SharkFest '18 Europe

To send or not to send?..

How TCP Congestion Control algorithms work

Vladimir Gerasimov

Packettrain.NET

About me

- In IT since 2005
- Working for Unitop (integration, distribution)
- Where to find me:
  - Twitter: @Packet_vlad
  - Q&A: https://ask.wireshark.org
  - Blog: packettrain.net
  - Social group https://vk.com/packettrain (Russian)

PCAPs

http://files.packettrain.net:8001/SF18/

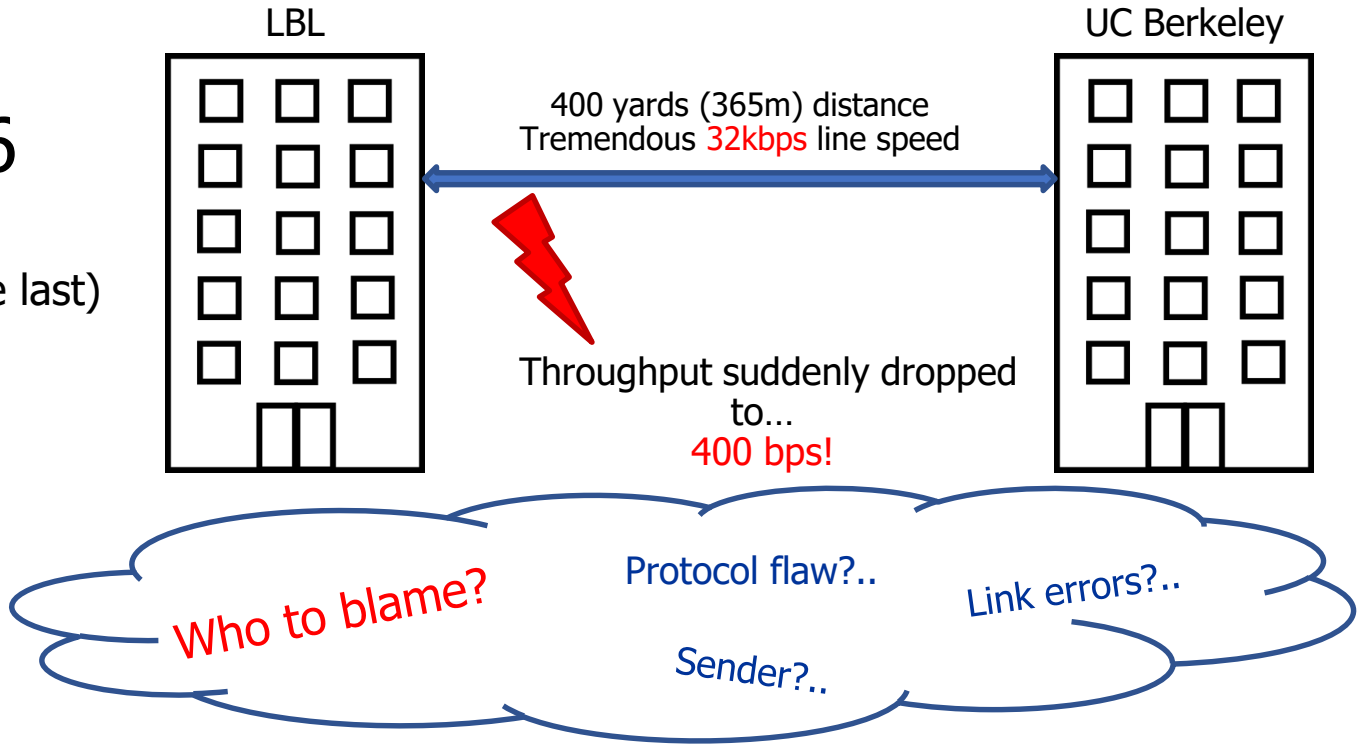
Login = password = sf18eu

(caution: size!!)

The beginning..

Oct 1986

The first (but not the last) occurrence.



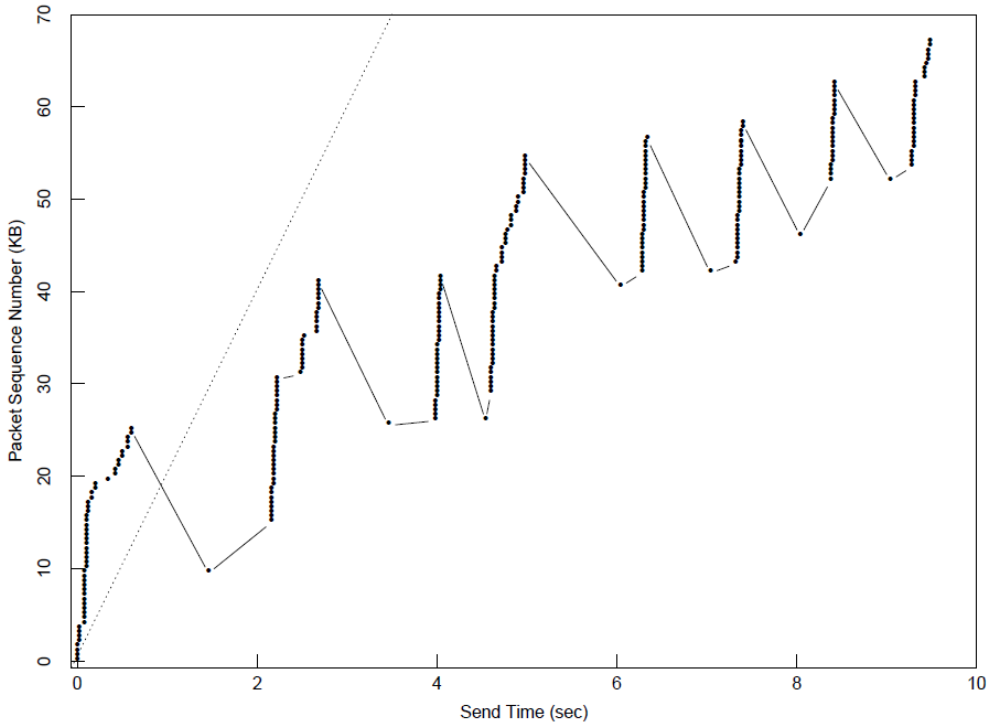
Let’s capture!

What was on the wire?

The sender (4.2 BSD) floods the link with tons of unnecessary retransmissions.

* because it sends on own full rate and have inaccurate RTO timer

* some chunks were retransmitted 5+ times!



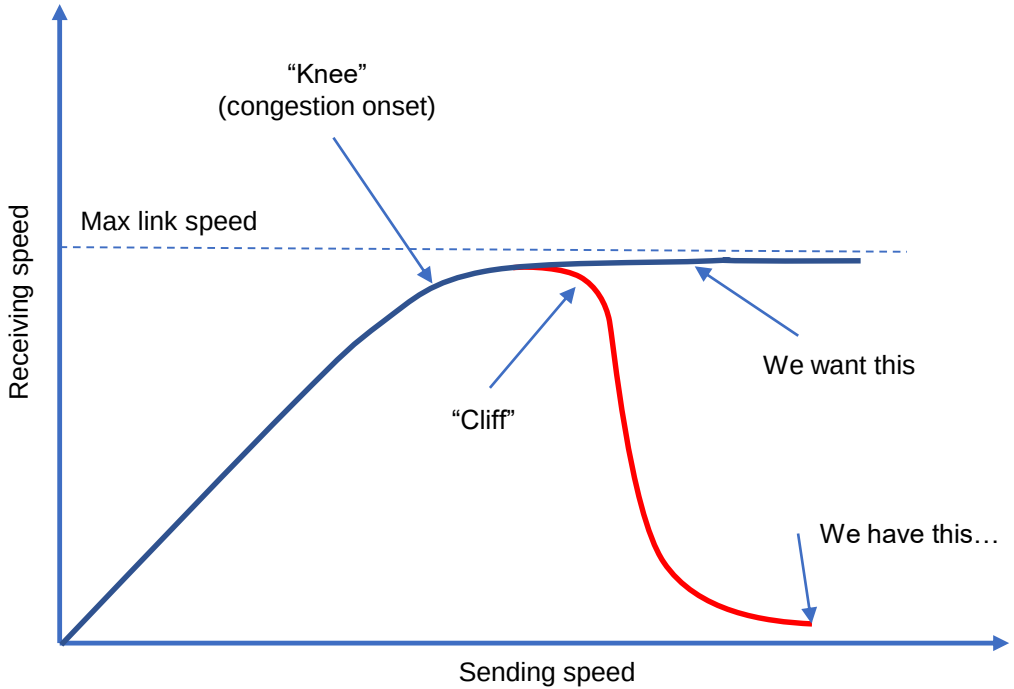
Congestion collapse

This is called "congestion collapse" – when goodput decreases by huge factor – up to 1000x!

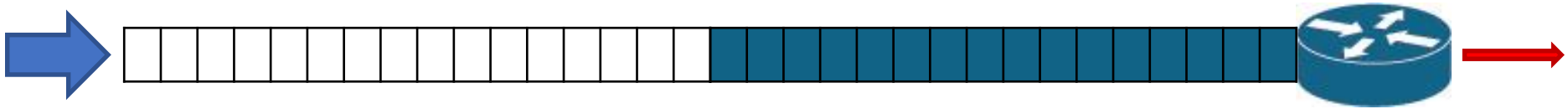
[Fact: it was predicted by Nagle in 1984 before it occured in real life]

- Bad news: it NEVER disappears without countermeasures.

- So a sender has to slow down its rate … or we just add more buffer to router?



Large buffers?

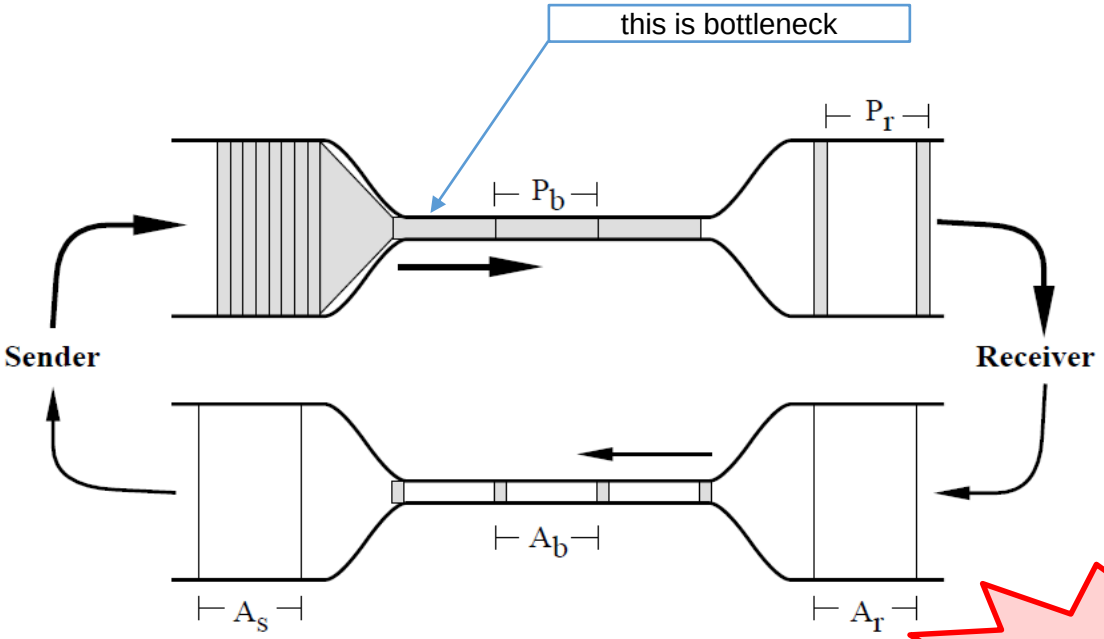


- Let’s never drop a packet!
- But… buffers could be large, but not endless.
- Good for absorbing bursts, but don’t help if long-term incoming rate > outgoing rate.
- Actually make things worse (high latency, “bufferbloat”) – so we don’t want to have even to have endless buffers if we could.

TCP Self-clocking

Equilibrium state is good, but… if you are the only one sender, you're already in it and there are no other variables.

Looks unrealistic.



TCP 'Self-clocking'

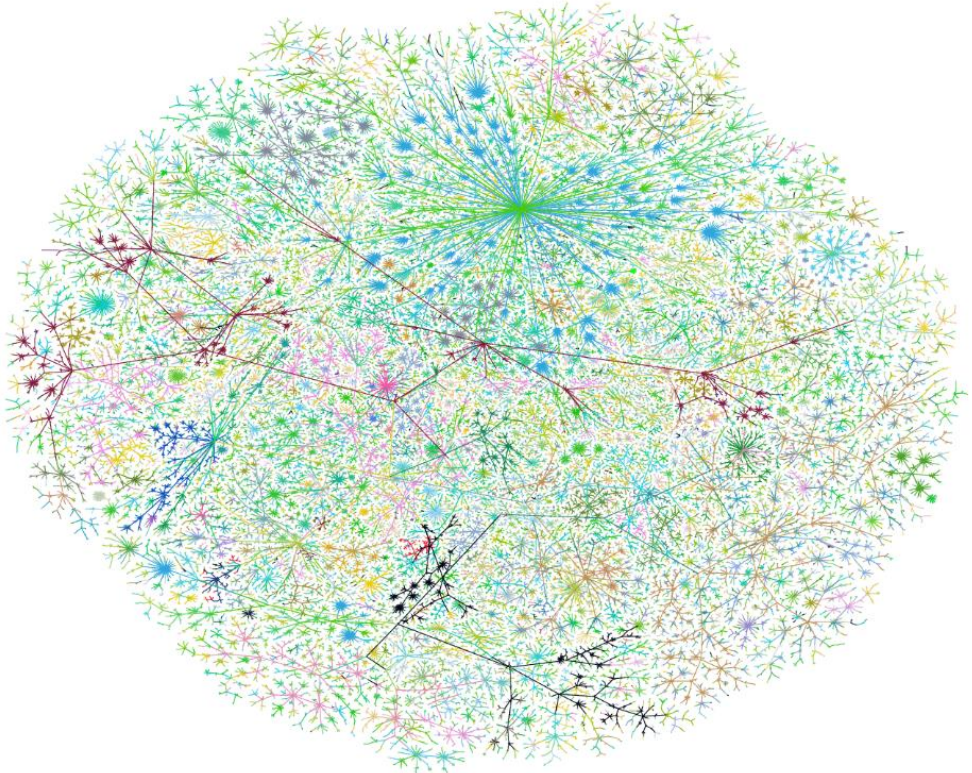
"Hand-weight diagram"

But in real world…

What about this topology?

- Random data transfers
- Random link parameters
- Unknown path

TCP has to do it’s job in this environment.



How to handle it?

Who must be responsible for handling?

Main decision made in [J88]:

“Smart endpoint, Dumb internet”

A sender (endpoint) has to slow down its transmission speed for some time giving up self-interest for the interest of the whole system.

Modified sender should be capable to handle congestion without any assistance from network nodes (though sometimes we’d like to have it… see later).

Focus on sender

Window-based or rate-based control??

Signaling (how do we know there is a congestion)??

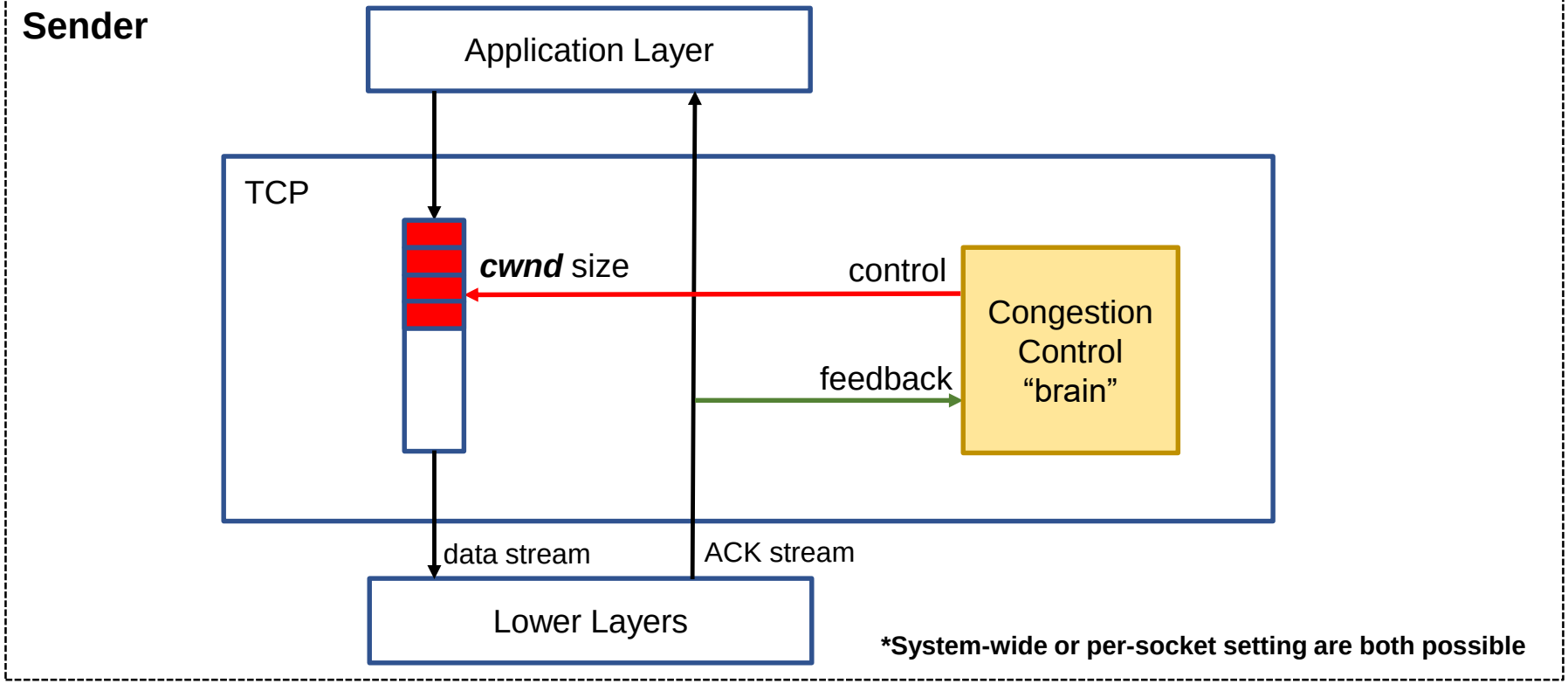
Priority to delay or utilization??

Increment rule if there is no congestion??

Decrement rule if there IS congestion??

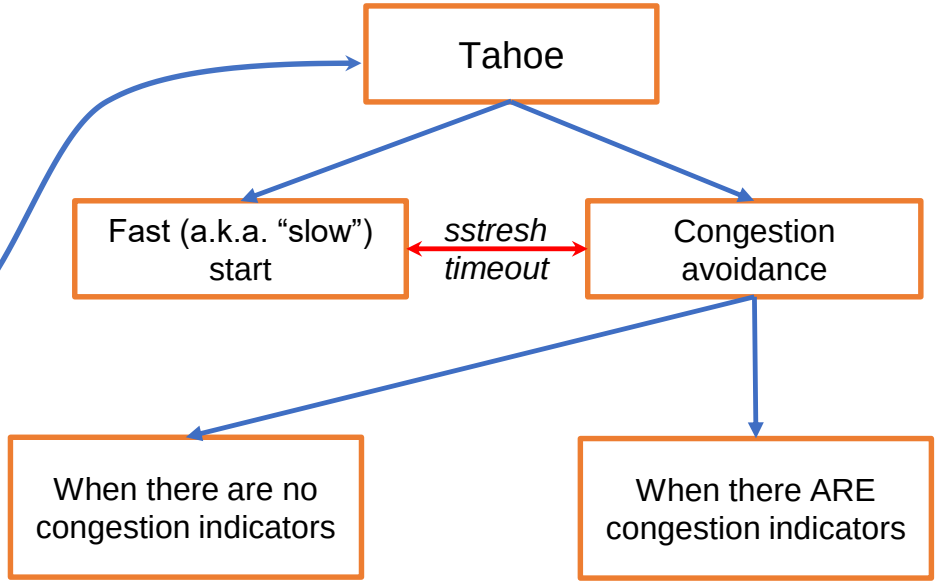
Fairness??

Design by [J88]



First solution by [J88]

1. Window-based control – hello, ***cwnd***! –
   W=min(*cwnd, awnd*), * where W – number of unacknowledged packets; we also assume there are no constraints in ***awnd*** in this session.
2. Feedback: packet loss as network congestion indicator
3. Action profile: several stages for different purpose each
4. RTO estimation enhancement
5. Fast retransmit mechanism
6. Focus on protection from collapse, not efficiency etc.



Tahoe – “slowfast kickstart”

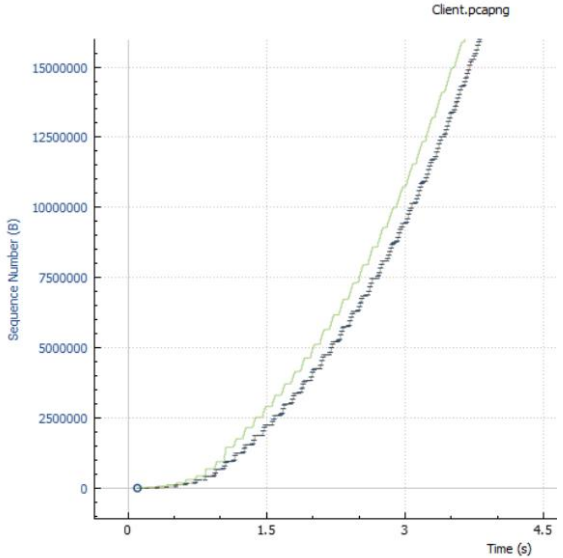
Two tasks:

1. **Establishing ACK clocking circuit (main one!)**
2. “Fast and dirty” probe for bandwidth.
3. Determining initial ***sstresh*** value for further use.

Operation:

1. Start from initial window IW.
2. For every ACKed SMSS increase ***cwnd*** by one SMSS.

*Refer to Christian’s session for details



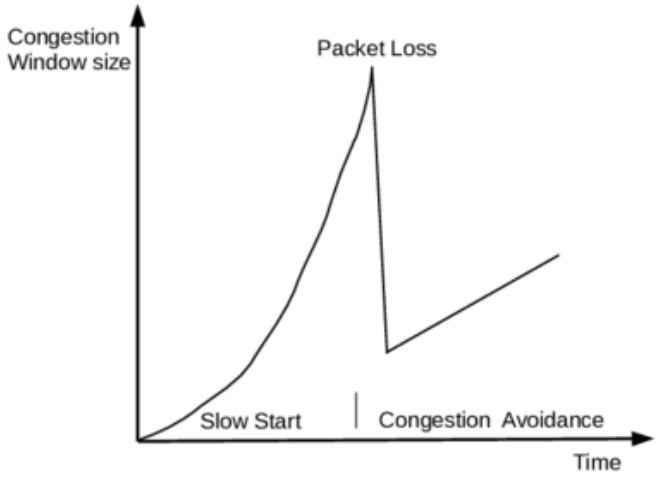
Fun fact

Initial window size nowadays usually equals 10 packets.
Refer to this link: https://iw.netray.io/stats.html
You can change it in Linux OS: #ip route change default via ip.address dev eth0 initcwnd 15
And in Windows OS: https://andydavies.me/blog/2011/11/21/increasing-the-tcp-initial-congestion-window-on-windows-2008-server-r2/

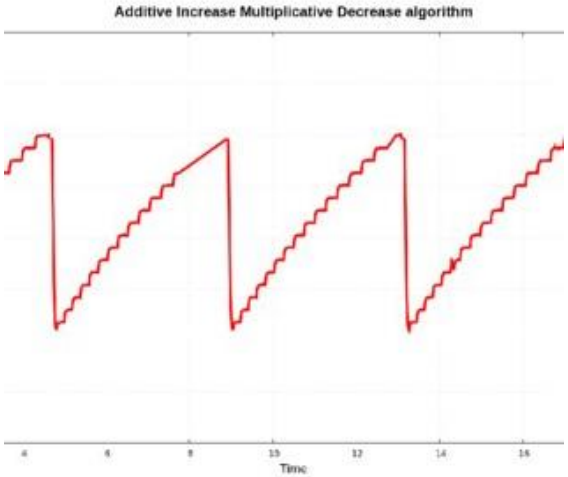
Fun fact

Fun fact

As ***cwnd*** increases/decreases at least by SMSS value, its real graph never contains inclined line segments, but only horizontal or vertical segments! So:



This is technically inaccurate! But totally OK to see the whole picture



In reality it is like that

Tahoe – Congestion avoidance

Core ideas:

1. Uses packet loss as a sign of congestion (feedback type/input).
2. Uses AIMD approach as action profile (control/output).

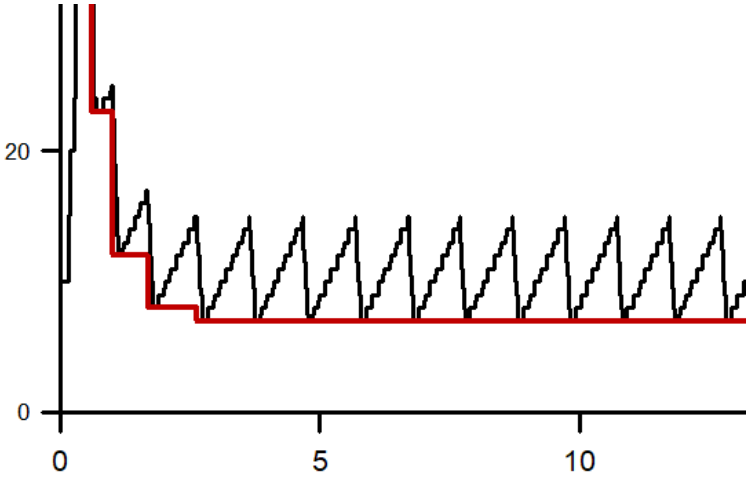
Has two modes (as any other algorithm):

1. With no observed signs of congestion.
2. With signs of congestion detected.

***cwnd* control rules:**

$$cwnd = \begin{cases} cwnd + a & if\ congestion\ is\ not\ detected \\ cwnd * b & if\ congestion\ is\ detected \end{cases}$$

For Tahoe, Reno $a = \frac{1}{cwnd}$; $b = 0.5$

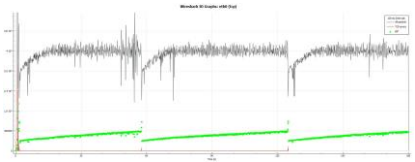


*Refer to Christian's session for more details

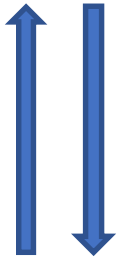
Fun Facts

Tahoe was created using “bottom-up” approach: packet-level rules first, macroscopic view (flow-level) second.

All subsequent CA algorithms (almost) were developed using the opposite “top-down” approach: flow-level first (this is what I want to achieve / address), packet-level rules second (this is how I achieve that).



Flow level



Packet level

$$cwnd = \begin{cases} cwnd + a & if\ congestion\ is\ not\ detected \\ cwnd * b & if\ congestion\ is\ detected \end{cases}$$

Fun Facts

Have you ever though about: why AIMD? Not MIMD, AIAD, MIAD?

The answer is simple – its behavior is the best among others.

- MIAD → not stable, tends to overload a network.
- MIMD → less fairness
- AIAD → less fairness
- AIMD → tends to achieve efficiency + fairness. “React quickly to bad news, react slowly to good news”.

Fun Facts

True or False?
- AIMD is an obsolete congestion control algorithm, nowadays we have better ones.

True or False?
- All congestion control algorithms since Tahoe react to packet loss.

True or False?
- ***cwnd*** in Reno in Congestion Avoidance phase grows as straight line until packet loss is detected.

Fun Facts

True or False?

- AIMD is an obsolete congestion control algorithm, nowadays we have better ones – Partially true!
- True: AIMD itself is not a congestion control algorithm, this is just an approach, pattern to behave while in congestion control stage. Many modern algorithms also use AIMD approach, but it's being eventually switched from. Remember also: AIMD ≠ Reno

True or False?

- All congestion control algorithms react to packet loss – FALSE!
- True: There many kinds of congestion control algorithms. Many of them indeed react to packet loss, but many others use different feedback type – delay. So, transition to congestion avoidance state could be done with no observed packet loss at all!

True or False?

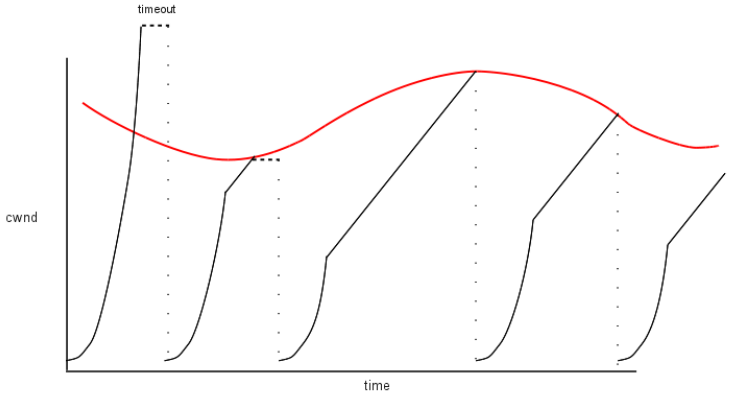
- ***cwnd*** in Reno (CA stage) grows as straight line until packet loss is detected – FALSE!
- True: In addition to "staircase-shape" although this line looks straight, it is not! The more ***cwnd*** size is, the less slope of this line is (refer to "Convergence" slide to see it!). Chances are we'll reach packet loss too early to spot this.

Let’s rate it!

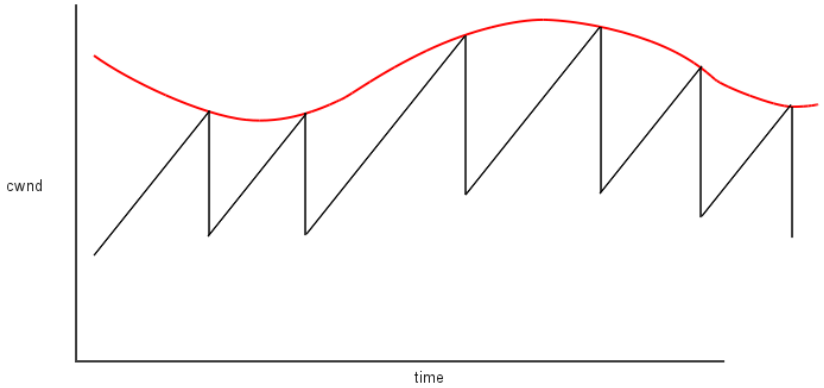
Well, how to decide which algorithm is better?

1. Efficiency (how full and steady is bottleneck utilization?)
2. Fairness (how we share bottleneck capacity?)
3. Convergence capability (how fast we approach equilibrium state? How much we oscillate later?)
4. “Collateral damage” (buffer overflow event rate, self-inflicted latency)

Efficiency



Tahoe: bad



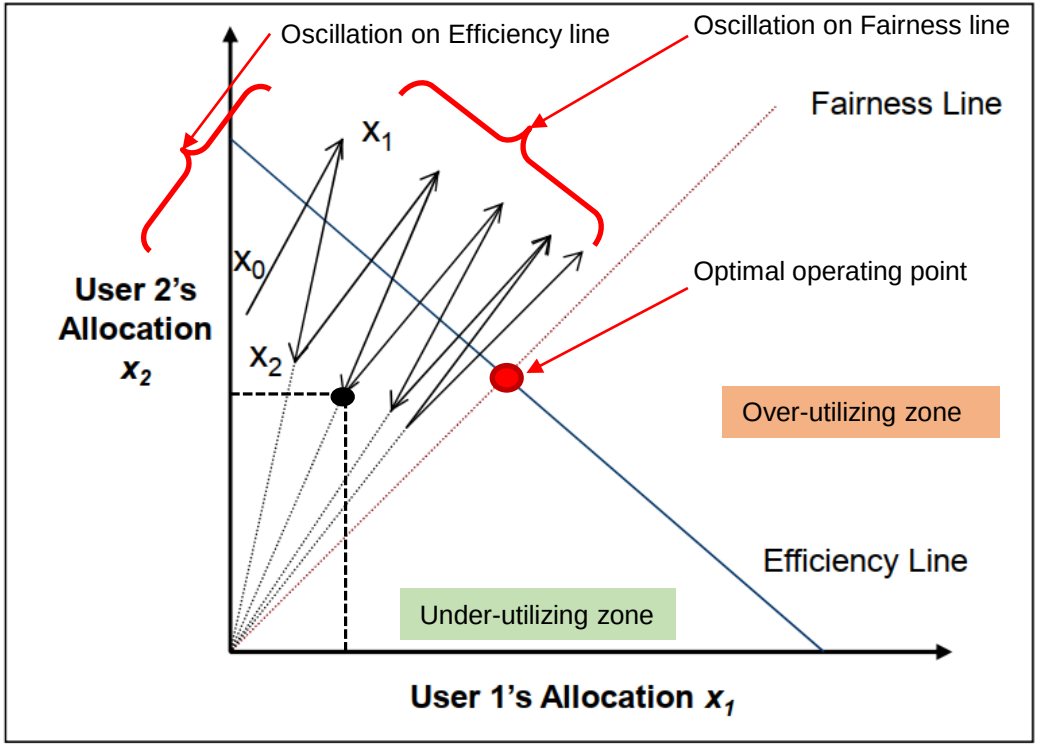
Reno: better, but not ideal

Fairness, convergence

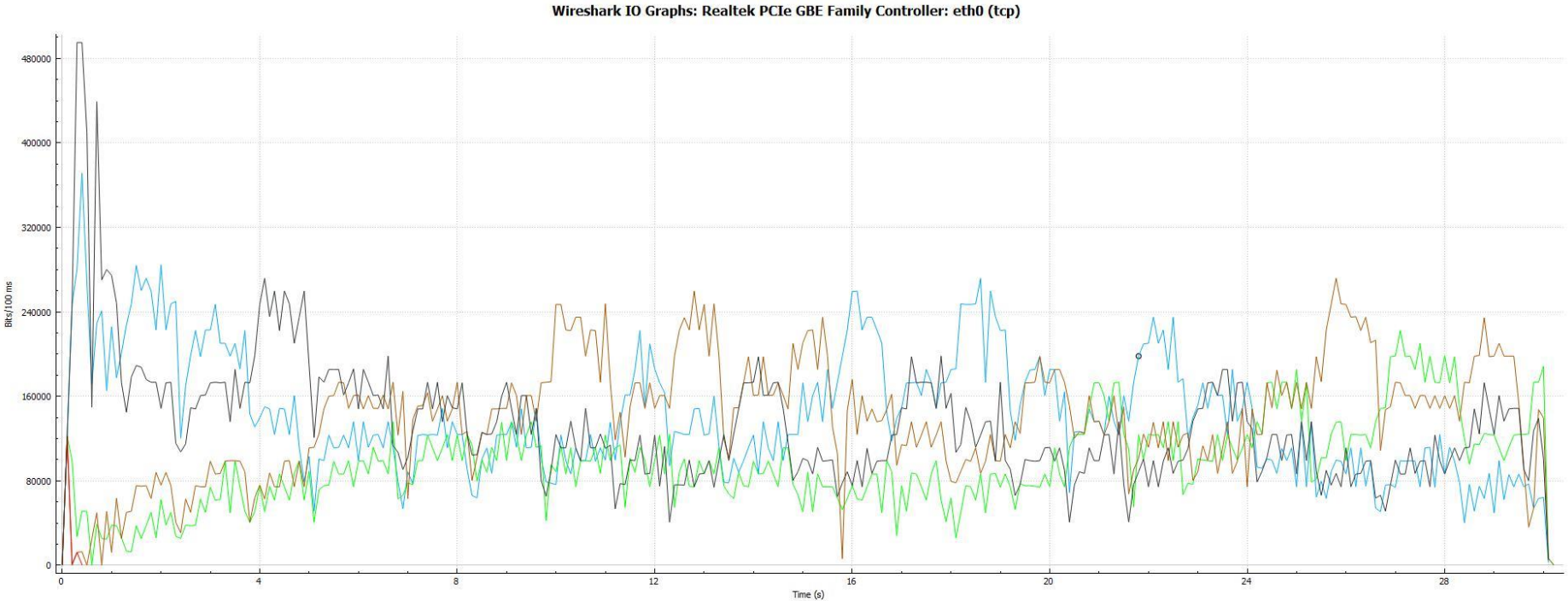
Introducing: **Phase graph**

Shows efficiency, fairness and convergence.

Here: an example for two senders.



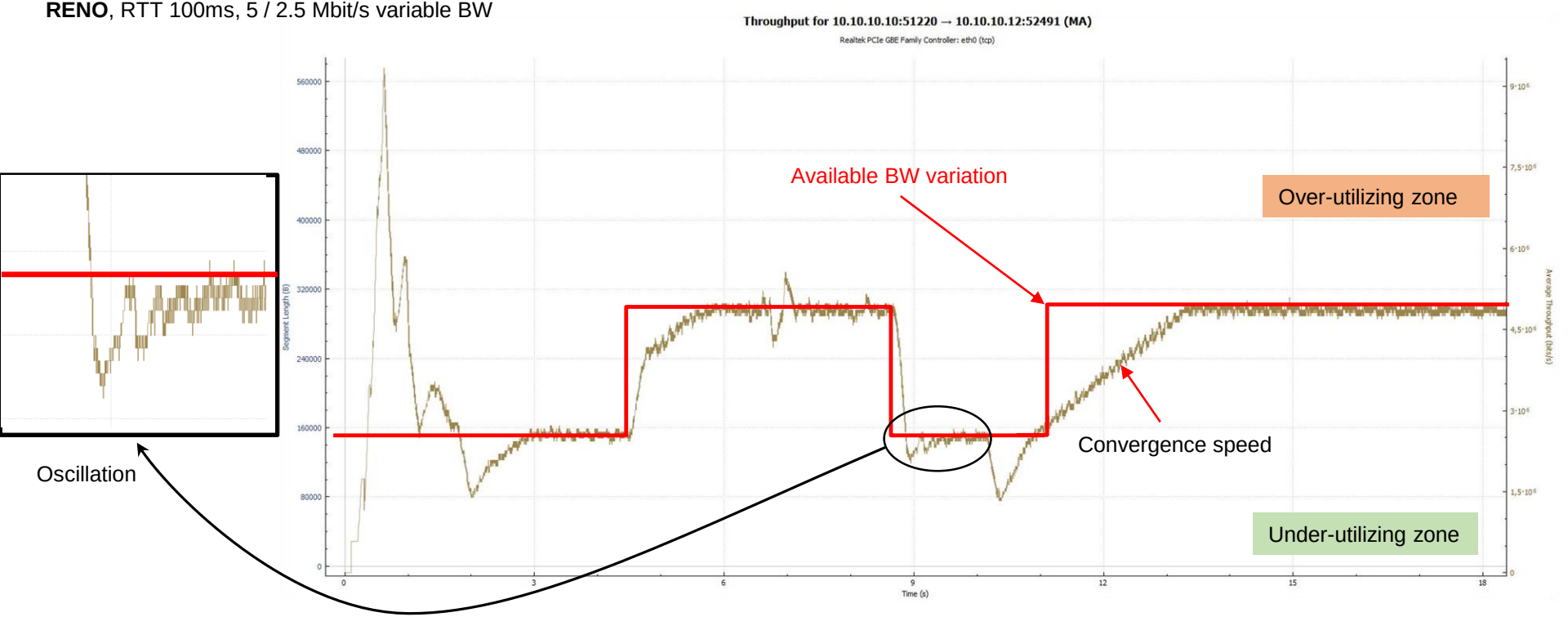
Fairness (5 streams Reno)



But what about non-TCP protocols? See later..

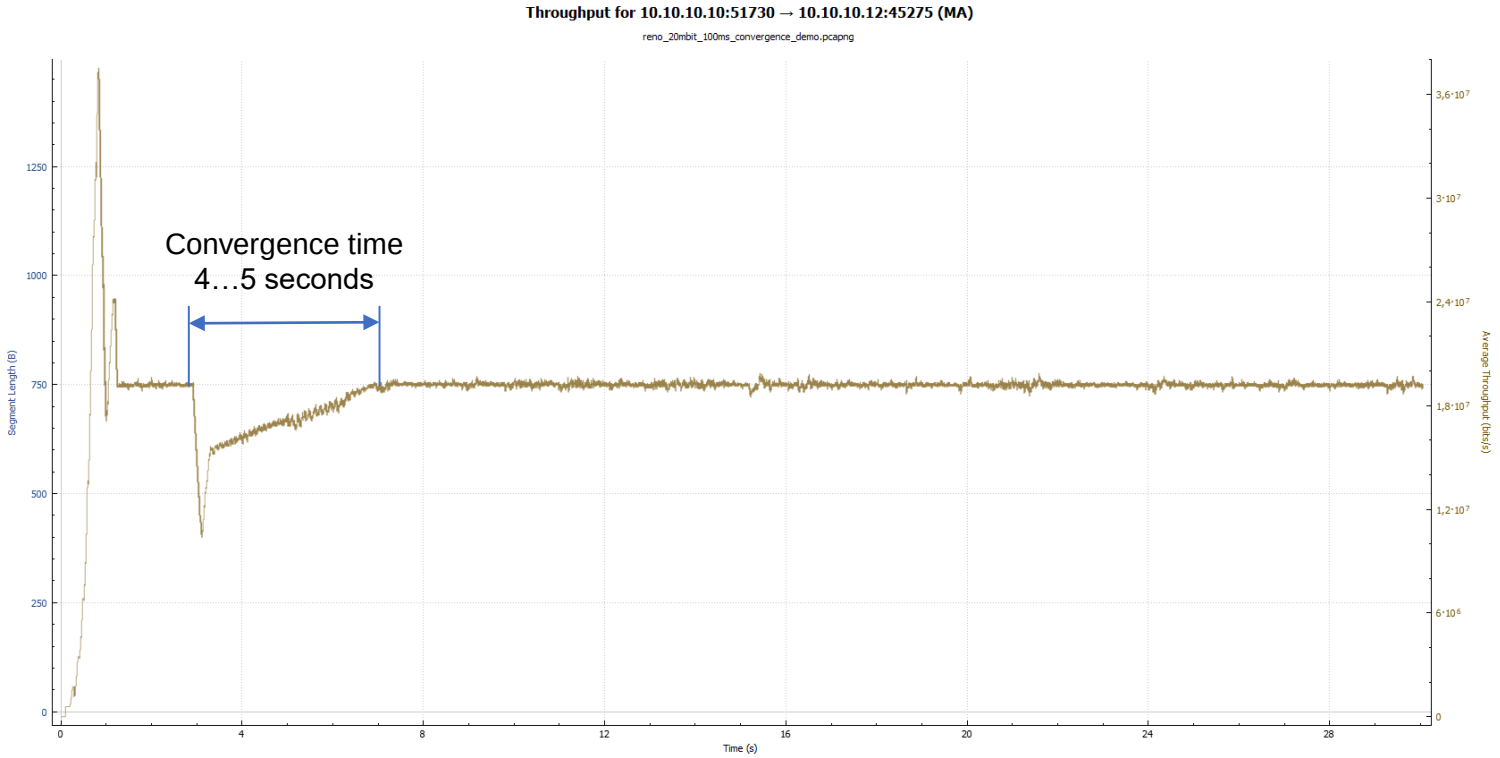
Convergence (1 stream)

RENO, RTT 100ms, 5 / 2.5 Mbit/s variable BW



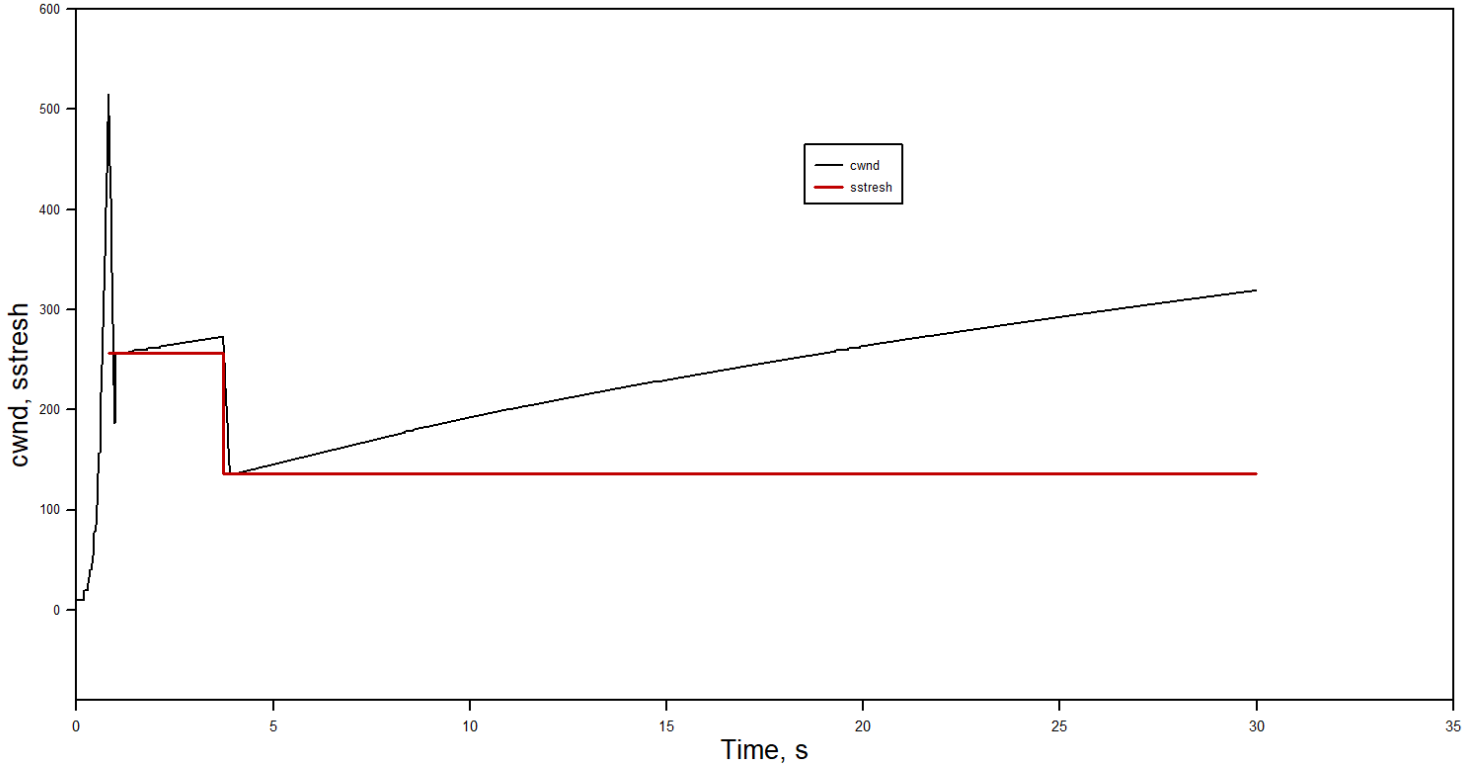
Convergence (1 stream)

RENO, RTT 100ms,
20 Mbit/s constant BW



Convergence (1 stream)

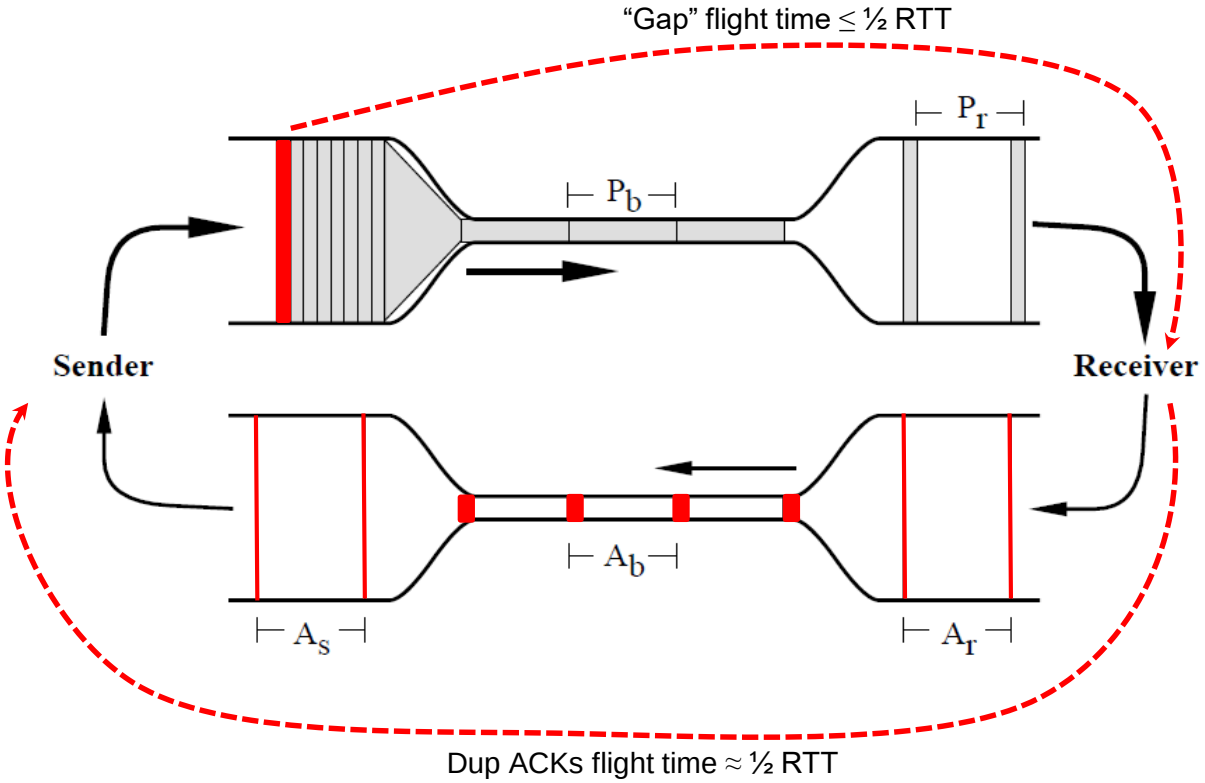
RENO, RTT 100ms,
20 Mbit/s constant BW



Complex challenges - 1

“Late news!”

- The sender will know about “data leaving network rate” not instantly, but only after ½ RTT.
- With packet drop at the beginning of a path – it’s getting worse.
- All this time the sender was sending more and more packets! Probably already starting to slide down the cliff.
- It is getting worse when RTT increases.



Complex challenges - 2

Non-TCP-compatible flows, unresponsive flows (“fairness” and “ TCP friendliness”).

- ✓ Non-TCP-compatible is a flow which reacts to congestion indicators differently, not like TCP.
- ✓ Unresponsive is a flow which does not react to congestion indicators at all.

“Fairness”	“TCP friendliness”
This is how TCP flows with the same CA algorithm share bottleneck BW with each other. A part of it is “RTT fairness”.	This is how non-TCP flows or TCP flows with different CA algorithms share bottleneck bandwidth.

2 possible solutions of this problem:

- ✓ TCP friendly rate control [RFC5348] concept – intentional rate limiting. *** a part of many modern CA algorithms.**
- ✓ Call for help (“network assisted congestion control”).

TCP friendly rate control

Core idea: create an equation for T (sending rate, packets/RTT) with argument *p* (packet loss coefficient).

$$T=f(p)$$

For standard TCP (Reno) the equation is:

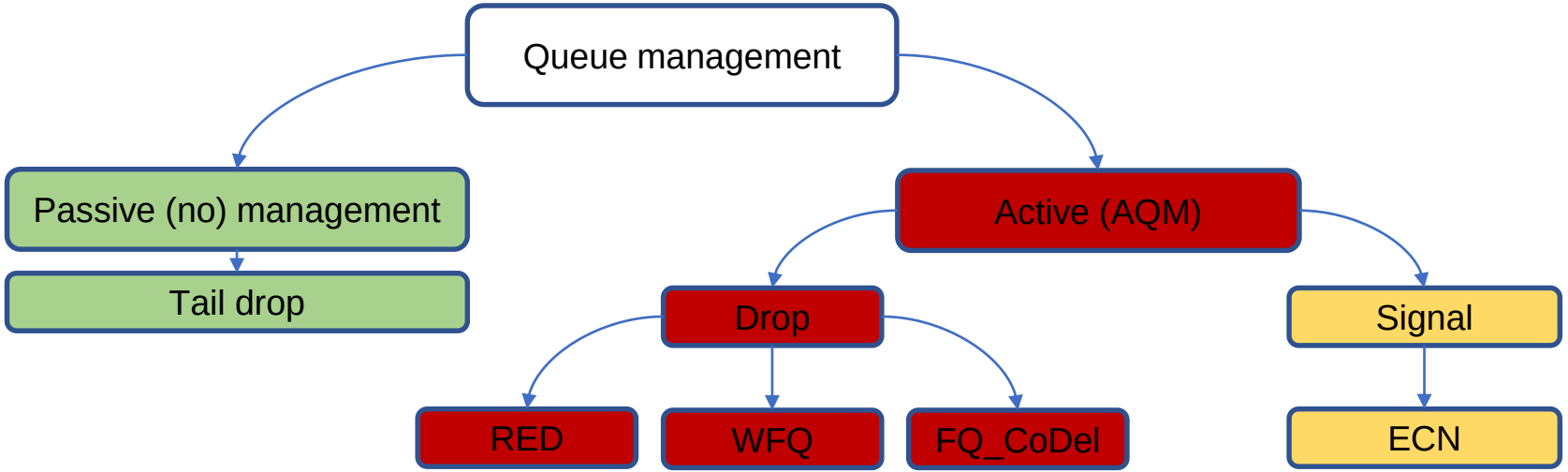
$$T=\frac{1.2}{\sqrt{p}}$$

- ✓ Comparing actual T to “Reno –T” we can analyze *relative fairness* i.e. how aggressive protocol is vs. standard TCP.
- ✓ Equations might be much more complex and take into account RTT, packet size.

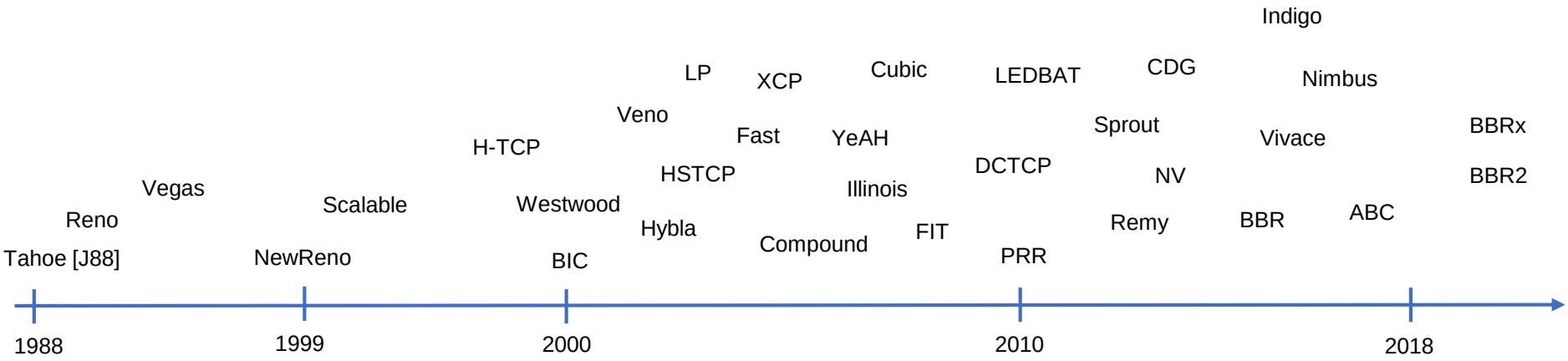
Call for help!

Sometimes this isn't enough so to **ask network for help** is a good idea!

- ✓ Routers know their own state (buffer load, link speed).
- ✓ Router can separate different kinds of flows.



Timeline


Indigo
LP
XCP
Cubic
LEDBAT
CDG
Nimbus
Veno
Sprout
BBRx
H-TCP
Fast
YeAH
Vivace
HSTCP
DCTCP
NV
BBR2
Vegas
Illinois
Scalable
Westwood
Reno
Hybla
Remy
BBR
ABC
FIT
Compound
Tahoe [J88]
NewReno
BIC
PRR
1988
1999
2000
2010
2018

Reno (1998)

Core idea:
Tahoe + “Fast Recovery”.

What do we address: **non-optimal behavior during loss recovery.**

Operation:
- Send Fast retransmission and then:
- Set ***sstresh*** to ***cwnd***/2, set ***cwnd*** to ***sstresh***+3.
- Increase ***cwnd*** on 1 SMSS for every received next Dup ACK (“inflate phase”).
- Decrease ***cwnd*** to ***sstresh*** after receiving higher ACK (“deflate phase”).

Reason: we treat Dup ACKs stream as ***good*** sign (because packets somewhere are leaving our network!) But we are stuck with “unacknowledged” window edge because of packet loss and can’t use capacity becoming available. So let’s manipulate ***cwnd*** temporarily for this period and bring things back when it ends.

New Reno [RFC3782]

Core idea: **"The Default One"**

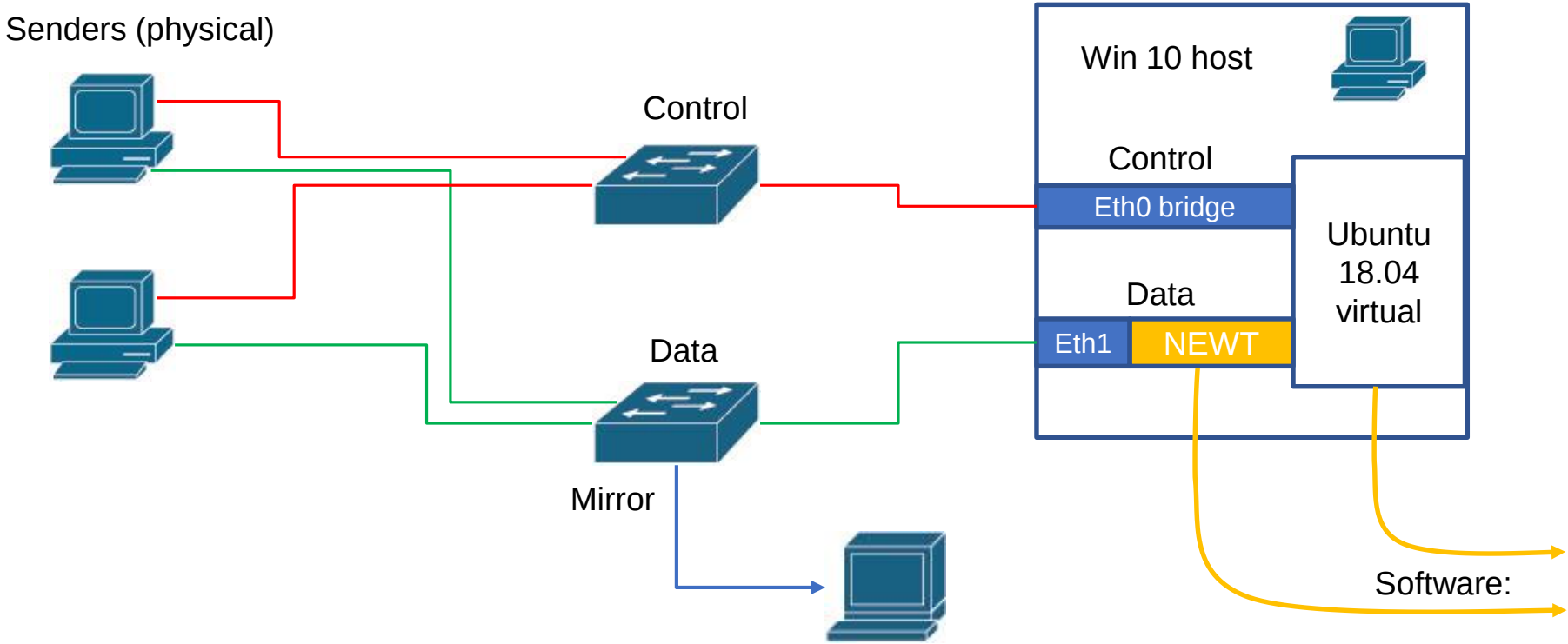
This is Reno + improved packet loss handling (**only for multiple segments loss**).

What do we address: **loss burst.**

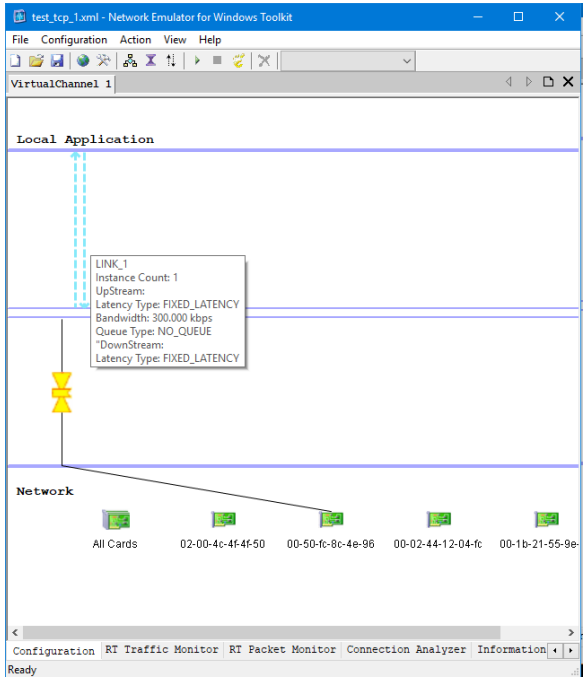
Reason:
If multiple segments were lost, this can mess up our "inflate-deflate" strategy. We'll deflate ***cwnd*** even if we receive *partial ACK* (higher than the one in Dup ACK stream, but lower than packet we sent last before loss). Therefore we'll deflate ***cwnd*** too early!

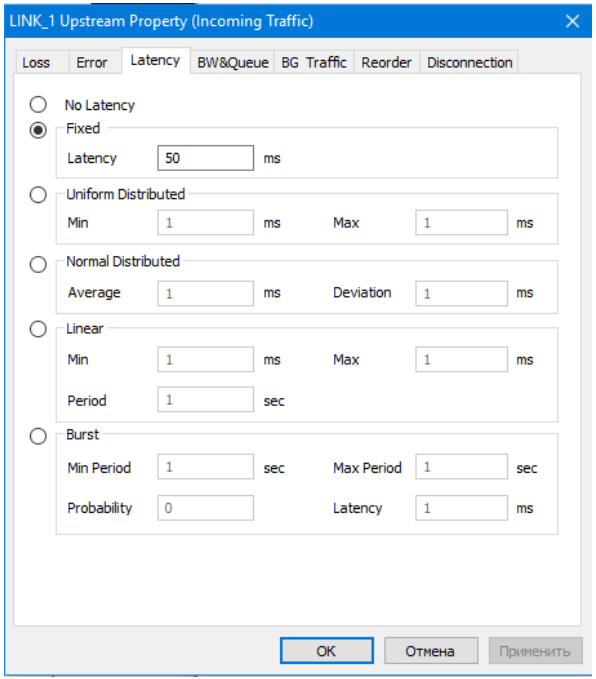
Solution:
- Remember highest SEQ at the moment of packet loss detection ("Recovery point").
- Do NOT deflate ***cwnd*** unless we receive an ACK for Recovery Point.

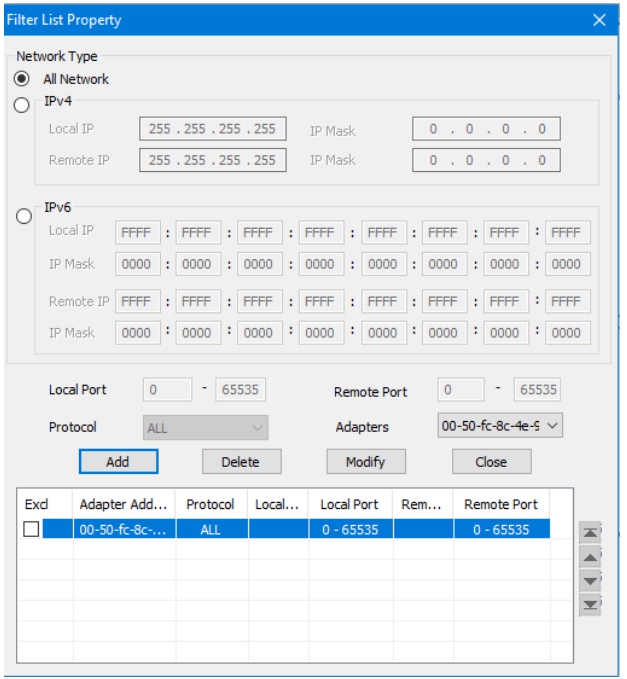
Test system


Senders (physical)
Control
Data
Mirror
Win 10 host
Control
Eth0 bridge
Data
Eth1
NEWT
Ubuntu
18.04
virtual
Software:

Software 1 - NEWT





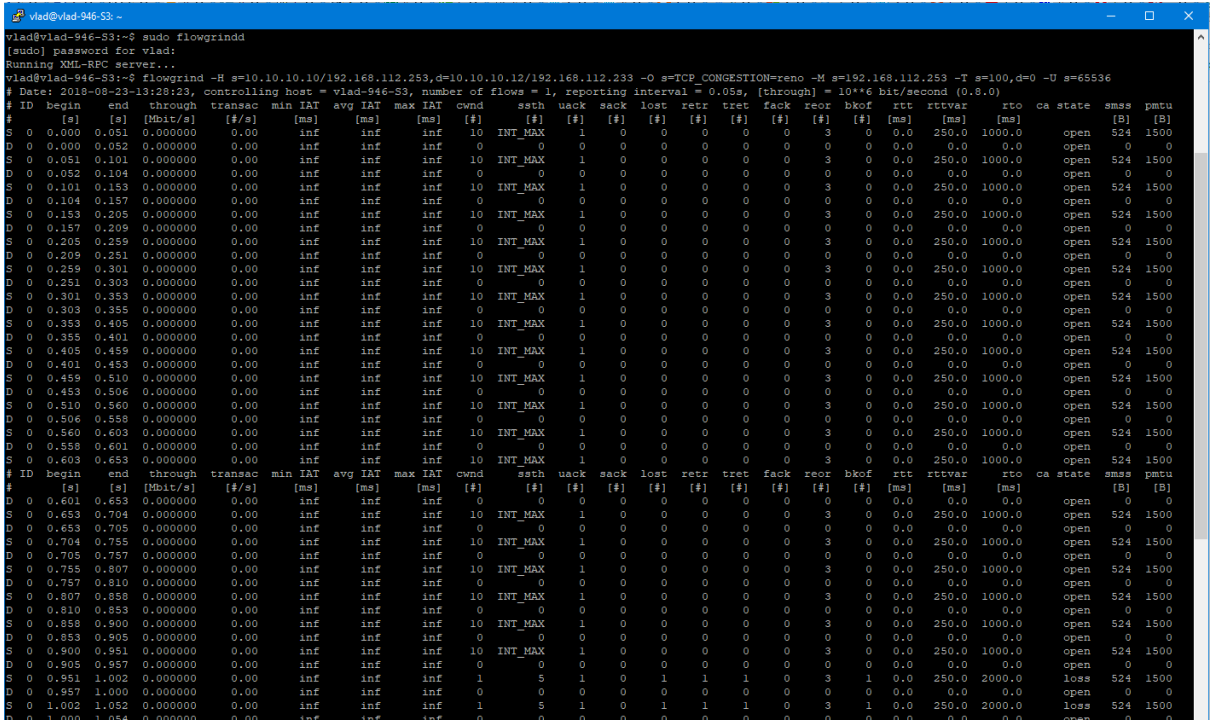


https://blog.mrpol.nl/2010/01/14/network-emulator-toolkit/

Software 2 - flowgrind

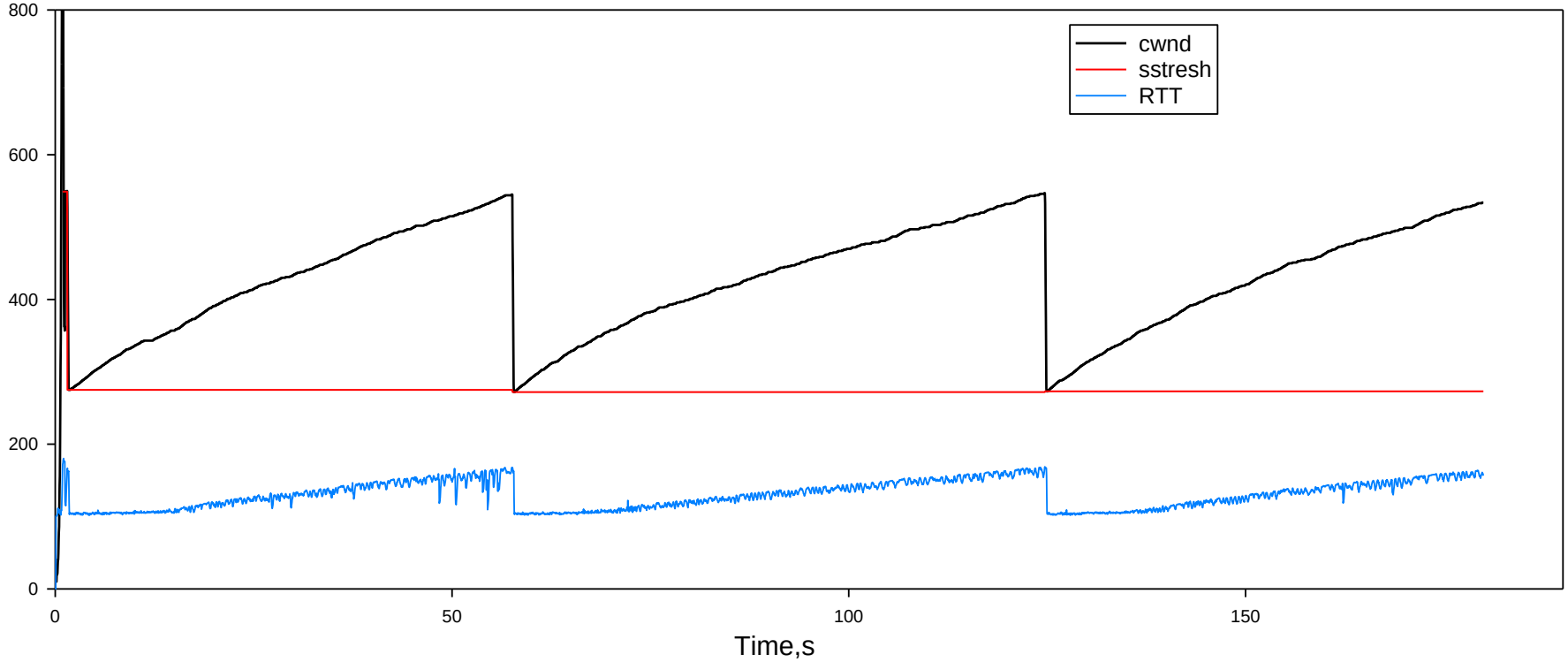
- ✓ Allows separation between control and data traffic.
- ✓ Large number of monitored values (including current ***cwnd*** and ***sstresh*** size, yeah!)
- ✓ Various traffic generation patterns.
- ✓ Individual TCP flow parameters setting.
- ✓ An ability to start flow from any PC running flowgrind daemon.
- ✓ Possibility to redirect output table to text file for parsing.

- Sensitive to incorrect arguments (often gets stuck and reboot is needed).
- Problems with NAT'ed endpoints.
- No Windows version = no Compound TCP.

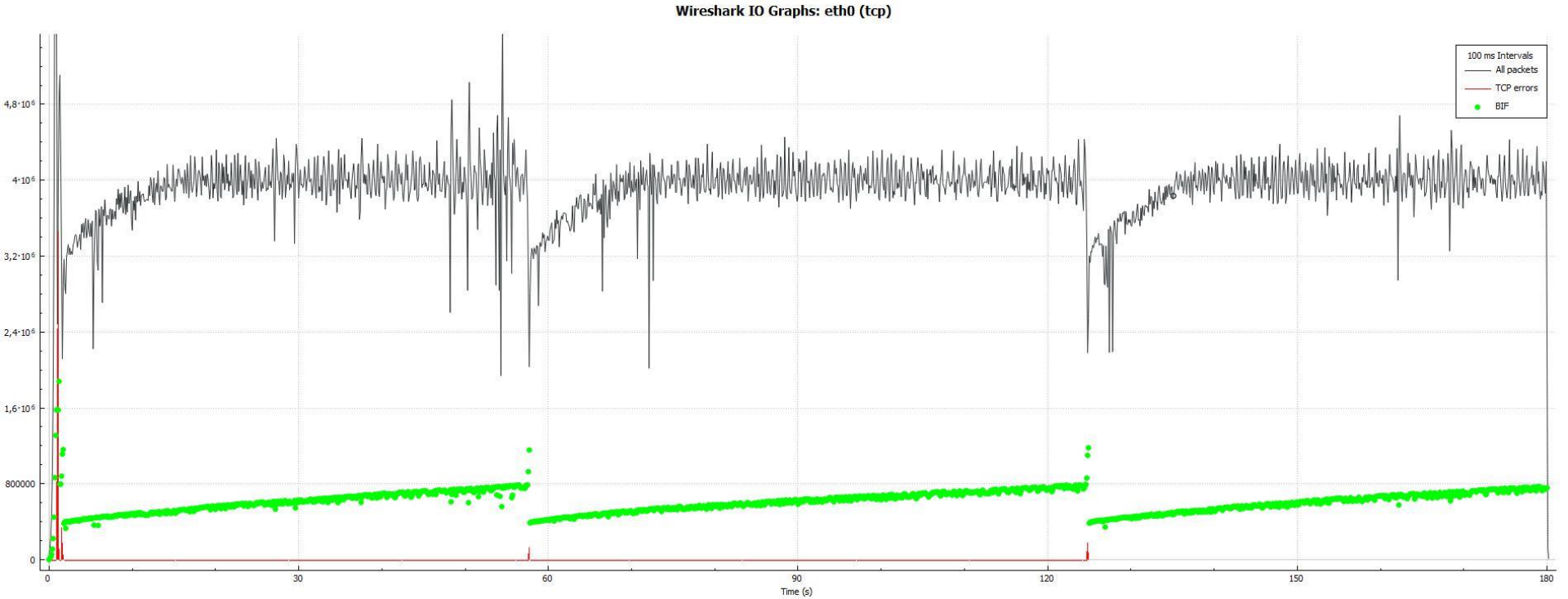


http://manpages.ubuntu.com/manpages/bionic/man1/flowgrind.1.html

NewReno on 40Mbps_100ms link

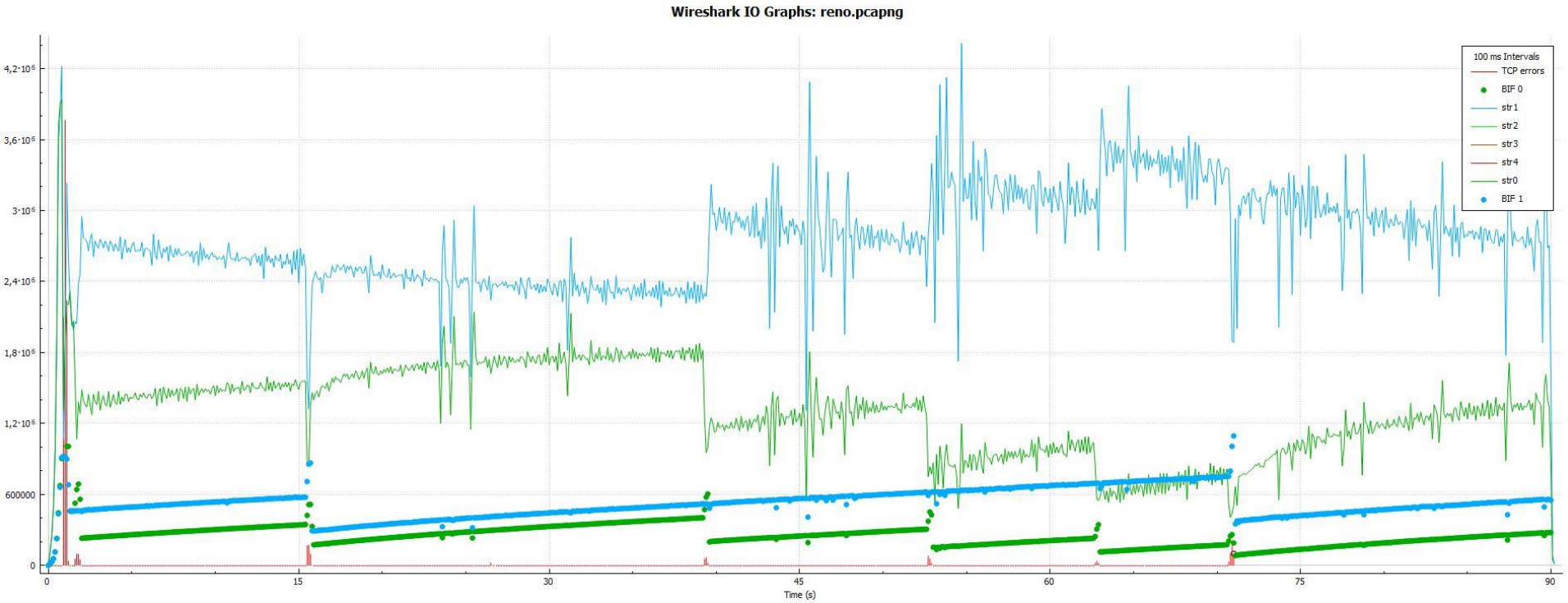


NewReno



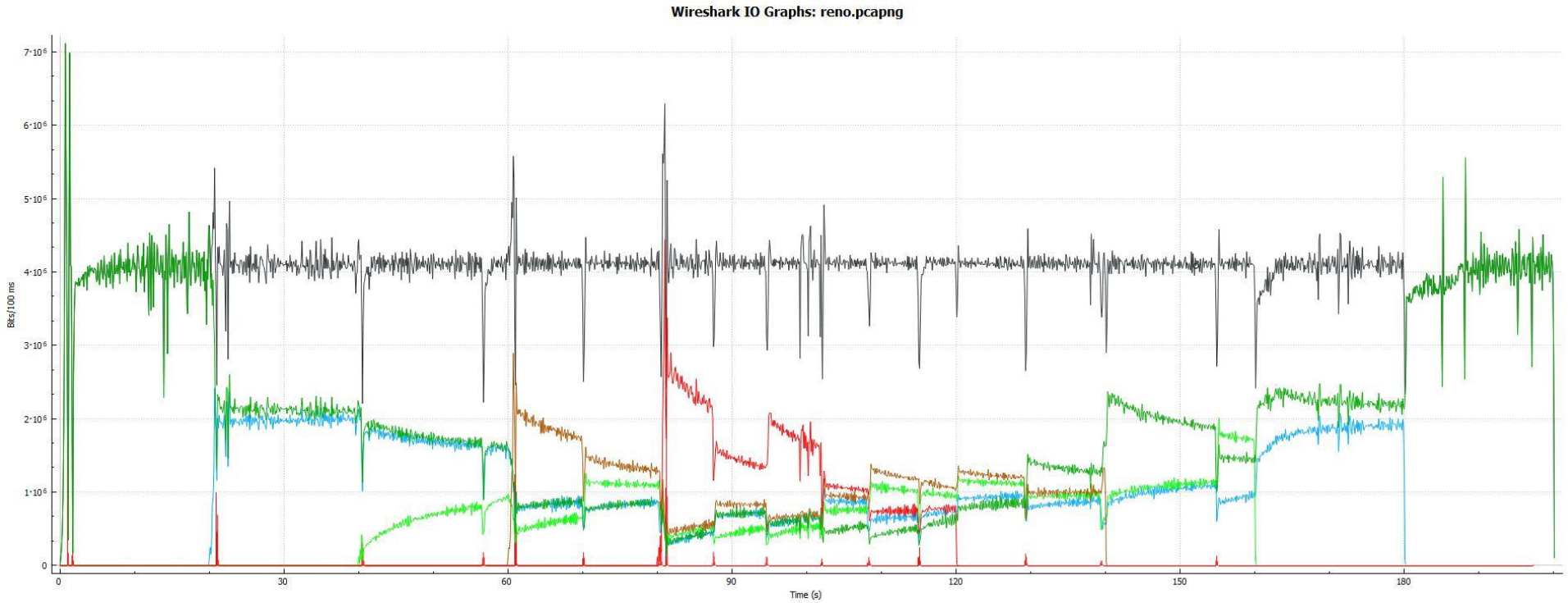
Collateral damage: Almost 3 Buffer overflows / 797k Total Packets

NewReno



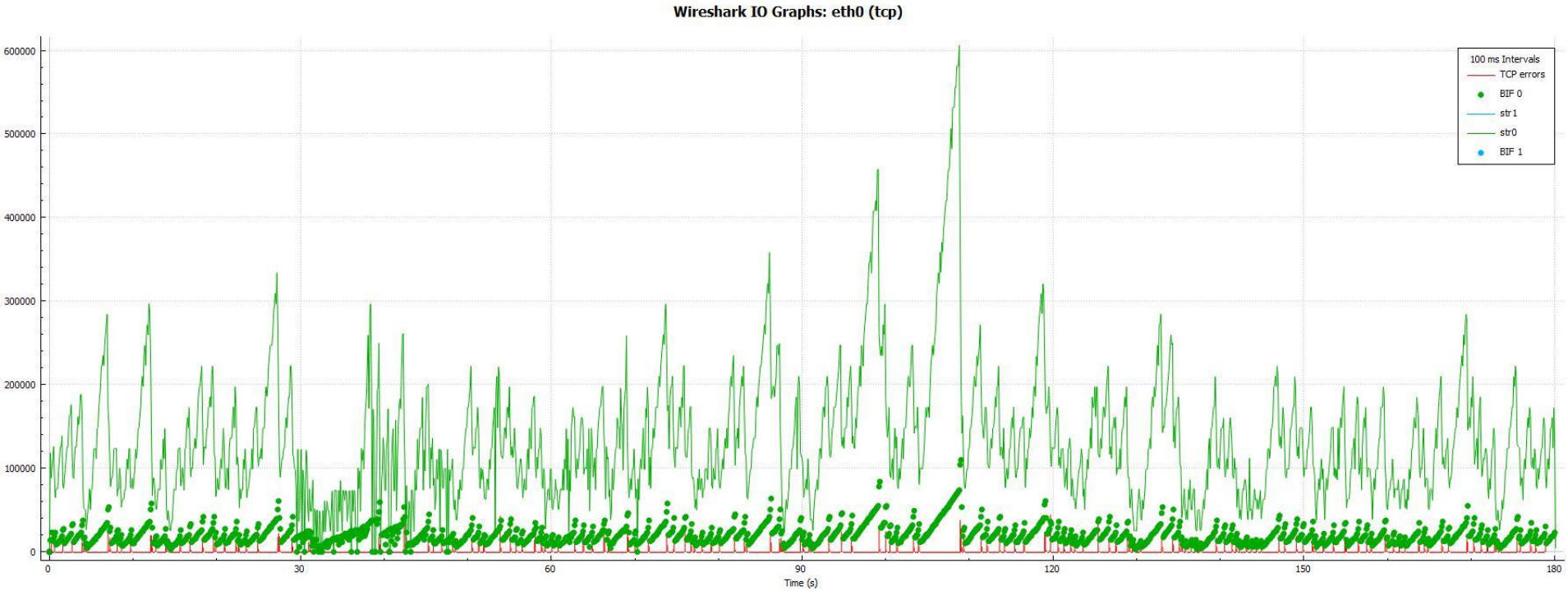
vs. Reno Friendliness

NewReno



5-stream convergence

NewReno



1% loss link behavior

Further progress

Several problems were observed with Reno:

- ✓ NewReno was doing its job fine those days, but later with the raise of LFN and wireless it became clear that…
- × It can't work efficiently on high-BDP links (because ***cwnd*** fixed additive increase algorithm is too slow and ½ ***cwnd*** drop is too much). To utilize fully 1Gbps link with 100ms RTT it needs packet loss rate of $2x10^{-8}$ or less. With 1% loss in this link it can't go faster than 3Mbps. After packet loss event it needs 4000 RTT cycles to recover.
- × It treats any packet loss as congestion indicator (not good for wireless networks).
- × Often visits "cliff" area (this is common among all loss-based algorithms).
- × Has 1-Bit congestion indicator = inevitable high oscillation level (this is common among all loss-based algorithms).

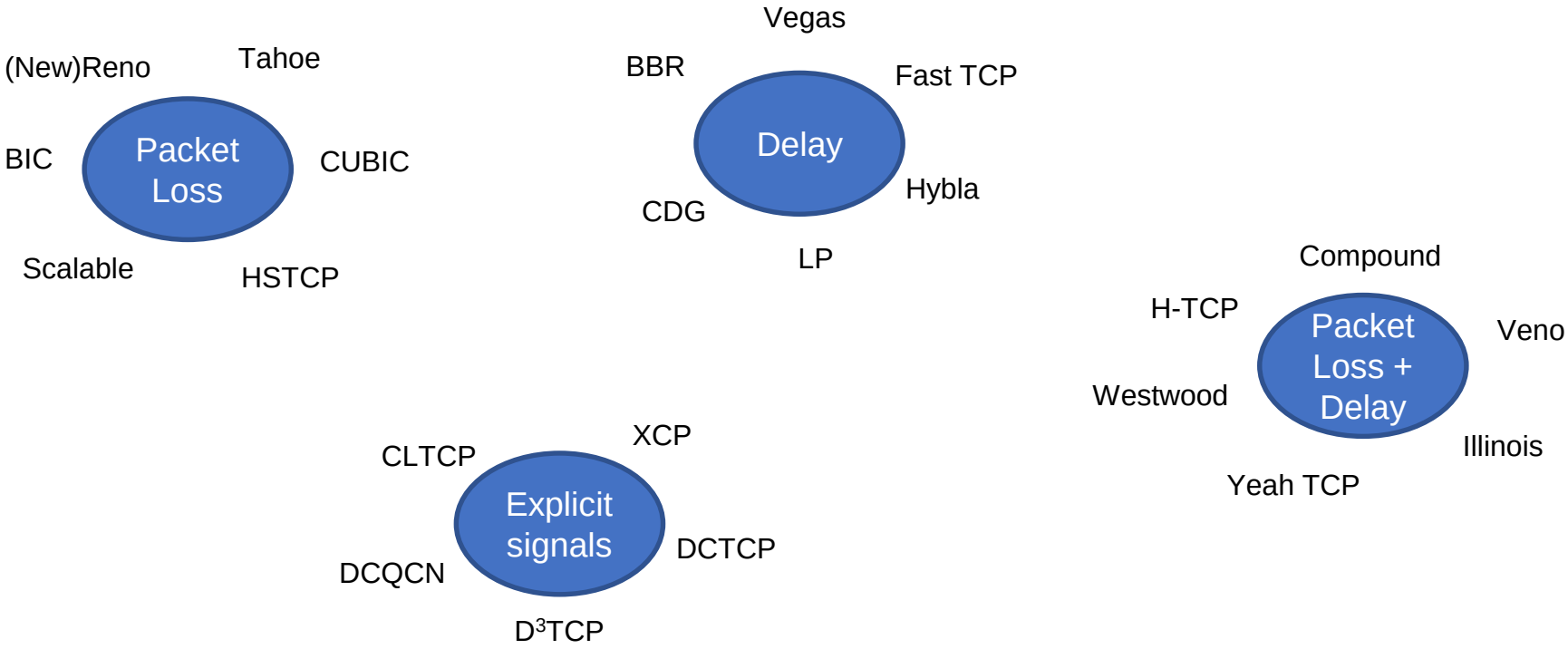
Further progress

How to make CC algorithm perform better? What to play with?
Remember feedback type and control? Let's play with them!

Feedback type:

- Packet loss
- Delay
- Both of them
- ACKs inter-arrival timing
- Explicit signals (ECN)

CA – feedback types



CA – control (action) tweaking

What about control?

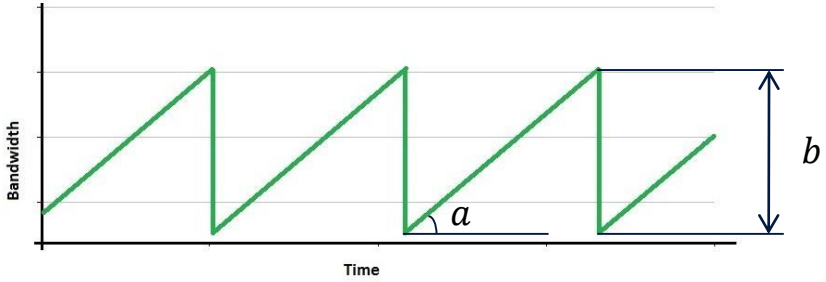
Step 1. Playing with AIMD factors.

We can play with a factor

$$cwnd = \begin{cases} cwnd + a & if\ congestion\ is\ not\ detected \\ cwnd * b & if\ congestion\ is\ detected \end{cases}$$

We can play with b factor

Therefore changing angle and "drop height".



Step 2. Adding variability.

Not constant a, but a=*f*(something)
Same with b.

Step 3. Shifting from AIMD to entirely different model.
(The most recent approach).

Scalable TCP – first “high BDP” try

Core ideas: **“Psycho”** Source

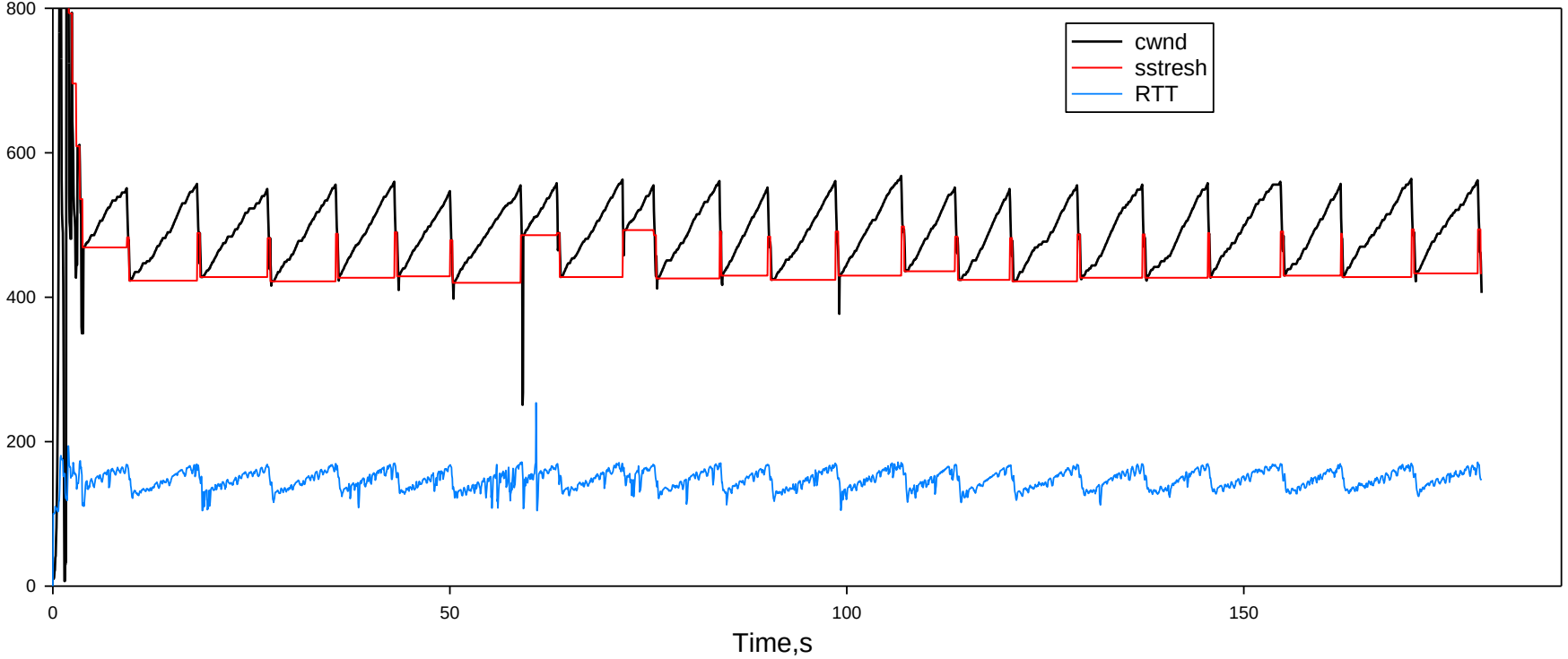
1. Aimed to deal with high BDP (first and simplest attempt to do it).
2. Uses packet loss as feedback (loss-based).
3. Uses MIMD approach as action profile (!).

***cwnd* control rules:**

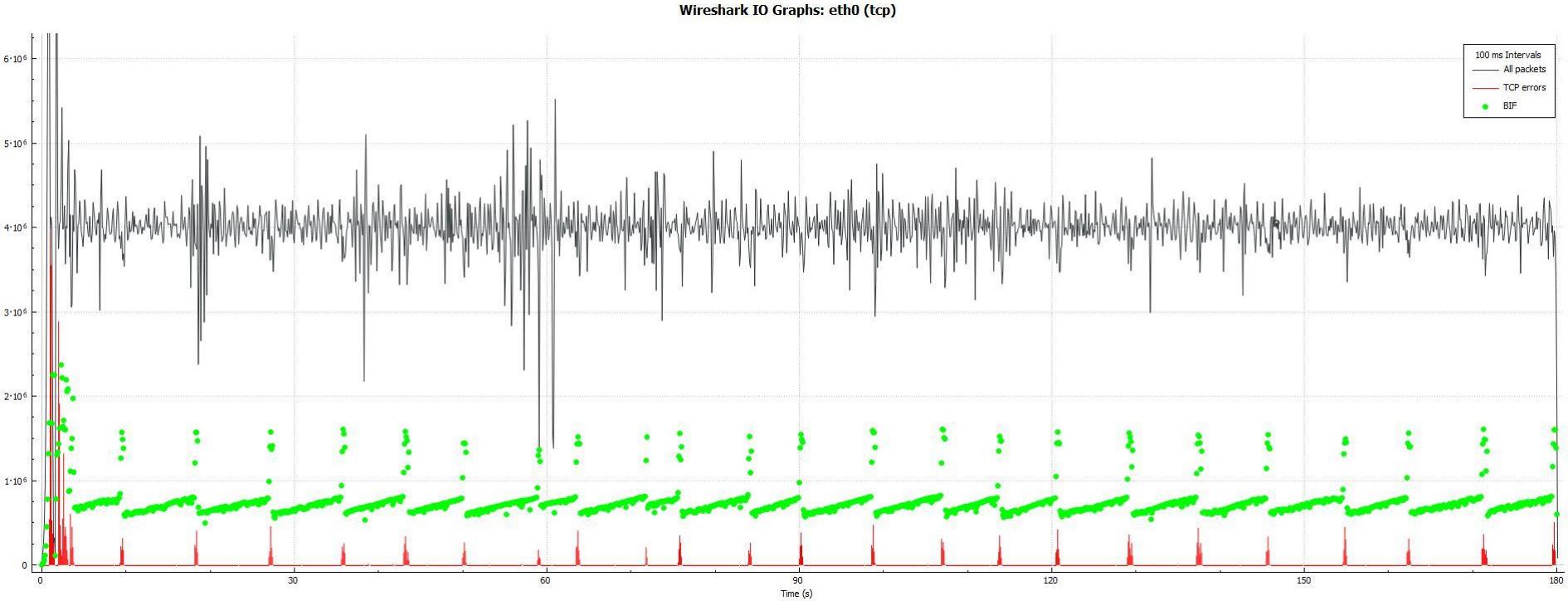
$$cwnd = \begin{cases} cwnd + 0.01 * cwnd & if\ congestion\ is\ not\ detected \\ cwnd * 0{,}875 & if\ congestion\ is\ detected \end{cases}$$

✓ Much more efficient than Reno in high BDP networks.
✓ Recovery time after packet loss (200ms RTT, 10Gbps link) – 2,7 sec.
× RTT fairness, TCP friendliness – terrible. Kills Reno easily.

Scalable TCP

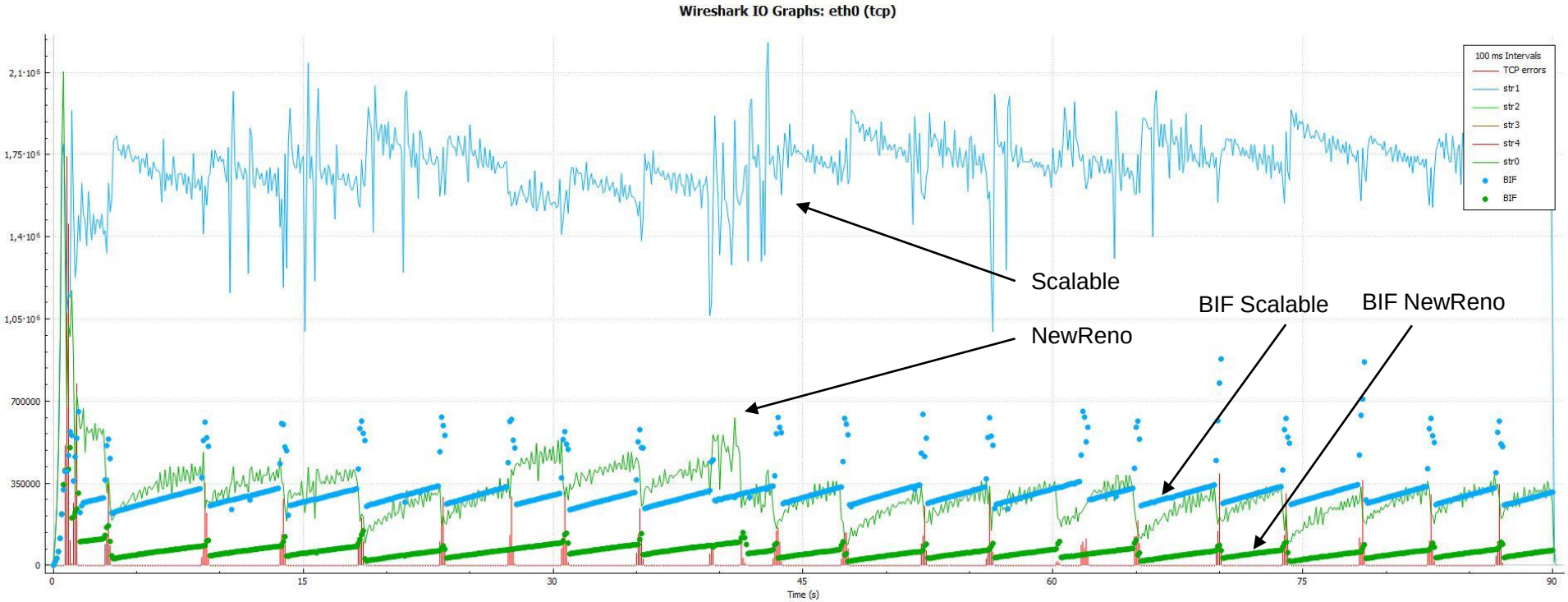


Scalable TCP



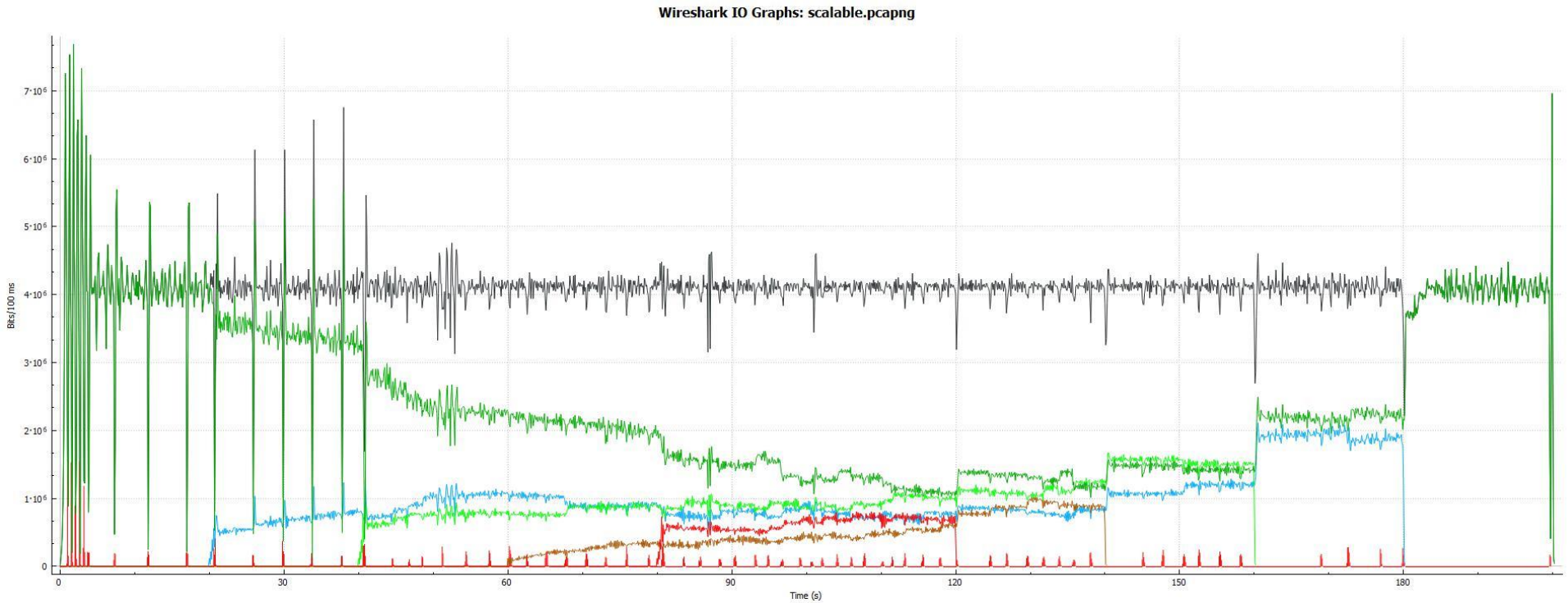
Collateral damage: 23 Buffer overflows / 812k Total Packets

Scalable vs NewReno



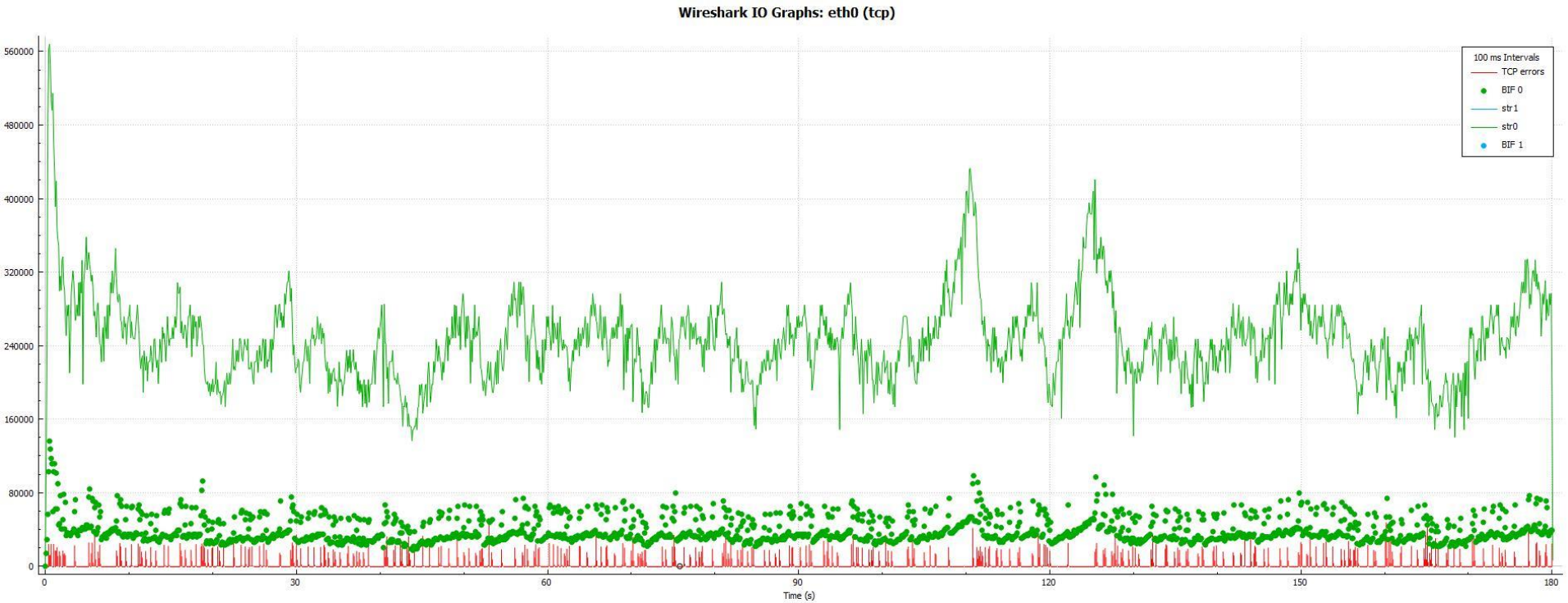
It's unfair to say the least. 20Mbps, 100ms link.

Scalable



5-stream convergence

Scalable



1% loss link behavior

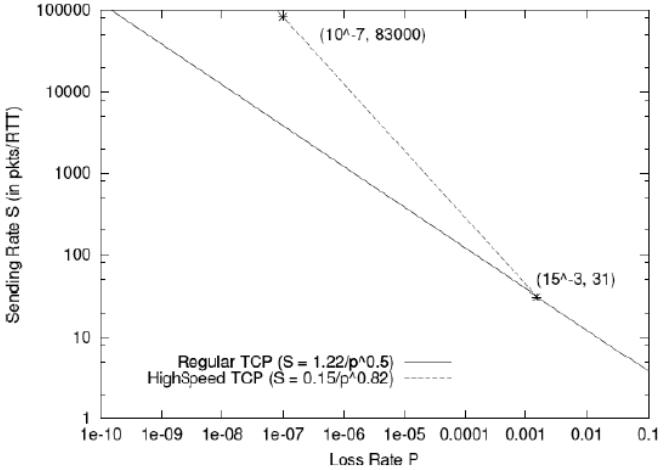
Highspeed TCP [RFC 3649]

Core ideas:

"Medicated psycho"

Source

1. Aimed to deal with high BDP.
2. Uses packet loss as feedback (loss-based).
3. Uses AIMD approach as action profile.
4. "Let's live with Reno on low-BDP, but take what it can't take on high-BDP"



***cwnd* control rules:**

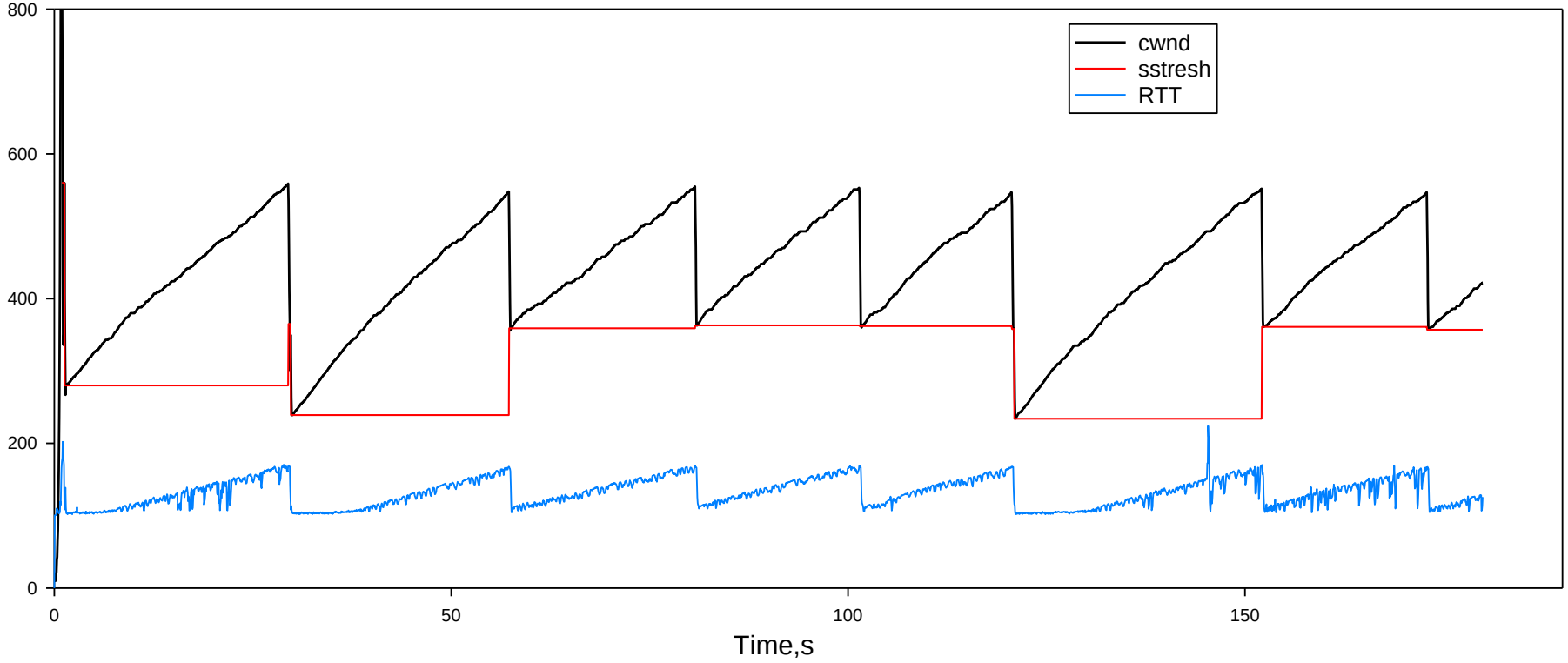
$$cwnd = \begin{cases} cwnd + a(cwnd)/cwnd & if\ congestion\ is\ not\ detected \\ cwnd - b(cwnd) * cwnd & if\ congestion\ is\ detected \end{cases}$$

Formula: $\mathbf{a}(w) = 2P_0W_0^2\,\mathbf{b}(w)/(2-\mathbf{b}(w))$

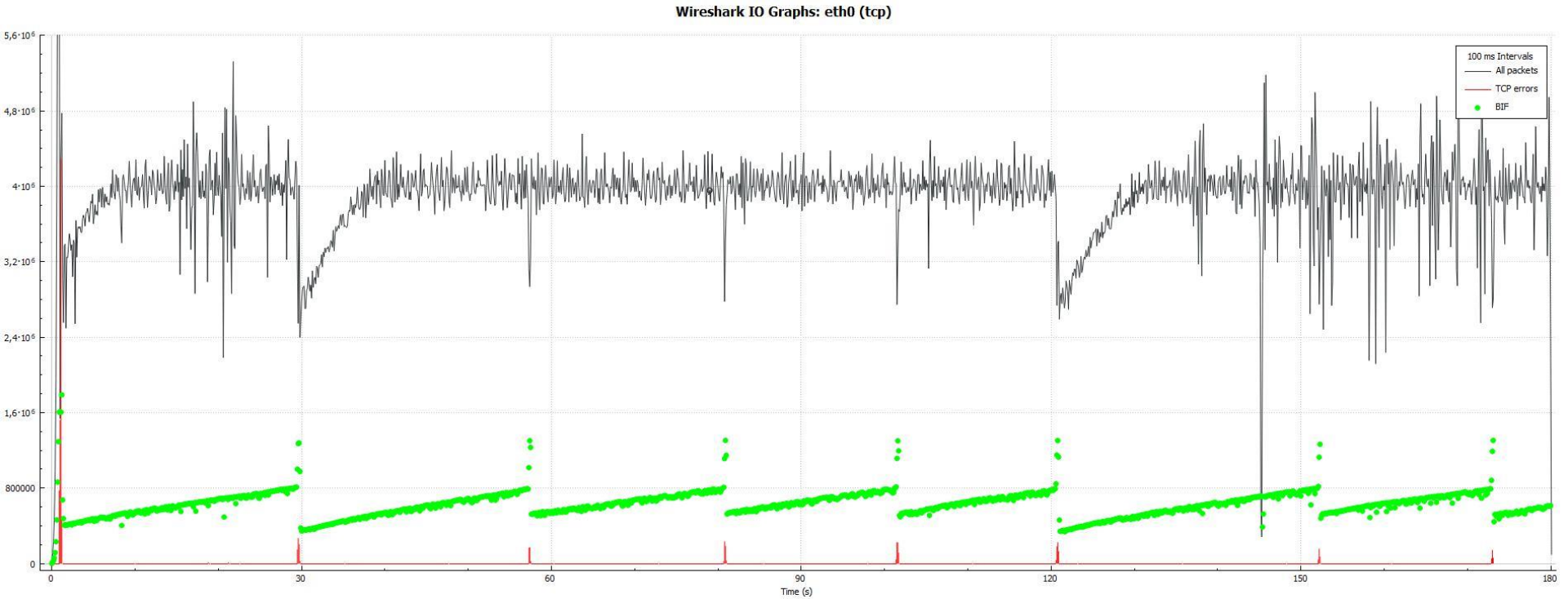
Main point is: a, b values depend on current ***cwnd*** size. If ***cwnd*** is less than 38*SMSS -> act as Reno (more bits in input!)

- Behaves less aggressive if a path is not LFN (for TCP friendliness).
- × RTT fairness - still bad.

Highspeed TCP

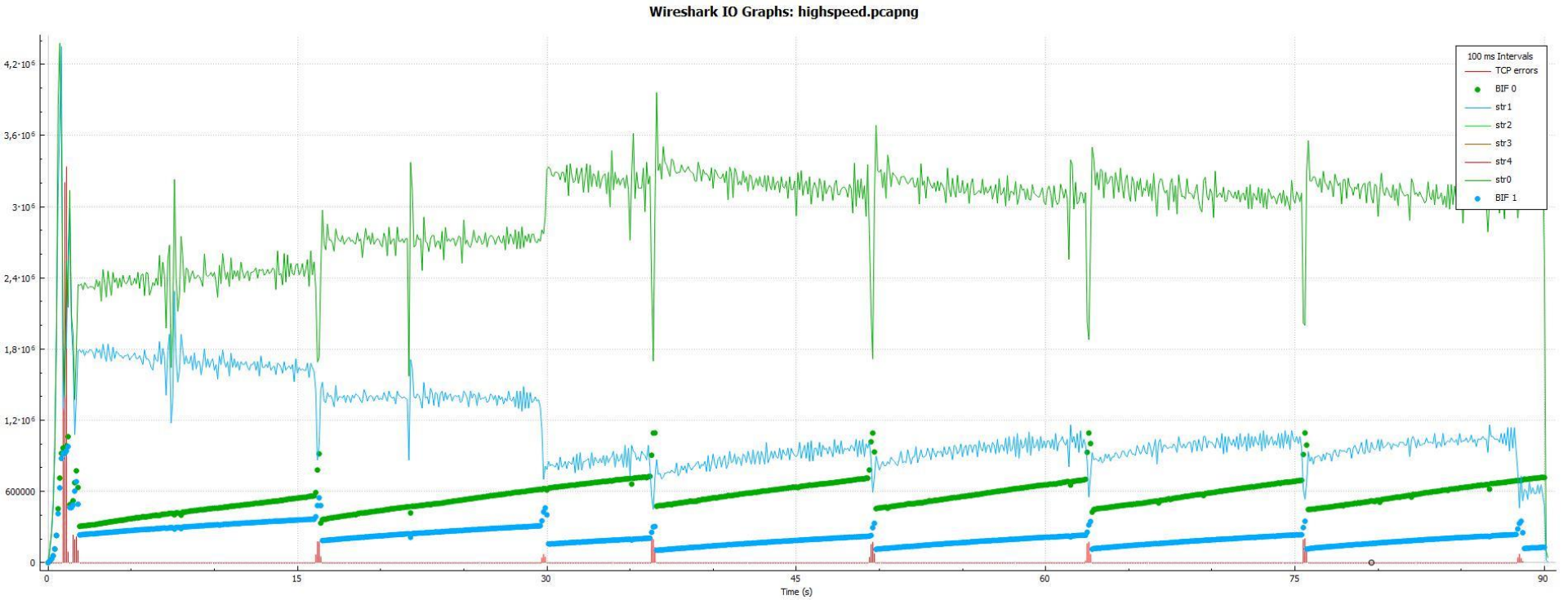


Highspeed TCP



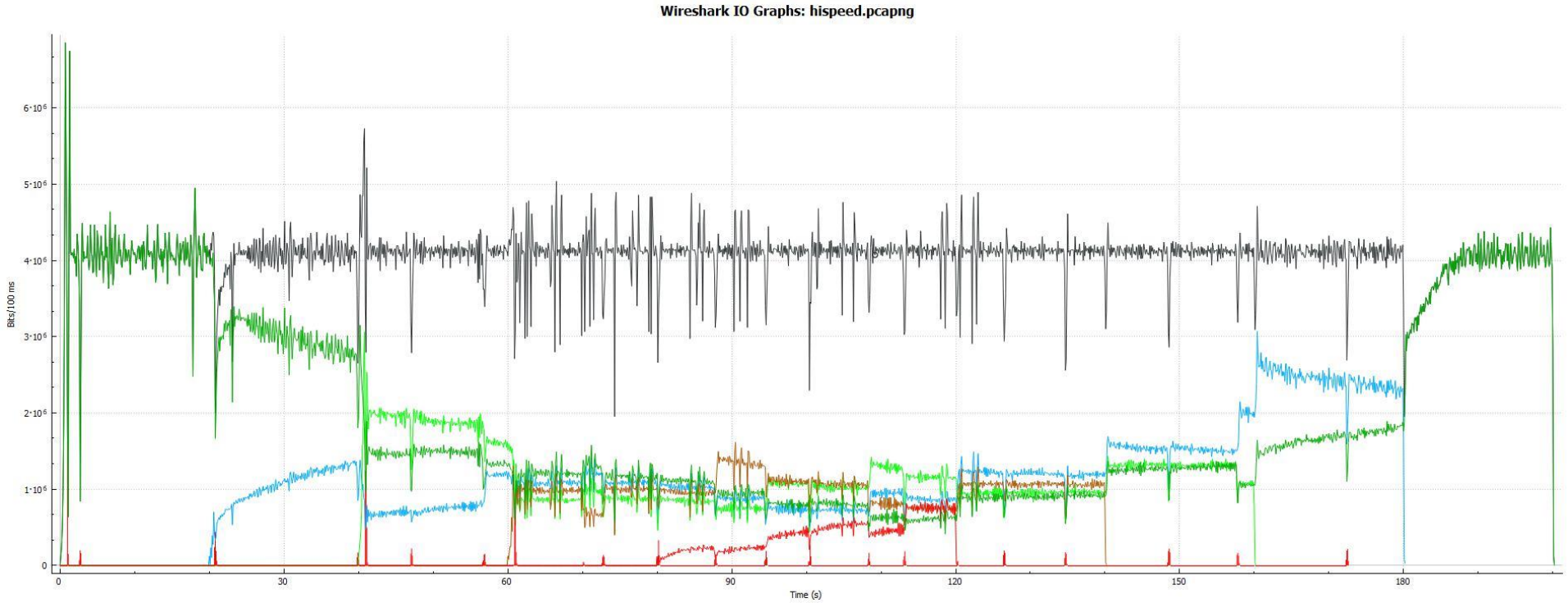
Collateral damage: 7 Buffer overflows / 790k Total Packets

Highspeed TCP



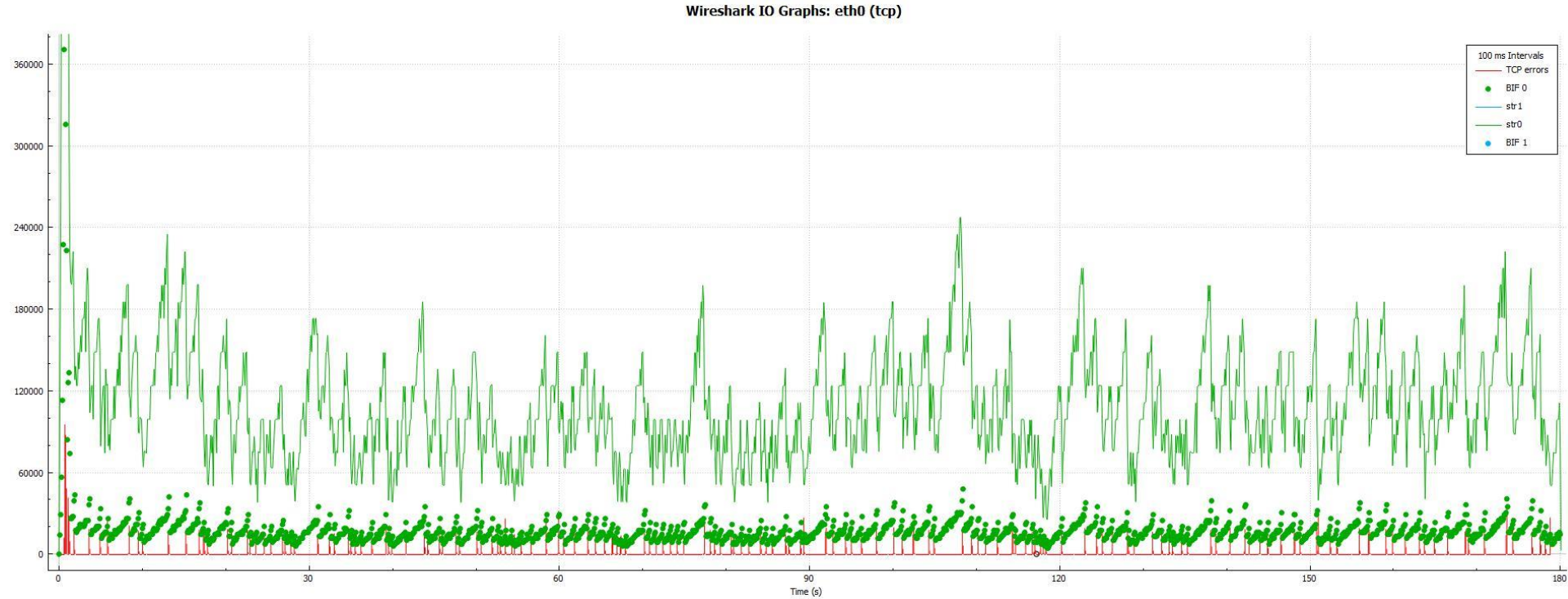
vs. Reno Friendliness

Highspeed TCP



5-stream convergence

Highspeed TCP



1% loss link behavior

CUBIC TCP

Core ideas:

"Ready-Steady-Go!"

Source

1. Aimed to deal with high BDP.
2. Uses packet loss as feedback.
3. Uses cubic function as action profile (concave/convex parts).
4. Default for all Linux kernels > 2.6.18, implemented in Windows since Win10.

***cwnd* control rules:**

In case of packet loss:

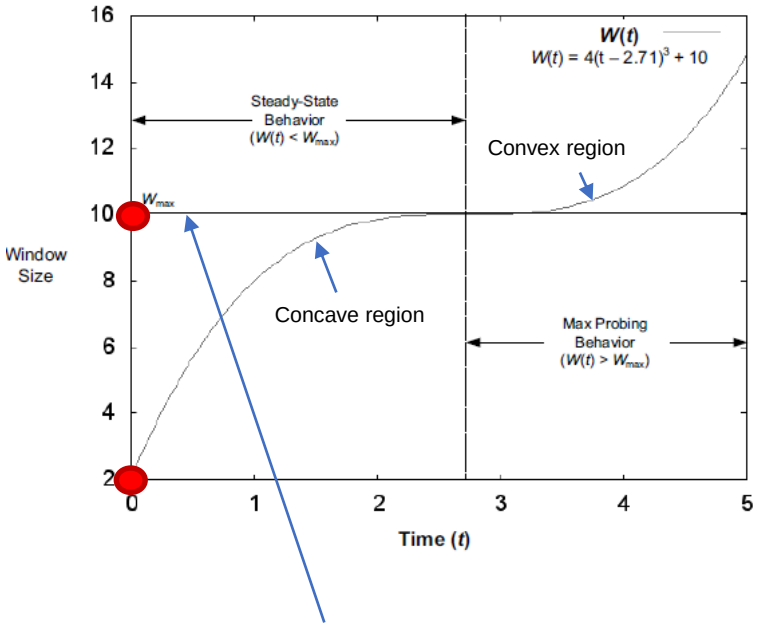
1. Set W_{max} to ***cwnd***;
2. Set ***cwnd, sstresh*** to (*1 - β)*cwnd* where default *β=0.8*
3. Grow ***cwnd*** using cubic function:

$$W(t) = C(t - K)^3 + W_{max} \quad \text{where:} \quad K = \sqrt[3]{\frac{\beta W_{max}}{C}}$$

Main point: approach last packet loss point slowly and carefully, but if there is no more packet loss here – begin ramp up to use possibly freed up resources.

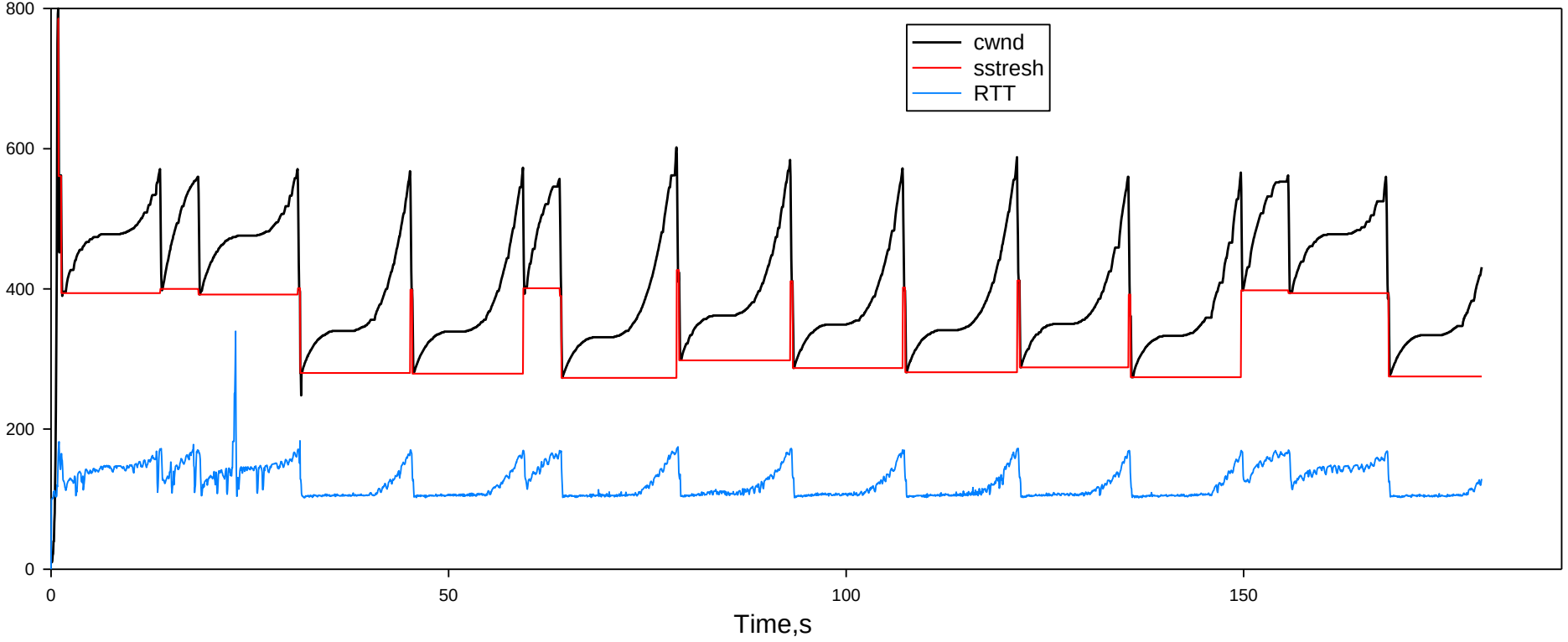
Additional techniques used: TCP friendly region, Fast convergence, Hybrid slow start.

✓ Coexistence with Reno on non-LFN links – moderate
✓ RTT fairness - good

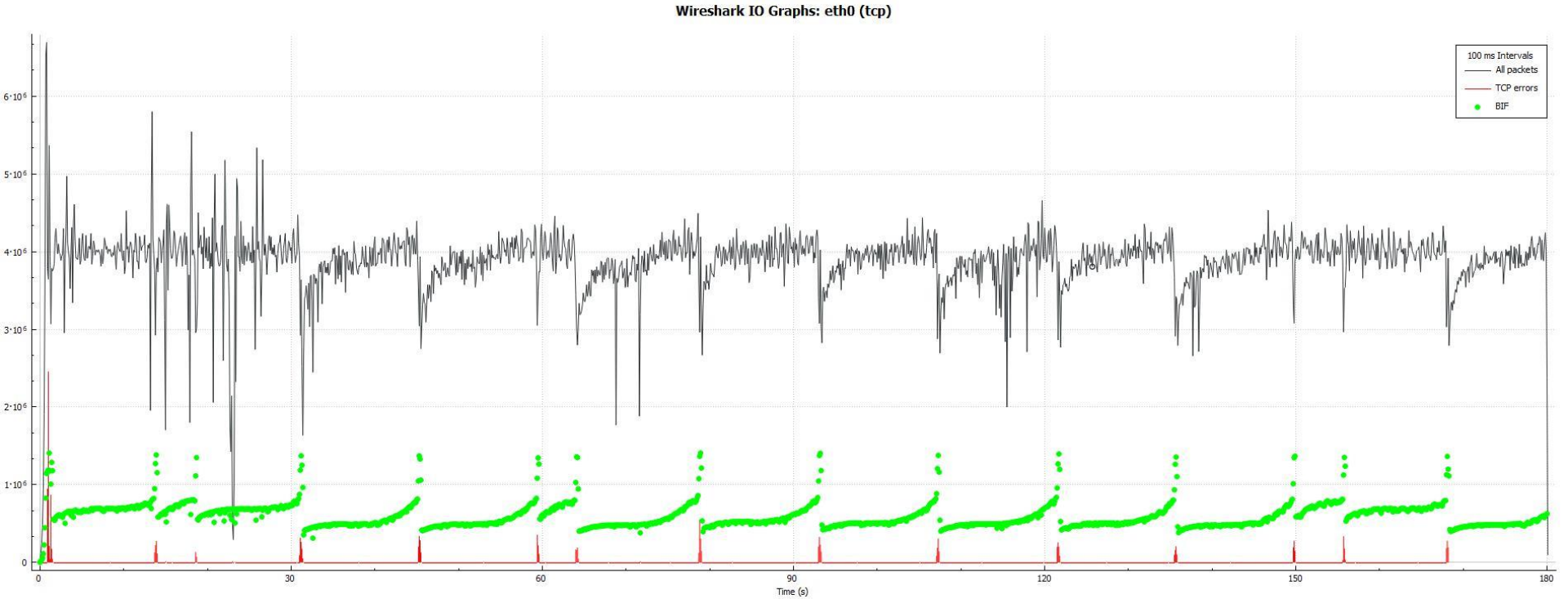


Last remembered value where packet loss happened

CUBIC TCP

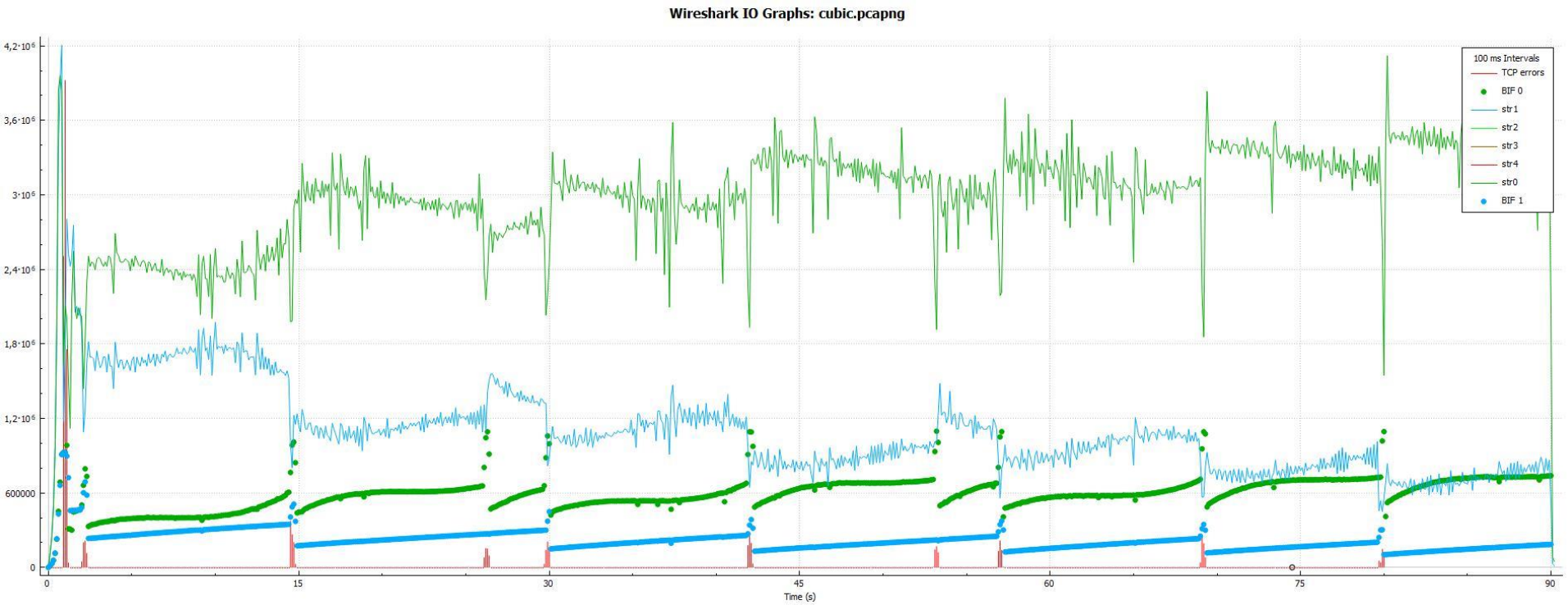


CUBIC TCP



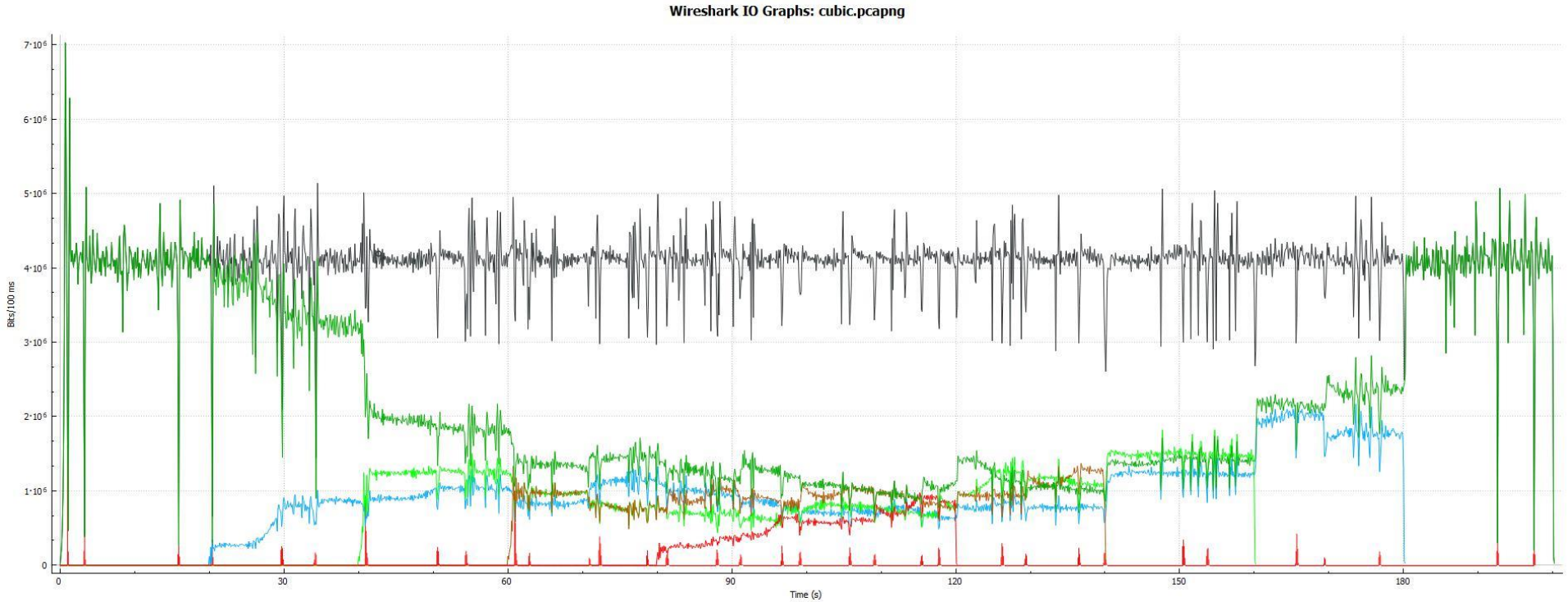
Collateral damage: 14 Buffer overflows / 790k Total Packets

CUBIC TCP



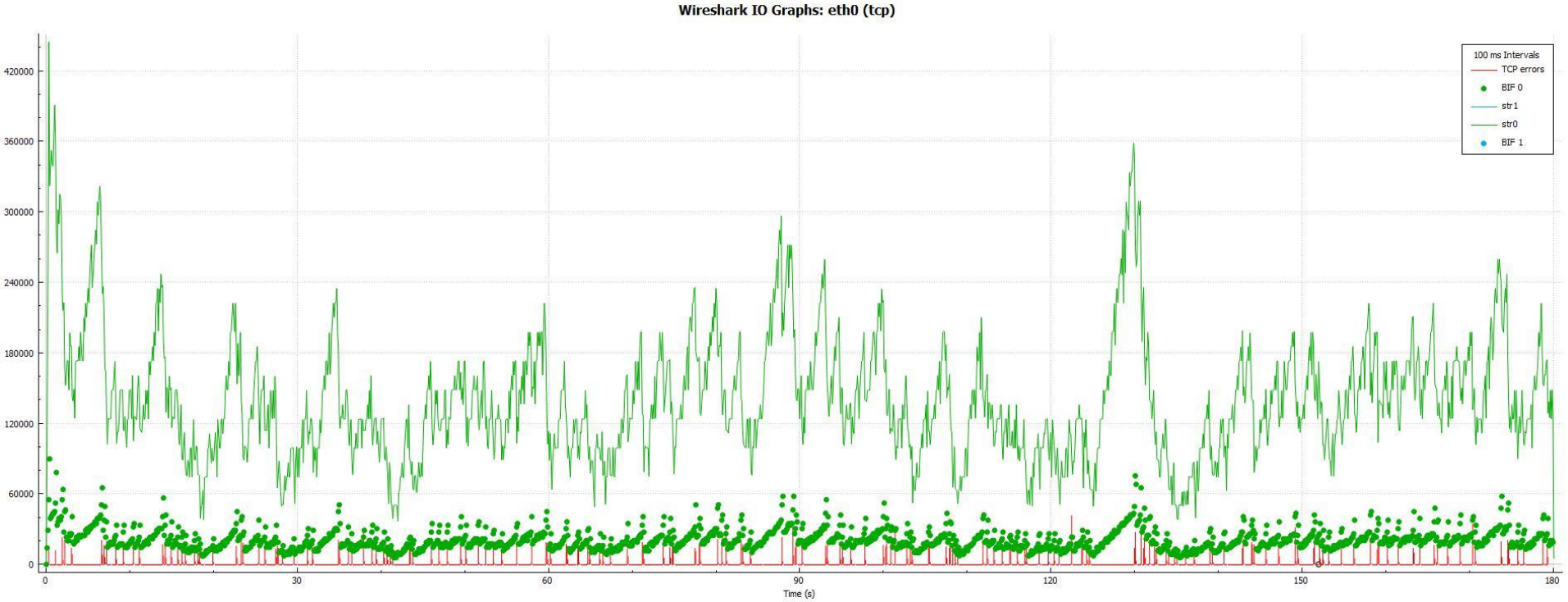
vs. Reno Friendliness

CUBIC TCP



5-stream convergence

CUBIC TCP

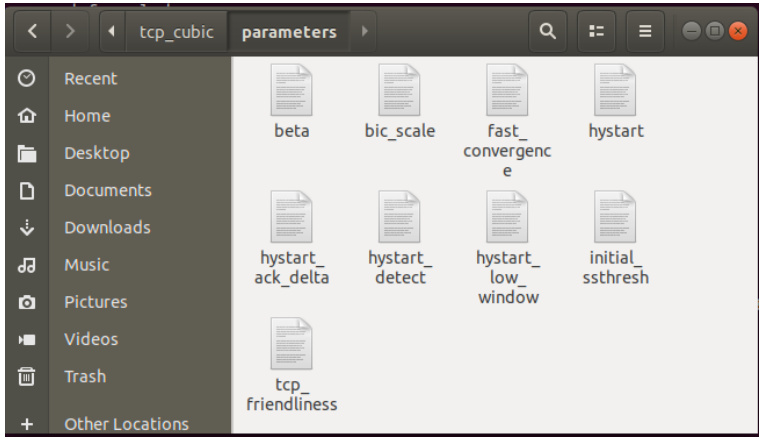


1% loss link behavior

CUBIC Fun Fact

If you look at CUBIC source code you'll spot some parameters can be tweaked!

```
61  /* Note parameters that are used for precomputing scale factors are read-only */
62  module_param(fast_convergence, int, 0644);
63  MODULE_PARM_DESC(fast_convergence, "turn on/off fast convergence");
64  module_param(beta, int, 0644);
65  MODULE_PARM_DESC(beta, "beta for multiplicative increase");
66  module_param(initial_ssthresh, int, 0644);
67  MODULE_PARM_DESC(initial_ssthresh, "initial value of slow start threshold");
68  module_param(bic_scale, int, 0444);
69  MODULE_PARM_DESC(bic_scale, "scale (scaled by 1024) value for bic function (bic_scale/1024)"
70  module_param(tcp_friendliness, int, 0644);
71  MODULE_PARM_DESC(tcp_friendliness, "turn on/off tcp friendliness");
72  module_param(hystart, int, 0644);
73  MODULE_PARM_DESC(hystart, "turn on/off hybrid slow start algorithm");
74  module_param(hystart_detect, int, 0644);
75  MODULE_PARM_DESC(hystart_detect, "hybrid slow start detection mechanisms"
76                   " 1: packet-train 2: delay 3: both packet-train and delay");
77  module_param(hystart_low_window, int, 0644);
78  MODULE_PARM_DESC(hystart_low_window, "lower bound cwnd for hybrid slow start");
79  module_param(hystart_ack_delta, int, 0644);
80  MODULE_PARM_DESC(hystart_ack_delta, "spacing between ack's indicating train (msecs)");
```



These knobs can be found (for Ubuntu) at **/sys/module/tcp_cubic/parameters**

Hybrid slow start

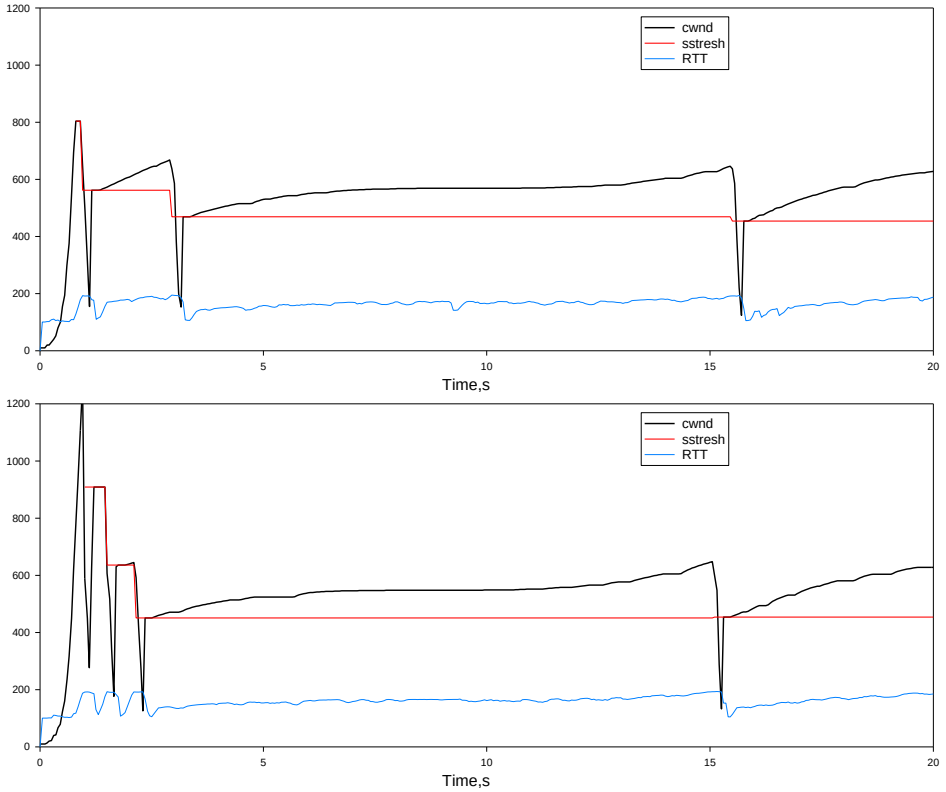
Problem: high aggressiveness during final slow start phase.

Solution: estimate a point where to exit Slow Start mode.

Methods:

- ACK train length measuring method.
- Inter-frame delay method.

Built-in in CUBIC algorithm.



VEGAS TCP

Core ideas:

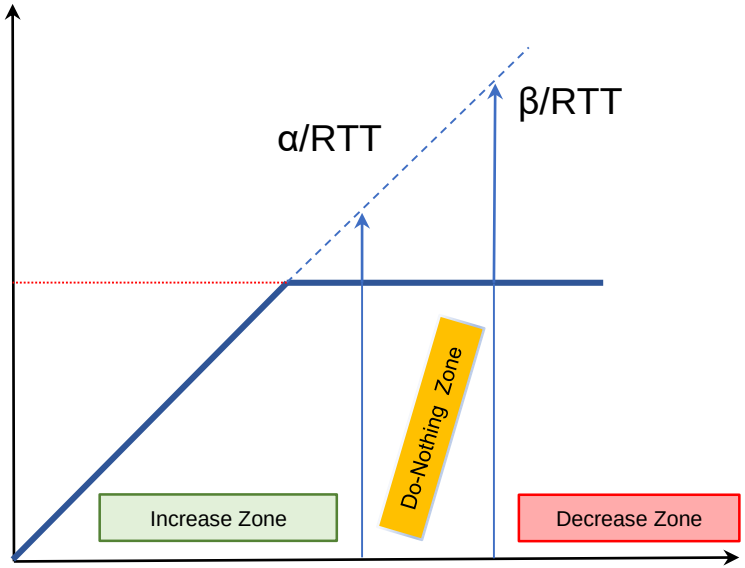
"Pacifist"

Source

1. First try to build delay-based algorithm (1994).
2. Uses delay as feedback (purely delay-based).
3. Uses AIAD as action profile.

***cwnd* control rules:**

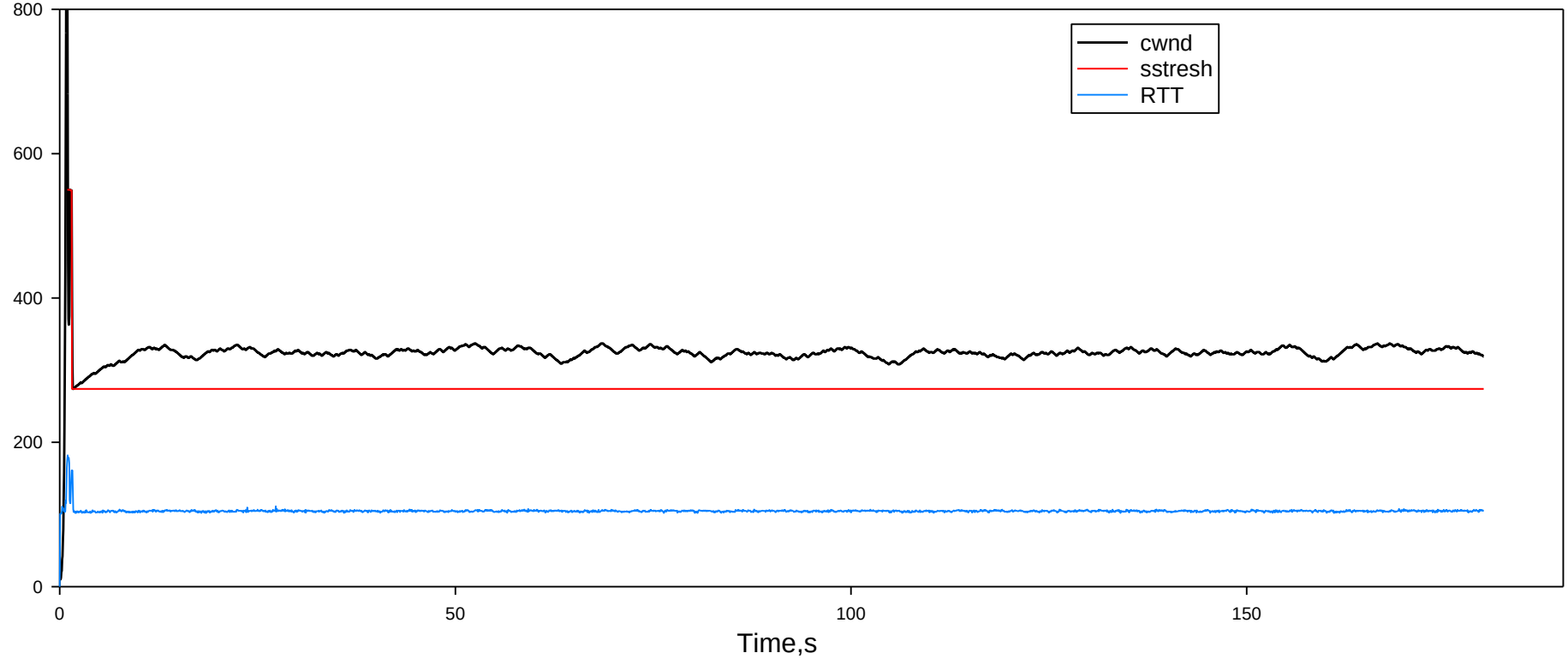
1. Measure and constantly update min RTT ("BaseRTT")
2. For every RTT compare Expected Throughput (***cwnd*** / BaseRTT) with Actual Throughput (***cwnd*** / RTT)
3. Compute difference = (Expected - Actual)/BaseRTT
4. Look where in range it lies and act accordingly (1 per RTT ***cwnd*** update frequency).
5. Switch to Reno if there are not enough RTT samples.



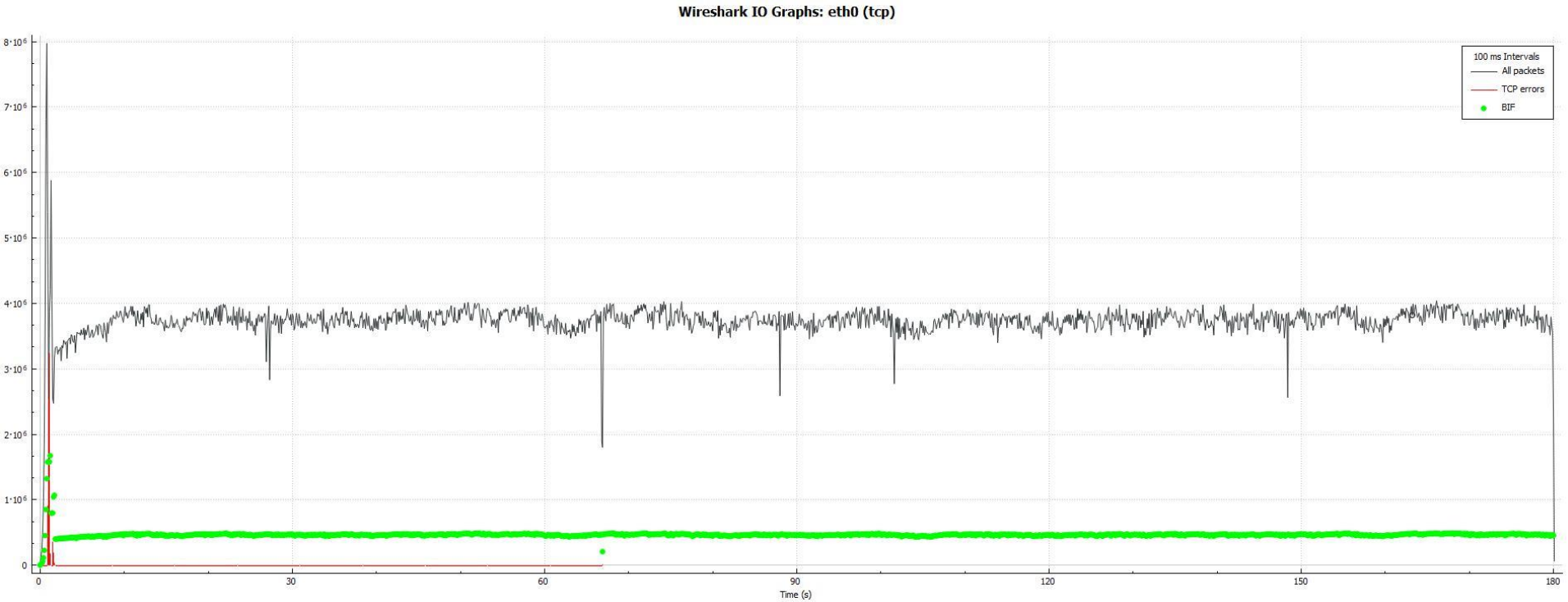
✓ Very smooth
✓ Doesn't act on Cliff zone
✓ Induces small buffer load, keeps RTT small

× Gets beaten by **any** loss-based algorithm
× Doesn't like small buffers
× Doesn't like small RTTs

VEGAS TCP

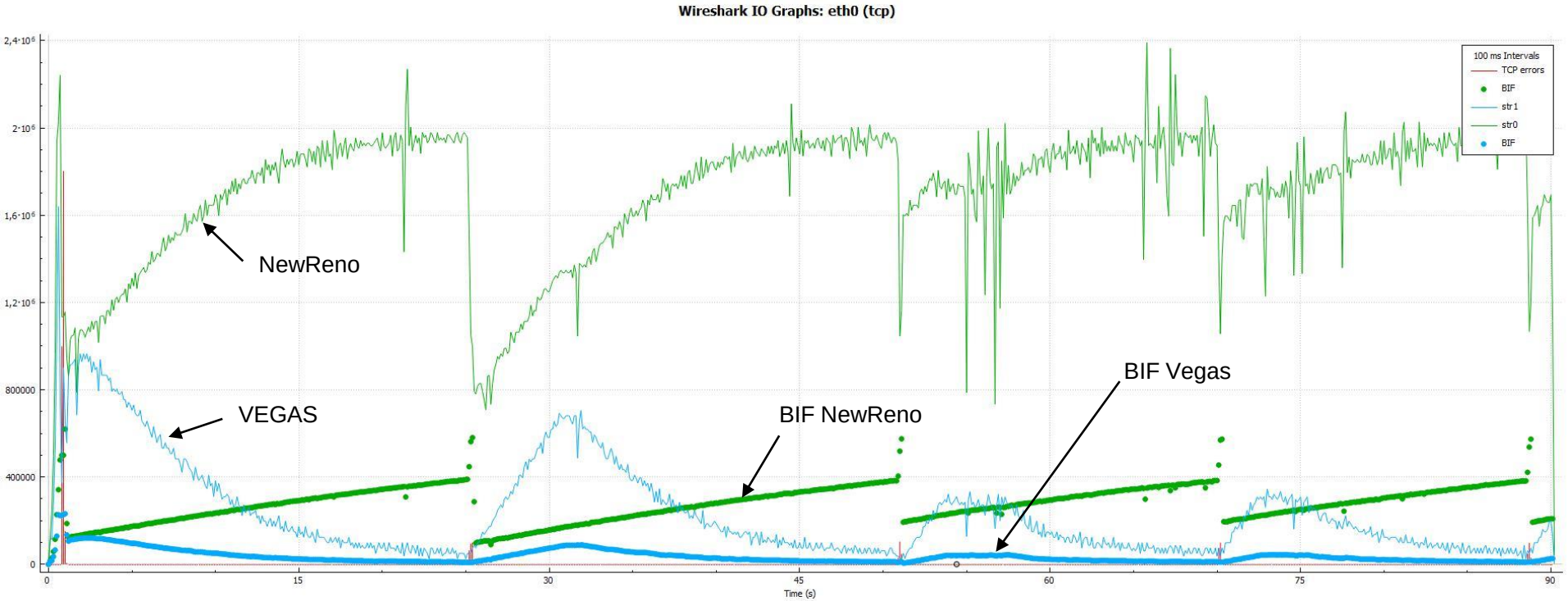


VEGAS TCP



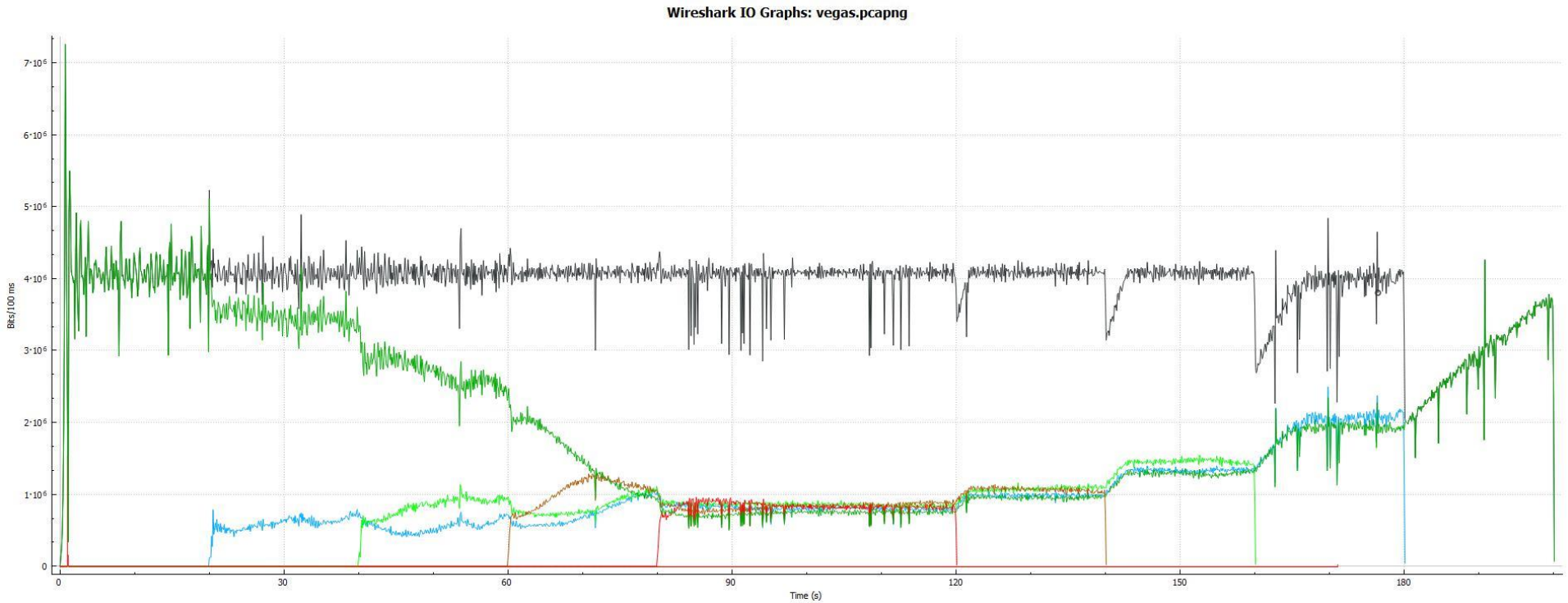
When it's alone – this is impressive! Collateral damage: almost unnoticeable / 767k Packets

NewReno vs. VEGAS TCP



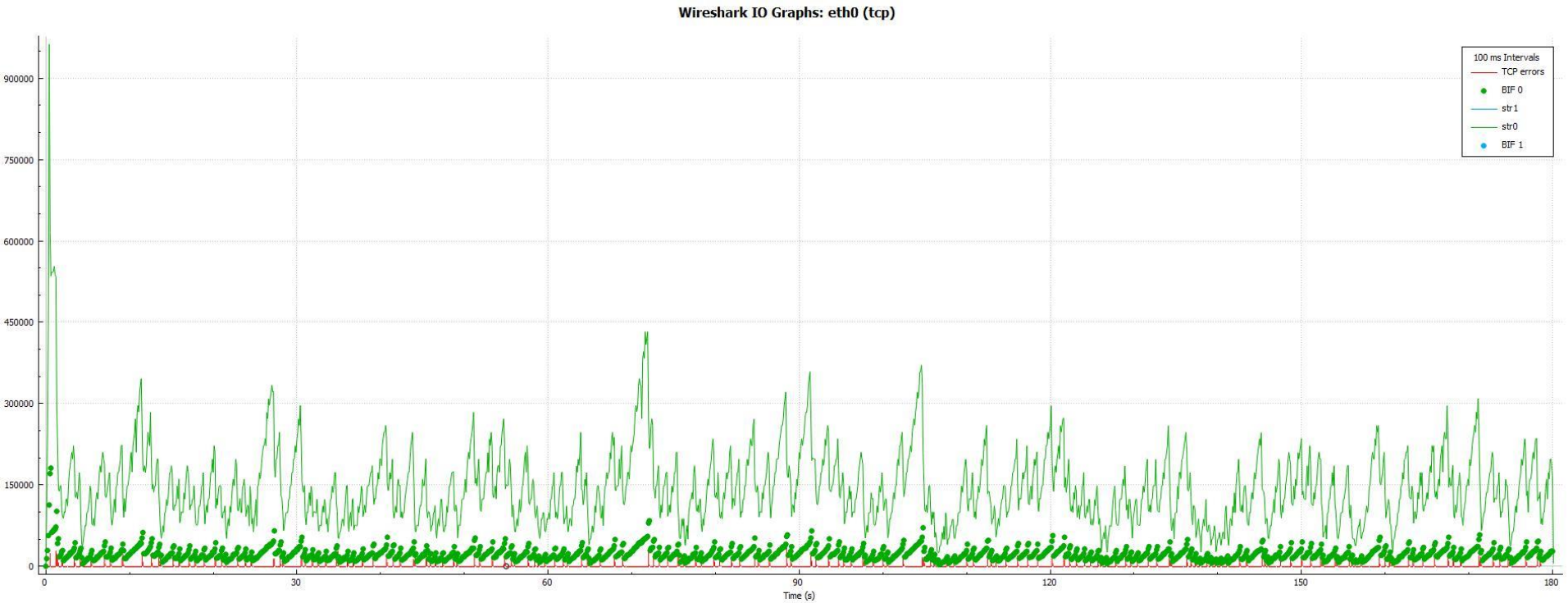
When VEGAS is not alone – this is a shame..

VEGAS TCP



5-stream convergence

VEGAS TCP



1% loss link behavior

ILLINOIS TCP

Core ideas:

"Careful"

Source

1. Uses packet loss and delay as feedback.
2. Uses modified AIMD with delay-dependent variables as action profile.

***cwnd* control rules:**

$$cwnd = \begin{cases} cwnd + \alpha/cwnd & if\ congestion\ is\ not\ detected \\ (1-\beta) * cwnd & if\ congestion\ is\ detected \end{cases}$$

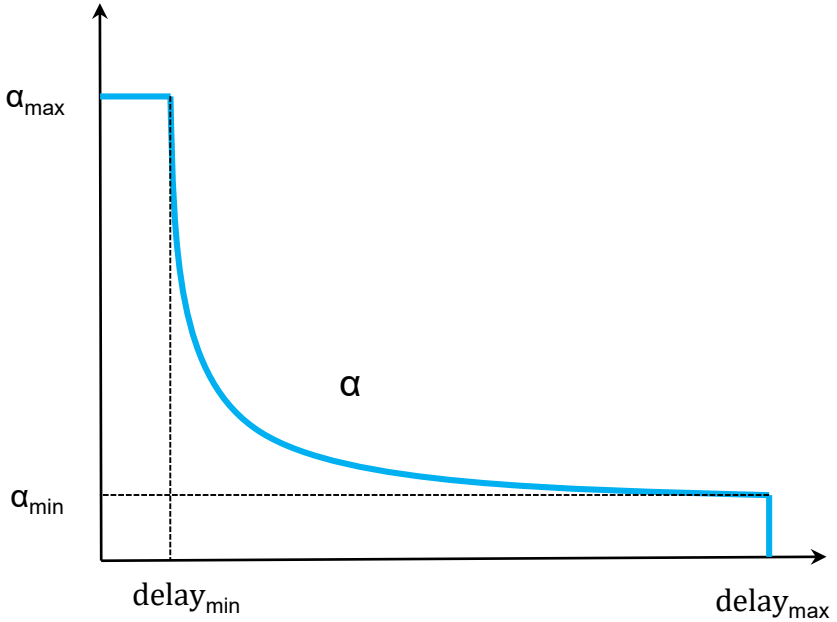
Measure ***min RTT*** and ***max RTT*** for each ACK. Track them.

Compute α:

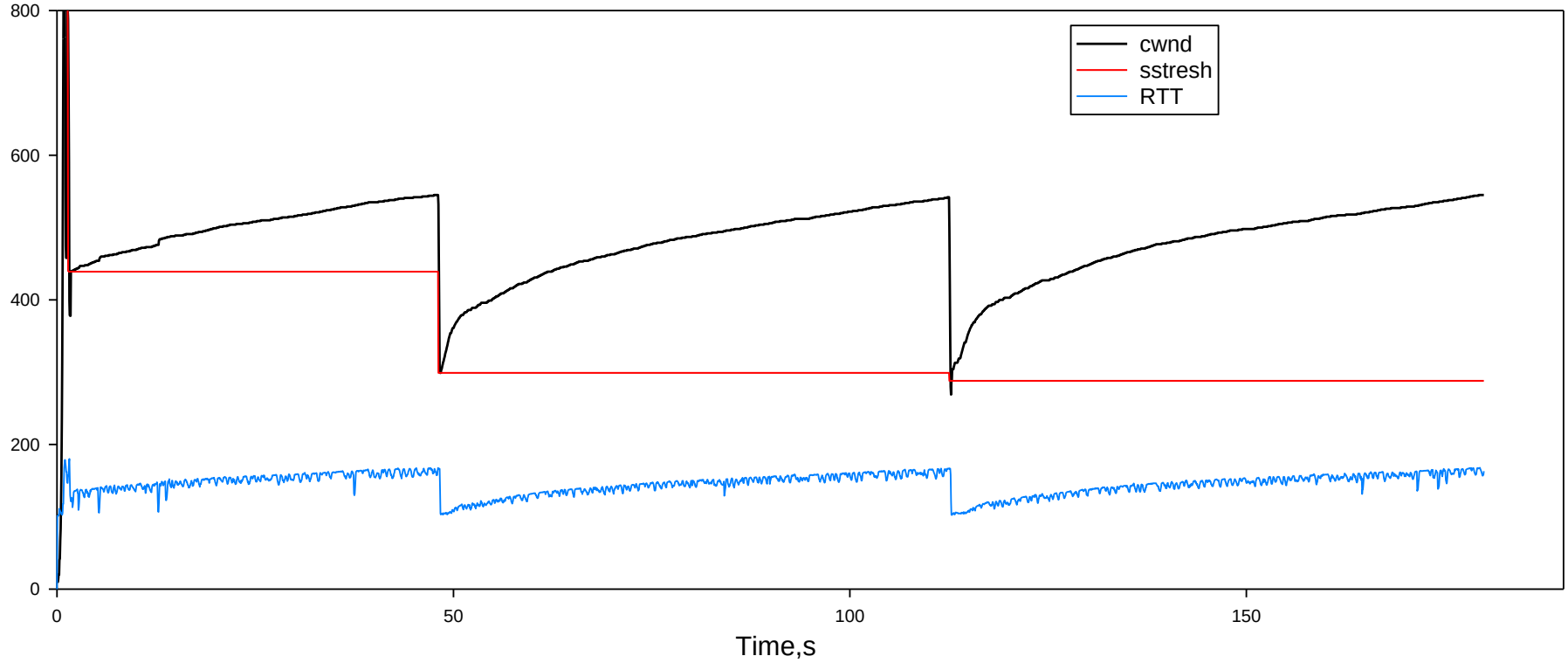
- If average delay is at minimum (we are uncongested), then use large alpha (10.0) to grow ***cwnd*** faster.
- If average delay is at maximum (getting congested) then use small alpha (0.3)

Compute β:

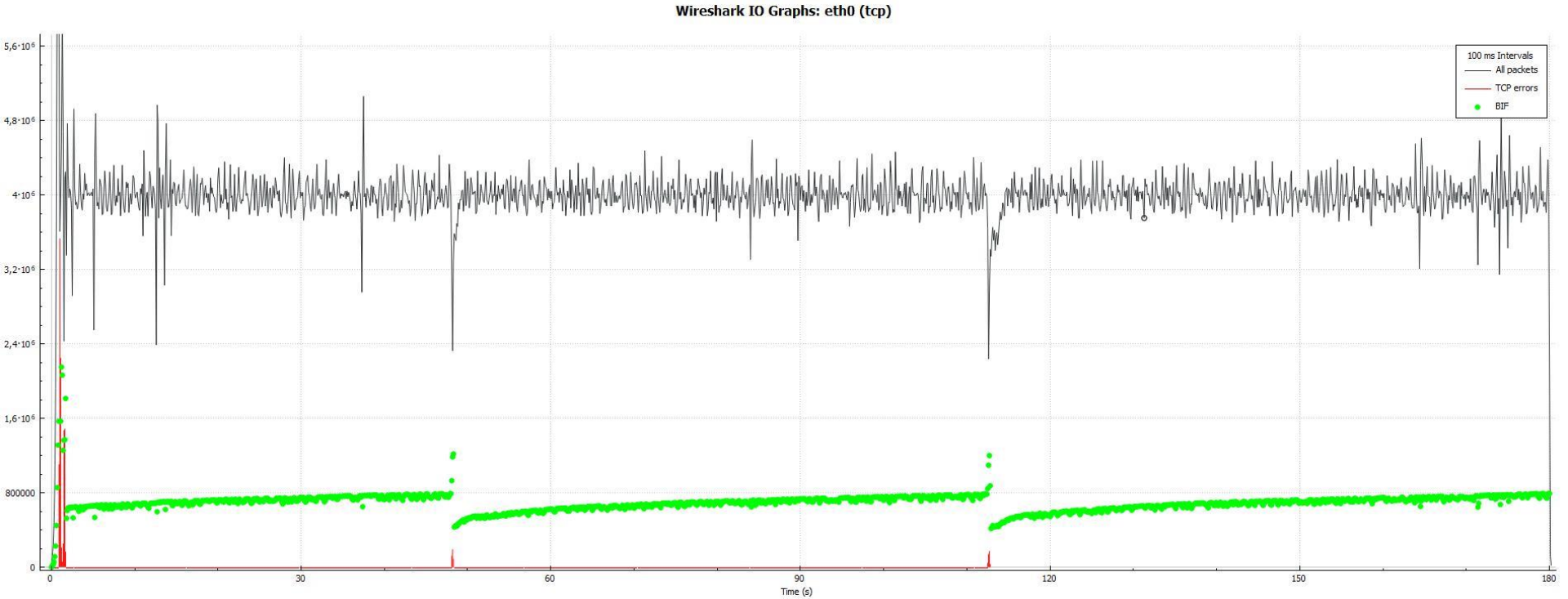
- If delay is small (10% of max) then β = 1/8
- If delay is up to 80% of max then β = 1/2
- In between is a linear function



ILLINOIS TCP

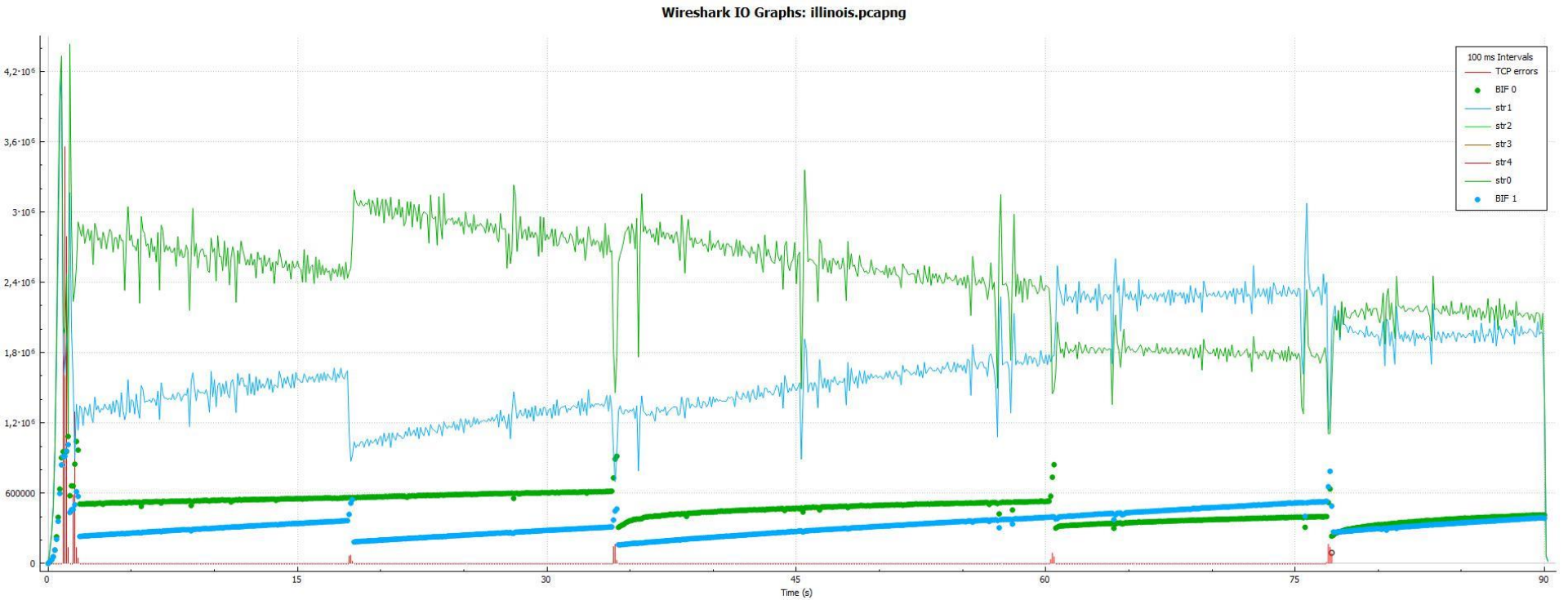


ILLINOIS TCP



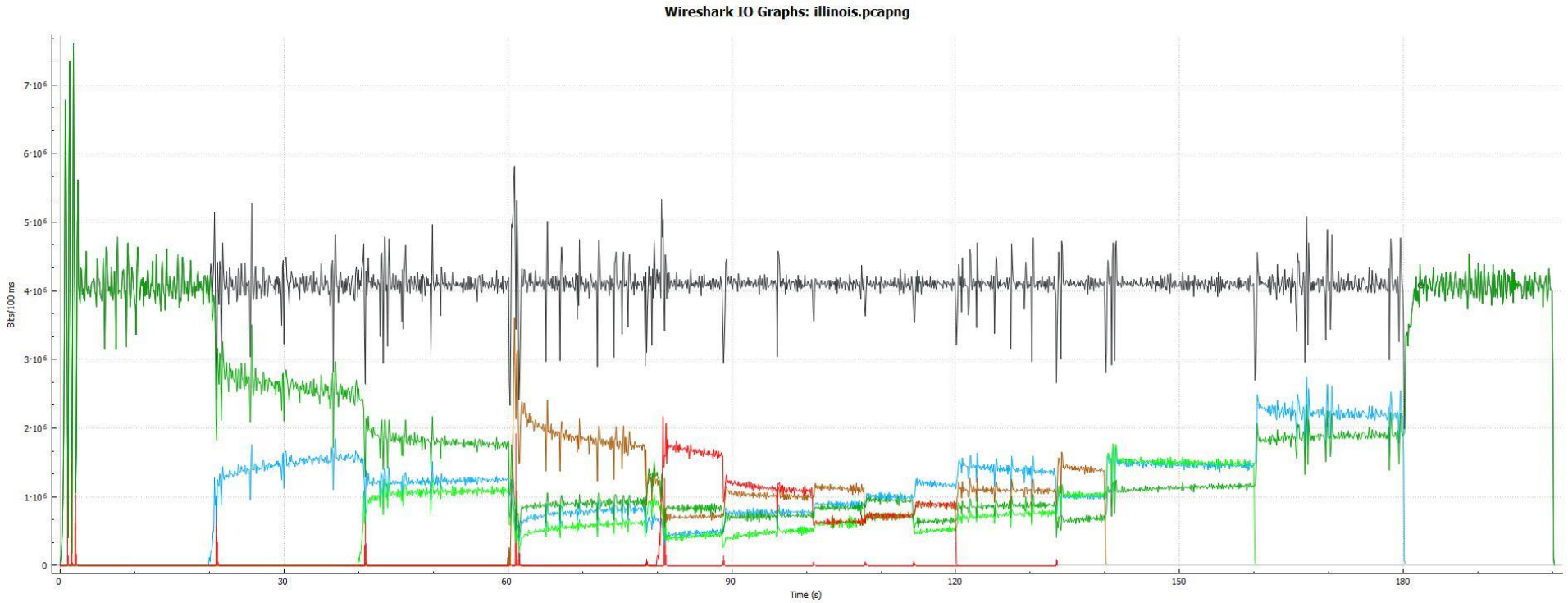
Collateral damage: 2 Buffer overflows / 815k Total Packets

Illinois TCP



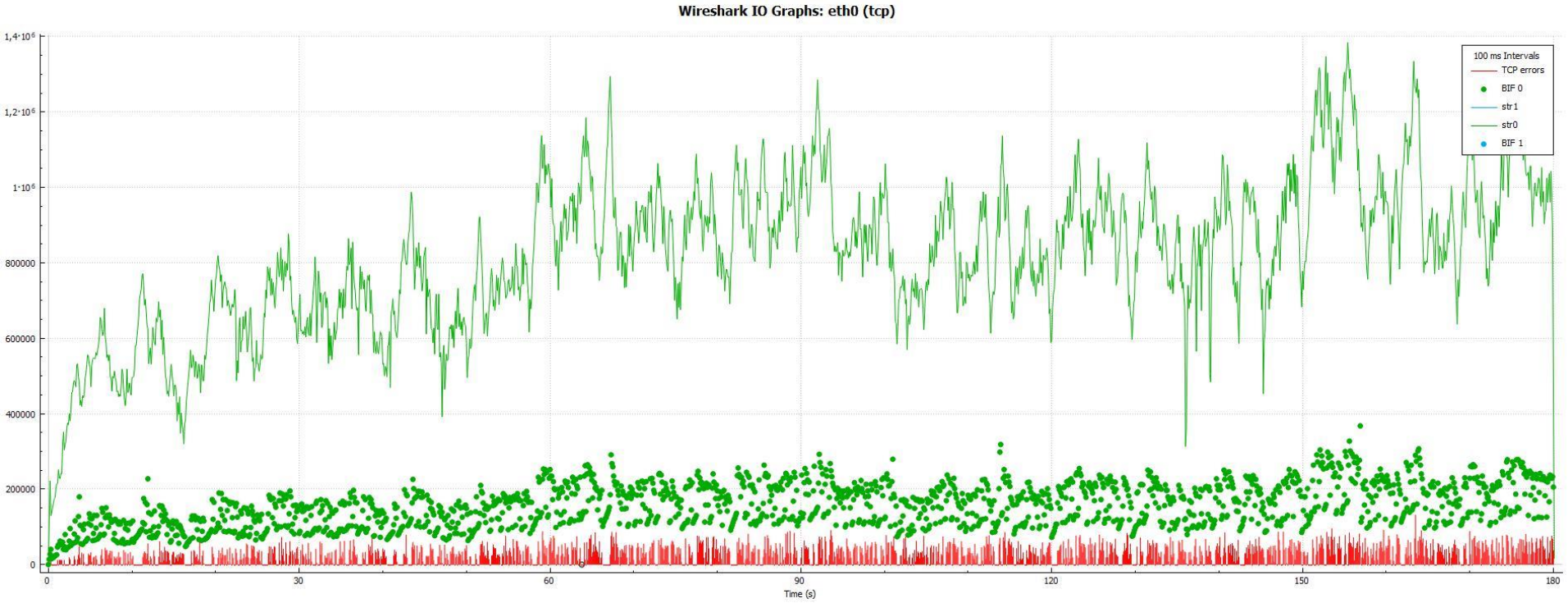
vs. Reno Friendliness

Illinois TCP



5-stream convergence

Illinois TCP



1% loss link behavior

Compound TCP

Core ideas:

"Windows"

1. Uses combination of packet loss and delay as feedback.
2. Uses AIMD additionally altered by delay window as action profile.
3. Only for Windows OS since Vista.

***cwnd* control rules:**

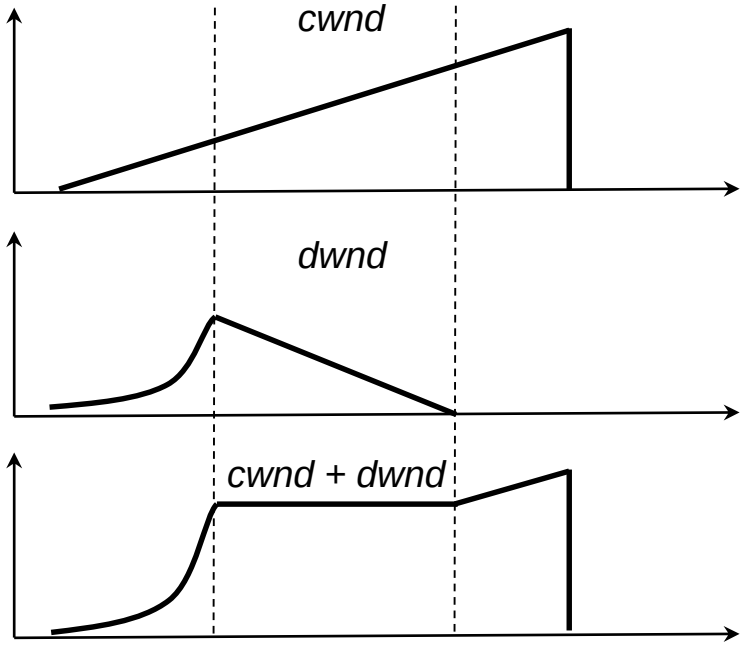
$$win = min\ (cwnd + dwnd,\ awnd)$$

Where: ***cwnd*** – as in Reno, ***dwnd*** – as in Vegas.

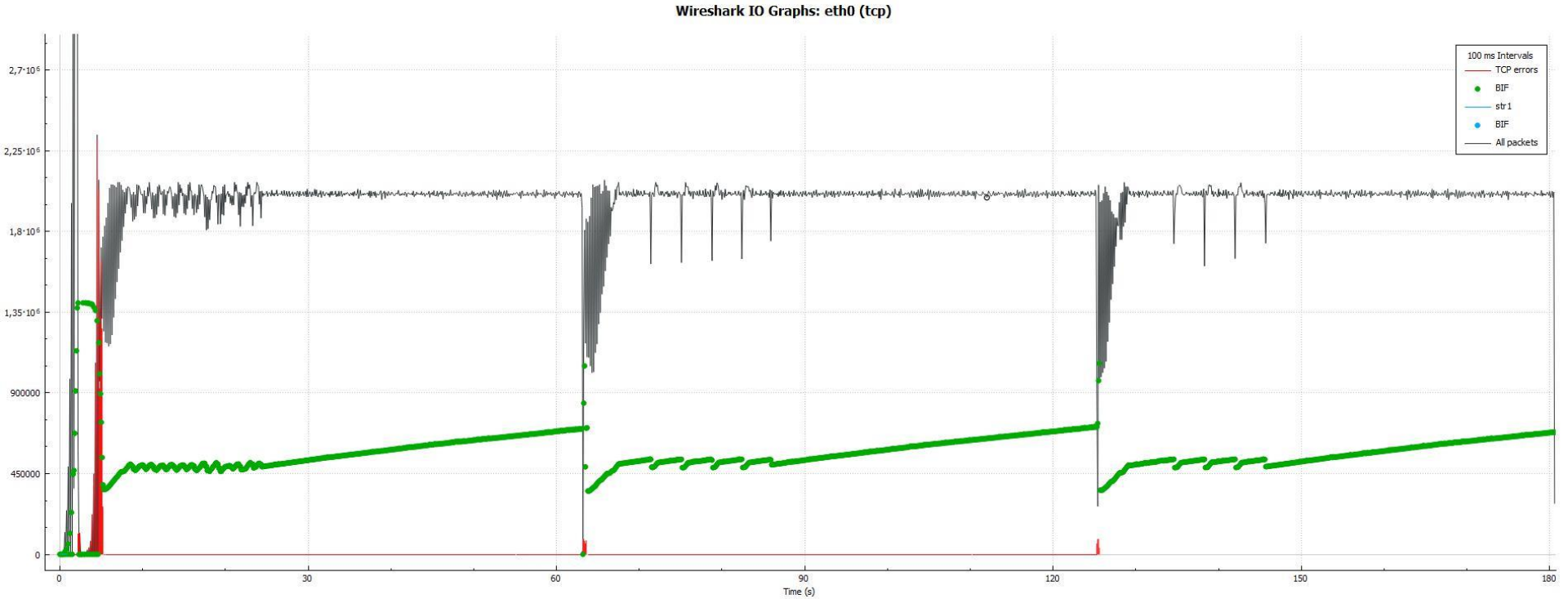
$$cwnd = \begin{cases} cwnd + 1/(cwnd + dwnd) & if\ congestion\ is\ not\ detected \\ (1 - \beta) * cwnd & if\ congestion\ is\ detected \end{cases}$$

Main point: combine fairness of delay-based CA with aggressiveness of loss-based CA.

- ✓ Coexistence with Reno on non-LFN links – good
- ✓ RTT fairness - good



Compound TCP



Link: 20mpbs, 200ms RTT. Tested using ntttcp.exe, Win10 – Win10. Sorry, no ***cwnd*** graph..

WESTWOOD TCP

Core ideas:

"Wireless warrior"

Source

1. Main idea: an attempt to distinguish between congestive and non-congestive losses.
2. Uses packet loss as feedback.
3. Uses modified AIMD as action profile.
4. Continuously estimates bandwidth (BWE, from incoming ACKs) and minimal RTT (RTT_{noLoad})

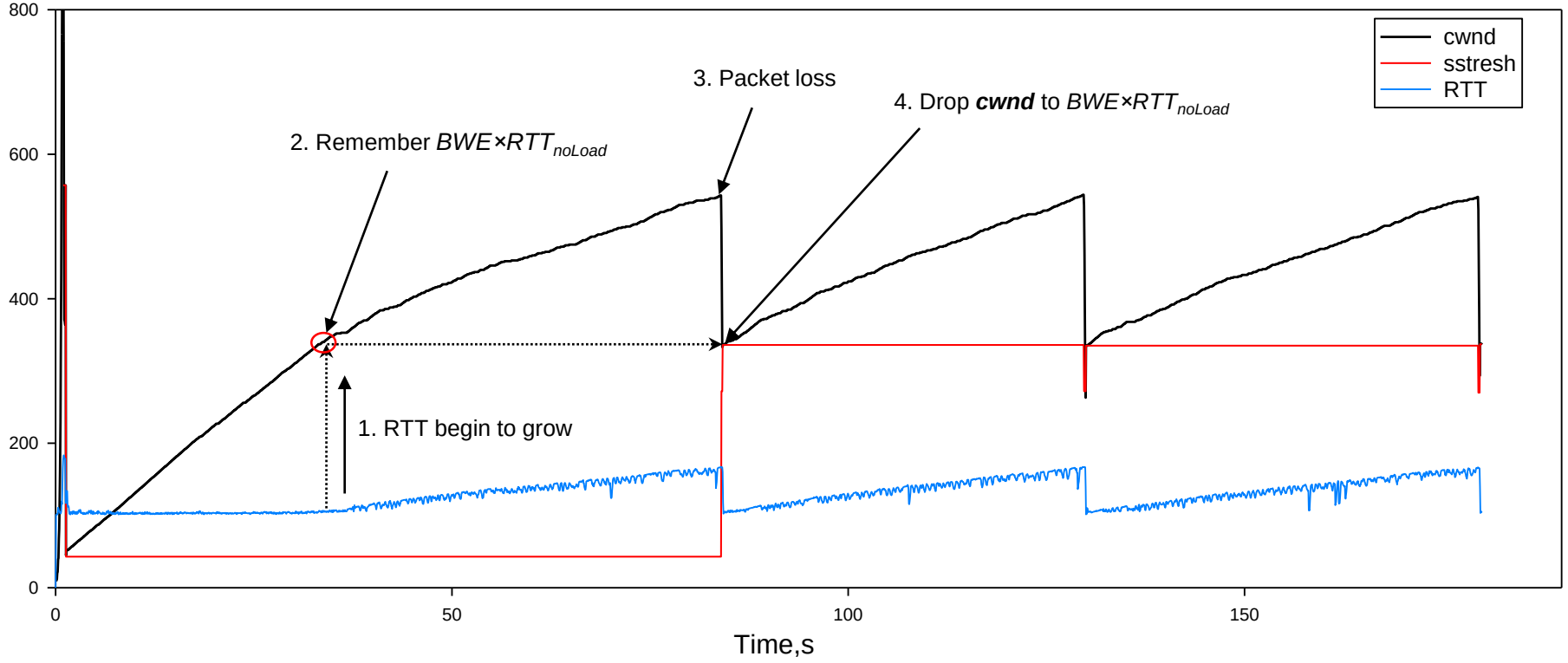
***cwnd* control rules:**

- Calculates "transit capacity" : $BWE \times RTT_{noLoad}$ (represents how many packets can be in transit)
- Never drops ***cwnd*** below estimated transit capacity.

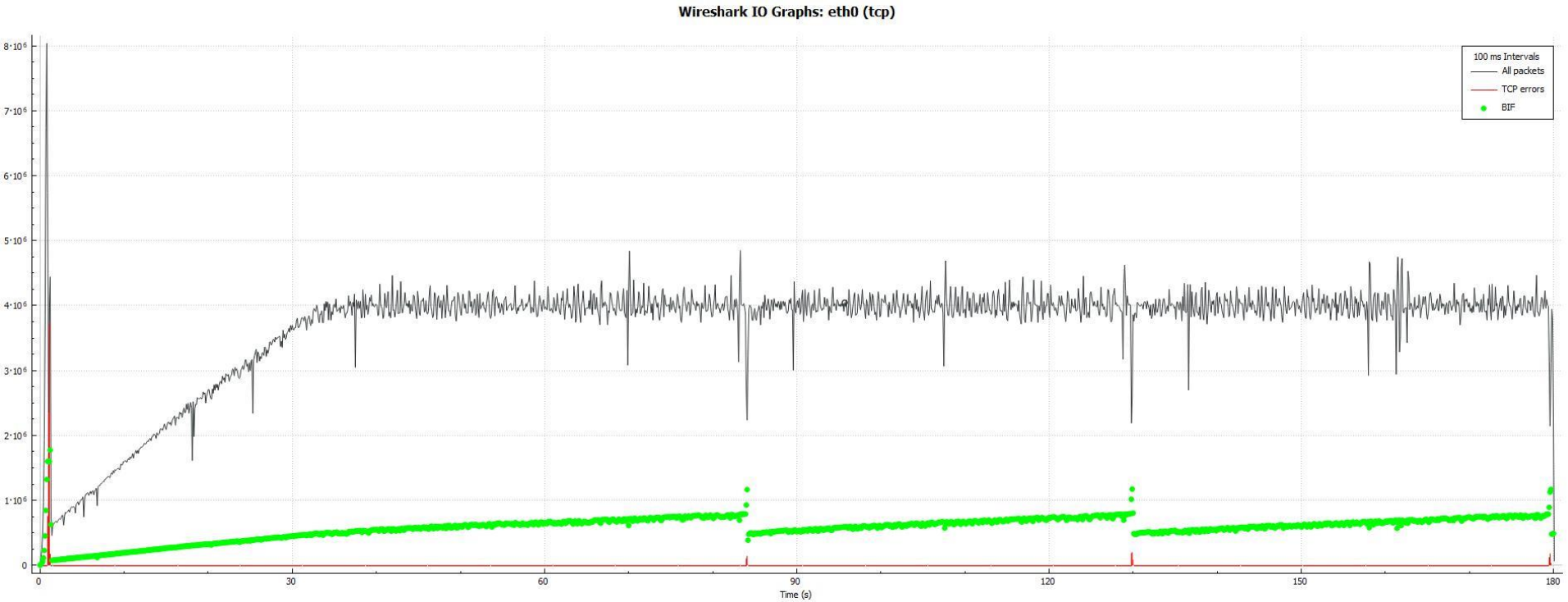
cwnd (on loss) = $\begin{cases} max(cwnd/2, BWE \times RTTnoLoad) \; if \; cwnd > BWE \times RTTnoLoad \\ no \; change, \; if \; cwnd \leq BWE \times RTTnoLoad \end{cases}$

- If no loss is observed – acts similarly to Reno.

WESTWOOD TCP

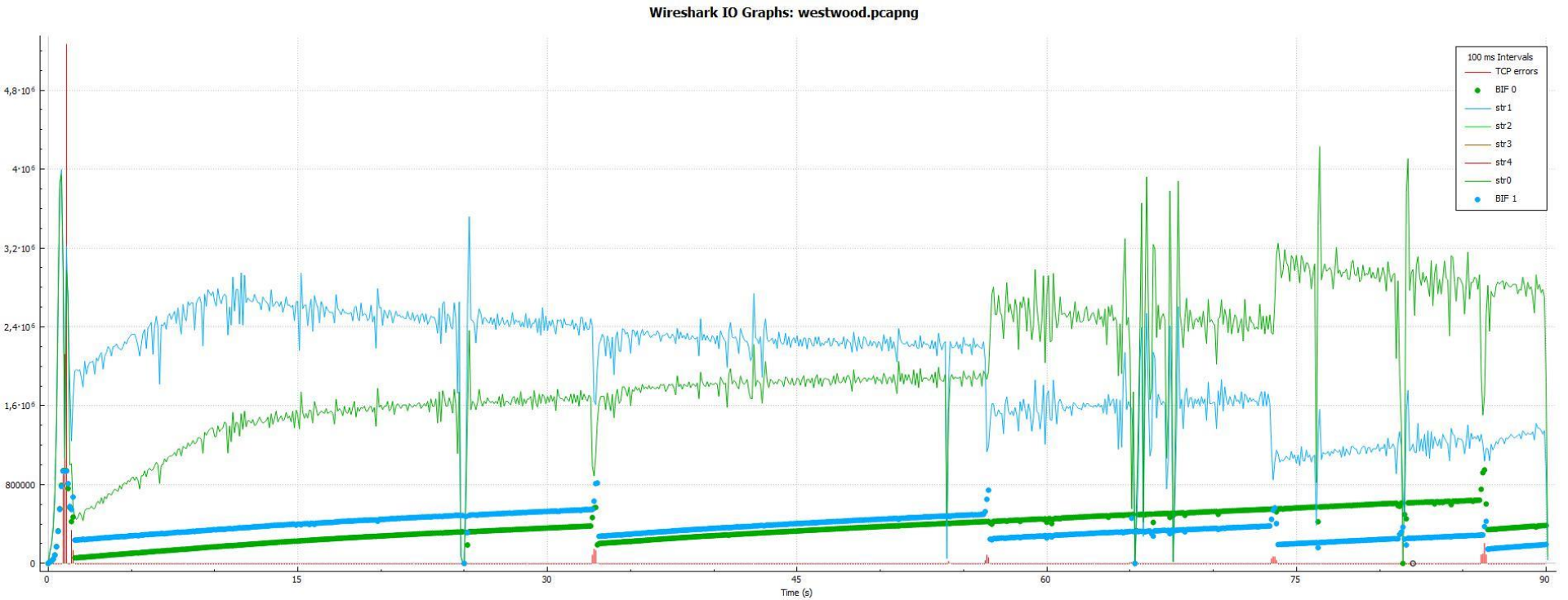


WESTWOOD TCP



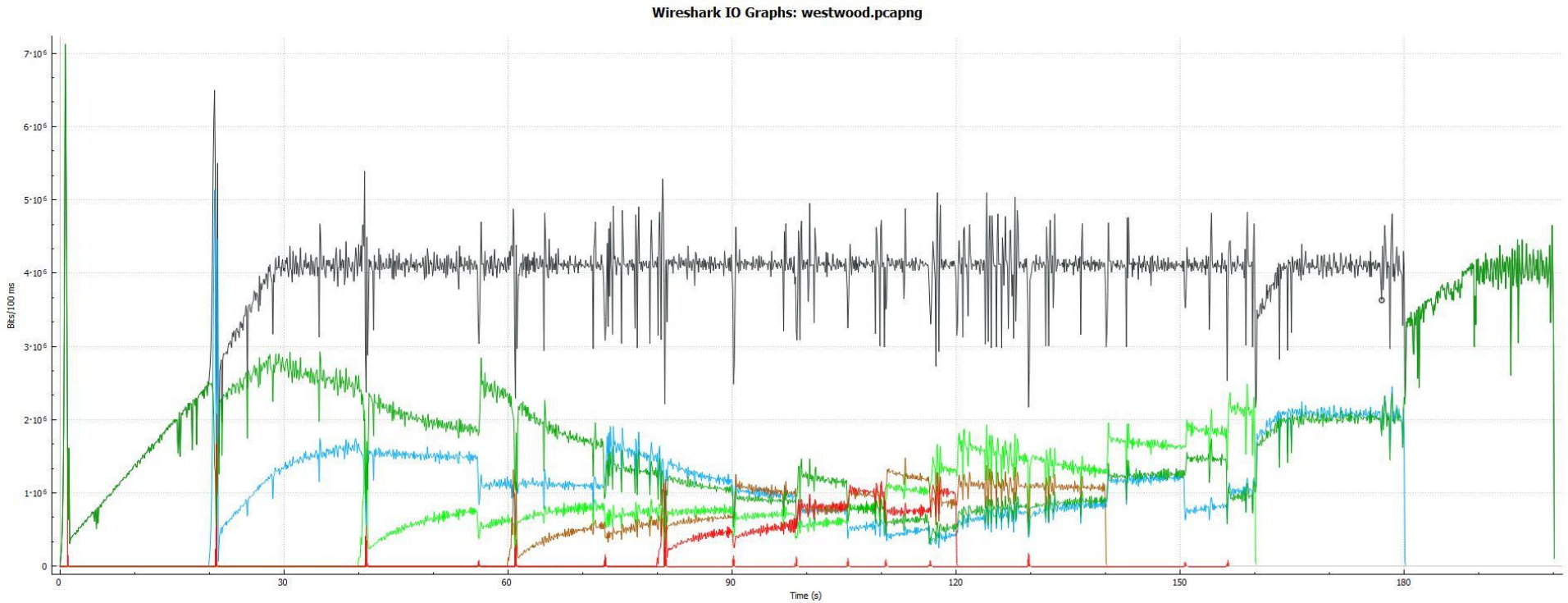
Collateral damage – same as Reno. Good for lossy links. With 1% loss beats CUBIC with x5 factor.

WESTWOOD TCP



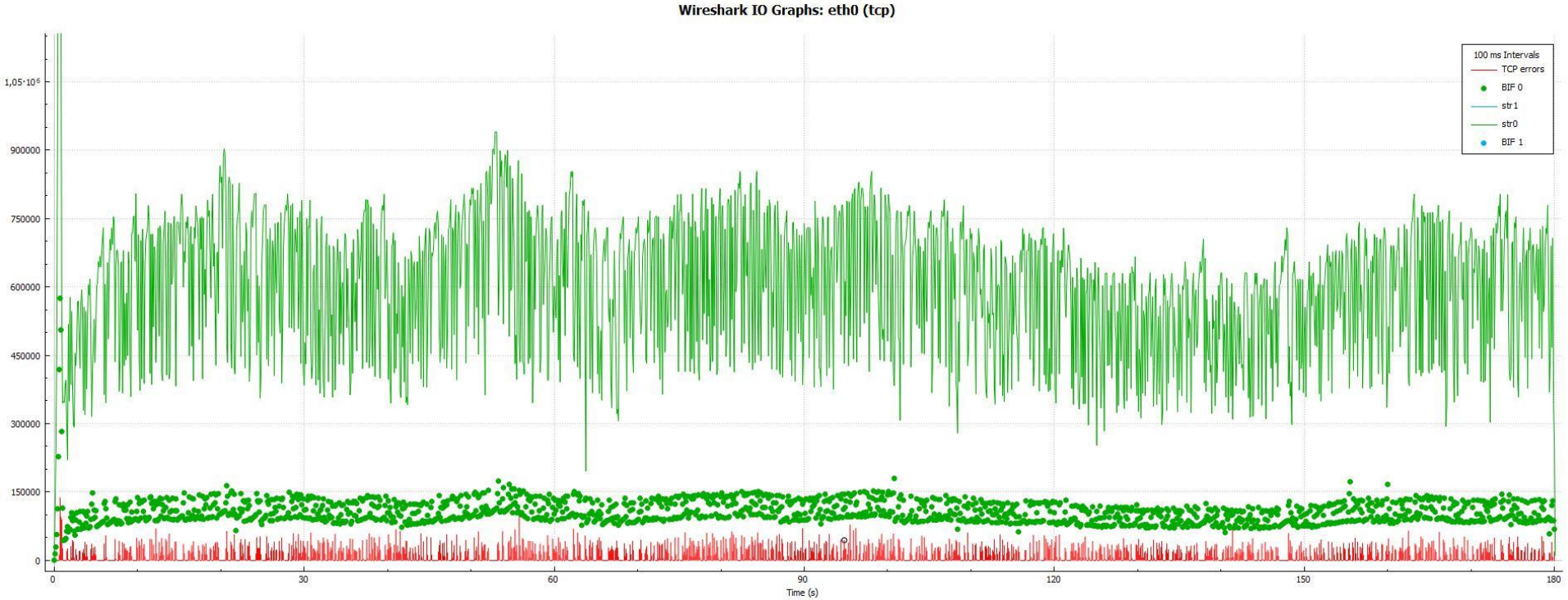
vs. Reno Friendliness

WESTWOOD TCP



5-stream convergence

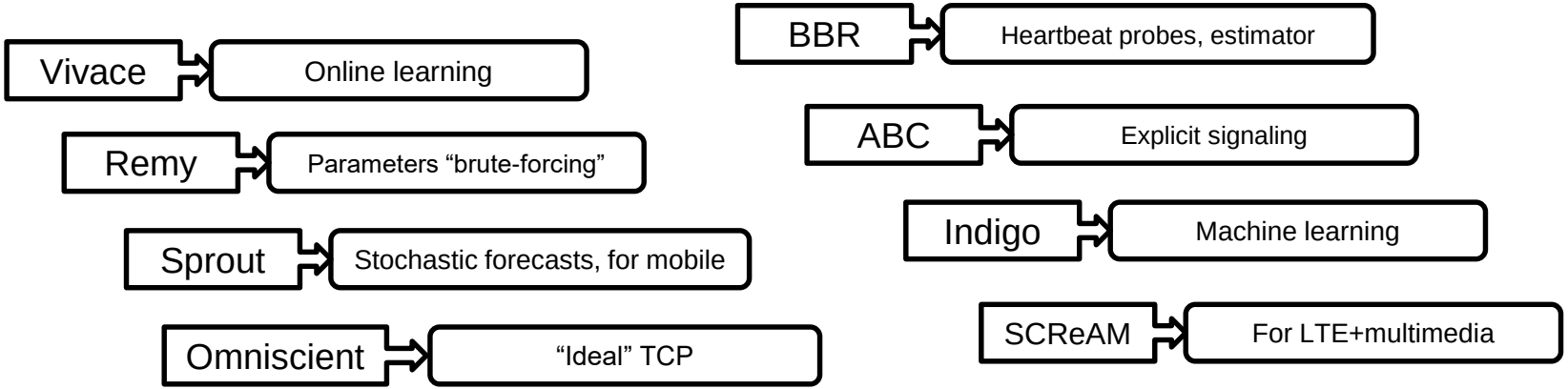
WESTWOOD TCP



1% loss link behavior

The future?

- Multiple signals (ACK inter-arrival time, timestamps, delay with minimal RTT value tracking, packet loss).
- Learning-based (the use of assumption model).
- No pure ACK clocking (switch to combination of ACK clocking + pacing model). ***cnwd*** + ***time gap*** from last sent packet.
- Integrating into other protocols (PCC, QUIC – on top of UDP).
- Pushing CA into user-space + using API (concept, Linux).
- Reinventing SlowStart (Flow-start, “Paced Chirping”, now more for datacenter environment).



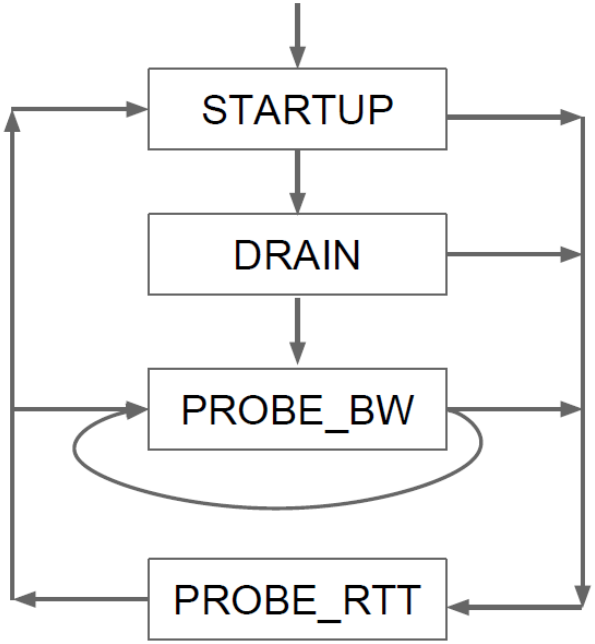
BBR TCP

Core ideas: **"Heartbeat"** Source

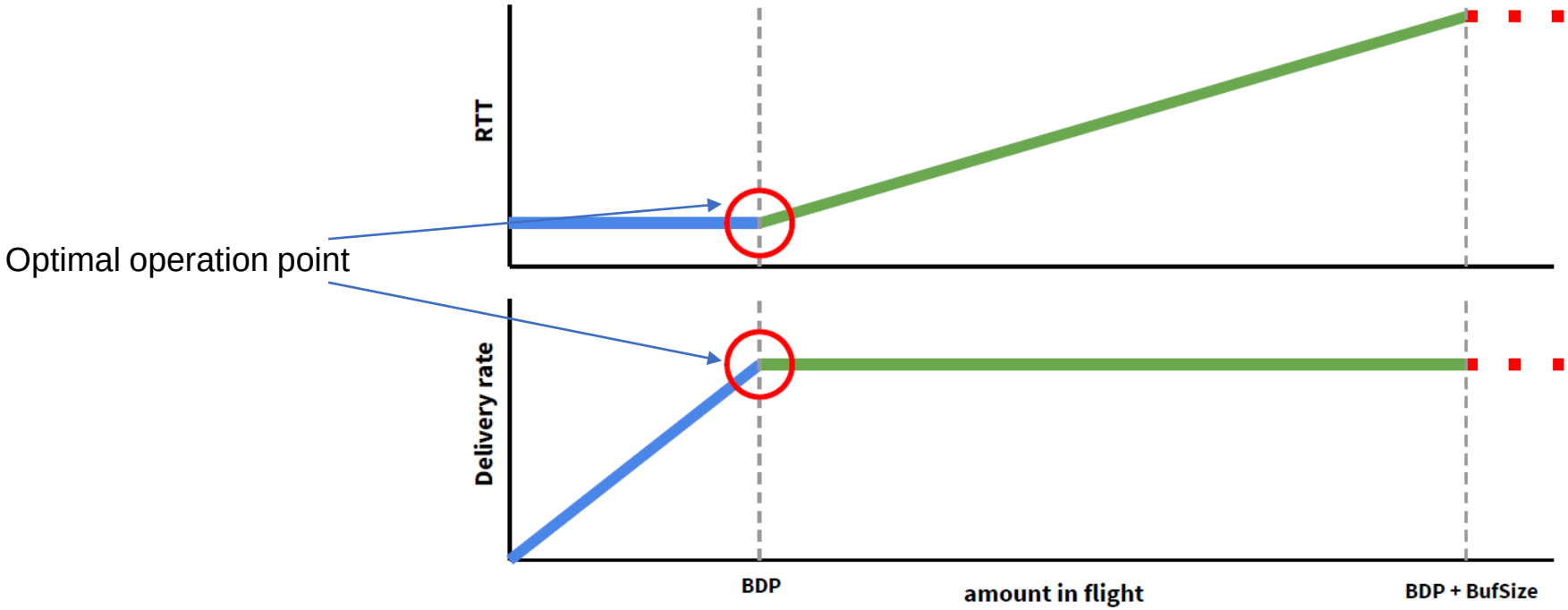
1. RTT and Bottleneck BW estimation ***(RTprop*** and ***BtlBw*** variables***)*** + active probing.
2. Uses periodic spike-looking active probes (+/- 25%) for Bottleneck BW testing.
3. Uses periodic pauses for "Base RTT" measuring.
4. Tracks App-limited condition (nothing to send) to prevent underestimation.
5. Doesn't use AIMD in any form or shape. Uses pacing instead. Can handle sending speeds from 715bps to 2,9Tbps.

***cwnd* control rules - 4 different phases:**

- **Startup** (beginning of the connection)
- **Drain** (right after startup)
- **Probe_BW** (most of the time)
- **Probe_RTT** (periodically every 10 seconds)

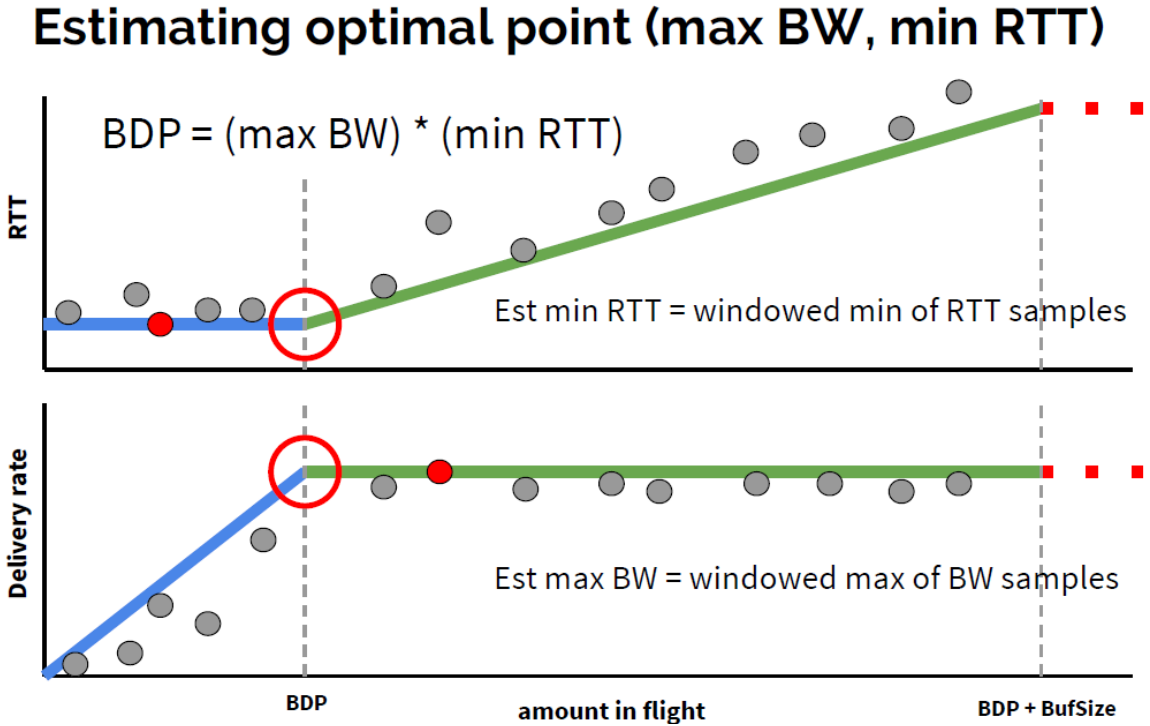


BBR TCP



* From “Making Linux TCP Fast”. Yuchung Cheng. Neal Cardwell. Netdev 1.2 Tokyo, October, 2016”

BBR TCP

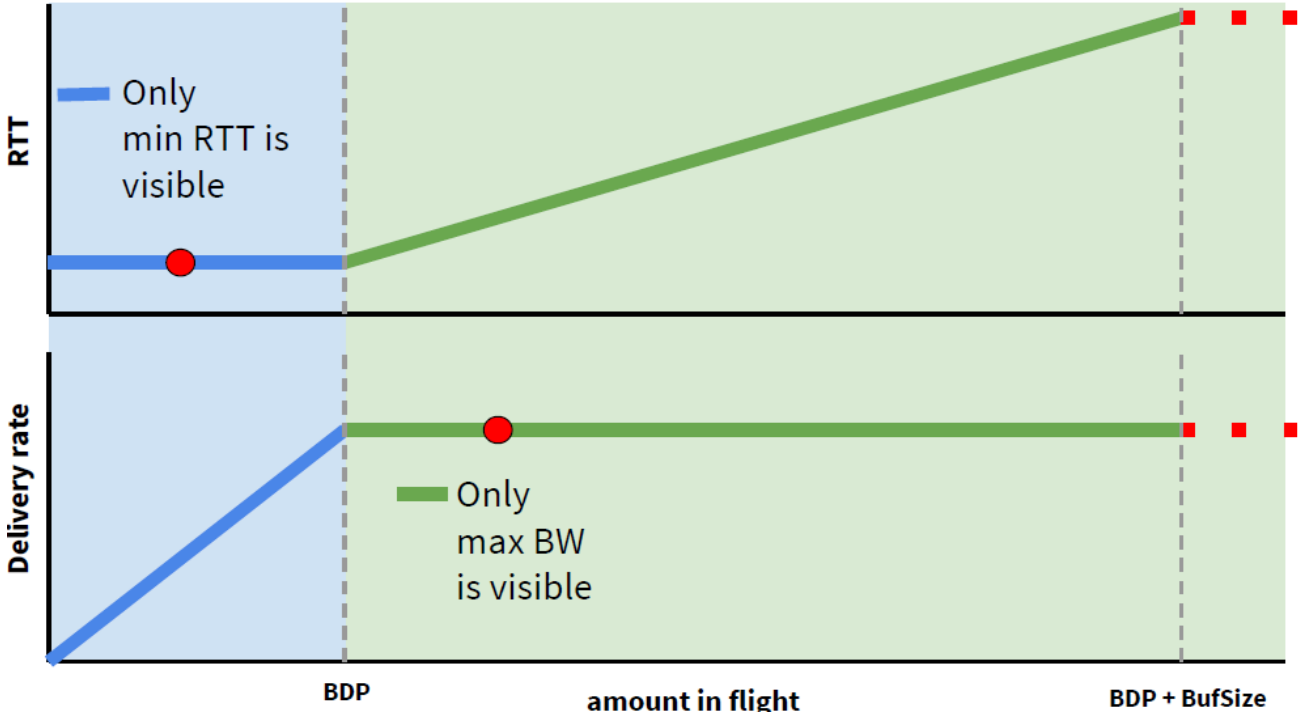


* From "Making Linux TCP Fast". Yuchung Cheng. Neal Cardwell. Netdev 1.2 Tokyo, October, 2016"

BBR TCP

Uncertainty principle:
We can not estimate max BW and min RTT at the same time (point)!

Solution: well, let's do it sequentially!

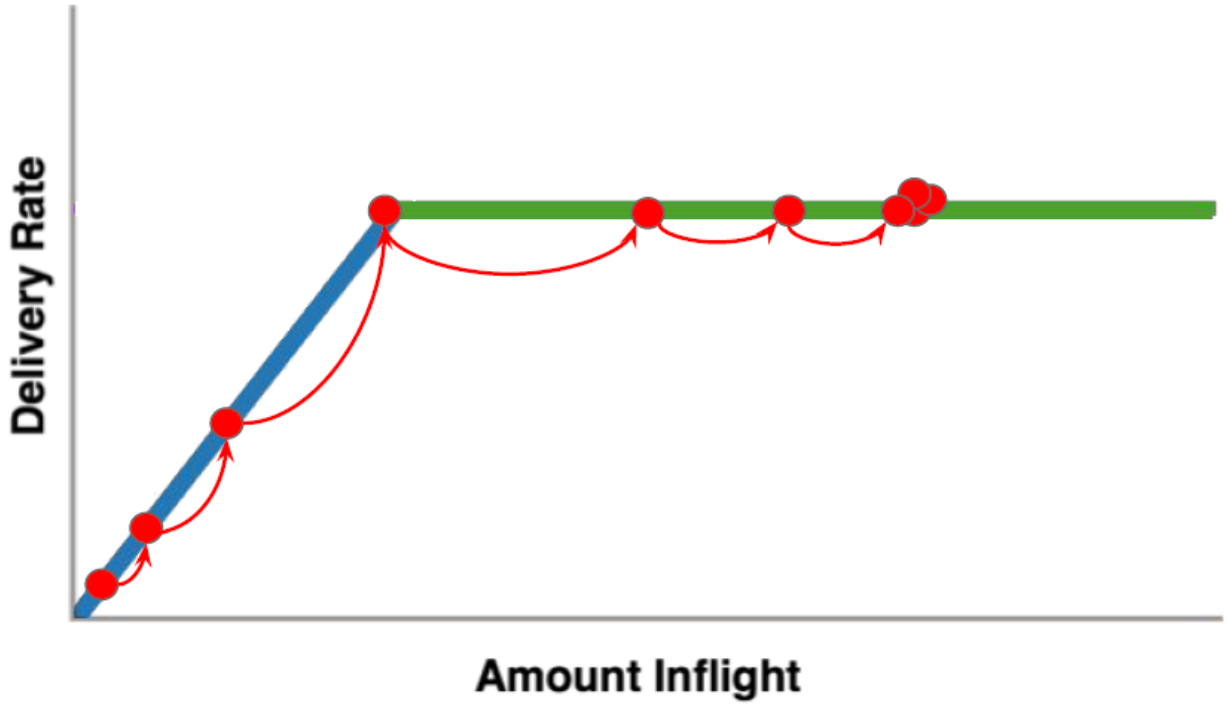


* From "Making Linux TCP Fast". Yuchung Cheng. Neal Cardwell. Netdev 1.2 Tokyo, October, 2016"

BBR TCP

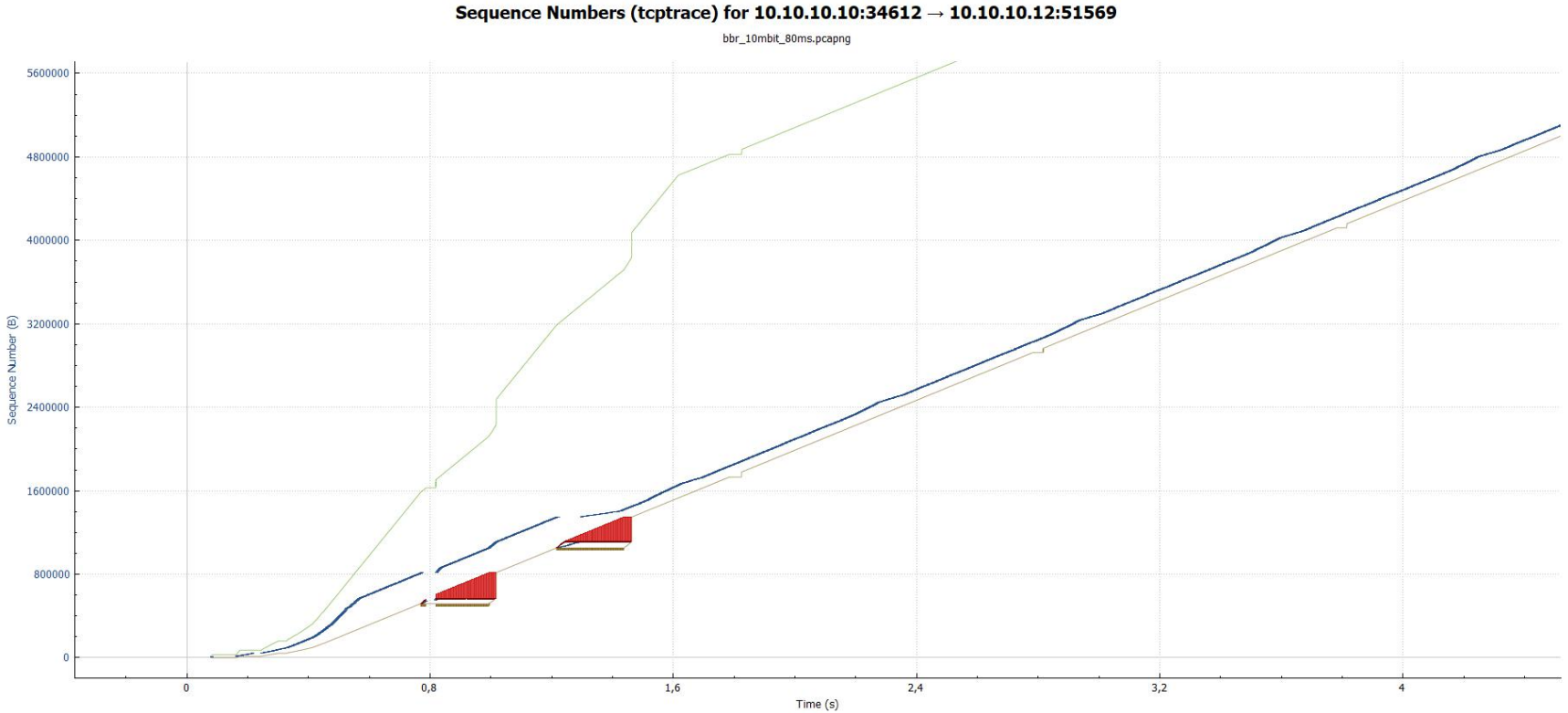
<u>Startup phase:</u> exponential probe for max BW.

Stopped if BW growth is less than 25% for 3 sequential probes.



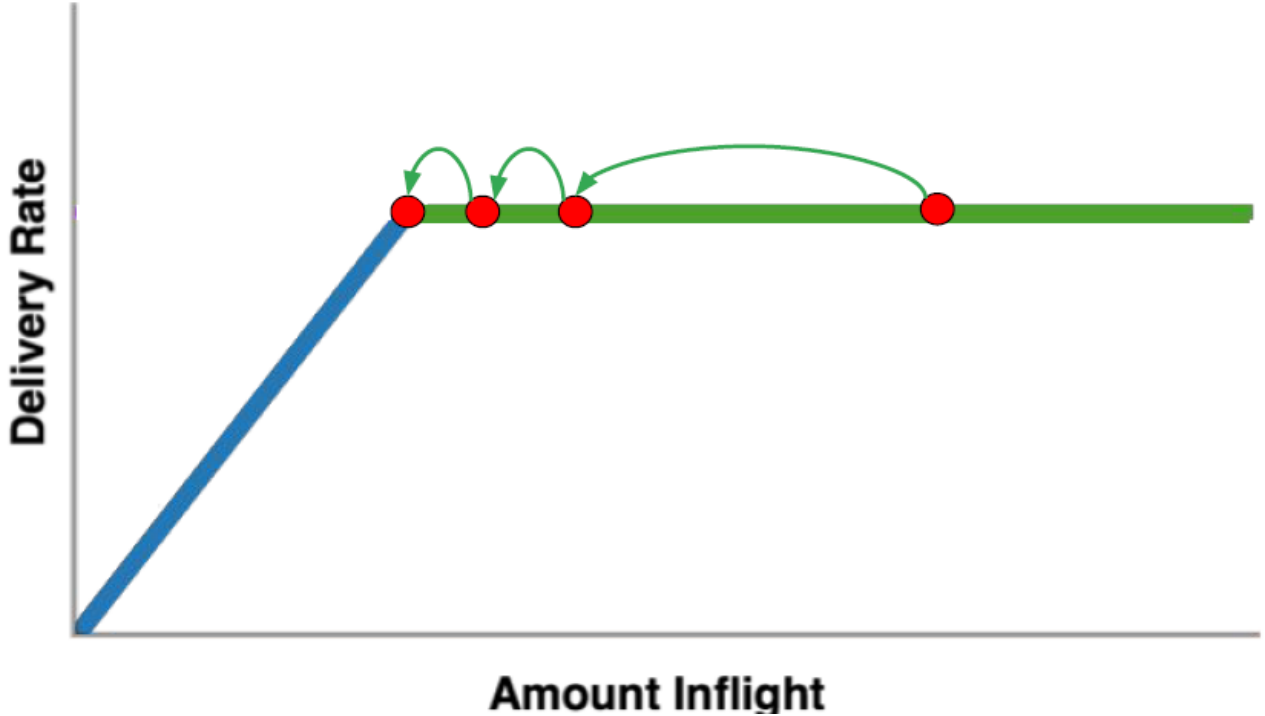
*** From "Making Linux TCP Fast". Yuchung Cheng. Neal Cardwell. Netdev 1.2 Tokyo, October, 2016"**

BBR TCP


Sequence Numbers (tcptrace) for 10.10.10.10:34612 → 10.10.10.12:51569
bbr_10mbit_80ms.pcapng
Sequence Number (B)
5600000
4800000
4000000
3200000
2400000
1600000
800000
0
0
0,8
1,6
2,4
3,2
4
Time (s)

BBR TCP

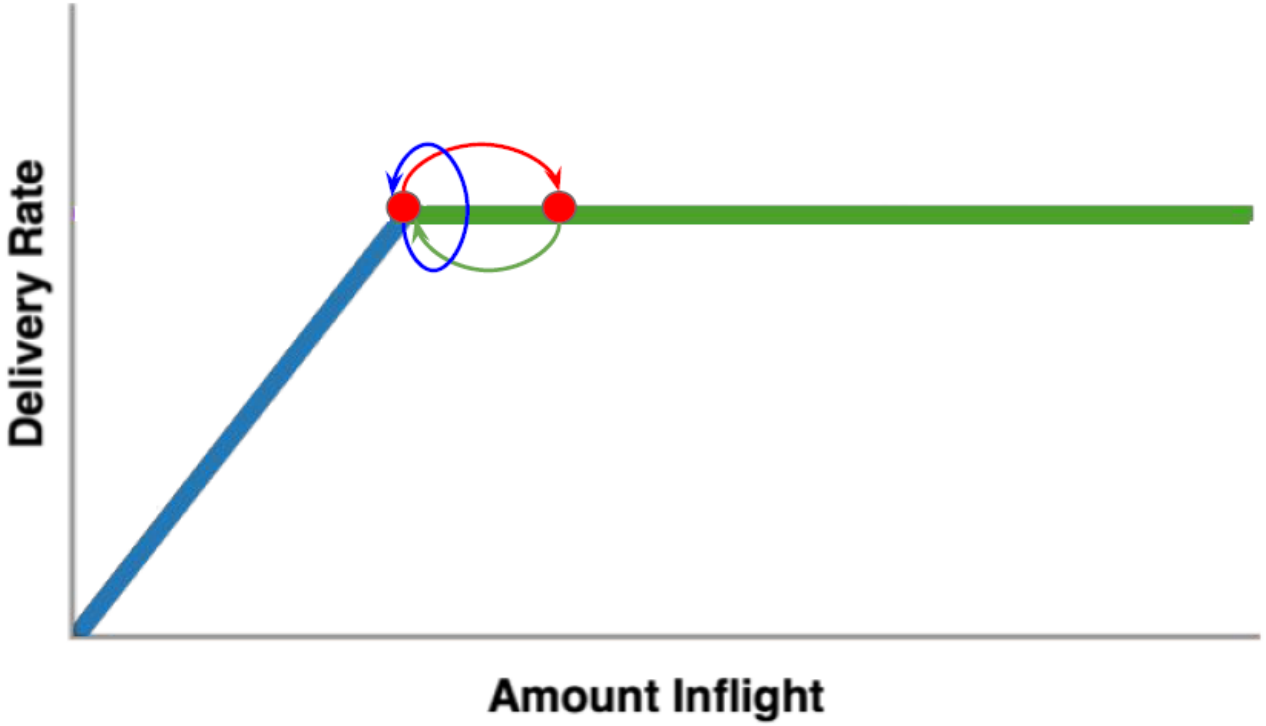
Drain phase: trying to get rid of queue formed during startup phase.



* From “Making Linux TCP Fast”. Yuchung Cheng. Neal Cardwell. Netdev 1.2 Tokyo, October, 2016”

BBR TCP

Probe BW phase:
do spikes in sending rate (1,25 followed by 0.75 gains, each one of RTT length)

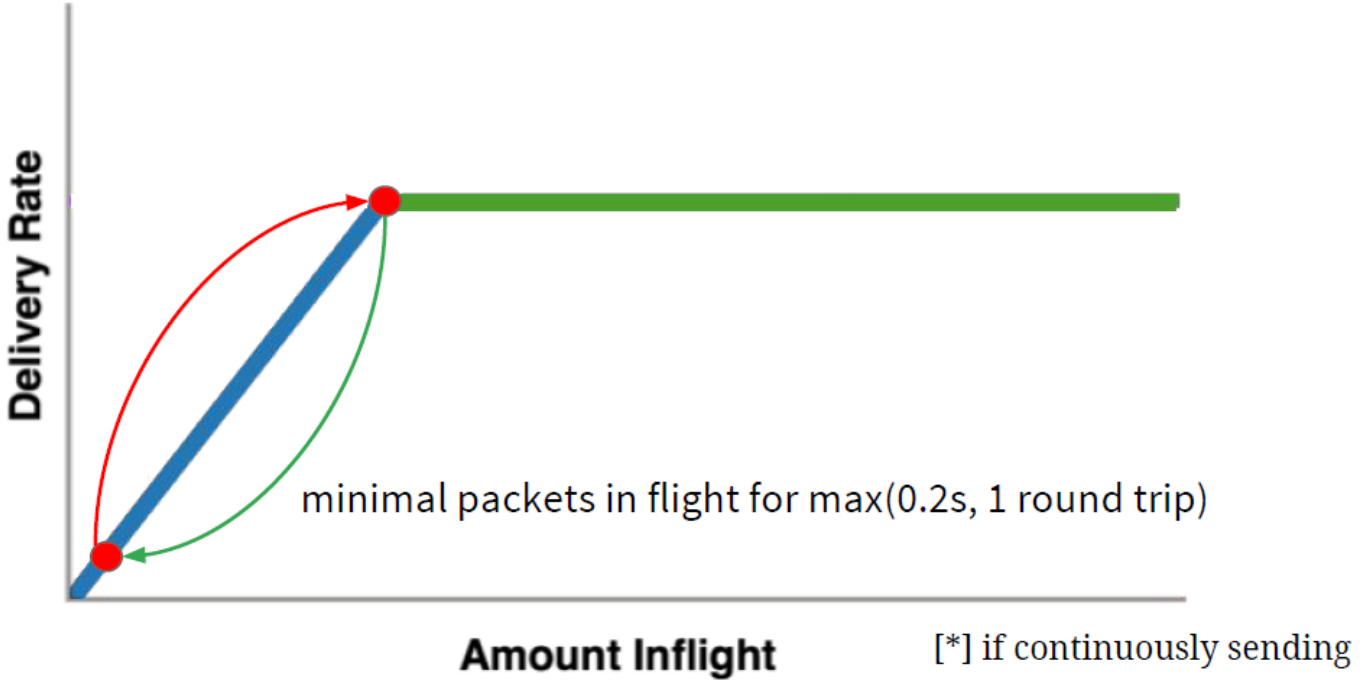


*** From "Making Linux TCP Fast". Yuchung Cheng. Neal Cardwell. Netdev 1.2 Tokyo, October, 2016"**

BBR TCP

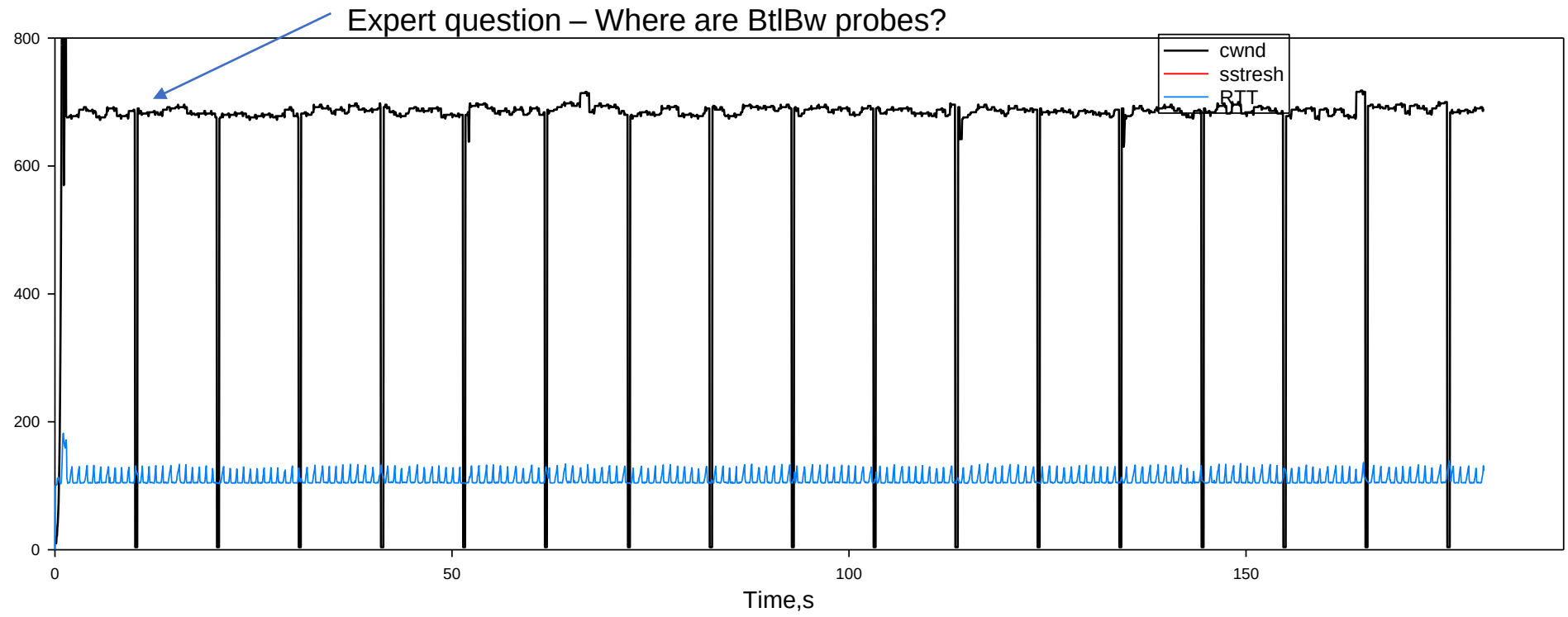
Probe RTT phase:

drop ***cwnd*** to 4 for 0,2 sec
every 10 sec

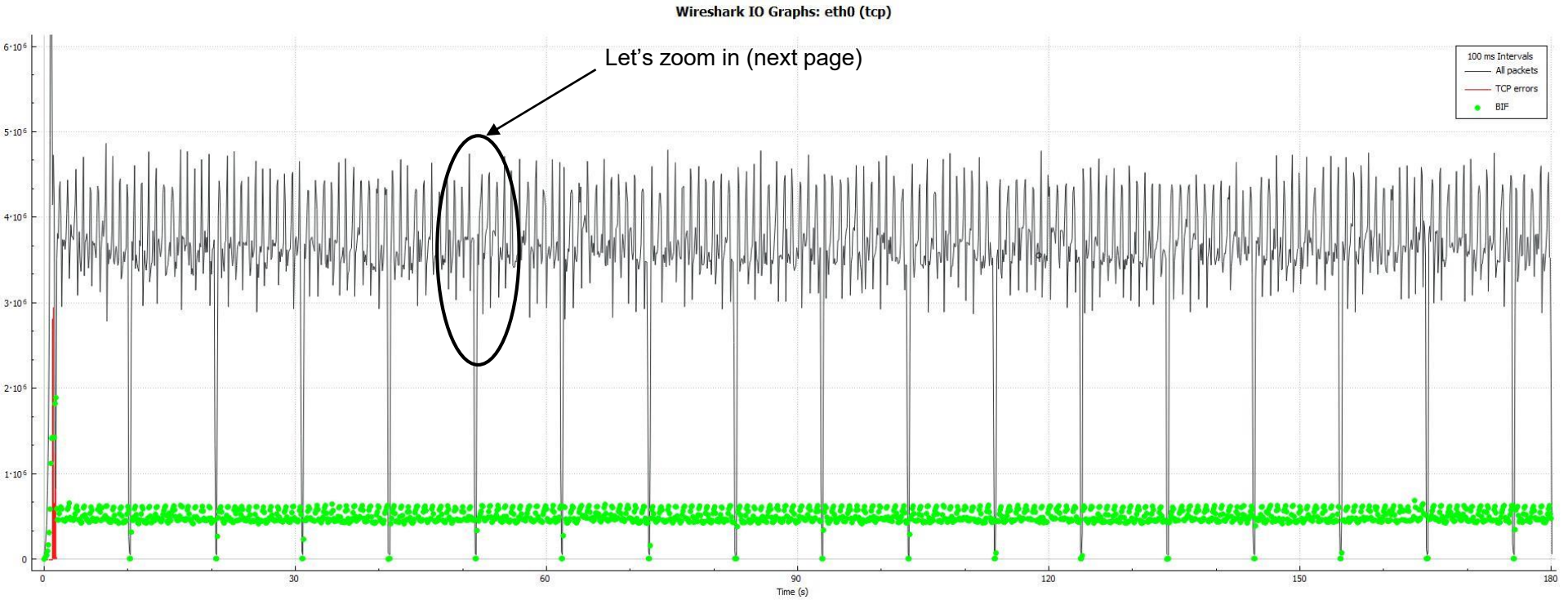


*** From "Making Linux TCP Fast". Yuchung Cheng. Neal Cardwell. Netdev 1.2 Tokyo, October, 2016"**

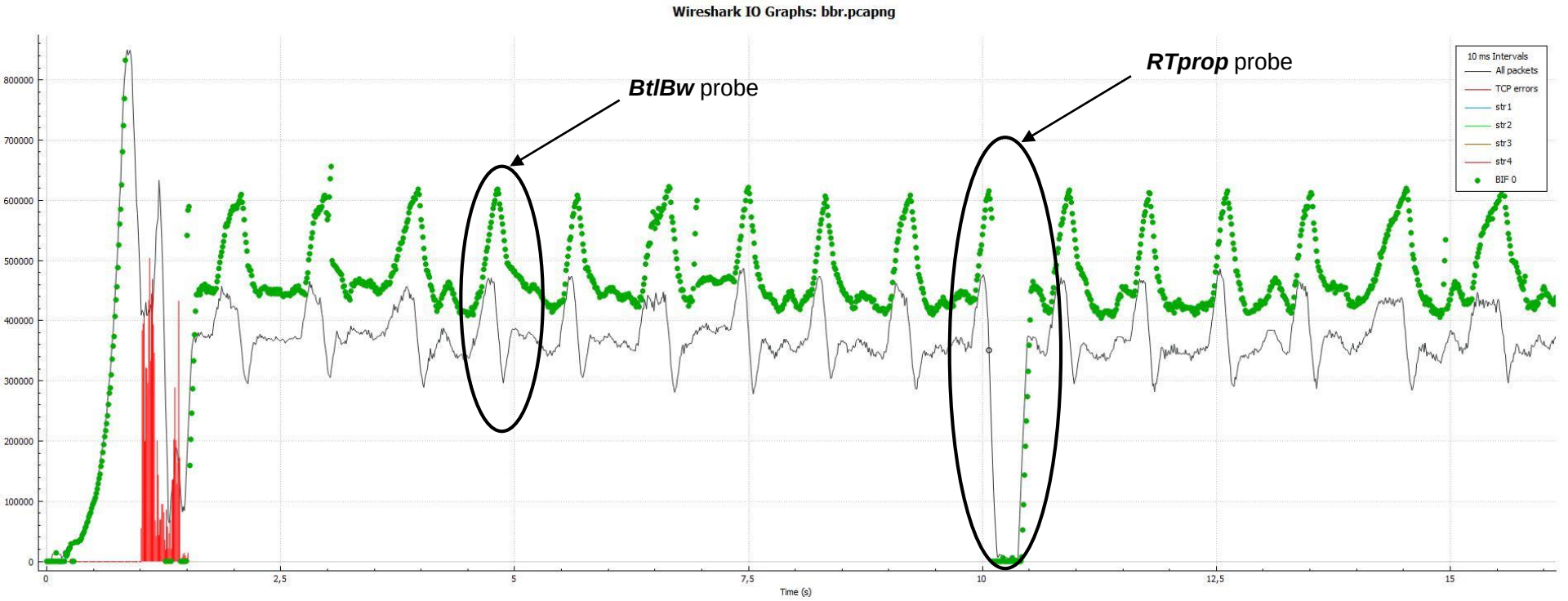
BBR TCP


Expert question – Where are BtlBw probes?
cwnd
sstresh
RTT
800
600
400
200
0
0
50
100
150
Time,s

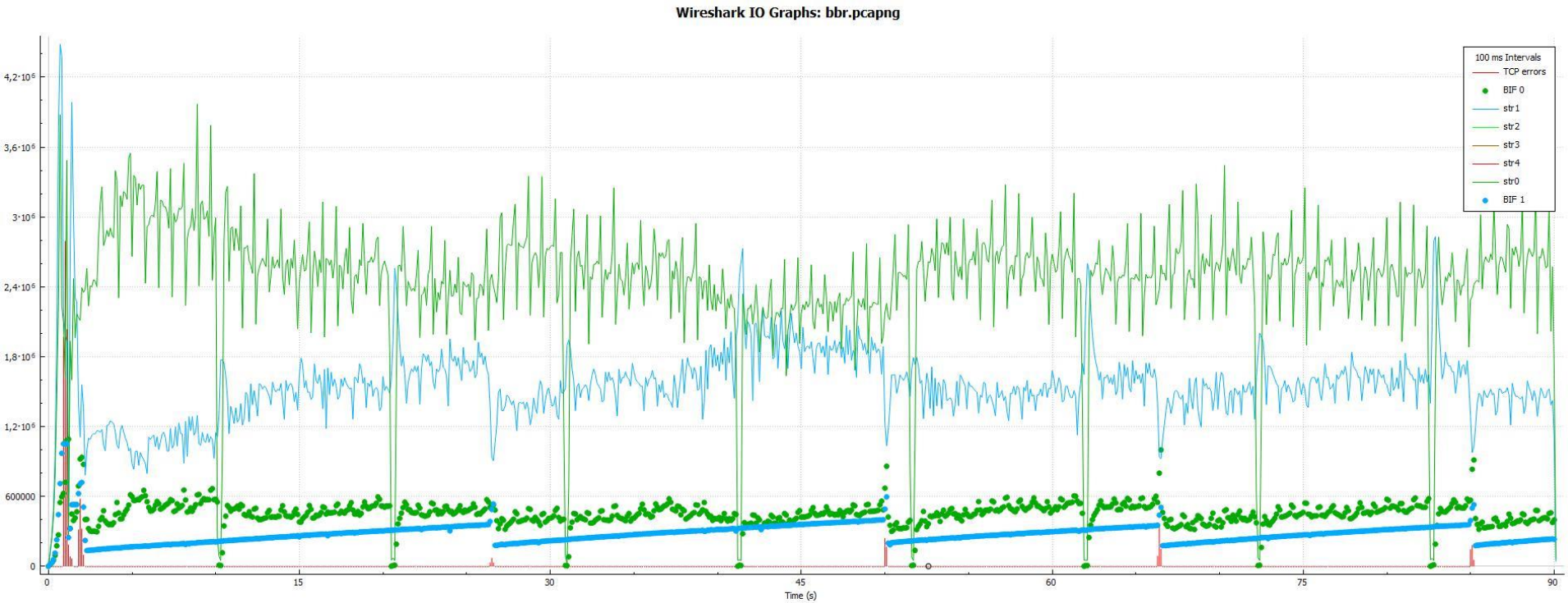
BBR TCP


Wireshark IO Graphs: eth0 (tcp)
Let's zoom in (next page)
100 ms Intervals
All packets
TCP errors
BIF
6·10⁶
5·10⁶
4·10⁶
3·10⁶
2·10⁶
1·10⁶
0
0
30
60
90
120
150
180
Time (s)

BBR TCP

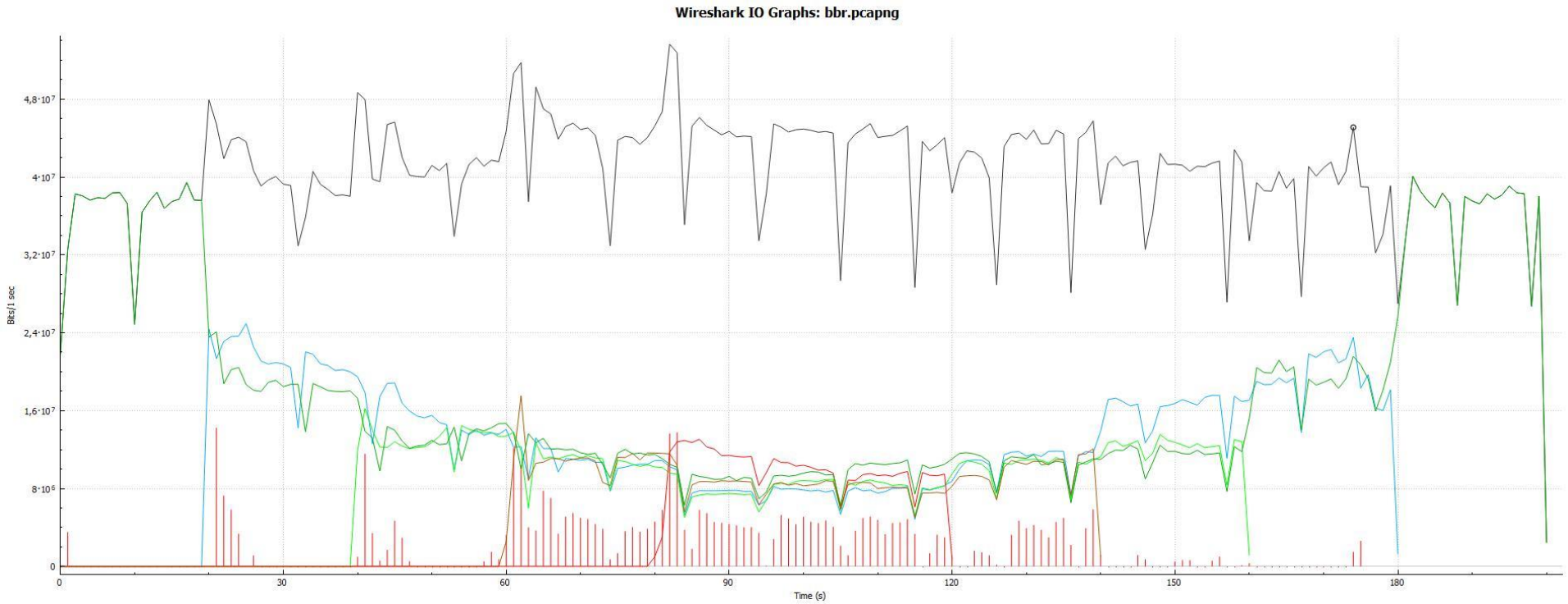

Wireshark IO Graphs: bbr.pcapng
BtlBw probe
RTprop probe
10 ms Intervals
All packets
TCP errors
str 1
str 2
str 3
str 4
BIF 0
800000
700000
600000
500000
400000
300000
200000
100000
0
0
2,5
5
7,5
10
12,5
15
Time (s)

BBR TCP



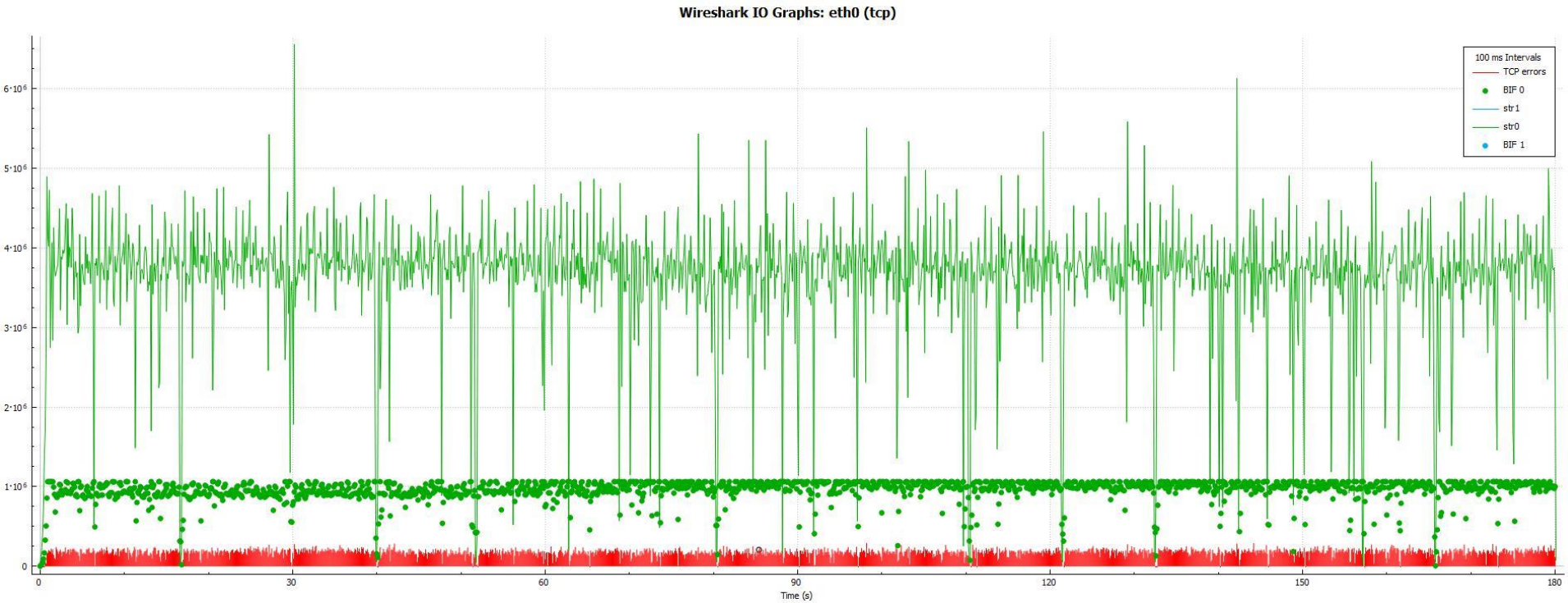
vs. Reno Friendliness

BBR TCP



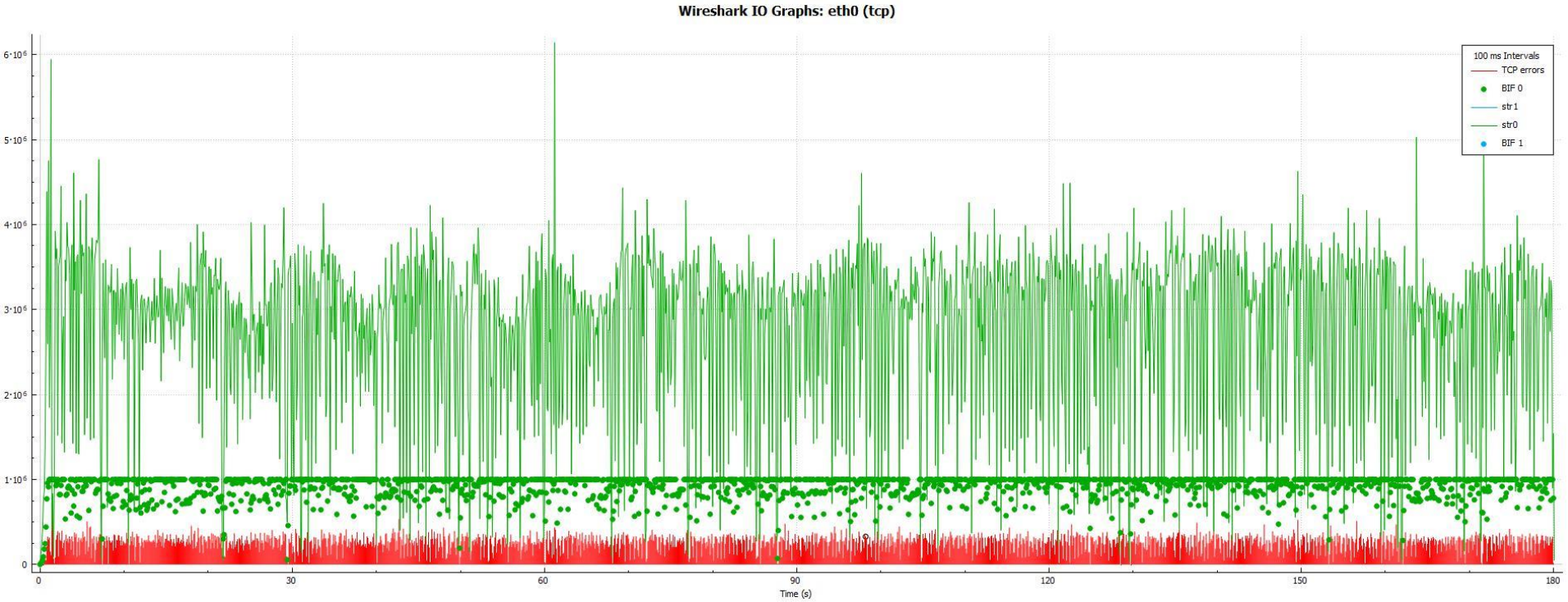
5-stream convergence

BBR TCP



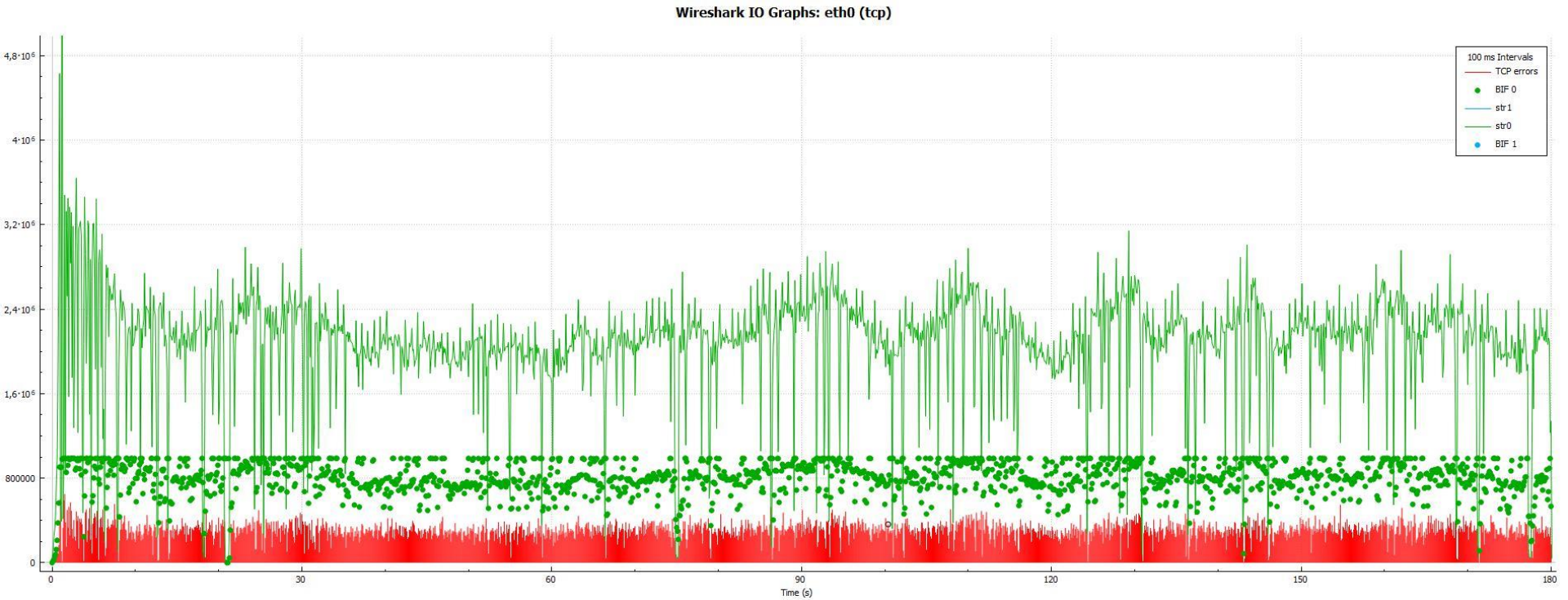
1% loss link behavior – Great, full BW rate!

BBR TCP



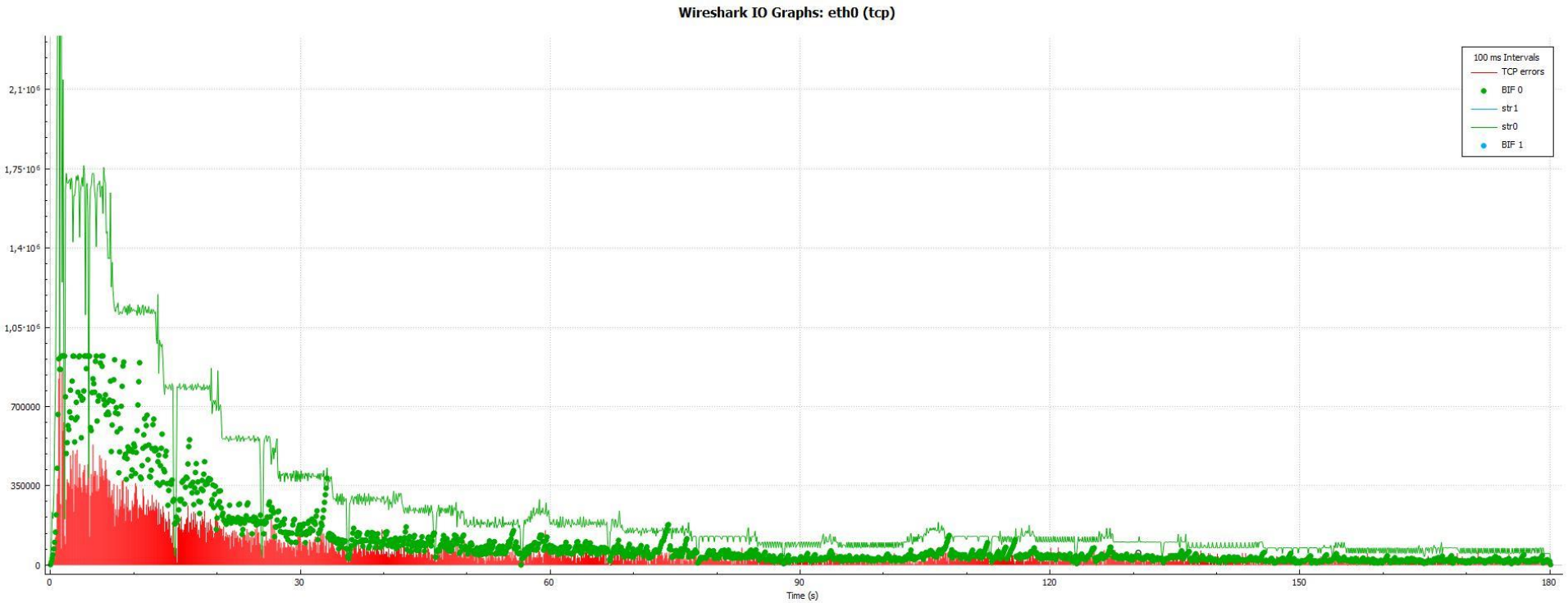
5% loss link behavior – 30Mbps out from 40, amazing!

BBR TCP



10% loss link behavior – 24Mbit out of 40 – is it possible to kill it at all?

BBR TCP



20% loss link behavior – alright, we went too far ☺, but…

Attacks on CA

Three different types of attack are aimed to make a sender faster:

1. ACK division attack (intentional accelerating of CA algorithm)
2. DUP ACK spoofing (influencing on Fast Recovery phase)
3. Optimistic ACKing

Used flowgrind commands

```
for cong in  'reno' 'scalable' 'htcp' 'bic' 'nv' 'cubic' 'vegas' 'hybla' 'westwood' 'veno' 'yeah' 'illinois' 'cdg' 'bbr' 'lp';
do flowgrind -H s=10.10.10.10/192.168.112.253,d=10.10.10.12/192.168.112.233 -i 0.005  -O s=TCP_CONGESTION=$cong -T s=60,d=0 | egrep ^S >
/home/vlad/csv_no_loss/{$cong}_60s_no_loss.csv;
sleep 10
done
```

Regular 1-stream probe

```
for cong in  'reno' 'scalable' 'htcp' 'highspeed' 'bic' 'cubic' 'vegas' 'hybla' 'nv' 'westwood' 'veno' 'yeah' 'illinois' 'cdg' 'bbr' 'lp';
do flowgrind -n 5 -H s=10.10.10.10/192.168.112.253,d=10.10.10.12/192.168.112.233  -O s=TCP_CONGESTION=$cong -T s=90,d=0  | egrep ^S >
/home/vlad/{$cong}_90s_intra_fair.csv;
sleep 30
done
```

5-stream intra-protocol fairness

```
for cong in  'reno' 'scalable' 'htcp' 'highspeed' 'bic' 'cubic' 'vegas' 'hybla' 'nv' 'westwood' 'veno' 'yeah' 'illinois' 'cdg' 'bbr' 'lp';
do flowgrind -n 2 -F 0 -H s=10.10.10.10/192.168.112.253,d=10.10.10.12/192.168.112.233  -O s=TCP_CONGESTION=$cong -T s=90,d=0 -F 1 -H
s=10.10.10.10/192.168.112.253,d=10.10.10.12/192.168.112.233 -O s=TCP_CONGESTION=reno -i 0.01 -T s=90,d=0 | egrep ^S >
/home/vlad/{$cong}_90s_reno_friendl.csv;
sleep 30
done
```

vs. Reno Friendliness

```
 flowgrind -n 5 -F 0 -H s=10.10.10.10/192.168.112.253,d=10.10.10.12/192.168.112.233  -O s=TCP_CONGESTION=$cong -T s=100,d=0 -F 1 -H
s=10.10.10.10/192.168.112.253,d=10.10.10.12/192.168.112.233 -O s=TCP_CONGESTION=$cong -Y s=10 -T s=80,d=0 -F 2 -H
s=10.10.10.10/192.168.112.253,d=10.10.10.12/192.168.112.233 -O s=TCP_CONGESTION=$cong -Y s=20 -T s=60,d=0 -F 3 -H
s=10.10.10.10/192.168.112.253,d=10.10.10.12/192.168.112.233 -O s=TCP_CONGESTION=$cong -Y s=30 -T s=40,d=0 -F 4 -H
s=10.10.10.10/192.168.112.253,d=10.10.10.12/192.168.112.233 -O s=TCP_CONGESTION=$cong -Y s=40 -T s=20,d=0| egrep ^S >
/home/vlad/{$cong}_100s_5str_converg.csv
```

5-stream fairness with displaced start and different length

Q&A

Any questions?

* Are we really at this point? I can't believe it ☺

Topic for next time? Network-assisted congestion control, state-of-the-art algorithms deep dive?

Thanks for your attention!