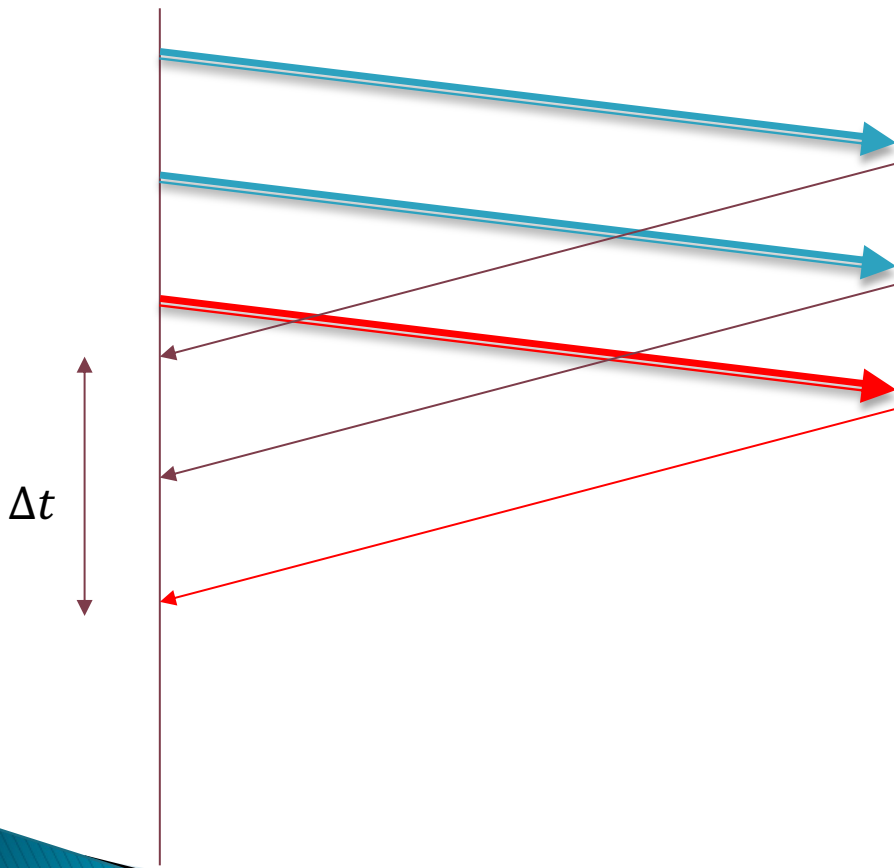
BBR *bdp* Estimation

- ▶ A BBR sender tries to determine the bdp of a network path by getting estimates for b_r and RTT_{\min} from measurements.
- ▶ b_r is estimated by using TCP ACKs and is calculated by dividing the amount of delivered data by the period Δt in which the measurement took place.

$$\hat{b}_r = \max(\text{delivery_rate}_t) \quad \forall t \in [\bar{T} - W_B, T]$$

W_B is typically six to ten RTTs. This estimate is updated with every received ACK.

b_r estimate



$$b_r = \frac{\Delta delivered}{\Delta t}$$

$$\hat{b}_r = \max(delivery_rate_t) \quad \forall t \in [\bar{T} - W_B, T]$$

TCP must record the departure time of each packet to compute RTT. BBR augments that record with the total data delivered so each ack arrival yields both an RTT and a delivery rate measurement that the filters convert to RT_{prop} and b_r estimates.

BBR Rate Changes

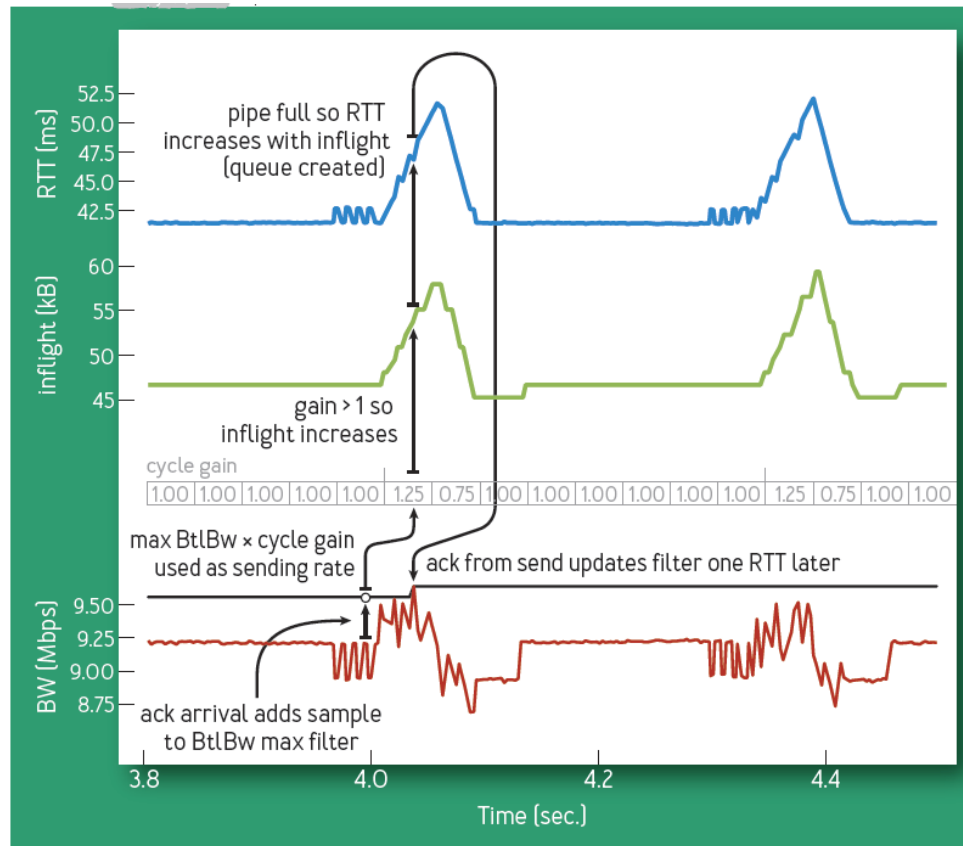
- ▶ BBR does not use ACK clocking to regulate its sending rate, instead it used *pacing* for every data packet sent.
- ▶ BBR does cap the inflight data to $2\widehat{bdp}$, where $\widehat{bdp} = \widehat{b}_r \widehat{RTT}_{min}$
- ▶ Note that if b_r decreases, then it takes at least W_B roundtrips for \widehat{b}_r to change.
- ▶ What if b_r increases?

The rate decrease policy is reminiscent of TCP Westwood and can be classified as Multiplicative Decrease

Probing for more BW Phase

- ▶ BBR probes for more bandwidth by **increasing the sending rate by a certain factor** (pacing_gain = 1.25) for an estimated RTT_{min} , then decreasing the sending rate by pacing_gain = 0.75 in order to compensate a potential excess of inflight data. Thus, if this excess amount filled the bottleneck queue, the queue should be drained by the same amount directly afterwards.
- ▶ Moreover, the sending rate is varied in an eight-phase cycle using a pacing_gain of $5/4, 3/4, 1, 1, 1, 1, 1, 1$, where each phase lasts for an \widehat{RTT}_{min} . The start of the cycle is randomly chosen with $3/4$ being excluded as initial phase.
- ▶ If the increased sending rate showed an increased delivery rate, the newly measured maximum delivery rate \widehat{b}_r is immediately used as new sending rate, otherwise the previous rate is maintained.

Probing for more BW

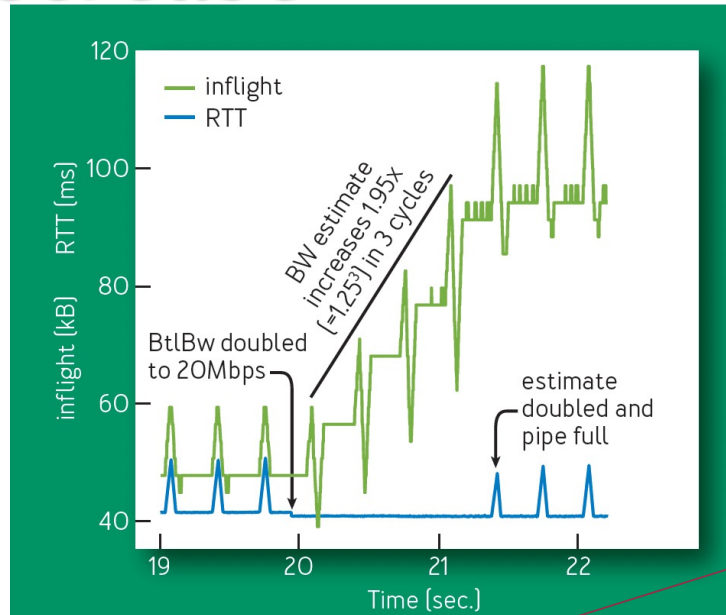


- Case when sending rate is not increased
- If bottleneck rate had increased, the the RTT would not have increased
- In that case the new sending rate = $1.25 \times \text{old rate}$, is retained

BBR is an example of Multiplicative Increase Multiplicative (MIMD) Decrease Algorithm

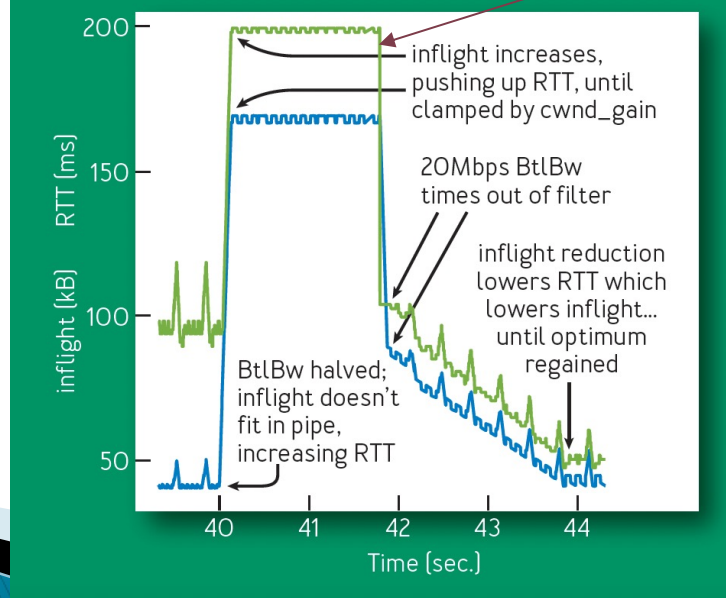
Single BBR Flow: Bt|BW Increase/Decrease

Increase in sending rate



Why does it take this long to react to a Decrease in Bt|BW?

Decrease in sending rate



$$\text{inflight} = \widehat{b}_r \text{ RTT}$$

RTT_{min} Estimation Phase

- ▶ RTT_{min} is calculated by using a minimum filter over a window W_R as $\widehat{RTT}_{min} = \min(RTT_t) \quad \forall t \in [T - W_R, T]$ with $W_R = 10s$
- ▶ In order to measure RTT_{min} , BBR uses a periodically occurring phase which is called *ProbeRTT*. *ProbeRTT* is entered when \widehat{RTT}_{min} has not been updated by a lower measured value for several seconds (default 10 sec).
- ▶ In *ProbeRTT*, the sender abruptly limits its amount of inflight data to 4 packets for $\max(RTT, 200 \text{ ms})$ and then returns to the previous state.
- ▶ This should drain the queue completely under the assumption that only BBR flows are present, so $\widehat{RTT}_{min} = RTT_{min}$ since no queuing delay exists.
- ▶ Large flows that enter *ProbeRTT* will drain many packets from the queue, so other flows will update their \widehat{RTT}_{min} , which creates a synchronization effect among all flows thus increasing the probability to actually measure \widehat{RTT}_{min} .
- ▶ Note that \widehat{RTT}_{min} only modifies the the inflight cap of $2b\widehat{dp}$, the sending rate itself is set to b_r

Issue: What if the queue is not empty?

Startup Phase

- ▶ In its startup phase BBR nearly doubles its sending rate every RTT as long as the delivery rate is increasing.
- ▶ This is achieved by using a pacing_gain of $2/\ln 2 = 2.885$ and an inflight cap of 3bdp , i.e., it may create up to 2bdp of excess queue.
- ▶ BBR tries to determine whether it has saturated the bottleneck link by looking at the development of the delivery rate.
- ▶ If for several (three) rounds attempts to double the sending rate results only in a small increase of the delivery rate (less than 25%), BBR has found the current limit of \widehat{b}_r and exits the startup phase.
- ▶ Then it enters a drain phase in order to reduce the amount of excess data that may have led to a queue. It uses the inverse of the startup's gain to reduce the excess queue and enters the ProbeBW phase once $D_{\text{inflight}} = \text{bdp}$ holds.

Single BBR Flow

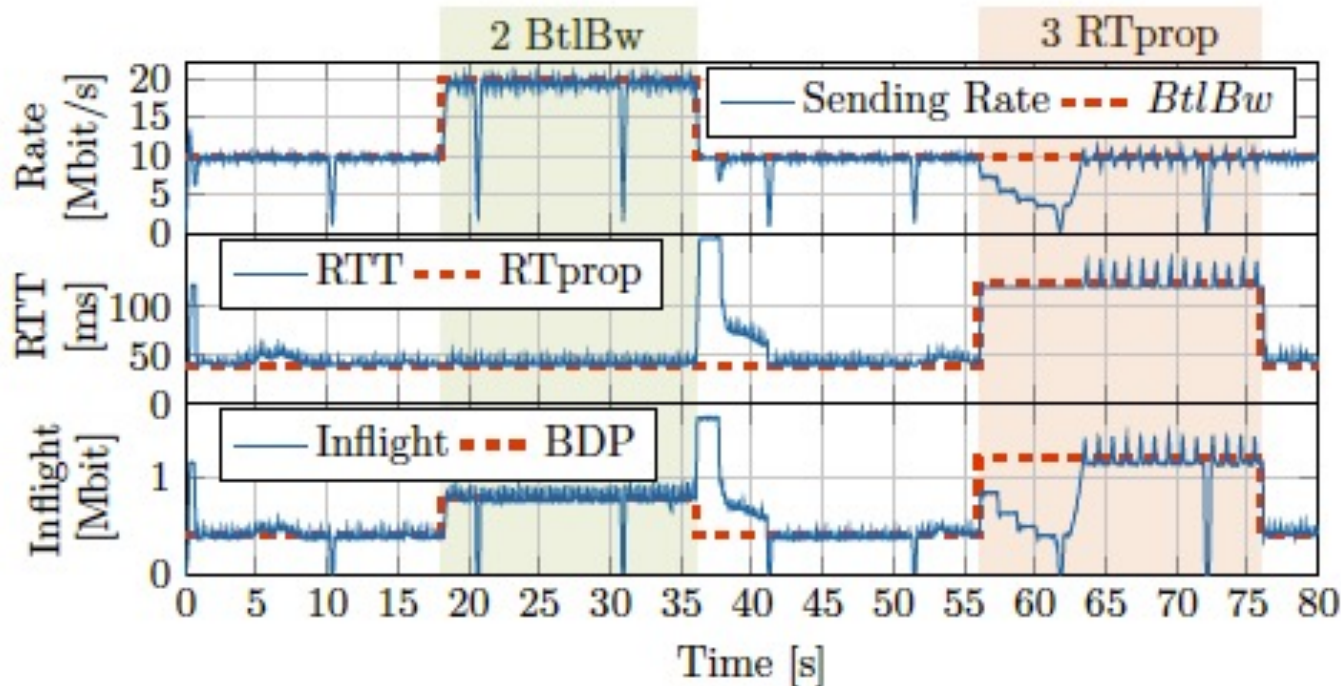
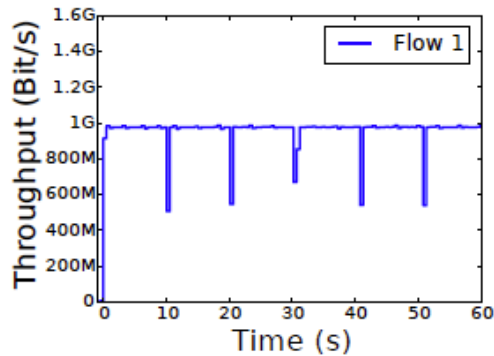


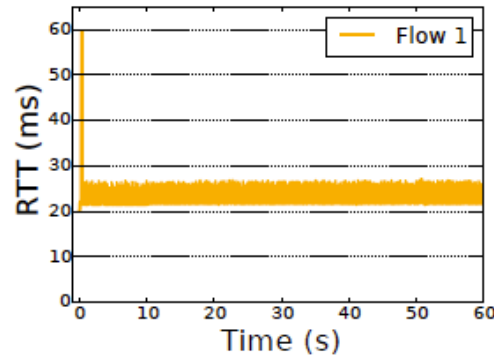
Figure 3: Single BBR flow (40 ms, 10 Mbit/s bottleneck) under changing network conditions. Values sampled every 40 ms.

Single BBR Flow

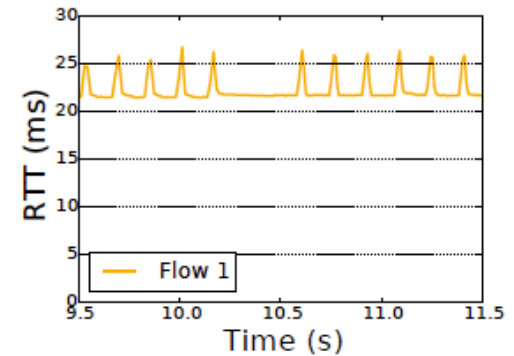
$RTT_{\min} = 20 \text{ ms}$



(a) Throughput



(b) RTT



(c) RTT (zoomed-in)

Fig. 4: A single BBR flow with 1 Gbit/s bottleneck

Issues with BBR v1

- ▶ BBR works well if there is only single flow
- ▶ With multiple flows each sender overestimates the bottleneck bandwidth leading to a too high total D_{inflight} . As a result it leads to:

- Increased queueing delays
- High packet losses
- Unfair rate sharing

$\widehat{b}_{r_i} > b_{r_i}$ and also $\sum_i \widehat{b}_{r_i} > b_r$, which leads to $\sum_i s_{r_i} > b_r$ (\Rightarrow overloaded link)

Sending rate

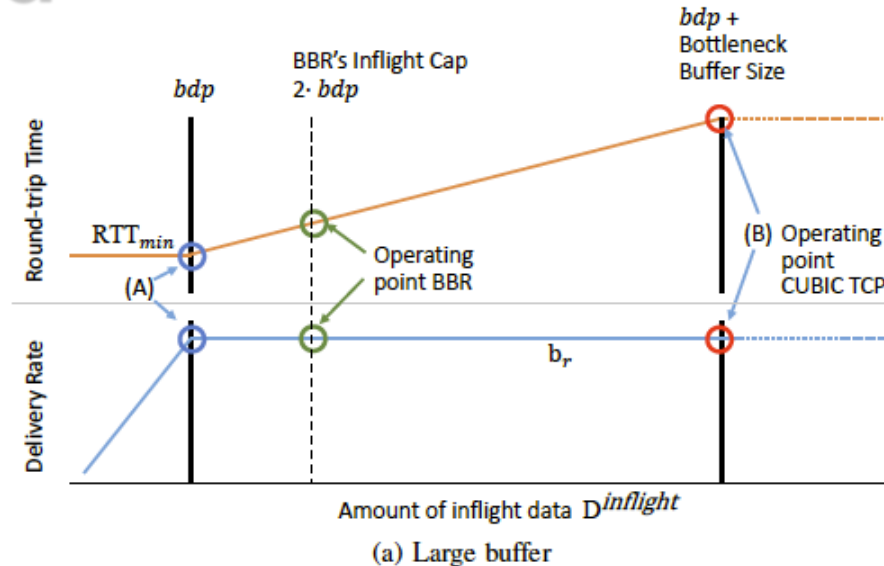


and this can happen for for sustained periods of time. They also showed that

$$\sum_i \widehat{b}_{r_i} \in [b_r, 1.25b_r)$$

- ▶ As a result, the amount of inflight data steadily increases until it gets limited by the inflight cap of $D_i^{cap} = 2bdp$.

BBR with Multiple Flows: Window Limited



As a result of BW overestimation, the inflight cap is regularly reached

$$\sum_i D_i^{cap} = 2 \sum_i \widehat{b_{r_i}} \cdot \widehat{RTT_{min}}$$

Since $\sum_i \widehat{b_{r_i}} \in [b_r, 1.25b_r)$ it follows that $2bdp \leq \sum_i D_i^{cap} < 2.5bdp$

This means that multiple BBR flows typically create a queuing delay of about one to 1.5 times the RTT

Multiple BBR flows operate at their in-flight cap in buffer-bloated Networks, i.e. it effectively becomes a windows based protocol!

BBR with a Multiple Flows

$$RTT_{\min} = 20 \text{ ms}$$

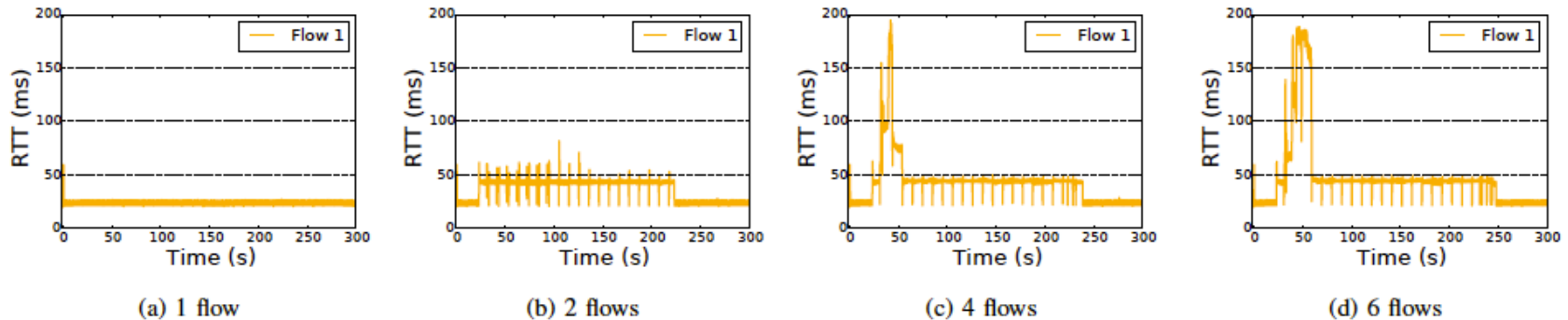
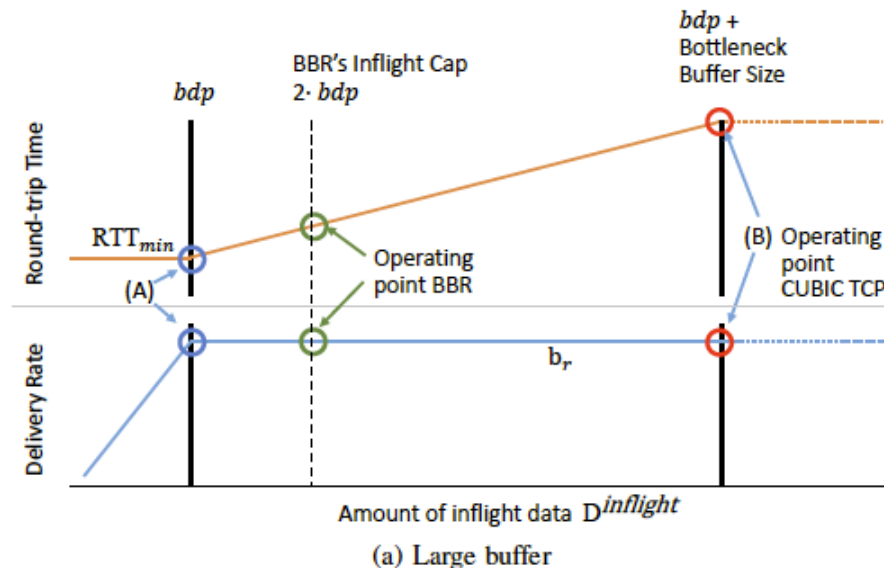


Fig. 5: RTT of different numbers of competing BBR flows at a 1 Gbit/s bottleneck (large buffer), $RTT_{\min} = 20$ ms

- It can be observed that the RTT is increased to a value around 40 ms most of the time
- The peak at the beginning is caused by the startup phase of BBR.

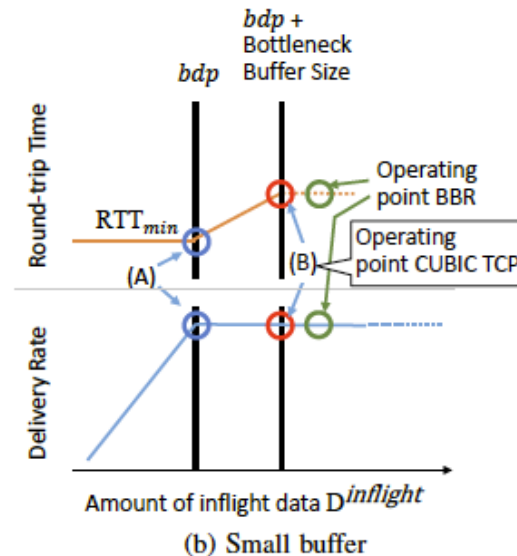
BBR with Large Buffers: Un-Fairness with RTT



If flows with different RTTs share a bottleneck, a flow i with a larger RTT than a flow j will usually get a larger rate share than j .

- Since D_i^{cap} depends on RTT, it follows that $D_i^{cap} > D_j^{cap}$
- Most of this data is queued at the bottleneck, which means that flow i can queue more data than flow j before reaching inflight cap
- This results in a larger rate share for flow i which further increases D_i^{cap}

BBR with Small Buffers



- If the bottleneck buffer is smaller than a bdp the bottleneck buffer is exhausted before the inflight cap is reached.
- This means point (B) is reached and packet loss occurs
- In order to handle non-congestion related packet loss, BBR does not back off if packet loss is detected. **But in this case the packet loss is caused by congestion.**
- Since BBR has no means to distinguish congestion related from non-congestion related loss, point (B) is actually crossed, which can lead to massive amounts of packet loss

BBR v1 does not have any mechanism to deal with packet loss (other than re-transmission)!

Intra Protocol Fairness (Same RTT_{min})

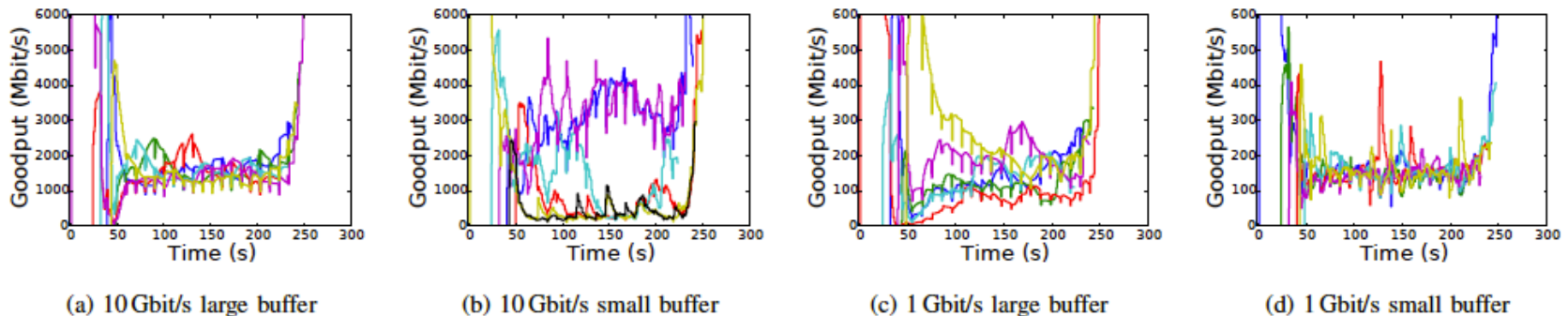


Fig. 7: Goodput of six BBR flows in different scenarios, $RTT_{min} = 20$ ms

- In the scenario with a 10 Gbit/s bottleneck and small buffers (shown in fig. 7b) two of the flows get a significantly larger rate share than the remaining four flows. These flows, in turn, only achieve very small rates, for prolonged timespans.
- Even with large buffers, rate differences are observed.

Intra Protocol Fairness (Same RTTmin): Packet Losses

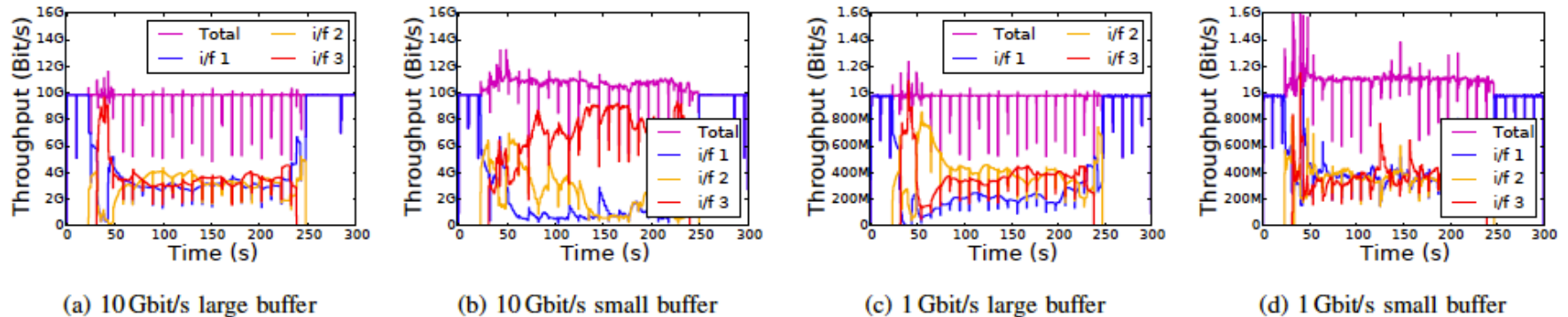


Fig. 8: BBR – Outgoing data of sender interfaces (same experiments as in fig. 7)

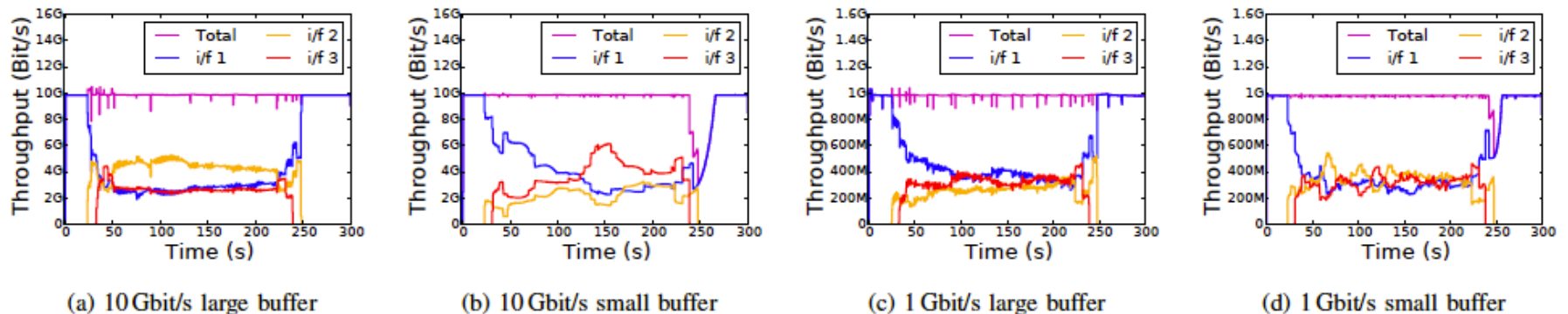


Fig. 9: CUBIC TCP – Outgoing data of sender interfaces (same setting as in fig. 8)

Intra Protocol Fairness (Different RTT_{min})

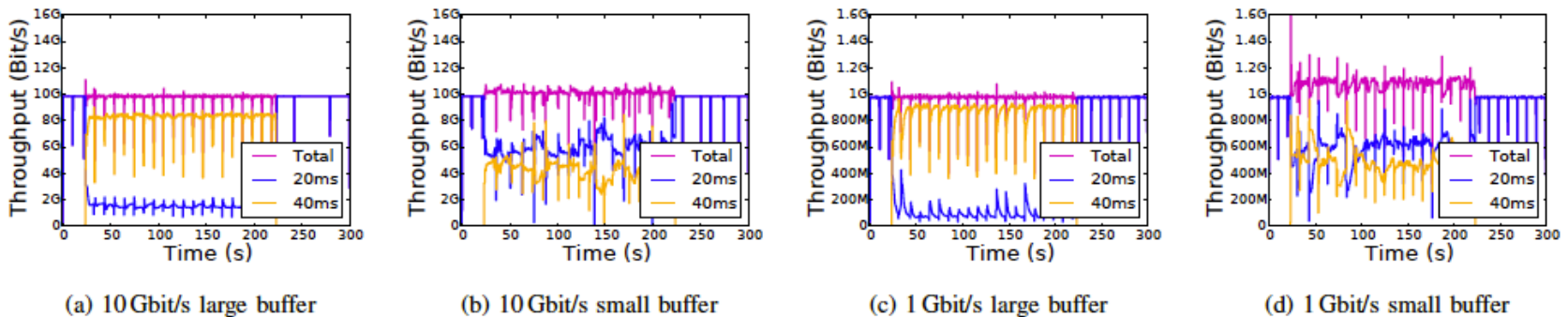


Fig. 13: Two BBR flows with different RTT_{min} (20 ms, 40 ms)

- The fairness among BBR flows with different RTTs strongly depends on the bottleneck buffer size.
- With small buffers (figs. 13b and 13d) flow1 (smaller RTT_{min}) gets more bandwidth than flow2. In this case both flows are most likely not limited by their inflight cap, due to the small buffer size.
- With large buffers (figs. 13a and 13c) flow2 (larger RTT_{min}) gets significantly more bandwidth than flow1, because the buffer size is large enough so that both flows are most likely limited by their inflight cap.
- Both flows can put one bdp into the bottleneck buffer. Due to the larger RTT_{min} , the bdp of flow2 is larger as well. This corresponds to a larger share of data in the bottleneck buffer, thereby resulting in a larger throughput for flow2

BBR and CUBIC Interaction: Large Buffers

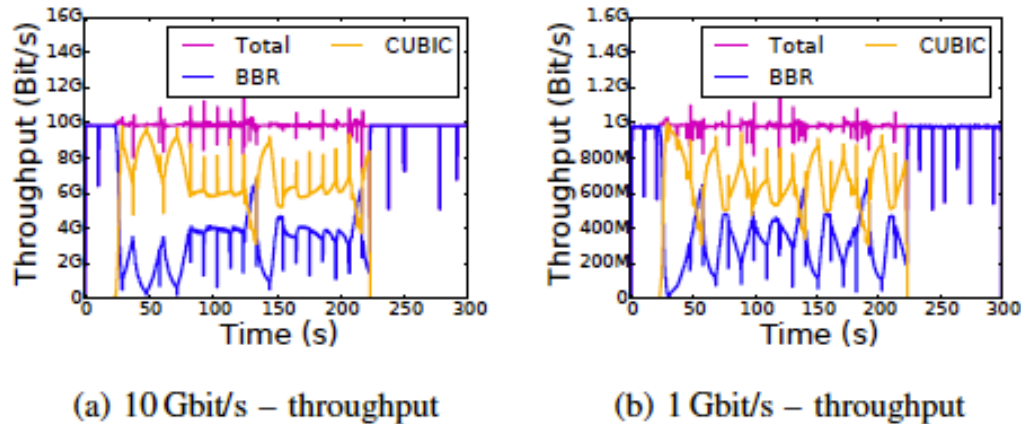
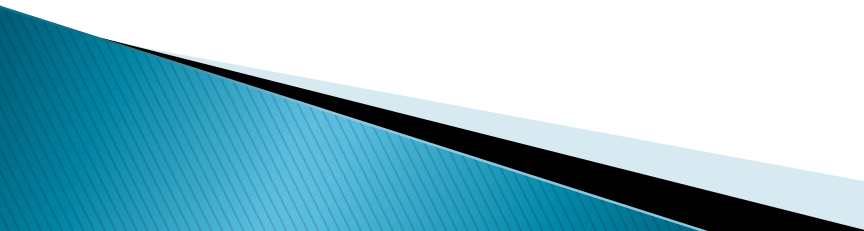


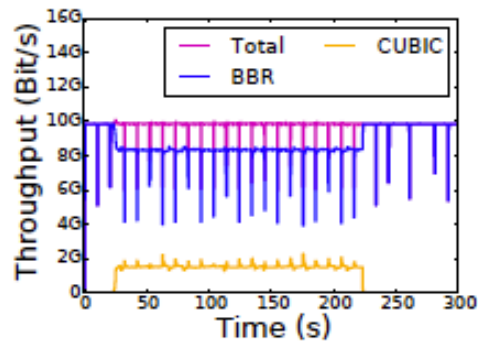
Fig. 15: BBR vs. CUBIC TCP (large buffer)

- As loss-based congestion control, CUBIC TCP tends to fill the bottleneck buffer up to exhaustion, no matter how big the buffer is, whereas BBR limits its inflight data to two bdp.
- This means, the larger the bottleneck buffer, the larger the rate share of CUBIC TCP.
- However, since CUBIC TCP produce long lasting standing queues, a competing BBR flow may not be able to see the actual RTT_{min} , even during its ProbeRTT phase.
- During ProbeRTT the BBR flow reduces its own inflight data close to zero, however, the CUBIC TCP flow does not.
- Thus, the bottleneck buffer is usually not drained completely.
- In this case, the BBR flow assumes a higher RTT_{min} and, thus, also increases the inflight cap to a larger value

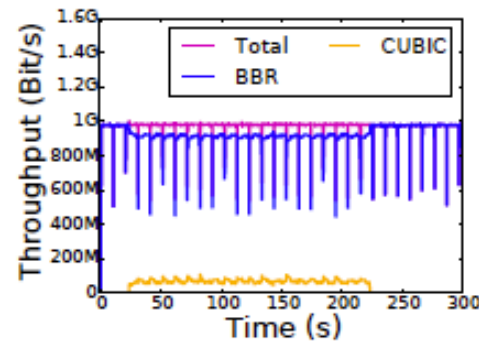
BBR and CUBIC Interaction: Windows Limited Regime

- ▶ Assume a link capacity C , where BBR and the loss-based CCAs, in aggregate, are consuming all of the available capacity.
 - ▶ By probing for 125% of its current share of bandwidth, BBR pushes extra data into the network (offered load $> C$) leading to loss for all senders.
 - ▶ Loss-based algorithms back off, dropping their window sizes and corresponding sending rate.
 - ▶ BBR does not react to losses and instead increases its sending rate, since it successfully sent more data during bandwidth probing than it did in prior cycles.
 - ▶ The loss-based CCA returns to ramping up its sending rate, and together the combined throughput of the two becomes slightly higher than the link capacity and the two flows begin to fill the bottleneck buffer.
 - ▶ This process continues until BBR hits an in-flight cap; we expect that in the absence of a cap it would consume the entire link capacity.
- 

BBR and CUBIC Interaction: Small Buffers



(a) 10 Gbit/s



(b) 1 Gbit/s

Fig. 16: BBR vs. CUBIC TCP (small buffer)

With small buffers, BBR gets a vastly bigger rate share than CUBIC TCP

BBR and CUBIC Interaction: Multiple CUBIC Flows

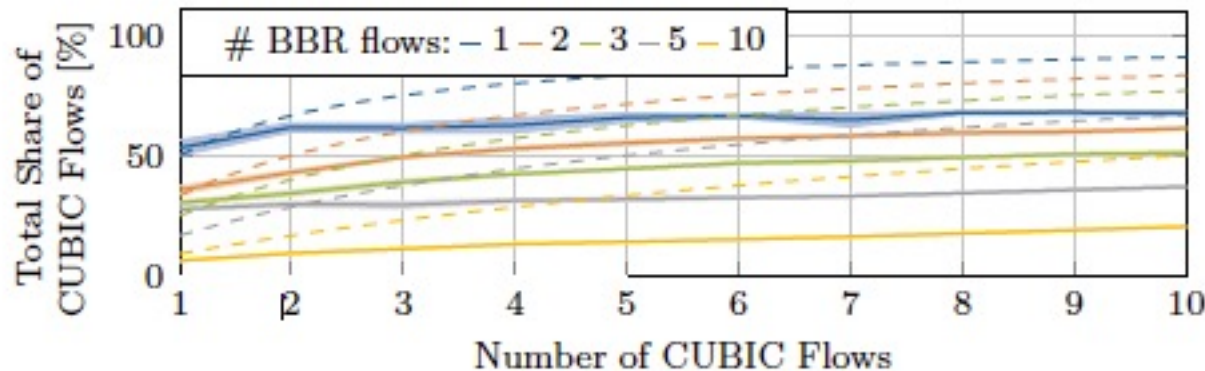
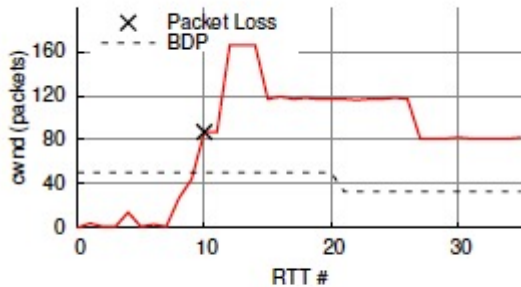


Figure 10: Bandwidth share of different number of CUBIC and BBR flows competing. Dashed lines show fair share.

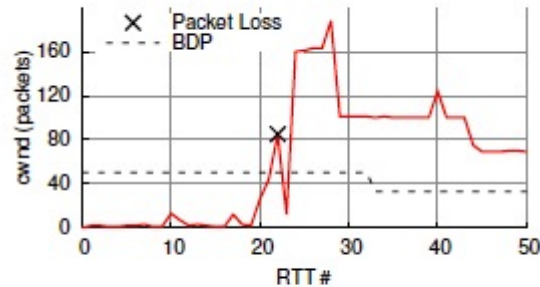
Scholz et. al. ran tests for up to 10 BBR flows competing with up to 10 Cubic flows in a large buffer and conclude that, “independent of the number of BBR and Cubic flows, BBR flows are always able to claim at least 35% of the total bandwidth.”

BBR v2 Goals

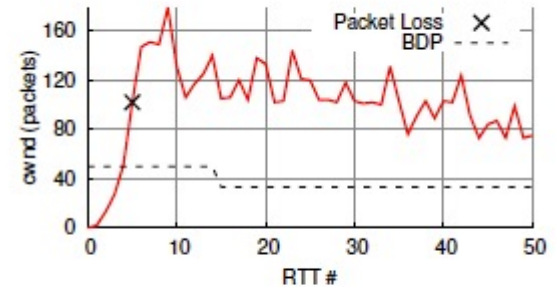
- ▶ Reduce loss rate in shallow buffers
- ▶ Reduce queueing delay
- ▶ Improved fairness with shallow buffers



(a) youtube.com Dec '18.

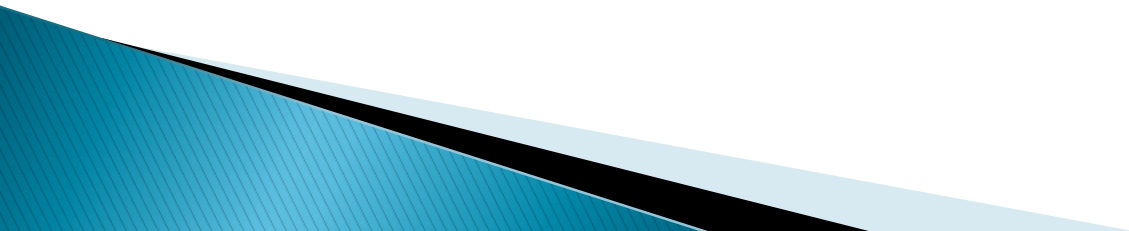


(b) youtube.com Feb '19



(c) BBRv2, Alpha release, July '19.

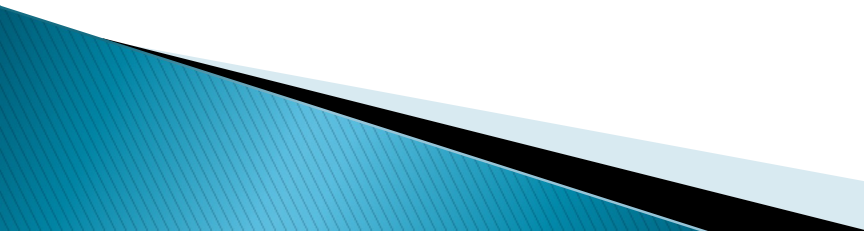
Express Control Protocol XCP



XCP Features

- ▶ The XCP congestion control algorithm is an example of a “clean slate” design, that is, it is a result of a fundamental rethink of the congestion control problem without worrying about the issue of backward compatibility.
- ▶ As a result, it cannot be deployed in a regular TCP/IP network because it requires multi-bit feedback, but it can be used in a self-contained network that is separated from the rest of the Internet (which can be done by using the split TCP architecture).
- ▶ XCP fundamentally changes the nature of the feedback from the network nodes by having them provide explicit window increase or decrease numbers back to the sources.
- ▶ It reverses TCP’s design philosophy in the sense that all congestion control intelligence is now in the network nodes.
- ▶ As a result, the connection windows can be adjusted in a precise manner so that the total throughput at a node matches its available capacity, thus eliminating rate oscillations.
- ▶ This allows the senders to decrease their windows rapidly when the bottleneck is highly congested while performing smaller adjustments when the sending rate is close to the capacity.
- ▶ To improve system stability, XCP reduces the rate at which it makes window adjustments as the round trip latency increases.

XCP Features (cont)

- ▶ Another innovation in XCP is the decoupling of efficiency control (the ability to get to high link utilization) from fairness control (the problem of how to allocate bandwidth fairly among competing flows).
 - ▶ This is because efficiency control should depend only on the aggregate traffic behavior, but any fair allocation depends on the number of connections passing through the node.
 - ▶ Hence, an XCP-based AQM controller has both an **efficiency controller (EC)** and a **fairness controller (FC)**, which can be modified independently of the other.
 - ▶ In regular TCP, the two problems coupled together because the AIMD window increase decrease mechanism is used to accomplish both objectives
- 

XCP Protocol Description: Source

- ▶ Traffic source k maintains a congestion window W_k , and keeps track of the round trip latency T_k . It communicates these numbers to the network nodes via a congestion header in every packet. Note that the rate R_k at source k is given by

$$R_k = \frac{W_k}{T_k}$$

- ▶ Whenever a new ACK arrives, the following equation is used to update the window:

$$W_k \leftarrow W_k + S_k$$

where S_k is explicit window size adjustment that is computed by the bottleneck node and is conveyed back to the source in the ACK packet. Note that S_k can be either positive or negative, depending on the congestion conditions at the bottleneck.

XCP Protocol Description: Network Node

- ▶ Network nodes monitor their input traffic rate. Based on the difference between the link capacity C and the aggregate rate Y given by

$$Y = \sum_{k=1}^K R_k,$$

the node tells the connections sharing the link to increase or decrease their congestion windows, which is done once every average round trip latency, for all the connections passing through the node (this automatically reduces the frequency of updates as the latencies increase). This information is conveyed back to the source by a field in the header.

- ▶ Downstream nodes can reduce this number if they are experiencing greater congestion, so that in the end, the feedback ACK contains information from the bottleneck node along the path.

XCP Protocol Description: Efficiency Controller at Network Node

- ▶ The EC's objective is to maximize link utilization while minimizing packet drops and persistent queues. The aggregate feedback ϕ (in bytes) is computed as follows:

$$\phi = \alpha T_{av} \left(C - \sum_{k=1}^K R_k \right) - \beta b$$

Equivalent to the Proportional + Integral Controller!

- ▶ where α ; β are constants whose values are set based on the stability analysis, T_{av} is the average round trip latency for all connections passing through the node, and b is the minimum queue seen by an arriving packet during the last round trip interval (i.e., it's the queue that does not drain at the end of a round trip interval).
- ▶ The first term on the RHS is proportional to the mismatch between the aggregate rate and the link capacity (multiplied by the round trip latency to convert it into bytes).
- ▶ The second term is useful for the following reason: When there is a persistent queue even if the rates in the first term are matching, then the second term helps to reduce this queue size.

XCP Protocol Description: Fairness Controller at Network Node

- ▶ The job of the FC is to divide up the aggregate feedback among the K connections in a fair manner. It achieves fairness by making use of the AIMD principle, so that:
 - If $\phi > 0$, then the allocation is done so that the increase in throughput for all the connections is the same.
 - If $\phi < 0$, then the allocation is done so that the decrease in throughput of a connection proportional to its current throughput.
- ▶ When ϕ is approximately zero, then the convergence to fairness comes to a halt. To prevent this, XCP does an artificial deallocation and allocation of bandwidth among the K connections by using the feedback h given by

$$h = \max \left(0, 0.1 \sum_{k=1}^K R_k - |\phi| \right),$$

so on every RTT, at least 10% of the traffic is redistributed according to AIMD.

XCP Protocol Description: Increasing the Window Allocation

- ▶ The per packet feedback to source k can be written as

$$\phi_k = p_k - n_k$$

where p_k is the positive feedback and n_k is the negative feedback.

- ▶ First consider the case when $\phi > 0$. This quantity needs to be equally divided among the throughputs of the K connections. The corresponding change in window size of the kth connection is given by

$$\Delta W_k = \Delta R_k T_k \propto T_k$$

because the throughput deltas are equal.

- ▶ This change needs to be split equally among the packets from connection k that pass through the node during time T_k , say m_k (per packet window change is $\Delta W_k / m_k$)
- ▶ Note that m_k is proportional to W_k / MSS_k and inversely proportional to T_k . It follows that the change in window size per packet for connection k is given by

$$p_k = K_p \frac{T_k^2 MSS_k}{W_k}$$

where K_p is a constant.

XCP Protocol Description: Increasing the Window Allocation (cont)

- ▶ It follows that

$$\text{the total increase in aggregate traffic rate} = \frac{h + \max(\phi, 0)}{T_{av}} = \sum^L \frac{p_k}{T_k}$$

- ▶ From which we obtain

$$K_p = \frac{h + \max(\phi, 0)}{T_{av} \sum^L \frac{T_k MSS_k}{W_k}}$$

XCP Protocol Description: Decreasing the Window Allocation

- ▶ When $\phi < 0$, then the decrease in throughput should be proportional to the current throughput.
- ▶ It follows that

$$\Delta W_k = \Delta R_k T_k \propto R_k T_k = W_k$$

- ▶ Splitting up this change in window size among all the m_k packets that pass through the node during an RTT, it follows that

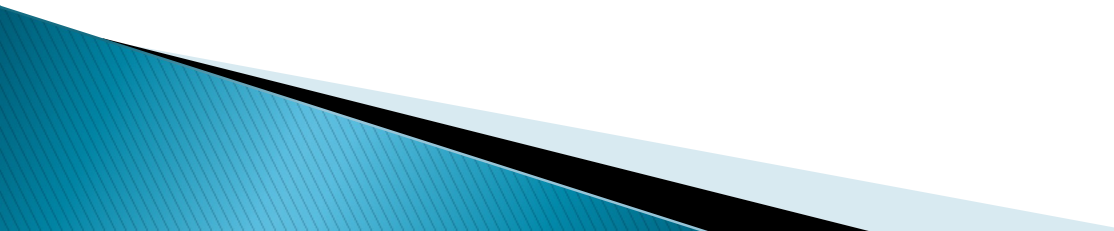
$$n_k = \frac{K_n W_k}{m_k} = K_n T_k MSS_k$$

where the constant K_n is given by

$$K_n = \frac{h + \max(-\phi, 0)}{T_{av} \sum^L MSS_k}$$

Rate Control Protocol RCP

RCP

- ▶ The RCP protocol was inspired by XCP and builds on it to assign traffic rates to connections in a way such that they can quickly get to the ideal Processor Sharing (PS) rate.
 - ▶ RCP is an entirely rate-based protocol (i.e., no windows are used).
 - ▶ Network nodes compute the ideal PS-based rate and then pass this information back to the source, which immediately changes its rate to the minimum PS rate that was computed by the nodes that lie along its path.
- 

RCP

RCP was designed to overcome the following issue with XCP:

- ▶ When a new connection is initiated in XCP and the nodes along its path are already at their maximum link utilizations, then XCP does a gradual redistribution of the link capacity using the bandwidth-shuffling process
- ▶ This process can be quite slow because new connections start with a small window and get to a fair allocation of bandwidth in a gradual manner by using the AIMD principle.
- ▶ However, it takes several round trips before convergence happens, and for smaller file sizes, the connection may never get to a fair bandwidth distribution.
- ▶ One of the main innovations in RCP, compared with XCP, is that the bandwidth reallocation happens in one round trip time, in which a connection gets its equilibrium rate.

RCP Description

- ▶ Every network node periodically computes a fair-share rate $R(t)$ that it communicates to all the connections that pass through it. This computation is done approximately once per round trip delay.
- ▶ RCP mandates two extra fields in the packet header:
 - Field 1: Each network node fills field 1 with its fair share value $R(t)$, and when the packet gets to the destination, it copies this field into the ACK packet and sends it back to the source.
 - Field 2: As in XCP, it is used to communicate the current average round trip latency T_k , from a connection to all the network nodes on its path. The nodes use this information to compute the average round trip latency T_{av} of all the connections that pass through it.
- ▶ Each source transmits at rate R_k , which is the smallest offered rate along its path.

Rate Allocation

- ▶ If the router has perfect information on the number of ongoing flows at time t , and there is no feedback delay between the congested link and the source, then the rate assignment per flow would simply be:

$$R(t) = \frac{C}{N(t)}$$

- ▶ But the router does not know $N(t)$ and it is complicated to keep track of. And even if it could, there is a feedback delay and so by the time $R(t)$ reached the source, $N(t)$ would have changed.
- ▶ In RCP routers have an adaptive algorithm that updates the rate assigned to the flows, to approximate processor sharing in the presence of feedback delay, without any knowledge of the number of ongoing flows

RCP Description

$$Y(t) = \sum_{k=1}^K R_k(t)$$

- ▶ Each network node updates its local rate $R(t)$ according to the equation below:

$$R(t) = R(t - T_{av}) + \frac{\left[\alpha(C - Y(t)) - \beta \frac{b(t)}{T_{av}} \right]}{\hat{N}(t)}$$

Change in rate per connection, updated once per RTT

where $\hat{N}(t)$ is the node's estimate of the number of connections that pass through it and the other variables are as defined for XCP.

- ▶ The basic idea behind this equation is the following:
 - If there is spare bandwidth available (equal to $C - Y(t)$), then share it equally among all the connections
 - If $C - Y(t) < 0$, then the link is over subscribed, and each connection is asked to decrease its rate evenly
 - If there is a queue build-up of $b(t)$, then a rate decrease of $b(t)/T_{av}$ will bring it down to zero within a round trip interval.

RCP Description

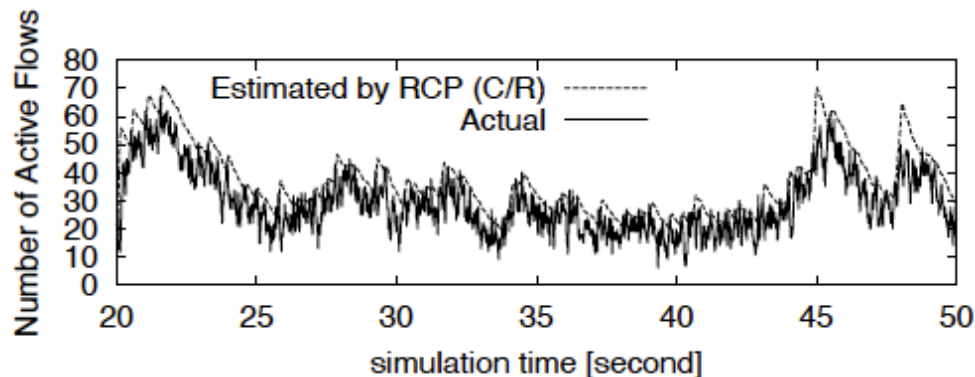
RCP replaces the interval at which the node recomputes $R(t)$ to T' , so that it is user configurable and uses the following estimate for the number of connections:

$$\hat{N}(t) = \frac{C}{R(t - T')}$$

so that [equation 91](#) can be written as

$$R(t) = R(t - T') \left[1 + \frac{\frac{T'}{T_{av}} \left(\alpha(C - Y(t)) - \beta \frac{b(t)}{T_{av}} \right)}{C} \right] \quad (92)$$

where the rate change has been scaled down by T'/T_{av} because it is done more than once per T_{av} seconds.



RCP Description

- ▶ RCP replaces the interval at which the node recomputes $R(t)$ to T_0 , so that it is user configurable and uses the following estimate for the number of connections:

$$\hat{N}(t) = \frac{C}{R(t - T')}$$

- ▶ The rate equation becomes

$$R(t) = R(t - T') \left[1 + \frac{\frac{T'}{T_{av}} \left(\alpha(C - Y(t)) - \beta \frac{b(t)}{T_{av}} \right)}{C} \right]$$

where the rate change has been scaled down by T'/T_{av} because it is done more than once per T_{av} seconds.

Further Reading

- ▶ “BBR: Congestion based Congestion Control”
Cardwell et.al.
 - ▶ Chapter 5, Sections 5.7–5.8 of Internet Congestion Control
- 