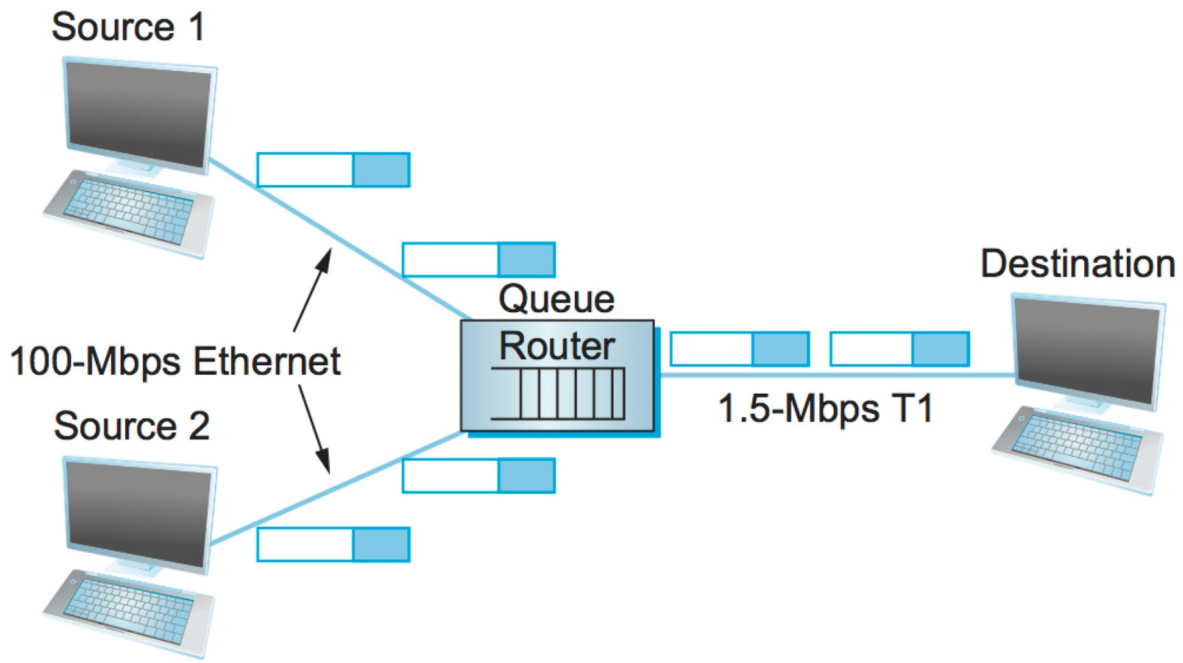
CSC 425:
Computer Networking

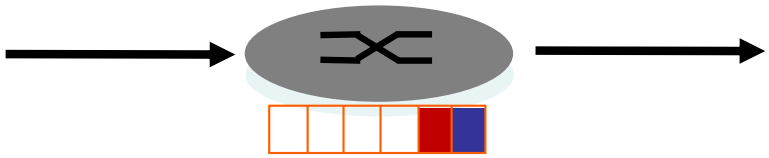
Network Congestion

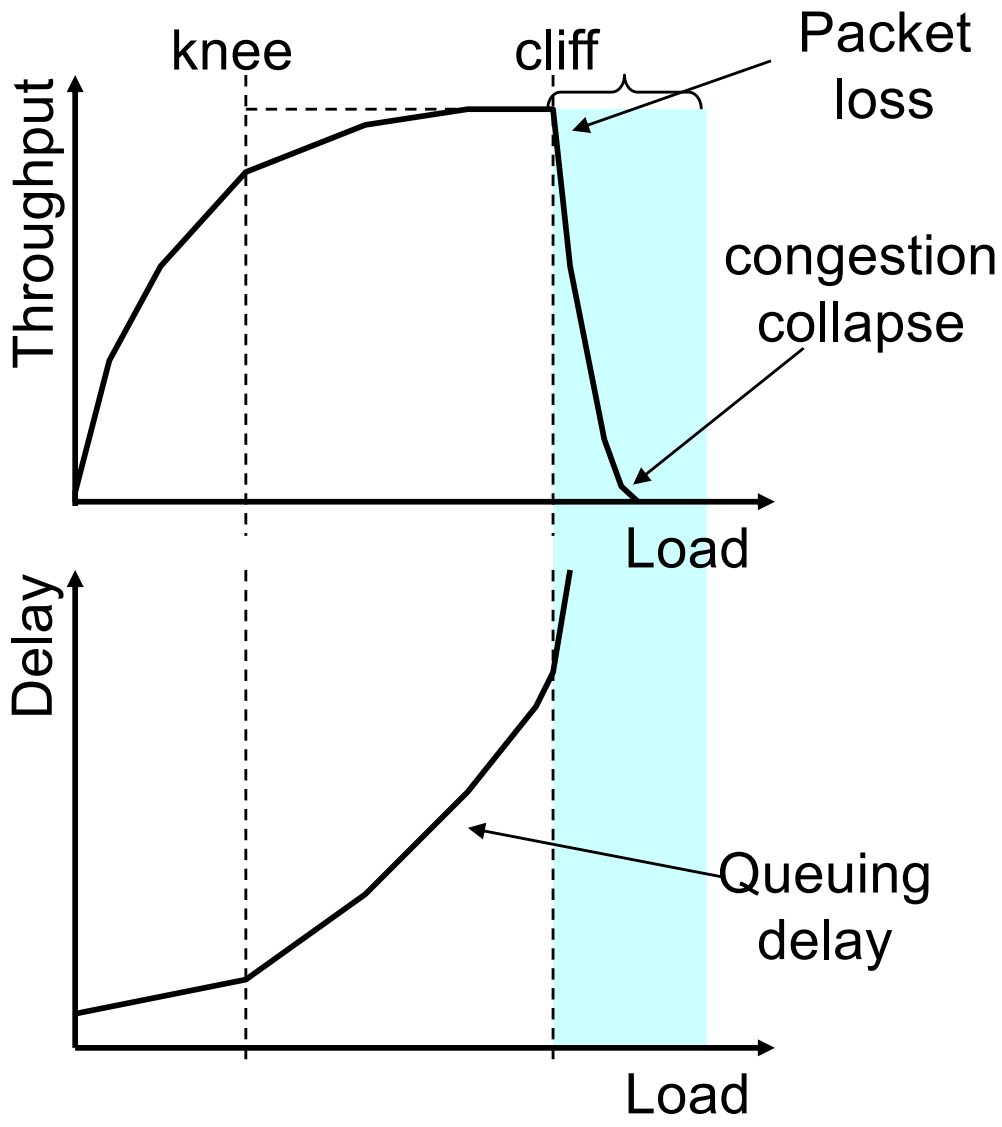


- How to manage sending rate so that the traffic will not overrun the router/link?

The Danger of Congestion

- Many sources sending data too fast for network to handle
- Knee – after which
  - Throughput increases slowly
  - Delay increases fast
  - Link is full.
- Cliff – after which
  - Throughput starts to decrease very fast (congestion collapse)
  - Delay approaches infinity
  - Buffer is full.

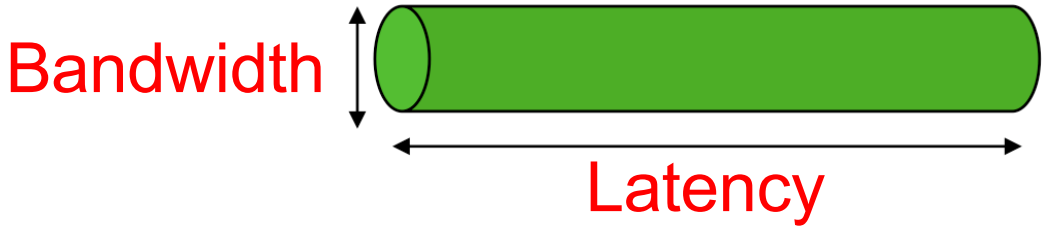




Congestion Control

- The throughput is limited by the **available bandwidth at the bottleneck link** along the path.
  - Only the bottleneck link matters
  - Available bandwidth varies due to cross traffic
- Requirements of congestion control solutions
  - Fully utilize the available bandwidth, i.e., high **throughput**
  - Minimize the queue in the router, i.e., short **delay**
  - Share with others **fairly**.
- Solution space
  - Source adaptation, e.g., TCP (only involves end hosts)
  - Router enforced, e.g., RED
  - Cooperation between sources and routers, e.g., ECN.

Keep the Pipe Full

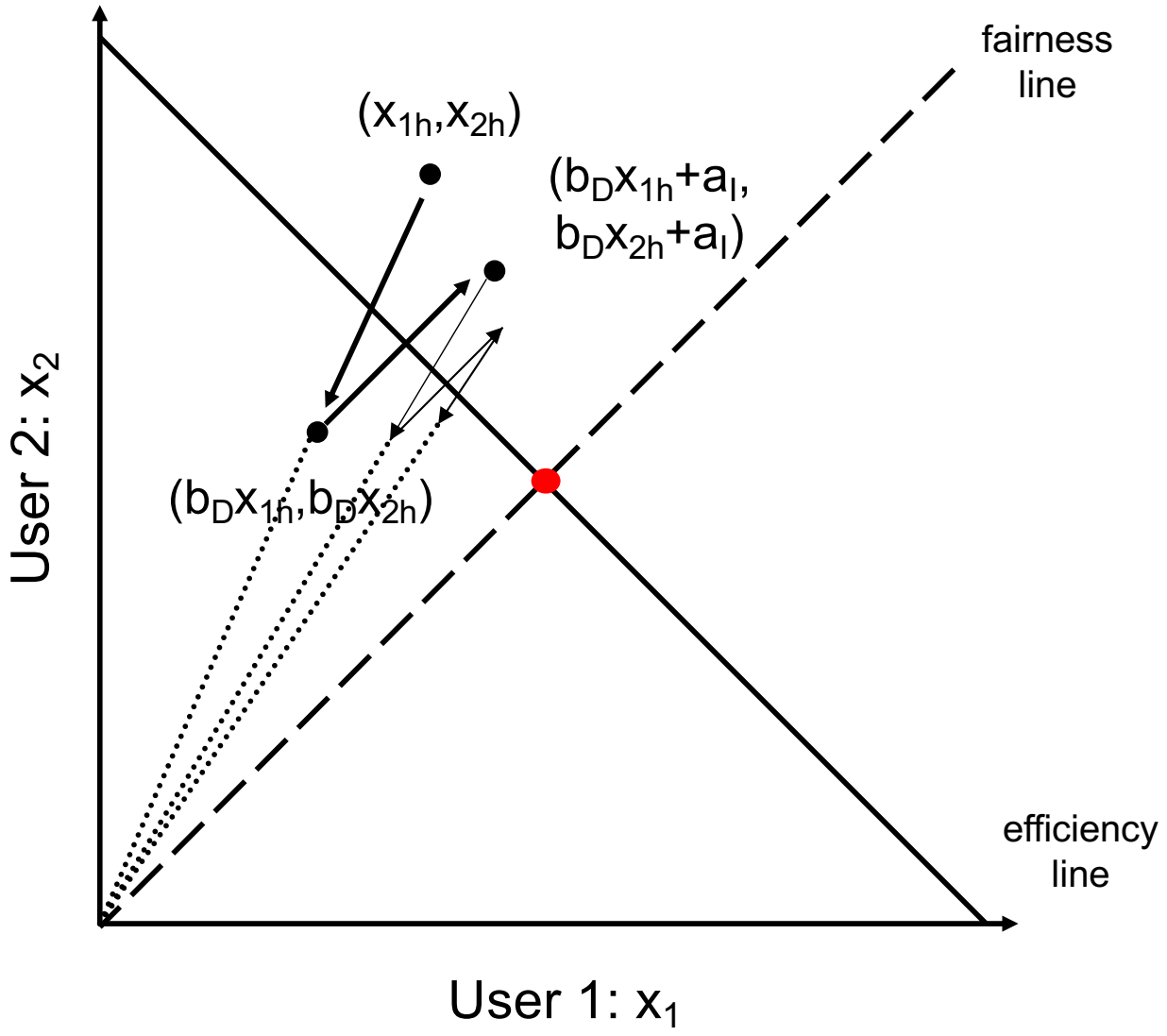


- You can abstract the entire path as a single pipe
  - available bandwidth at the bottleneck
  - round-trip time (latency)
  - The bandwidth delay product BW*RTT is the capacity of the pipe
  - The sender can send at most BW*RTT of data before an ACK.
  - ideal window size = BW*RTT/ segment size
  - More than this will cause congestion, less than this is under utilization.

- But both BW and RTT vary over time due to cross traffic.

Fair Play

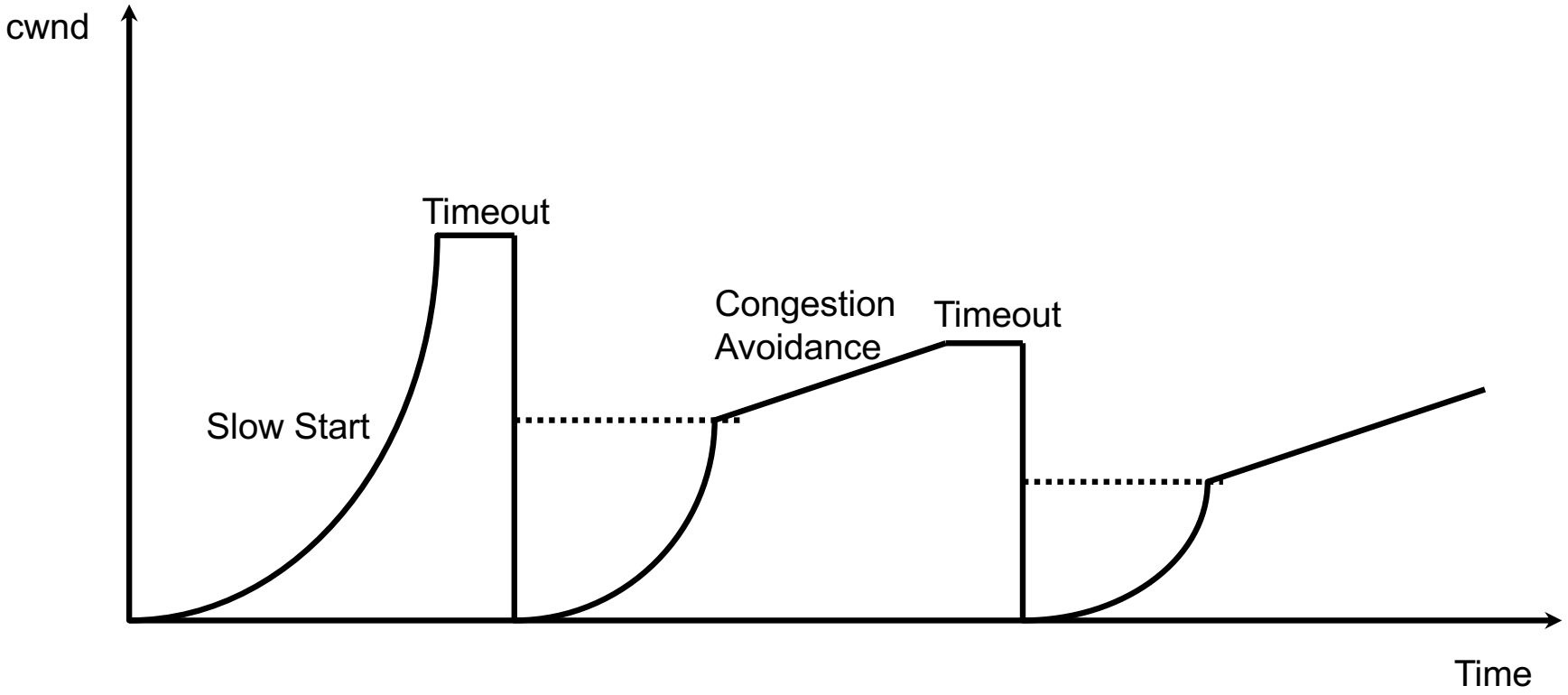
- Two TCP flows share a single bottleneck link.
- In an ideal design, they should get equal share of the bottleneck bandwidth, and total traffic is within the bottleneck capacity, i.e., the red dot.



TCP congestion control

- Sender sends data packets, receiver replies with ACKs
  - Measure RTT (to be used in retransmission timeout)
  - Sender adjust window size or sending rate to match available bottleneck bandwidth
- Many design variants that differ in
  - How to detect or predict congestion
  - How to adapt window size or sending rate

TCP Tahoe



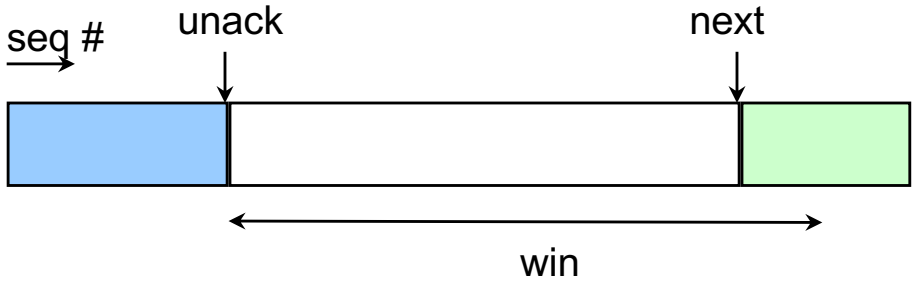
- The original design that fixed congestion collapse in early Internet
- Slow start: initially increase window size exponentially
- Once reached the threshold, change pace to linear increase
- Use timeout (packet loss) as congestion signal.
- Reset window size and threshold to start over.

Putting Everything Together

```
Initially:
    cwnd = 1;
    ssthresh = recvWin;
Every new ack received:
    if (cwnd < ssthresh)
        /* Slow Start*/
        cwnd = cwnd + 1;
    else
        /* Additive Increase*/
        cwnd = cwnd + 1/cwnd;
Timeout:
    /* Multiplicative decrease */
    ssthresh = cwnd/2;
    cwnd = 1;
```

```
while (next < unack + win)
    transmit next packet;

where   win = min(cwnd,recvWin);
```



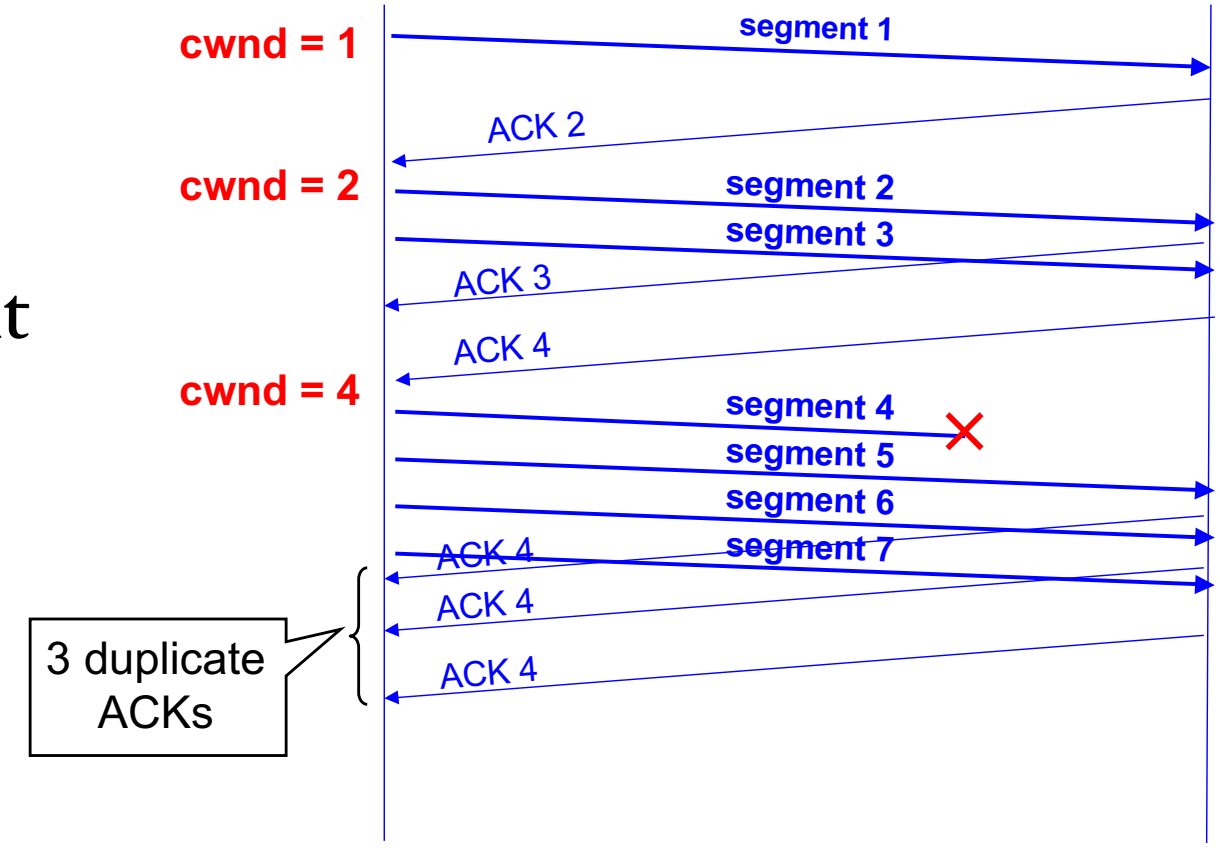
The problem with TCP Tahoe

- It leaves lots of room to improve link utilization.

- It has to cause packet loss in order to detect congestion.

Fast Retransmission

- Don't wait for window to drain
- Resend a segment after 3 duplicate ACKs



Fast Recovery

- After a fast retransmission, set *cwnd* to *cwnd/2*
  - i.e., don't reset *cwnd* to 1
- But when RTO happens still do *cwnd* = 1
  - The network condition is so bad that (duplicate) ACKs didn't come through.
- Fast Retransmit and Fast Recovery
  - Implemented in TCP Reno
- Lesson: avoid RTO

Fast Retx and Fast Recovery

Initially:

```
    cwnd = 1;
    ssthresh = recvWin;
```

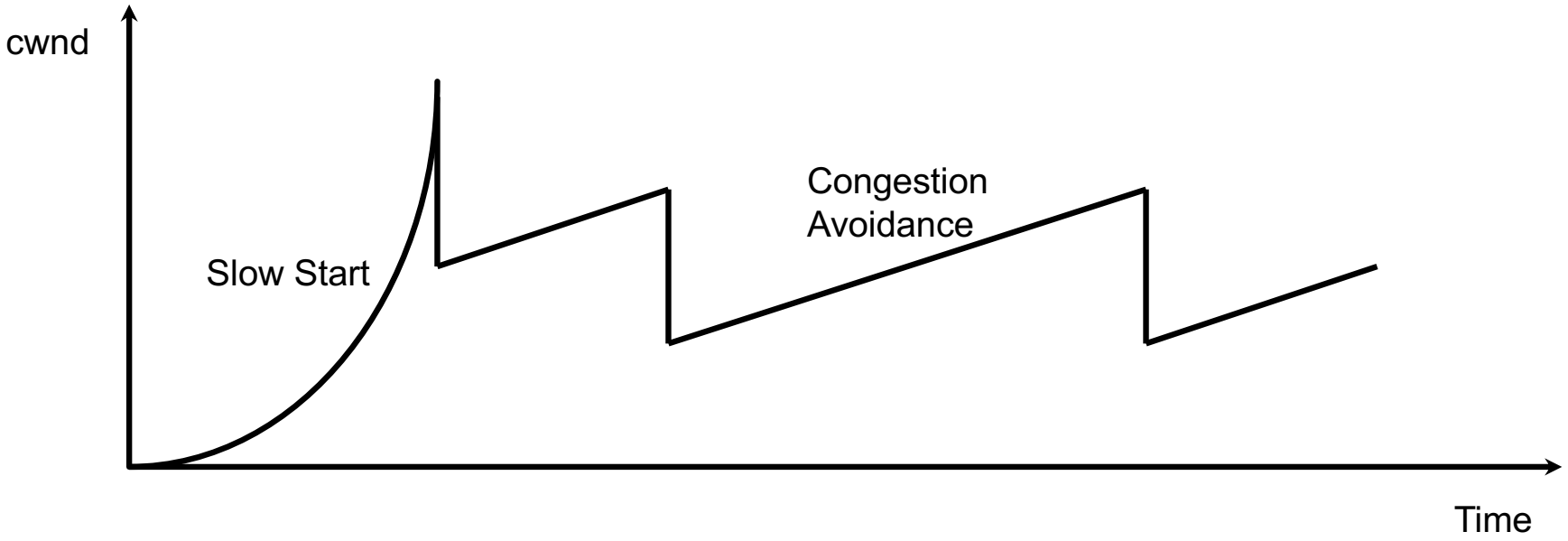
Every new ack received:

```
    if (cwnd < ssthresh)          /* slow start*/
        cwnd = cwnd + 1;
    else                          /* additive increase*/
        cwnd = cwnd + 1/cwnd;
```

Loss detected:

```
    ssthresh = cwnd/2;            /* multiplicative decrease */
    if (3 duplicated ACKs)        /* fast retransmission */
        cwnd = ssthresh;          /* fast recovery*/
    else                          /* fall back to the beginning */
        cwnd = 1;
```

Fast Retransmit and Fast Recovery



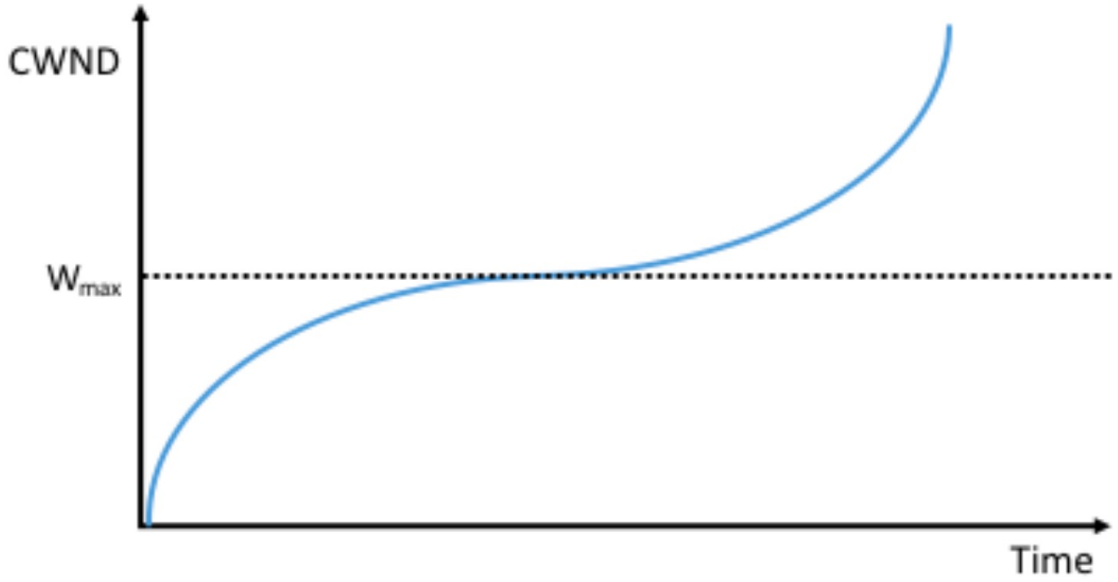
- Retransmit after 3 duplicated acks
  - prevent expensive timeouts
- No need to slow start again
- At steady state, *cwnd* oscillates around the optimal window size.
- This is commonly known as TCP Reno.

Problems

- Traditional TCP doesn't perform well in case of very large bandwidth or very long delay.
  - Oscillatory and tend to be unstable
  - Cannot grab (big) spare bandwidth quickly due to additive increase or long RTT
    - Flows with short duration don't get the benefit of high bandwidth
    - Flows with long delay respond/adapt too slow.

TCP CUBIC

- Default congestion control mechanism in Linux
- To support links with large delay x bandwidth product.
- Increase window size faster when it is far away from previous congestion point (Wmax), but slows down significantly when it is getting close to it.



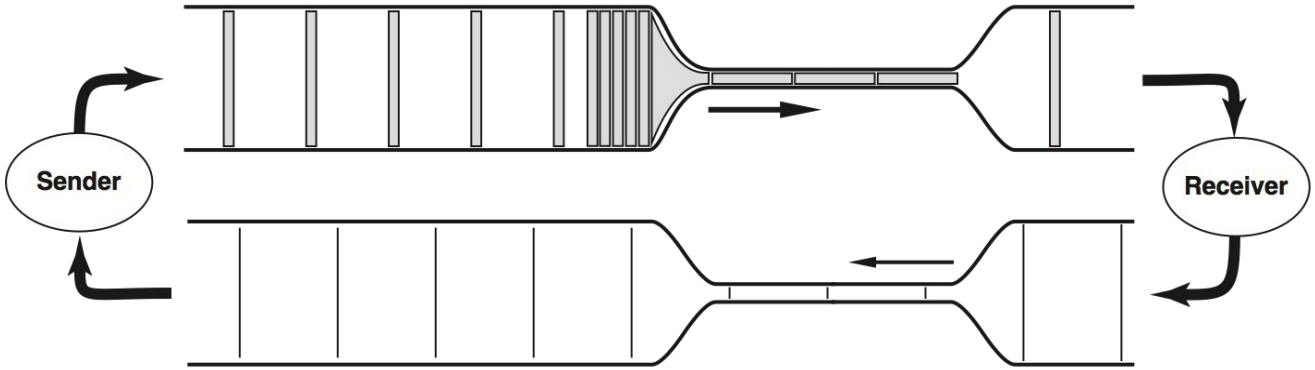
The buffer bloat problem

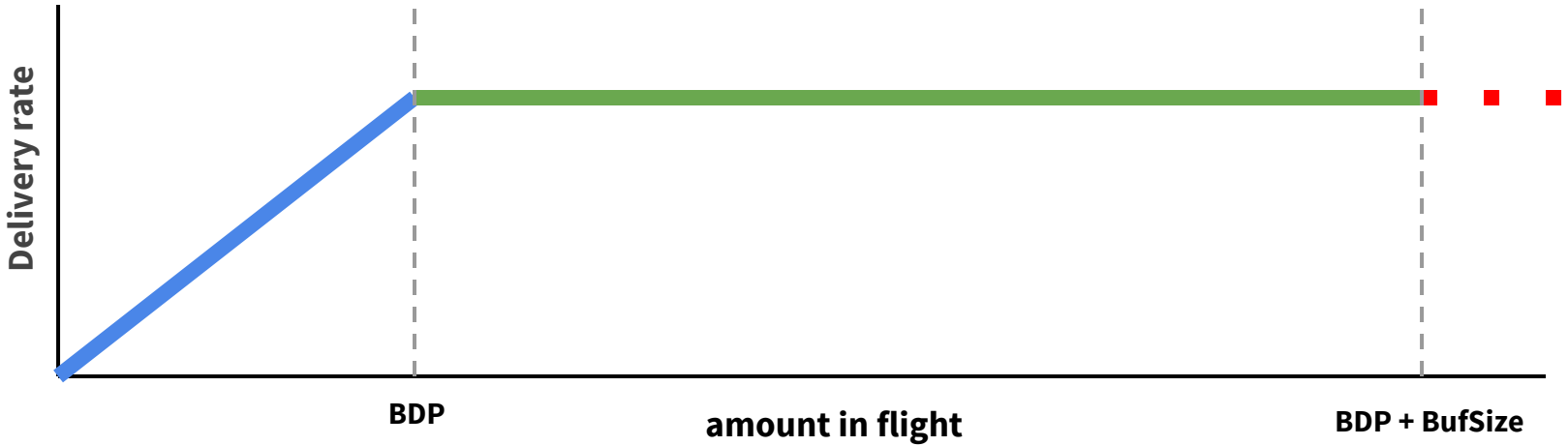
- Traditional TCP congestion control schemes are so-called loss-based
  - Fill up the queue to cause a packet loss as a signal of congestion.
- Over years vendors have been adding more and more buffer memory to devices to avoid packet loss.
- The result is long queuing delay in many network devices, including user devices such as home routers.

BBR

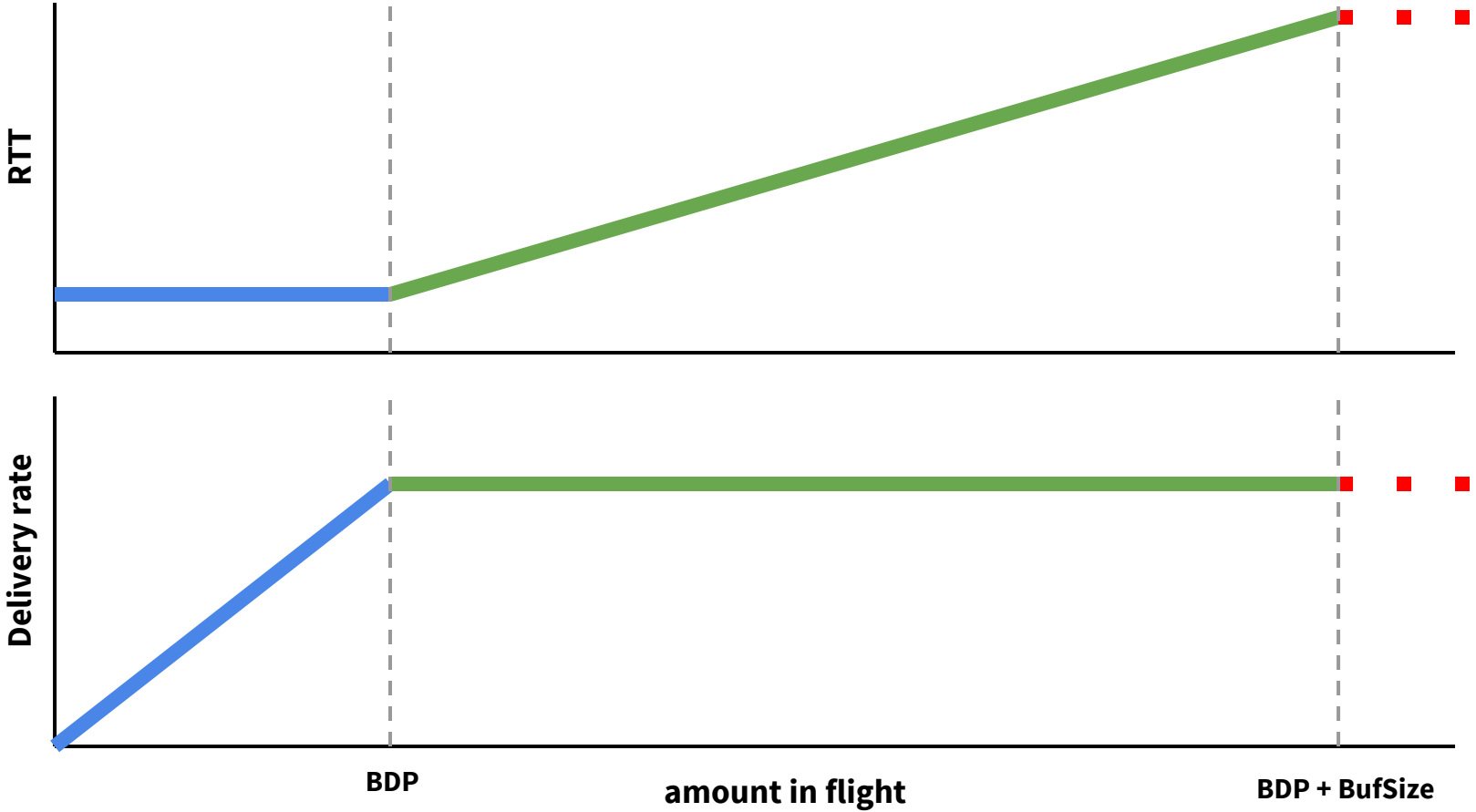
- Google's new TCP congestion control design
- Based on measurement of throughput and delay rather than packet loss.
- Try to achieve high throughput without causing long queuing delay.
- Only require deployment on the sender side
  - deployed on most Google servers
- Available in recent Linux kernel as well.

Network congestion and bottlenecks: bandwidth

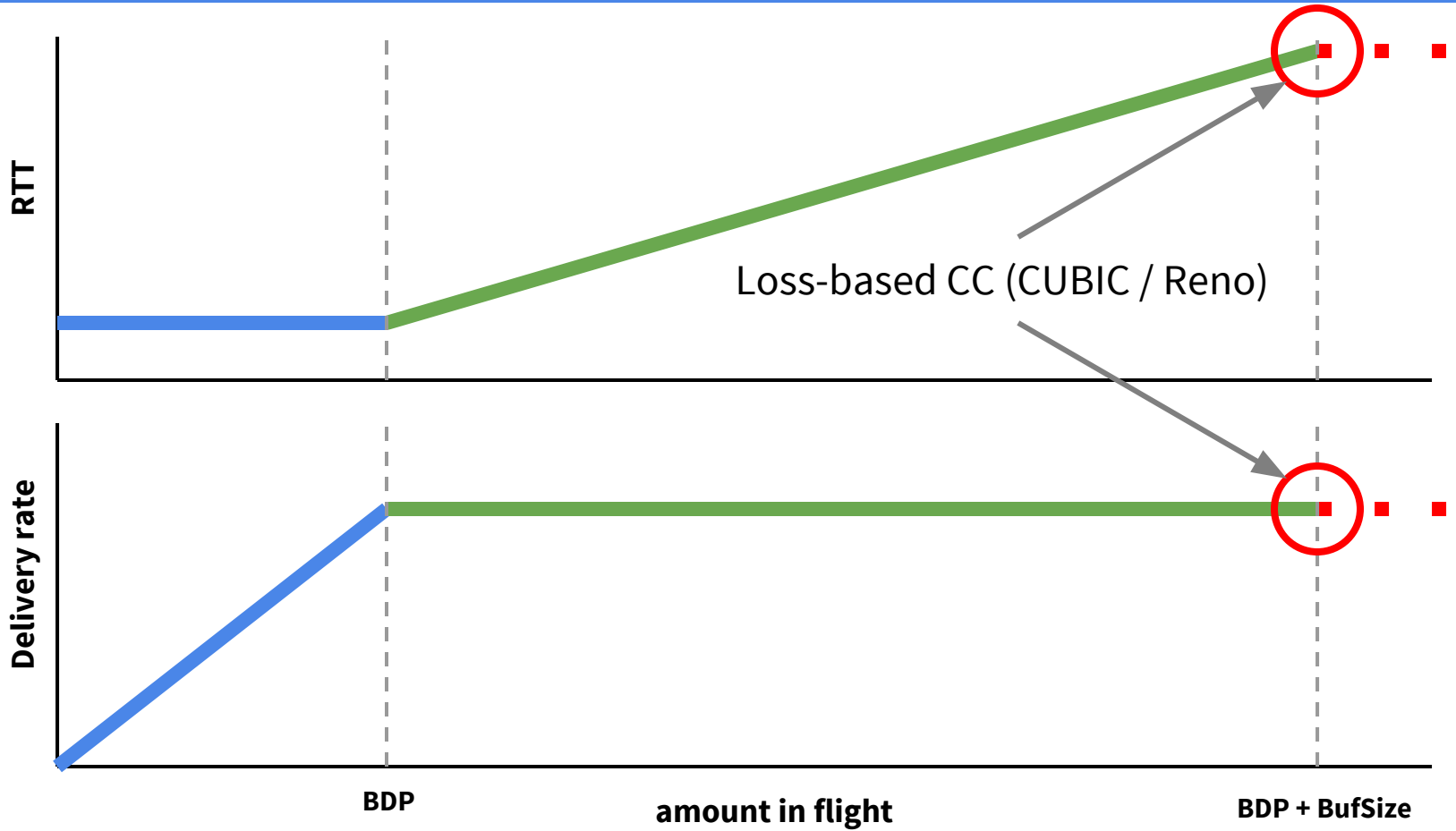




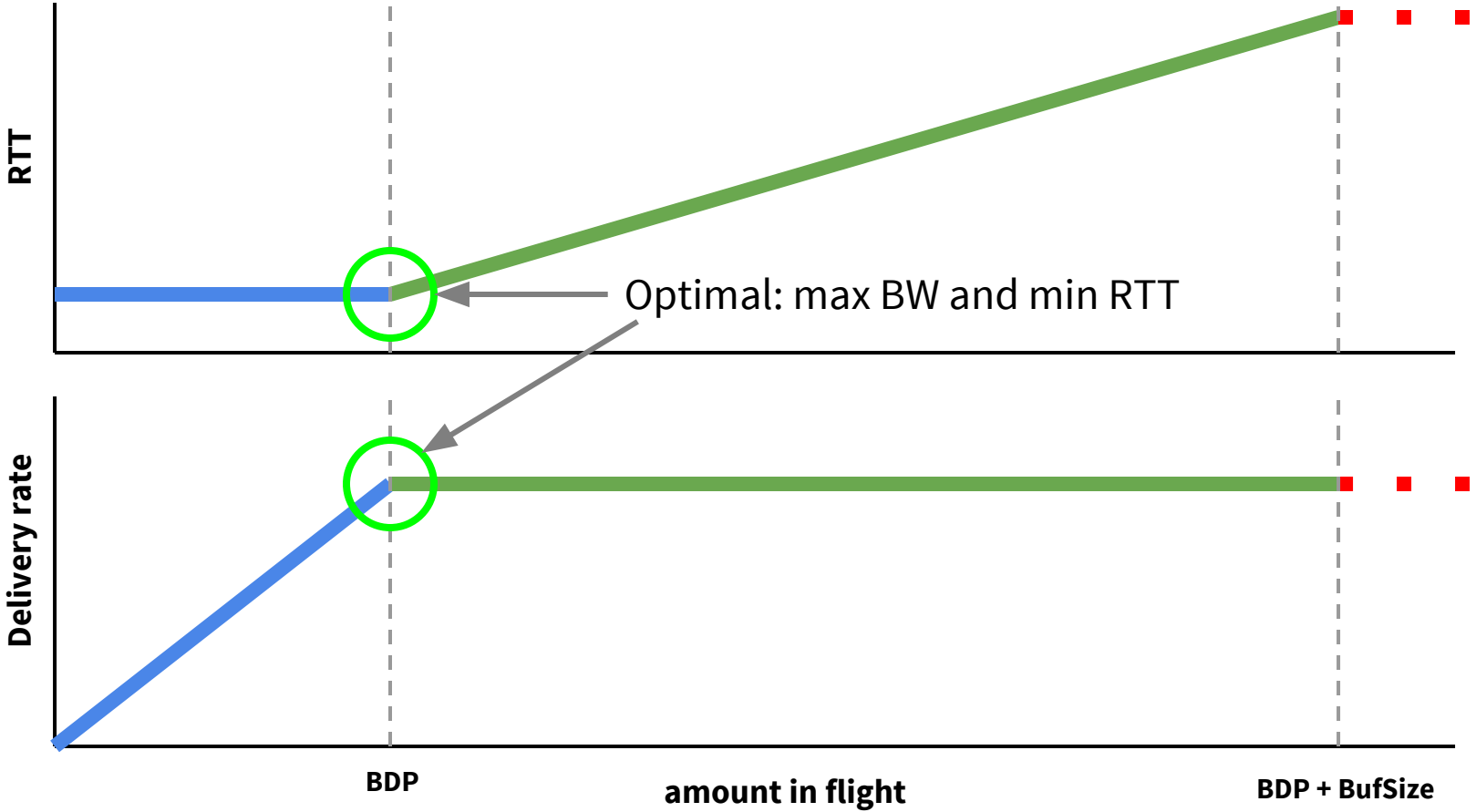
Network congestion and bottlenecks: delay



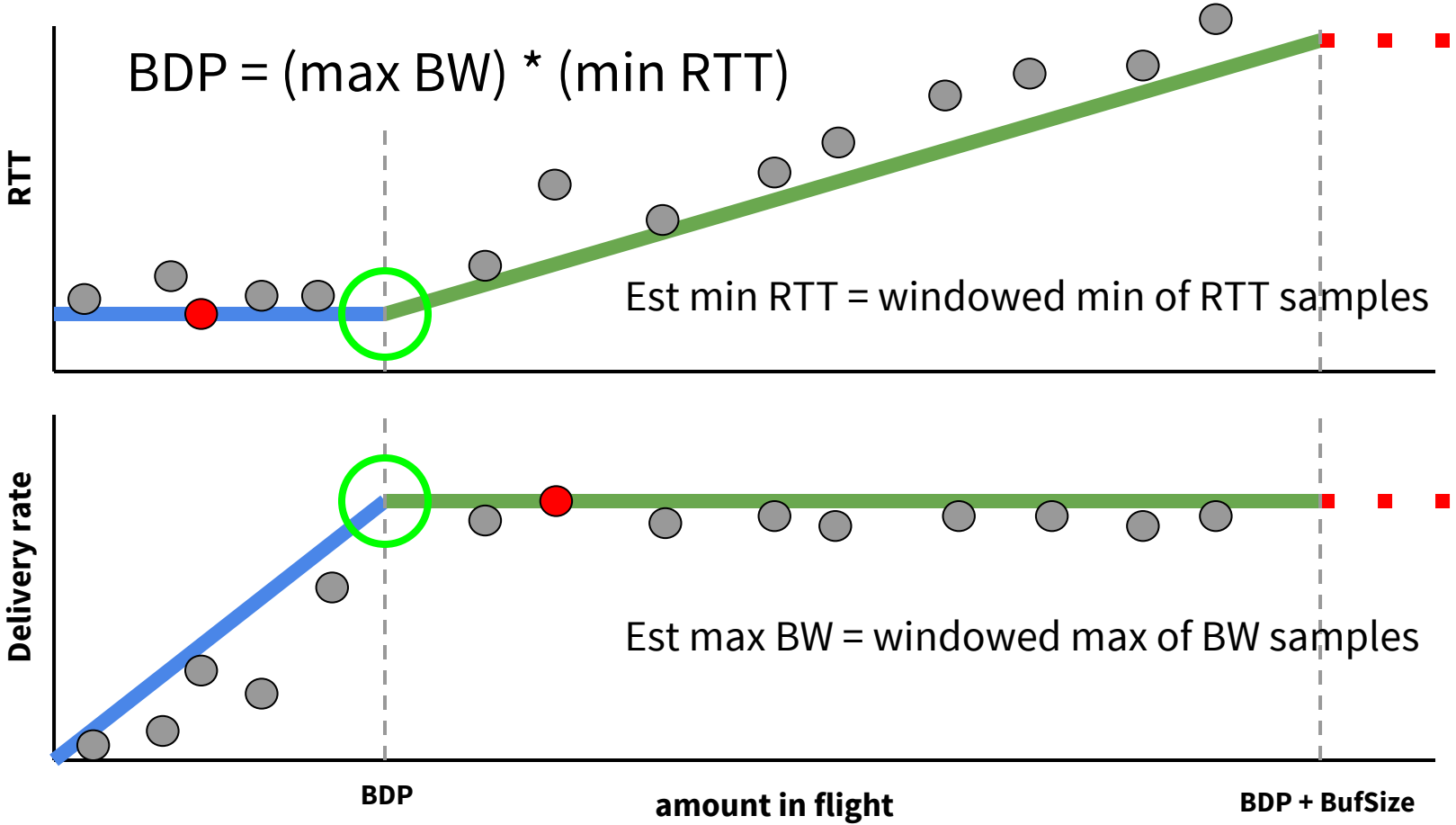
Loss-based Congestion Control



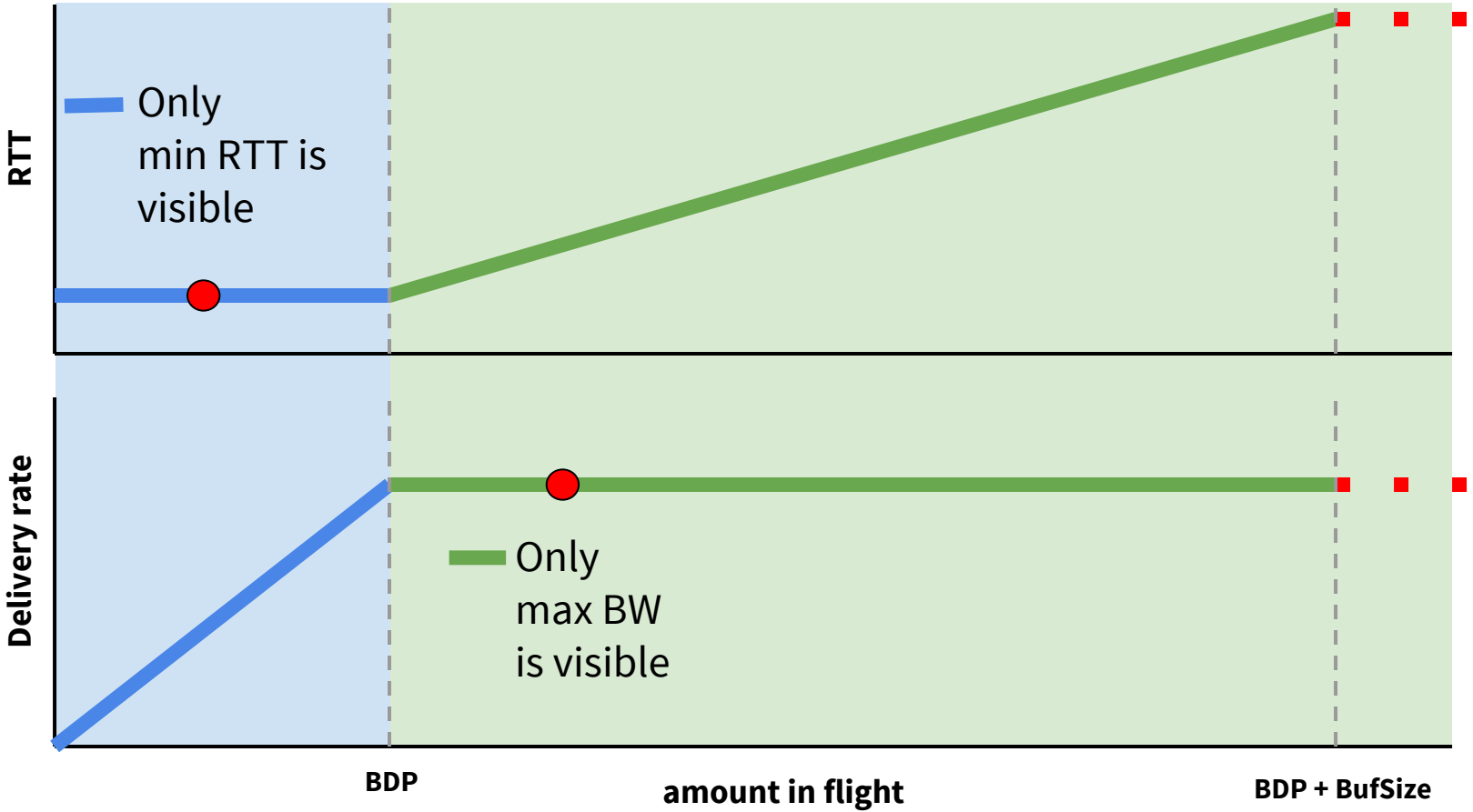
Optimal Operating Point



Estimating optimal point (max BW, min RTT)



To see max BW, min RTT: probe both sides of BDP



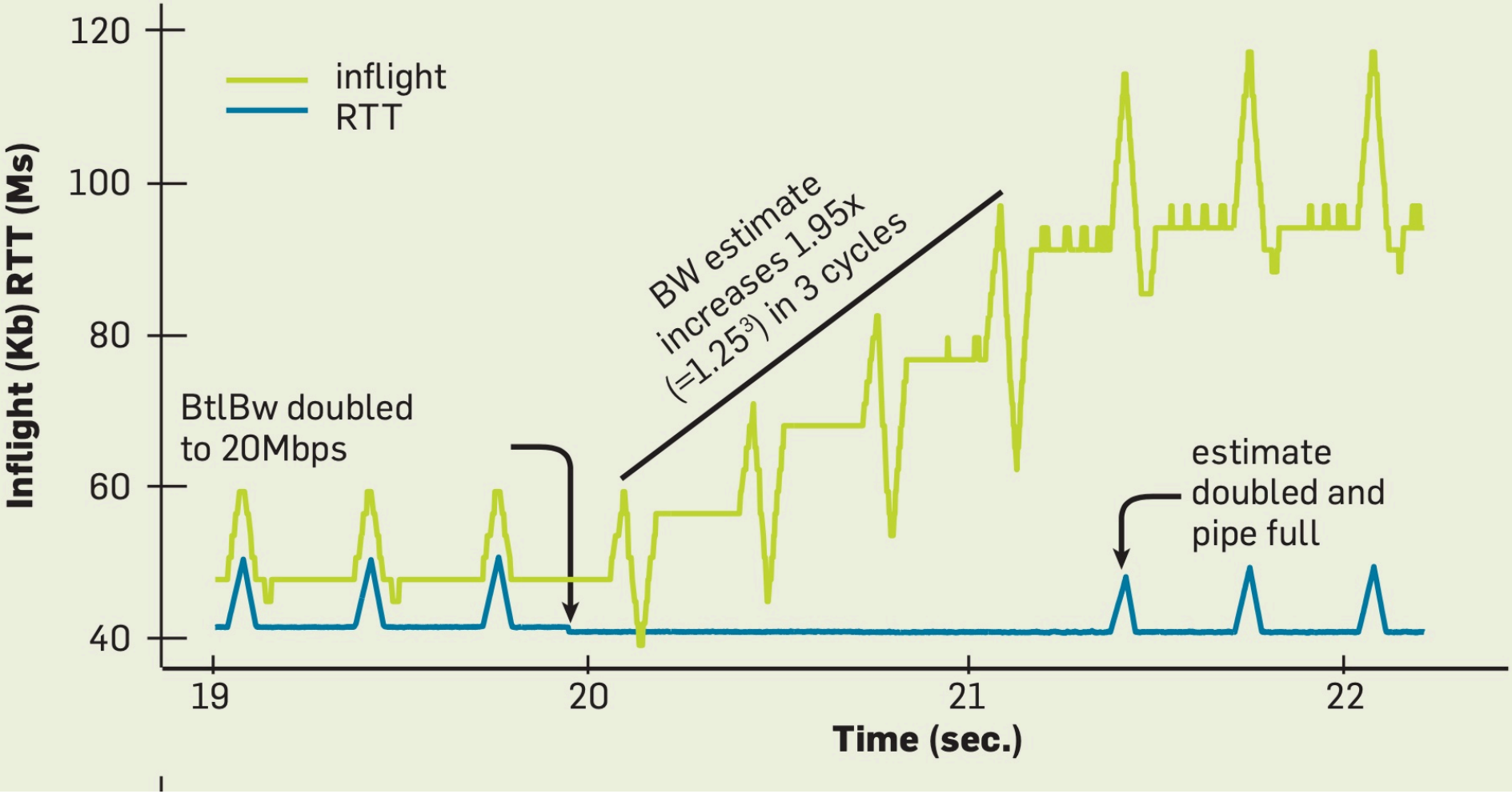
Measurement

- For each packet sent, the sender keeps track of the time and seq#.
- When the corresponding ACK is received, the sender can get a sample of
  - RTT
  - Bytes in flight (current seq# - ACK#)
- These samples are used to derive
  - Path propagation delay = minimal RTT
  - Bottleneck bandwidth = bytes delivered / time

Rate control

- Pace the packet sending so that bytes in flight does not exceed estimated BW*RTT product.
- Periodically send more bytes in flight to probe if more bandwidth has become available.
- Periodically send less bytes in flight to see if minimal RTT remains the same.

Figure 5. Bandwidth change.



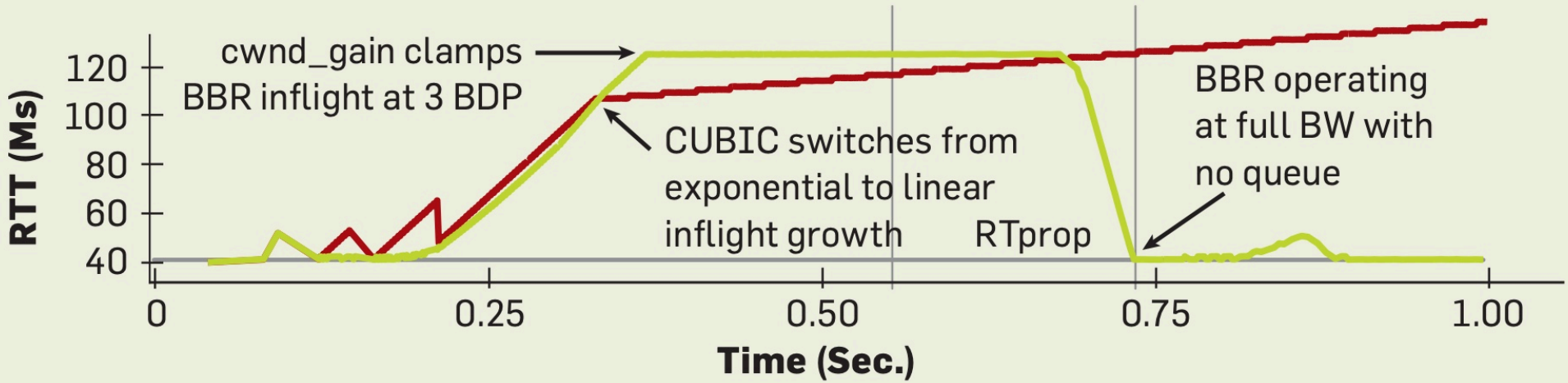

RTT (Ms)
120
100
80
60
40
cwnd_gain clamps
BBR inflight at 3 BDP
CUBIC switches from
exponential to linear
inflight growth
RTprop
BBR operating
at full BW with
no queue
0
0.25
0.50
0.75
1.00
Time (Sec.)

Figure 7. First eight seconds of 10Mbps, 40ms cubic and BBR flows.