

# **Congestion Control in High Speed Networks: Part 2**

Lecture 6  
Subir Varma

# TCP Cubic

# TCP CUBIC

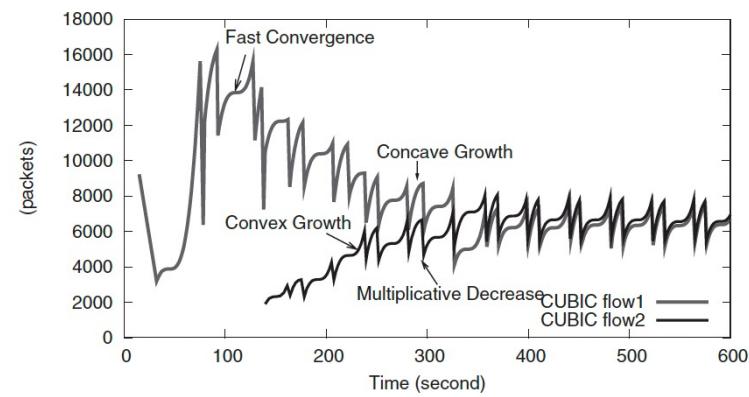
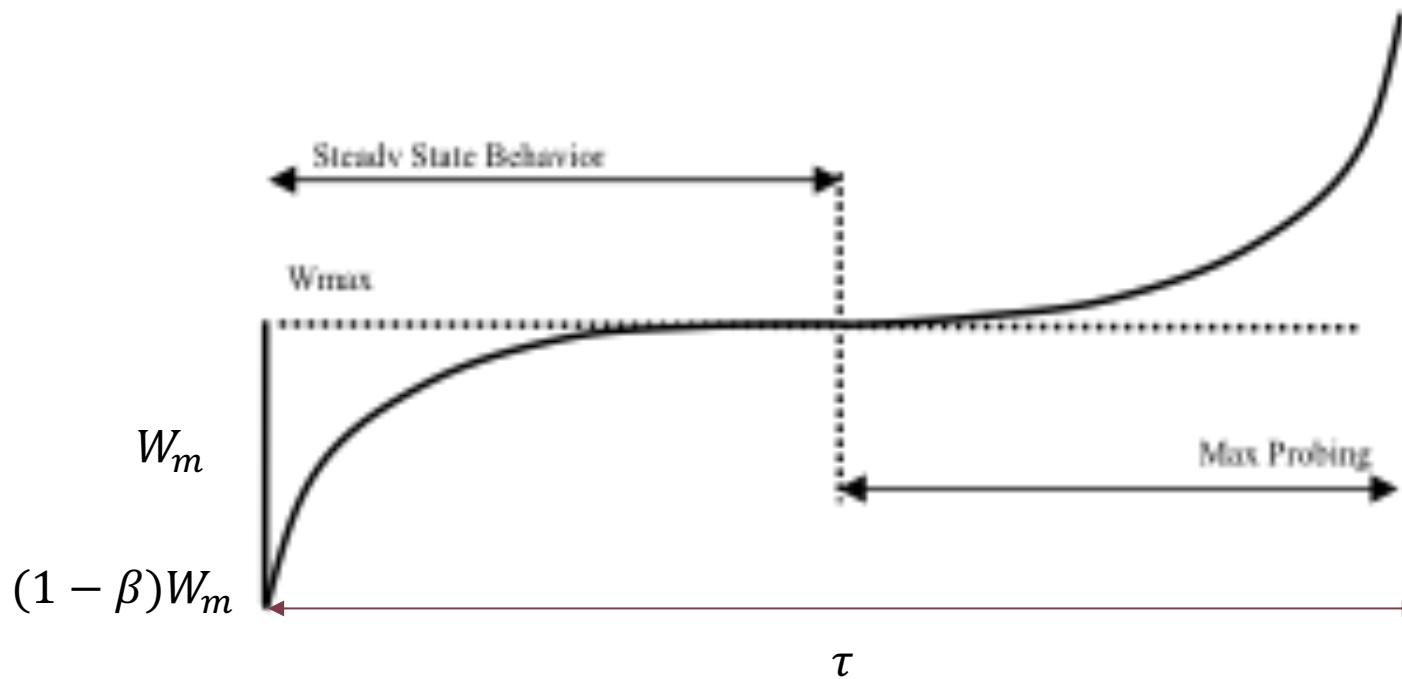
TCP CUBIC is a follow-on design from the same research group that had earlier come up with TCP BIC. Their main motivation in coming up with CUBIC was to improve on BIC in the following areas:

- (1) BIC's window increase function is too aggressive for Reno, especially under short RTT or low-speed networks, and
- (2) Reduce the complexity of BIC's window increment/decrement rules to make the algorithm more analytically tractable and easier to implement.
- (3) Solve the issue of high RTT unfairness in lower speed networks

# Changes from BIC to CUBIC

- ▶ To reduce the resulting complexity of the algorithm, CUBIC replaced the binary search by a cubic function, which contains both concave and convex portions.
- ▶ Another significant innovation in CUBIC is that **the window growth depends only on the real time between consecutive congestion events**, unlike TCP BIC or Reno in which the growth depends at the rate at which ACKs are returning back to the source.
  - This makes the window growth independent of the round trip latency, so that if multiple TCP CUBIC flows with differing RTTs are competing for bandwidth, then their windows are approximately equal. This results in a big improvement in the RTT fairness for CUBIC.

# Window Growth Function for CUBIC



(a) CUBIC window curves.

# CUBIC Window Increase Function

$W(t)$ : Window size at time  $t$ , given that the last packet loss occurred at  $t = 0$

$W_m$ : Window size at which the last packet loss occurred

$\tau$ : Average length of a cycle

$\alpha, K$ : Parameters of the window increase function

$\beta$ : Multiplicative factor by which the window size is decreased after a packet loss, so that the new window is given by  $(1 - \beta)W_m$

The window size increase function for TCP CUBIC is given by

$$W(t) = \alpha(t - K)^3 + W_m$$

Note that at  $t = 0^+$ , the window size is given by  $W(0^+) = (1 - \beta)W_m$ , so that

$$(1 - \beta)W_m = \alpha(-K)^3 + W_m, \text{ i.e.,}$$

$$K = \sqrt[3]{\frac{\beta W_m}{\alpha}}$$

Note that  $K$  is not a constant

**Important Property:** The Cubic window increase function is not determined by the rate ACKs are returning, but solely by the real time interval elapsed since the last packet drop

# CUBIC Modes

- ▶ Upon receiving an ACK during congestion avoidance, CUBIC computes the window growth rate during the next RTT period using  $W(t)$  formula
- ▶ It sets  $W(t + RTT)$  as the candidate target value of congestion window.
- ▶ Suppose that the current window size is  $cwnd$ . Depending on the value of  $cwnd$ , CUBIC runs in three different modes.
  1. If  $cwnd$  is less than the window size that (standard) TCP would reach at time  $t$  after the last loss event, then CUBIC is in the TCP mode.
  2. If  $cwnd$  is less than  $W_{max}$ , then CUBIC is in the concave region
  3. If  $cwnd$  is larger than  $W_{max}$ , CUBIC is in the convex region.

# TCP Mode in CUBIC

- ▶ CUBIC is in the TCP Mode if  $W(t)$  is less than the window size that TCP Reno would reach a time  $t$ .
- ▶ In short RTT networks, TCP Reno can grow faster than CUBIC because its window increases by one every RTT, but CUBIC's window increase rate is independent of the RTT.
- ▶ To keep CUBIC's growth rate the same as that of Reno in this situation, CUBIC emulates Reno's window adjustment algorithm after a packet loss event using an equivalent AIMD( $a, b$ ) algorithm as shown next.

# TCP Mode in CUBIC (cont)

Recall that the average throughput for TCP Reno is given by

$$R = \frac{1}{T} \sqrt{\frac{3}{2p}}$$

For an AIMD( $a,b$ ) algorithm  $R$  is given by

$$R = \frac{1}{T} \sqrt{\frac{a(2-b)}{2bp}}$$

Hence, given  $b$ , a choice of  $a = \frac{3b}{2-b}$  will ensure that the AIMD algorithm has the same average throughput as TCP Reno

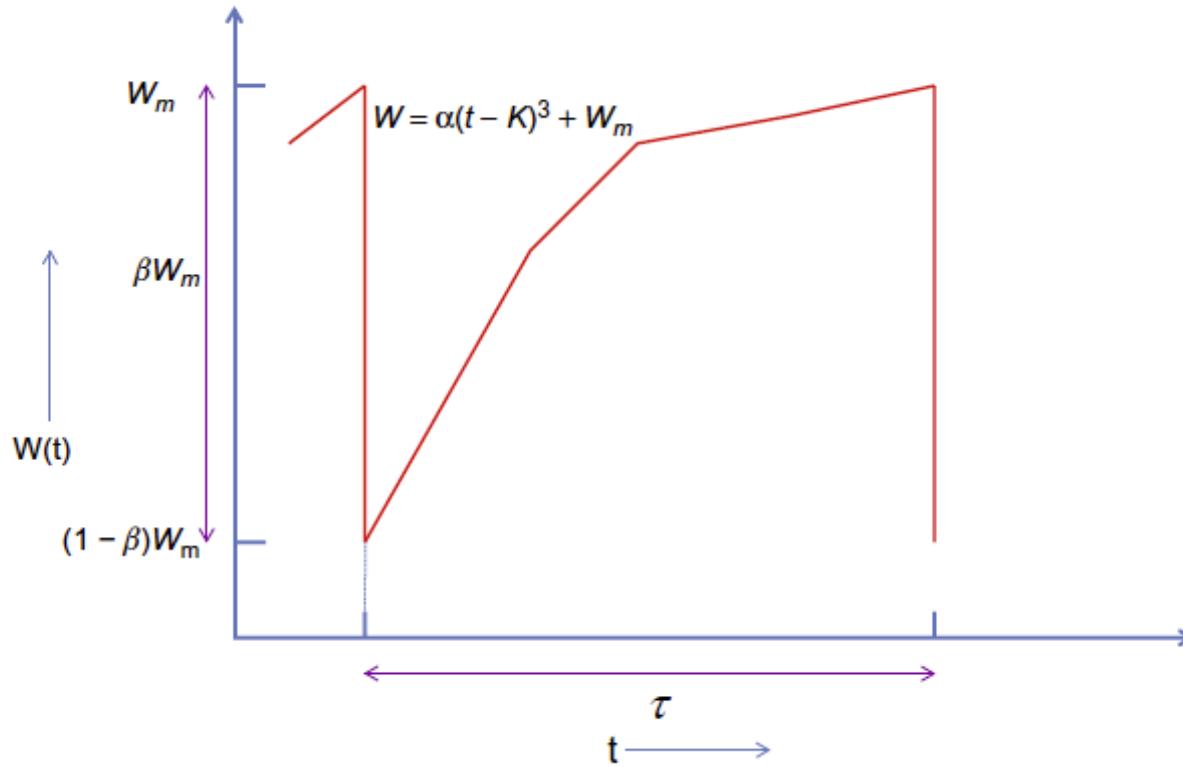
Because the window size increases by  $a$  in every round trip and there are  $t/T$  round trips in time  $t$ , it follows that the emulated CUBIC window size after  $t$  seconds is given by

$$W_{AIMD} = (1 - b)W_m + \frac{3b}{(2 - b)} \frac{t}{T}$$

If  $W_{AIMD}$  is larger than  $W_{CUBIC}$ , then  $W_{CUBIC}$  is set equal to  $W_{AIMD}$ .

Otherwise,  $W_{CUBIC}$  is used as the current congestion window size.

# CUBIC Analysis



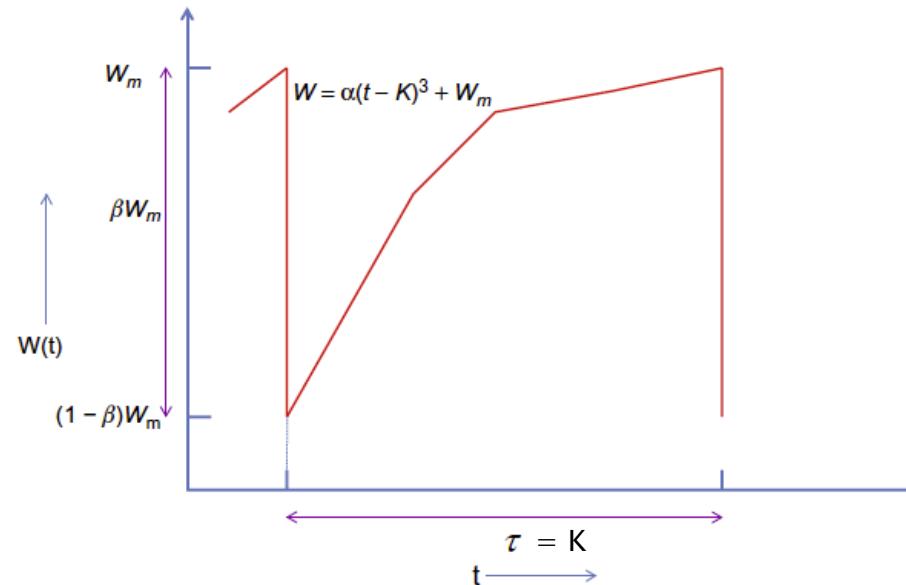
Assume that CUBIC is in steady state with one packet lost every  $\tau$  sec

# CUBIC Analysis

During the course of this cycle, the TCP window increases from  $(1 - \beta)W_m$  to  $W_m$ , and we will assume that this takes on the average  $\tau$  seconds. From [equation 15](#), it follows that

$$\tau = \sqrt[3]{\frac{\beta W_m}{\alpha}} \quad (32)$$

$W(t) = \alpha(t - K)^3 + W_m$  so that  $t = 0$  we have  $W(0) = (1 - \beta)W_m$  and at  $t = K$ , we have  $W(K) = W_m$



# CUBIC Analysis: Throughput

Using the deterministic assumption from Section 2.3 of Chapter 2, it follows that the average throughput is given by

$$R_{avg} = \frac{N}{\tau} \quad \text{where} \quad N = \frac{1}{p} \quad \text{is the number of packets transmitted during a cycle.}$$

N is also given by

$$N = \int_0^{\tau} \frac{W(t)}{T} dt$$

Following the usual recipe, we equate N to  $1/p$ , where p is the packet drop rate

$$\frac{1}{p} = \int_0^{\tau} \frac{W(t)}{T} dt$$

# CUBIC Analysis: Throughput

Note that

$$\int_0^\tau W(t)dt = \int_0^\tau [\alpha(t-\tau)^3 + W_m]dt = \tau W_m - \frac{\alpha\tau^4}{4}$$

It follows that

$$\begin{aligned}\frac{T}{p} &= W_m \left[ \frac{\beta W_m}{\alpha} \right]^{1/3} - \frac{\alpha}{4} \left[ \frac{\beta W_m}{\alpha} \right]^{4/3} \\ &= W_m^{4/3} \frac{(4-\beta)}{4} \left( \frac{\beta}{\alpha} \right)^{1/3}\end{aligned}$$

This implies that

$$W_m = \sqrt[4]{\left(\frac{T}{p}\right)^3 \frac{\alpha}{\beta} \left(\frac{4}{4-\beta}\right)^3}$$

and

$$\tau = \sqrt[3]{\left(\frac{\beta W_m}{\alpha}\right)} = \sqrt[4]{\frac{T}{p} \left(\frac{4}{4-\beta}\right) \left(\frac{\beta}{\alpha}\right)}$$

The average throughput is then given by

$$R_{avg} = \frac{N}{\tau} = \frac{1/p}{\tau} = \sqrt[4]{\frac{\alpha(4-\beta)}{4p^3 T \beta}}$$

# CUBIC Analysis: Response Function

$$R_{avg} = \frac{N}{\tau} = \frac{1/p}{\tau} = \sqrt[4]{\frac{\alpha(4-\beta)}{4p^3T\beta}}$$

Because CUBIC is forced to follow TCP Reno for smaller window sizes, its response function becomes

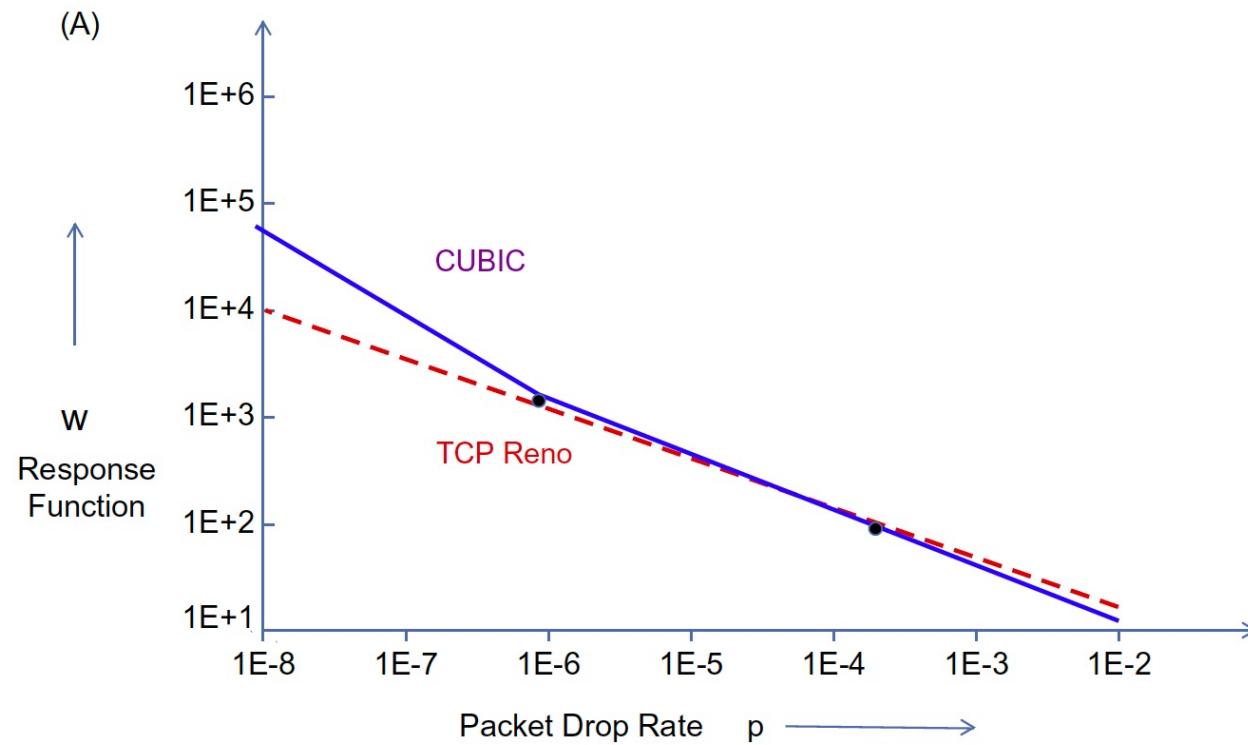
$$\log w = \begin{cases} 0.25 \log \frac{\alpha(4-\beta)}{4\beta} + 0.75 \log T - 0.75 \log p & \text{if } p < \bar{p} \\ 0.09 - 0.5 \log p & \text{if } p \geq \bar{p} \end{cases}$$

where

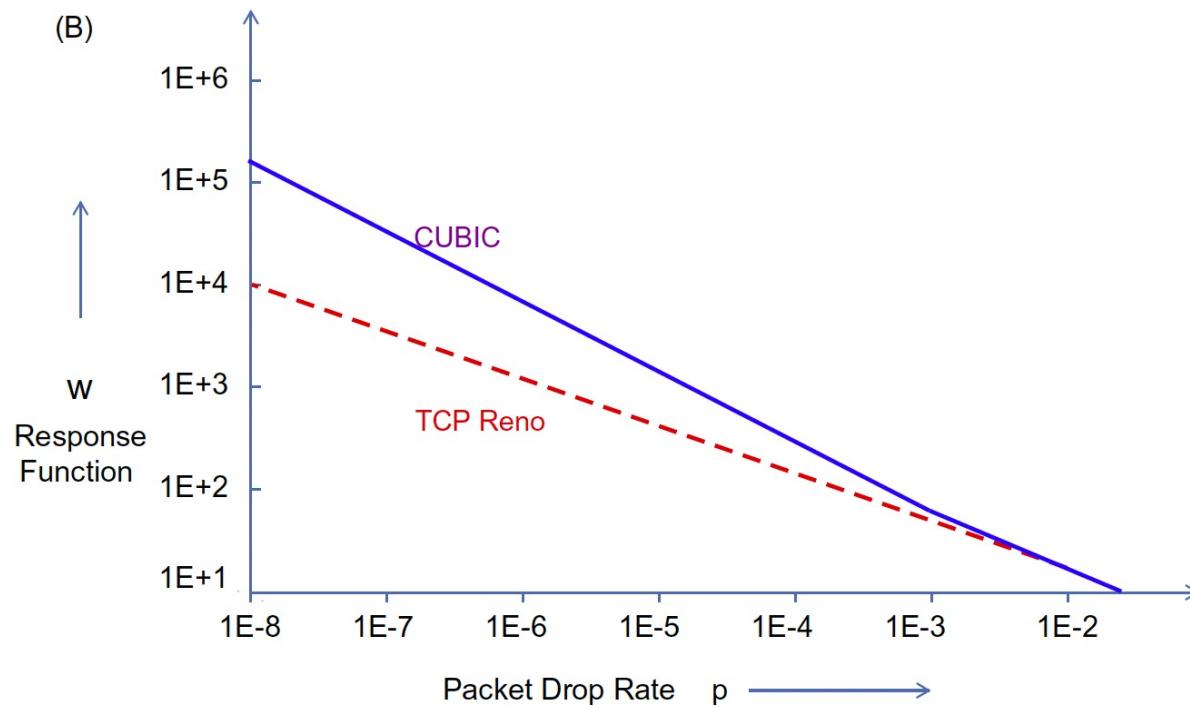
$$\log \bar{p} = \log \frac{\alpha(4-\beta)}{4\beta} + 3 \log T - 0.36.$$

Cubic Response Function is Piecewise Linear

# Response Function for $T = 10\text{ms}$



# Response Function for $T = 100\text{ms}$



The crossover point  $\bar{p}$  increases as  $T$  increases

# TCP CUBIC: Response Function

The switch between Cubic and Reno is a function of the round trip latency  $T$ !

Why is this a good property to have?

- ▶ In LAN environments in which  $T$  is small, the Response Functions for Cubic and Reno coincide for most of the range.
- ▶ For WAN environments with large  $T$ , the Cubic Response Function grows much faster.

This is an interesting contrast with the behavior of HSTCP, in which the switch between HSTCP and TCP Reno happens as a function of the window size  $W$  alone, irrespective of the round trip latency  $T$ .

# CUBIC vs Reno

The dependency of the Reno CUBIC cross-over point on T is a very attractive feature because most traffic, especially in enterprise environments, passes over networks with low latency, for whom regular TCP Reno works fine.

If a high-speed algorithm is used over these networks, we would like it to coexist with Reno without overwhelming it, and CUBIC satisfies this requirement very well.

Over high-speed long-distance networks with a large latency, on the other hand, TCP Reno is not able to make full use of the available bandwidth, but CUBIC naturally scales up its performance to do so.

# RTT Fairness for CUBIC

$$R = \sqrt[4]{\frac{\alpha(4 - \beta)}{4p^3T\beta}}$$

So that  $e = 0.25$ ,  $d = 0.75$

$$R_{avg} = \frac{h}{T^e p^d}$$

$$\frac{R_1}{R_2} = \left(\frac{T_2}{T_1}\right)^{\frac{e}{1-d}}$$

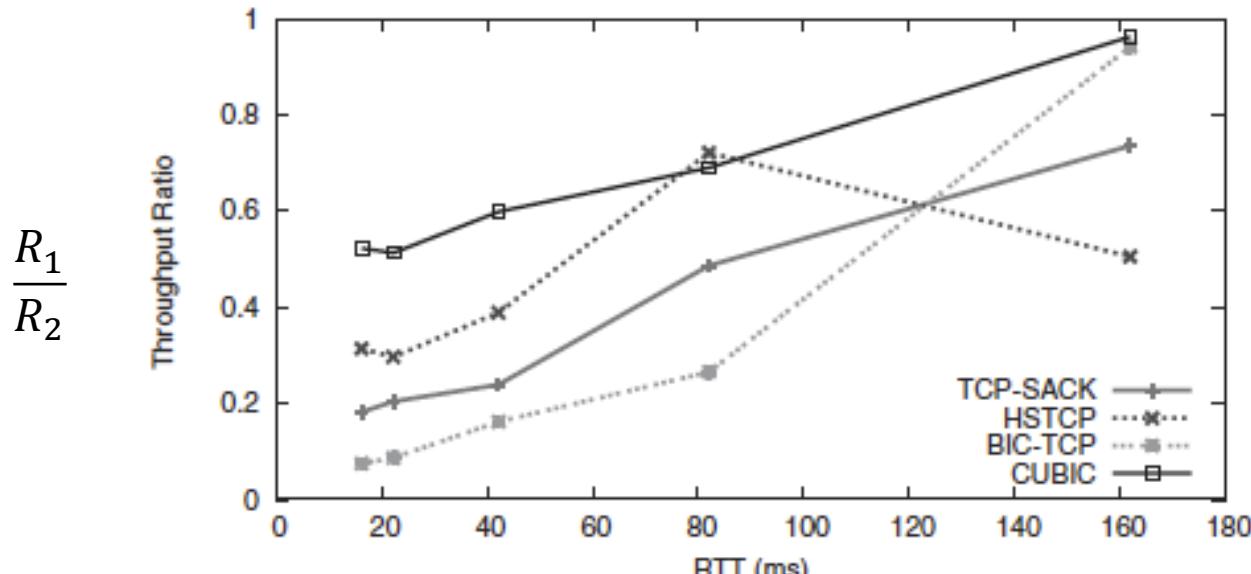
Hence

$$\frac{R_1}{R_2} = \left(\frac{T_2}{T_1}\right)^{\frac{0.25}{1-0.75}} = \left(\frac{T_2}{T_1}\right).$$

Hence, TCP CUBIC has better RTT fairness than any of the other high-speed protocol and even TCP Reno.

This is because it does not use ACK clocking to time its packet transmissions.

# RTT Fairness



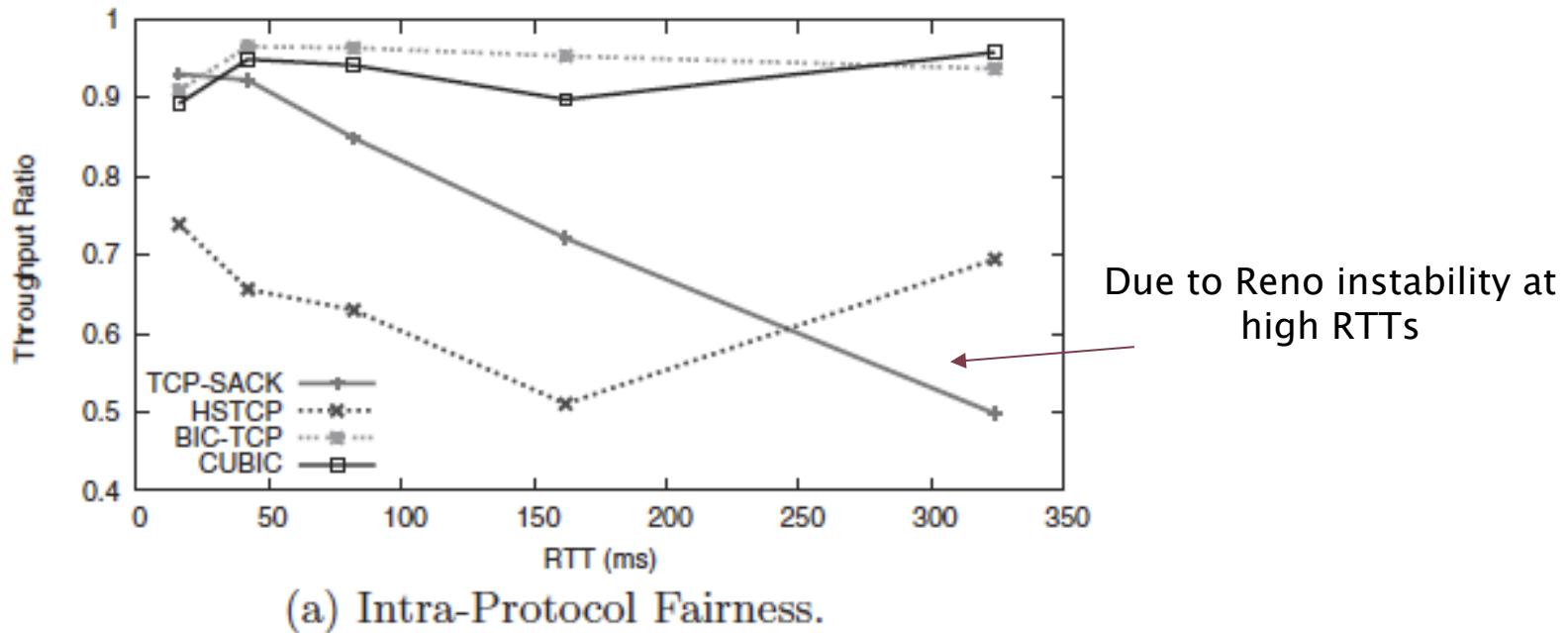
$T_1 > T_2$

$T_1$  is fixed at 162 ms

(a) Inter-RTT Fairness.

- Figure shows the ratio between the throughputs of two flows of CUBIC, BIC, HSTCP, and TCP SACK, for a bottleneck link capacity of 400 mbps, and with one of the flows with a fixed RTT of 162 ms. The RTT of the other flow is varied between 16 ms and 162 ms.
- This clearly shows that CUBIC has better inter-RTT fairness than the other protocols.

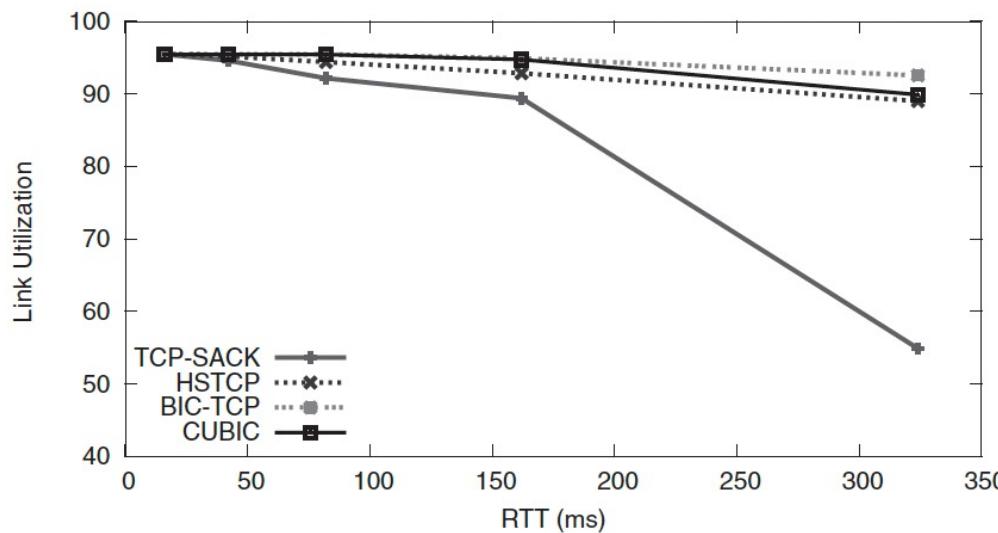
# Intra-Protocol Fairness as a Function of RTT



- Measurement of the intra-protocol fairness between two flows of a protocol with the same RTT. Throughput ratio between these two flows is used for representing the intra-protocol fairness.
- This metric represents a degree of bandwidth shares between two flows of the same protocol. For this experiment, RTT was varied between 16 ms and 324 ms

# System Stability with BIC and CUBIC

- ▶ Both BIC and CUBIC show stability with high RTTs and link capacities, even with plain tail drop routers. Why is that?
- ▶ There has been no formal proof for this, but the Averaging Principle offers a clue as to why this is the case.
- ▶ Both protocols reduce the rate of window increase as the link approaches saturation, and this seems to be the crucial property.



Link utilization at the bottleneck node with varying RTT

# **CTCP**

# **Compound TCP**

# The Return of Delay based Congestion Control

- ▶ All the algorithms we have considered use packet loss as their main indicator of network congestion, with the exception of TCP Vegas.
- ▶ The problem with TCP Vegas was that it didn't do very well when competing with loss based connections.
- ▶ In the last 10–15 years there have been several proposals that aim to introduce some aspect of delay based congestion control, while avoiding the pitfalls of TCP Vegas. Strategies include:
  - Combine loss based and delay based congestion indicators (example Compound TCP)
  - Limit queueing delay without having to measure end to end latency (example BBR)
  - Allow for larger changes in window size in response to changing delay (example TCP FAST)
  - Use optimization based congestion control in which delay is included as one of the optimization components (example COPA)
  - React to delay gradient rather than just delay (example TIMELY)

# CTCP: Mixes Loss and Delay Based Congestion Indicators

Main Idea:

- ▶ In order to avoid overloading the network, the system should keep track of congestion using the TCP Vegas queue size estimator and use an aggressive window increase rule only when the estimated queue size is less than some threshold.
- ▶ When the estimated queue size exceeds the threshold, then CTCP switches to a less aggressive window increment rule, which approaches the one used by TCP Reno.

# CTCP: Window Control Rule

- ▶ CTCP Window Size is decomposed into two components:

$$W(t) = W_c(t) + W_d(t)$$

- ▶  $W_c(t)$  is the part of the window that reacts to **loss-based congestion signals** and changes its size according to the rules used by TCP Reno.
- ▶  $W_d(t)$ , on the other hand, reacts to a **delay-based congestion signal** and uses a new set of rules for changing its window size.
- ▶ Specifically,  $W_d(t)$  has a rapid window increase rule when the network is sensed to be underutilized and gracefully reduces its size when the bottleneck queue builds up.
- ▶ Hence the delay indicator is not being used to control latency (as in Vegas) but to boost throughput when the link is under utilized, and conversely to default to Reno when the link becomes congested.

# CTCP: Definitions

$W(t)$ : CTCP Window size at time t

$W_c(t)$ : Congestion component of the CTCP window size at time t

$W_d(t)$ : Delay component of the CTCP window size at time t

T: Base value of the round trip latency

$T_s(t)$ : Smoothed estimate of the round trip latency at time t

$R_E(t)$ : Expected estimate of the CTCP throughput at time t

$R(t)$ : Observed CTCP throughput at time t

$$R(t) = \frac{W(t)}{T_s(t)}$$

$\Theta(t)$ : Delay-based congestion indicator at time t

$$\theta(t) = (R_E(t) - R(t))T$$

$\gamma$ : Delay threshold, such that the system is considered to be congested if  
 $\theta(t) \geq \gamma$

Note that  $\theta(t) = \frac{W(t)}{T_s(t)}(T_s(t) - T)$

which by Little's law equals the number of queued packets in the network

# CTCP Window Size Evolution

CTCP specifies that the congestion window should evolve according to the following equations on a per RTT basis:

$W \leftarrow W + \alpha W^k$ , when there are no packet losses or queuing delays and

$W \leftarrow (1 - \beta)W$  on one or more packet losses during a RTT

Given that the standard TCP congestion window evolves according to (per RTT):

$W_c \leftarrow W_c + 1$  with no loss, and

$W_c \leftarrow \frac{W_c}{2}$  with loss

$$W(t) = W_c(t) + W_d(t)$$

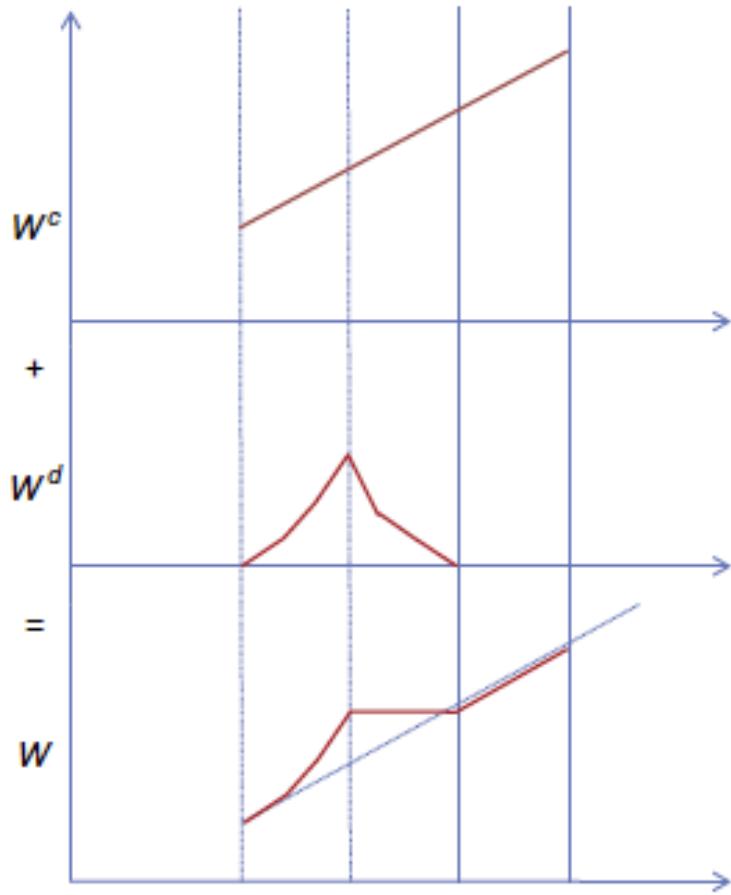
It follows that:

$$W_d \leftarrow W_d + [\alpha W^k - 1]^+ \quad \text{if} \quad \theta \leq \gamma$$

$$W_d \leftarrow [W_d - \varsigma \theta]^+ \quad \text{if} \quad \theta \geq \gamma \quad \xleftarrow{\text{Delay caused decrease}}$$

$$W_d \leftarrow \left[ (1 - \beta)W_d - \frac{W_c}{2} \right]^+ \quad \text{with loss}$$

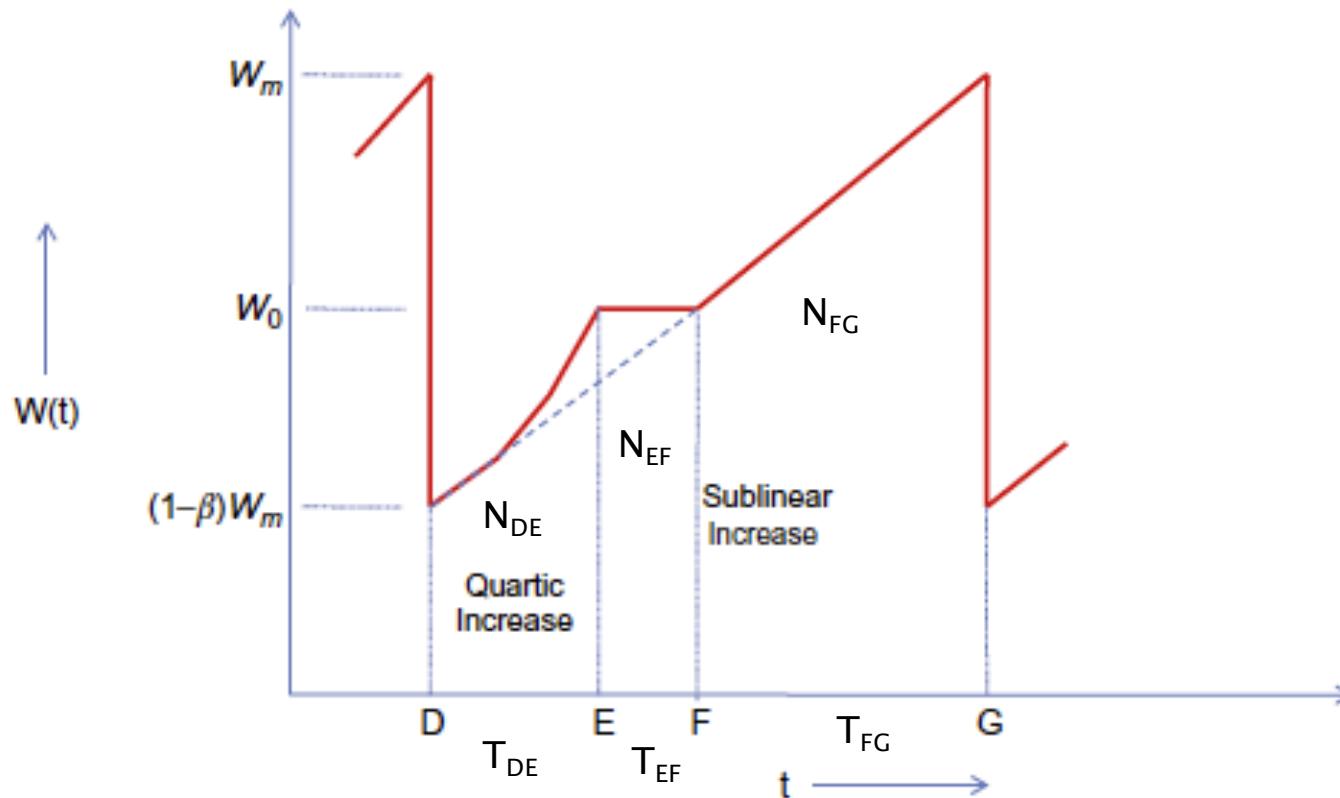
# Evolution of the Individual Window Sizes in CTCP



- When the network queuing backlog is smaller than  $\gamma$ , the delay-based congestion window  $W_d$  grows rapidly.
- When the congestion level reaches  $\gamma$ , the congestion window continues to increase at the rate of one packet per round trip time, and the delay window starts to ramp down.
- If the rate of increase of the congestion window  $W_c$  equals the rate of decrease of the delay window  $W_d$ , then the resulting CTCP window  $W$  stays constant at  $W_0$  until the size of the congestion window  $W_c$  exceeds  $W_0$ .
- At that point, the CTCP window  $W$  resumes its increase at the same linear rate as the congestion window, and the delay window dwindle to zero

# CTCP Throughput Estimate

$$R_{avg} = \frac{N_{DE} + N_{EF} + N_{FG}}{T_{DE} + T_{EF} + T_{FG}}$$



$W_0$  estimate

Since  $\gamma = (T_s - T) \frac{W_0}{T_s}$  It follows that  $W_0 = \gamma \frac{T_s}{T_s - T}$

# CTCP Analysis: Phase 1

Phase 1: Interval  $T_{DE}$  (Queueing delay is below threshold)

The CTCP window increases according to

$$\frac{dW}{dt} = \frac{\alpha W^k}{T_s}, \text{ so that}$$

$$\frac{W^{1-k}}{1-k} = \frac{\alpha t}{T_s}$$

Since the window increases by  $\alpha W^k$  every  $T_s$  seconds.

$$\text{For } k = 0.75 \quad W(t) = \left( \frac{\alpha t}{4T_s} \right)^4, \quad t_D \leq t \leq t_E$$

Shows rapid increase in window size during Phase 1

Since  $W$  increases from  $(1 - \beta)W_m$  to  $W_0$ , it follows that

$$T_{DE} = \frac{T_s}{\alpha(1-k)} [W_0^{1-k} - (1-\beta)^{1-k} W_m^{1-k}]$$

and the number of packets transmitted during this interval is given by (with  $T_{DE} = t_D - t_E$ )

$$\begin{aligned} N_{DE} &= \int_{t_d}^{t_E} \frac{W}{T_s} dt = \left[ \frac{\alpha(1-k)}{T_s} \right]^{\frac{1}{1-k}} \frac{1}{T_s} \int_{t_d}^{t_E} t^{\frac{1}{1-k}} dt \\ &= \frac{1}{\alpha(2-k)} [W_0^{2-k} - (1-\beta)^{2-k} W_m^{2-k}] \end{aligned}$$

# CTCP Analysis: Phase 2

## Phase 2: Interval $T_{EF}$ (Queueing delay is at threshold)

Note that the interval  $T_{EF}$  comes to an end when the size of the congestion window  $W_c$  becomes equal to  $W_0$ . Because  $\frac{dW^c}{dt} = \frac{1}{T_s}$ , it follows that

$$T_{DF} = [W_0 - (1 - \beta)W_m]T_s \quad \text{and}$$

$$T_{EF} = T_{DF} - T_{DE} = [W_0 - (1 - \beta)W_m]T_s - \frac{T_s}{\alpha(1 - k)} [W_0^{1-k} - (1 - \beta)^{1-k} W_m^{1-k}]$$

$$\text{and } N_{EF} = W_0 T_{EF}.$$

# CTCP Analysis: Phase 3

## Phase 3: Interval $T_{FG}$ (Queueing delay is above threshold)

the CTCP window increases linearly as for TCP Reno, so that

$$\frac{dW}{dt} = \frac{1}{T_s} \quad \text{so that} \quad \Delta W = \frac{\Delta t}{T_s}. \quad \text{and} \quad T_{FG} = (W_m - W_0)T_s$$

$$N_{FG} = \frac{1}{T_s} \int_{T_F}^{T_G} W(t) dt = \frac{1}{T_s} [W_0 T_{FG} + \frac{1}{2}(W_m - W_0)T_{FG}] = \frac{W_m + W_0}{2T_s} T_{FG} = \frac{W_m^2 - W_0^2}{2}$$

# CTCP Analysis

Putting it all together

$$\frac{1}{p} = N_{DE} + N_{EF} + N_{FG} \quad \text{so that} \quad R_{avg} = \frac{1/p}{T_{DG}}$$

This can be numerically solved to obtain  $W_m$  for a given value of  $1/p$   
Substitute this back to obtain  $T_{DG}$

For the special case when the packet loss occurs in Phase 1:

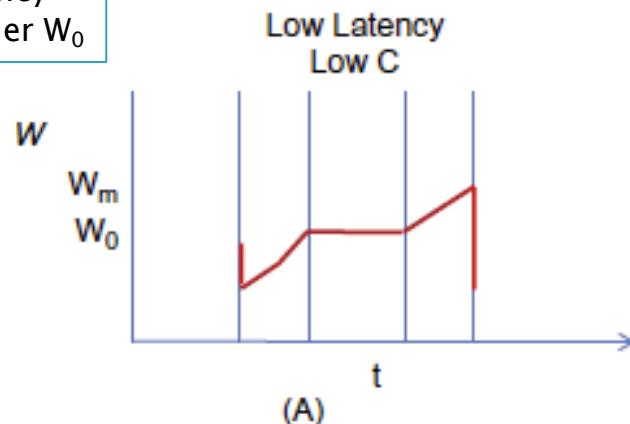
$$\begin{aligned} R_{avg} &= \frac{\frac{1}{p}\alpha(1-k)}{T_s W_m^{1-k}(1-(1-\beta)^{1-k})} \\ &= \left(\frac{\alpha}{p}\right)^{\frac{1}{2-k}} \frac{1}{T_s(1-(1-\beta)^{1-k})} \left[\frac{1-(1-\beta)^{2-k}}{2-k}\right]^{\frac{1-k}{2-k}} \end{aligned}$$

$$\boxed{\begin{aligned} R_{avg} &= \frac{h}{T^e p^d} \\ \frac{R_1}{R_2} &= \left(\frac{T_2}{T_1}\right)^{\frac{e}{1-d}} \end{aligned}}$$

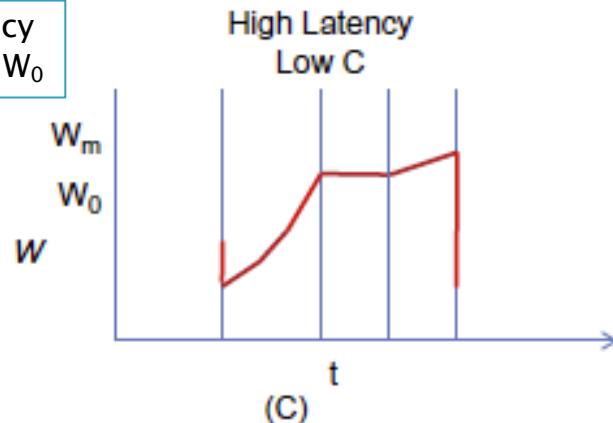
For  $k = 0.75$ , it follows that  $d = 0.8$  and  $1/(1-d) = 5$

# Variation of TCP Window with Varying Latency and Link Capacities

Low Latency  
=> Smaller  $W_0$

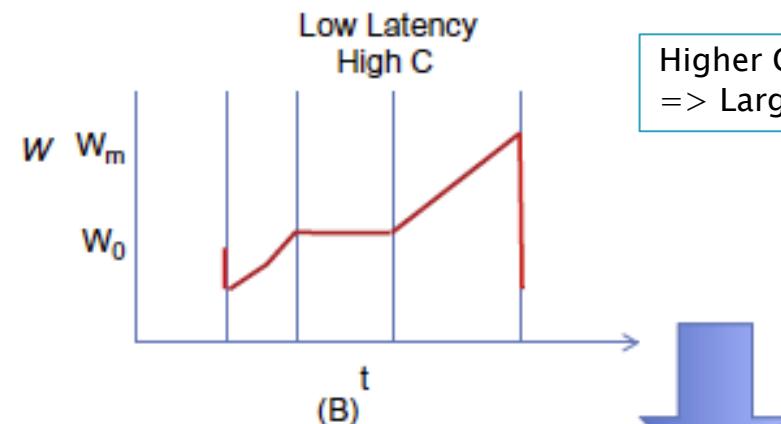


High Latency  
=> Larger  $W_0$

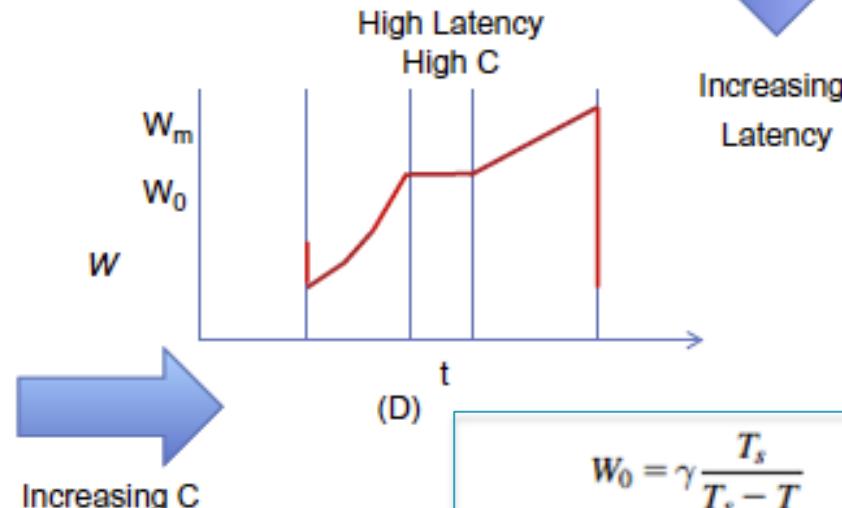


$$T_{FG} = (W_m - W_0)T_s$$

Higher Capacity  
=> Larger  $W_m$



Increasing  
Latency



Increasing C

$$W_0 = \gamma \frac{T_s}{T_s - T}$$

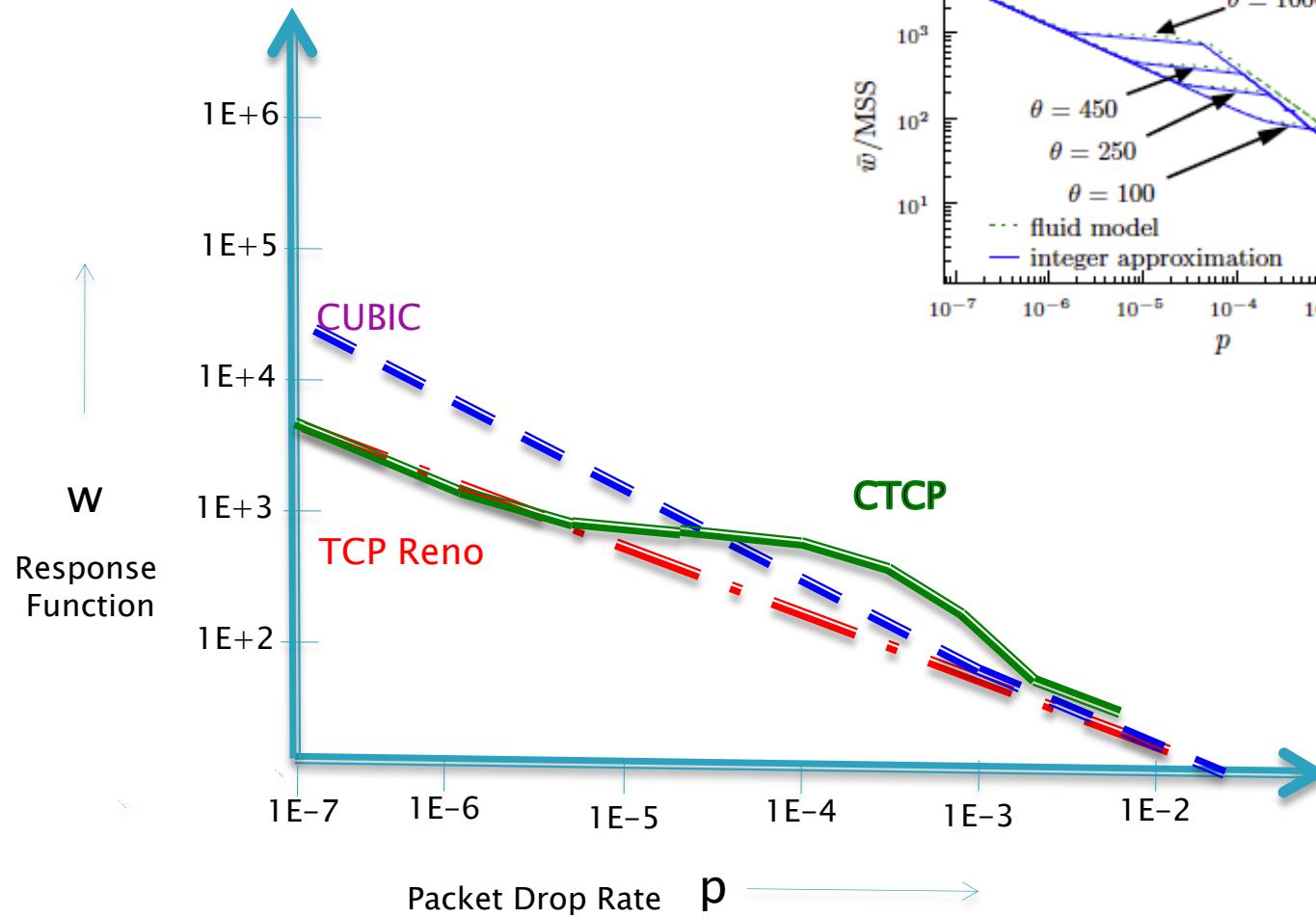
Note:  $W_0$  does not depend on C

$$W_m \approx CT$$

# Variation of TCP Window with Varying Latency and Link Capacities

- ▶ As link speed  $C$  increases, for a fixed value of end-to-end latency, then  $W_m$  increases, and as a result, the number of packets transmitted in the linear portion of window increases. This can also be seen by the formula for  $N_{FG}$ . As a result, at higher link speeds, the CTCP throughput converges toward that for TCP Reno.
- ▶ As the end-to-end latency increases, then  $W_0$  increases, and the number of packets transmitted in the quartic region of the CTCP window increases. This can also be seen by the formula for  $N_{DE}$ . As a result, an increase in latency causes the CTCP throughput to increase faster compared with that of Reno.
- ▶ This behavior is reminiscent with that of CUBIC because CUBIC's throughput also increased with respect to Reno as the end-to-end latency increases.
- ▶ However, the CTCP throughput will converge back to that for Reno as the link speed becomes sufficiently large, while the CUBIC throughput keeps diverging.

# Response Functions for CTCP, CUBIC and Reno



# Misc Algorithms

# TCP Africa

- ▶ TCP Africa is a dual state algorithm; the congestion window is updated differently in the two operation modes.
- ▶ Specifically the algorithm switches between the “slow” mode state in which the congestion window is updated according to Reno algorithm, and the “fast” mode state in which the congestion window is updated according to HSTCP increase rule.
- ▶ Switching between states is governed by the number of queued packets in the bottleneck buffer, inferred through a delay-based approach.
- ▶ Hence, TCP Africa is aggressive when the pipe is not full and it behaves like Reno when the full link utilization is achieved.
- ▶ Similar to CTCP (actually TCP Africa came first), with the difference that TCP Africa switches between algorithms, while CTCP keeps two different windows.

# Yeah TCP

$$\theta(t) = \frac{W(t)}{T_s(t)}(T_s(t) - T)$$

- ▶ Yeah TCP also has two modes: “Fast” and “Slow”, like Africa TCP and CTCP.
- ▶ During the “Fast” mode, YeAH-TCP increments the congestion window according to an aggressive rule. In the “Slow” mode, it acts as Reno TCP.
- ▶ The state is decided according to the estimated number of packets in the bottleneck queue, which is estimated as

$$\theta(t) = \frac{W(t)}{T_{min}(t)}(T_{min}(t) - T)$$

Where  $T_{min}(t)$  is the minimum RTT estimated in the current data window of  $cwnd$  packets (this is a better estimate of persistent congestion at the node).

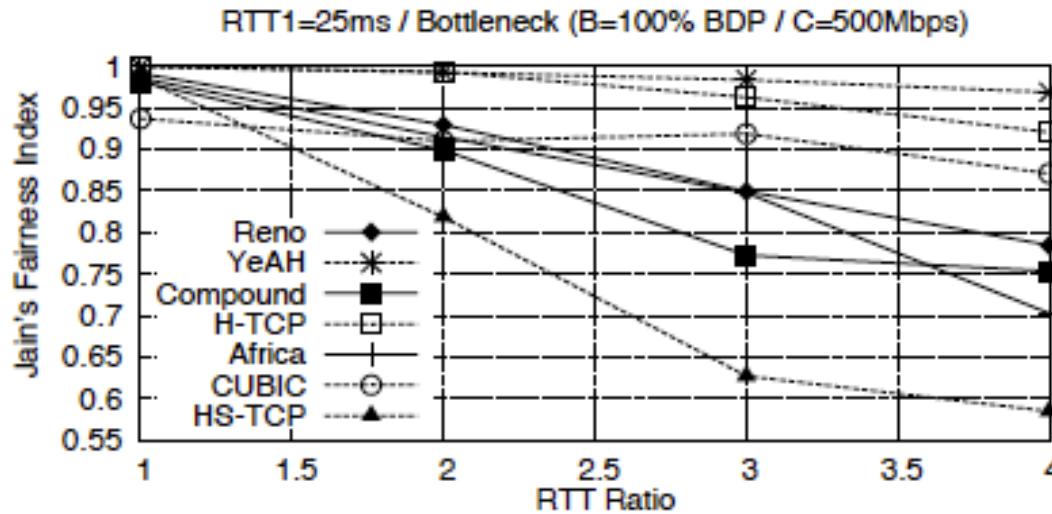
- ▶ Define  $L = \frac{T_{min}(t)}{T}$ , which is a measure of the network congestion level.
- ▶ If  $\theta < \theta_{max}$  and  $L < 1/\varphi$ , the algorithm is in the “Fast” mode, otherwise it is in the “Slow” mode.
- ▶ Whenever  $\theta > \theta_{max}$ , the congestion window is diminished by  $\theta$  and ssthresh set to  $cwnd/2$ . Note that this action, called precautionary decongestion, is done even in the absence of a packet loss event.

# Yeah TCP

- ▶ The precautionary decongestion is optimal only when the flows that implement it do not compete with “greedy” sources, such as Reno. When competing with “greedy” flows, the precautionary decongestion makes the conservative flow lose capacity, since it releases bandwidth to the greedy sources.
- ▶ To avoid unfair competition with legacy flows, YeAH-TCP implements a mechanism to detect if it is competing with “greedy” sources:
  - When  $\theta > \theta_{max}$  YeAH-TCP attempts to remove packets from the queue. If the queuing delay increases further because Reno flows are “greedily” filling up the buffer, then YeAH-TCP will stay hardly ever in “Fast” mode state and frequently in “Slow” mode.
  - On the other hand, when competing with non greedy flows, the YeAH algorithm will cause a state change from “Fast” to “Slow” whenever buffer content builds up above  $\theta_{max}$  and back as soon as the precautionary decongestion becomes effective.
- ▶ On detecting packet loss, the window decrease rule used is similar to that of Westwood, i.e., cwnd is decreased to the BDP rather than by half as in Reno. This helps sustain performance in high loss environments.

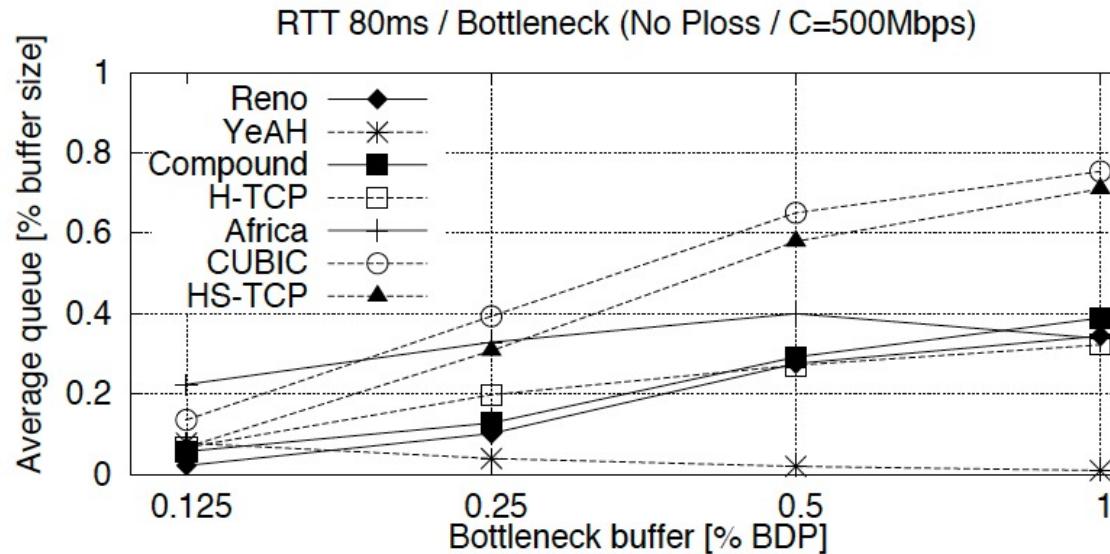
This is an example of a mechanism that allows delay based algorithms to compete fairly with loss based algorithms

# RTT Fairness



YeAH TCP is RTT fair, because every flow attempts to keep in the bottleneck buffer a fixed number of packets independently of RTT thus every flow attempts to share the bottleneck buffer fairly.

# Buffer Size



All the loss based algorithms experience a serious goodput degradation when operating with low buffer sizes.

# Further Reading

- ▶ Chapter 5, Sections 5.4–5.5 of Internet Congestion Control