Lecture #8: BBR

WPI CS4516 Spring 2019 D term

Instructor: Lorenzo De Carli (ldecarli@wpi.edu)

(slides include material from Christos Papadopoulos, CSU)

Evolution of TCP

TCP Flavors

- TCP Tahoe (distributed with 4.3BSD Unix)
  - Original implementation of van Jacobson's mechanisms (VJ paper)
  - Includes:
    - Slow start (exponential increase of initial window)
    - Congestion avoidance (additive increase of window)
    - Fast retransmit (3 duplicate acks)

TCP Reno

- 1990: includes:
  - All mechanisms in Tahoe
  - Addition of fast-recovery (opening up window after fast retransmit)
  - Delayed acks (to avoid silly window syndrome)
  - Header prediction (to improve performance)
    - Instead of checking if new packet is in window, check first if it is next expected segment
    - Less important nowadays
      - Especially since nw adapters can perform LRO

What is header prediction?

- (Van Jacobson ‘90) mechanism to optimize TCP receive processing on fast links
- On fast links, timestamps may be used to deal with wraparound
  - For every new packet, receiver must check if (1) packet is in window and (2) timestamp of packet is more recent of preceding packet in same window
- **Header prediction:** Instead of checking if new packet is in window, check first if it is next expected segment
  - Less important nowadays (especially since network adapters can perform LRO)

Fast recovery refresh

- In TCP Tahoe, loss causes *cwnd* to be set to 1 and slow start to be initiated
- In TCP Reno, losses detected via duplicate ACKs do not induce slow start – instead, the protocol enter into a **fast recovery** state
  - *ssthresh* still halved, but *cwnd* is also set to *ssthresh* (instead of 1)
  - Then, increase *cwnd* by 1 for each duplicate ack received (because we know that something arrived)

TCP New-Reno

- In Reno's fast recovery, multiple packet drops within window can cause window to deflate prematurely
- In New-Reno
  - Remember outstanding packets at start of fast recovery
  - If new ack is only a partial ACK, assume following segment was lost and resend, don't exit fast recovery
- Default in Windows up to Windows XP

TCP Sack

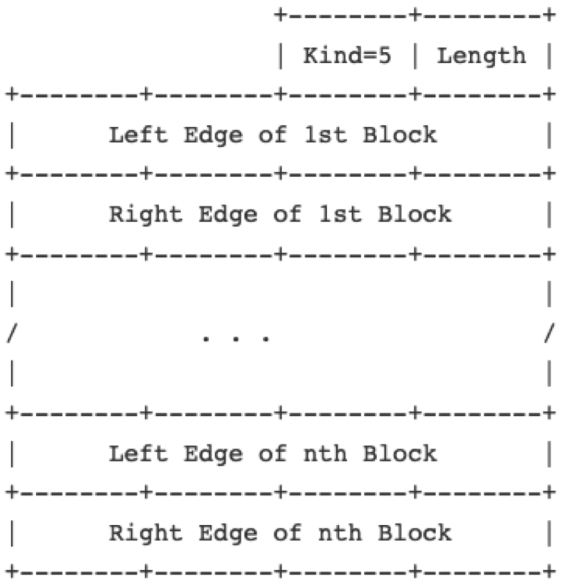
- New-Reno can only re-send *one dropped packet per RTT*
  - Because it can learn of multiple losses only once per RTT
- TCP SACK
  - Implements the SACK option in TCP
  - Can transmit more than one dropped packet because the sender *now knows which packet was dropped*
  - Sends dropped packets in preference to new data

What is the SACK option?

- Issue with basic duplicate ACK: it only signifies that the packet w/ the seq number after the acknowledged one was lost
- However, multiple packets from previous window may have been lost
- Receiver can:
  - Retransmit the next packet (will need to wait until next ACK to know if other packets were lost)
  - Retransmit all packets previously transmitted (may unnecessarily retransmit already received data)

What is the SACK option? / 2

- SACK is a TCP header option (i.e., uses an header extension)
- It allows to selectively acknowledge non-contiguous blocks of data:



(source: RFC 2018, October '96)

TCP Vegas (Brakmo et al. ‘94)

- First attempt to go beyond the “use-loss-as-sign-of-congestion” mechanism
- Idea: when connection is approaching congestion, pumping more data does not cause an equivalent increase in throughput
  - Expected rate: *WindowSize/RTT*
  - Actual rate: *<Bytes in segment> / <Segment RTT>*
- Congestion window linearly decreased when *Expected – Actual* > L_B, linearly increased when *Expected – Actual* < U_B

TCP Vegas (Brakmo et al. ‘94) /2

- Other improvements in TCP Vegas:
- More aggressive retransmission policy using more precise timeouts
- Modified slow-start:
  - In TCP Reno, slow start can overshoot the bandwidth significantly (congestion window can grow to twice the bandwidth = half window lost)
  - TCP Vegas double congestion window every **other** RTT
  - In the meanwhile, checks the difference between expected and actual rate – if large, switches to congestion avoidance

TCP BIC (Xu et al., 2004)

- Motivation: TCP congestion window additive increase is too slow for links w/ high bandwidth and RTT
  - E.g., fully utilizing a 10Gbps path w/ 100ms RTT takes one hour
  - Unattainable in practice: network cannot guarantee 1 hour of error-free transmissions at 10Gbps
- **Proposed solution:** adjust window growth rate based on network conditions

TCP BIC / 2

- Core idea: combine **binary search increase** with **additive increase**
- **Binary search:** Consider the state of a TCP state machine after a loss event:
  - W_{max}*:* window size before loss
  - W_{min}*:* window size after fast recovery
  - Perform binary search between them to find out window size (use midpoint as W_{max} if loss, W_{min} if not)
- But, if binary search initially requires sudden increase > threshold S_{max}, then use **additive increase** (by S_{max})
- Default in Linux until 2006

TCP CUBIC

- A modern variation of TCP BIC
- Based on the observation than BIC growth-rate may still be too aggressive in certain situations
- Replaces BIC growth rate function w/ a cubic function with similar behavior, but gentler growth around W_{max}: $W_{cubic} = C(T-K)^3 + W_{max}$

$$\text{C is a scaling constant, and K} = \sqrt[3]{\frac{W_{max}\beta}{C}}$$

Wmax is the window size just before the last window reduction, T is the elapsed time from the last window reduction, and beta is the multiplicative decrease factor after a packet loss event.

- Default in Linux after 2006

CUBIC growth rate

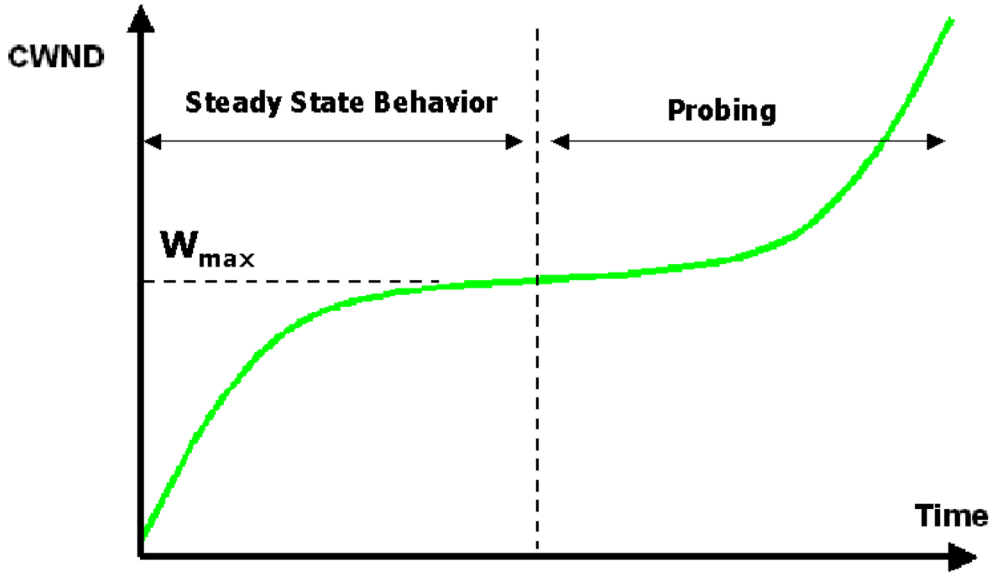


Fig.2: the cubic window growth function of CUBIC

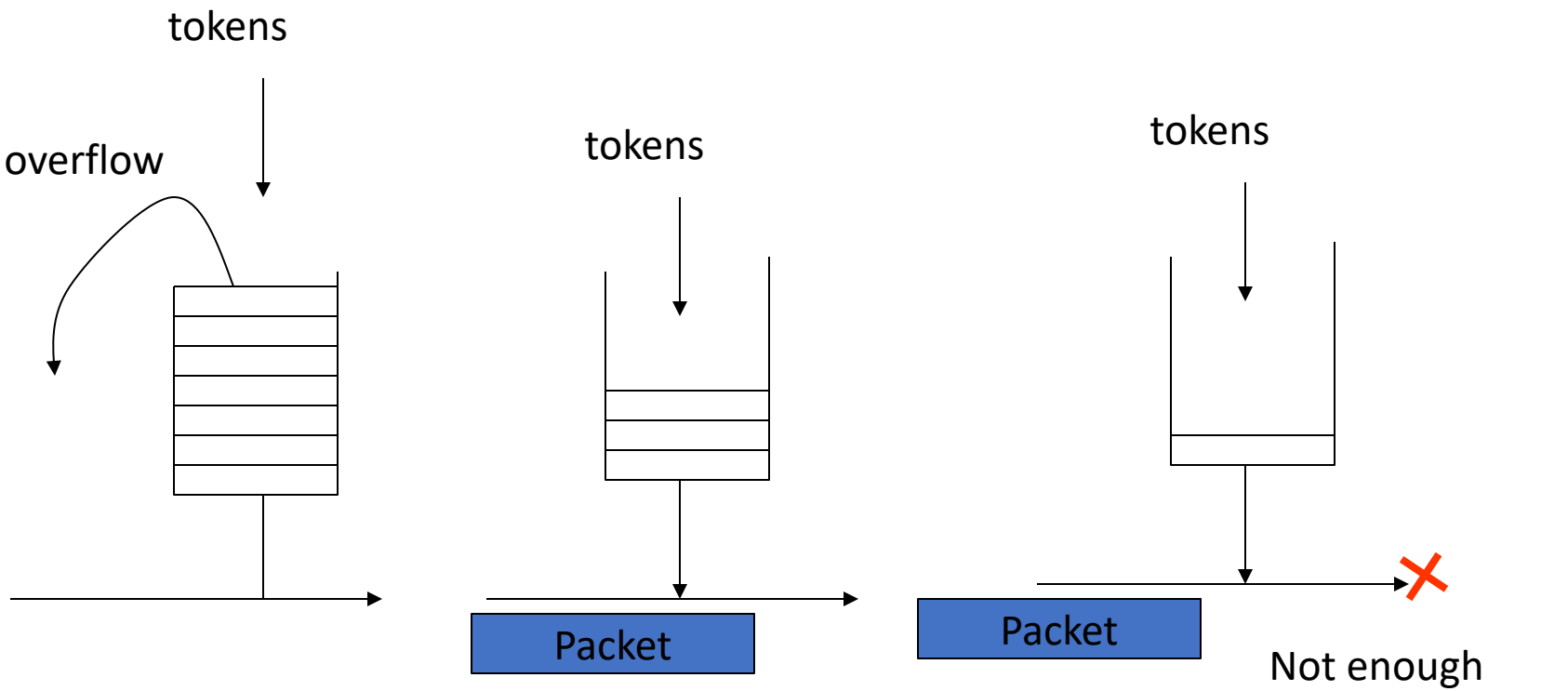
Source: *http://research.csc.ncsu.edu/netsrv/?q=content/bic-and-cubic*

Review of other concepts

Characterizing Traffic: Token Bucket Filter

- Parsimonious model to characterize traffic
- Described by **2 parameters**:
  - **token rate r:** rate of tokens placed in the bucket
  - **bucket depth B:** capacity of the bucket
- Operation:
  - tokens are **placed in bucket** at rate r
  - if bucket fills, **tokens are discarded**
  - sending a packet of size P **uses P tokens**
  - if bucket has P tokens, **packet sent at max rate, else must wait for tokens to accumulate**

Token Bucket Operation



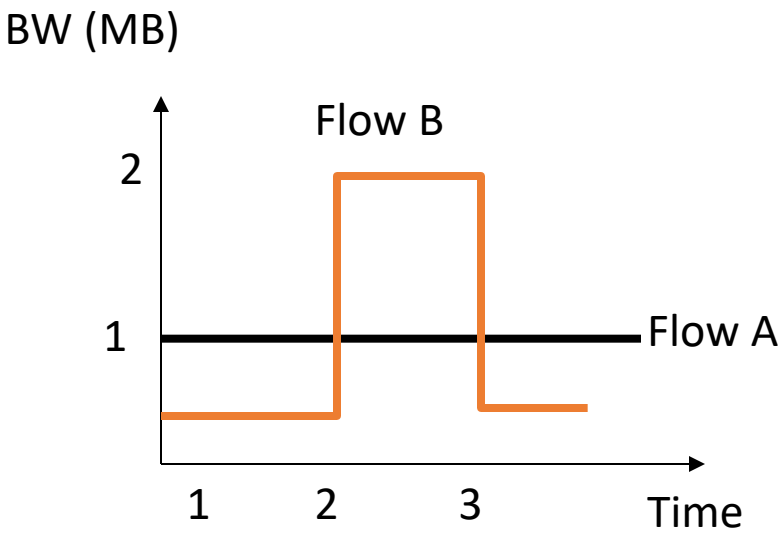
Enough tokens packet goes through, tokens removed

Not enough tokens - wait for tokens to accumulate

Token Bucket Characteristics

- In the long run, **rate is limited to r**
- In the short run, **a burst of size B can be sent**
- Amount of traffic entering at interval *T* is bounded by:
  - *traffic = B + r*T*
- Information useful to admission algorithm

Token Bucket Specs



Flow A: r = 1 MBps, B=1 packet

Flow B: r = 1 MBps, B=1MB

Possible Token Bucket Uses

- **Shaping, policing, marking**
  - **delay pkts** from entering net (shaping)
  - **drop pkts** that arrive without tokens (policing)
  - **let all pkts pass through, mark ones without tokens**
    - Then, network drops pkts without tokens during congestion

Shallow vs deep buffers

- Different network equipment design philosophies:
  - **Shallow-buffer:** small packet queues (can hold limited number of packets waiting to be transmitted)
    - Cheaper
    - Keeps max queuing latency low
    - Easier to incur packet losses
  - **Deep-buffer:** large packet queues
    - More expensive
    - Can introduce significant latency
    - Less likely to cause losses

Bufferbloat

- If:
  - network equipment is deep-buffered, and
  - the network is somewhat congested
- Then:
  - Even if congestion does not get to the point where losses occur, the **average queue utilization is high**, which means that **packets wait for a long time to be transmitted**
- Networking researchers call this situation “bufferbloat”

RTT

- Round Trip Time
- Time interval between sending a packet P and receiving the corresponding ack
- Includes **propagation delay (constant)** and **queuing delay (variable)**:

$$RTT_t = RTprop_t + \eta_t$$

Bandwidth x delay product (BDP)

- Simply a measure of **how much data a sender can push into the network** before the network **"saturates"**
- E.g., if the network path exhibits bandwith of *10Kb/s*, and the end-to-end delay (= 0.5RTT) is *10s*, then up to to *10Kb/s * 10s = 100Kb* can be in transit
- Useful parameter to **understand path characteristics, buffering & congestion**
- If I push more than BDP Kbs, then the excess data must get buffered somewhere (or dropped)

Goodput

- “Useful” bandwidth as seen by the application sending data
- Equivalent to the bandwidth seen by the transport protocols minus control data (header), retransmissions

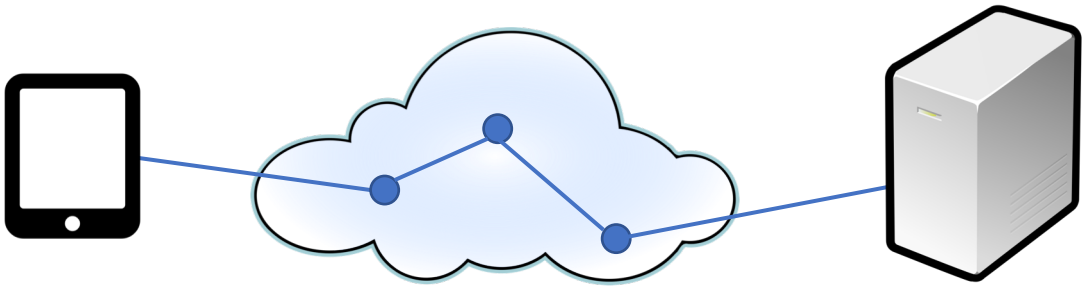
Why do people want to change TCP?

TCP limitations

- TCP was designed in an era of relatively **low bandwidths and small buffers**
- Protocol design assumes routers have enough buffering to deal with temporary bursts, but not much more

TCP limitations/2

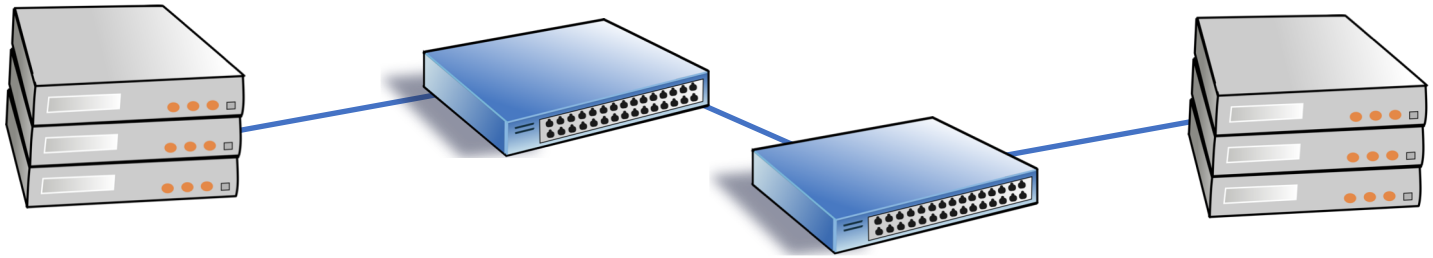
- Problem #1: **aggressive increase of tx window**
  - Scenario: communication over Internet path



  - TCP assumes: No losses -> we can have more data in flight
  - However, memory is less expensive than it used to be…
  - …and Internet routers are designed **with large buffer** to minimize losses…
  - So TCP may be “pumping” more packets thinking that it is not yet saturating the path, while in fact it is just **increasing queue sizes**

TCP limitations/3

- Problem #2: **losses as symptom of congestion**
  - Scenario: data center network



  - Low latency important
    - Smaller buffer help
  - With small buffers, some amount of losses w/o congestion is to be expected
  - **Throttling on loss unnecessary!**

Can we do better?

BBR

- A new logic for adjusting window size in transport protocols
  - **Not a new protocol!**
  - Packet format, header fields etc. as defined in TCP
- BBR: ”**B**ottleneck **B**andwidth & **R**ound Trip Time”

BBR ideas in a nutshell

- **Ignore losses** (i.e., retransmit lost packets, but do not interpret loss as strong signal of congestion)
  - Rationale: in modern networks, loss does not imply presence of congestion
- Instead, **estimate path characteristics**…
  - …and use them to define packet sending rate
- Goals:
  - **High BW:** Send packets as close as possible to the maximum rate the path can deliver...
  - **Low latency:** …while keeping buffer utilization low

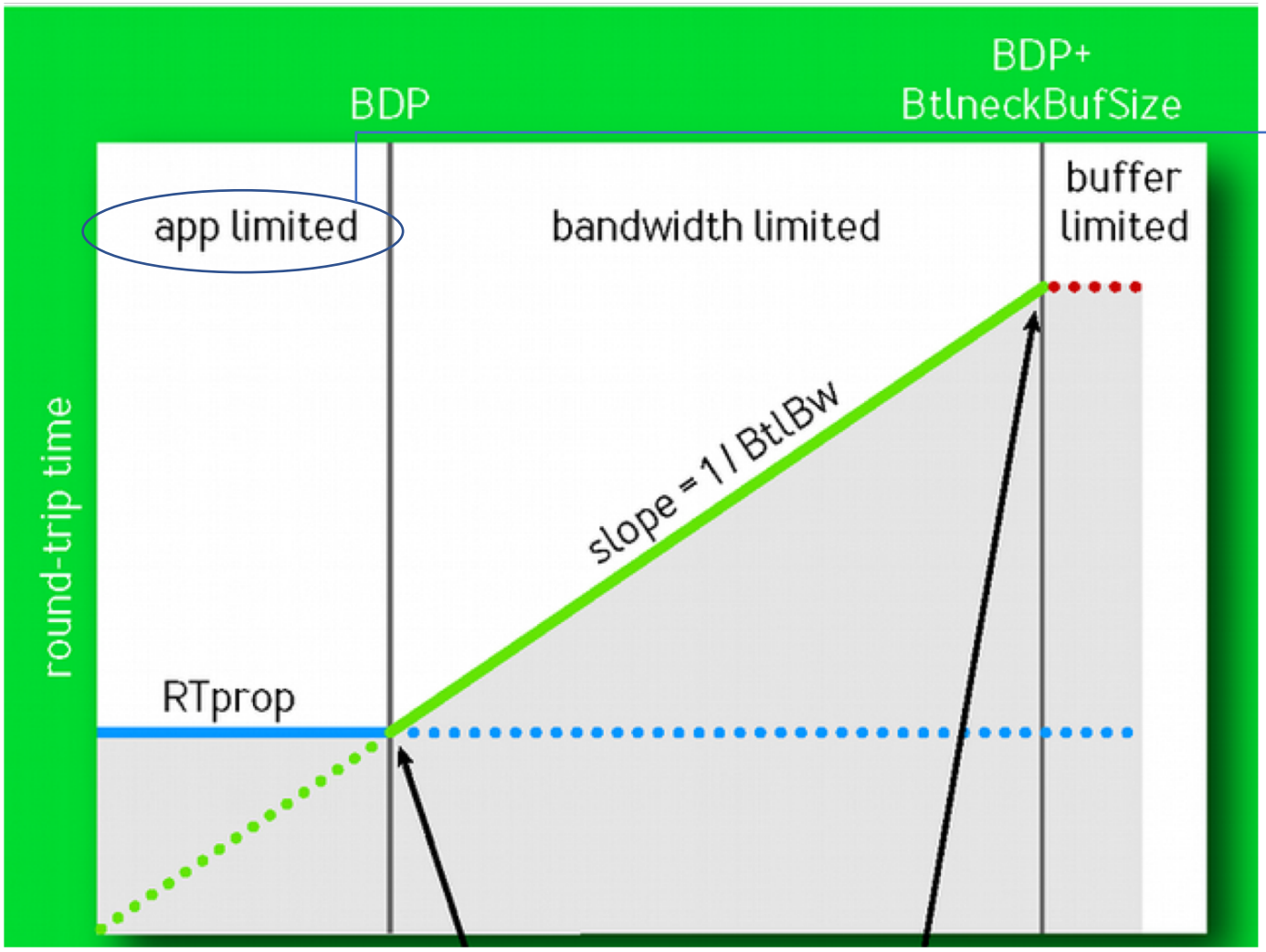
BBR path model

- Any model must simplify the actual network in order to be manageable
- BBR abstracts a network path, however complex, as:
  - **Bandwidth of bottleneck link** *BtlBw*
  - **Propagation delay along the path** *RTprop*
    - Physical propagation +
    - Processing time in routers assuming no queue

Why does this model make sense?

- BBR goal is to achieve *optimal use* of network resources along the path…
- …by which it is meant to:
  - **Saturate the network path** (i.e., send data at a rate as close as possible to *BtlBw*
  - …**but keep RTT as close as possible to *RTTprop*** (i.e., keep router queues as close empty as possible

X axis: amount of data in flight, i.e. tx window size



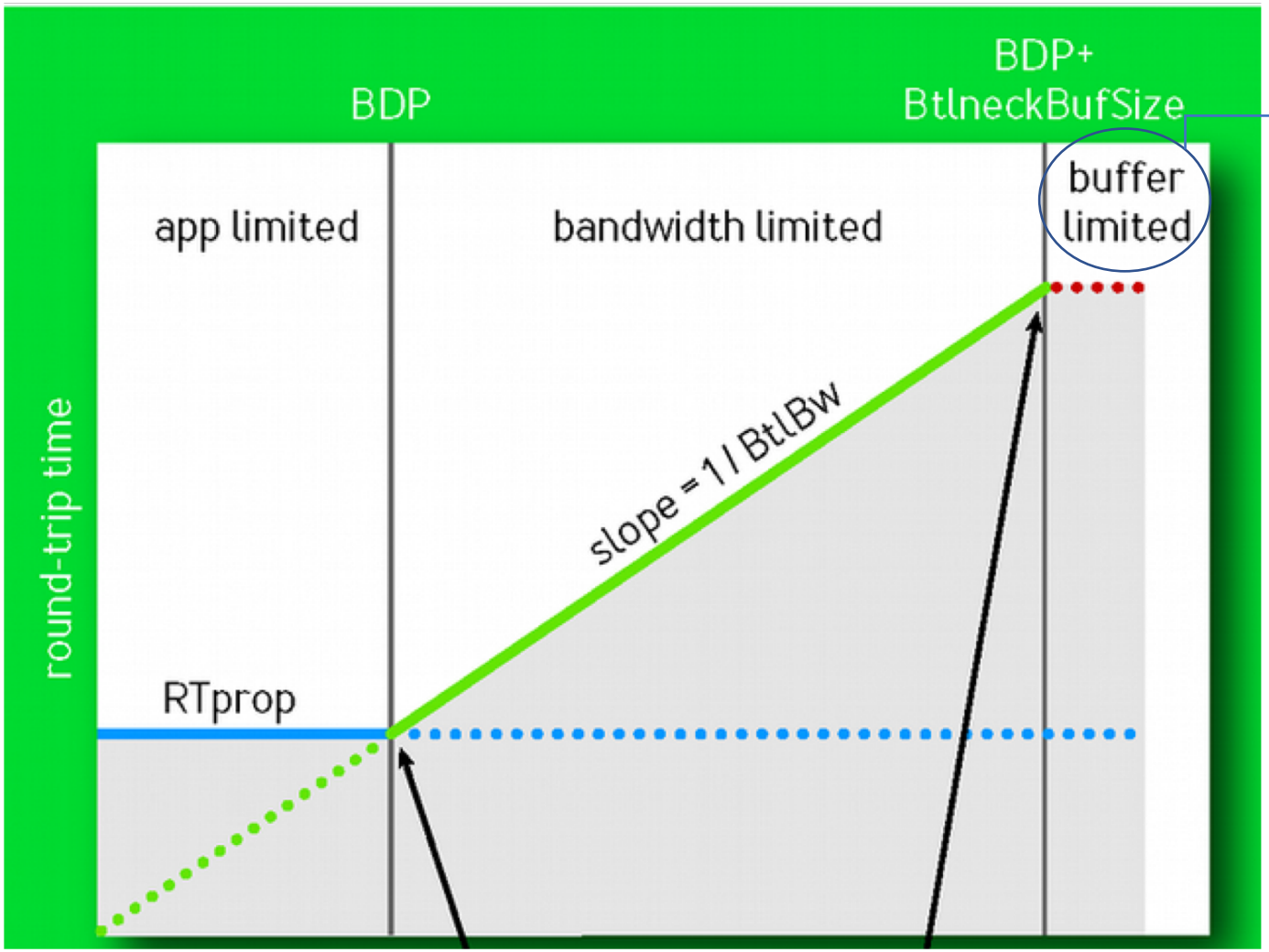
What happens in this range?

- Tx sends less bytes than the pipe capacity
- No queuing occur
- RTT == Rtprop!

Optimum operating point

Loss-based congestion control operates here

X axis: amount of data in flight, i.e. tx window size



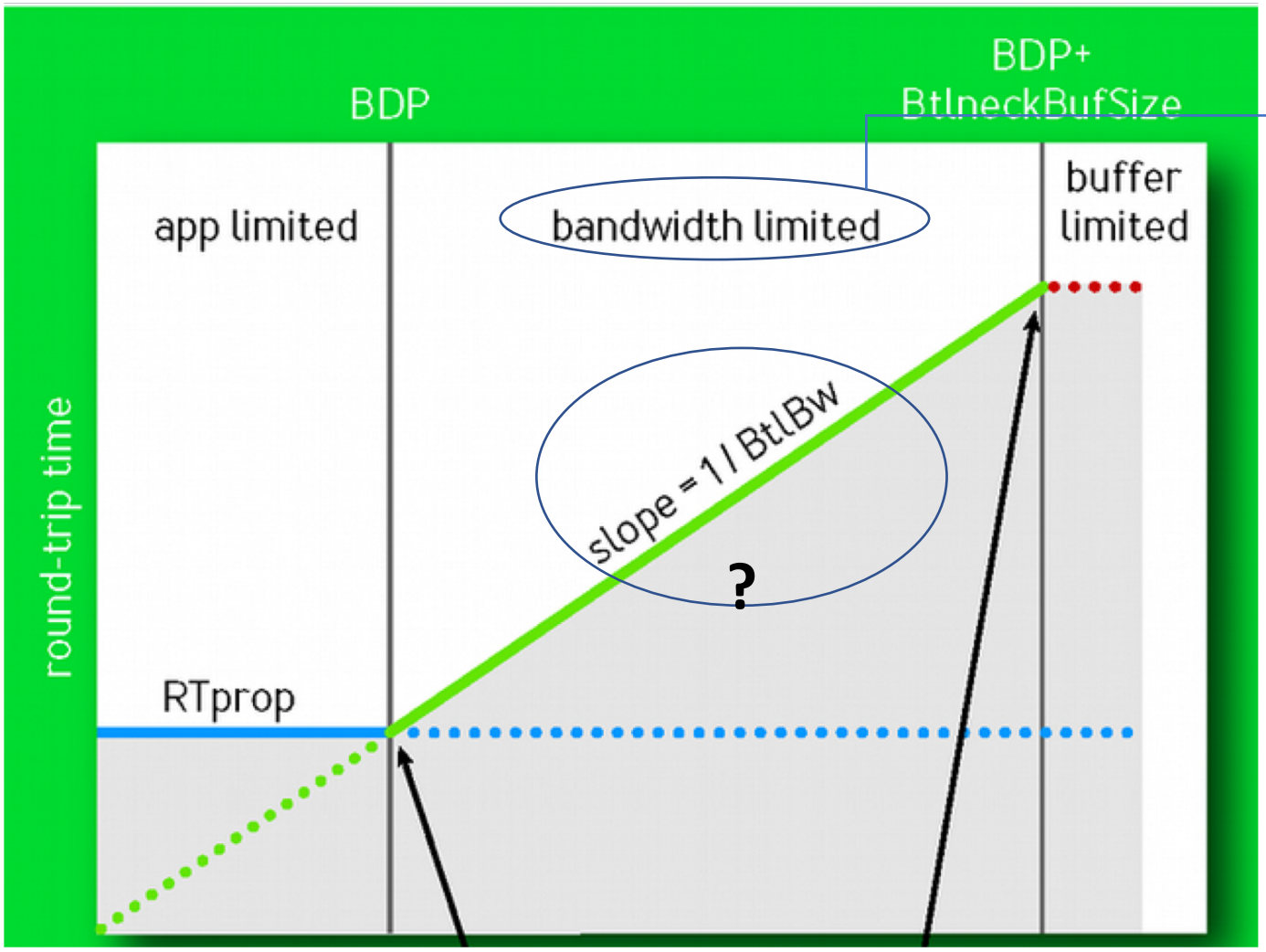
Optimum operating point

Loss-based congestion control operates here

What happens in this range?

- Tx sends more data in one round than the bottleneck router queue can hold
- So, for every packet that does not get dropped queuing delay is constant (and maximum)

X axis: amount of data in flight, i.e. tx window size

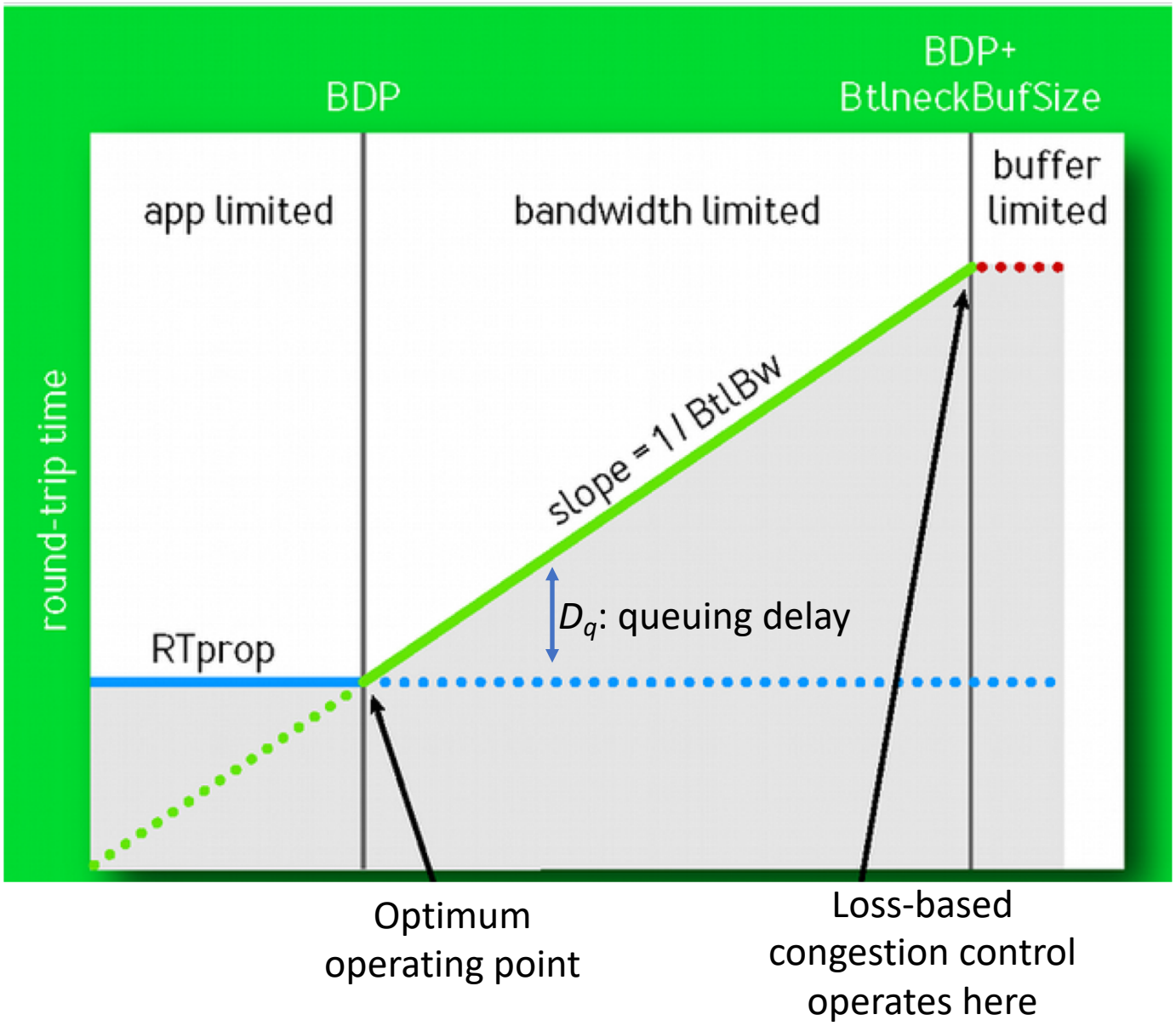


Optimum operating point

Loss-based congestion control operates here

What happens in this range?

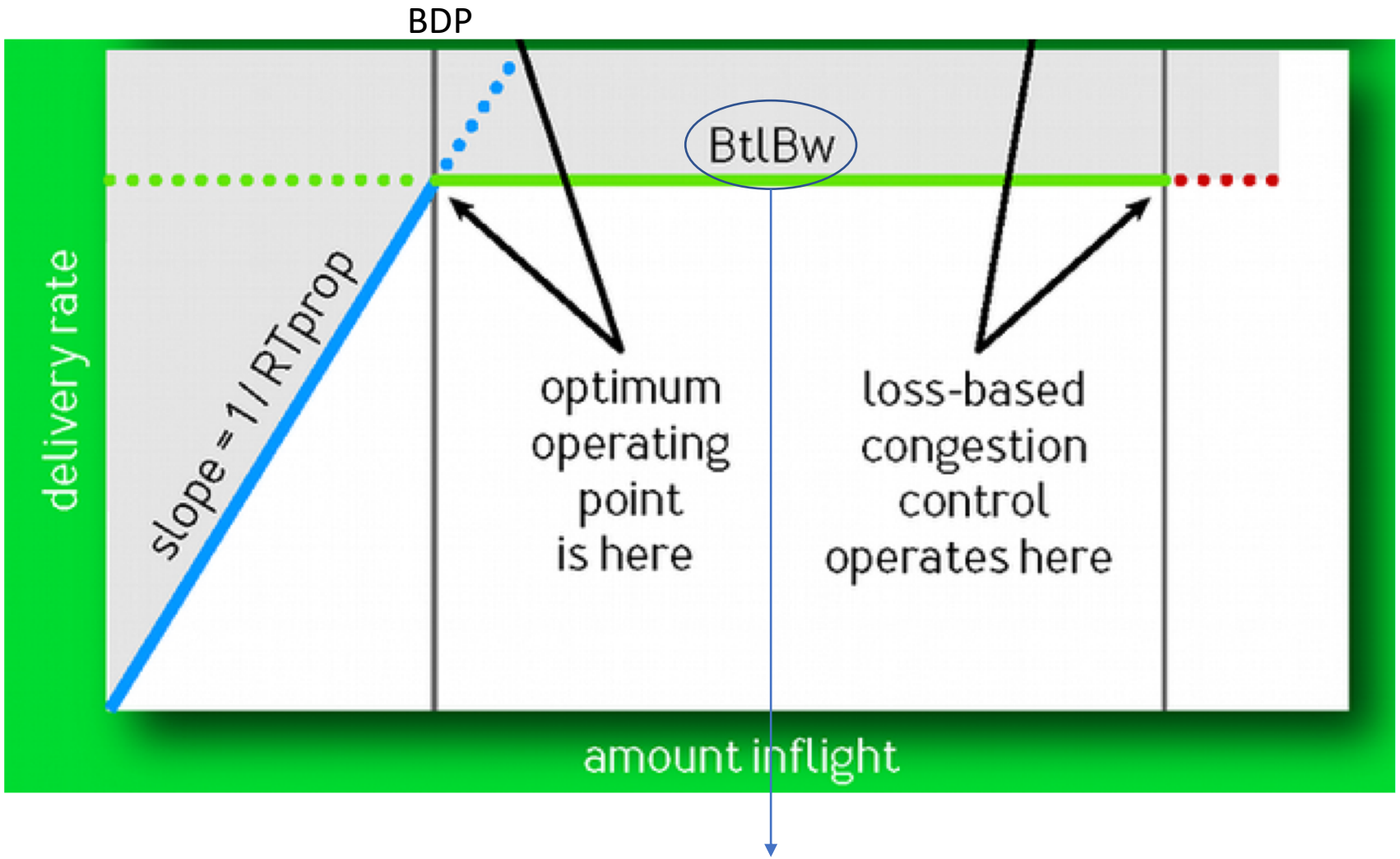
- Tx sends more data than the path can hold…
- But not enough to fill the bottleneck buffer
- The more data, the higher the queuing delay!



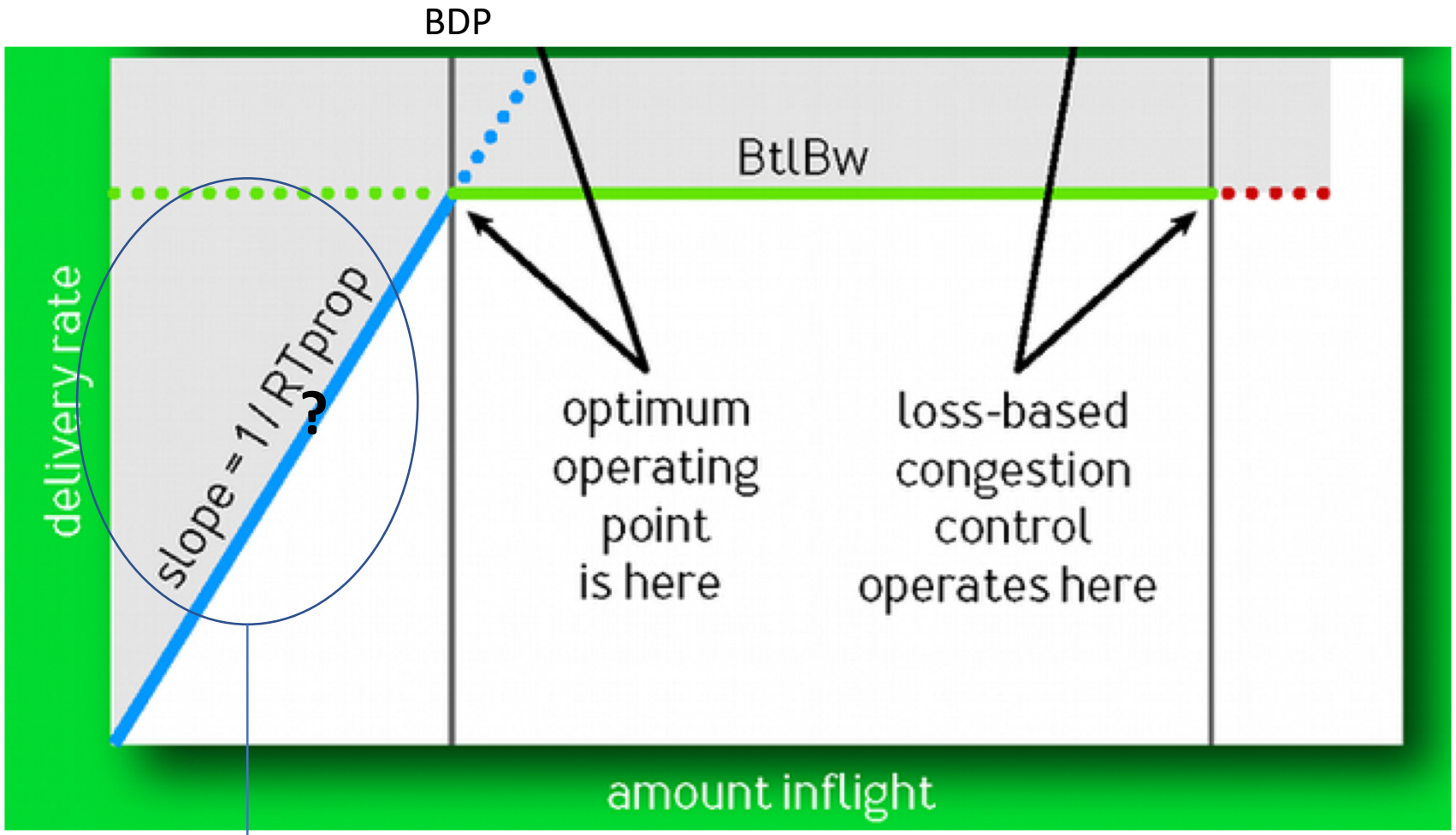
RTT= RTprop + D_q

- D_q depends on the amount of time data in excess of BDP (*E*) stay in the queue...
- ...which in turn depends on the bandwidth of the bottleneck link *BtlBw*...
- Therefore:

$D_q = E / BtlBw$

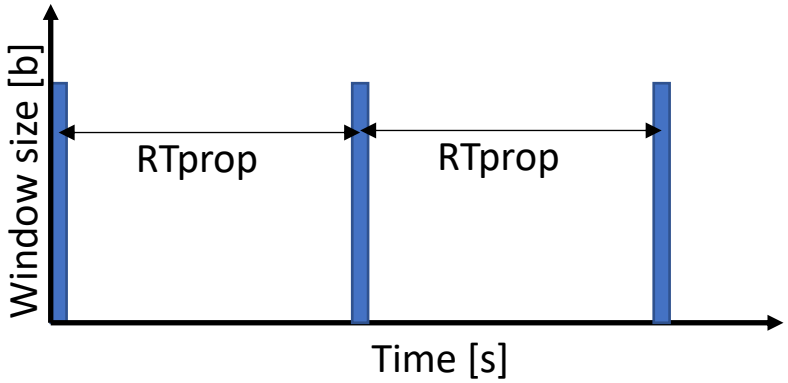


If I send more data than the pipe can take, then obviously the delivery rate is capped at the BW of the slowest link



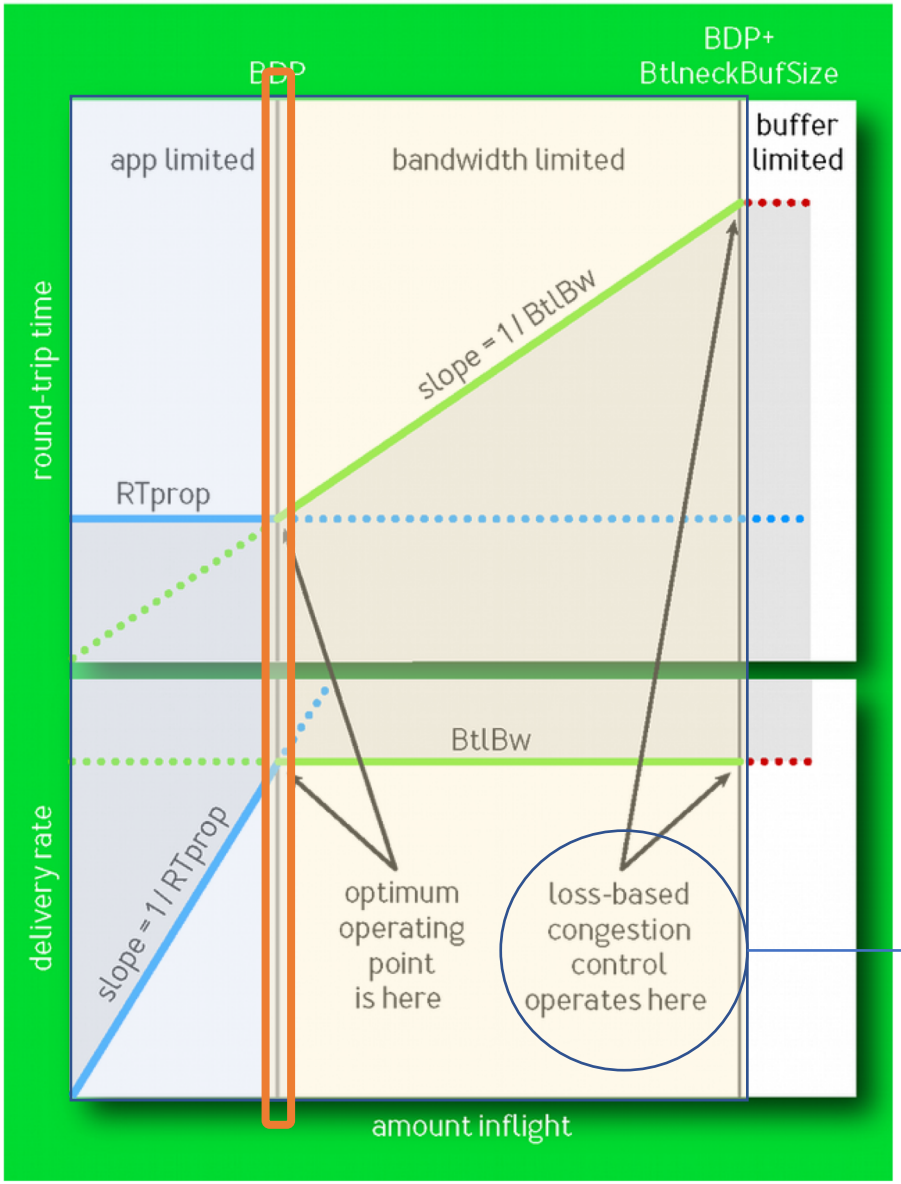
If I send less data than the pipe can take, then my delivery rate is determined by how long I have to wait for ACKs (which in turn depends on Rtprop)

To make things simple, imagine transmission as a sort of “duty cycle” when the tx sends a window worth of data (*W*) and then waits for ACKs until transmitting the next window:



- In each cycle, tx sends *W* bytes and waits for *RTprop* s
- Therefore, the average BW (delivery rate) is *W/RTprop*

Putting it all together…



1. I want to be in this region, since it minimizes latency…
2. … but I also want to be in this region, since it maximizes bandwidth
3. Goal: tune sending rate to keep delay as close to *RTprop* as possible while keeping delivery rate as close to *BtlBw* as possible

Can you explain me why this is the case?

How to estimate RTprop?

- **Observation:** *RTprop* is a lower bound for RTT
- So just measure RTT on ACK reception and take the minimum of the most recent observations:

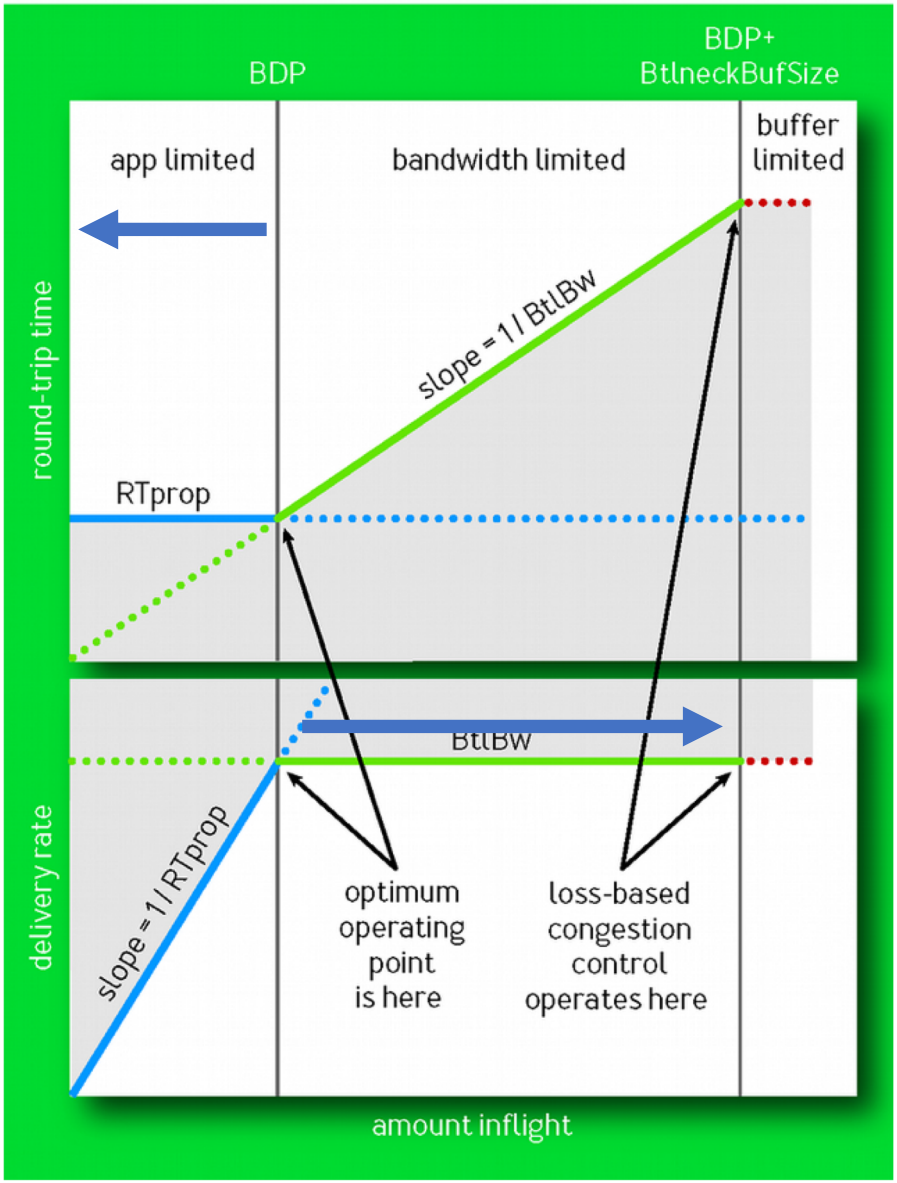
$$\widehat{RTprop} = RTprop + \min(\eta_t) = \min(RTT_t) \quad \forall t \in [T - W_R, T]$$

How to estimate BtlBw?

- **Observation:** *BtlBw* is an upper bound for the delivery rate of the connection
- So measure the rate at which data is delivered and compute the max of recent observations:

$$\widehat{BtlBw} = \max(deliveryRate_t) \quad \forall t \in [T - W_B, T]$$

Some trickery



- I can only estimate RTprop in this region…
- … but I can only estimate BtlBw here…
- So I cannot estimate them at the same time!

Yes, but how does it work in practice?

Estimate path parameters on ACK reception…

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

Estimate RTprop

Estimate BtlBw

…pace outgoing data on send

```
function send(packet)
  bdp = BtlBwFilter.currentMax × RTpropFilter.currentMin
  if (inflight >= cwnd_gain × bdp)
     // wait for ack or retransmission 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)
```

If path is already saturated, wait

Otherwise, send

Pace next packet according to estimate bottleneck BW

High-level control loop

- BBR algorithm ensures that **no more than 1 BDP is in flight,**
- And that **data is sent at a rate as close as possible to *BtlBw***
- Result: **queuing** along the path (and therefore delay) **is minimized**, while **path bandwidth is fully exploited**

Problem #1: detecting additional bandwidth

- The control loop discussed so far is stable but it is **not able to detect additional BW that may become available**
- Can you explain me how BBR solves this?
- **Solution:** periodically and temporarily bump up sending rate

BBR BW discovery algorithm

- Periodically **send at rate higher than estimated BtlBw**, then **look at what happens to BtlBw and RTT**:
  - If BtlBw estimate **increases**, we are good
  - If BtlBw **stays constant but RTT increases**, then we are filling router queues but not sending data faster, so BtlBw has not increased
  - One last trick: if I just send data faster and go back to normal, **I may create a queue that will not dissipate**…
  - …so after probing BBR spends one Rtprop **sending data at lower rate**
  - Probing cycle gains: 5/4, 3/4, 1, 1, 1, 1, 1, 1

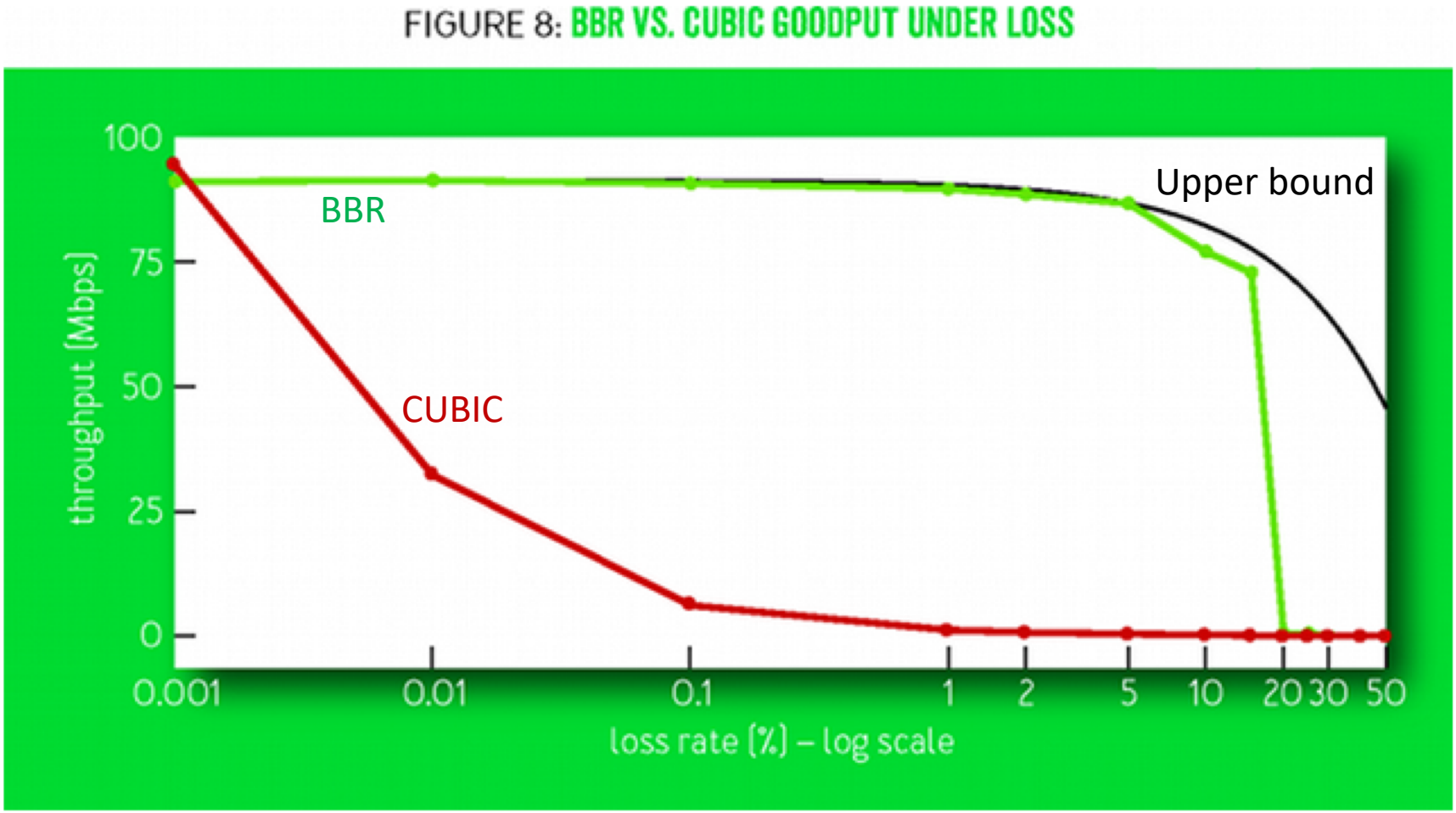
Problem #2: bootstrapping connection

- Same problem as in TCP…
- …do you remember how TCP solves it?
- BBR **doubles sending rate until BtlBw stops increasing**, then back off to drain queue
- Note how **there is no place in BBR control loops where decisions are made based on packet losses**

Other notes

- BBR flows ”play nice” with each other
- BBR flows “play nice” with conventional loss-based congestion control
  - Or, do they?
- BBR **does not interact well w/ token bucket**: token bucket is transparent until the token is emptied, but then forces a BW < BtlBw.
  - **Longer startup** – BBR is fooled into believing that there is BtlBw available bandwidth, then throttled
  - Losses **every time BBR runs bandwidth discovery**
  - **Needs explicit model of token bucket policer** in the protocol state machine

Some results



Some results/2

Figure 10 shows steady-state median RTT variation with link buffer size on a 128-Kbps/40-ms link with eight BBR (green) or CUBIC (red) flows. BBR keeps the queue near its minimum, independent of both bottleneck buffer size and number of active flows. CUBIC flows always fill the buffer, so the delay grows linearly with buffer size.

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