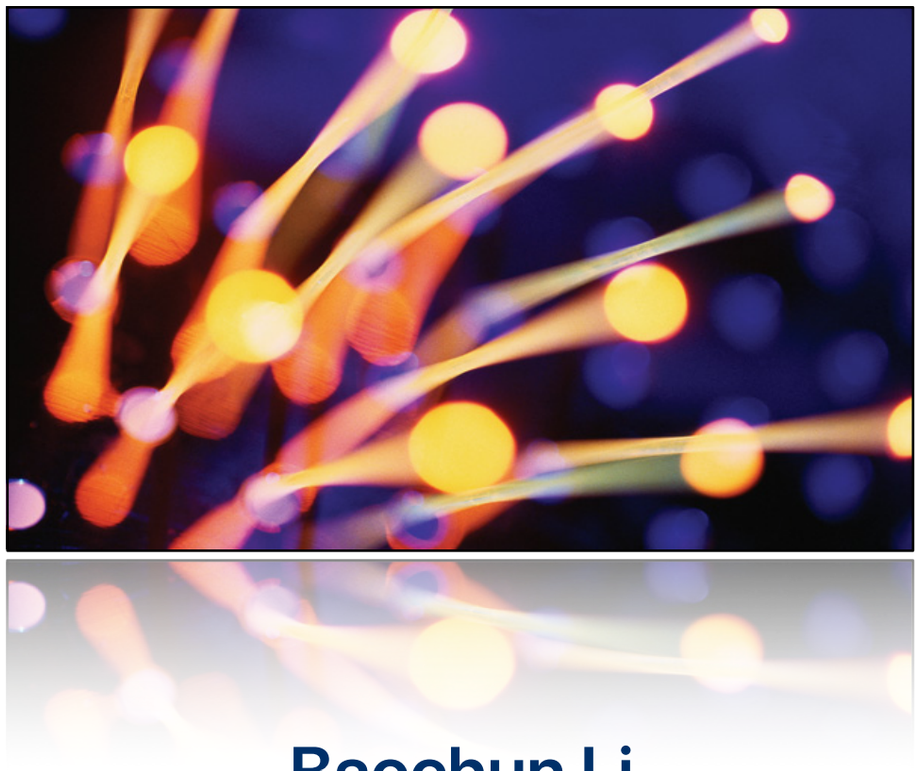
Episode 4. Principles of Congestion Control



Baochun Li
Department of Electrical and Computer Engineering
University of Toronto

Congestion control — a network-wide problem of managing shared resources

Saltzer Chapter 7.6.1, 7.6.2, 7.6.3, 7.6.4, 7.6.5; Keshav Chapter 9.7, CUBIC paper, BBR paper

Shared resources: everywhere in a system

Resource sharing examples in systems:

Many virtual processors (threads) sharing a few physical processors using a thread manager

A multilevel memory manager creates the illusion of large, fast virtual memories by combining a small and fast shared memory with large and slow storage devices

In networks, the resource that is shared is a set of communication links and the supporting packet forwarding switches

They are geographically and administratively distributed — managing them is more complex!

Analogy: Supermarket vs. Packet Switch

Queues are started to manage the problem that packets may arrive at a switch at a time when the outgoing link is already busy transmitting another packet

Just like checkout lines in the supermarket

Any time there is a shared resource, and the demand for that resource comes from several statistically independent sources, there will be fluctuations in the arrival of load

Thus there will be fluctuations in the length of the queue, and the time spent waiting for service in the queue

Offered load > capacity of a resource: overloaded

Congestion Collapse

Competition for resource may lead to waste of resource

Counter-intuitive, but the supermarket analogy can help understand it

Customers who are tired of waiting may just walk out, leaving filled shopping carts behind

Someone has to put the goods from abandoned carts back to the shelves

One of two of the checkout clerks leave their registers to do so

The rate of sales being rung up drops while they are away

The queues at the remaining registers grow longer

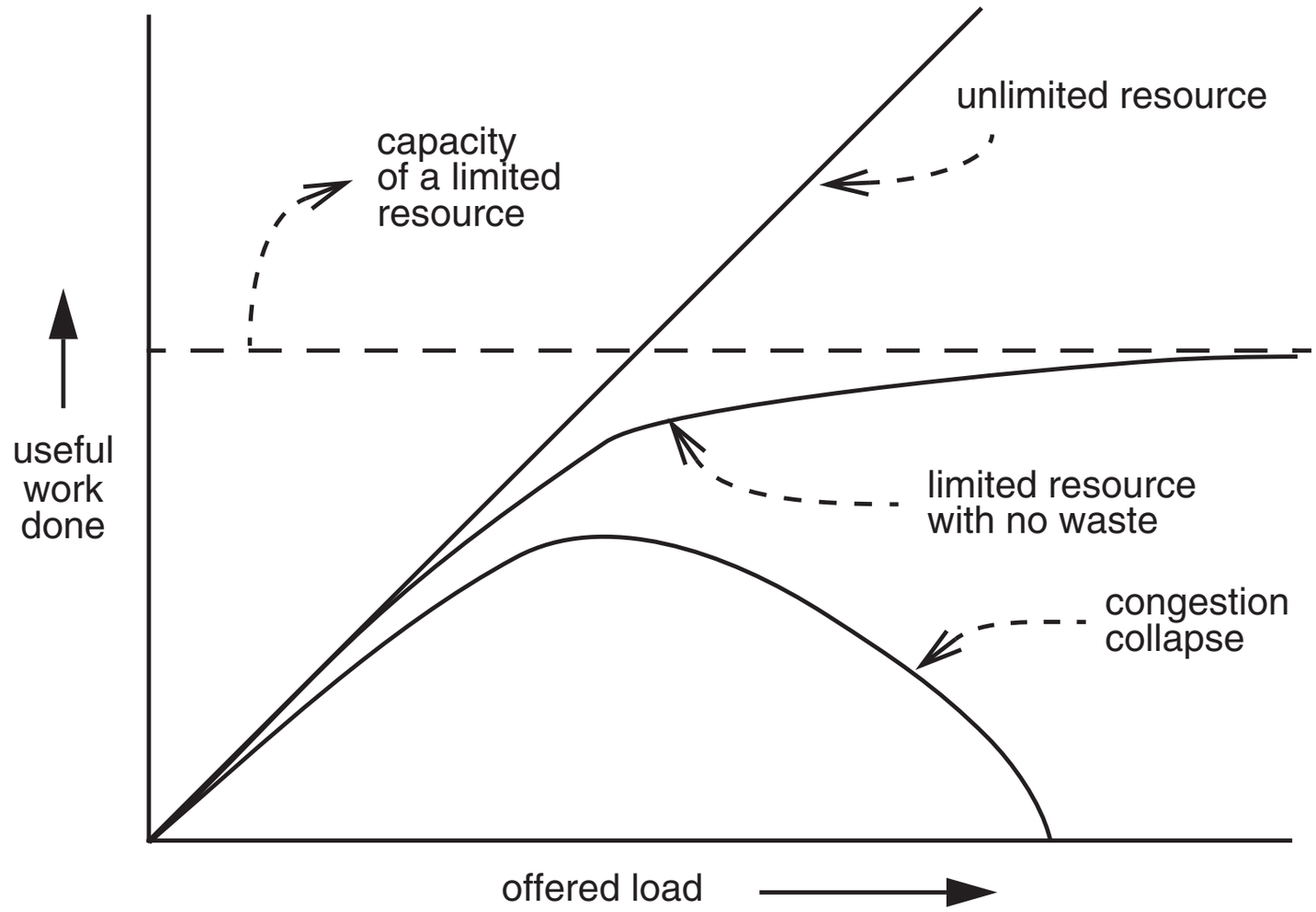
Causing more people to abandon their carts

Eventually, the clerks will be doing nothing but restocking

Self-sustaining nature of congestion collapse

Once temporary congestion induces a collapse, even if the offered load drops back to a level that the resource can handle, the already induced waste rate can continue to exceed the capacity of the resource

This will cause it to continue to waste the resource, remain congested indefinitely



Primary goal of resource management

Avoid congestion collapse!

by increasing the capacity of the resource

by reducing the offered load

A congestion control system is fundamentally a feedback system

A delay in the feedback path can lead to oscillations in load

The Supermarket and Call Centre Analogies

In a supermarket, a store manager can be used to watch the queues at the checkout lines

Whenever there are more than two or three customers in any line, the manager calls for staff elsewhere in the store to drop what they are doing, and temporarily take stations as checkout clerks

This practically increases capacity

When you call customer service, you may hear an automatic response message

"Your call is important to us. It will be 30 minutes to we can answer."

This may lead some callers to hang up and try again at a different time

This practically decrease load

Both may lead to oscillations

Possible ideas to address these challenges (Ch. 7.6.5)

Overprovisioning

Basic idea: configure each link of the network to have 125% or 200% as much capacity as the offered load at the busiest minute of the day

- Works best on interior links of a large network, where no individual client represents more than a tiny fraction of the load
- Average load offered by a large number of statistically independent sources is relatively stable

Problems

- Odd events can disrupt statistical independence
- Overprovisioning on one link will move congestion to another
- At the edge, statistical averaging stops working — **flash crowd**
- User usage patterns may adapt to the additional capacity

Pricing in a market: the “invisible hand”

Since network resources are just another commodity with limited availability, it should be possible to use pricing as a congestion control mechanism

- If demand for a resource temporarily exceeds its capacity, clients will bid up the price
- The increased price will cause some clients to defer their use of the resource until a time when it is cheaper, thereby reducing offered load
- It will also induce additional suppliers to provide more capacity

Challenges —

- How do we make it work on the short time scales of congestion?
- Clients need a way to predict the costs in the short term, too
- There has to be a minimal barrier of entry by alternate suppliers

How do we address these challenges?

Decentralized schemes are extremely scalable

Case in point:

the Internet

Cross-layer Cooperation: Feedback (Ch. 7.6.3)

Cross-layer feedback: basic idea

The packet forwarder that notices congestion provides feedback to one or more end-to-end layer sources

The end-to-end source responds by reducing its offered load

The best solution: the packet forwarder simply discards the packet

Simple and reliable!

Which packet to discard?

The choice is not obvious —

The simplest strategy, **tail drop**, limits the size of the queue, and any packet that arrives when the queue is full gets discarded

A better technique, called **random drop**, chooses a victim from the queue at random

- The sources that are contributing the most to congestion are the most likely to receive the feedback

Another refinement, called **early drop**, begins dropping packets before the queue is completely full, in the hope of alerting the source sooner

The goal of early drop is to start reducing the offered load as soon as the possibility of congestion is detected, rather than waiting till congestion is confirmed — **avoidance** rather than **recovery**

Random drop + early drop: random early detection (RED)

Cross-layer Cooperation: Control

What should the end-to-end protocol do?

The end-to-end protocol learns of a lost packet, what now?

One idea: just retransmit the lost packet, and continue to send more data as rapidly as its application supplies it

This way, it may discover that by sending packets at the greatest rate it can sustain, it will push more data through the congested packet forwarder

The problem: If this is the standard mode of operation of all end hosts, congestion will set in and all will suffer

"The tragedy of the commons"

Two things the end-to-end protocol can do

Be careful about the use of timers

involves setting the timer's value

Pace the rate at which it sends data — automatic rate adaptation

involves managing the flow control window

Both require having an estimate of the round-trip time between the two ends of the protocol

The retransmit timer

With congestion, an expired timer may imply that either a queue in the network has grown too long, or a packet forwarder has intentionally discarded the packet

We need to reduce the rate of retransmissions

Idea: round trip time estimates

- develop an estimate of the round trip time by directly measuring it
- Longer queuing delays will increase the observed round-trip times
- These observations will increase the round-trip estimate used for setting future retransmit timers

When a timer does expire, exponential backoff for the timer interval should be used for retransmitting the same packet

- Effectively avoids contributing to congestion collapse

Automatic rate adaptation

The flow control window and the receiver's buffer should both be at least as large as the bottleneck data rate multiplied by the round trip time — the BDP ("bandwidth-delay product")

But if it is larger, it will result in more packets piling up in the queue of the bottleneck link

We need to ensure that the flow control window is no larger than necessary

The original design of TCP

In the original TCP design, the only form of acknowledgment to the sender was "I have received all the bytes up to X."

But not in the form of "I am missing bytes Y through Z."

The consequences —

When a timer expired due to a lost packet, as soon as the sender retransmitted that packet, the timer of the next packet expired, causing its retransmission

This will repeat until the next acknowledgment returns, a full round trip later

On long-delay routes, the flow control window may be large

Each discarded packet will trigger retransmission of a window full of packets — congestion collapse!

Solution?

By the time this effect was noticed, TCP was already widely deployed, so changes to TCP were severely constrained — they have to be "backward compatible"

The result — one expired timer leads to slow start

- Send just one packet, and wait for its acknowledgment
- For each acknowledged packet, add one to the window size
- In each RTT, the number of packets that the sender sends doubles
- This repeats till
  - The receiver's window size has been reached — the network is not the bottleneck
  - a packet loss has been detected — **how?**

Duplicate acknowledgment

The receiving TCP implementation is modified slightly

Whenever it receives an out-of-order packet, it sends back a duplicate of its latest acknowledgment

Such a duplicate can be interpreted by the sender as a NAK

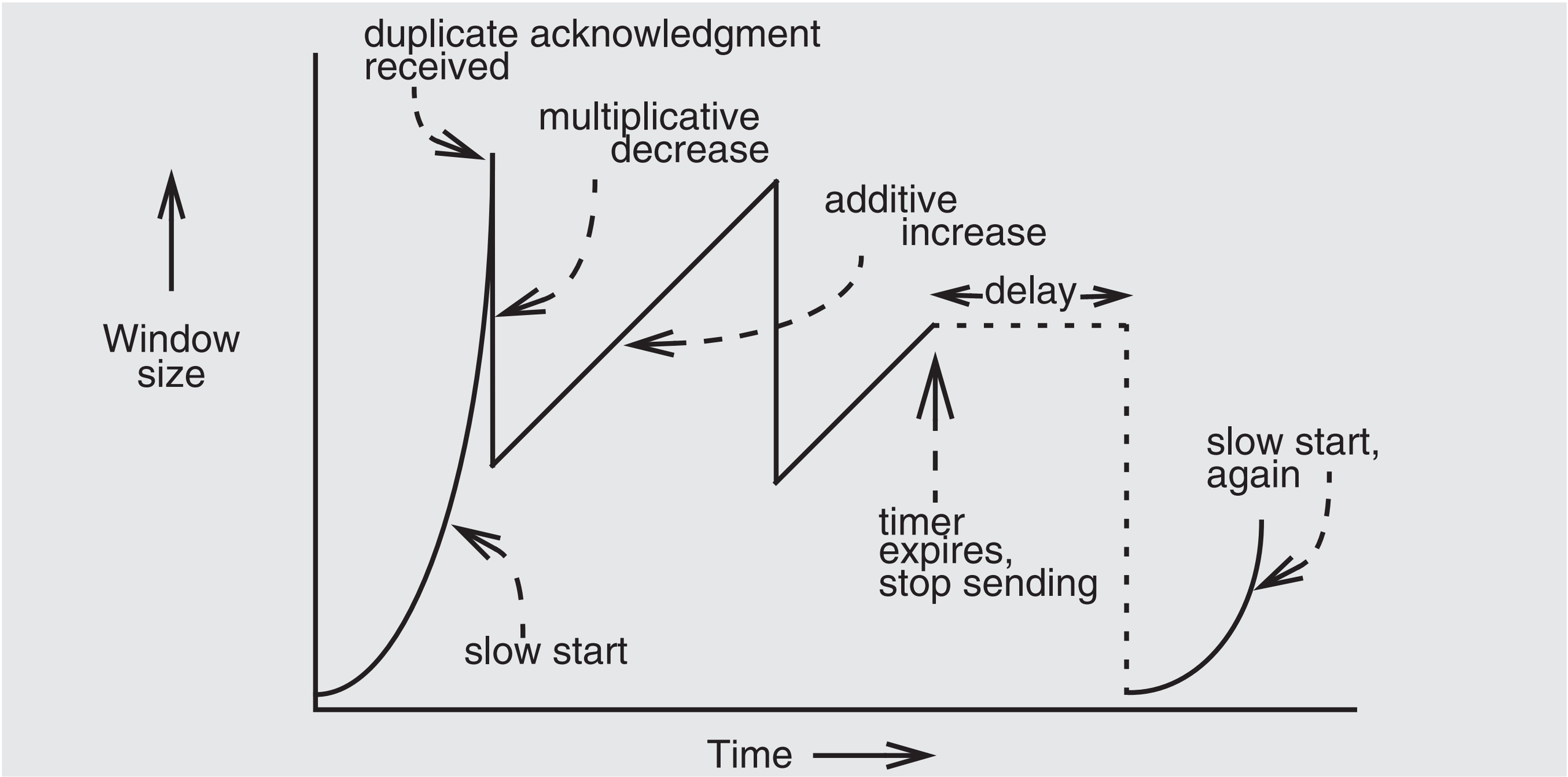
The sender then operates in an “equilibrium mode”

Upon duplicate acknowledgment, the sender retransmits just the first unacknowledged packet and also drops its window size to some fixed fraction of its previous size — after this it probes gently for more capacity by doing

Additive increase: whenever all the packets in a round-trip time are successfully acknowledged, the sender increases the window size by 1

Multiplicative decrease: Whenever a duplicate acknowledgment arrives again, the sender decreases the size of the window again, by a fixed fraction

The AIMD TCP


duplicate acknowledgment received
multiplicative decrease
additive increase
delay
Window size
slow start
timer expires, stop sending
slow start, again
Time

This is where the story stops in a typical textbook

But it is a story far from what happens in reality!

Real-world measurement studies have shown

By probing 5000 popular web servers, researchers have found that only 15-20% uses AIMD TCP!

> P. Yang, *et al.* "TCP Congestion Avoidance Algorithm Identification," IEEE ICDCS 2011.

What about the other 80%?

It turns out that web servers, which are what define the Internet, use a wide variety of different TCP protocols, each with its own congestion control protocol

> But why?

The fundamental problem with AIMD TCP

As Internet evolves, the number of long latency and high bandwidth networks grows

The Bandwidth and Delay Product (BDP) — The total number of packets in flight, determined by the flow control window size on the sender, must fully utilize the bandwidth

Standard AIMD TCP increases its congestion window size too slowly in high BDP environments

Example: Bandwidth 10Gbps, RTT 100ms, Packet size 1250 bytes, takes 10,000 seconds to fully utilize

New congestion control — design objectives

Highly scalable to high BDP environments

Very slow (gentle) window increase at the saturation point

Fair to AIMD TCP flows — backward compatibility

Binary Increase Congestion (BIC)

After a packet loss, reduces its window by a multiplicative factor of β, the default is 0.2

The window size just before reduction is set to W_{max} and after reduction W_{min}

In the next step, it finds the midpoint using these two sizes and jump there — binary search

But if midpoint is very far from W_{min}, a constant value called S_{max} is used

If no loss, W_{min} is set to the new window size

Binary Increase Congestion (BIC)

The process continues until the increment is less than a constant value of S_{min}

Then it is set to the maximum window

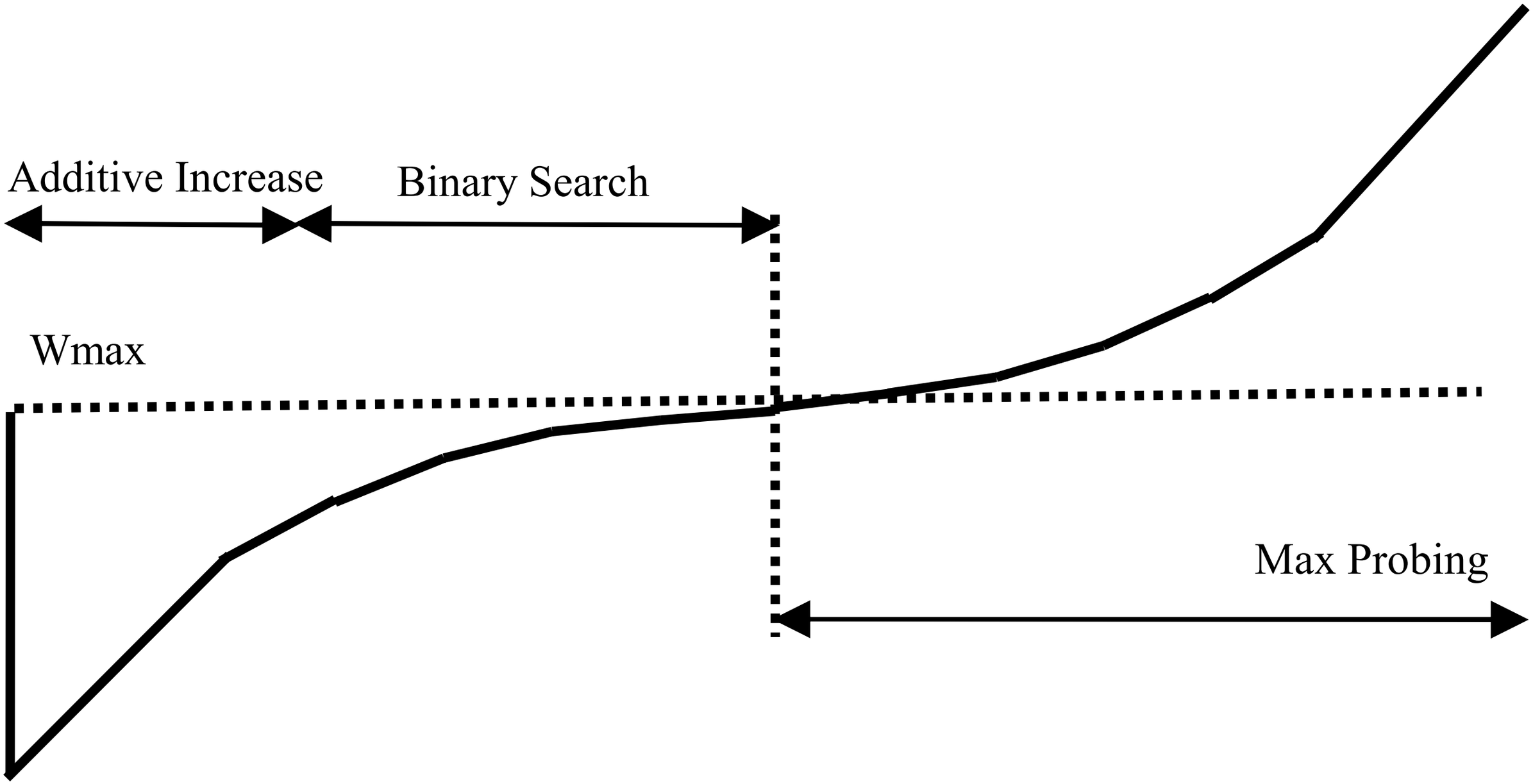
If no loss, new maximum must be found and it enters a "max probing" phase

Window growth function is exactly symmetric to the previous part

BIC is proposed by Injong Rhee's group at NCSU in an INFOCOM 2004 paper, and later was used in the Linux kernel and set to the default TCP (since 2.6.13)

BIC: During the “congestion epoch”

Congestion epoch: defined as the interval between two packet losses



Problems with BIC

BIC works very well in production, but in low speed or short RTT networks it is too aggressive for TCP

Different phases like binary search increase, max probing, S_{max} and S_{min}, make its implementation not very efficient

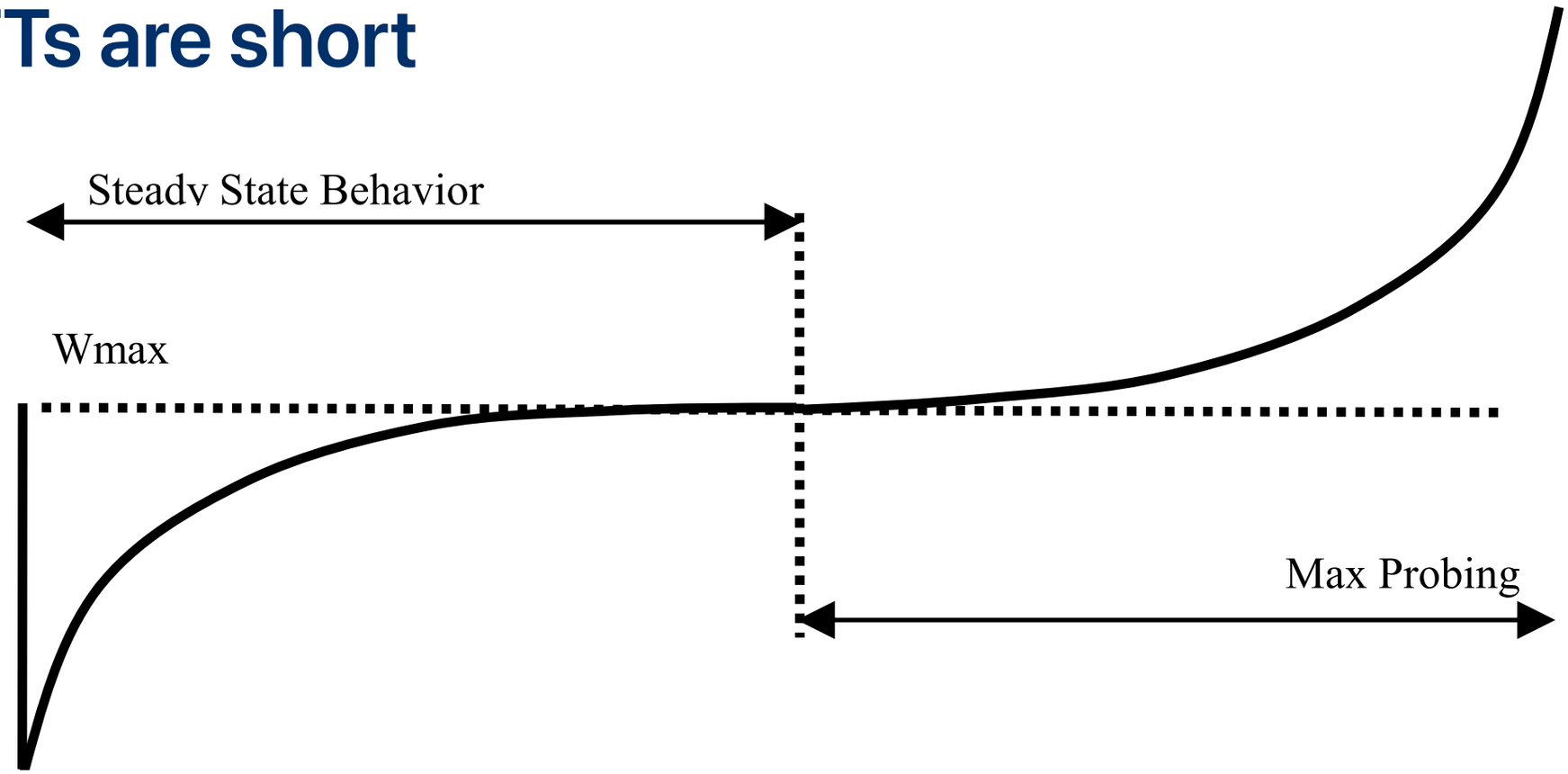
A new congestion control protocol is required to solve these problems, while keeping its advantages of stability and scalability

CUBIC

As the name suggests, it uses cubic function for window growth

It uses time instead of RTT to increase the window size

It contains a "TCP mode" to behave the same as AIMD TCP when RTTs are short



CUBIC, again, was proposed by Injong Rhee's group at NCSU, and was later used in the Linux kernel and set to the default TCP (since 2.6.19)

CUBIC: Controlling the window size

After a packet loss, reduces its window by a multiplicative factor of β, the default is 0.2

The window size just before reduction is set to W_{max}

After it enters into congestion avoidance, it starts to increase the window using a cubic function

The plateau of cubic function is set to W_{max}

Size of the window grows in concave mode to reach W_{max}, then it enters the convex part

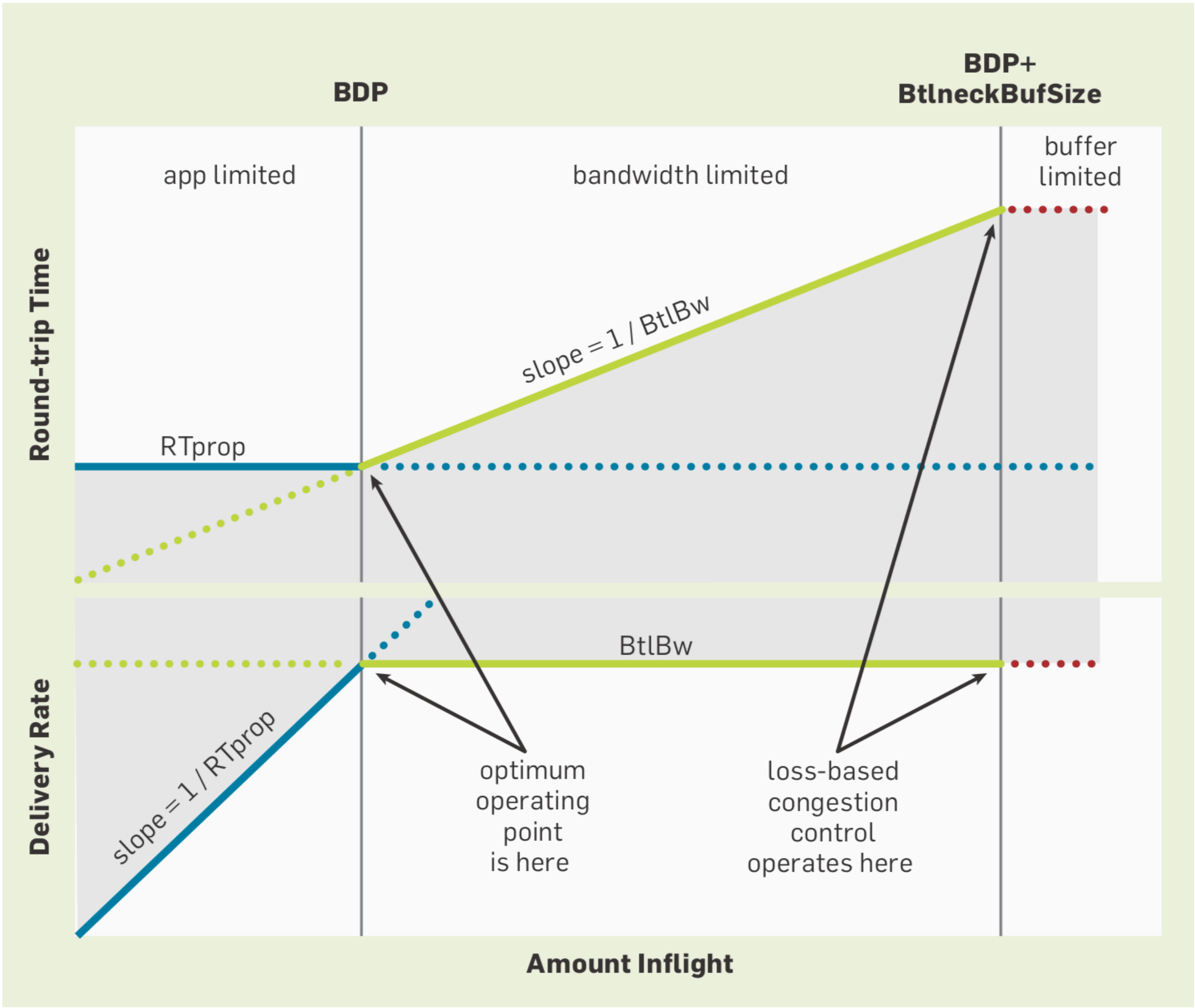
Real-world measurement studies

Over 40% of the web servers uses BIC/CUBIC TCP!

P. Yang, *et al.* "TCP Congestion Avoidance Algorithm Identification," IEEE ICDCS 2011.

BBR: Congestion-Based Congestion Control

(Google Inc., make-tcp-fast project, available in Linux Kernel since 4.9)


BDP
BDP+
BtlneckBufSize
app limited
bandwidth limited
buffer
limited
Round-trip Time
slope = 1 / BtlBw
RTprop
BtlBw
Delivery Rate
slope = 1 / RTprop
optimum
operating
point
is here
loss-based
congestion
control
operates here
Amount Inflight

BBR: Main Ideas

BBR algorithm differs from loss-based congestion control in that it pays relatively little attention to packet loss

Instead, it focuses on the actual bottleneck bandwidth

In the "startup" state, it behaves like most traditional congestion-control algorithms

quickly ramps up the sending rate in an attempt to measure the bottleneck bandwidth

BBR: Stop Ramping Up Early

Instead of continuing to ramp up until it experiences a dropped packet, it watches the bottleneck bandwidth measured for the last several round-trip times

Once the bottleneck bandwidth stops rising, BBR concludes that it has found the effective bandwidth of the connection and can stop ramping up

This has a good chance of happening well before packet loss would begin

BBR: Drain State

In measuring the bottleneck bandwidth, BBR probably transmitted packets at a higher rate for a while

Some of them will be sitting in queues waiting to be delivered

To drain those packets out of the buffers where they languish, BBR will go into a “drain” state, during which it will transmit below the measured bandwidth until it has made up for the excess packets sent before

BBR: Steady State

Once the drain phase is done, BBR goes into the steady-state mode where it paces outgoing packets so that the packets in flight is more-or-less the calculated BDP

Scale the rate up by 25% periodically to probe for an increase in effective bandwidth

Saltzer Chapter 7.6.1, 7.6.2, 7.6.3, 7.6.4, 7.6.5; Keshav Chapter 9.7, CUBIC paper, BBR paper