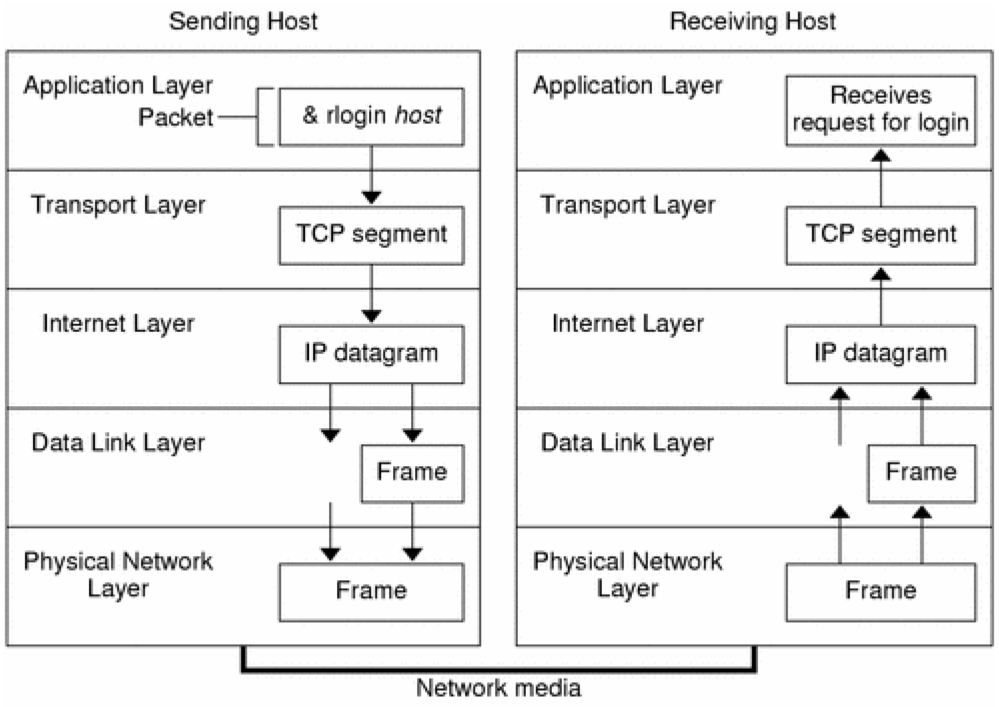
BBR:TCP Congestion Control

@Yonghui

TCP: **T**ransmission **C**ontrol **P**rotocol

- Connection
- Reliable transmission
  - Sequence number
  - Ack
- Flow control
- Congestion control



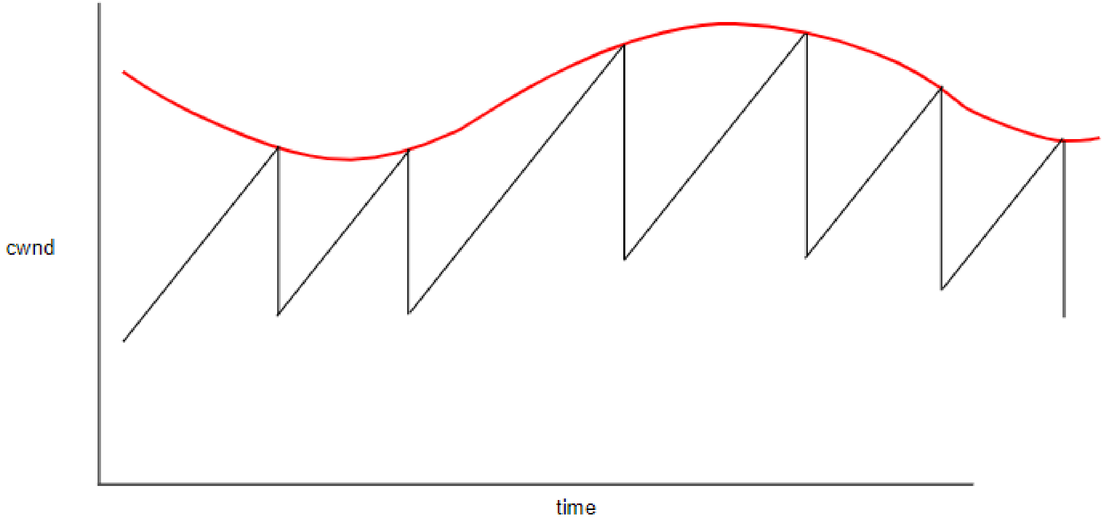
What is TCP Congestion control

- We have limited bandwidth from sender to receiver
- We need to avoid network congestion as much as possible
  - Avoid send too much data into network to cause meaning less congestion
- Send data in a proper speed, to maximum network utilization
- Fairness: multiple can share a network infrastructure

Existing Congestion Control Algorithm

- Congestion Window:
  - all the congestion control algorithm finally control this parameter
  - It control how many data can be sent but not acked
  - i.e., Max-inflight
  - Most congestion control works in the sender side
- How to set a CWND?
  - How to scale up? How to scale down
- Loss based congestion control
  - Reduce the CWND(Congestion window) when loss is detected
  - Liner increase the CWND until next congestion happen
    - 在拥塞的边缘反复试探

Reno



TCP Sawtooth, red curve represents the network capacity

- At least, we call it reno, from textbook
- additive increase/multiplicative decrease (AIMD)
  - 线性加，乘性减
- Scale up: Liner increase
  - MSS/CWND each ACK，result in MSS per RTT
- Scale Down: Half the CWND when congestion happen

maximum segment size (MSS)：Usually 1450 bytes or so

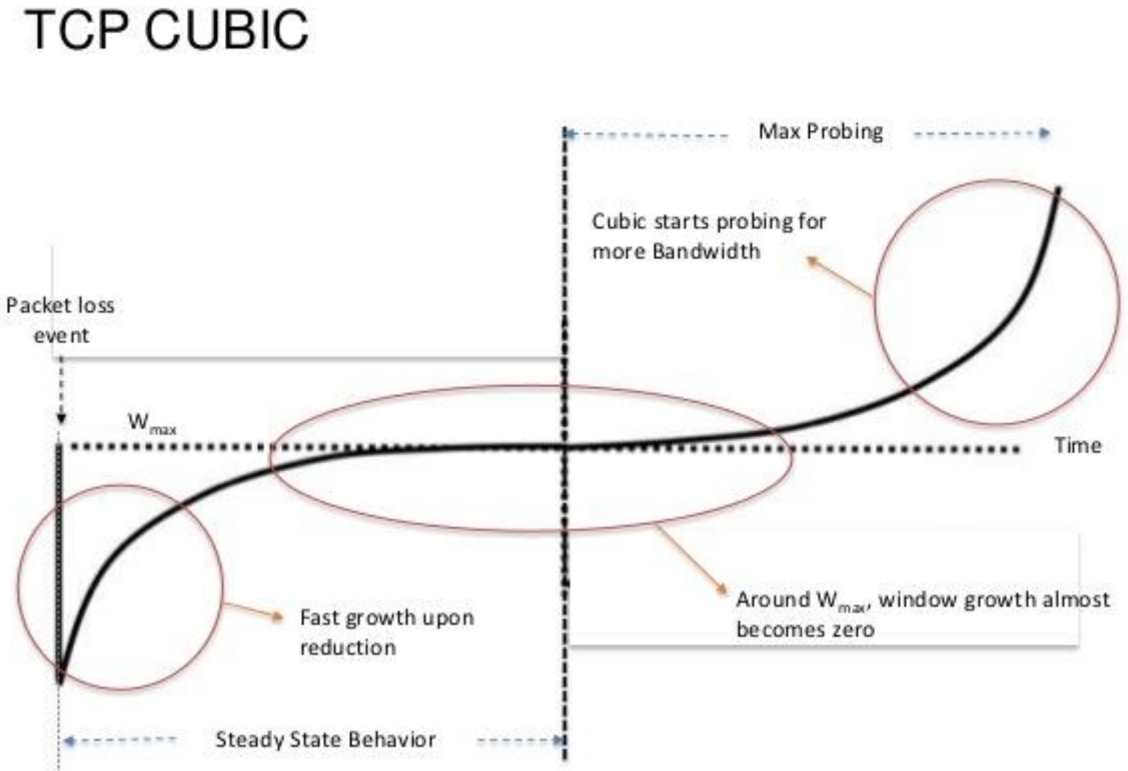
round-trip delay time (RTT)：1ms 200km； 200ms over Pacific

Reno etc.: Starting

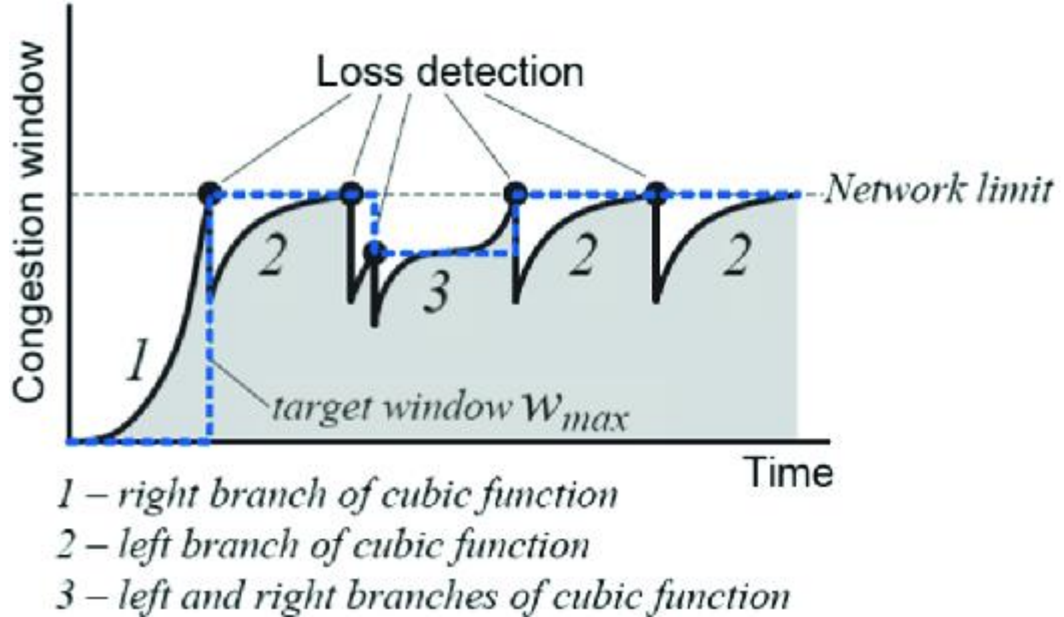
- Slow start:
  - “Slow start” increase CWND very rapidly
  - Increase CWND on every ACK, result in exponential CWND increase
    - 1xRTT;1 sent; 1 ack; CWND = 2
    - 2xRTT;2,3 sent; 2,3 ack; CWND = 4
    - 3xRTT;4,5,6,7 sent; 4,5,6,7 ack; CWND = 8
    - 4xRTT;8,9,10,11,12,13,14,15,16 sent; …
- ssthresh (slow start threshold)
  - When CWND > ssthresh, then increase cwnd linerly

CUBIC:
Linux kernel default

- How to scale up? How to scale down? This is a question
- When loss is detected, Scale down CWND and slowly approach last CWND
- If loss is not detected when last CWND is reached, speed up the scale up



TCP CUBIC
Max Probing
Cubic starts probing for more Bandwidth
Packet loss event
W_{max}
Time
Fast growth upon reduction
Around W_{max}, window growth almost becomes zero
Steady State Behavior



Loss detection
Congestion window
Network limit
1
2
3
2
2
target window W_{max}
Time
1 – right branch of cubic function
2 – left branch of cubic function
3 – left and right branches of cubic function

CUBIC

- Friendly to long distance link, which have large RTT
  - Reno is RTT based, the CWND scale up is driven by ack
  - CUBIC CWND is time based, much friendly to the large RTT link

$$cwnd \ = \ C(T-K)^3 + w_{max}$$

$$\textbf{where } K = \sqrt[3]{\frac{w_{max}(1-\beta)}{C}}$$

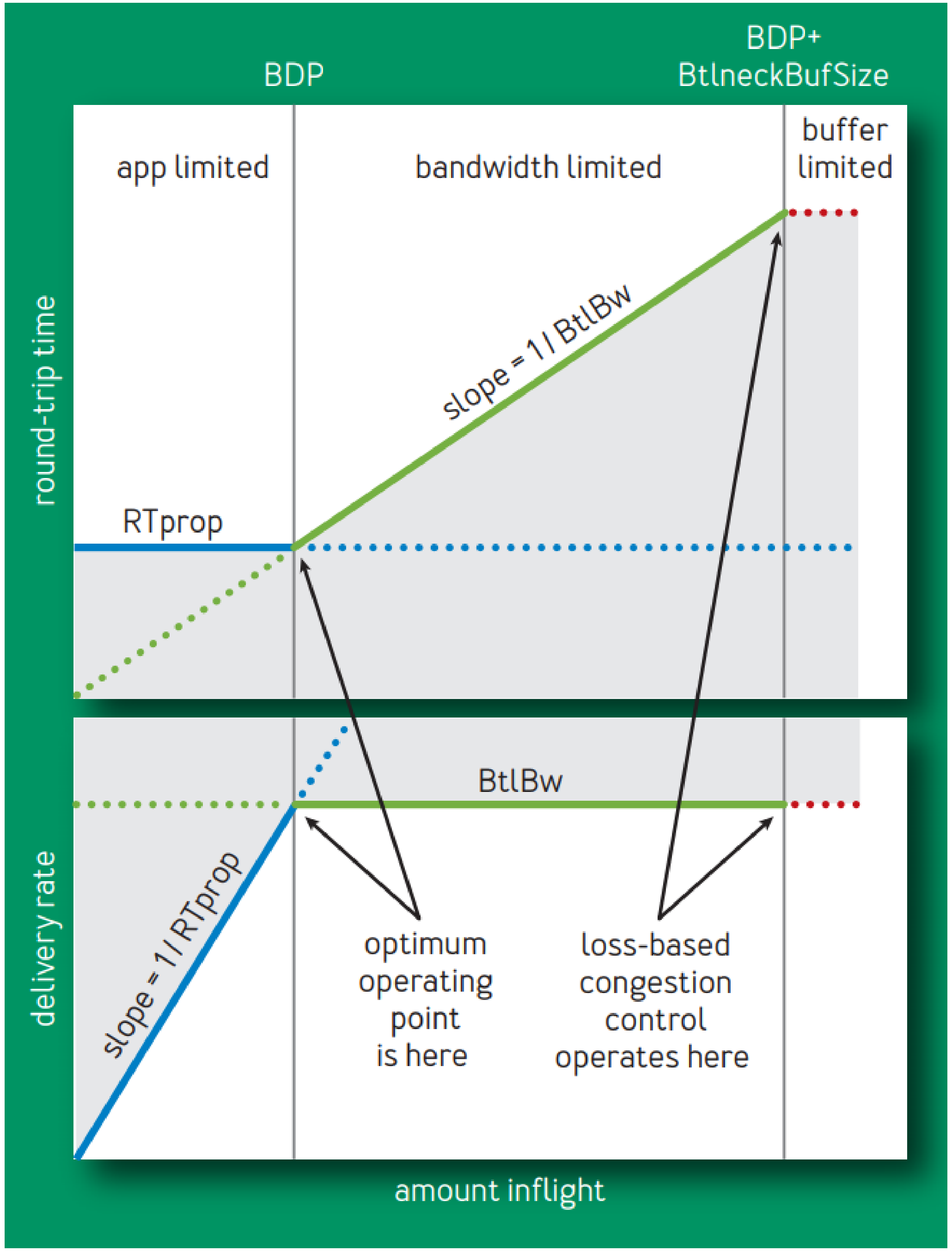
- When T = 0, cwnd = β w_{max}
  - w_{max}: Window size just before the last reduction
  - T: time

Loss based control is not optimal

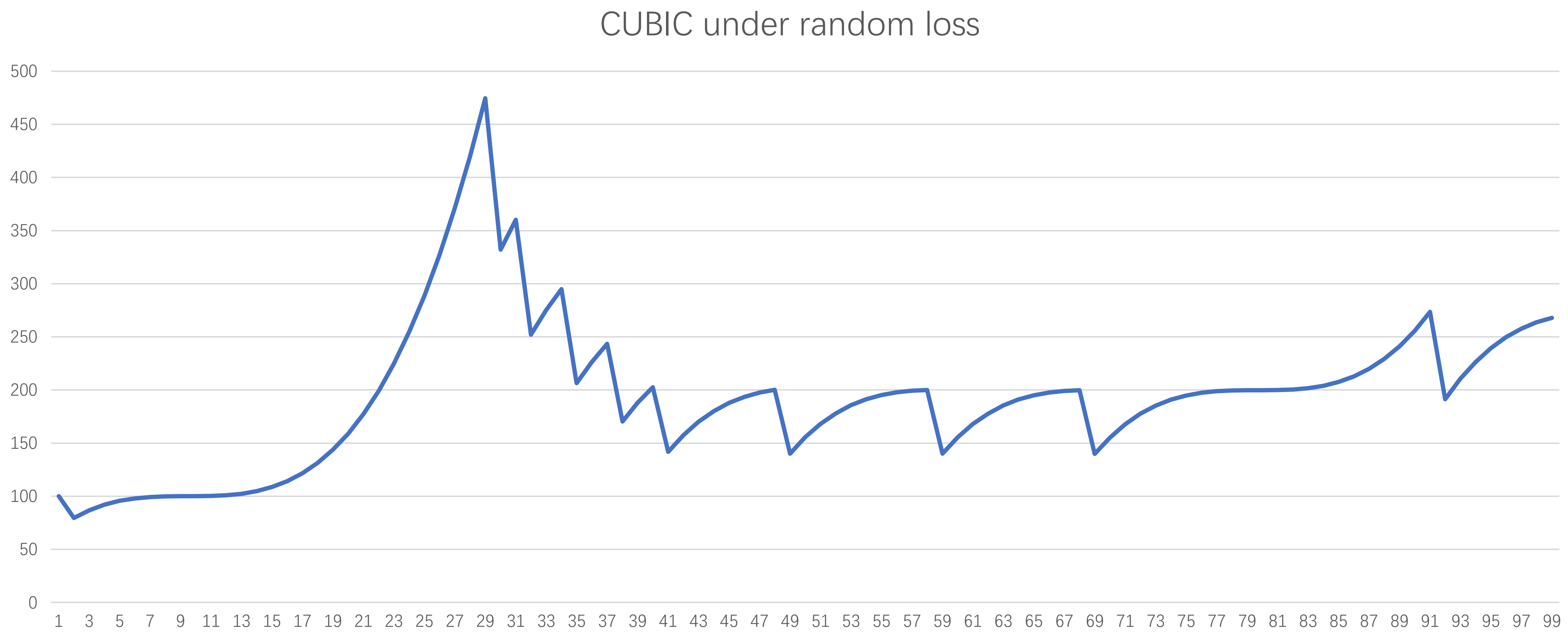
- We have buffer over the network
  - You are not going to loss packet before the buffer is full
  - Your latency will increase when datagram start queueing

- **RTprop**: round-trip propagation time
  - Physical time without queueing
- **BtlBw**: bottleneck bandwidth
- **BDP**:bandwidth-delay product

FIGURE 1: DELIVERY RATE AND ROUND-TRIP TIME VS. INFLIGHT



What if we have random loss?



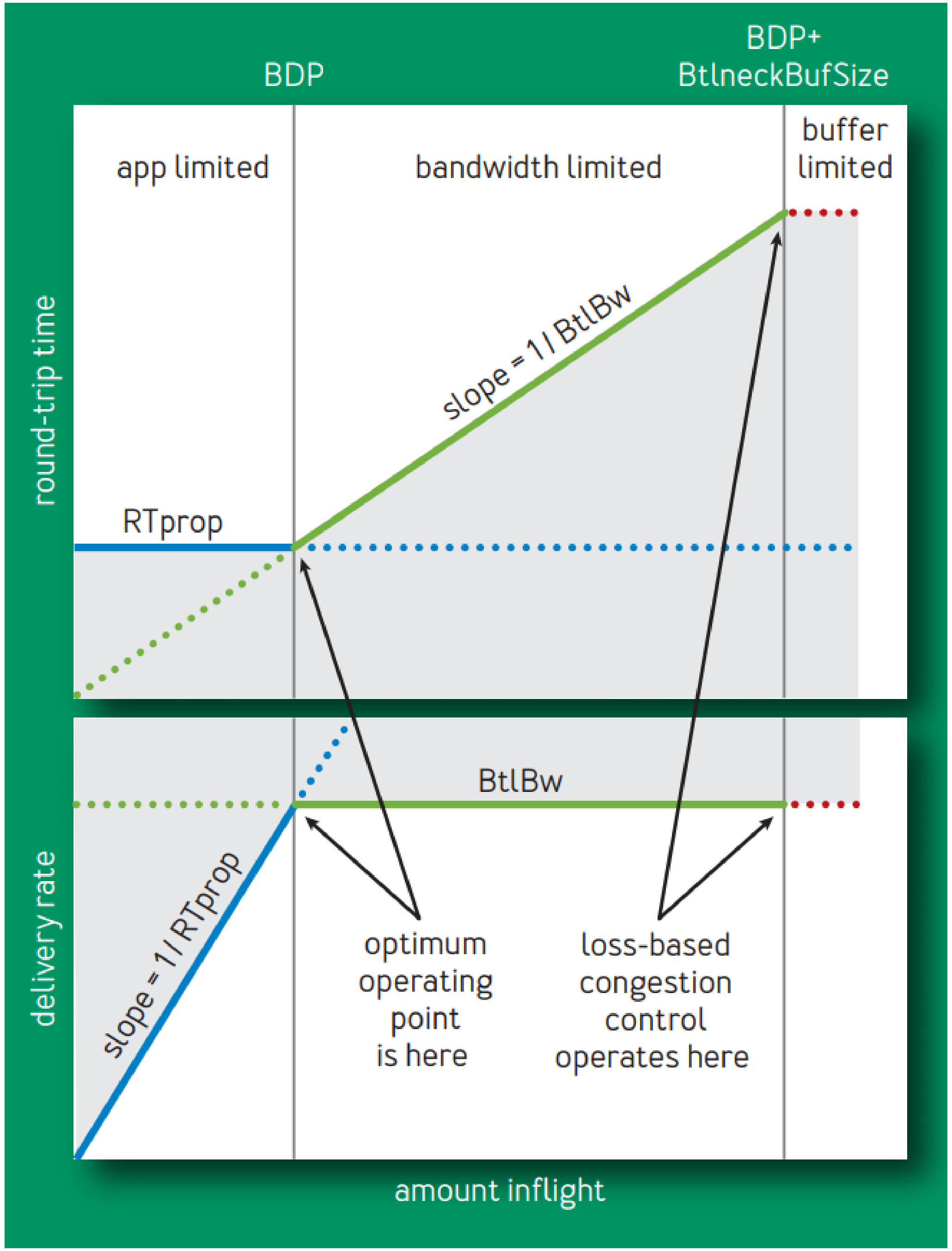
Model the bottleneck and TCP connection

- bottleneck
  - It determines the connection’s maximum data-delivery rate.
  - It’s where persistent **queues** form
    - Data only queueing at the bottleneck
- Queuing will increase the RTT
- From TCP point of view, complex link can be simplified as single link with RTT and bandwidth
  - **RtProp**
  - **BtlBw**

How BBR set the congestion window?

- **BBR: b**ottleneck **b**andwidth and **r**ound-trip propagation time
- Set the CWND = BDP
- BDP = RtProp × BtlBw

FIGURE 1: **DELIVERY RATE AND ROUND-TRIP TIME VS. INFLIGHT**



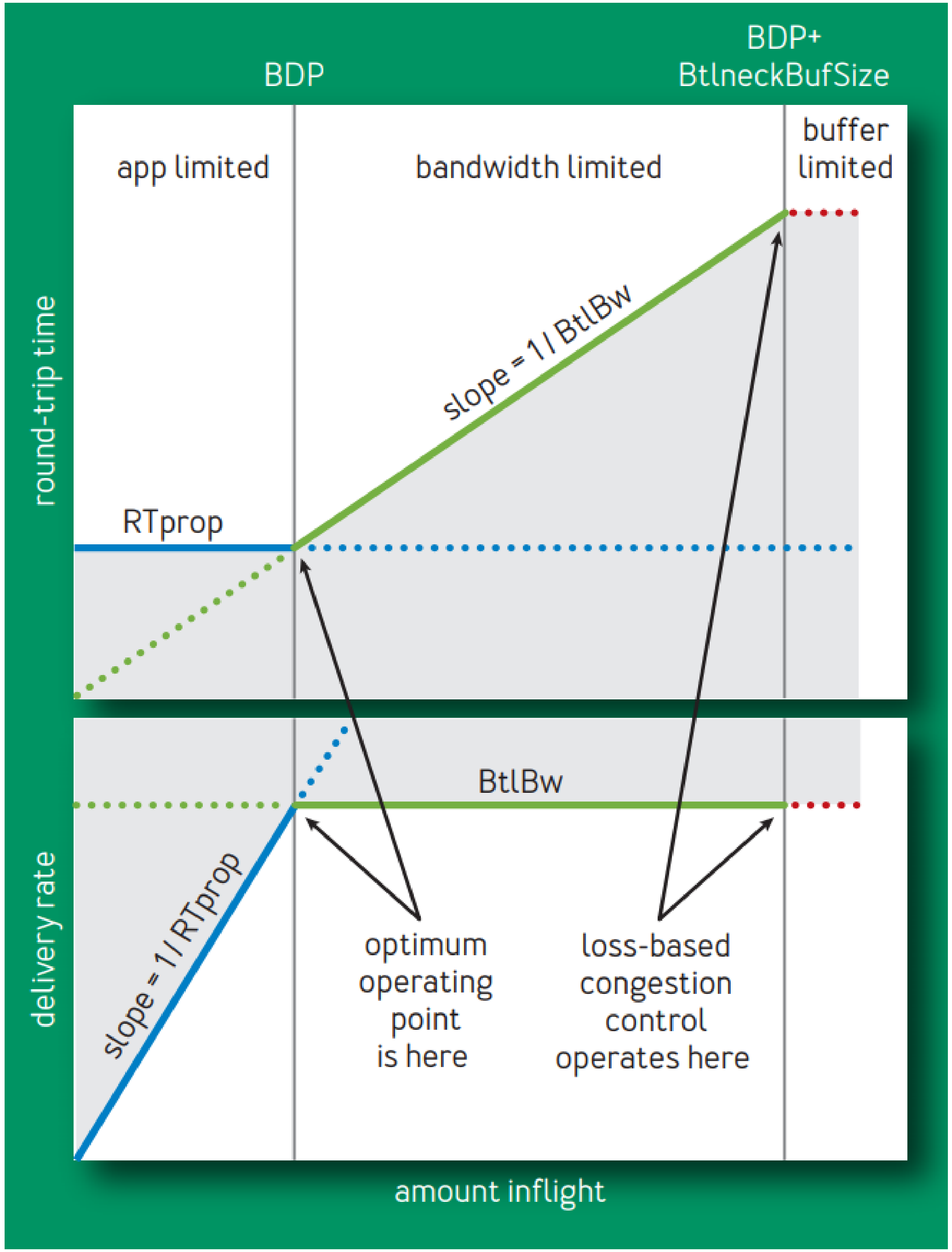
How to estimate RTprop and BtlBW?

- $\widehat{RT_{prop}} = RT_{prop} + min(\eta_t) = min(RTT_t) \quad \forall t \in [T - W_R, T]$
  - Choose the minimum RTTt as Rtprop
  - TCP will track RTT by measure the time from packet send to it get acked
- $\widehat{BtlBw} = max(deliverRate_t) \quad \forall t \in [T - W_R, T]$
  - Choose the maximum BW as BtlBW

“测不准原理”

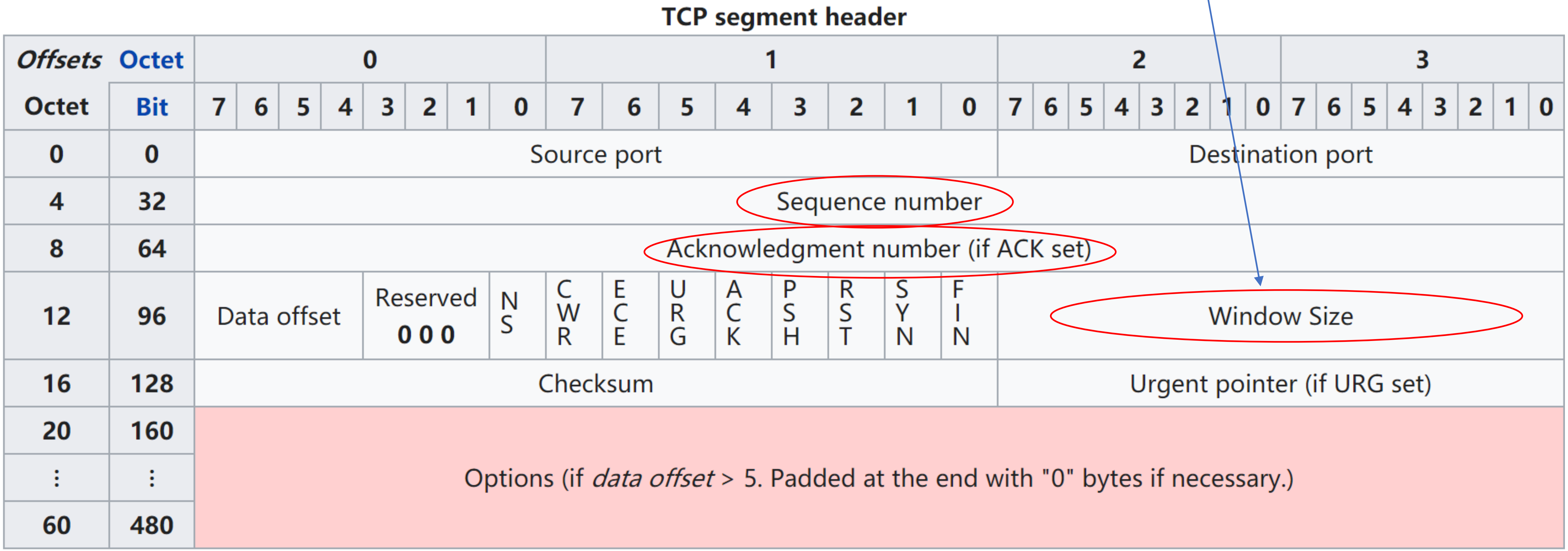
- Since RTprop is visible only to the left of BDP and BtlBw only to the right in figure 1, they obey an uncertainty principle: **whenever one can be measured, the other cannot**.

FIGURE 1: DELIVERY RATE AND ROUND-TRIP TIME VS. INFLIGHT



Revisit TCP Header

This receive window
is for **Flow** control



TCP segment header

Offsets	Octet	0								1								2								3							
Octet	Bit	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
0	0	Source port																Destination port															
4	32	Sequence number																															
8	64	Acknowledgment number (if ACK set)																															
12	96	Data offset				Reserved 0 0 0			NS	CWR	ECE	URG	ACK	PSH	RST	SYN	FIN	Window Size															
16	128	Checksum																Urgent pointer (if URG set)															
20	160	Options (if *data offset* > 5. Padded at the end with "0" bytes if necessary.)																															
⋮	⋮																																
60	480																																

BBR: On Ack

```
function onAck(packet)
  rtt = now - packet.sendtime
  update_min_filter(RTpropFilter, rtt)
  delivered += packet.size
  delivered_time = now
  deliveryRate = (delivered - packet.delivered)
/(now - packet.delivered_time)
  if (deliveryRate > BtlBwFilter.currentMax || !
packet.app_limited)
    update_max_filter(BtlBwFilter, deliveryRate)
  if (app_limited_until > 0)
    app_limited_until - = packet.size
```

- RTT & BW measure by tracking packet send time & sequence
- Sometime deliveryRate is limited by application.
    - BBR tracking if the deliveredRate is limited by application and take all the link limited sample.

BBR: Sending

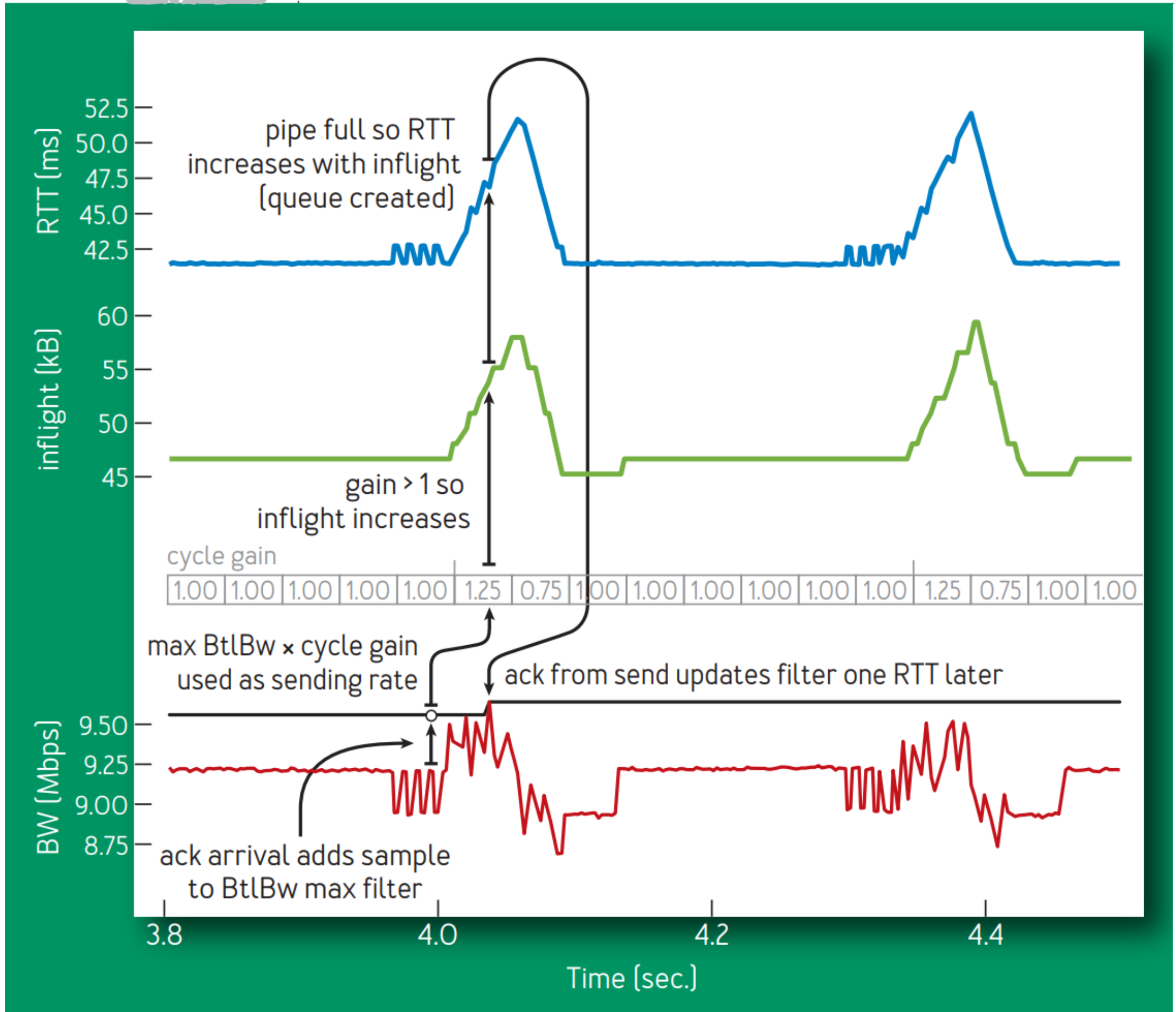
```
function send(packet)
  bdp = BtlBwFilter.currentMax * RTpropFilter.currentMin
  if (inflight >= cwnd_gain * bdp)
    // wait for ack or timeout
    return
  if (now >= nextSendTime)
    packet = nextPacketToSend()
    if (! packet)
      app_limited_until = inflight
      return
    packet.app_limited = (app_limited_until > 0)
    packet.sendtime = now
    packet.delivered = delivered
    packet.delivered_time = delivered_time
    ship(packet)
    nextSendTime = now + packet.size /(pacing_gain * BtlBwFilter.currentMax)
  timerCallbackAt(send, nextSendTime)
```

- cwnd_gain: small multiplier to deal with network condition
  - Loss, etc.
- pacing_gain is controlled by BBR, to do the RTprobe and BWprobe
  - Can be larger than 1 or smaller than 1 depend on BBR state
    - Like 1.25 or 0.75

BBR control cycle in steady-state

- Increase the pace_gain to probe Bandwidth
- Decrease pace_gain to probe RTT

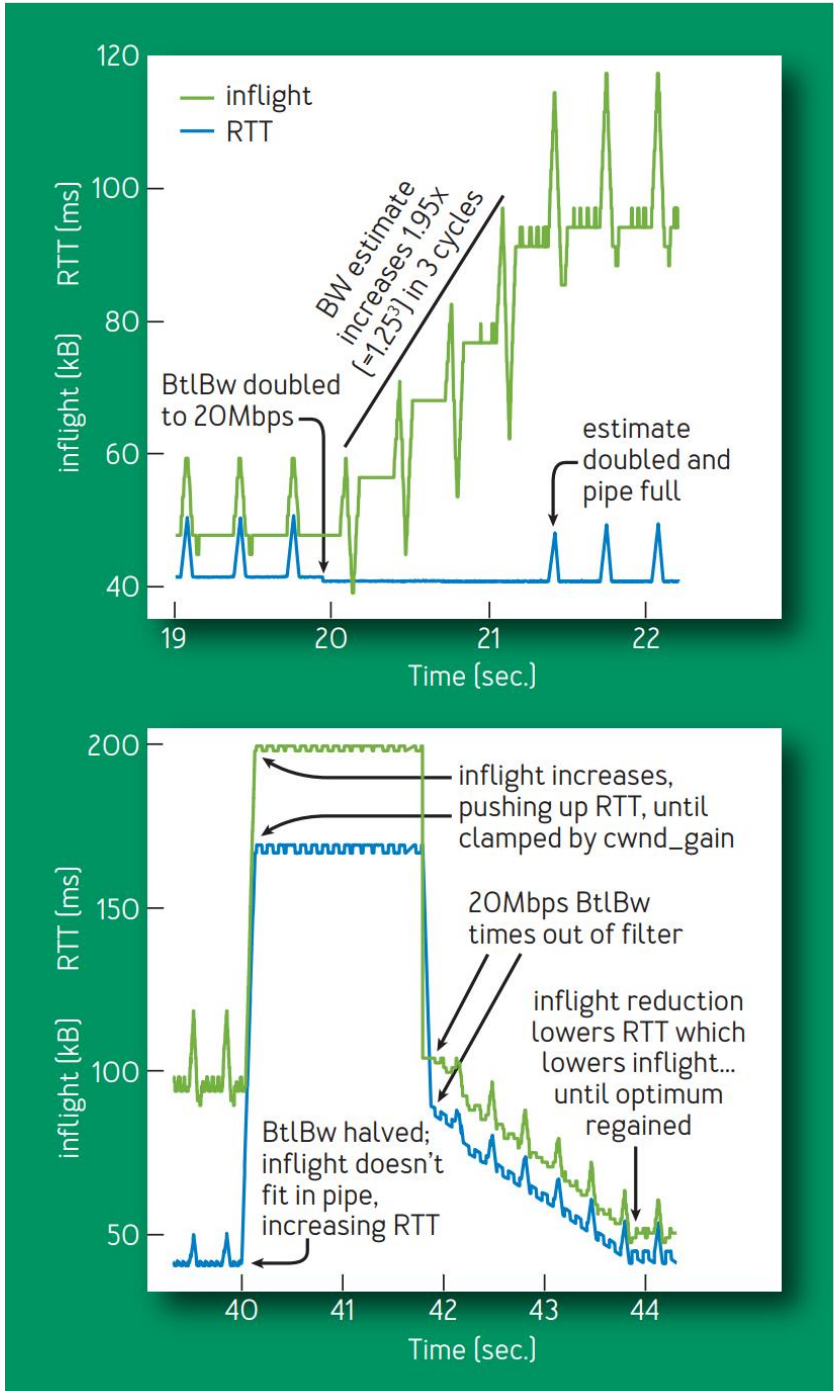
FIGURE 2: **RTT (BLUE), INFLIGHT (GREEN) AND DELIVERY RATE (RED) DETAIL**



BBR: bandwidth Change

- Figure3 shows
  - 10Mpbs -> 20Mpbs
  - 20Mpbs -> 10Mpbs

FIGURE 3: **BANDWIDTH CHANGE**



BBR: Single connection startup

- Startup:
  - Binary search with a gain of 2/ln2
  - This discovers BtlBw in $\log_2$ BDP RTTs
  - but creates up to 2BDP excess queue in the process
- Drain
  - Use inverse startup gain to get rid of queue

FIGURE 4: **FIRST SECOND OF A 10-MBPS, 40-MS BBR FLOW**

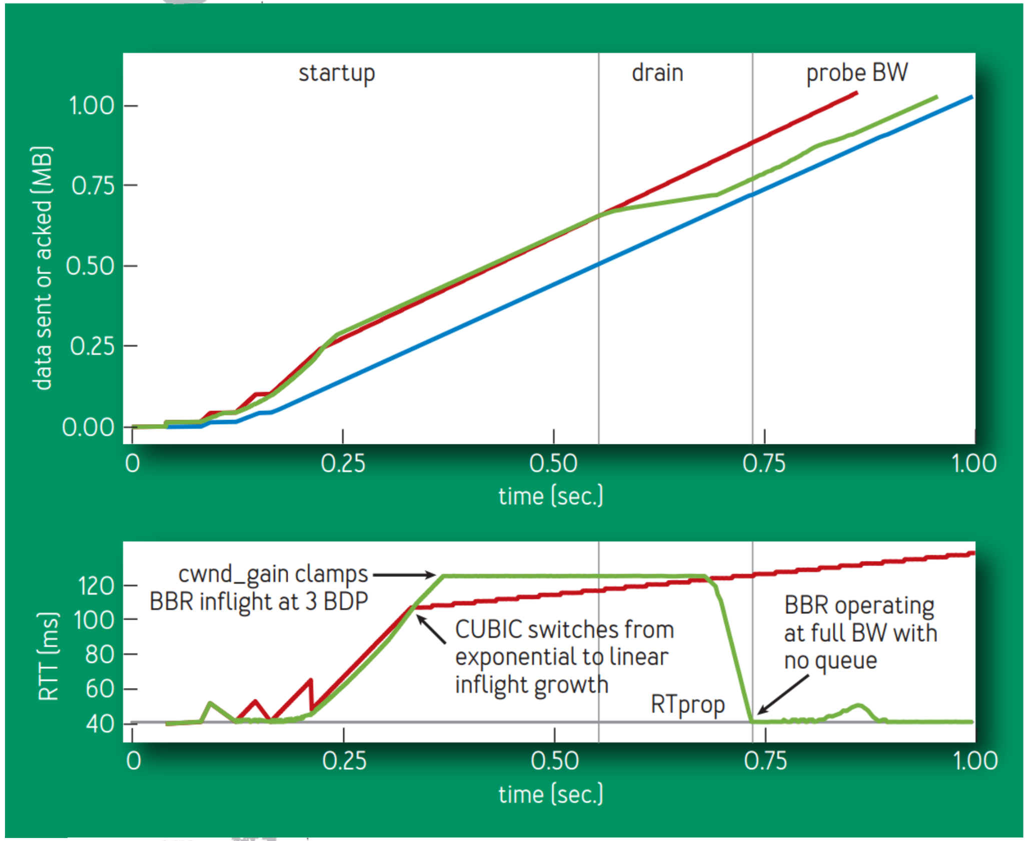
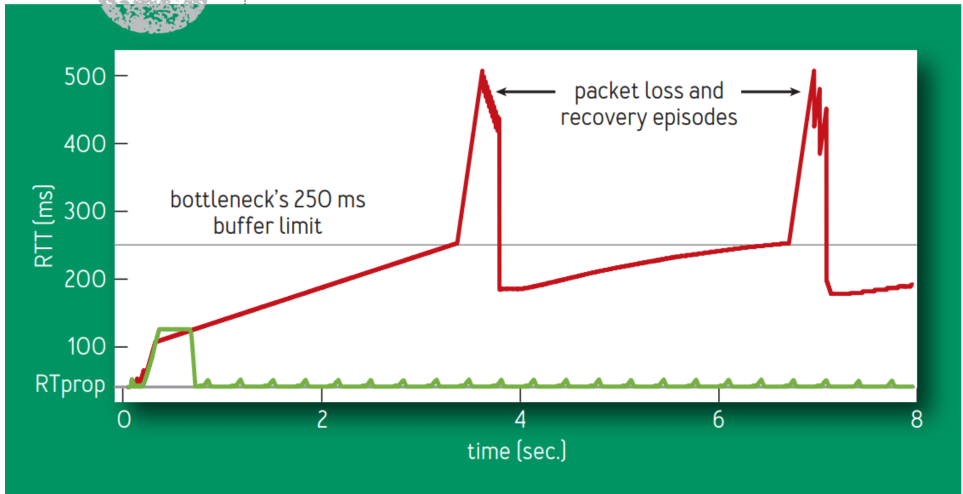


FIGURE 5: **FIRST 8 SECONDS OF 10-MBPS, 40-MS CUBIC AND BBR FLOWS**

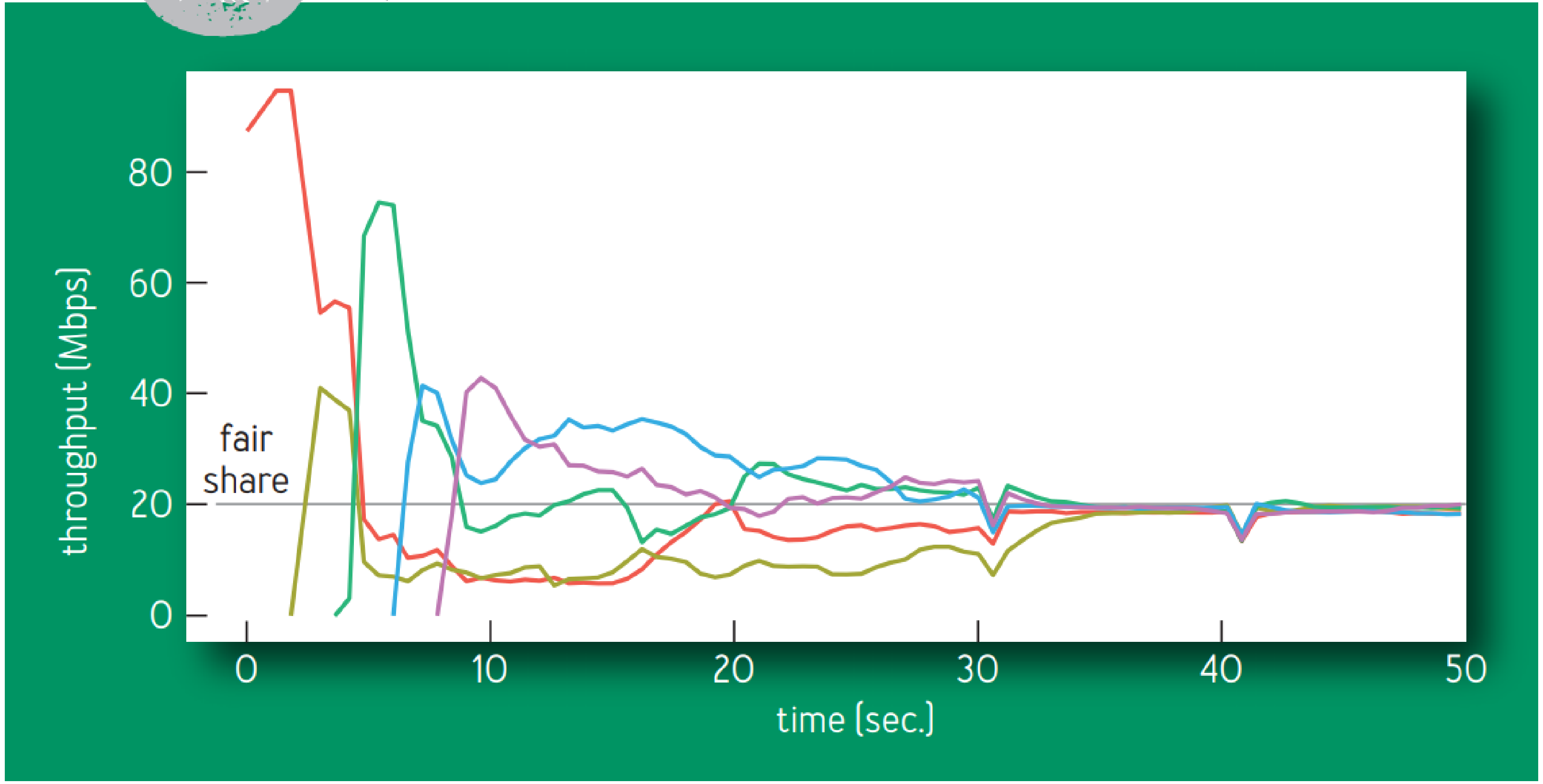


BBR: Multiple connection

- when the RTProp estimate has not been updated (i.e., by measuring a lower RTT) for many seconds, BBR enters ProbeRTT, which reduces the inflight to four packets for at least one round trip, then returns to the previous state.
- BBR self synchronizes
  - Large flows entering ProbeRTT drain many packets from the queue, so several flows **see** a new RTprop (new minimum RTT). This makes their RTprop estimates **expire** at the same time, so they enter ProbeRTT together

BBR: self synchronizes ProbeRT

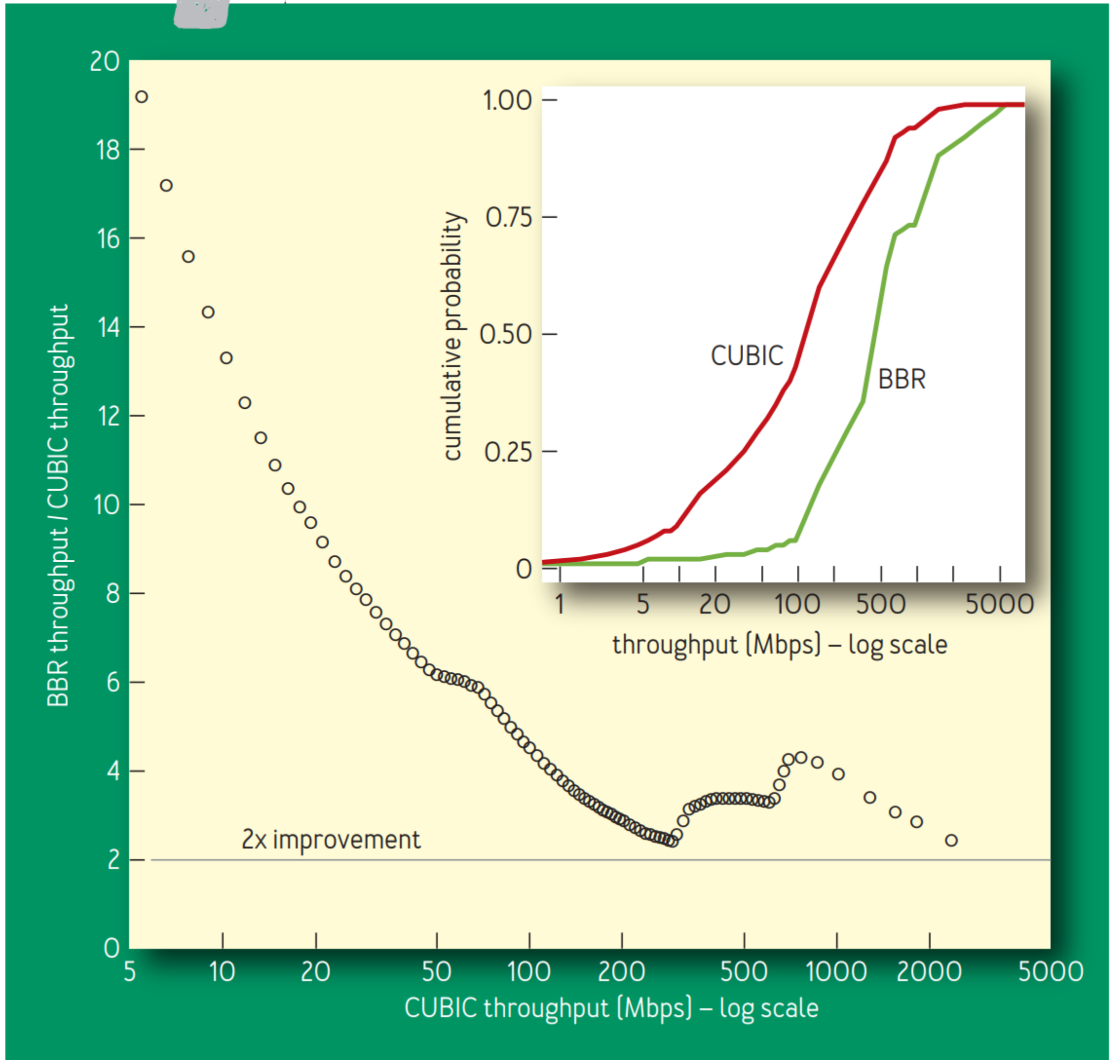
FIGURE 6: **THROUGHPUTS OF 5 BBR FLOWS SHARING A BOTTLENECK**



Compare BBR with CUBIC

- Google gain 2-25x CUBIC Bandwidth in their intercontinental network
  - Also compare cumulative distribution functions of BBR and CUBIC

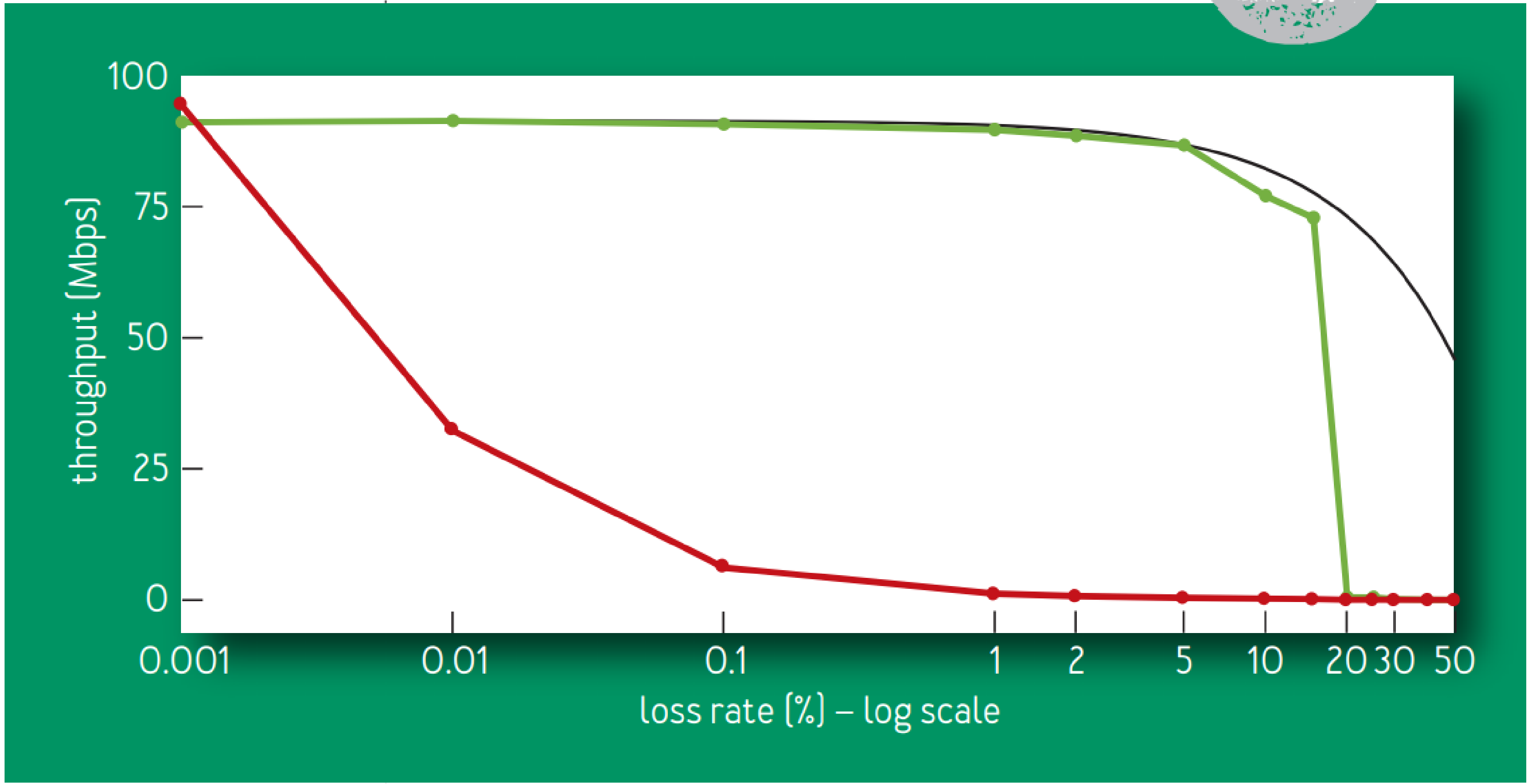
FIGURE 7: **BBR VS. CUBIC RELATIVE THROUGHPUT IMPROVEMENT**



Random loss tolerance

- CUBIC have very poor loss tolerance

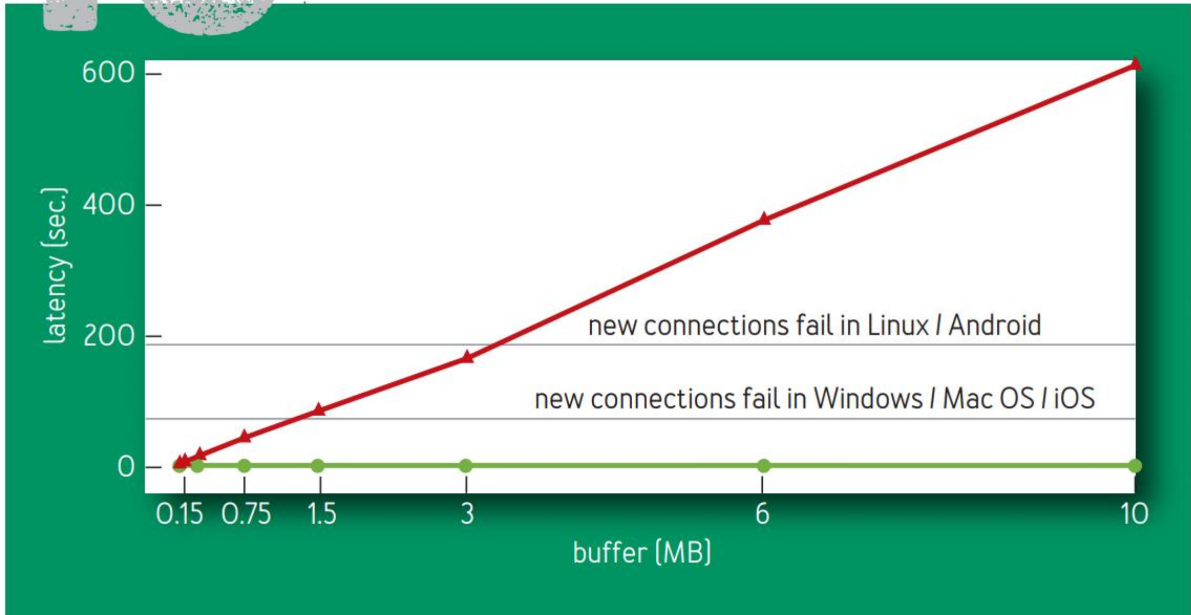
FIGURE 8: **BBR VS. CUBIC GOODPUT UNDER LOSS**



Interesting Case of SGSN

- SGSN(serving GPRS support node)
  - It is a standard PC so it have decent memory, which can maintain a large queue
  - The queueing time is so long, even longer than TCP SYN timeout.
  - If network congestion happen, the user can not even establish a single TCP connection with server.

FIGURE 10: **STEADY-STATE MEDIAN RTT VARIATION WITH LINK BUFFER SIZE**



Weakness

- CPU usage(slightly higher)
- Fairness working with other congestion algorithm

Thanks