TCP, a network process

Joe R. Doupnik
Mindworks UK
jrd@cc.usu.edu
joe.doupnik@mindworksuk.com

What this talk is about

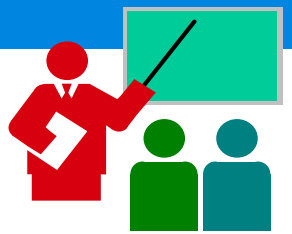
Normally I would review IP, UDP, TCP etc header details and what they do

I happen to like that stuff, but it can be a little boring

Instead I decided to discuss another aspect:

TCP working for a living - its algorithms which permit the Internet to work well under widely varying conditions. This is looking at packets, but at the packet process rather than at static structures.

A little academics is added first, for seasoning



Protocol basics

Items necessary for robust protocols

Checksums for data integrity

Checksums on both data and ACKs

- **IP: covers only IP header**
- **UDP: optional, covers UDP header and data**
- **TCP: covers TCP header and data**
- **Simple linear addition (1's complement of 1's complement sum)**

Protocol basics

Sequence numbers to distinguish old, new, duplicate data

IP: none. IP ident number is different for each datagram, used to reassemble fragments

UDP: none, each datagram is the entire message

TCP: full, 32-bit, identifies starting octet in this segment, starting point is random and set in SYN segment. Packets are not otherwise numbered.

Protocol basics

Timers to break deadlocks from lost packets

- **IP: none, no feedback**
- **UDP: none, no feedback**
- **TCP: full. Measure round trip delay, for timing out lost packets. ACKs may be delayed to group many into one, keep-alive probes, etc. Granularity is often tens to 200 milliseconds, which is very coarse.**
- **TCP uses arriving ACKs to clock out new data, operates at full network speed**

Protocol basics

ACKs to confirm delivery, and provide flow control, must have sequence numbers to avoid confusion about what is sent and ACK'd

IP: none. Pure connectionless datagram

UDP: none. Pure connectionless datagram

TCP: full, connection oriented. Rules say all TCP data must be ACK'd sooner or later, even if old, repeated, or far future data. Soon means < 0.5 sec and that is often 200ms in wide practice.

Protocol basics

Flow control

- IP: none, except a few ICMP “source quench” packets
- UDP: never heard of the topic. Manual throttling required. Poor on congested networks.
- TCP: full featured
  - Dynamic estimation of network capacity (Van Jacobson’s work). Congestion avoidance adapts to changing network conditions.
  - Each packet announces receiver buffer space available via its window size
  - Arriving ACKs announce resource space

TCP is a rich protocol

- **Session oriented, thus a start, middle and end**
- **Data is a stream of bytes, no record/message boundaries, no binding of units of data into one packet (but UDP does bind)**
- **Each direction of data flow has its own sequence numbering of individual bytes, packets are not numbered**
- **Every data byte (including virtual data SYN/FIN) must be ACK'd, sooner or later**
- **Timers break deadlocks from lost transmissions**

TCP is a rich protocol

Transmission rates are adjusted dynamically to accommodate the observed network. Unlike TCP, UDP has no controls at all (no feedback)

Self-adjustment is the key and a primary reason the Internet survives to this day

A set of heuristics govern dynamic behavior. One man, Van Jacobson, is largely responsible for the working heuristics of today.

A TCP network dialogue

Big Transmitter: I have _this many_ buffer bytes to send, ready to fly through the ether

Big Receiver: I have acres of empty buffer space, send me lots of shinny new bytes, now!

Invisible Network: Fellas, I can't deal with that much data at once, lack of buffer space you see. Router memory costs a fortune and there are other users.

But if you do blast away, I warn you, packets can be lost. Discarding is my way of dealing with overload. Learn from the experience, if you can.

Speed limits, no cameras

Transmitter sends as much as it can, limited by smallest condition below, ignores ACKs

Receiver portion obtains ACKs, controls cwin, reports remote's receive window capacity

1. **Amount of unsent data available**
2. **Free space in remote receiver's buffer (window size carried in ACKs), after deducting in-flight data (sent but not yet ACK'd)**
3. **Congestion window size, cwin: estimated network capacity to hold in-flight data, initially one packet, estimated from successful or lost packets**

Timing successful transmissions

ACKs provide measurement of round trip time (rtt)

Events are counted in intervals of rtt duration (one tick)

Delayed ACKs just provide fewer measurements

Timeout, rto, is computed as running average:

rto = avg(rtt) + 4 * stddev(rtt)

long pipes noisy networks

Timeout:

- **computed from each rtt measurement, adapts, doubles on each repeated retransmission**
- **breaks deadlocks from lost ACKs**
- **tell us if the network is overloaded (lost packets)**

ACK is data clock, rtt is rate

Sender blocks awaiting permission to send

ACKs release buffer space, allow sending new data

A low rate data link delivers packets slowly to the receiver, hence a slow rate of ACKs to sender

One tick is one round trip time (data out, ACK back)

Slow ACK rate means slow transmission rate, which matches slow network bit rates. "It just works."

Long paths are filled by buffering at both sides and sending many packets before ACKs arrive

Timeout prevents deadlock from lost ACKs

Silly window syndrome == slow

- **Silly window syndrome avoidance: avoid sending small things, wait until there is a lot to do**
- **Nagle: send full segments, hang onto the tag end until all preceding data have been ACK'd, there might be more data coming along to fill a packet**
- **Delayed ACKs, don't bother ACKing every data arrival when another may come shortly: one ACK can cover data in both arrivals. Thus try ACKing every other successful data packet, else timeout after 100-200ms and send a pending ACK.**
- **Both are wishful thinkers, and often wrong**

Silly window syndrome == slow

- **If Nagle holds back the tag end, and the other side holds back an ACK, we wait and wait until the ACK timer fires on the receiver. Deadlock, un-good.**
- **Request/response systems, such a web serving, backups & more, crawl to pace of ACK timer when these two mechanisms interact (as they do)**
- **Better is to turn off Nagle mode, leave Delayed ACKs alone. Serious applications do this internally.**
- **Window openings are usually announced only when large space become available**

Van Jacobson's work

Slow Start State: fill the pipe quickly, get ACK clock ticking.

Set slow start threshold (safe amount of in-flight) to huge

Set congestion window cwin to 1-2 packets, allow sending

cwin is network's learned window, a limit on in-flight amount

Arrival of an ACK for data changes cwin:

If slow start threshold is not yet been reached, grow quickly

cwin ⇦ cwin + 1 "slow start", fill the pipe

else use congestion avoidance tactic

cwin ⇦ cwin + 1/cwin grow gently, probe net

If timeout (lost ACK) then

slow start threshold ⇦ cwin/2 safe amount of in-flight data

cwin ⇦ 1 restart slowly re-learn the net

resend all old data

Congestion avoidance

Slowly probe for & adjust to more network capacity

Each successful packet adds a future transmission credit of 1/cwin packets to the account of cwin

When a full credit has accumulated, add it to cwin

Result is additive growth of cwin

cwin ⇦ cwin (1 + 1/cwin)

Example: at cwin of 10, ten successful packets earns credit to increase cwin to 11, whether needed or not

Cwin successful packets are needed to step cwin by 1

Expect packet loss if cwin grows to be too large

Transmitter's restrictions

Transmission is restricted to the smallest of three window sizes:

- **Transmitter's available unsent data**
- **Receiver's available space (deducting in-flight data)**
- **Network's estimated capacity cwin, subject to two behavioral regimes**

- **Additionally, fast retransmit/fast recovery makes temporary adjustment of cwin**

The transmitter just sends as much as permitted, as quickly as it can (typically back to back packets)

Reading the red line of plots

Red line is number of *bytes in-flight*, as observed by the most recent transmission and reception

The line goes up as the transmitter sends new segment numbers

The line goes down when ACKs confirm delivery

Tiny + signs indicate ACK arrival

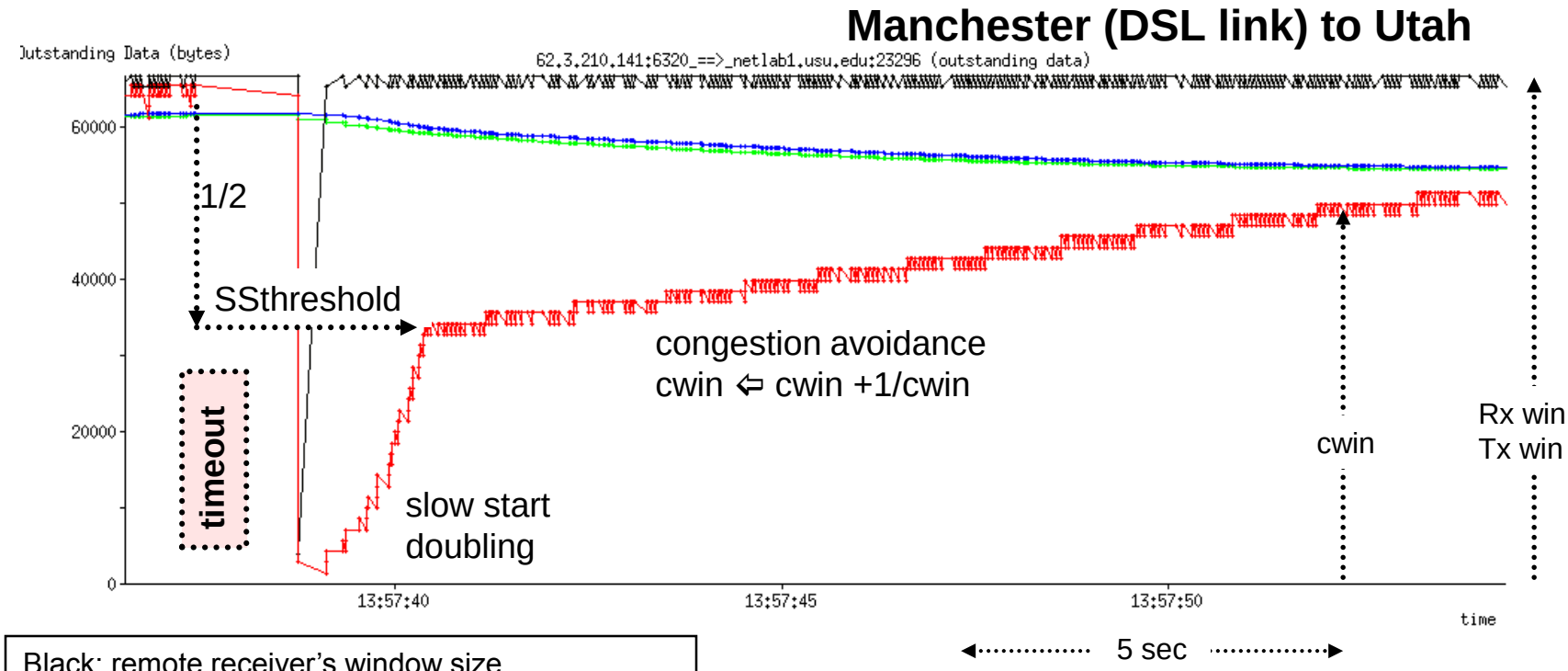
The line is high when the transmitter is well ahead of confirmations

The line is low when the receiver catches up

The height is the number of bytes in-flight, ≤ cwin

Classical Van Jacobson plot

Detail of a loss recovery interval, illustrating slow start and 1/cwin steps
Example starts with packet loss at 64KB full network capacity



Black: remote receiver's window size

Red: bytes in flight (sent - ACK'd)

Blue: simple average bytes in flight

Green: long time weighted average bytes in flight

Duration at a step is set by waiting for ACKs, "net window" cwin is full

Fast retransmit, fast recovery

- If a packet is lost in a series, ACK sequence numbers are stuck at the last good byte + 1
- Each following packet must be ACK'd, but contents cannot be dealt with until the hole is filled. Receiver sends duplicate ACKs immediately, each with same sequence number.
- Transmitter sees three or more ACKs having same sequence number (dups), three is a clincher for not just reordering
- Resend oldest packet now, hoping it will fill the hole without a timeout. Inflate cwin by number of dup ACKs (packets which have left the net). New cwin can permit sending new data(!). When ACK for replacement arrives reset cwin to ssthreshold for a safe running restart. No timeouts, hence ssthreshold remains unchanged.
- Two or more missing packets results in timeout and doing slow start from the beginning (network state and clock are lost)

Fast retransmit example

sent 1 2 3 4 5 6 7 8 9 10

received 1 2 4 5 6 7 8 9 10
reply A A D D D D D D D D=dup ACK (A)

rcv'd reply A A D D D D D D D

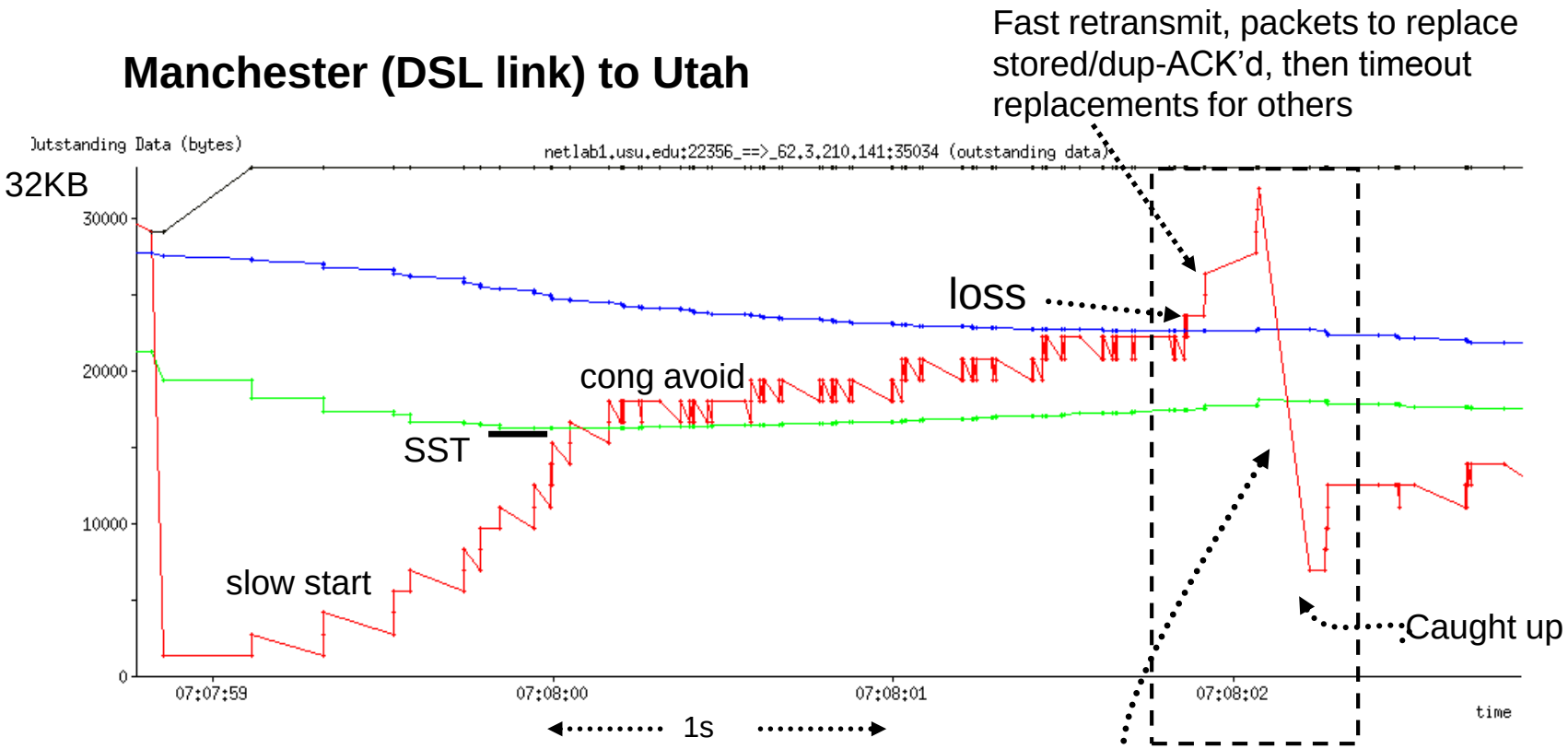
fast retransmit 3 11 12 13…17

(replaces dup ACK'd packets which have left the net)

This mechanism is quicker (4 rtt) than waiting for a timeout

Fast Retransmit observation

Manchester (DSL link) to Utah



The spike/blip occurring after a loss is a telltale of the fast retransmit mechanism refilling the net after receiving duplicate ACKs

Same event in Ethereal (400ms)

No.	Time	Source	Destination	Protocol	Info
707	07:08:01.866306	129.123.28.201	62.3.210.141	TCP	22356 > 35034 [ACK] Seq=433681 Ack=1 Win=66720 Len=1390 TSV=126837757 TSER
708	07:08:01.866311	129.123.28.201	62.3.210.141	TCP	22356 > 35034 [ACK] Seq=435071 Ack=1 Win=66720 Len=1390 TSV=126837757 TSER
709	07:08:01.911141	62.3.210.141	129.123.28.201	TCP	35034 > 22356 [ACK] Seq=1 Ack=414221 Win=33360 Len=0 TSV=2510438635 TSER=1
710	07:08:01.911160	129.123.28.201	62.3.210.141	TCP	22356 > 35034 [ACK] Seq=436461 Ack=1 Win=66720 Len=1390 TSV=126837802 TSER
711	07:08:01.917887	62.3.210.141	129.123.28.201	TCP	[TCP Dup ACK 709#1] 35034 > 22356 [ACK] Seq=1 Ack=414221 Win=33360 Len=0 T
712	07:08:01.917900	129.123.28.201	62.3.210.141	TCP	22356 > 35034 [ACK] Seq=437851 Ack=1 Win=66720 Len=1390 TSV=126837808 TSER
713	07:08:01.919636	62.3.210.141	129.123.28.201	TCP	[TCP Dup ACK 709#2] 35034 > 22356 [ACK] Seq=1 Ack=414221 Win=33360 Len=0 T
714	07:08:01.919648	129.123.28.201	62.3.210.141	TCP	22356 > 35034 [ACK] Seq=439241 Ack=1 Win=66720 Len=1390 TSV=126837810 TSER
716	07:08:02.019458	62.3.210.141	129.123.28.201	TCP	[TCP Dup ACK 709#3] 35034 > 22356 [ACK] Seq=1 Ack=414221 Win=33360 Len=0 T
717	07:08:02.019494	129.123.28.201	62.3.210.141	TCP	[TCP Fast Retransmission] 22356 > 35034 [ACK] Seq=414221 Ack=1 Win=66720 L
718	07:08:02.030073	62.3.210.141	129.123.28.201	TCP	[TCP Dup ACK 709#4] 35034 > 22356 [ACK] Seq=1 Ack=414221 Win=33360 Len=0 T
719	07:08:02.033696	62.3.210.141	129.123.28.201	TCP	[TCP Dup ACK 709#5] 35034 > 22356 [ACK] Seq=1 Ack=414221 Win=33360 Len=0 T
720	07:08:02.035819	62.3.210.141	129.123.28.201	TCP	[TCP Dup ACK 709#6] 35034 > 22356 [ACK] Seq=1 Ack=414221 Win=33360 Len=0 T
721	07:08:02.037818	62.3.210.141	129.123.28.201	TCP	[TCP Dup ACK 709#7] 35034 > 22356 [ACK] Seq=1 Ack=414221 Win=33360 Len=0 T
722	07:08:02.060556	62.3.210.141	129.123.28.201	TCP	[TCP Dup ACK 709#8] 35034 > 22356 [ACK] Seq=1 Ack=414221 Win=33360 Len=0 T
723	07:08:02.064429	62.3.210.141	129.123.28.201	TCP	[TCP Dup ACK 709#9] 35034 > 22356 [ACK] Seq=1 Ack=414221 Win=33360 Len=0 T
724	07:08:02.065927	62.3.210.141	129.123.28.201	TCP	[TCP Dup ACK 709#10] 35034 > 22356 [ACK] Seq=1 Ack=414221 Win=33360 Len=0
725	07:08:02.066677	62.3.210.141	129.123.28.201	TCP	[TCP Dup ACK 709#11] 35034 > 22356 [ACK] Seq=1 Ack=414221 Win=33360 Len=0
726	07:08:02.066692	129.123.28.201	62.3.210.141	TCP	22356 > 35034 [ACK] Seq=440631 Ack=1 Win=66720 Len=1390 TSV=126837957 TSER
727	07:08:02.068926	62.3.210.141	129.123.28.201	TCP	[TCP Dup ACK 709#12] 35034 > 22356 [ACK] Seq=1 Ack=414221 Win=33360 Len=0
728	07:08:02.068939	129.123.28.201	62.3.210.141	TCP	22356 > 35034 [ACK] Seq=442021 Ack=1 Win=66720 Len=1390 TSV=126837959 TSER
729	07:08:02.071799	62.3.210.141	129.123.28.201	TCP	[TCP Dup ACK 709#13] 35034 > 22356 [ACK] Seq=1 Ack=414221 Win=33360 Len=0
730	07:08:02.071811	129.123.28.201	62.3.210.141	TCP	22356 > 35034 [ACK] Seq=443411 Ack=1 Win=66720 Len=1390 TSV=126837962 TSER
731	07:08:02.074048	62.3.210.141	129.123.28.201	TCP	[TCP Dup ACK 709#14] 35034 > 22356 [ACK] Seq=1 Ack=414221 Win=33360 Len=0
732	07:08:02.074062	129.123.28.201	62.3.210.141	TCP	22356 > 35034 [PSH, ACK] Seq=444801 Ack=1 Win=66720 Len=1390 TSV=126837964
733	07:08:02.116025	62.3.210.141	129.123.28.201	TCP	[TCP Dup ACK 709#15] 35034 > 22356 [ACK] Seq=1 Ack=414221 Win=33360 Len=0
734	07:08:02.123645	62.3.210.141	129.123.28.201	TCP	[TCP Dup ACK 709#16] 35034 > 22356 [ACK] Seq=1 Ack=414221 Win=33360 Len=0
735	07:08:02.127018	62.3.210.141	129.123.28.201	TCP	[TCP Dup ACK 709#17] 35034 > 22356 [ACK] Seq=1 Ack=414221 Win=33360 Len=0
737	07:08:02.224463	62.3.210.141	129.123.28.201	TCP	35034 > 22356 [ACK] Seq=1 Ack=440631 Win=33360 Len=0 TSV=2510438948 TSER=1
738	07:08:02.224499	129.123.28.201	62.3.210.141	TCP	22356 > 35034 [ACK] Seq=446191 Ack=1 Win=66720 Len=1390 TSV=126838115 TSER
739	07:08:02.271060	62.3.210.141	129.123.28.201	TCP	35034 > 22356 [ACK] Seq=1 Ack=442021 Win=33360 Len=0 TSV=2510438995 TSER=1

Lost data packets, dup ACKs from follow-ons, fast retransmit fixed one (3 dups in a row) but more were lost and need timeouts to fix

Summary

- Slow start threshold is recent memory of what is the largest safe successful amount of in-flight data
- Doubling by slow start can easily overwhelm a net
- Upon timeout, the dramatic back off to half the previous in-flight amount is required to maintain stability of the network with many stations. Slower back off can lead to instability
- Linear growth learns about more network capacity over time, but it keeps pushing the boundary
- Times are measured continuously, adapts to net state
- Without these basics the Internet would collapse

Time to fill a window

Let us estimate the time needed to fill a network window

If capacity is say C bytes, and using P byte packets,

slow start's doubling needs S steps, $C = P * 2^S$

For C = 64KB and P = 1.4KB packets we find

$64K/1.4K = 46 \text{ packets} \leq 2^S$

S = 6, nominally 64 packets

To go half way up, as for recovery,

5 levels using 31 packets

Going from step to step requires ACKs to free prior step, total elapsed time is $1+2+4+8.. \text{ rtt} = 2^S - 1 \text{ rtt} \approx C/P \text{ rtt}$

Filling time scales as C/P rtt for slow start ←

Time to fill a window

For congestion avoidance, cwin transmissions are needed to increase cwin by one unit

If 64KB is divided into 46 levels of 1.4KB each

The number of packets at each level is cwin, thus

1+2+3+…+46 = (C/P)((C/P)+1)/2 = 1081 packets

To do only the top half, as for recovery, requires

23 top levels using 1081-276=805 packets

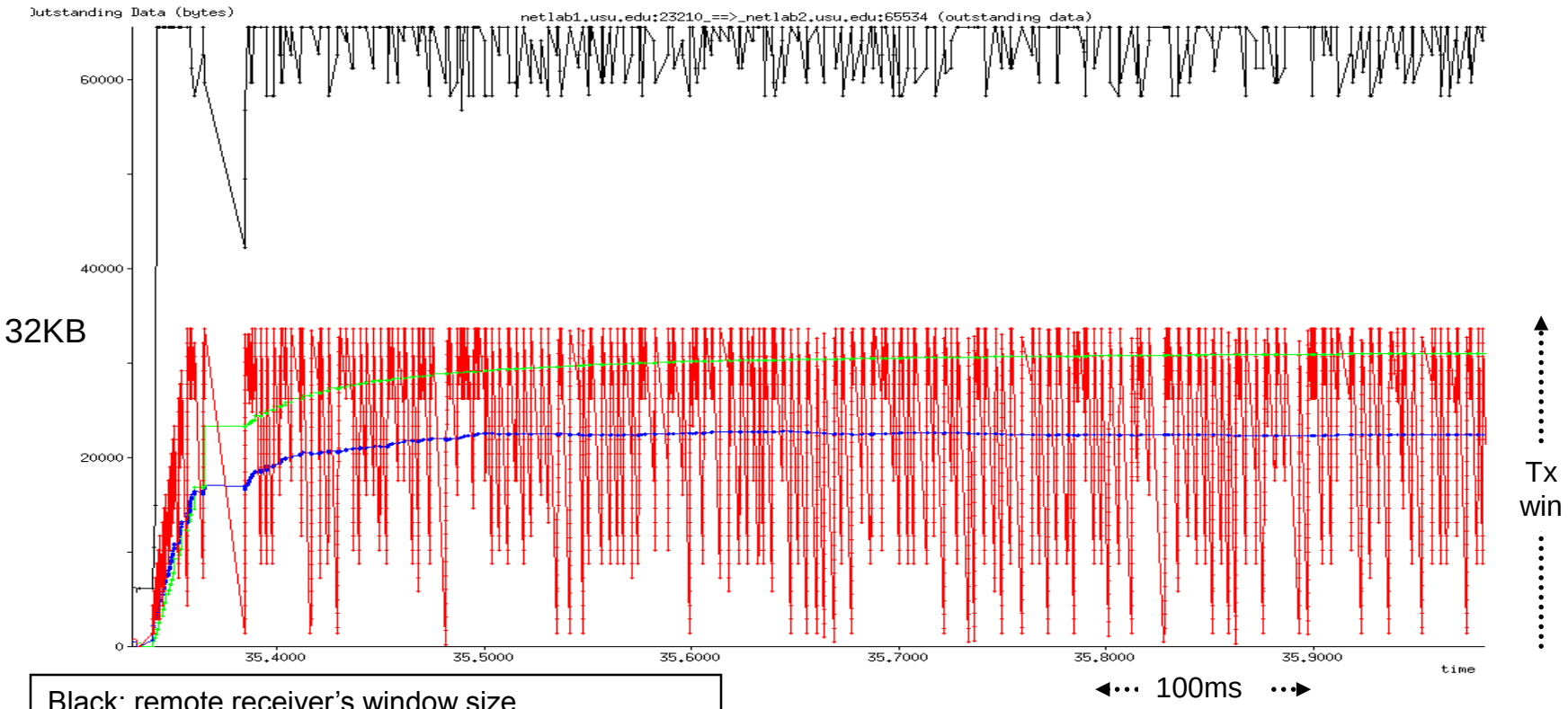
Elapsed time, waiting for ACKs, is 1+2+3… rtt

Filling time scales as $(C/P)^2$ rtt for cong-avoid ←

Speed of light does not change with bit rate

Local link, 100Mbps HDX

Waiting on ACKs is the throttle, no network loss



Black: remote receiver's window size

Red: bytes in flight (sent - ACK'd)

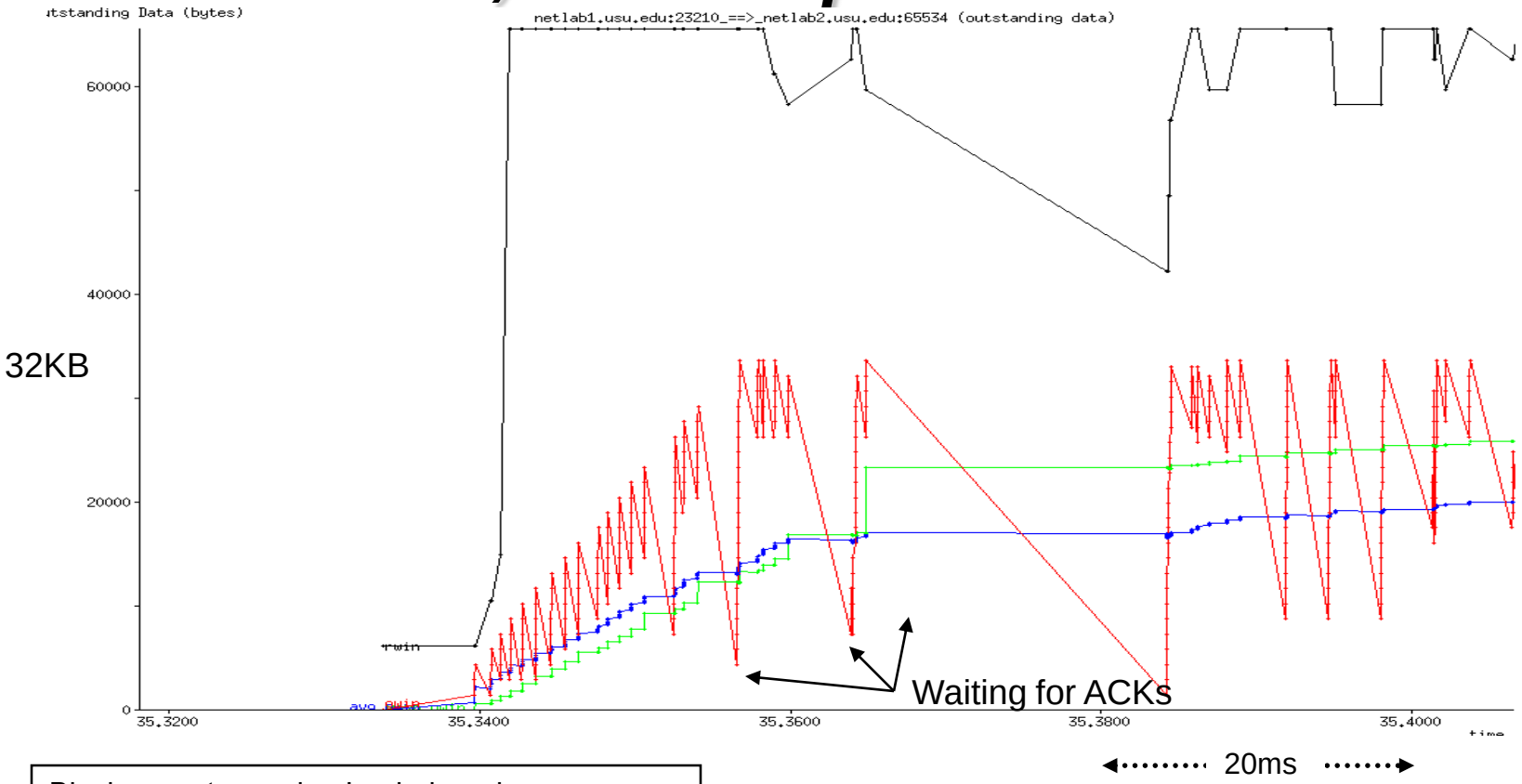
Blue: simple average bytes in flight

Green: long time weighted average bytes in flight

About 9MBps, steady

But why so noisy?

Local link, startup details



Black: remote receiver's window size

Red: bytes in flight (sent - ACK'd)

Blue: simple average bytes in flight

Green: time weighted average bytes in flight

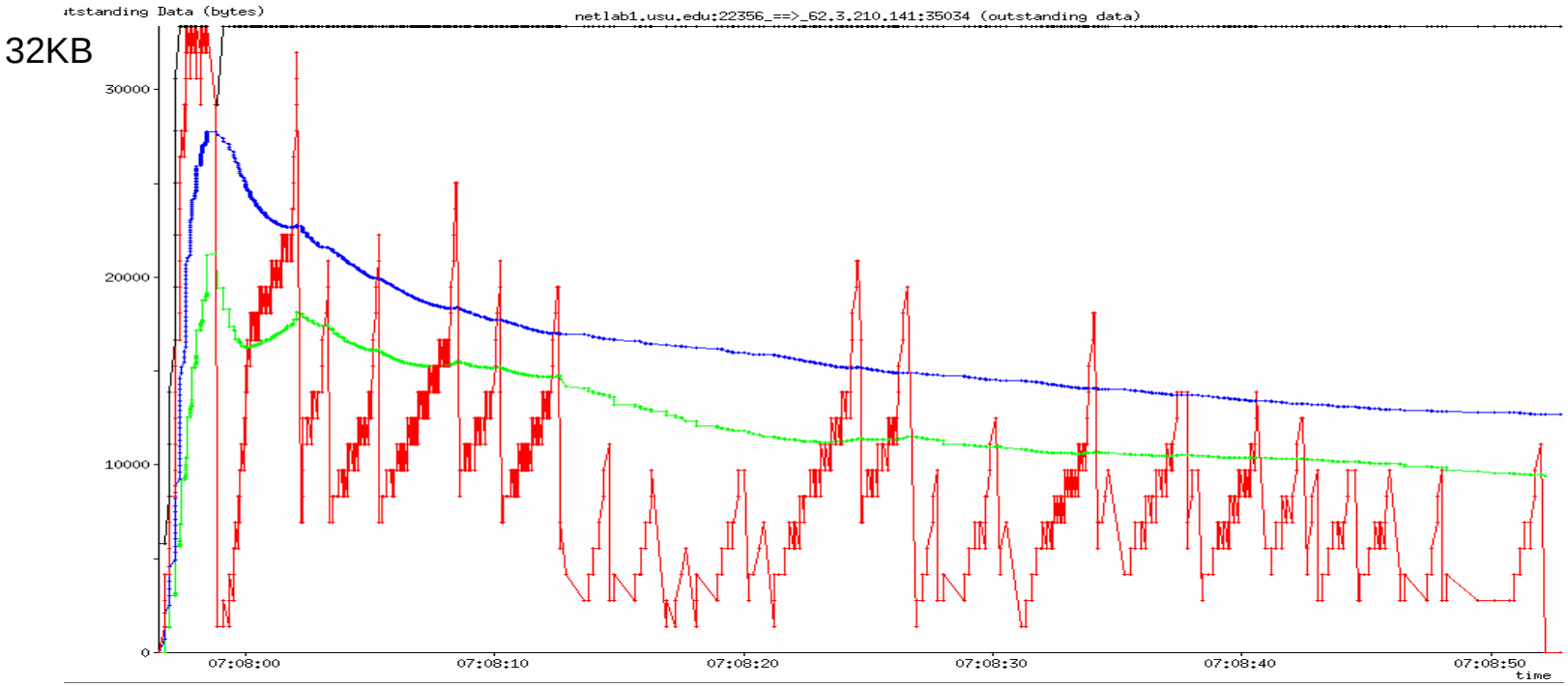
Receiver falling behind, gasps for breath, sends ACKs in bursts

Noisy local measurements

- The "noise" is from the delicate balance between sending and receiving at high speeds with a large sending permission
- Factors contributing to breaking pace are -
  - Driver fixation on task at hand:
    - Keep emptying present queue, plus system scheduler, etc
  - Ethernet capture effect with half duplex:
    - Send data packets back to back, blocks ACKs as other end backs off from collisions. ACKs queue up, then are released in a burst
  - Queuing of packets within the switch (equivalent to capture effect)
- Thus red line (bytes sent - ACK'd) undergoes large empty/full excursions

Utah to Manchester DSL

Network congestion causes packet loss, restart & relearning



◄····· 20s ·····►

Black: remote receiver's window size

Red: bytes in flight (sent - ACK'd)

Blue: simple average bytes in flight

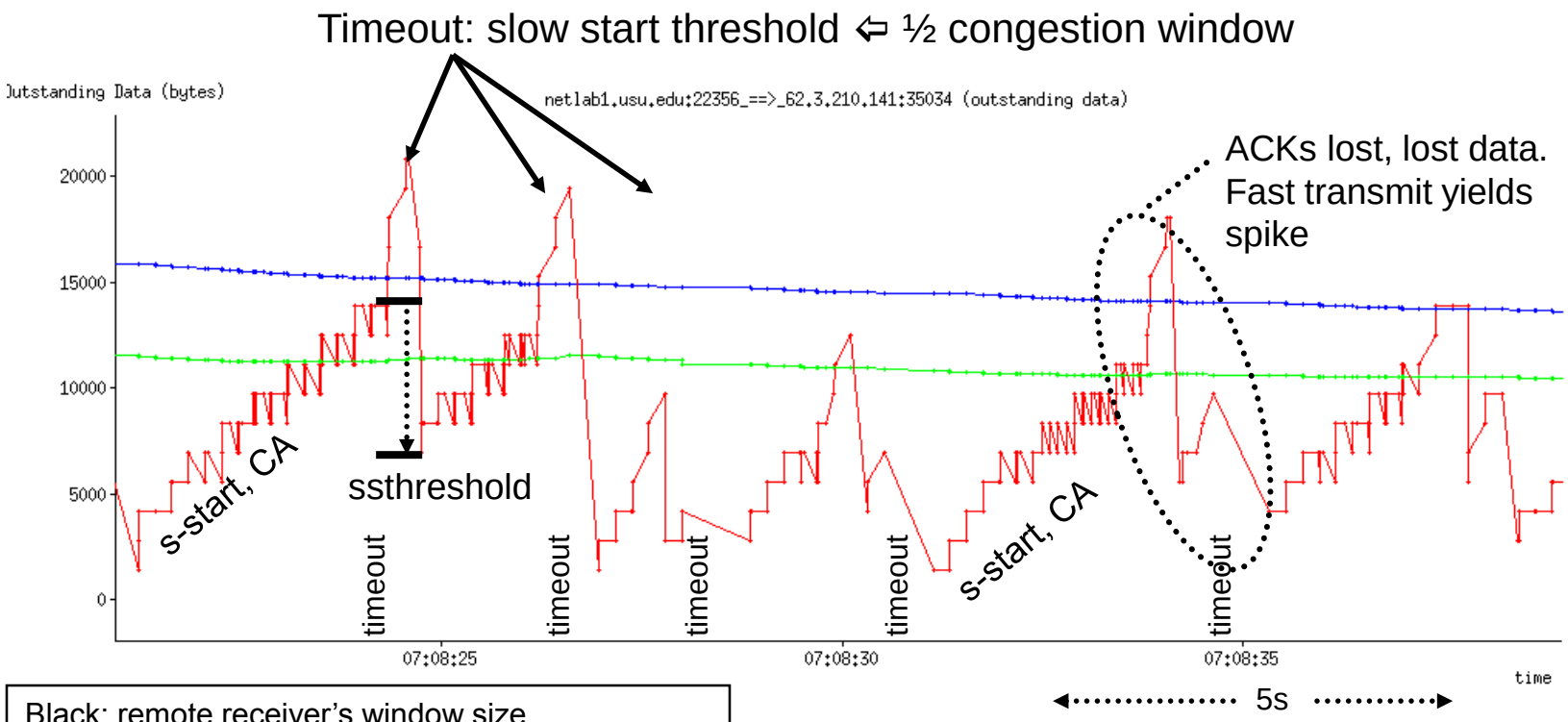
Green: long time weighted average bytes in flight

About 30KBps, declining, rtt 200ms

"Slow start" really is not slow

Utah-MAN link, learning details

Much congestion loss sending from Utah into the DSL pathway



Black: remote receiver's window size

Red: bytes in flight (sent - ACK'd)

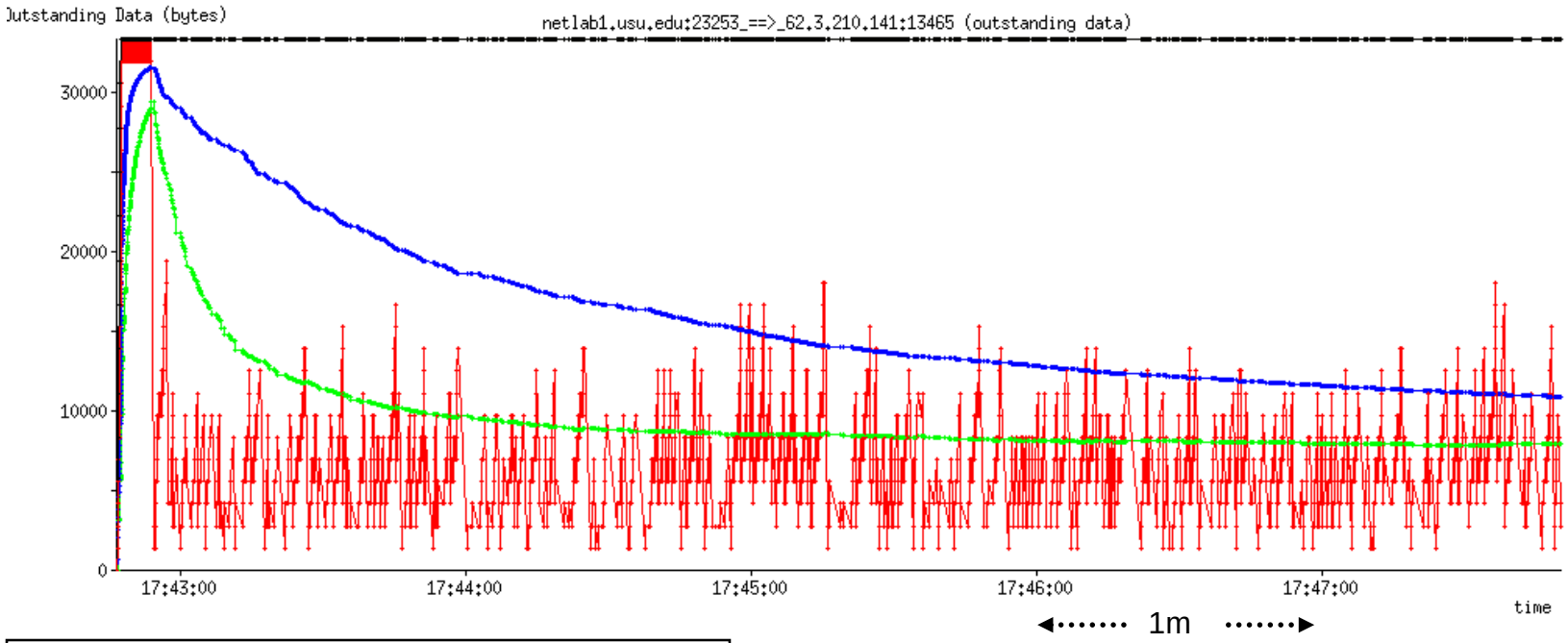
Blue: simple average bytes in flight

Green: long time weighted average bytes in flight

A router's queue drops packets, its congestion signal, busy here

Utah to Manchester (via DSL)

Longer 5 minute run to examine convergence behavior



Black: remote receiver's window size

Red: bytes in flight (sent - ACK'd)

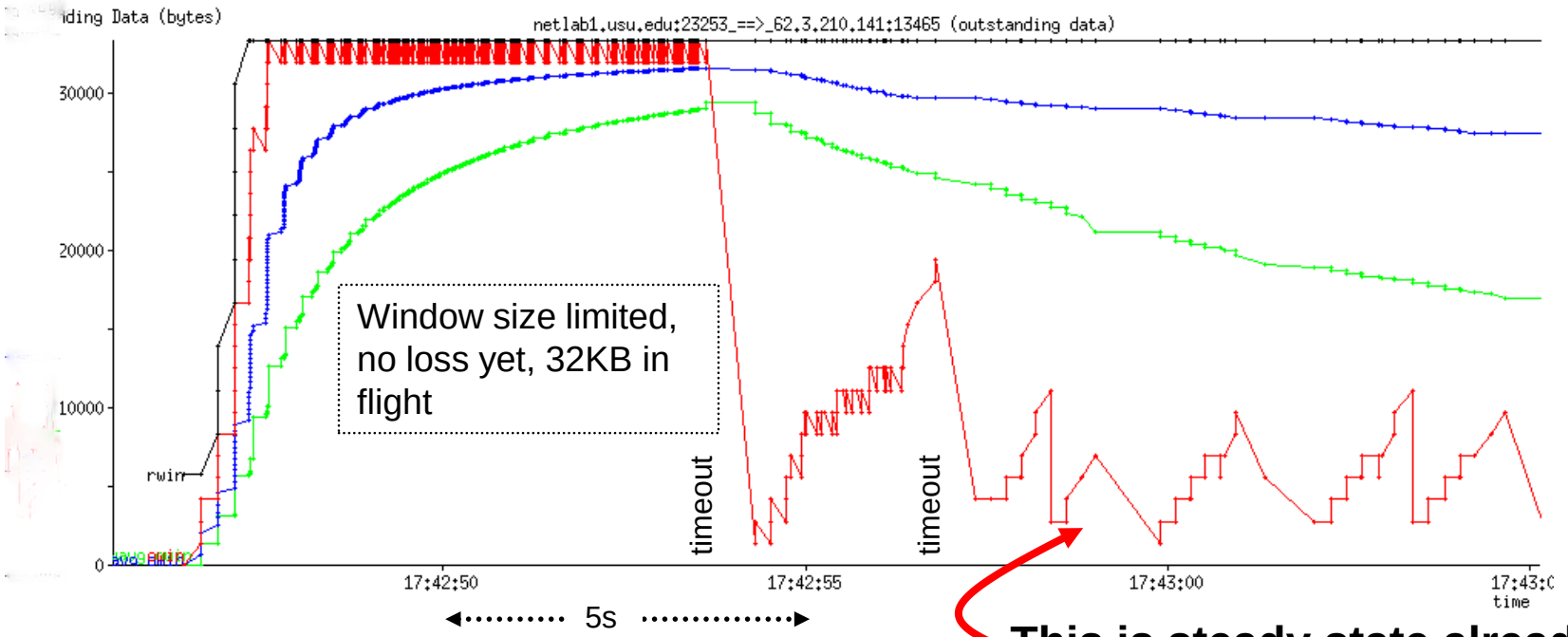
Blue: simple average bytes in flight

Green: long time weighted average bytes in flight

No improvement, tapers down to 20KB/sec
Looks like an ISP's QoS change after using up fast queue

Utah-Manchester, startup details

Dramatic change of capacity after startup, possibly ISP's QoS policy



Black: remote receiver's window size

Red: bytes in flight (sent - ACK'd)

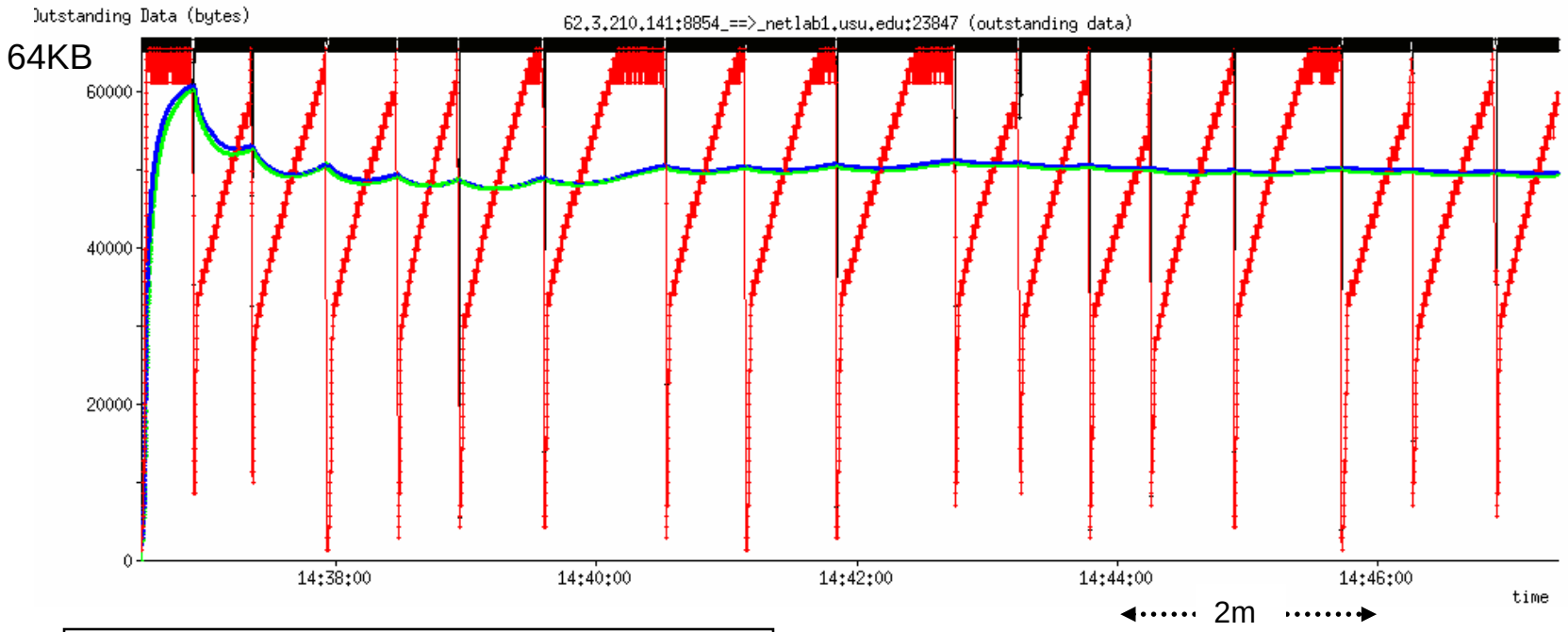
Blue: simple average bytes in flight

Green: long time weighted average bytes in flight

Looks as if receiving net can buffer only about 8-16KB of data.

Man-Utah, sending from slow side

Going outward (DSL "Upload") is much less congested



Black: remote receiver's window size

Red: bytes in flight (sent - ACK'd)

Blue: simple average bytes in flight

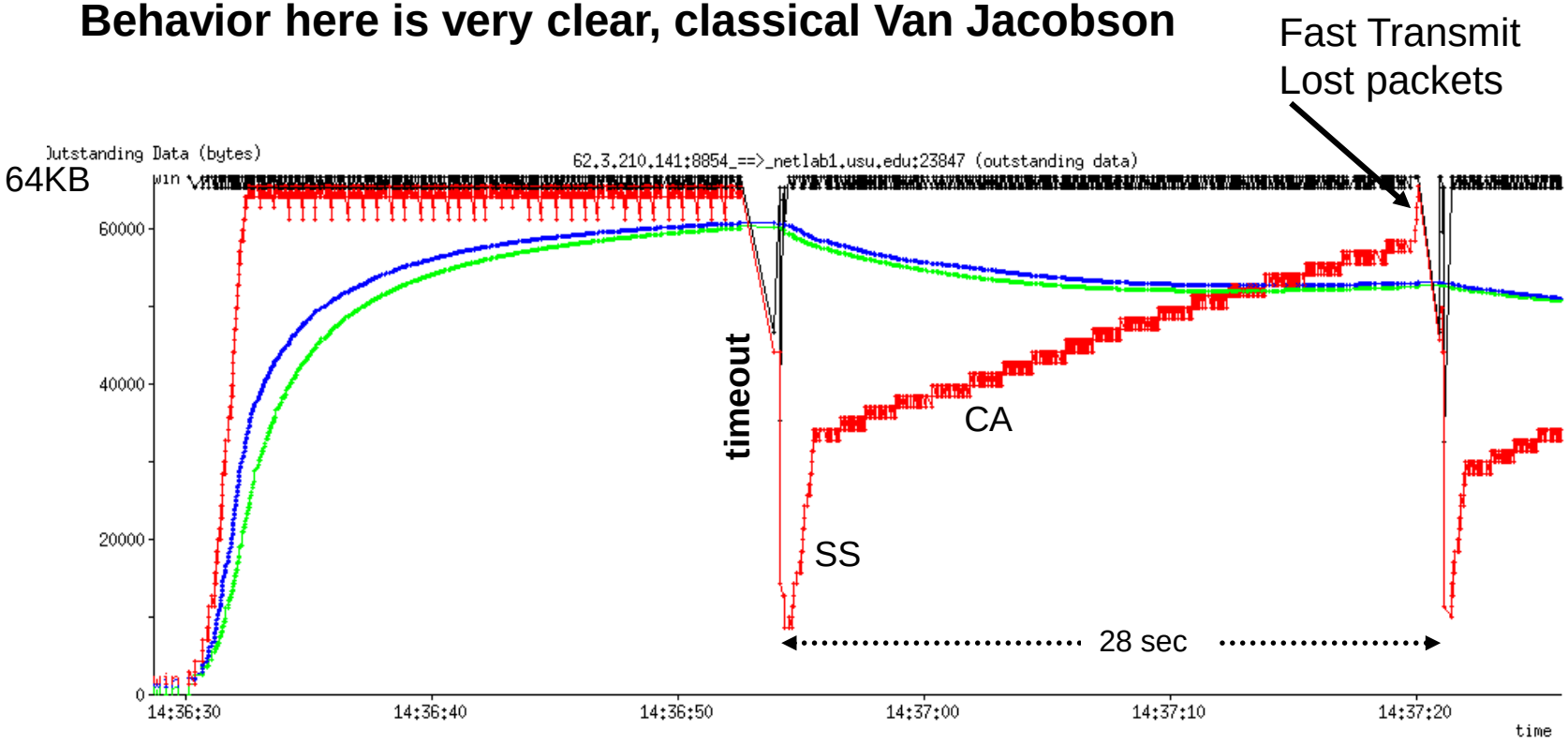
Green: long time weighted average bytes in flight

Ten minutes of steady transferring, 56KB/sec

Nearly all time is in congestion avoidance mode

Man-Utah, sending from slow side

Behavior here is very clear, classical Van Jacobson



Black: remote receiver's window size

Red: bytes in flight (sent - ACK'd)

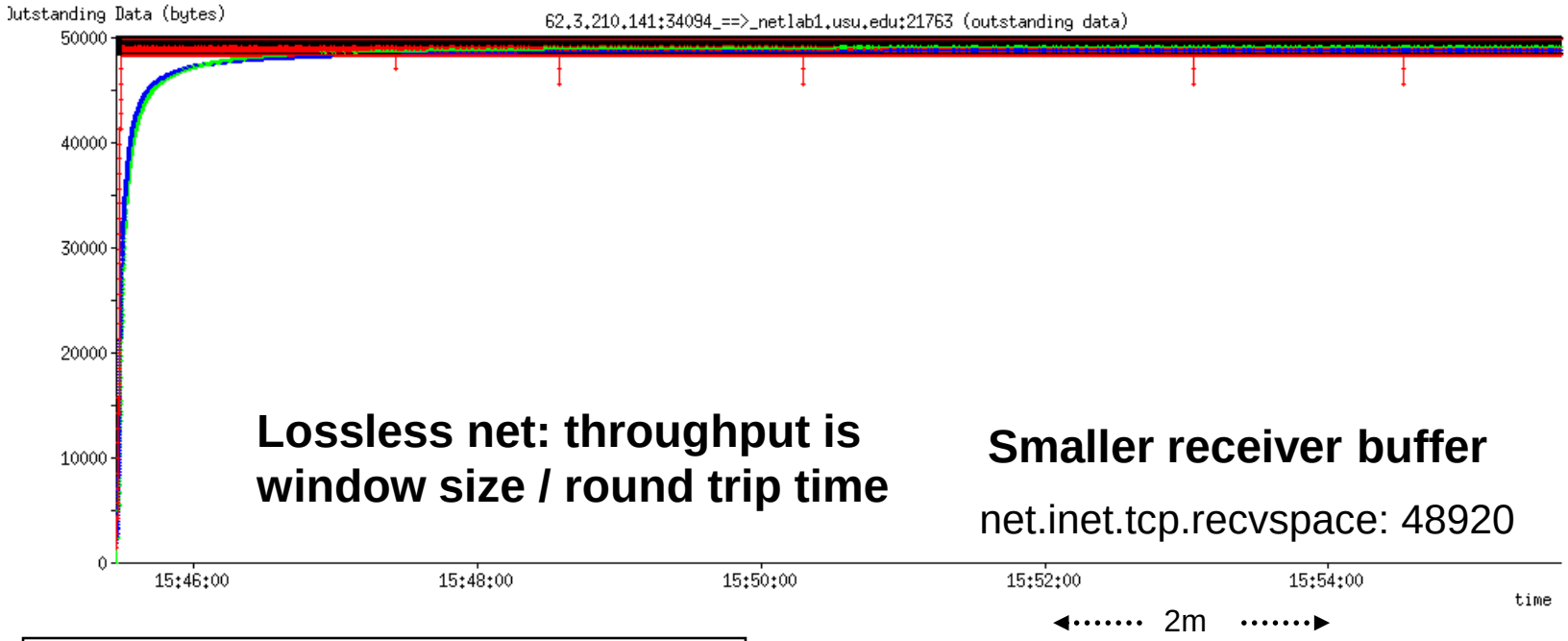
Blue: simple average bytes in flight

Green: long time weighted average bytes in flight

TCP buffer sizes, then recovery from loss, dominate this picture

Man-Utah, sending from slow side

Receiver's buffer reduced to 48KB to avoid overloading slow net



Lossless net: throughput is window size / round trip time

Smaller receiver buffer

net.inet.tcp.recvspace: 48920

Black: remote receiver's window size

Red: bytes in flight (sent - ACK'd)

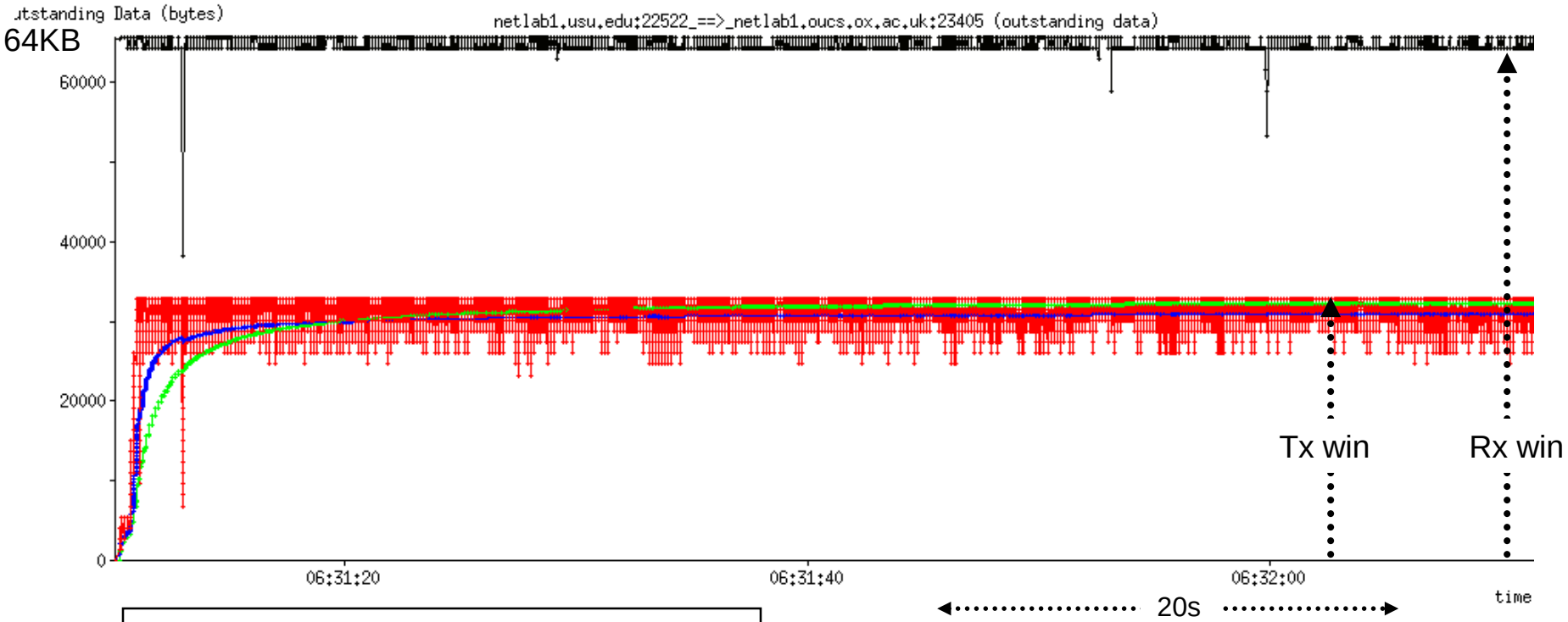
Blue: simple average bytes in flight

Green: long time weighted average bytes in flight

Ten minute transfer, 50KB/s
Going the other way is still horrid.
Big receiver buffer was faster

Utah to Oxford, via Janet2

Minimal network loss, bytes in flight quickly empty Tx buf, wait on ACKs



Black: remote receiver's window size (via ACKs)

Red: bytes in flight (sent - ACK'd)

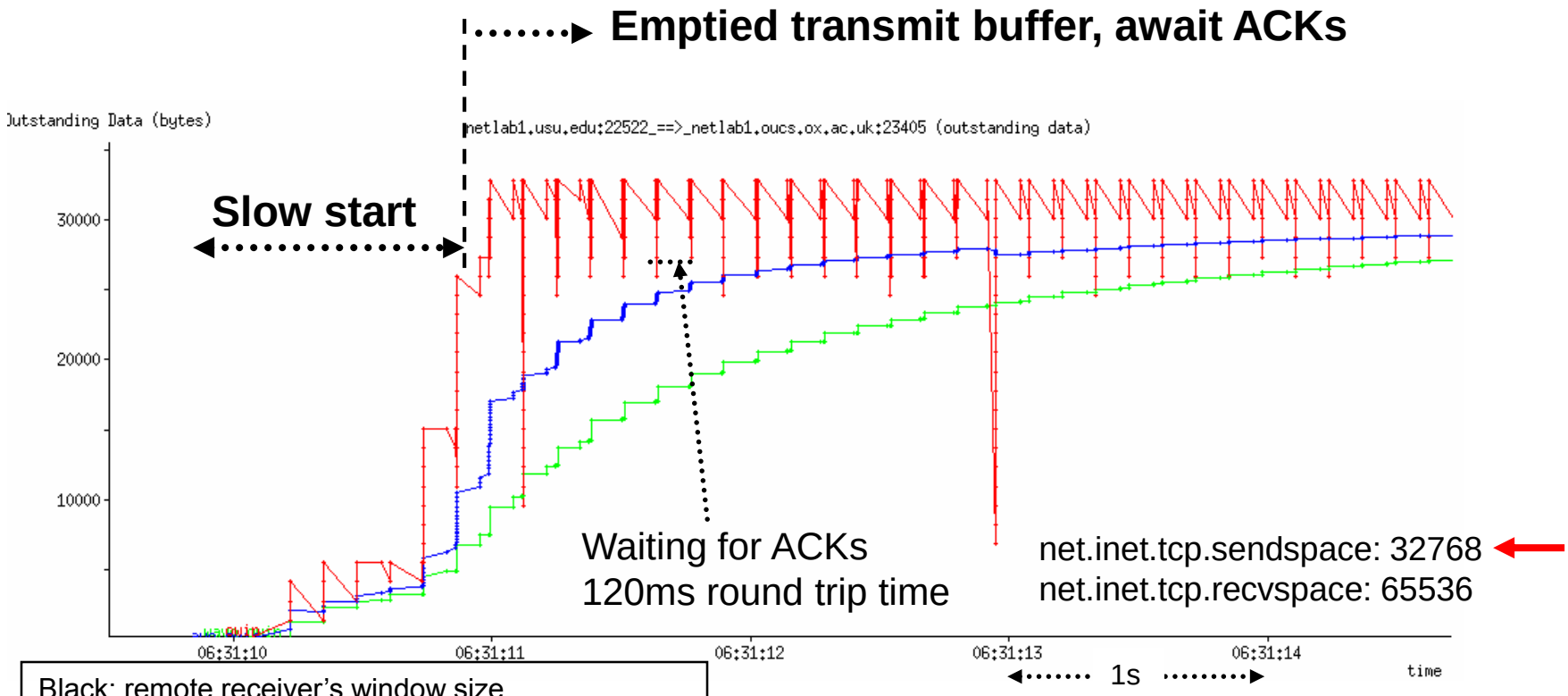
Blue: simple average bytes in flight

Green: long time weighted average bytes in flight

About 250KB/sec, limited by buffer sizes vs delay of ACKs

FreeBSD 32KB Tx buffer, rtt 120ms

Utah to Oxford, startup details



Black: remote receiver's window size

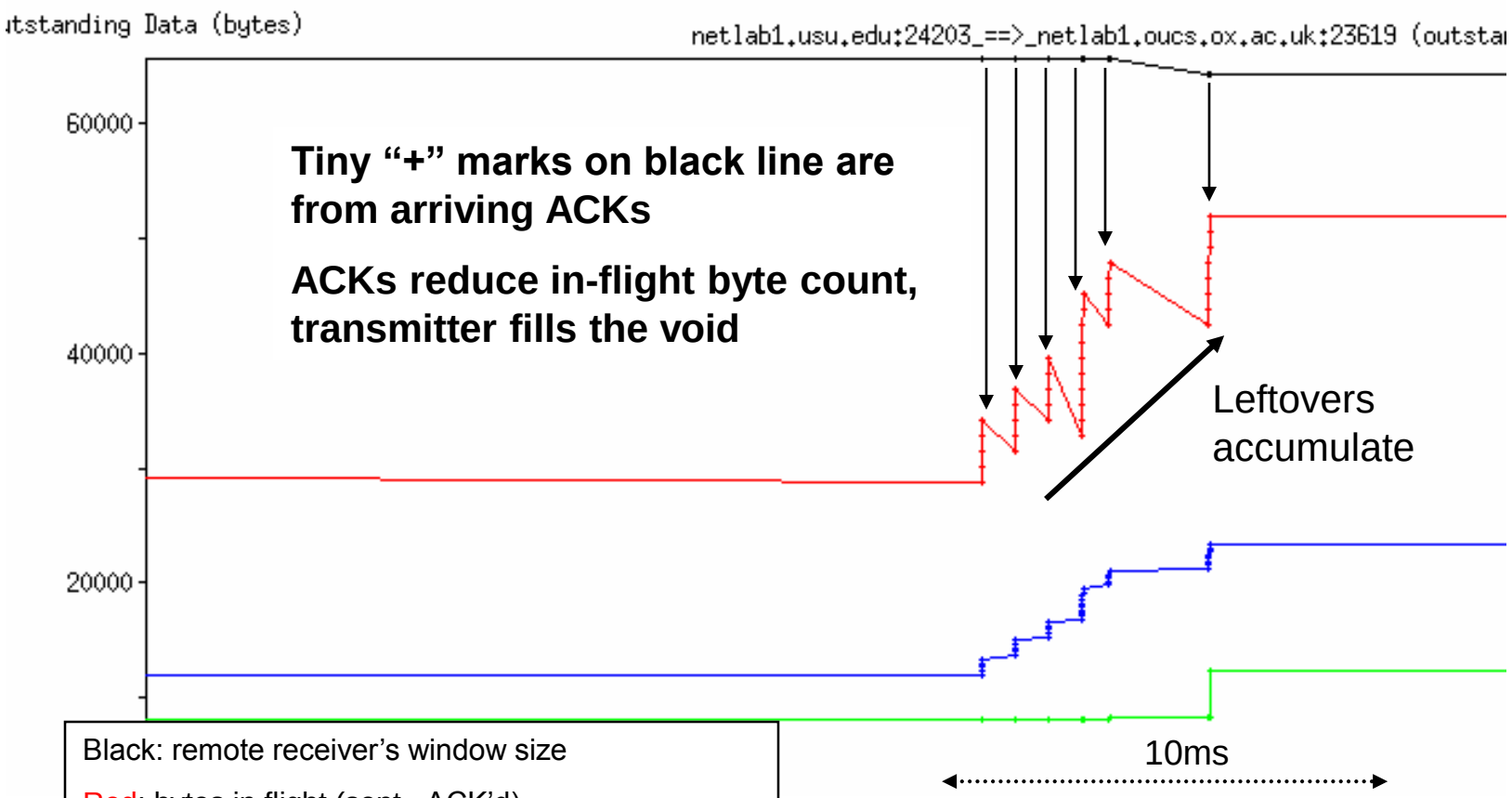
Red: bytes in flight (sent - ACK'd)

Blue: simple running average bytes in flight

Green: long time weighted average bytes in flight

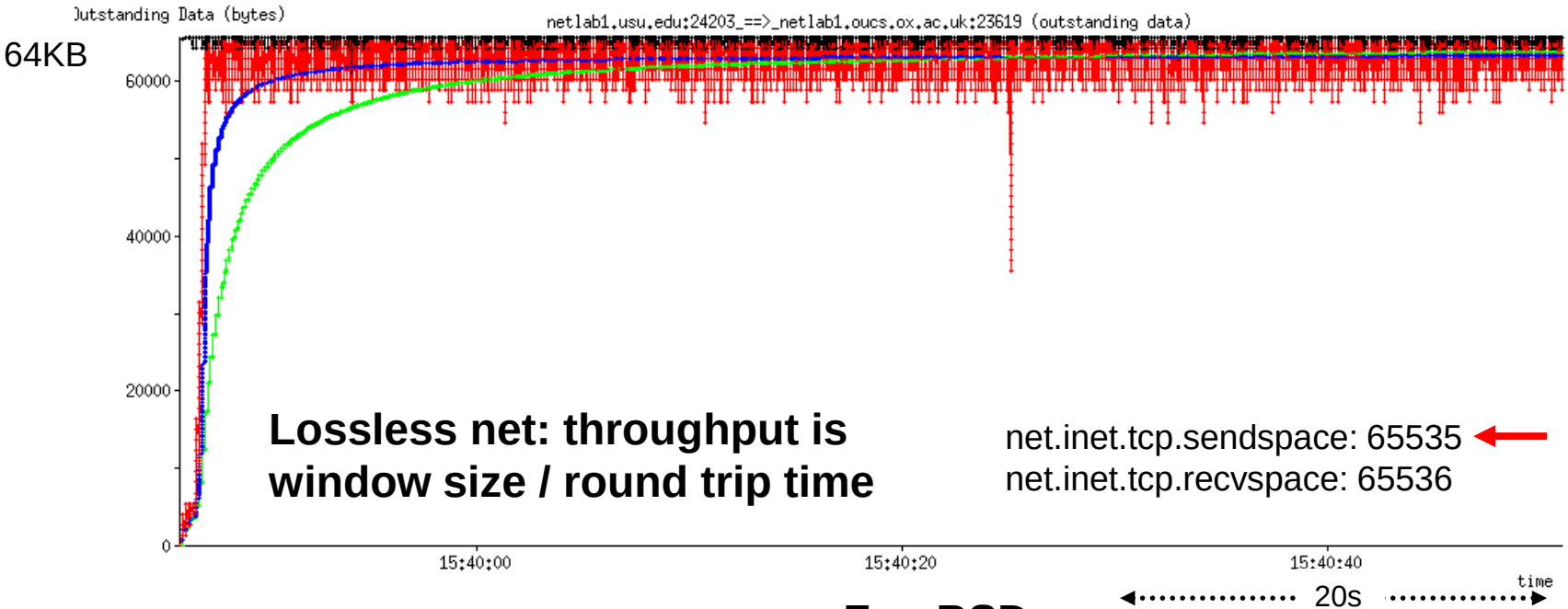
Variation of in-flight byte count is small, most bytes are stored in the pipe, delivery delay is throttle

Finer detail on arriving ACKs



Utah to Oxford, via Janet2

Doubling transmit buffer helps fill the pipe, doubles throughput



Lossless net: throughput is window size / round trip time

net.inet.tcp.sendspace: 65535
net.inet.tcp.recvspace: 65536

Black: remote receiver's window size

Red: bytes in flight (sent - ACK'd)

Blue: simple running average bytes in flight

Green: long time weighted average bytes in flight

FreeBSD
485KB/sec sustained
1.2MB buffers would be optimum

Linux resists large transmit buffers, gets 165KB/sec

Deductions from experiments

Utah-Oxford, excellent long fat pipe, speed limited by delay: Tx buffer empties well before pipe fills

Utah-Manchester, very noisy slow local link in Man, limited by packet loss (timeout, retransmission)

Manchester-Utah, near perfect behavior, interface speed protected net from frequent overloads

Utah-Utah (four feet), limited by buffer emptying

Simple thoughts

- **Slow start is really quick at filling the net**
- **Congestion avoidance is effective, but clearly could use a better design with a longer memory of the network and faster recovery from loss**
- **The more bytes in-flight the longer it takes to build back to full rate after a timeout : more 1/cwin steps and the more packets in succeeding steps**
- **Fast transmit fast recovery helps by avoiding waiting on timeouts to sense trouble and not restarting network learning from scratch**

Simple thoughts

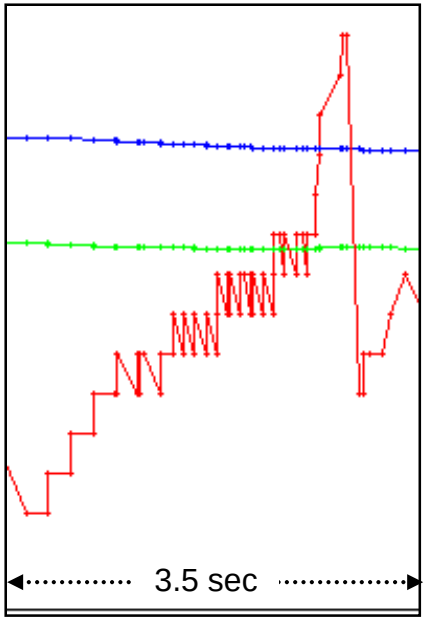
Transmitter sends all it can in one burst, back to back, which aggravates network overload and timeouts

Arriving ACKs indicate rate of draining of the net

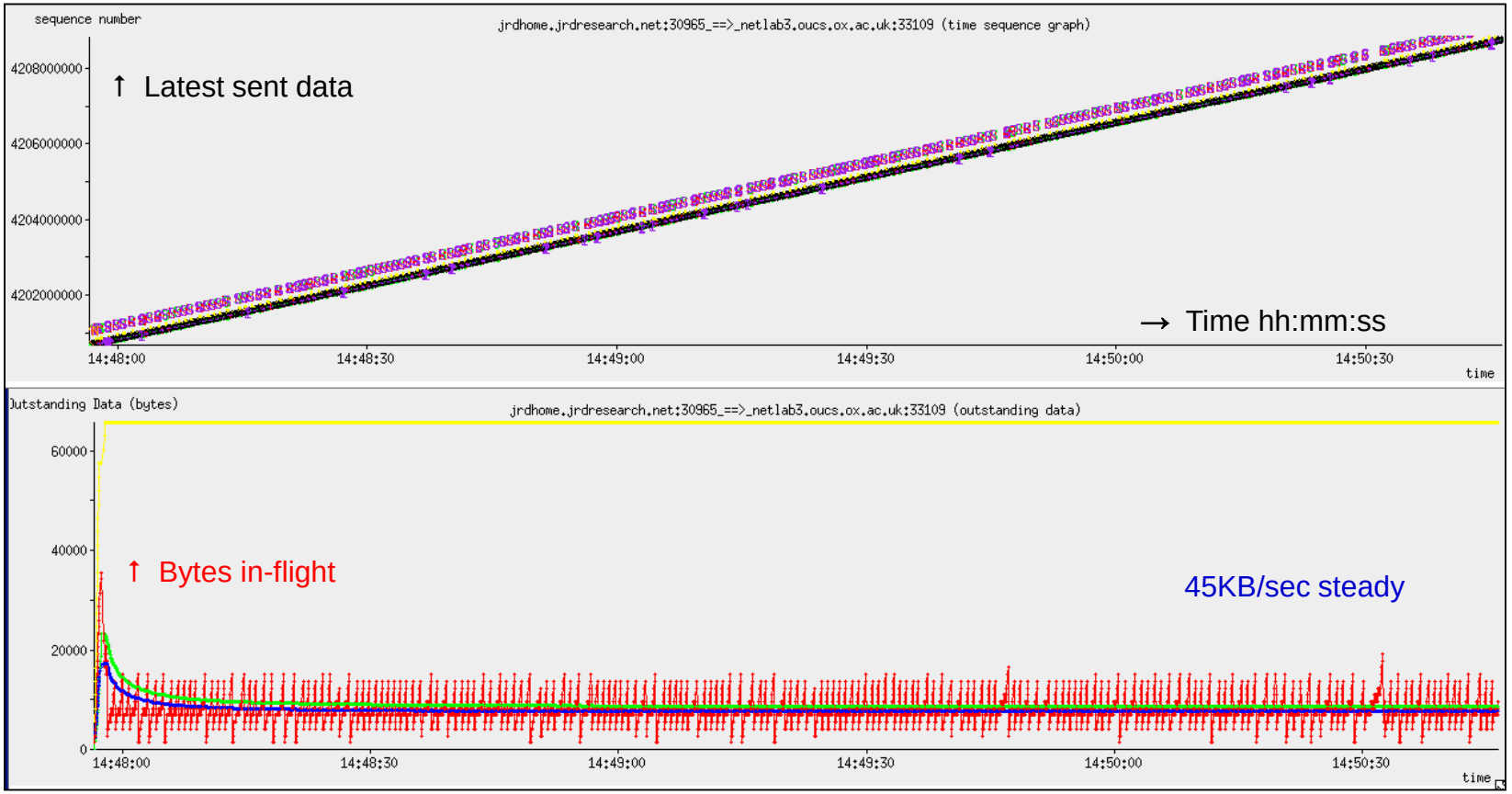
Better would be to pace Transmitter to match ACKs, but that is expensive in kernels. QoS etc.

Pacing does control if waiting for window free space. cwin is the network's window.

At the right, visible delay between packet transmissions is waiting for ACKs while cwin is filled. Down-up steps confirm.

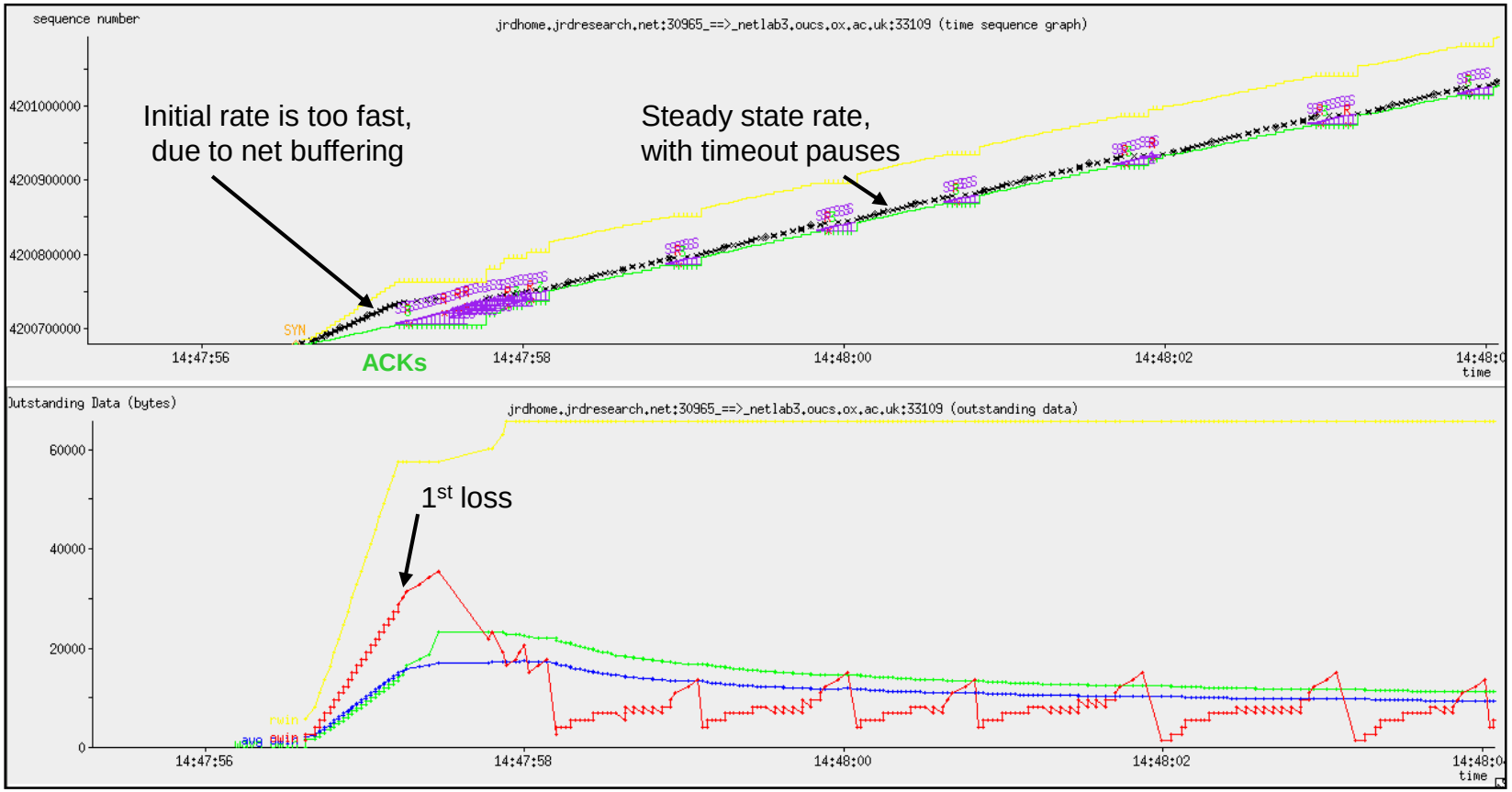


Oxford home to work, two views



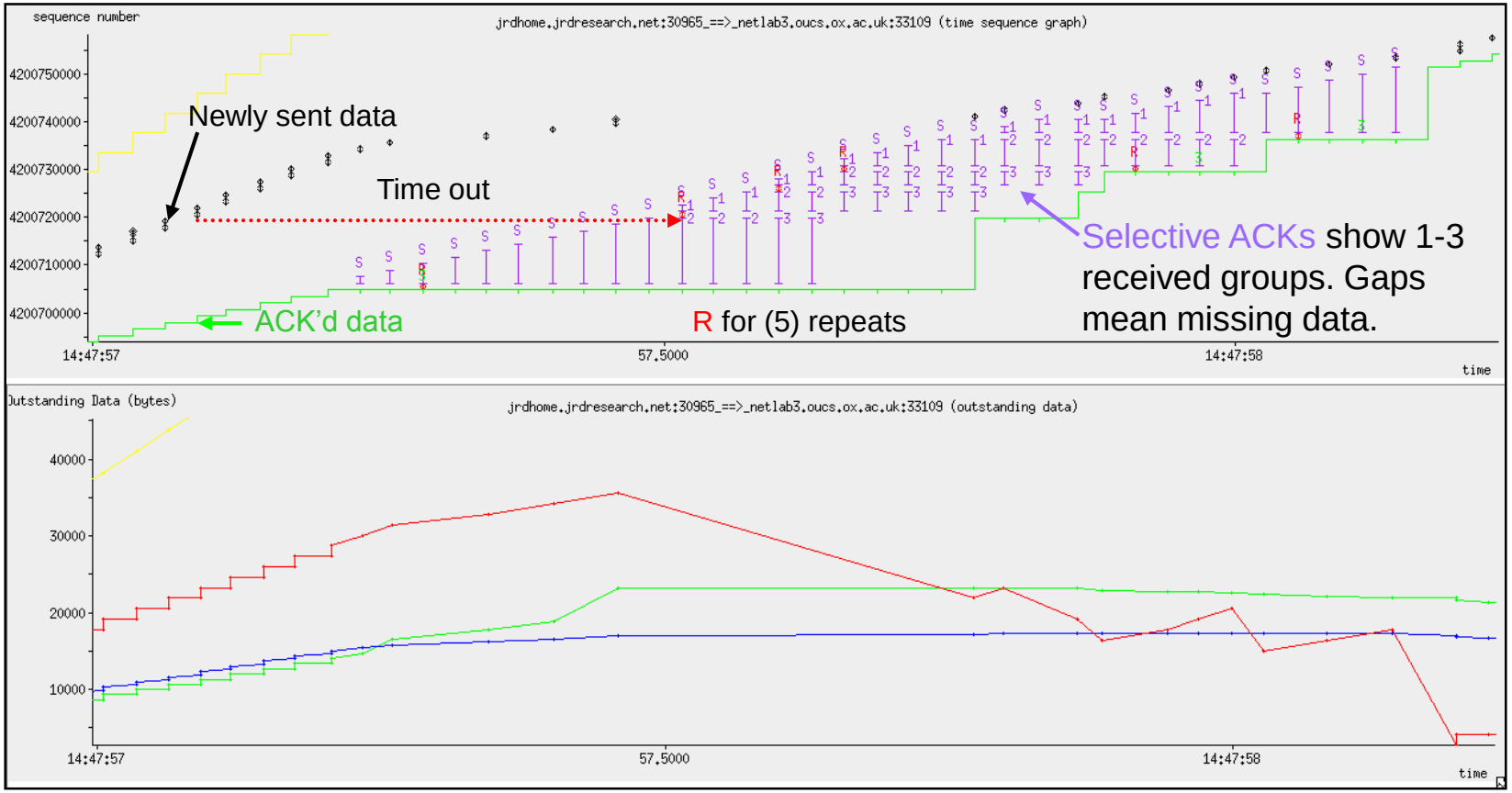
Small net capacity at sender's end, yields packet loss

Home to work, startup details



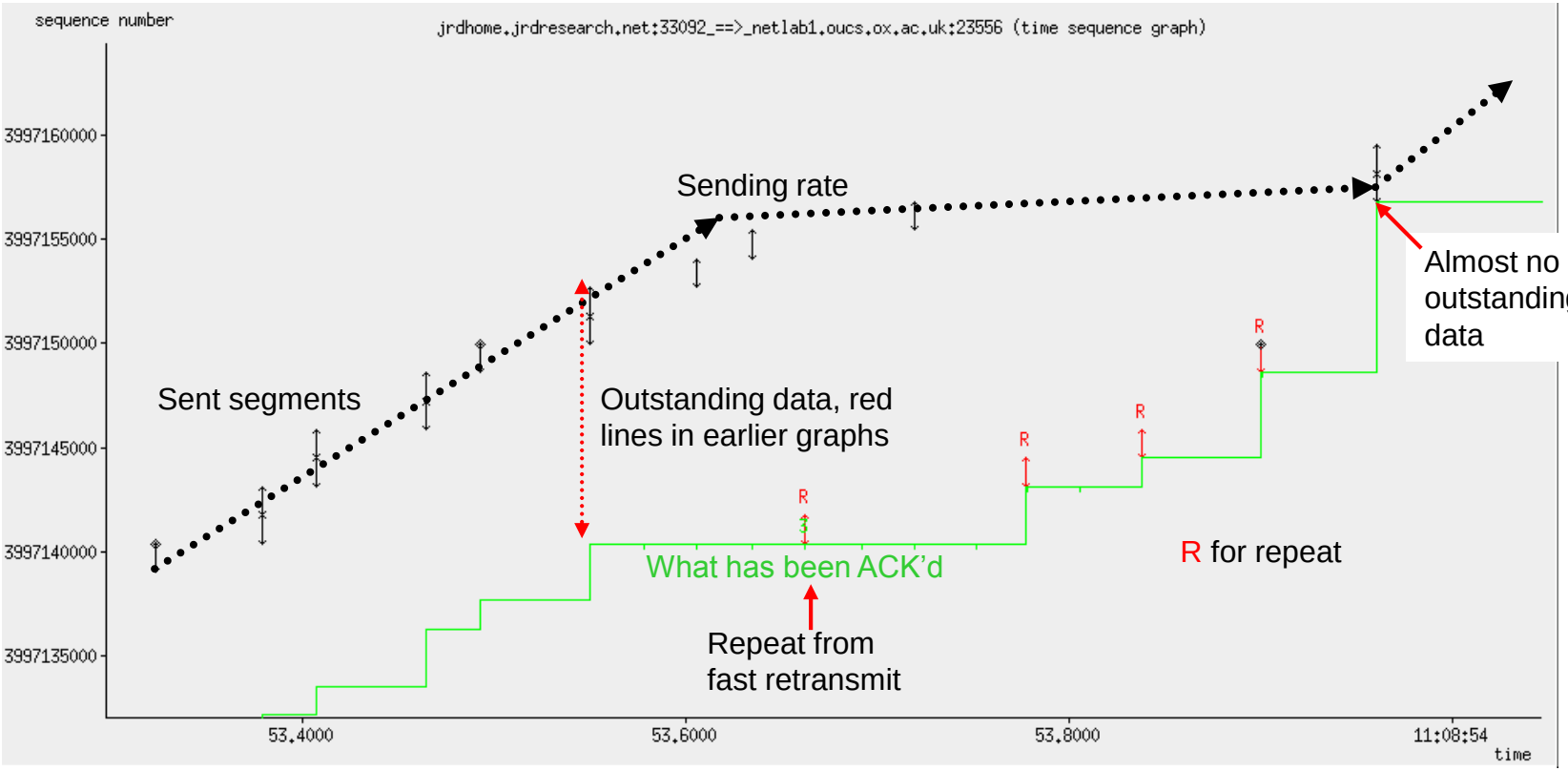
Fast ramp to fill net, loss when net overfilled, repetitive reprobing

Details of first loss recovery



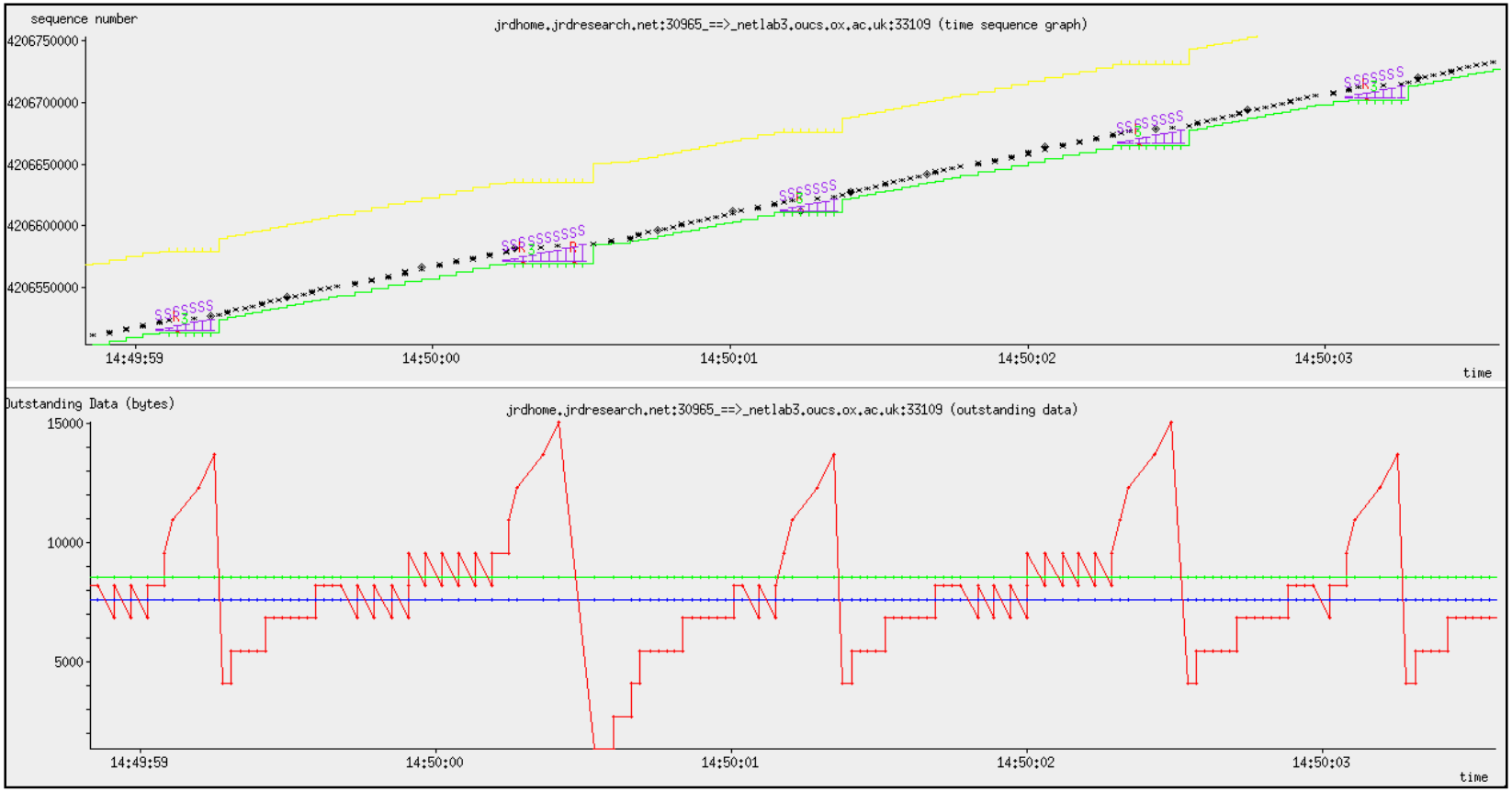
Selective ACK info helps Xmtr avoid resending data buffered by Rcvr. ACK packets have room for only three spans of buffered data

No SACK, more repeats

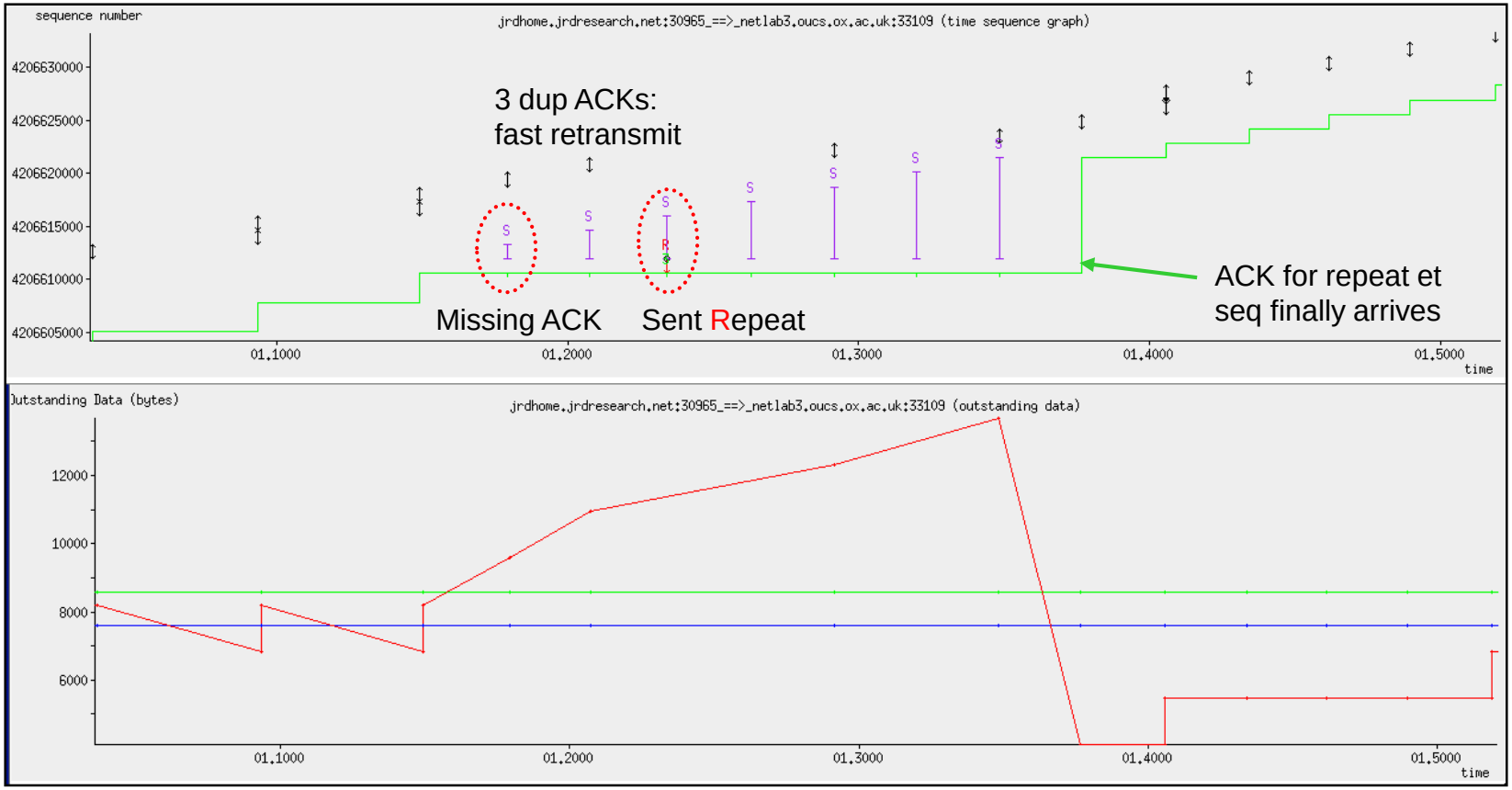


Extra timeouts and retransmissions to discover more missing pieces

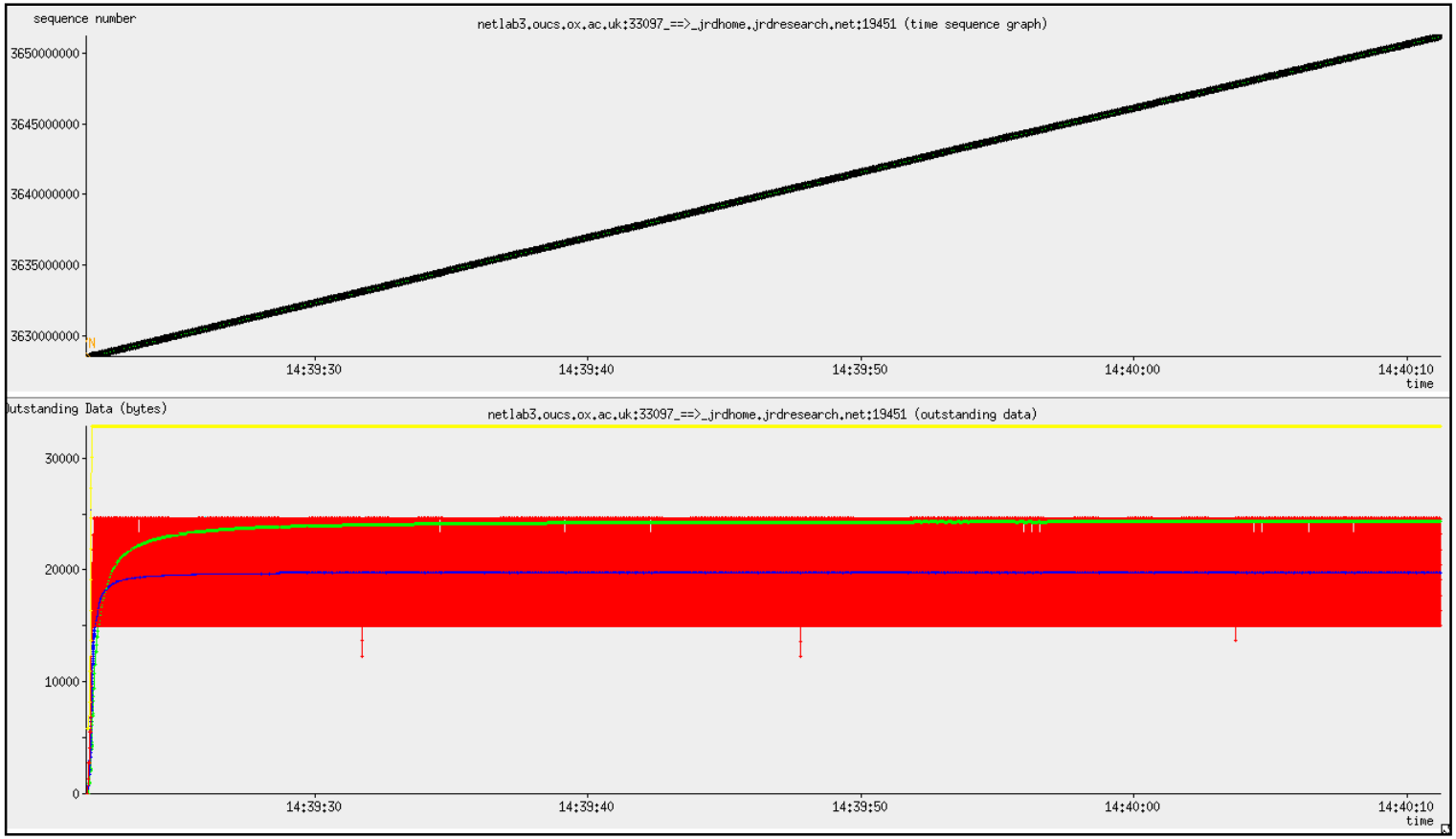
Typical loss/recover sequences



Detail of a typical recovery

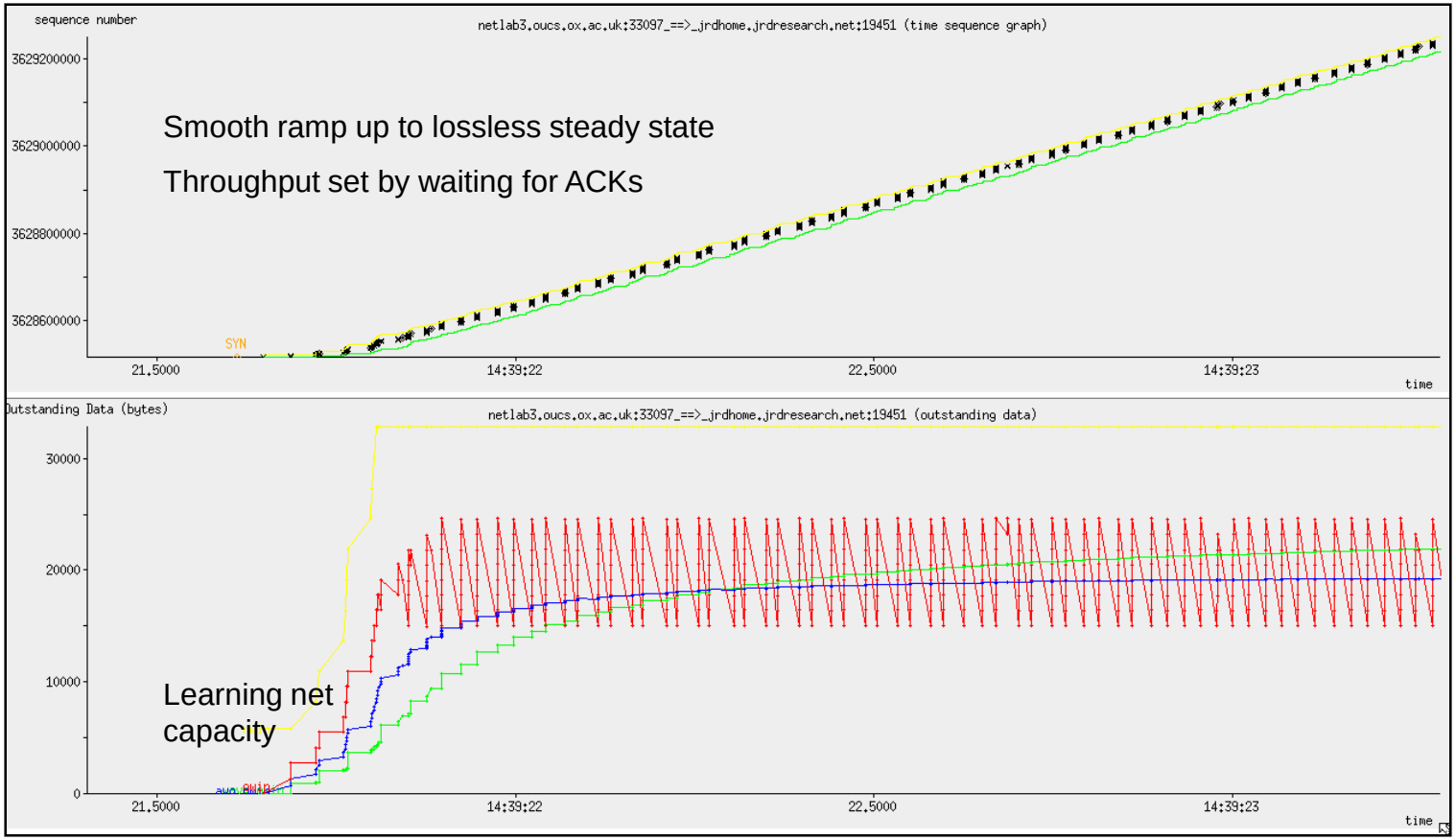


Work to home: smooth, why?



Large network capacity, no loss, slow output, transmitter waits for ACKs

Work to home, startup details



See-saw from ACK bursts, indicative of waiting on (delayed) ACKs

Simple thoughts

- With a clean network throughput is governed by network speed and round trip time
  - If the pipe can be kept full (large enough windows) then the slowest network bit rate is the limit
  - If we can not fill the pipe then round trip time governs by waiting for ACKs to release new data: window size / rtt
- With a congested network, throughput depends upon recovery time from losses
- Shorter round trip times mean quicker ACK ticks, shorter timeout values, thus quicker refilling of the network
- All because events are paced by ACKs

Lessons from TCP heuristics

- Data transfer mostly self clocking, adapt to the net
- Fill the pipe quickly with slow start, use congestion avoidance to keep nibbling at net capacity
- Back away from trouble exponentially fast, else the net may go unstable (can't show this, trust me..)
- Fast retransmit is a good idea, but limited to one loss
- The more data that is stored in the net the longer it takes to fully recover from a timeout
- The same mechanisms work on lossless as well as horrid links, no hand tuning is required

Final comments

TCP performance characteristics on long distance very high bandwidth links is of much interest to the scientific community because recovery times from loss can be prohibitive

Various improvements have been proposed, but narrowly focused ones are not suitable for general application

The problem of robust quick recovery from packet loss remains an open topic

An appendix and references follow

Appendix: experimental technique

On sending machine:

tcpdump -p -tt -S -w outfile.dump

Or Ethereal, but it is unstable with large captures

ftp to remote, send a file, ^C both to finish early

tcptrace -G outfile.dump (-G for create all graphs)

Each flow is written to a separate file, choose ftp data flow for plotting

xplot -1 -x a2b_owin.xpl a2b_tsg.xpl

(in X window draw two graphs synchronised in time)

References

From RFC 2001 “TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms” by Richard Stevens:

[1] B. Braden, ed., "Requirements for Internet Hosts -- Communication Layers," RFC 1122, Oct. 1989.

[2] V. Jacobson, "Congestion Avoidance and Control," Computer Communication Review, vol. 18, no. 4, pp. 314-329, Aug. 1988. ftp://ftp.ee.lbl.gov/papers/congavoid.ps.Z.

[3] V. Jacobson, "Modified TCP Congestion Avoidance Algorithm," end2end-interest mailing list, April 30, 1990. ftp://ftp.isi.edu/end2end/end2end-interest-1990.mail.

[4] W. R. Stevens, "TCP/IP Illustrated, Volume 1: The Protocols", Addison-Wesley, 1994.

[5] G. R. Wright, W. R. Stevens, "TCP/IP Illustrated, Volume 2: The Implementation", Addison-Wesley, 1995.

MindWorks UK

MindWorks Inc. Ltd
210 Burnley Road
Weir
Bacup
OL13 8QE UK

Telephone: +44 (0) 170 687 1900

Fax: +44 (0) 170 687 8203

Web: www.mindworksuk.com

Email: training@mindworksuk.com