TCP/IP Protocols

Transmission Control Protocol / Internet Protocol

The purpose of this presentation is to show that the TCP/IP protocols, and the way that higher layer applications make use of them, can have significant impacts on data flow throughput.

Often, when “the application is slow”, the correct fix is to modify the application and/or the TCP settings. This fact is not usually readily apparent.

We discuss the behaviours of TCP/IP packet flows under various circumstances and define a variety of protocol terms. We also compare the original TCP and “new” TCP performance differences.

The effects of packet losses – and how TCP/IP handles them - will also be discussed and presented.

Small packet losses can have an un-expectedly large impact on throughput.

The NetData Pro packet analysis software tool is used to display the transaction and packet flow data.

Phil Storey
Phil@NetworkDetective.com.au

Real World Performance Problem

A question appeared on the site “Ask Wireshark”.

https://ask.wireshark.org/questions/55972/slow-writes-even-slower-reads-spanning-wan-to-netapp

The problem had been occurring for a long time, happened the same way every time and nobody knew what was the cause.

ask.wireshark.org/questions/55972/slow-writes-even-slower-reads-spanning-wan-to-netapp?page=1&focusedAnswerId=59542#59

WIRESHARK

Questions | Tags | Users | Badges | Unanswered

Search

Questions Tags Users

slow writes, even slower reads spanning WAN to Netapp

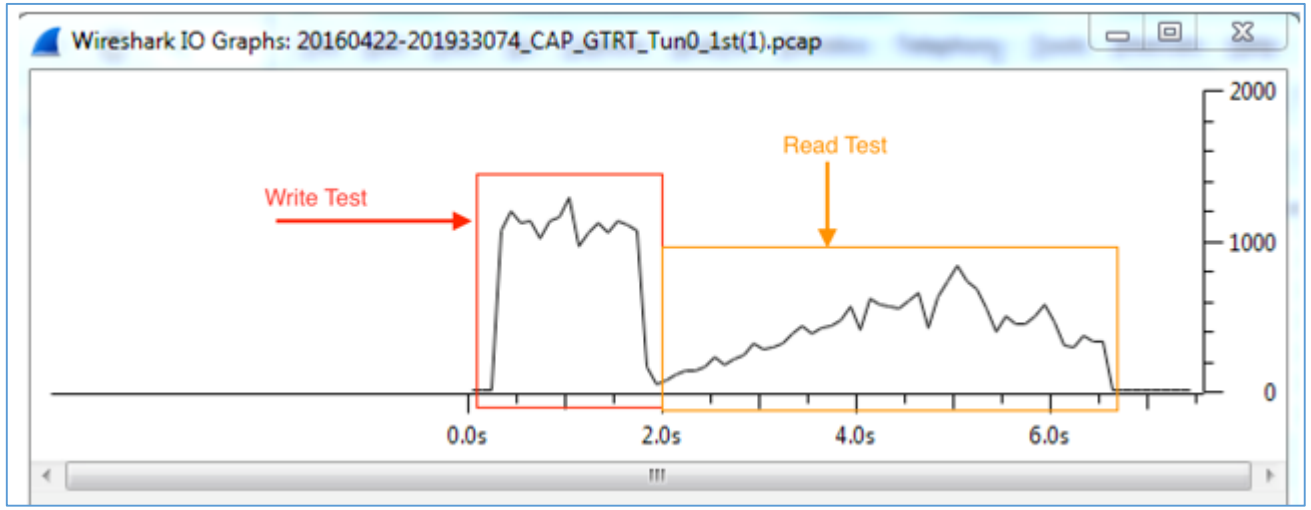
Win7 workstation -> LAN -> ASA -> Cisco ASR -> DMVPN -> ASR -> Palo Alto -> Nexus -> NetApp

2

We are experiencing the symptoms described in the title. This is not new, it predates me, and it happens at multiple spoke sites in our DMVPN. Each vendor just seems to point the finger at the other with no real data reinforcing their point. Cisco has cleared any real issues at the hardware level.

The symptom was that an SMB file transfer of the exact same file, between the same user PC and the same NetApp file share server, always took longer to download than to upload.

The Wireshark throughput charts showed that the throughput of the download (SMB Read) versus the upload (SMB Write) varied significantly and formed “sawtooth” patterns.

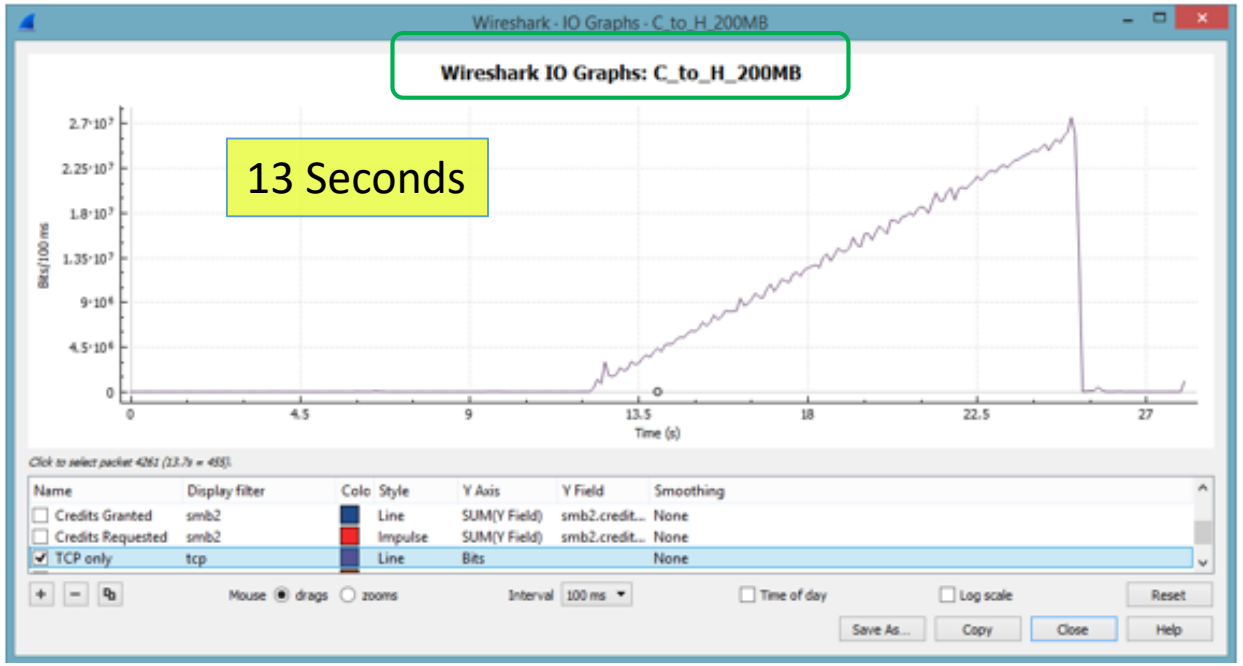


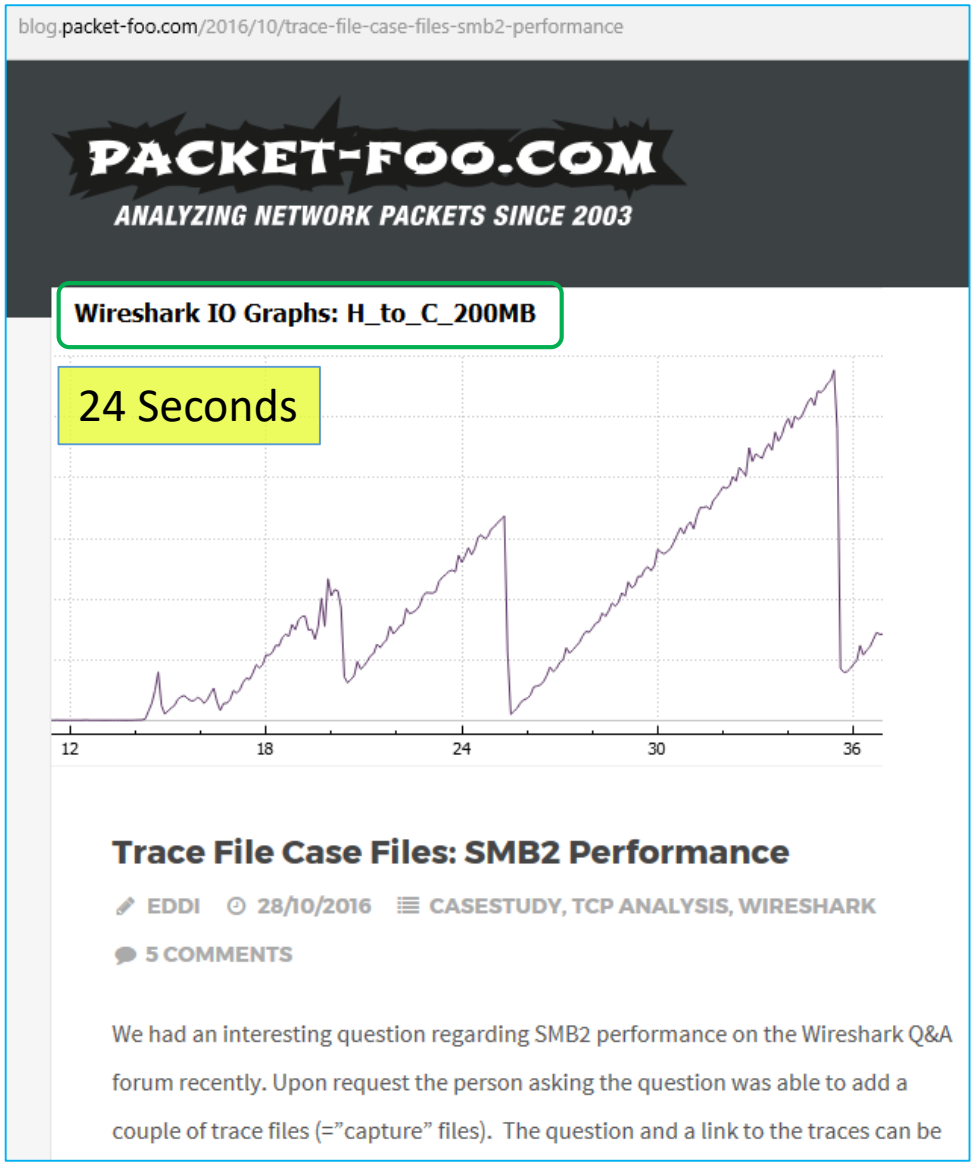
Wireshark Experts Also Looked At It

Wireshark experts in Europe blogged about it.

https://blog.packet-foo.com/2016/10/trace-file-case-files-smb2-performance

Note the multiple sawtooth pattern in the “H-to-C” case (SMB Read) with the 200 MB file taking **24 seconds**. The “C-to-H” case had only one big sawtooth and took just **13 seconds** to transfer the same 200 MB file.



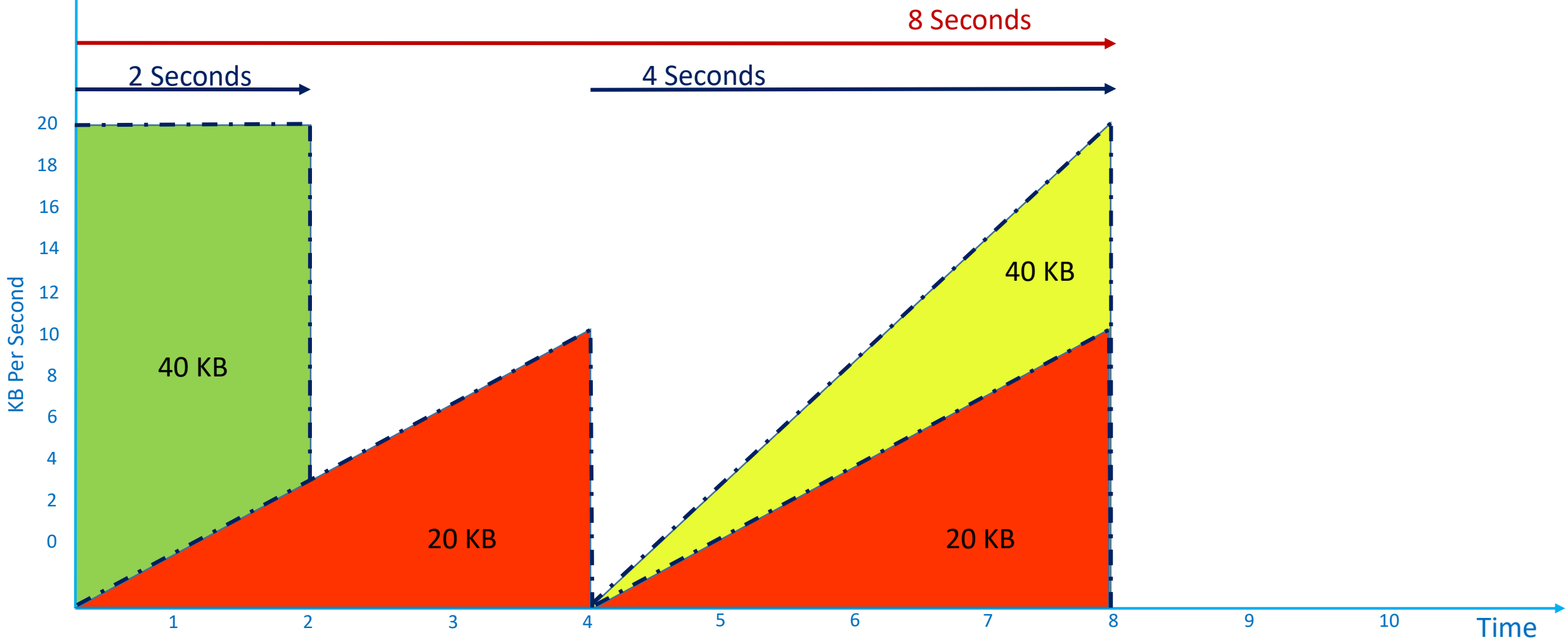


Geometry Refresher

At 20 KB/s, it would take just 2 seconds to transfer 40 KB of data (green).

To transfer the same 40 KB with a linearly increasing throughput rate would take twice as long (yellow). Assuming that we eventually ramp up to the full 20 KB/s.

If we only ever ramp-up to 10 KB/s, and form a sawtooth pattern (red), the overall time doubles yet again.



TCP/IP Protocol “Education”

Before we can run through the live packet analysis, which is fairly complex, we first need to run through some underlying details of:

- How TCP/IP works.
- How to understand the charts and graphs that NetData produces.

I’ve tried to “kill two birds with one stone” by combining the two lessons.

This presentation can be downloaded (as a PDF) from:
http://www.networkdetective.com.au/downloads

TCP/IP Protocol Terminology

These terms are described in this presentation.

“Acknowledgement (ACK)” “Duplicate ACK” “Delayed ACK” “Partial ACK”
“Fast Retransmission” “Retransmission Timeout (RTO)”
“Round Trip Time (RTT)” “Bytes In Flight” “Bandwidth Delay Product (BDP)”
“Receive Window” “Transmit Window” “Window Scaling” “Long, Fat Network (LFN)”
“Slow Start” mode “Congestion Avoidance” mode

“3-Way Handshake” “Maximum Segment Size (MSS)”
“Selective Acknowledgement (SACK)” “Duplicate SACK”

“Out of Order (OOO)”

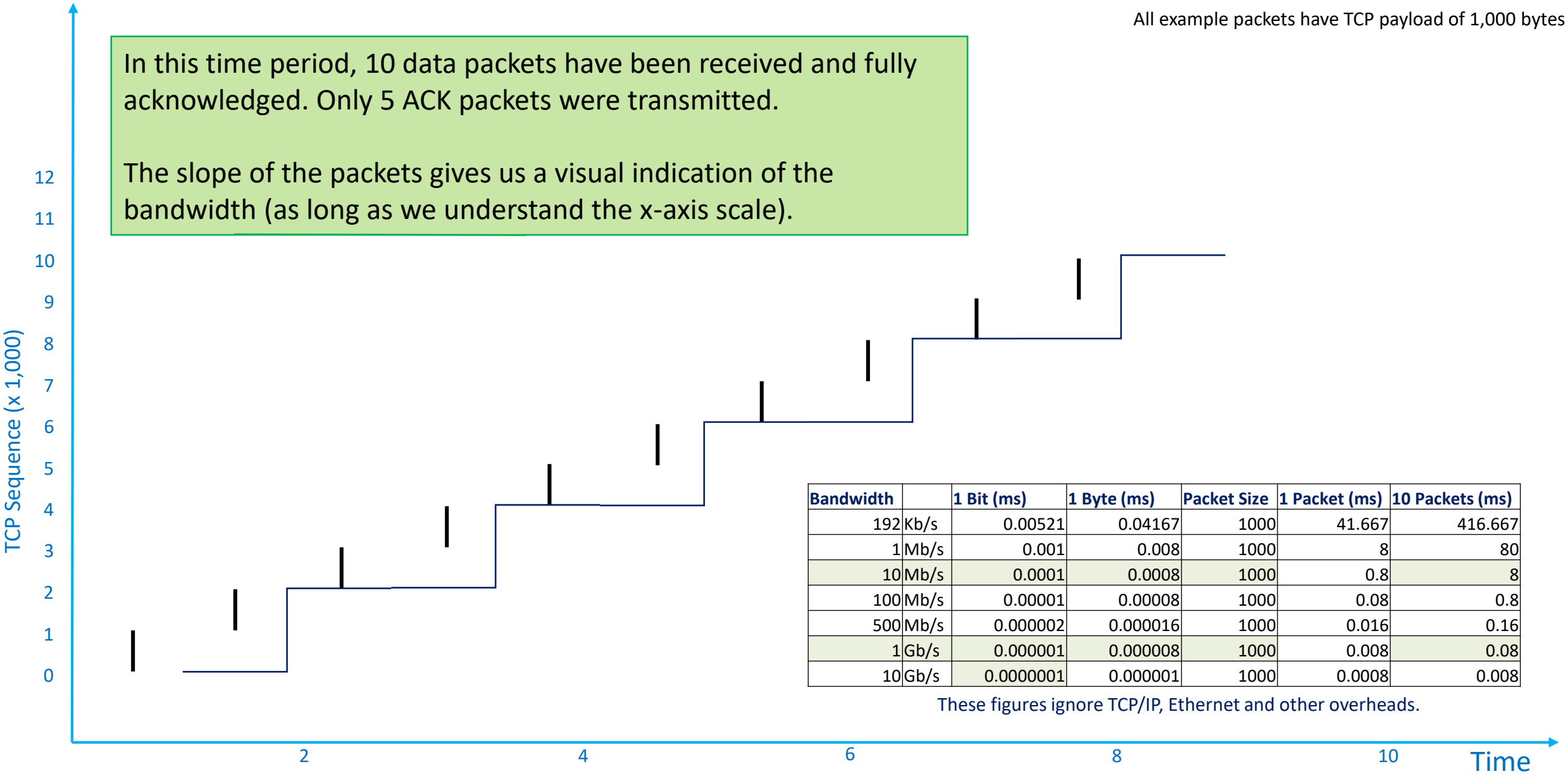
“Window Closure” aka “Zero Window”

Good Flow – Only "Normal" ACKs Needed

All example packets have TCP payload of 1,000 bytes

In this time period, 10 data packets have been received and fully acknowledged. Only 5 ACK packets were transmitted.

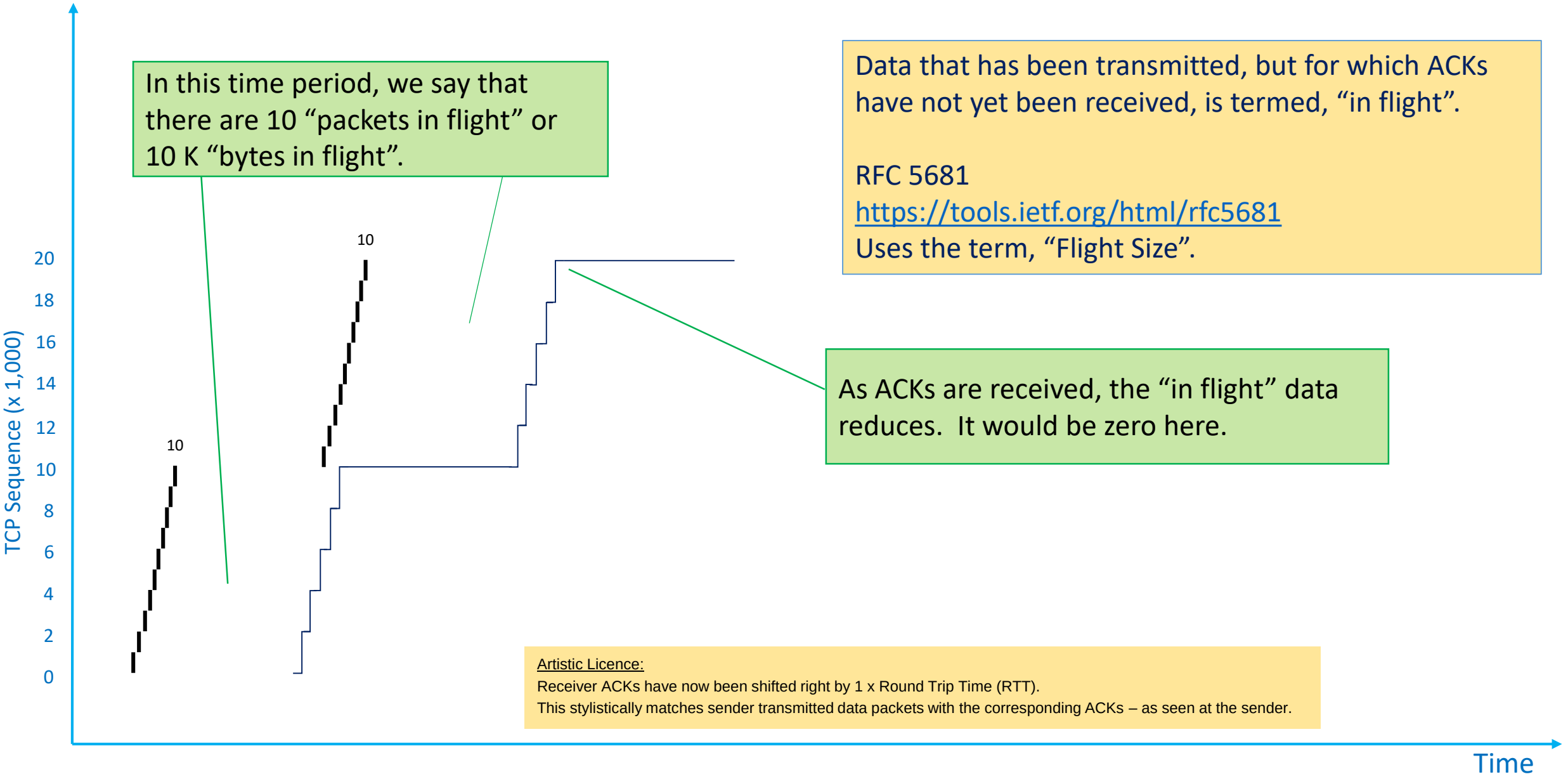
The slope of the packets gives us a visual indication of the bandwidth (as long as we understand the x-axis scale).



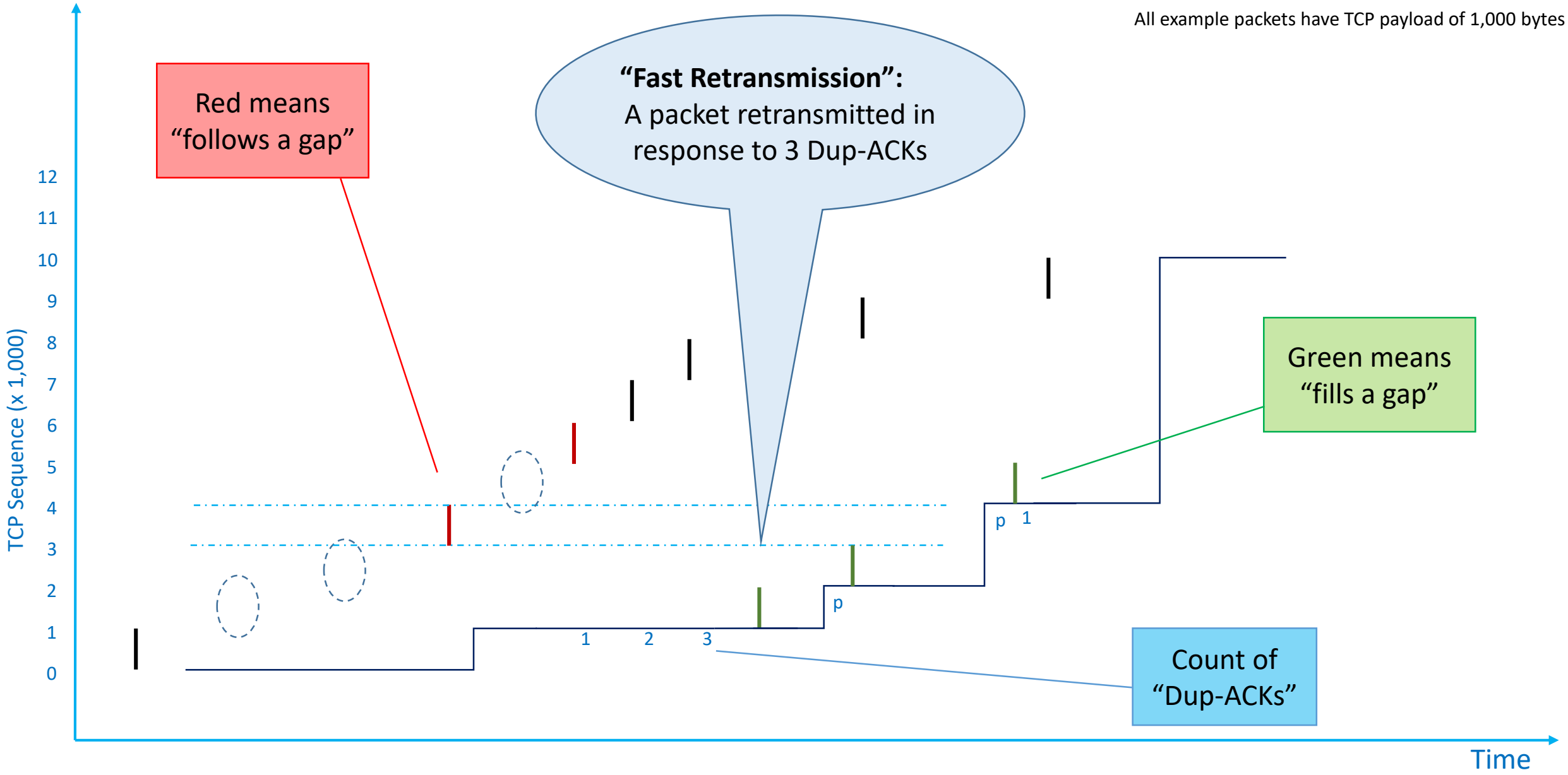
Bandwidth		1 Bit (ms)	1 Byte (ms)	Packet Size	1 Packet (ms)	10 Packets (ms)
192	Kb/s	0.00521	0.04167	1000	41.667	416.667
1	Mb/s	0.001	0.008	1000	8	80
10	Mb/s	0.0001	0.0008	1000	0.8	8
100	Mb/s	0.00001	0.00008	1000	0.08	0.8
500	Mb/s	0.000002	0.000016	1000	0.016	0.16
1	Gb/s	0.000001	0.000008	1000	0.008	0.08
10	Gb/s	0.0000001	0.000001	1000	0.0008	0.008

These figures ignore TCP/IP, Ethernet and other overheads.

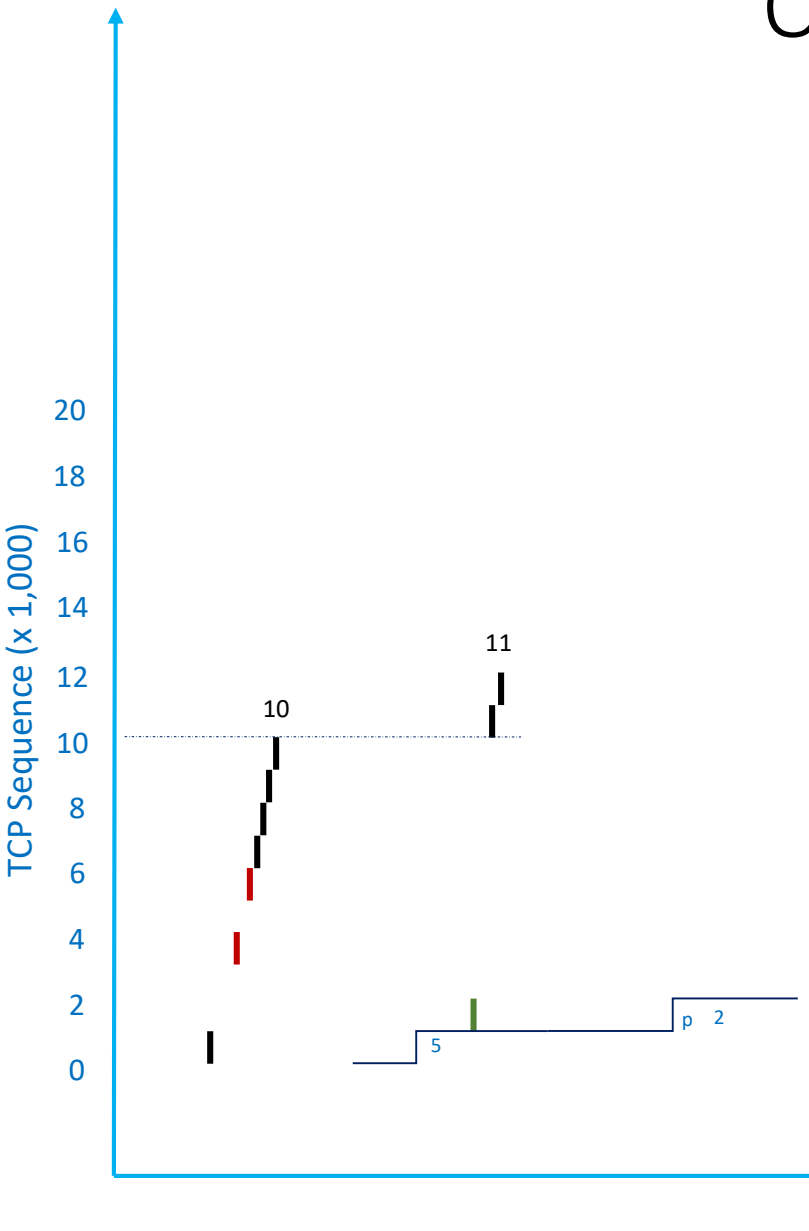
Good Flow – “In Flight” Data



Bad Flow – Only “Normal” ACKs Allowed



Bad Flow – Only "Normal" ACKs Allowed. One Round Trip Time Later



On receipt of our first ACK, the sender knows that we have received the first packet.

The 4 x Dup-ACKs indicate that Packet #2 didn't arrive – but at least 4 (out of the remaining 8) other packets must have arrived (each triggering a Dup-ACK).

However, the sender can't tell how many or which of the last 8 packets went missing.

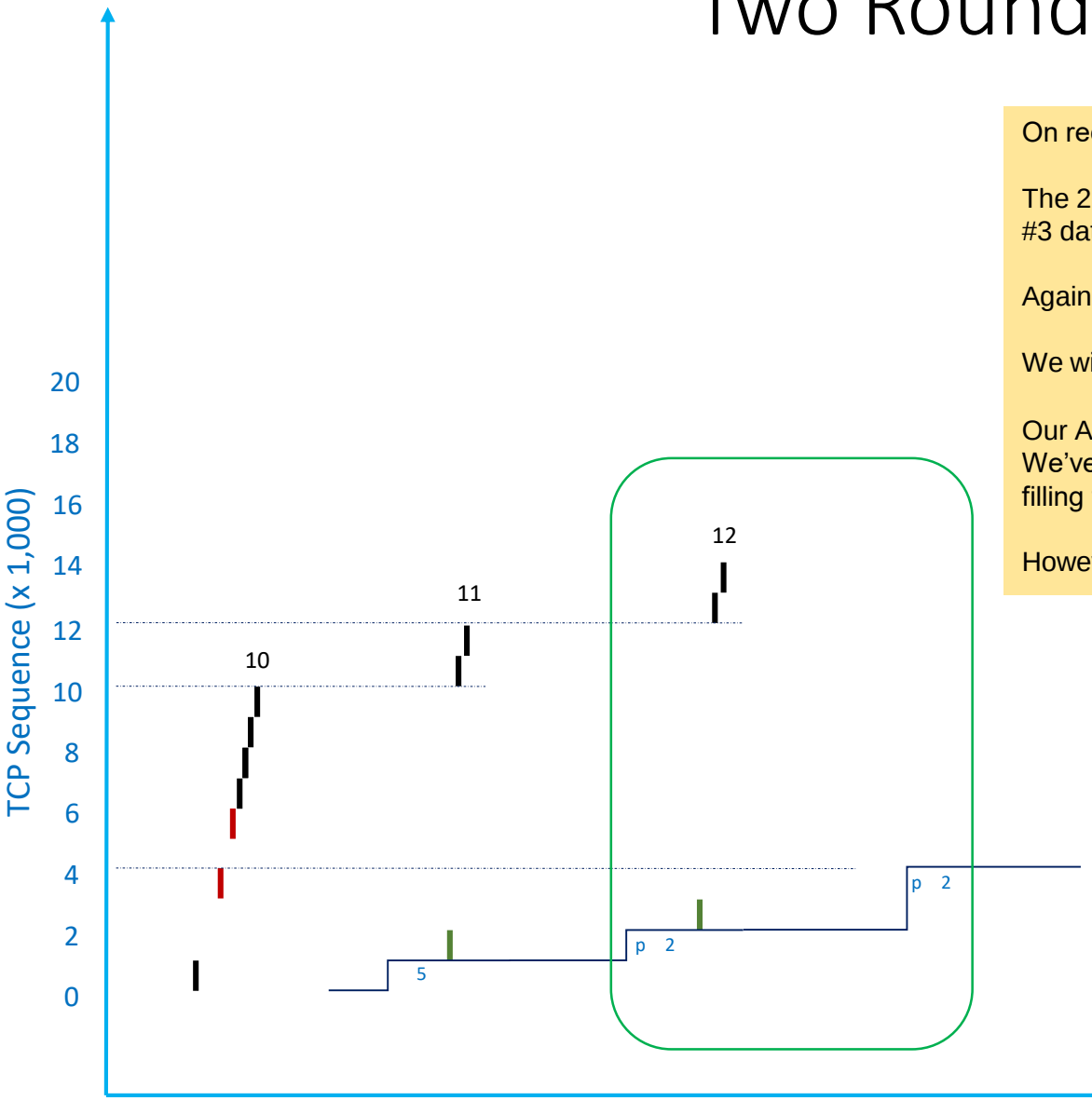
To avoid unnecessary retransmissions, the sender retransmits just the missing Packet #2 data – and may send more data too. 2 new data packets are shown here.

We will receive these 3 new packets one RTT after the first packet burst.

Our response will be to send a normal ACK for Packet #2, followed by 2 x Dup-ACKs – one each for those 2 new data packets.

Our first ACK is known as a "Partial ACK", because we have still not yet acknowledged all outstanding data.

Bad Flow – Only “Normal” ACKs Allowed. Two Round Trip Times Later



On receipt of our “partial” normal ACK, the sender knows that we’ve now received the Packet #2 data.

The 2 x Dup-ACKs now indicate that Packet #3 had never arrived, so the sender retransmits the Packet #3 data (in that new green packet).

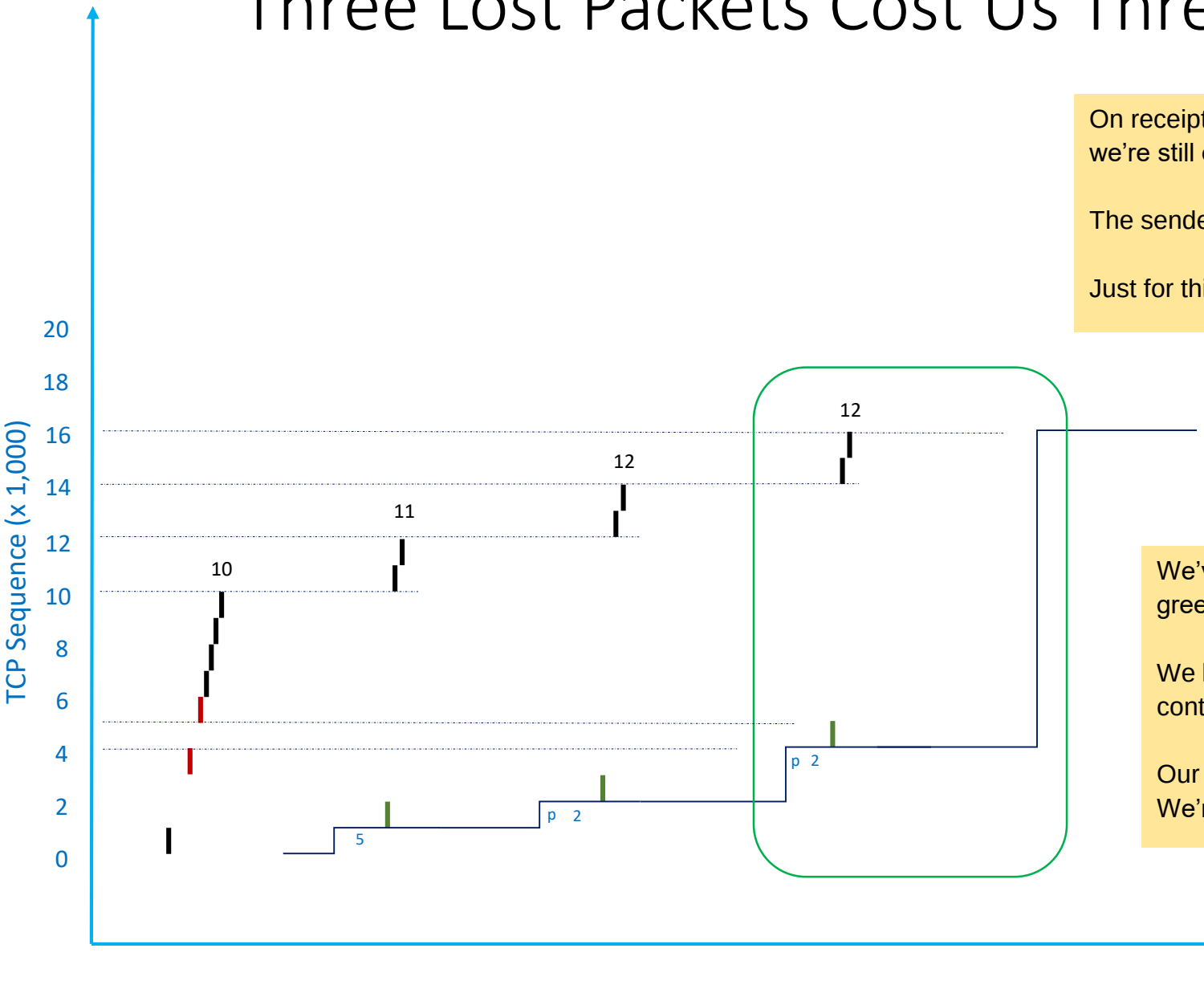
Again, there can be fresh data – so 2 new data packets are shown here.

We will receive these new packets after another RTT.

Our ACK response, shown here, will be to send a normal ACK all the way up to 4,000.
We’ve had the Packet #4 data since the beginning and the newly received green Packet #3 data finishes filling the first original “gap”.

However, we still have the original second gap, so we can only send more Dup-ACKs at the 4,000 level.

Bad Flow – Only “Normal” ACKs Allowed.
Three Lost Packets Cost Us Three Round Trip Times



On receipt of our normal ACK, plus the two Dup-ACKs, the sender knows that we’re still only up to 4,000.

The sender retransmits the Packet #5 data (in yet another new green packet).

Just for this example, we continue with 2 packets of fresh data.

We’ve had the Packet #6 data since the beginning and the newly received green Packet #5 data finishes filling the second original “gap”.

We have now received a total of 16 packets, containing 16 KB of contiguous data.

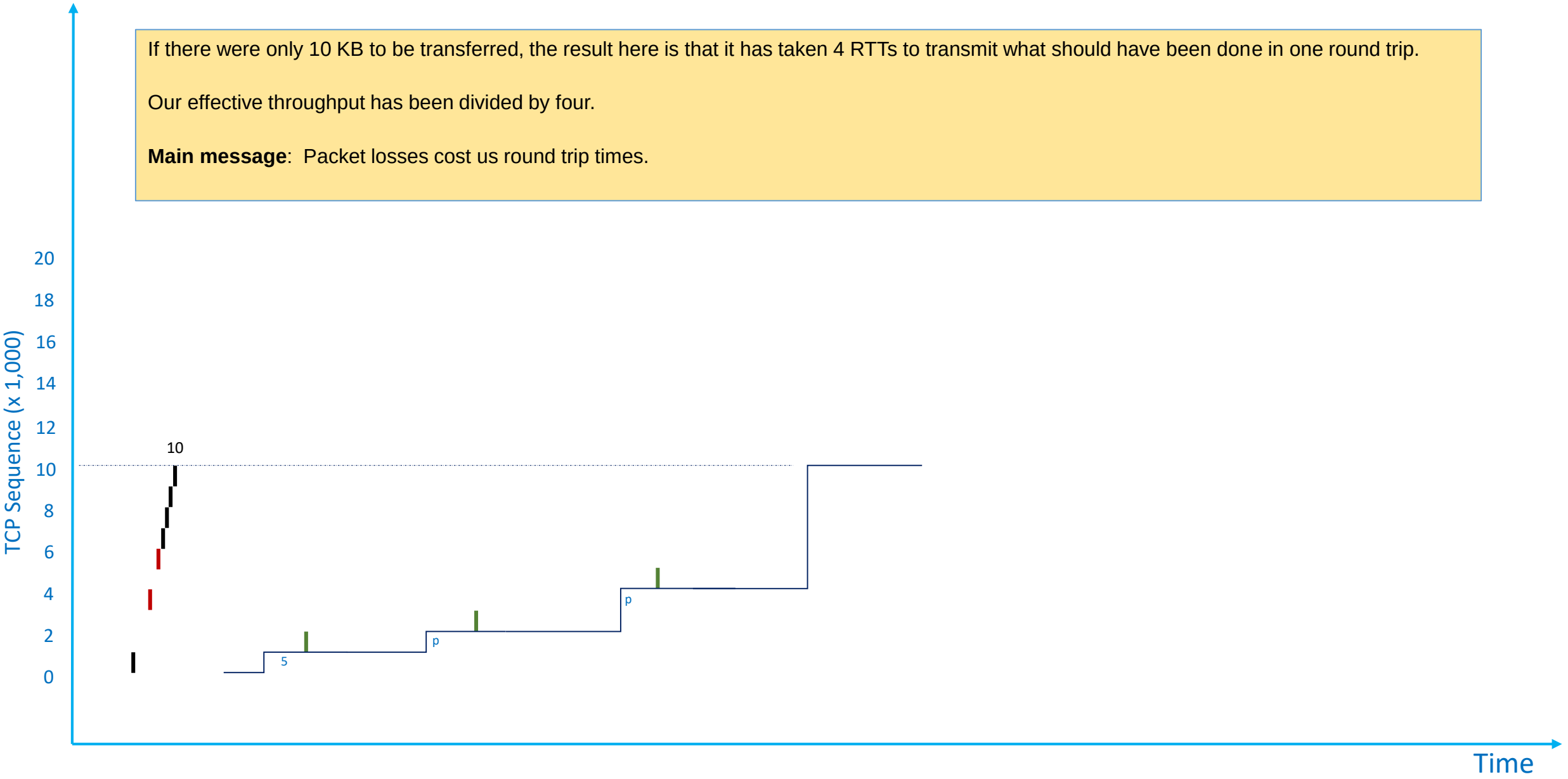
Our ACK response will be to send a normal ACK all the way up to 16,000. We’re now “fully ACKed” and back to normal.

If There Was Only 10 KB.

If there were only 10 KB to be transferred, the result here is that it has taken 4 RTTs to transmit what should have been done in one round trip.

Our effective throughput has been divided by four.

Main message: Packet losses cost us round trip times.



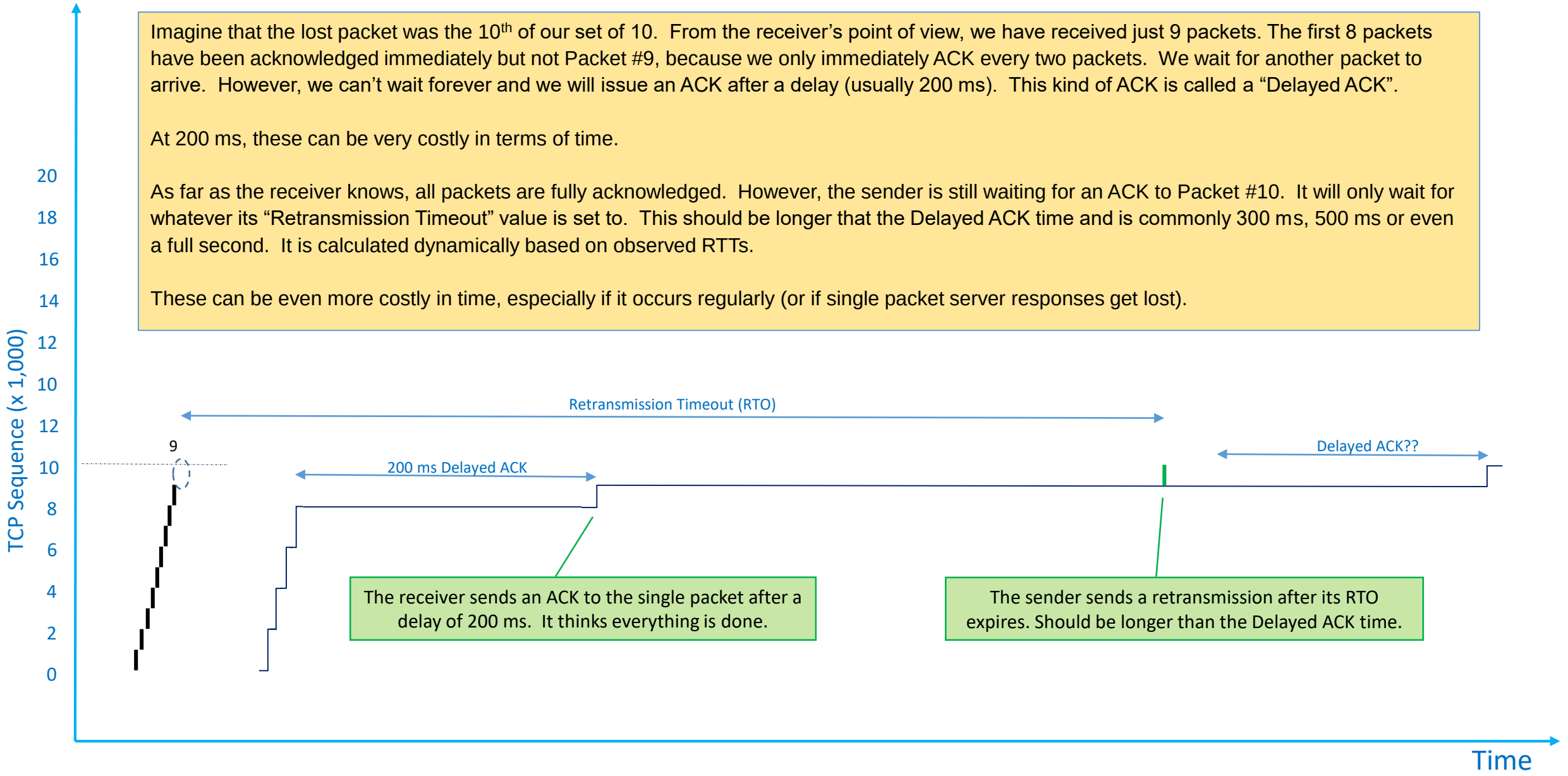
Delayed ACK and Retransmission Timeout (RTO)

Imagine that the lost packet was the 10th of our set of 10. From the receiver's point of view, we have received just 9 packets. The first 8 packets have been acknowledged immediately but not Packet #9, because we only immediately ACK every two packets. We wait for another packet to arrive. However, we can't wait forever and we will issue an ACK after a delay (usually 200 ms). This kind of ACK is called a "Delayed ACK".

At 200 ms, these can be very costly in terms of time.

As far as the receiver knows, all packets are fully acknowledged. However, the sender is still waiting for an ACK to Packet #10. It will only wait for whatever its "Retransmission Timeout" value is set to. This should be longer that the Delayed ACK time and is commonly 300 ms, 500 ms or even a full second. It is calculated dynamically based on observed RTTs.

These can be even more costly in time, especially if it occurs regularly (or if single packet server responses get lost).



TCP “Slow Start”

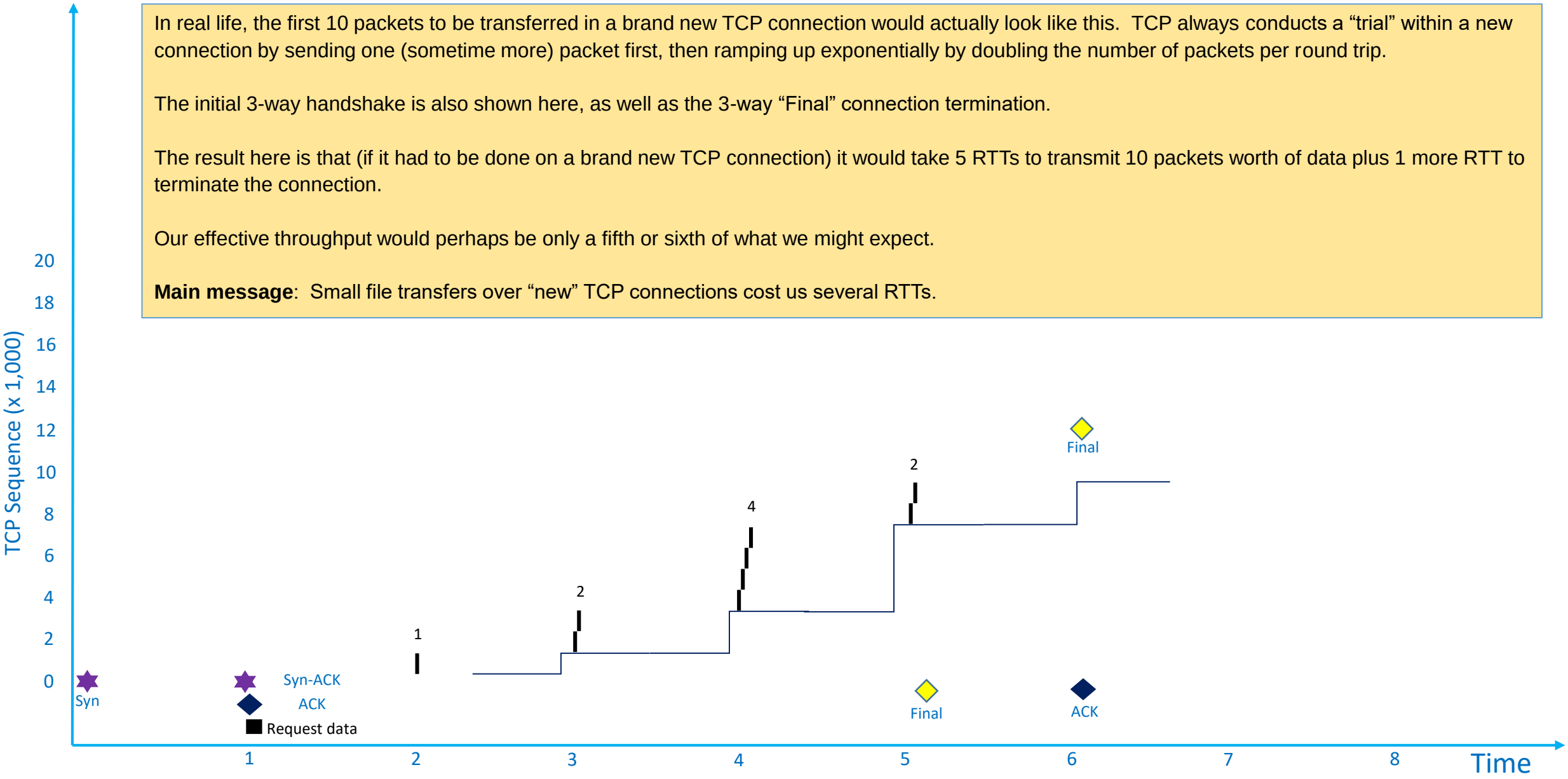
In real life, the first 10 packets to be transferred in a brand new TCP connection would actually look like this. TCP always conducts a “trial” within a new connection by sending one (sometime more) packet first, then ramping up exponentially by doubling the number of packets per round trip.

The initial 3-way handshake is also shown here, as well as the 3-way “Final” connection termination.

The result here is that (if it had to be done on a brand new TCP connection) it would take 5 RTTs to transmit 10 packets worth of data plus 1 more RTT to terminate the connection.

Our effective throughput would perhaps be only a fifth or sixth of what we might expect.

Main message: Small file transfers over “new” TCP connections cost us several RTTs.



TCP/IP Protocol Terminology

Where are we up to?

“Acknowledgement (ACK)” “Duplicate ACK” “Delayed ACK” “Partial ACK”
“Fast Retransmission” “Retransmission Timeout (RTO)”
“Round Trip Time (RTT)” “Bytes In Flight” “Bandwidth Delay Product (BDP)”
“Receive Window” “Transmit Window” “Window Scaling” “Long, Fat Network (LFN)”
“Slow Start” mode “Congestion Avoidance” mode

“3-Way Handshake” “Maximum Segment Size (MSS)”
“Selective Acknowledgement (SACK)” “Duplicate SACK”

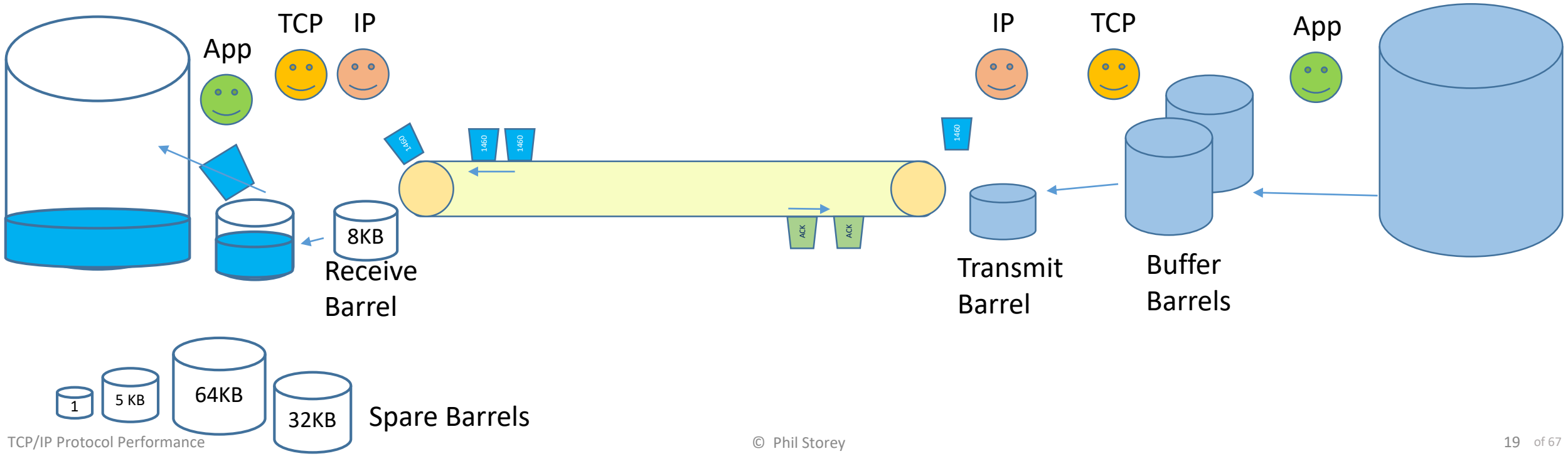
“Out of Order (OOO)”

“Window Closure” aka “Zero Window”

Buckets and Barrels

At the receiver: The buckets fill up a "Receive Barrel" which is of a size determined by TCP/IP and can be changed at any time.
The application removes data from the Receive Barrel at its own pace.

At the sender: TCP receives large blocks of data from the sending application. It assigns sequence numbers and breaks the data into smaller buckets (segments). It is IP's job to put the buckets onto the conveyor but TCP's job to track sequence numbers.

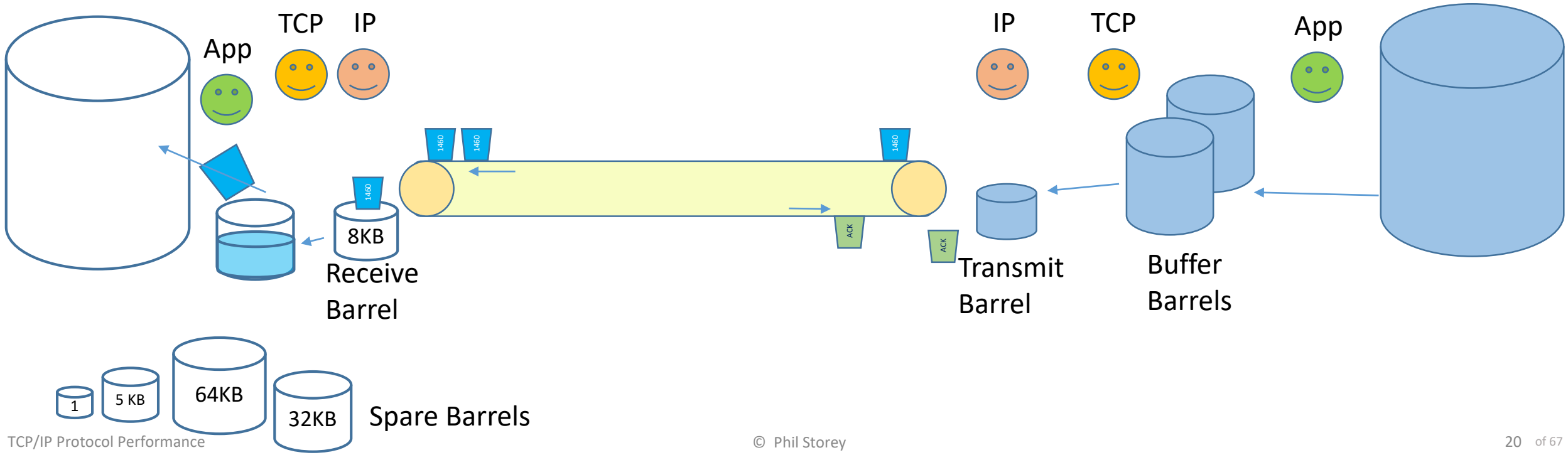


Buckets and Barrels

The sender cannot put too many buckets onto the conveyor at once, because the Receive Barrel cannot be allowed to overflow. The receiver can vary the size and informs the sender of the current size of the Receive Barrel with every ACK (or data) bucket.
TCP can acknowledge the bucket data even though the application has not yet taken it.

More buckets can be placed on the belt as ACK buckets arrive back from the receiver. Buckets that have been placed on the belt but which have not yet been ACKed are called “In Flight”.

The sender can choose to send fewer – but it must never have more “in flight” data than the allowed Receive Barrel size.



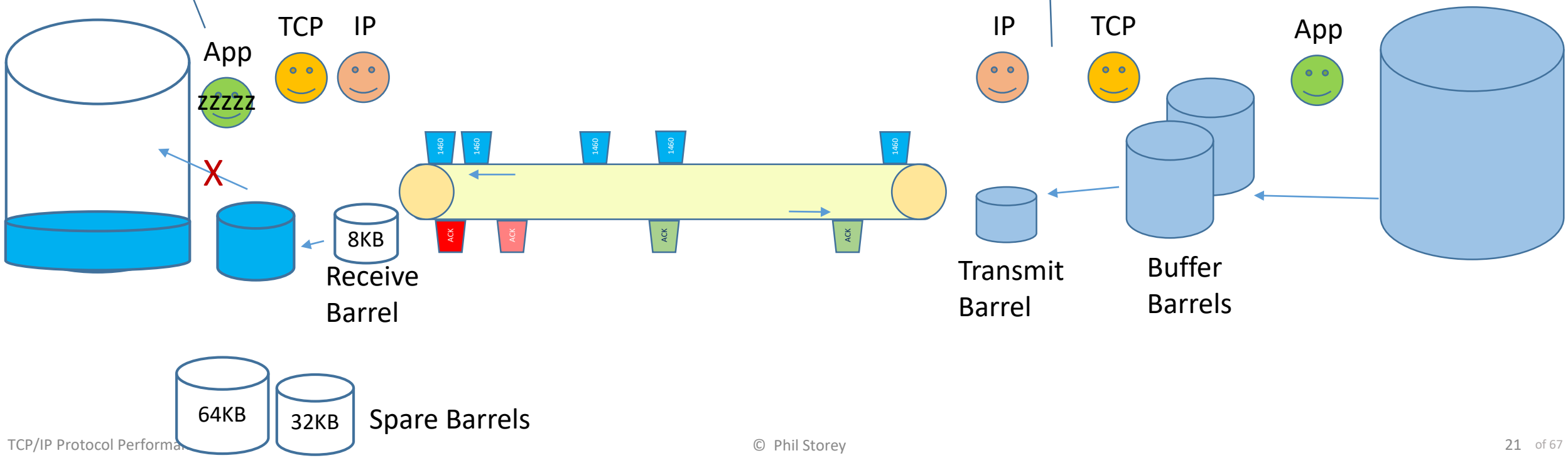
Buckets and Barrels

If the receiver's application stops "emptying" the receive buffers. The receiver's TCP will begin to reduce the advertised Receive Barrel size and, if necessary, advertise a "Zero Barrel" size – which we term, "Closing the Receive Window".

When the receiver does this, it needs to make sure that it still has enough Receive Barrel capacity to catch the buckets already "in flight".

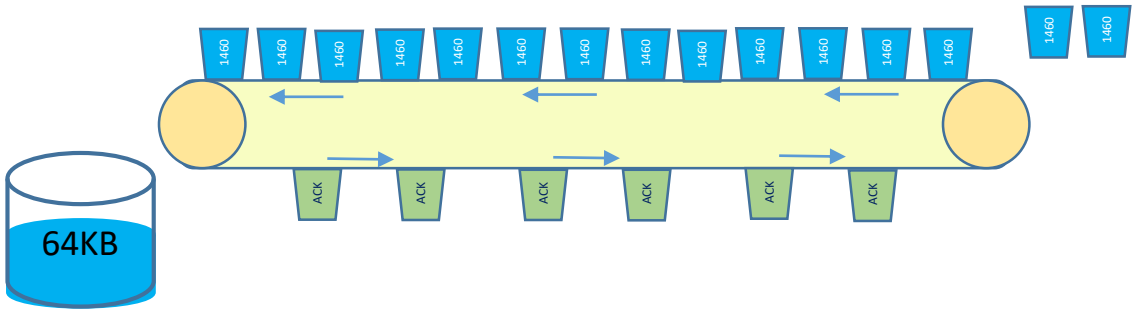
If the receiving application is "busy" or just can't keep up with the flow. It stops or slows taking data out of TCP's receive buffer.

If you run a packet capture on a PC or server, the data is captured here.
We'll see some "artefacts" of this later.



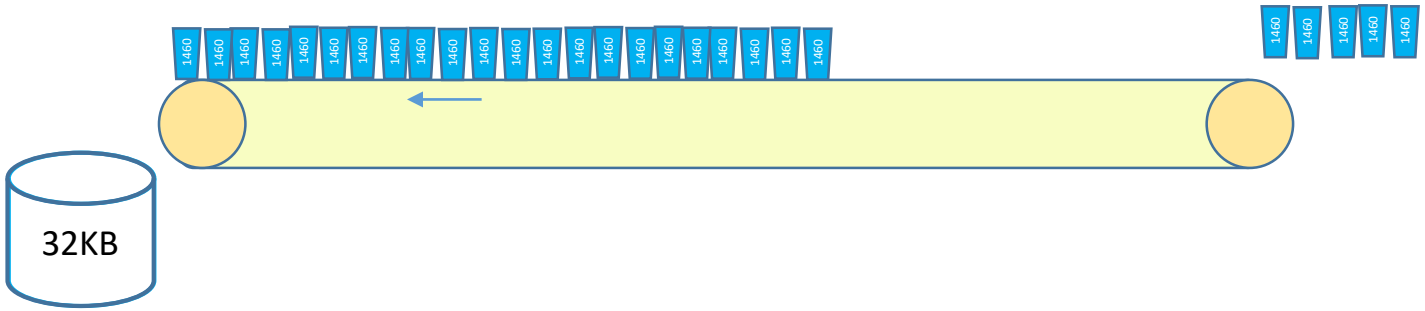
Optimum Throughput

Optimum data throughput is achieved when the sender can keep the conveyor full.



Here we have 13 buckets on the way (13 x 1460 = 18,980) but also 6 ACKs on the way back (which would represent 12 x 1460 = 17520) for a total of 25 buckets “in flight” (36,500 bytes).

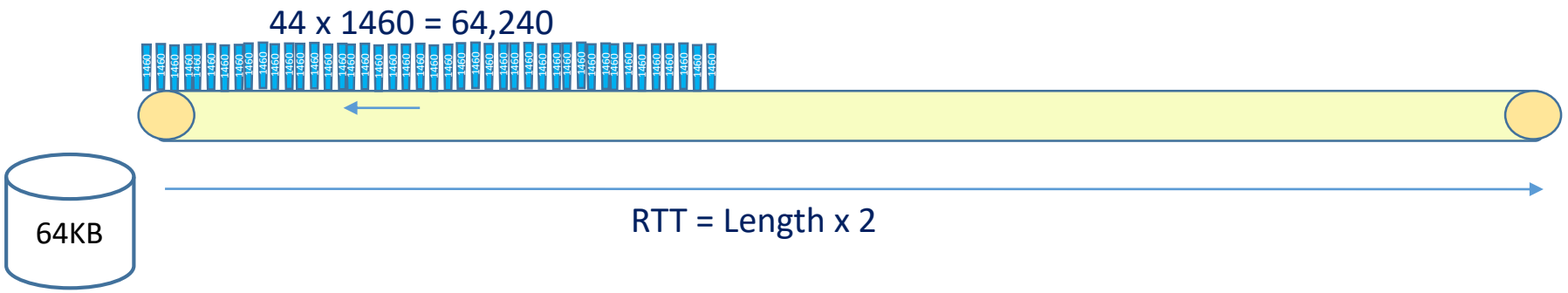
The receiver’s 64 KB “Receive Bucket” is more than enough to handle this flow, in fact, would only ever be half full.



The receiver’s 32 KB “Receive Bucket” and longer, faster belt mean that we have to stop after 22 buckets (22 x 1460 = 32,120).
The sender has to wait for the ACKs before adding more buckets to the belt.
The “Receive Bucket” will fill up, then empty, over and over.

64 KB in Milliseconds

Faster and longer networks mean that a 64 KB Receive Window is not enough these days.
Today we have “Long, Fat Networks (LFNs)”. Long = high RTT ; Fat = high bandwidth.



Time to Transmit 64 KB for Various Bandwidths*

Bandwidth		1 Bit (ms)	1 Byte (ms)	Packet Size	1 Packet (ms)	44 Packets (ms)
192	**Kb/s**	0.00521	0.04167	1460	60.833	**2676.667**
1	**Mb/s**	0.001	0.008	1460	11.68	**513.92**
10	**Mb/s**	0.0001	0.0008	1460	1.168	**51.392**
100	**Mb/s**	0.00001	0.00008	1460	0.1168	**5.1392**
500	**Mb/s**	0.000002	0.000016	1460	0.02336	**1.02784**
1	**Gb/s**	0.000001	0.000008	1460	0.01168	**0.51392**
10	**Gb/s**	0.0000001	0.000001	1460	0.001168	**0.051392**

*This ignores TCP/IP, Ethernet and other overheads.

Sample Round Trip Times

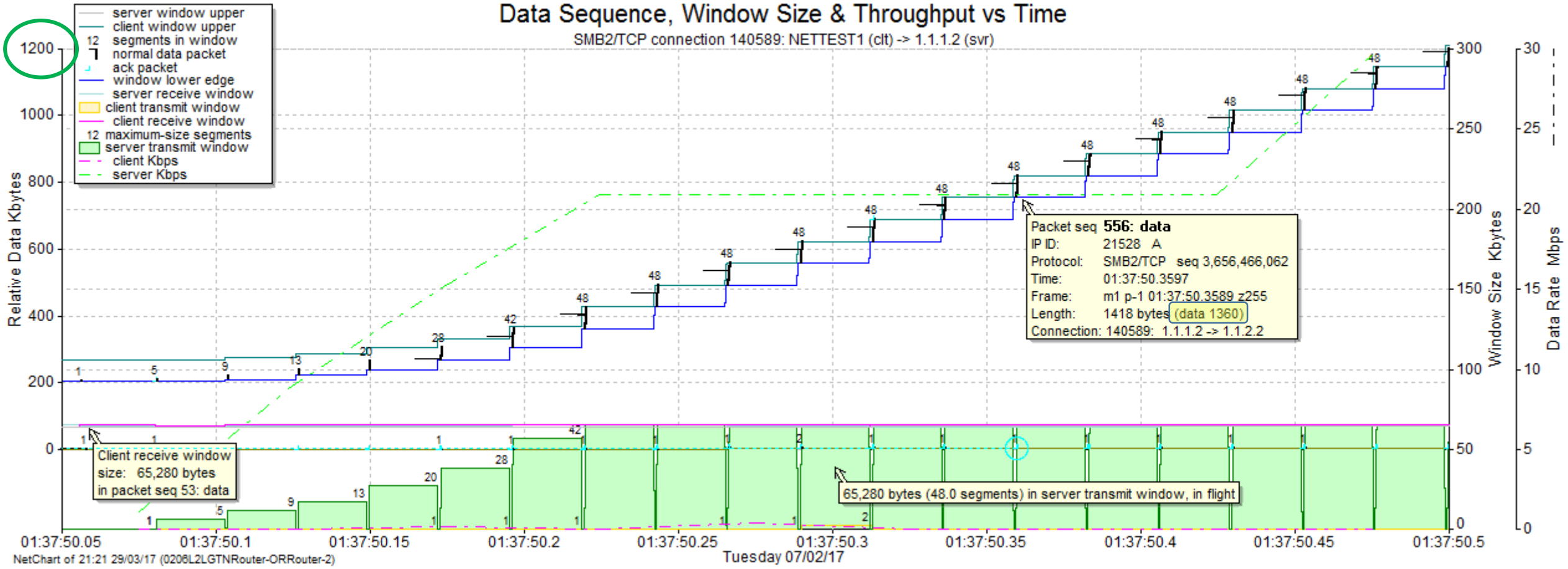
Effective Throughput

Source	Destination	RTT (ms)	Bytes per RT	Throughput (b/s)	AKA
Sydney	Sydney (DWDM)	**1**	64 KB	524,288,000	**500 Mb/s**
		2	64 KB	262,144,000	**250 Mb/s**
Sydney	Sydney (Cloud)	**5**	64 KB	104,857,600	**100 Mb/s**
		10	64 KB	52,428,800	**50 Mb/s**
Sydney	Melbourne	**15**	64 KB	34,952,533	**33 Mb/s**
		20	64 KB	26,214,400	**25 Mb/s**
Sydney	Perth	**50**	64 KB	10,485,760	**10 Mb/s**
		100	64 KB	5,242,880	**5 Mb/s**
Sydney	Los Angeles	**200**	64 KB	2,621,440	**2.5 Mb/s**
Sydney	New York	**250**	64 KB	2,097,152	**2 Mb/s**
		500	64 KB	1,048,576	**1 Mb/s**

64 KB Per RTT(=23ms) [0.5 secs = 1.2 MB]

Here is what a 64 KB Receive Window on a 1 GB/s network with RTT=23ms looks like. We get an effective throughput just over 20 Mb/s. There are 48 packets per RTT because the MSS=1360. 48 x 1360 = 65,280. In this ~half second time period we transfer 1.2 MB.

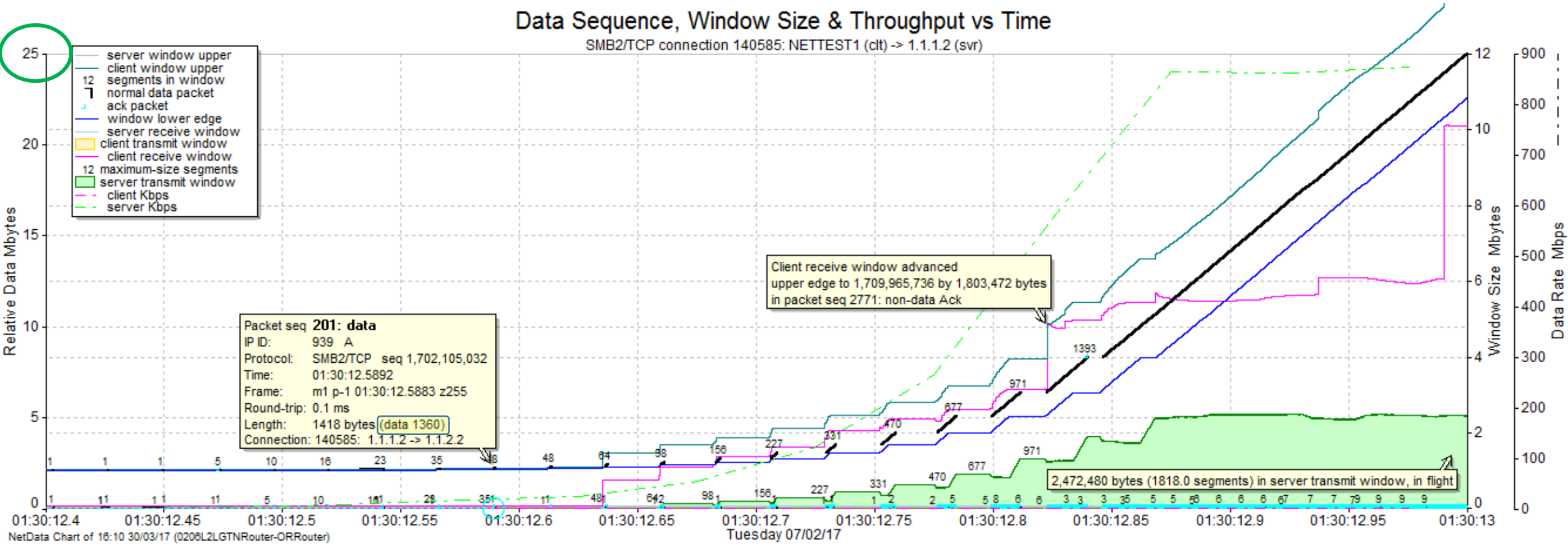
20 Mb/s is only 2.2% of the theoretical maximum throughput of ~900 Mb/s (allowing for overheads in the 1 Gb/s link).



2.5 MB Per RTT(=23ms) [0.5 secs = 25 MB]

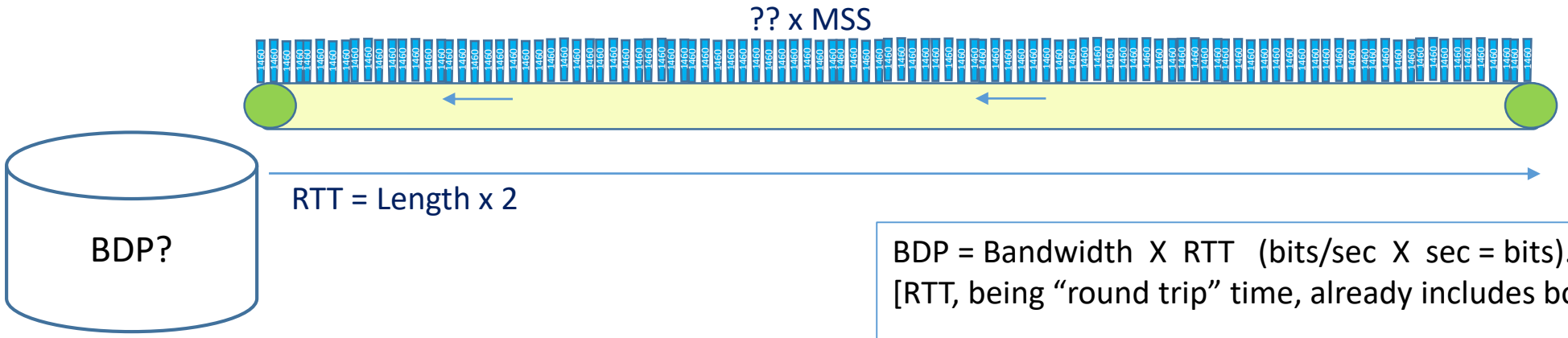
Here is what a 2.5 MB Transmit Window on a 1 GB/s network with RTT=23ms looks like.
We get an effective throughput of just under 900 Mb/s. We manage to transfer 25 MB in just under half a second!
There are 1800+ packets per RTT (2.5 MB).

This is 100% of the theoretical maximum throughput of a 1 Gb/s link.



Bandwidth Delay Product (BDP)

The question about throughput can be phrased differently: "Given a particular bandwidth and RTT,"
- "How big does the Receive Window need to be in order to fill up the path?" or
- "What Receive Window will enable the maximum (i.e., constant) throughput?"



BDP = Bandwidth X RTT (bits/sec X sec = bits).
[RTT, being "round trip" time, already includes both directions].

10**M**b/s X 10**m**s =
10Kb/s X 10 sec = 100 Kb (= **12.5 KB**).

BDP (**Bytes**) for Various Bandwidth/RTT Combinations

	RTT (ms)										
Bandwidth	**1**	**2**	**5**	**10**	**15**	**20**	**50**	**100**	**200**	**250**	**500**
192 Kb/s	24	48	120	240	360	480	1,200	2,400	4,800	6,000	12,000
1 Mb/s	125	250	625	1,250	1,875	2,500	6,250	12,500	25,000	31,250	62,500
10 Mb/s	1,250	2,500	6,250	12,500	18,750	25,000	62,500	125,000	250,000	312,500	625,000
100 Mb/s	12,500	25,000	62,500	125,000	187,500	250,000	625,000	1,250,000	2,500,000	3,125,000	6,250,000
500 Mb/s	62,500	125,000	312,500	625,000	937,500	1,250,000	3,125,000	6,250,000	12,500,000	15,625,000	31,250,000
1 Gb/s	125,000	250,000	625,000	1,250,000	1,875,000	2,500,000	6,250,000	12,500,000	25,000,000	31,250,000	62,500,000
10 Gb/s	1,250,000	2,500,000	6,250,000	12,500,000	18,750,000	25,000,000	62,500,000	125,000,000	250,000,000	312,500,000	625,000,000

Bandwidth Delay Product (BDP)

Bandwidth-Delay Product refers to the product of a data link's capacity (in bits per second) and its round-trip delay time (in seconds). The result is an amount of data, measured in bits (or bytes), that is equivalent to the maximum amount of data on the network circuit at any given time, i.e., “in flight” data that has been transmitted but not yet acknowledged.

A network with a large bandwidth-delay product is commonly known as a **long fat network** (shortened to **LFN** and often pronounced "elephen"). As defined in RFC 1072, a network is considered an LFN if its bandwidth-delay product is significantly larger than 10^5 bits (12500 bytes).

Ultra-high speed LANs may fall into this category, where protocol tuning is critical for achieving peak throughput, on account of their extremely high bandwidth, even though their delay is not great.

An important example of a system where the bandwidth-delay product is large is that of GEO satellite connections, where end-to-end delivery time is very high and link throughput may also be high. The high end-to-end delivery time makes life difficult for stop-and-wait protocols and applications that assume rapid end-to-end response.

Sourced from:
https://en.wikipedia.org/wiki/Bandwidth-delay_product

TCP/IP Protocol History

Modern version(s) of TCP/IP have features that improve performance over the original “old” TCP/IP

These new features became necessary as bandwidths increased and exposed the limitations of the original protocol designs. (64 KB Receive Window and “normal” Acknowledgements).

The most important update in this area was:

RFC 1072	**October 1988**	https://tools.ietf.org/html/rfc1072
RFC 1323	May 1992	https://tools.ietf.org/html/rfc1323
RFC 7323	September 2014	https://tools.ietf.org/html/rfc7323

The two most important of the “new” features are:

- Window Scaling
- Selective ACKs (SACKs)

These features weren’t commonly implemented until well into the 2000’s! [Linux (2.6.8) - Aug 2004 ; Windows Vista - 2008]

Bear in mind that the “new” features had to work with “old” TCP implementations that knew nothing about them.

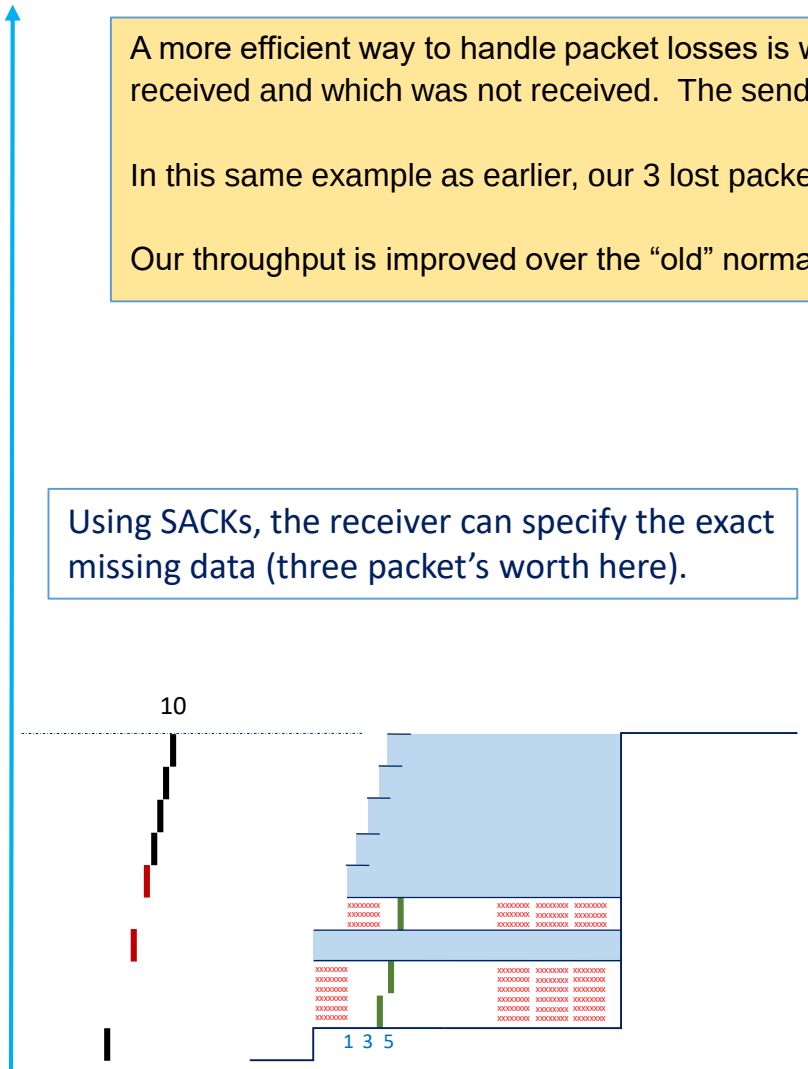
Selective Acknowledgement (SACK)

A more efficient way to handle packet losses is with the “newer” Selective Acknowledgement method. SACKs tell the sender exactly which data was received and which was not received. The sender can respond by retransmitting the exact missing data all in one go.

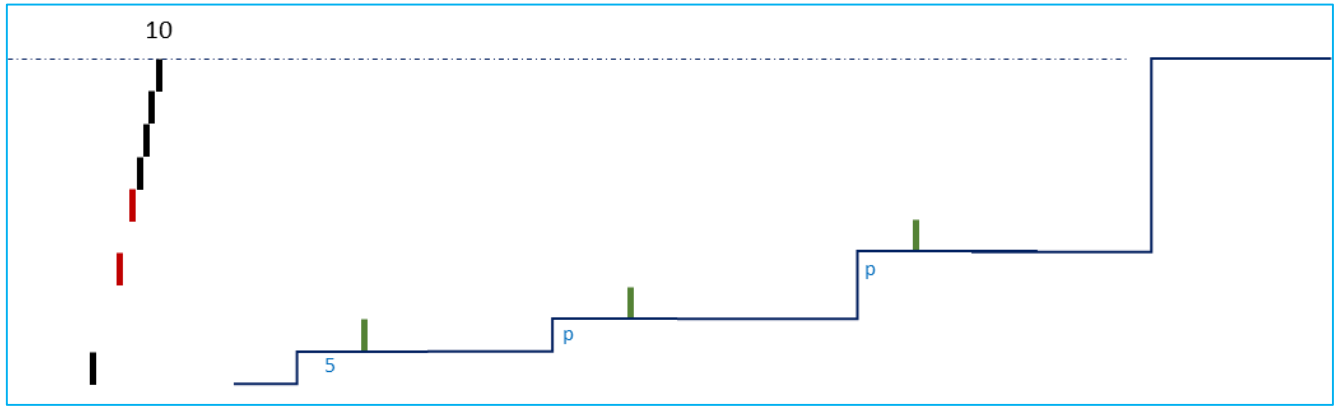
In this same example as earlier, our 3 lost packets now only cost us one extra round trip for the retransmissions.

Our throughput is improved over the “old” normal ACKs. Note that SACKs also count as Dup-ACKs.

Using SACKs, the receiver can specify the exact missing data (three packet’s worth here).



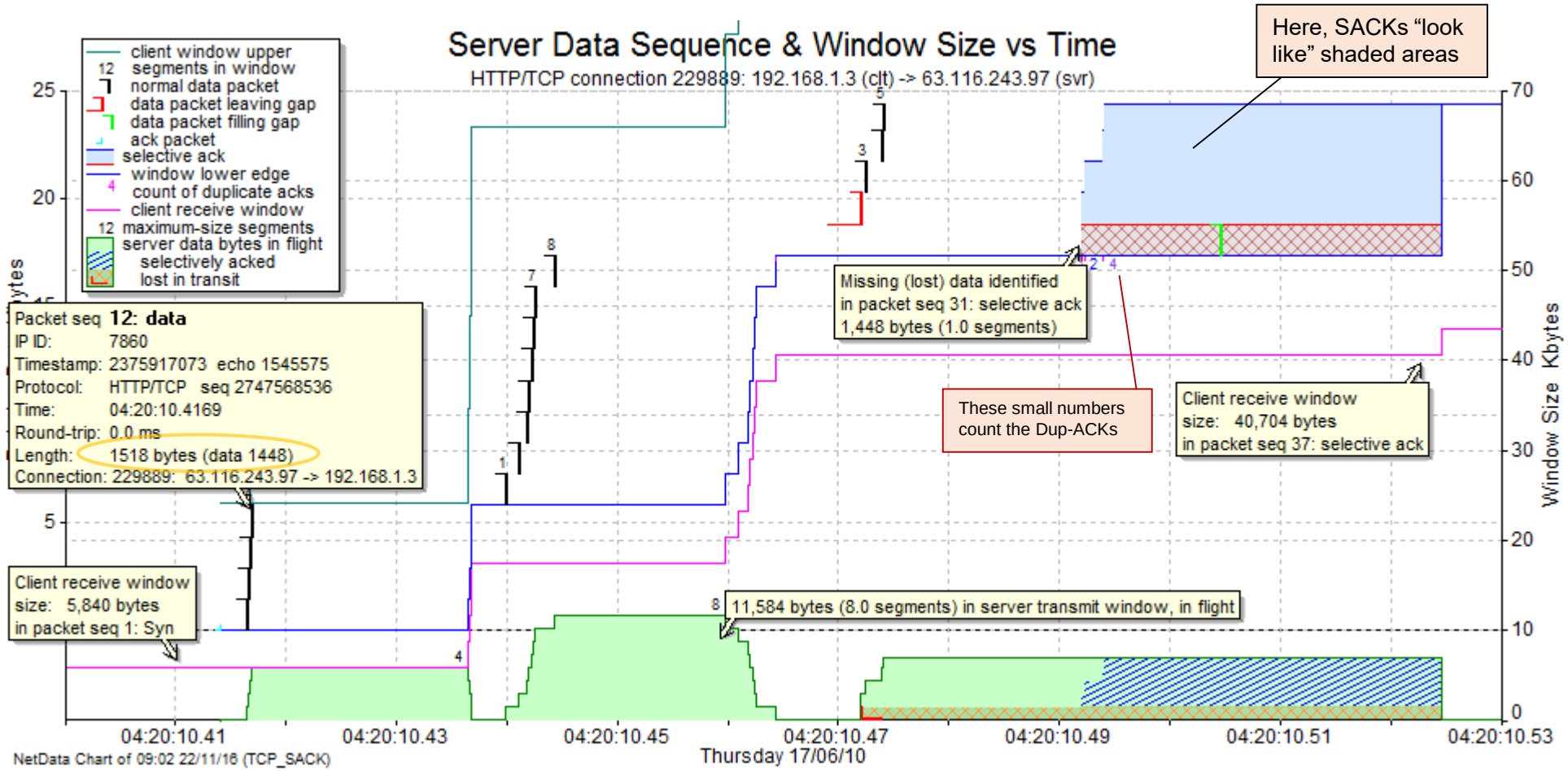
Without SACKs – 3 RTTs



Selective Acknowledgement (NetData)

Observe that our payload sizes here are 1448 instead of the usual 1460 because TCP Timestamps are enabled.
The left y-axis measures bytes transferred (TCP Sequence Numbers) and the x-axis is time of day. This means that normal flows will work up from the bottom left to the top right of the chart. Chart items related to the various TCP windows use the right y-axis.

Shapes, colours and position are used to present different packet types and sizes. TCP acknowledgements are displayed as window edge lines that will "step up" at the time the ACKs are estimated to be received by the sender.



TCP/IP Protocol Headers

TCP Header

Offsets	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Source port																Destination port															
4	32	Sequence number																															
8	64	Acknowledgment number (if ACK set)																															
12	96	Data offset				Reserved 0 0 0				N S	C W R	E C E	U R G	A C K	P S H	R S T	S Y N	F I N	Window Size														
16	128	Checksum																Urgent pointer (if URG set)															
20 ...	160 ...	Options (if *data offset* > 5. Padded at the end with "0" bytes if necessary.) ...																															

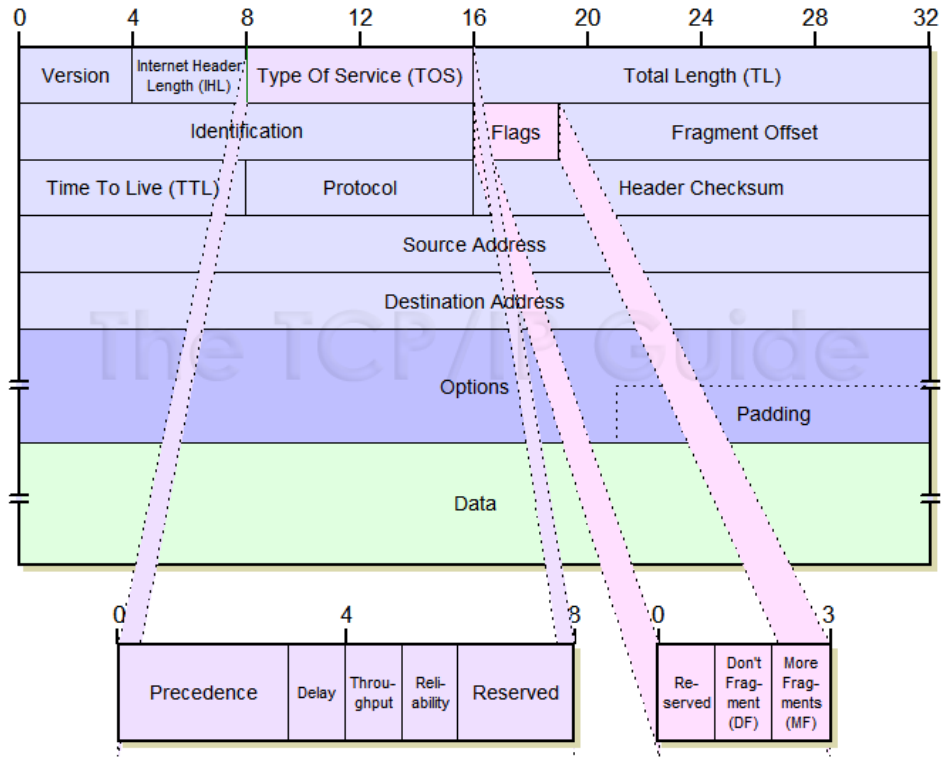


Figure 86: Internet Protocol Version 4 (IPv4) Datagram Format

This diagram shows graphically the all-important IPv4 datagram format. The first 20 bytes are the fixed IP header, followed by an optional *Options* section, and a variable-length *Data* area. Note that the *Type Of Service* field is shown as originally defined in the IPv4 standard.

Main Information:

- Smallest possible TCP+IP header size is 20+20=40 bytes.
- TCP can be larger (but then TCP payload must be reduced).
- Sequence numbers and ACK numbers are 32-bits each.
- **Window size is 16-bits (0-65535 = max 64 KB).**

Sourced from:
https://en.wikipedia.org/wiki/Transmission_Control_Protocol
http://www.tcpipguide.com/free/t_IPDatagramGeneralFormat.htm

TCP Optional Header Fields

Options (Variable 0–320 bits, divisible by 32)

The length of this field is determined by the data offset field. Options have up to three fields: Option-Kind (1 byte), Option-Length (1 byte), Option-Data (variable). The Option-Kind field indicates the type of option, and is the only field that is not optional. Depending on what kind of option we are dealing with, the next two fields may be set: the Option-Length field indicates the total length of the option, and the Option-Data field contains the value of the option, if applicable. For example, an Option-Kind byte of 0x01 indicates that this is a No-Op option used only for padding, and does not have an Option-Length or Option-Data byte following it. An Option-Kind byte of 0 is the End Of Options option, and is also only one byte. An Option-Kind byte of 0x02 indicates that this is the Maximum Segment Size option, and will be followed by a byte specifying the length of the MSS field (should be 0x04). This length is the total length of the given options field, including Option-Kind and Option-Length bytes. So while the MSS value is typically expressed in two bytes, the length of the field will be 4 bytes (+2 bytes of kind and length). In short, an MSS option field with a value of 0x05B4 will show up as (0x02 0x04 0x05B4) in the TCP options section.

Some options may only be sent when `SYN` is set; they are indicated below as [SYN]. Option-Kind and standard lengths given as (Option-Kind,Option-Length).

- 0 (8 bits): End of options list
- 1 (8 bits): No operation (NOP, Padding) This may be used to align option fields on 32-bit boundaries for better performance.
- 2,4,*SS* (32 bits): Maximum segment size (*see maximum segment size*) [SYN]
- 3,3,*S* (24 bits): Window scale (*see window scaling for details*) [SYN][6]
- 4,2 (16 bits): Selective Acknowledgement permitted. [SYN] (*See selective acknowledgments for details*)[7]
- 5,*N*,*BBBB*,*EEEE*,... (variable bits, *N* is either 10, 18, 26, or 34)- Selective ACKnowledgement (SACK)[8] These first two bytes are followed by a list of 1–4 blocks being selectively acknowledged, specified as 32-bit begin/end pointers.
- 8,10,*TTTT*,*EEEE* (80 bits)- Timestamp and echo of previous timestamp (*see TCP timestamps for details*)[9]

(The remaining options are historical, obsolete, experimental, not yet standardized, or unassigned)

Main Information:

- Can increase header size by up to 40 bytes.
- **Some options used only in the SYN/SYN-ACK Sequence.**
- 32-bit MSS implies 4,294,967,296 but practice = **1460**
- Window Scale says 24-bits (but **max scale = 14**).

Sourced from:
https://en.wikipedia.org/wiki/Transmission_Control_Protocol

TCP 3-Way Handshake

Main Information:
- The initiator is the “client”.
- The timings in your capture can tell you where the capture was taken.
- Client and server negotiate or make offers for supported features.
- Intermediate devices (e.g., routers, firewalls) can modify parameters as they pass through.

Main “Agreements”:
- Maximum Segment Size (MSS).
- Window Scaling supported? If so, Max Scale?
- Selective ACK supported?

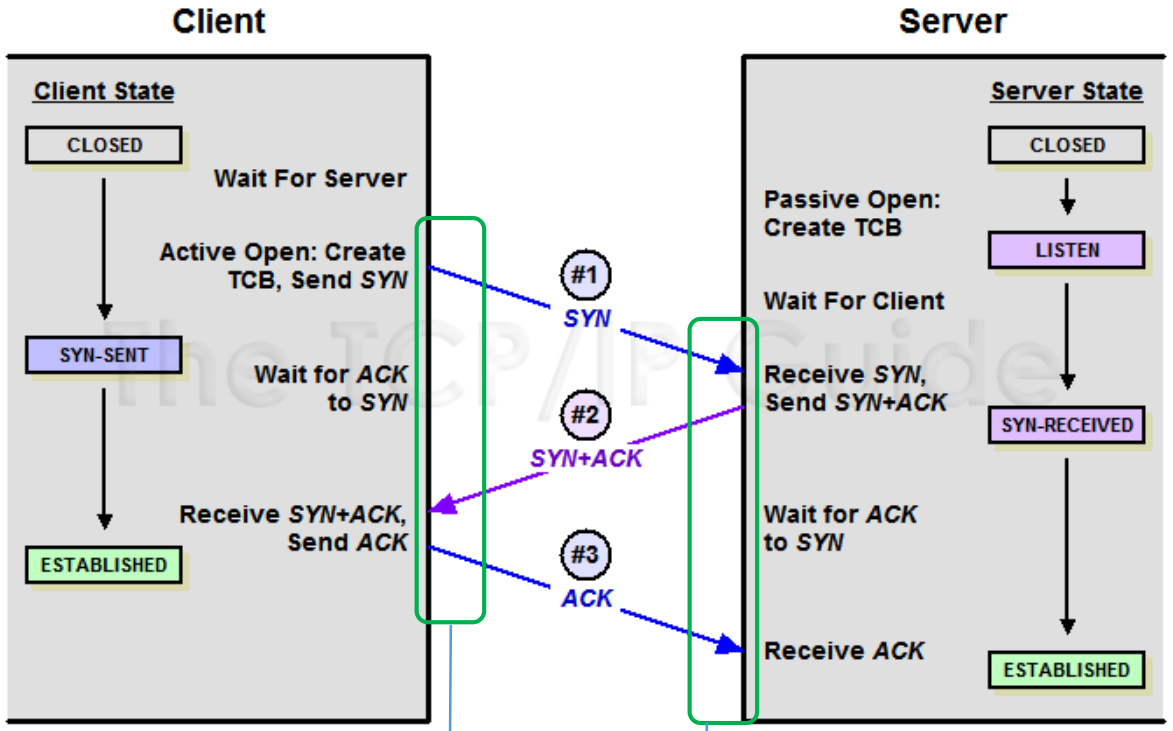


Figure 211: TCP “Three-Way Handshake” Connection Establishment Procedure

This diagram illustrates how a conventional connection is established between a client and server, showing the three messages sent during the process and how each device transitions from the *CLOSED* state through intermediate states until the session is *ESTABLISHED*.

Capture at client timings: SYN, 🕔, SYN-ACK, ACK

Capture at server timings: SYN, SYN-ACK, 🕔, ACK

Sourced from:
http://www.tcpipguide.com/free/t_TCPConnectionEstablishmentProcessTheThreeWayHandsh-3.htm

Window Scaling

TCP Window Scale option is needed for efficient transfer of data when the bandwidth-delay product is greater than 64 KB.

The Window Scaling option needed to be compatible with “old” TCP/IP implementations – so had to keep the 16-bit “Window Size” header field. An extended header “Window Scale” field is used.

By using the window scale option, the Receive Window size can be increased up to a maximum value of 1,073,725,440 bytes (1 GB). This maximum occurs when the scale factor is 14.

65536 * 2^14 = 1,073,725,440

The scale is agreed in the TCP setup and never seen again in any packets. Each side must remember the other side’s value. If you don’t capture the 3-way handshake, then you have to guess the scale values.

Due to the “Multiplier”, variations in the Receive Window values can only be made in “multiplier” units.

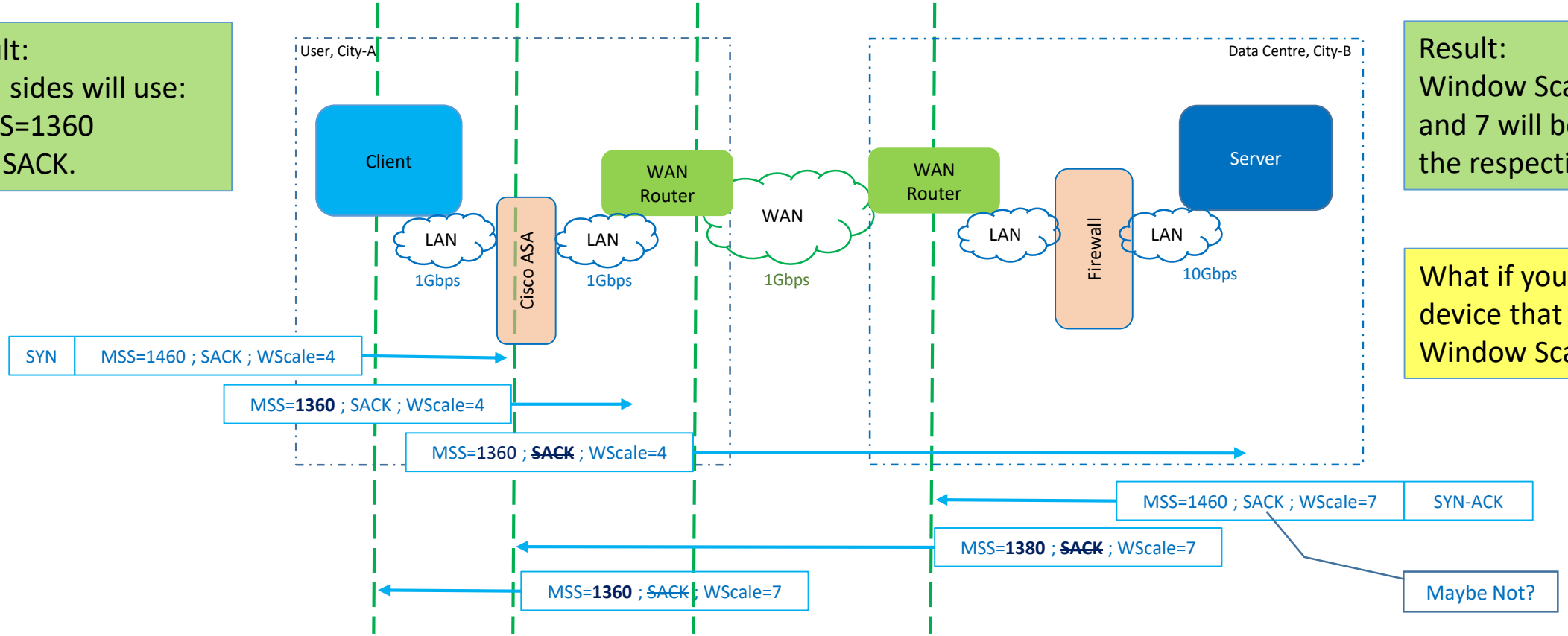
The RFCs dictate that a field value of 15 must be interpreted as 14.

Scale Factor	Multiplier	x 65536	AKA
0	1	65536	64 KB
1	2	131072	128 KB
2	4	262144	256 KB
3	8	524288	512 KB
4	16	1048576	1 MB
5	32	2097152	2 MB
6	64	4194304	4 MB
7	128	8388608	8 MB
8	256	16777216	16 MB
9	512	33554432	32 MB
10	1024	67108864	64 MB
11	2048	134217728	128 MB
12	4096	268435456	256 MB
13	8192	536870912	512 MB
14	16384	1073741824	1 GB

Intermediate Devices Can Modify SYN / SYN-ACK

In this "contrived" example:
- The client supports MSS=1460 ; SACK and Window Scaling with scale 4 (x 16)
- The Cisco ASA FW reduces the MSS to 1360 but supports the other options
- The WAN Router doesn't support SACK so turns it off. The MSS is already smaller than the MSS=1380 that it supports.
- The server supports MS=1460 ; SACK and Window Scaling with scale 7 (x 128)
- The WAN Router doesn't support SACK so turns it off. It also reduces the MSS to 1380
- The Cisco ASA FW further reduces the MSS to 1360 but leaves the other options unchanged

Result:
Both sides will use:
- MSS=1360
- No SACK.



Result:
Window Scaling of 4 and 7 will be used at the respective sides.

What if you have a device that turns off Window Scaling?

Common MSS Values

This table lists some commonly observed TCP/IP packet payload values.

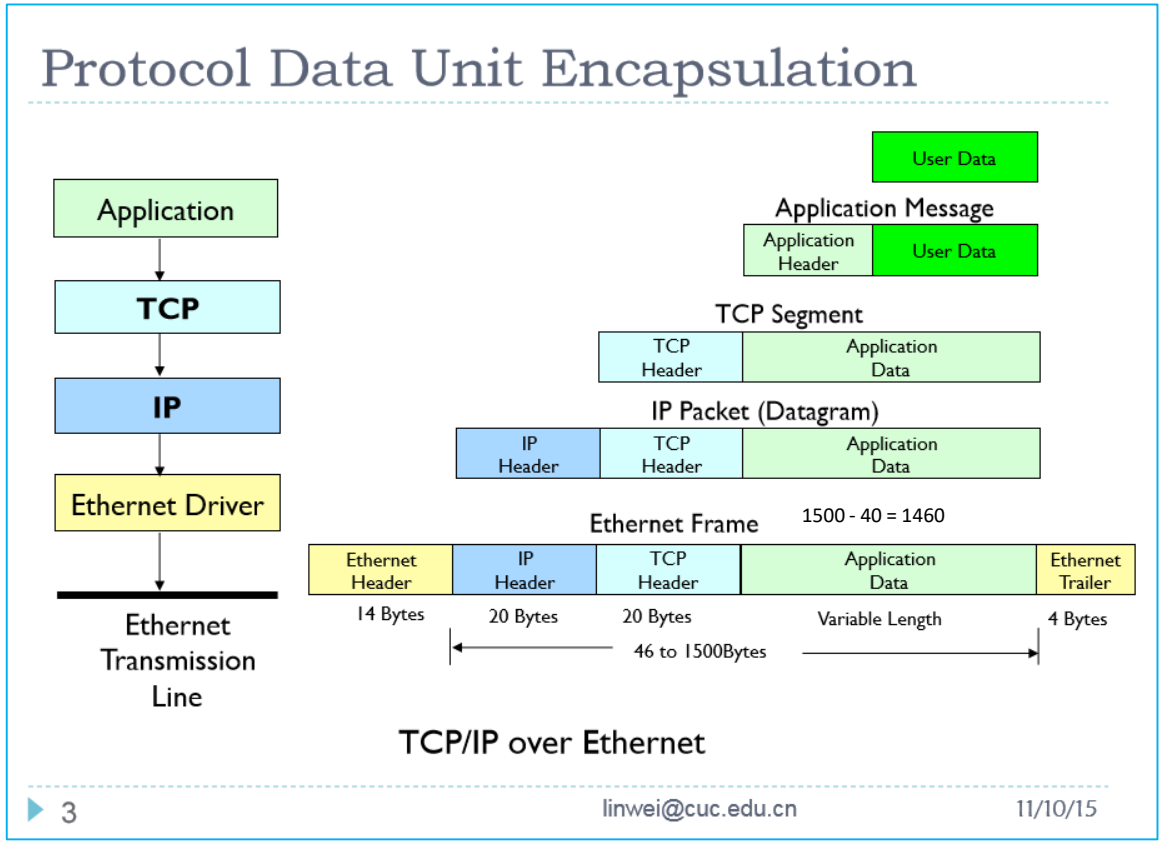
MSS	Comment
1460	Maximum possible on Ethernet (max Ethernet payload is 1500, minus 40 byte TCP/IP header)
1448	Maximum if TCP Timestamps are enabled (which use "extended" TCP Header fields)
1440	Maximum with Ethernet and IPv6 (due to larger IP header).
1380	Cisco ASA Firewalls commonly modify MSS to this as the SYN / SYN-ACKs pass through.
1360	Cisco ASA Firewalls commonly modify MSS to this as the SYN / SYN-ACKs pass through.
536	"Default" MSS. This must be used if not agreed in 3-way handshake. Can be carried on all network types without needing fragmentation. Used to be very common in Windows systems. Some systems will automatically drop packets to this size if they get no response to full-sized "black holed" packets.

RFC 879
https://tools.ietf.org/html/rfc879

"Long established rule".

Hosts must not send datagrams larger than 576 octets (-40 = 536) unless they have specific knowledge that the destination host is prepared to accept larger datagrams.

The diagram shows why 1460 is the maximum possible MSS.



Sourced from:
http://icourse.cuc.edu.cn/networkprogramming/lectures/Unit6_TCP.pdf

TCP/IP Protocol Terminology

Where are we up to?

“Acknowledgement (ACK)” “Duplicate ACK” “Delayed ACK” “Partial ACK”
“Fast Retransmission” “Retransmission Timeout (RTO)”
“Round Trip Time (RTT)” “Bytes In Flight” “Bandwidth Delay Product (BDP)”
“Receive Window” “Transmit Window” “Window Scaling” “Long, Fat Network (LFN)”
“Slow Start” mode “Congestion Avoidance” mode

“3-Way Handshake” “Maximum Segment Size (MSS)”
“Selective Acknowledgement (SACK)” “Duplicate SACK”

“Out of Order (OOO)”

“Window Closure” aka “Zero Window”

Real World Performance Problem

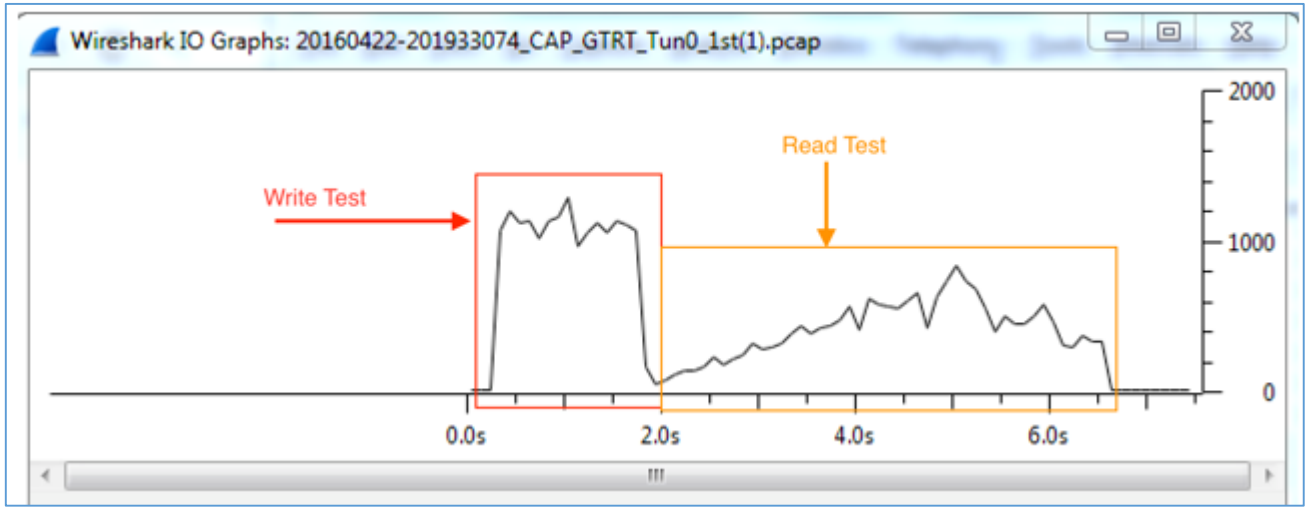
A question appeared on the site “Ask Wireshark”.

https://ask.wireshark.org/questions/55972/slow-writes-even-slower-reads-spanning-wan-to-netapp

The problem had been occurring for a long time, happened the same way every time and nobody knew what was the cause.

The symptom was that an SMB file transfer of the exact same file, between the same user PC and the same NetApp file share server, always took longer to download than to upload.

The Wireshark throughput charts showed that the throughput of the download (SMB Read) versus the upload (SMB Write) varied significantly and formed “sawtooth” patterns.

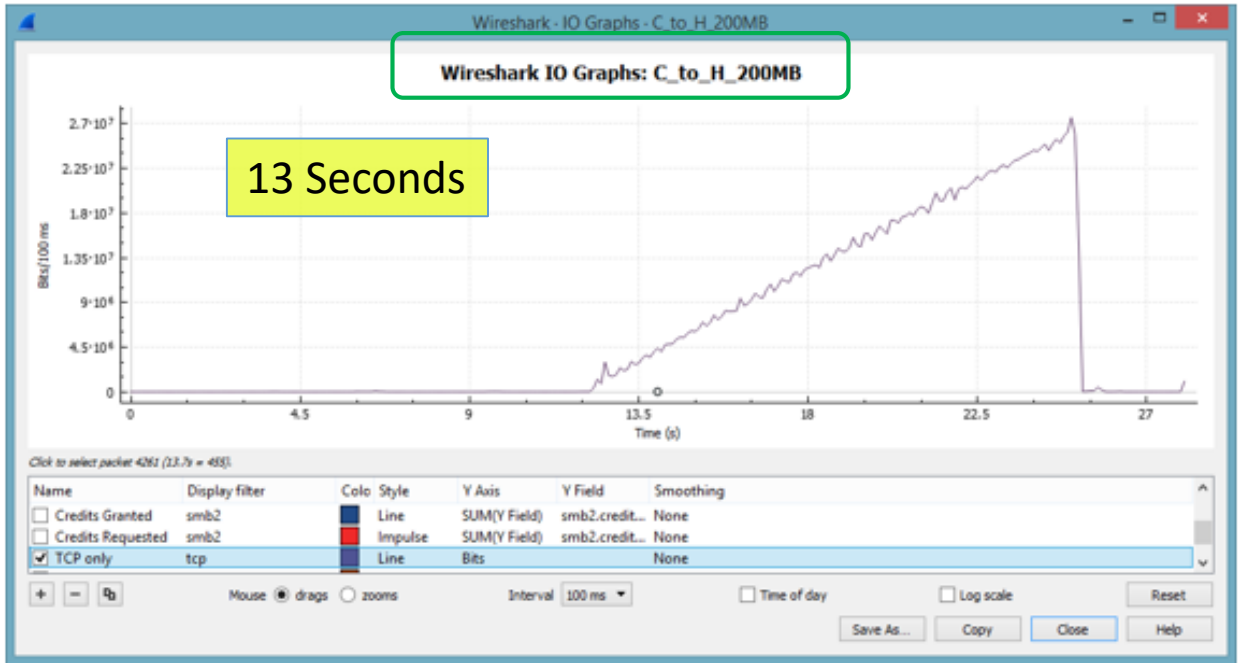


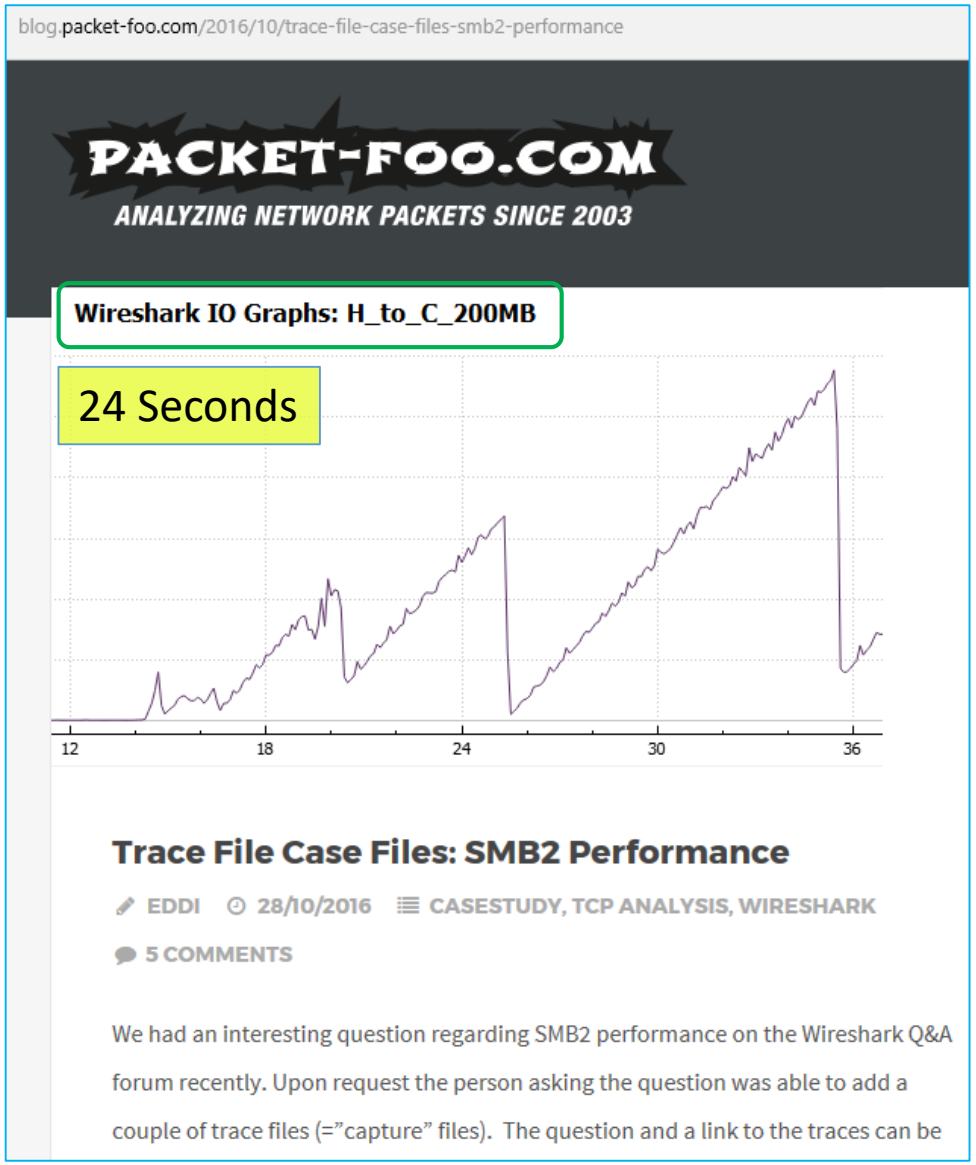
Wireshark Experts Also Looked At It

Wireshark experts in Europe blogged about it.

https://blog.packet-foo.com/2016/10/trace-file-case-files-smb2-performance

Note the multiple sawtooth pattern in the “H-to-C” case (SMB Read) with the 200 MB file taking **24 seconds**. The “C-to-H” case had only one big sawtooth and took just **13 seconds** to transfer the same 200 MB file.





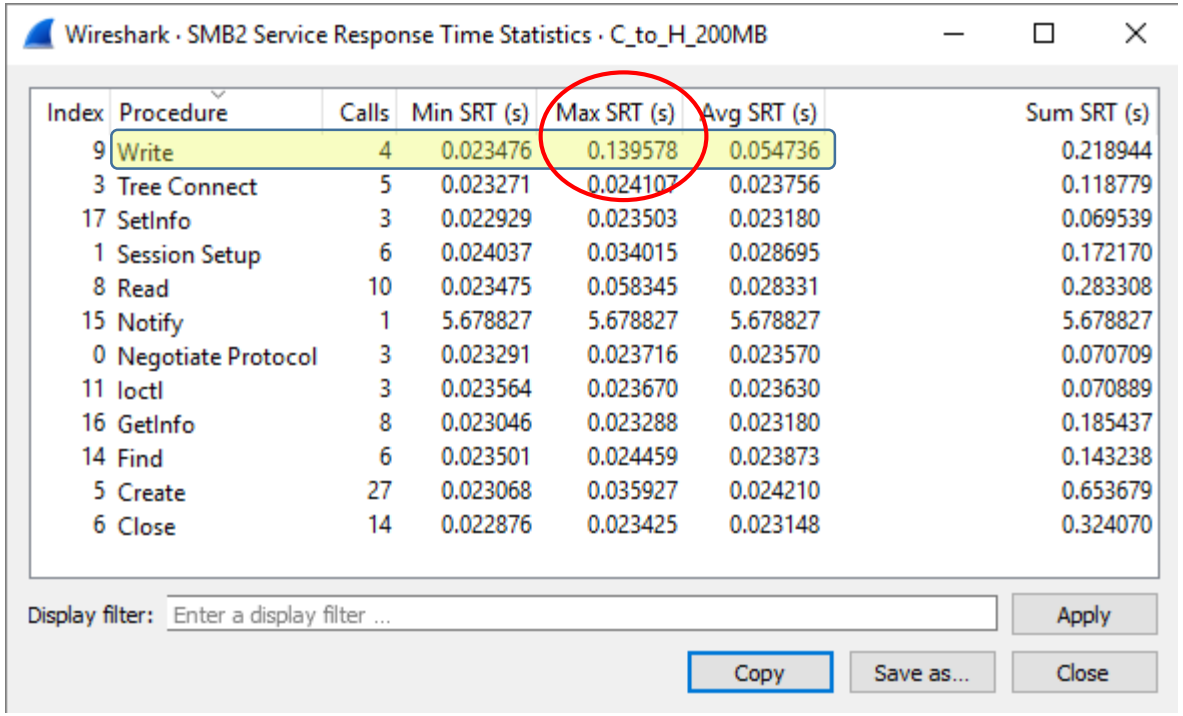
Wireshark “Server Response Time (SRT)” Values

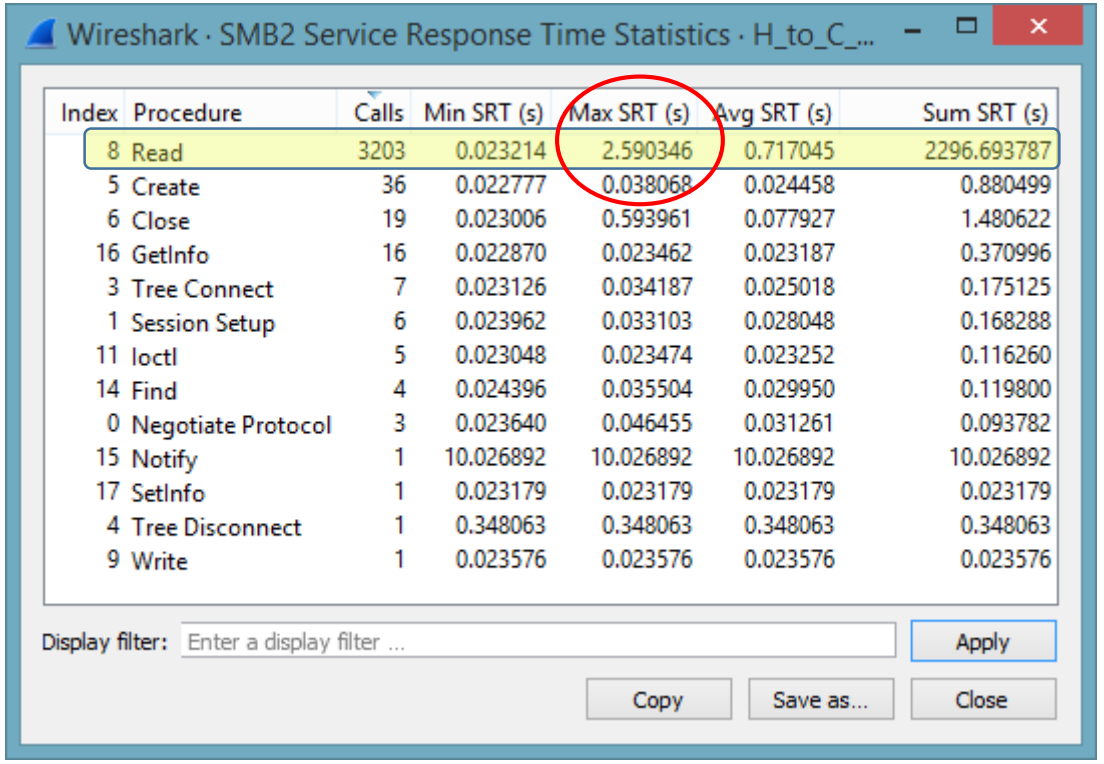
Wireshark can calculate “server response time” values for transactions using some well known protocols.

The SRT for an individual transaction is measured as the time between the first client request packet to the first packet of the server’s response. It attempts to measure the time that the server had to think about the answer before it began delivering the response.

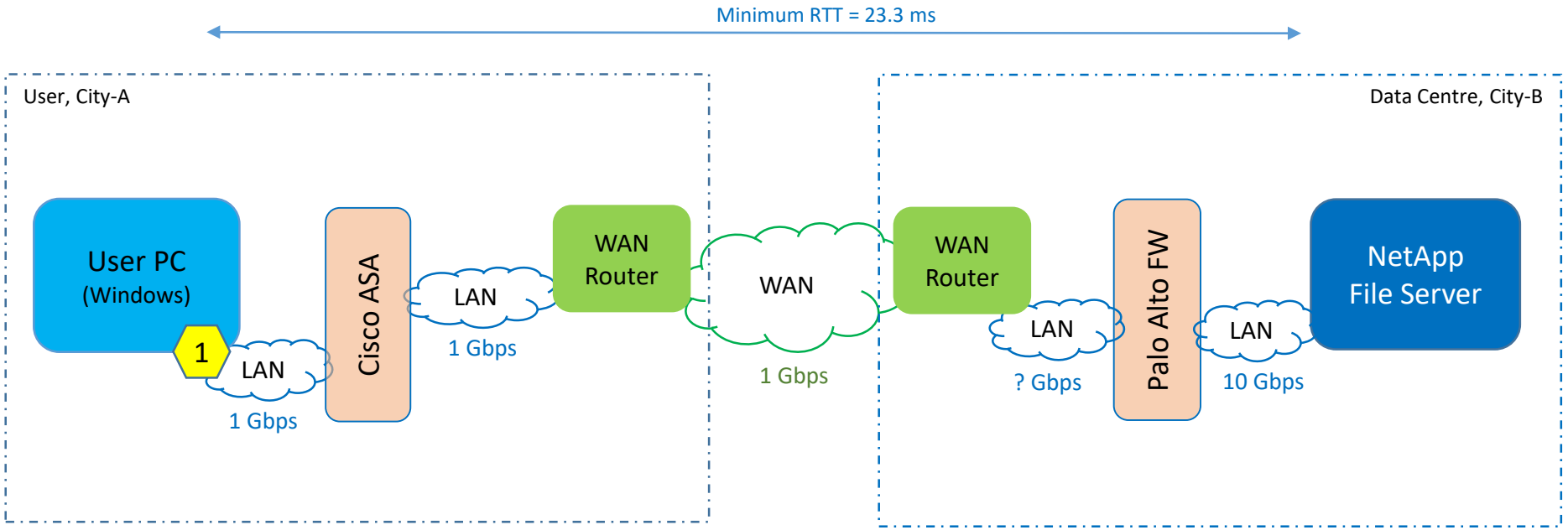
For captures taken at the client end, these times also include the network RTT, because the times are measured as at the client.
That’s why the minimums here are 23 ms.

Why would the NetApp spend more time to process Read requests?





PC-to-NetApp Network



Live Analysis of “C_to_H” & “H_to_C”

Will now examine the actual “pcap” files that were made available in the Ask Wireshark question.

First the “C to H” capture, where the client PC sends the 200 MB file up to the NetApp filer shared drive.

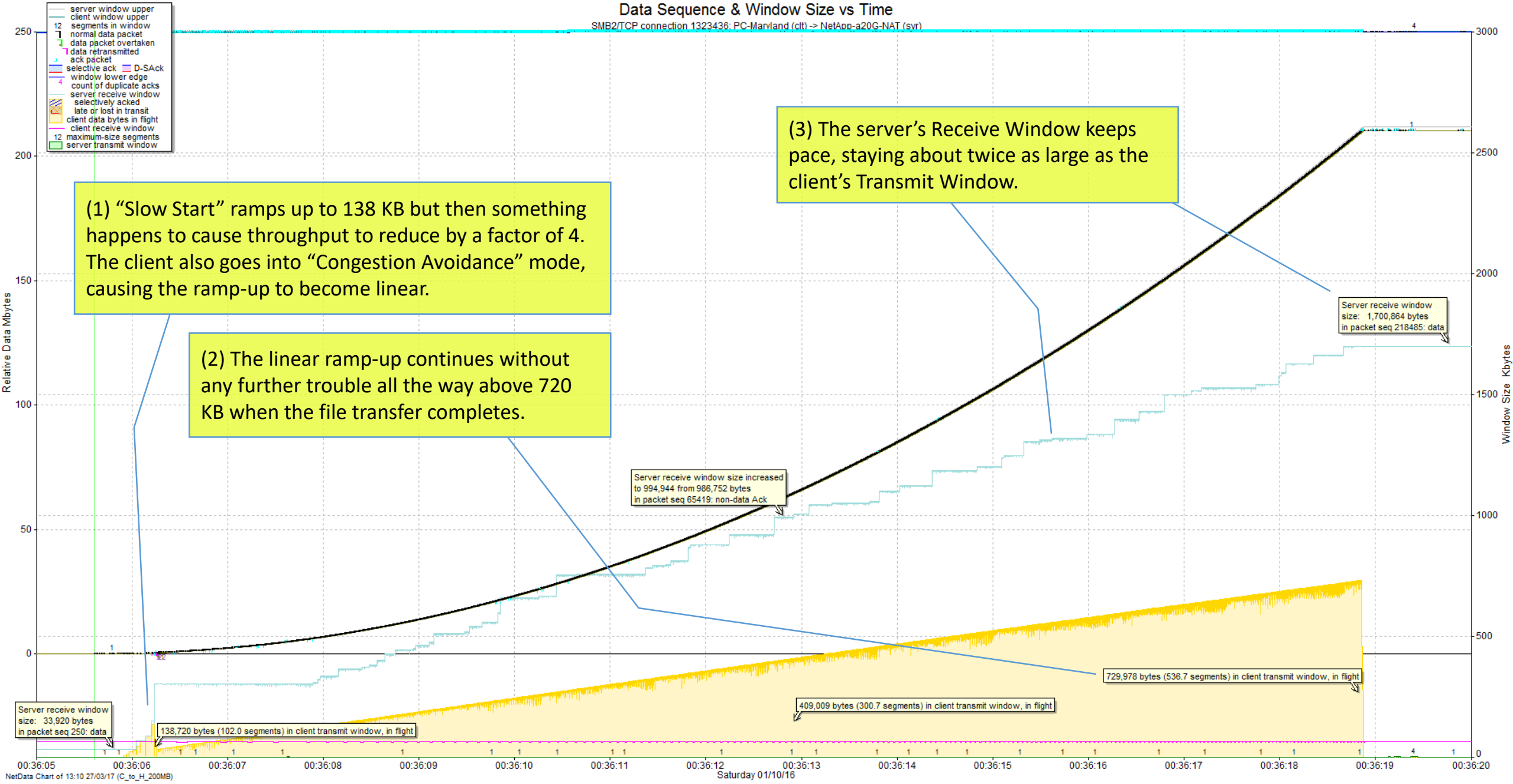
Then the “H to C” sample, where the client retrieves the same 200 MB file from the NetApp, copying it back to the local PC.

- Plot the Flow Charts for each file transfer and examine where things went wrong.

- Plot the file transfer throughputs (we know that the WAN bandwidth is 1 Gb/s and the RTT is 23 ms). [BDP = 2.5 MB]

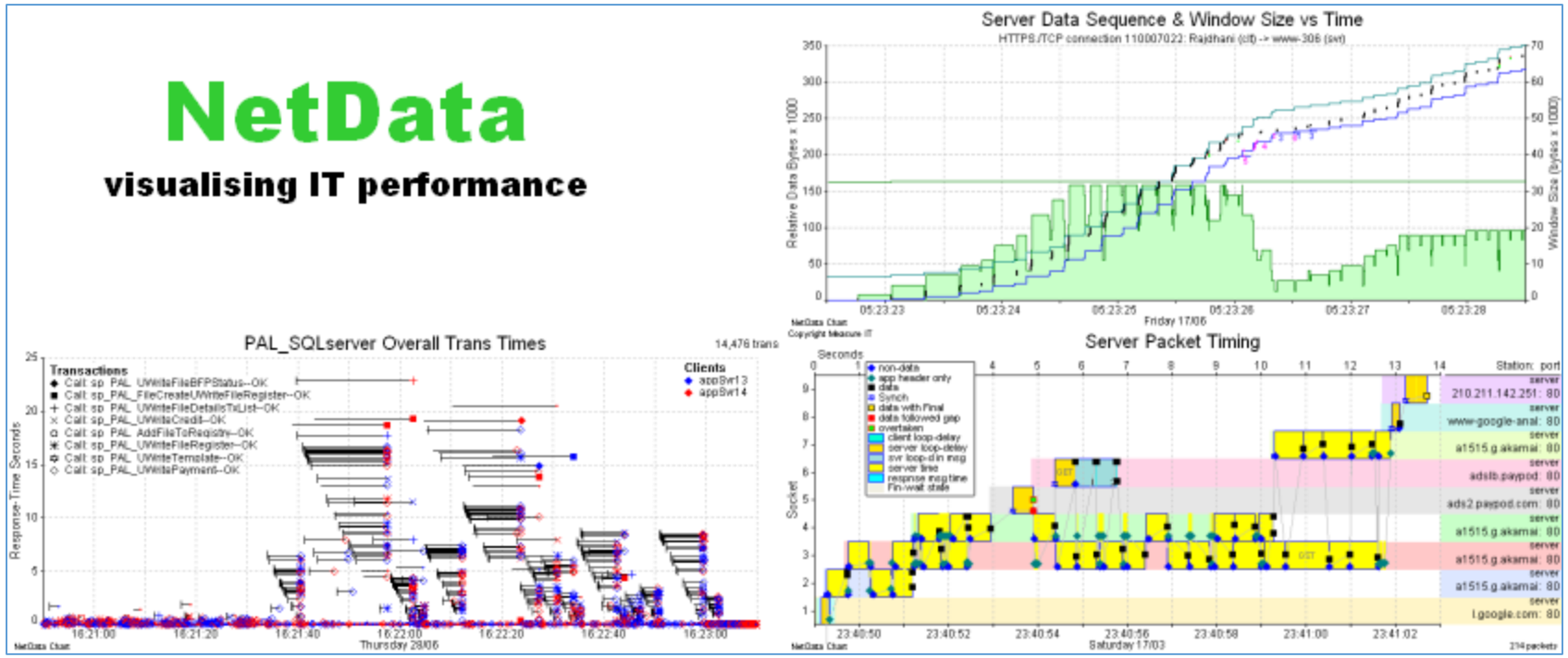
- Look at the individual SMB2 transactions.
- Look at the “Transactions in Progress”.
- Compare the SMB2 protocol behaviours.

C to H Full Flow (13 Seconds)

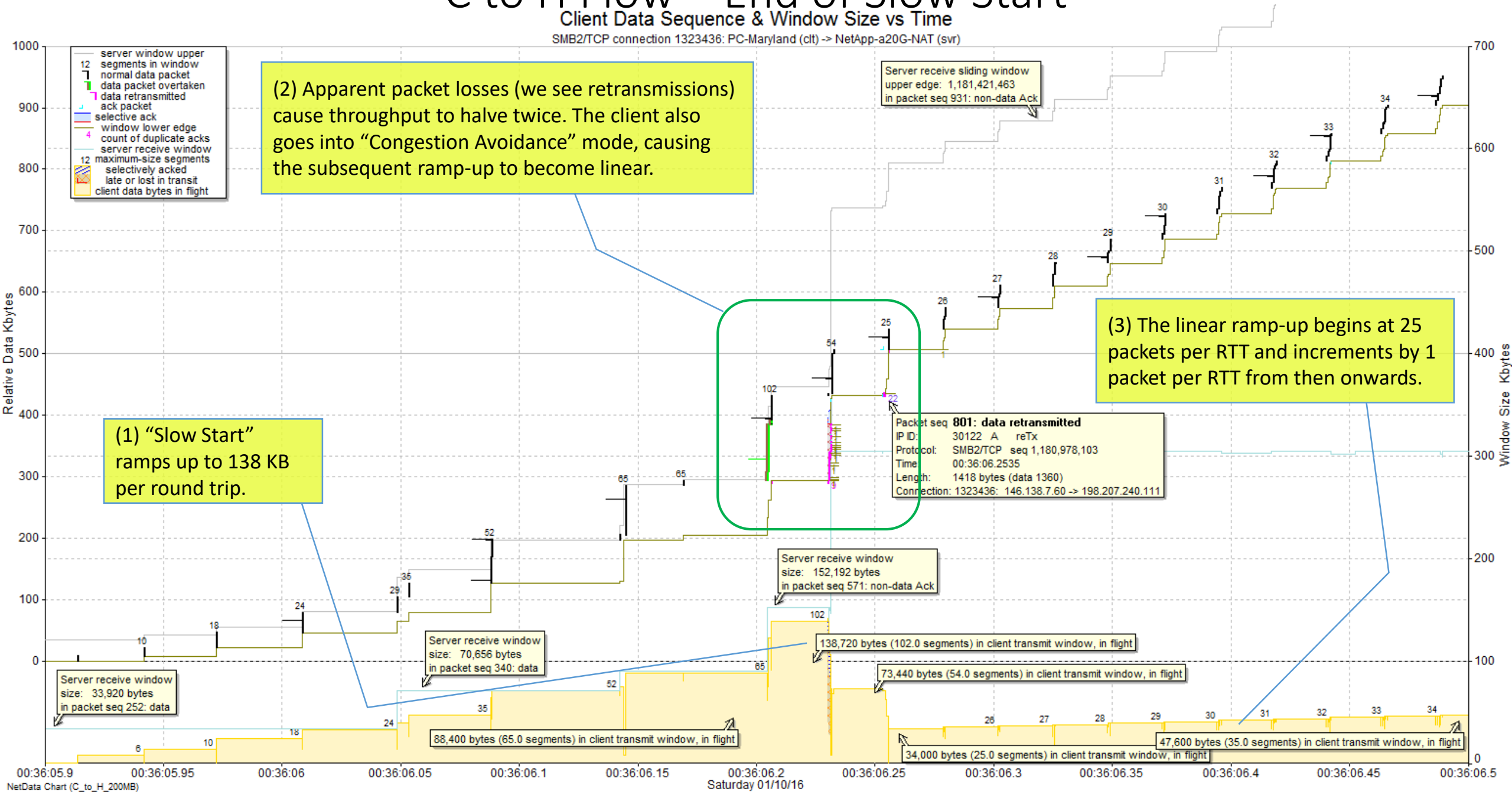


Live Analysis

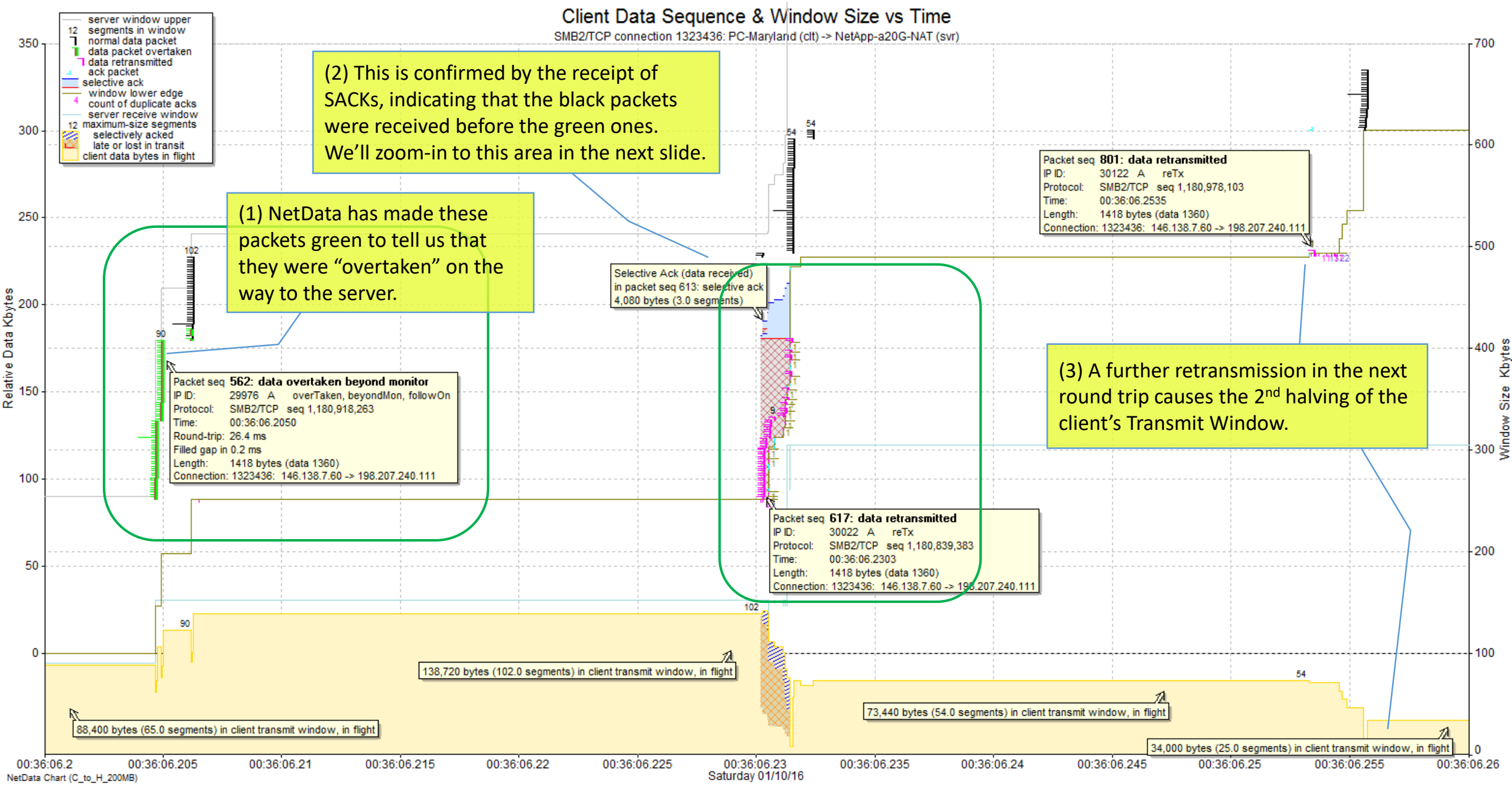
Now for the “live” analysis of the capture files.



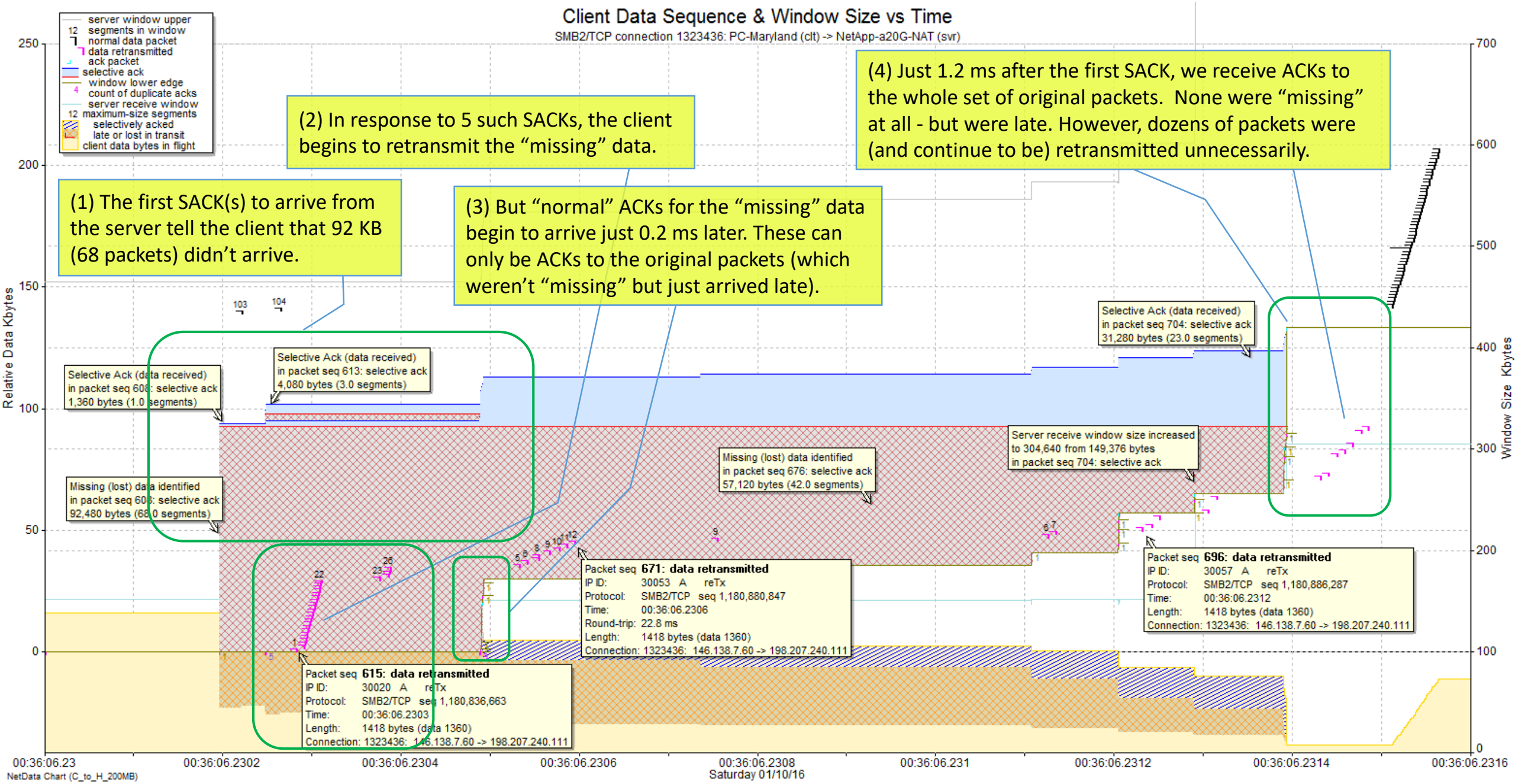
C to H Flow – End of Slow Start



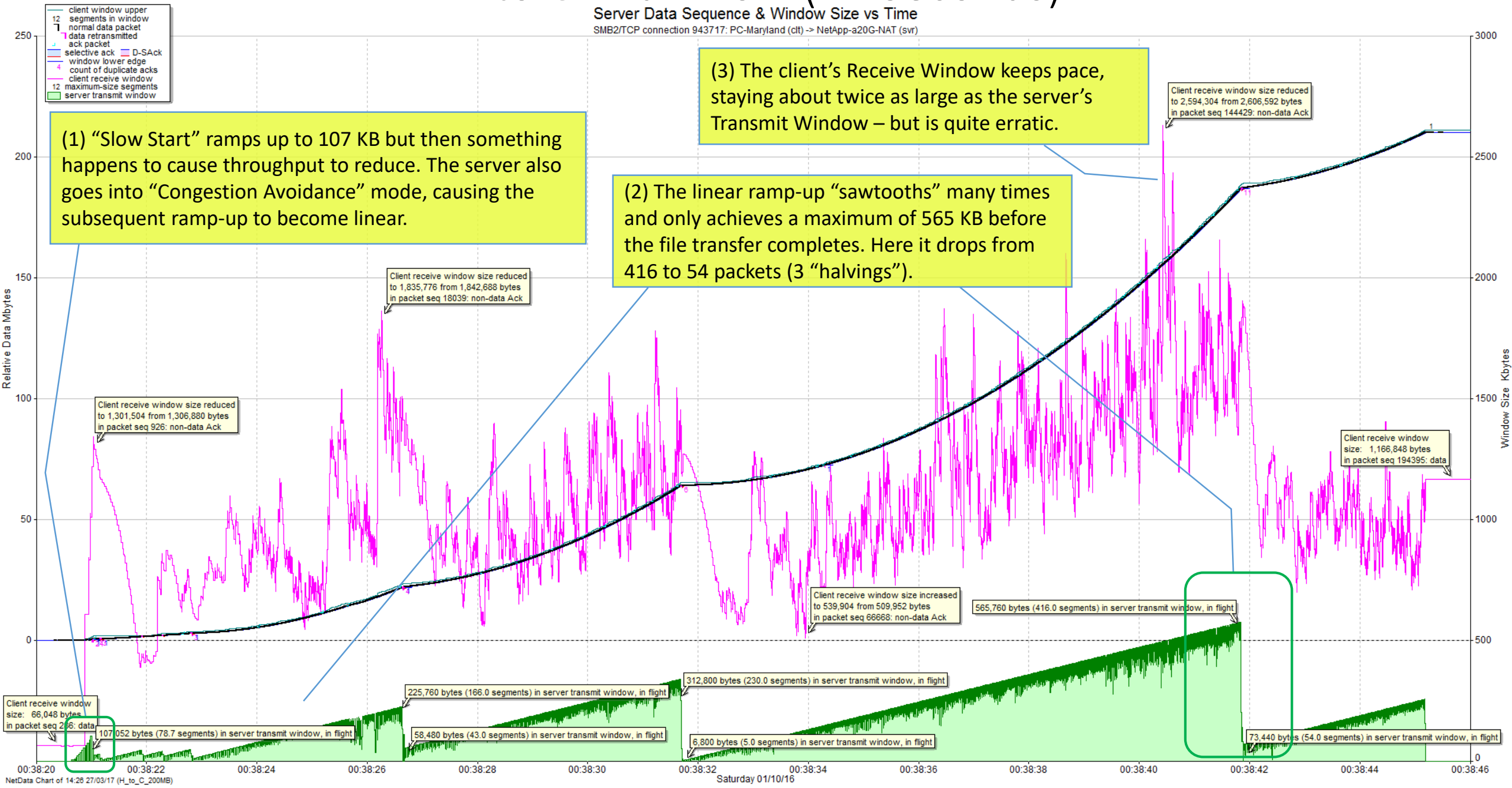
C to H Flow – Zoomed-In



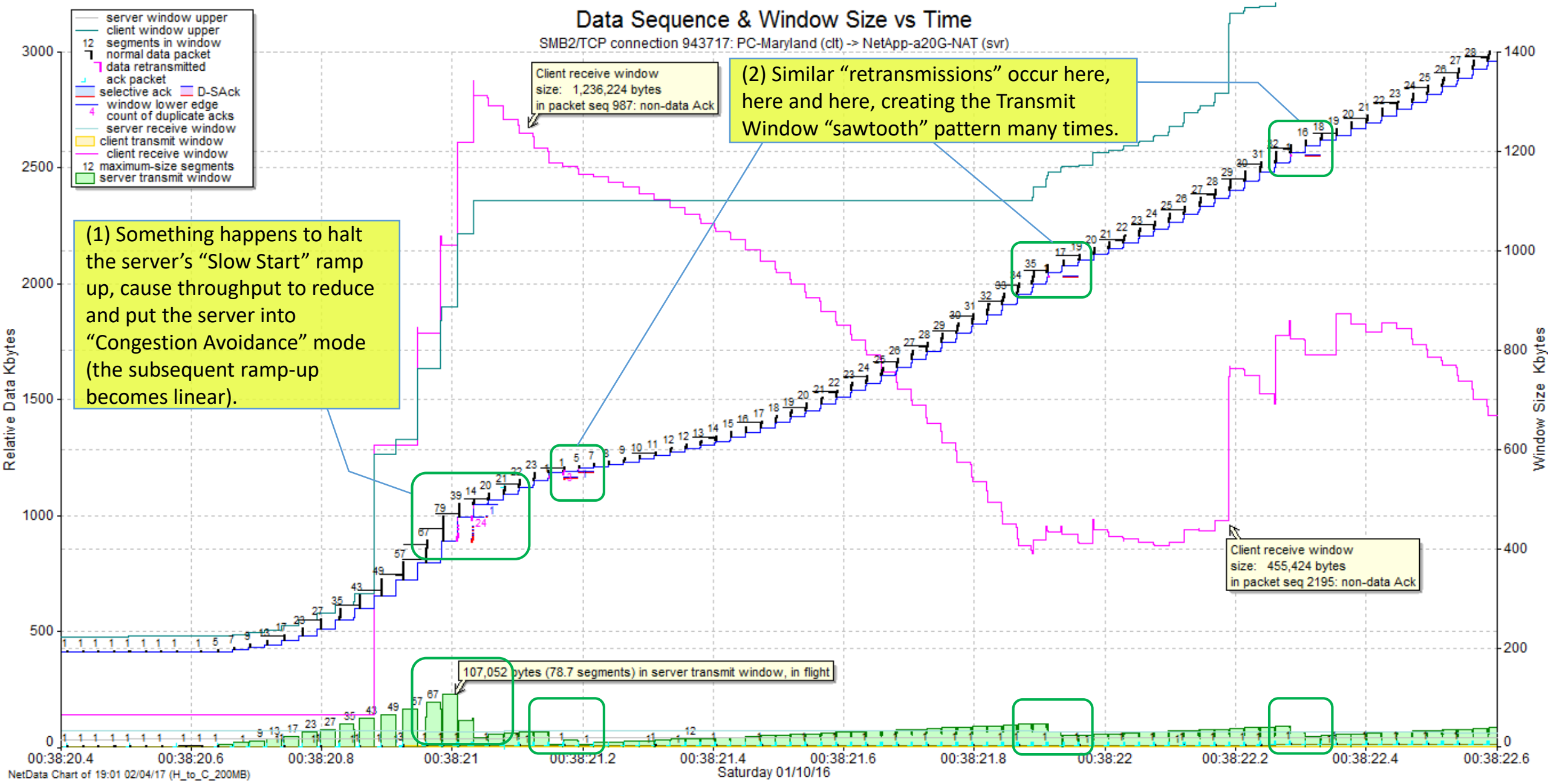
C to H Flow – Zoomed-In Further



H to C - Full Flow (24 Seconds)



H to C Flow – End of Slow Start

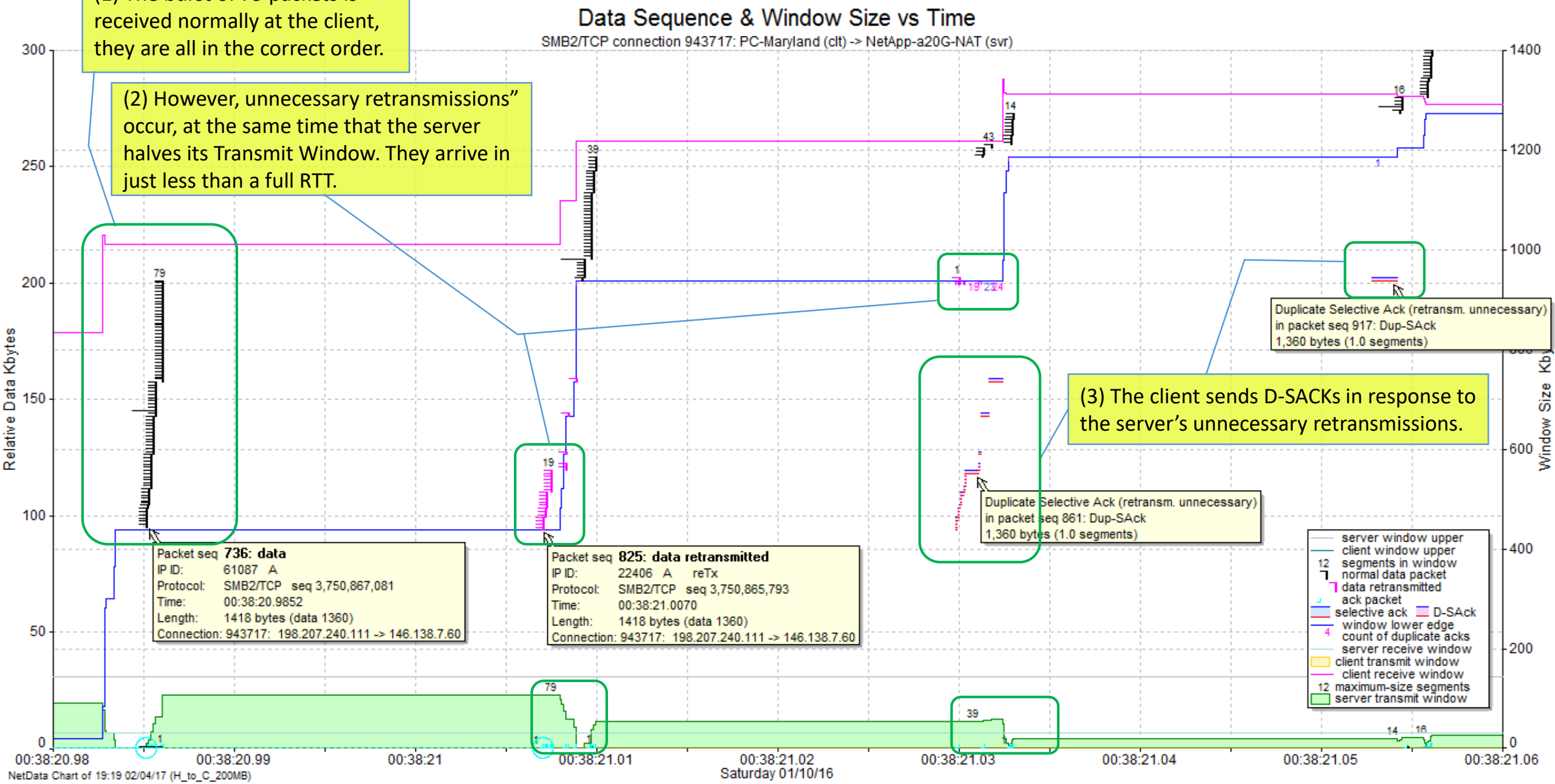


H to C Flow – Zoomed-In Further

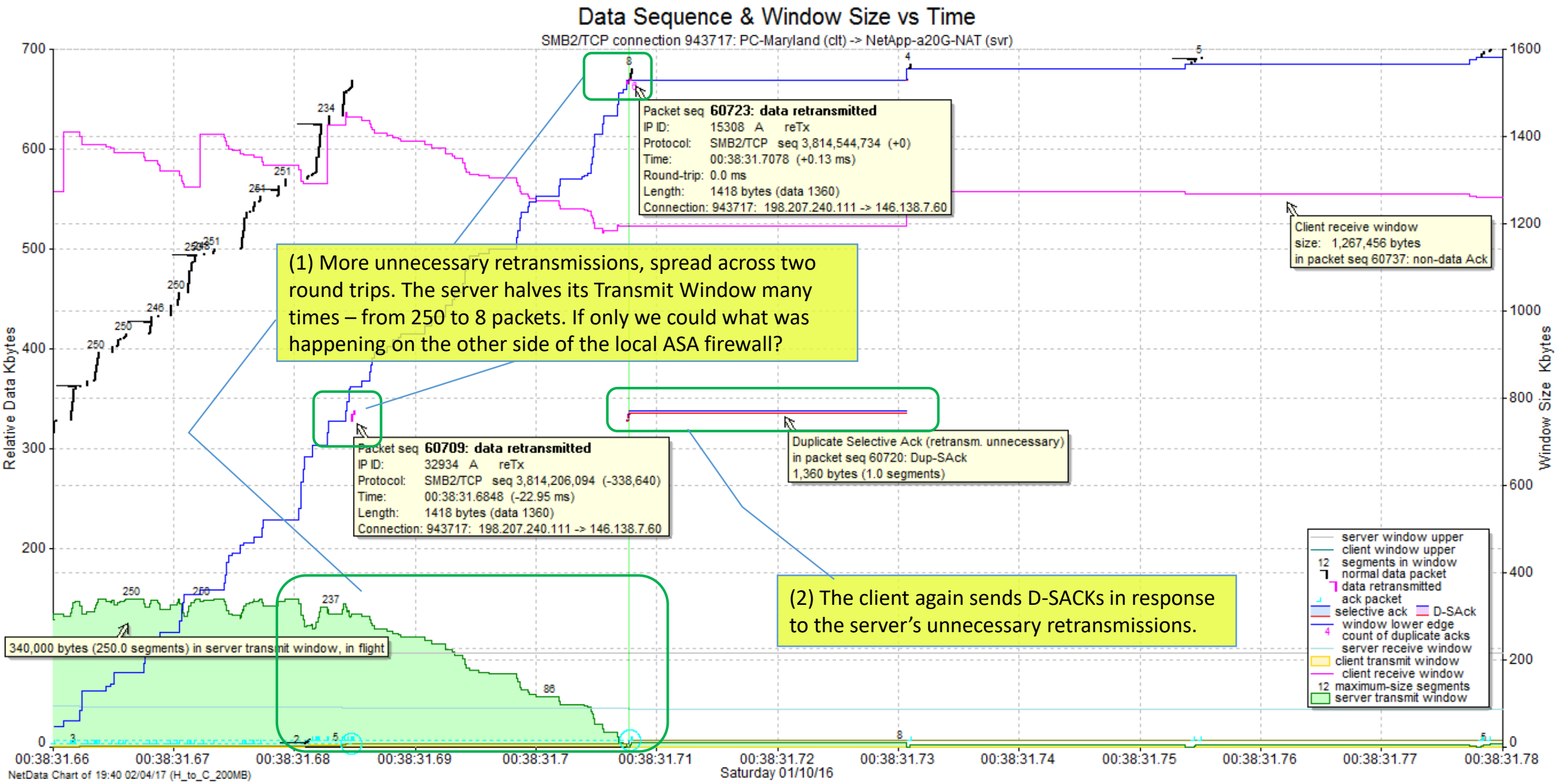
(1) The burst of 79 packets is received normally at the client, they are all in the correct order.

(2) However, unnecessary retransmissions" occur, at the same time that the server halves its Transmit Window. They arrive in just less than a full RTT.

(3) The client sends D-SACKs in response to the server's unnecessary retransmissions.



H to C – Later in Flow



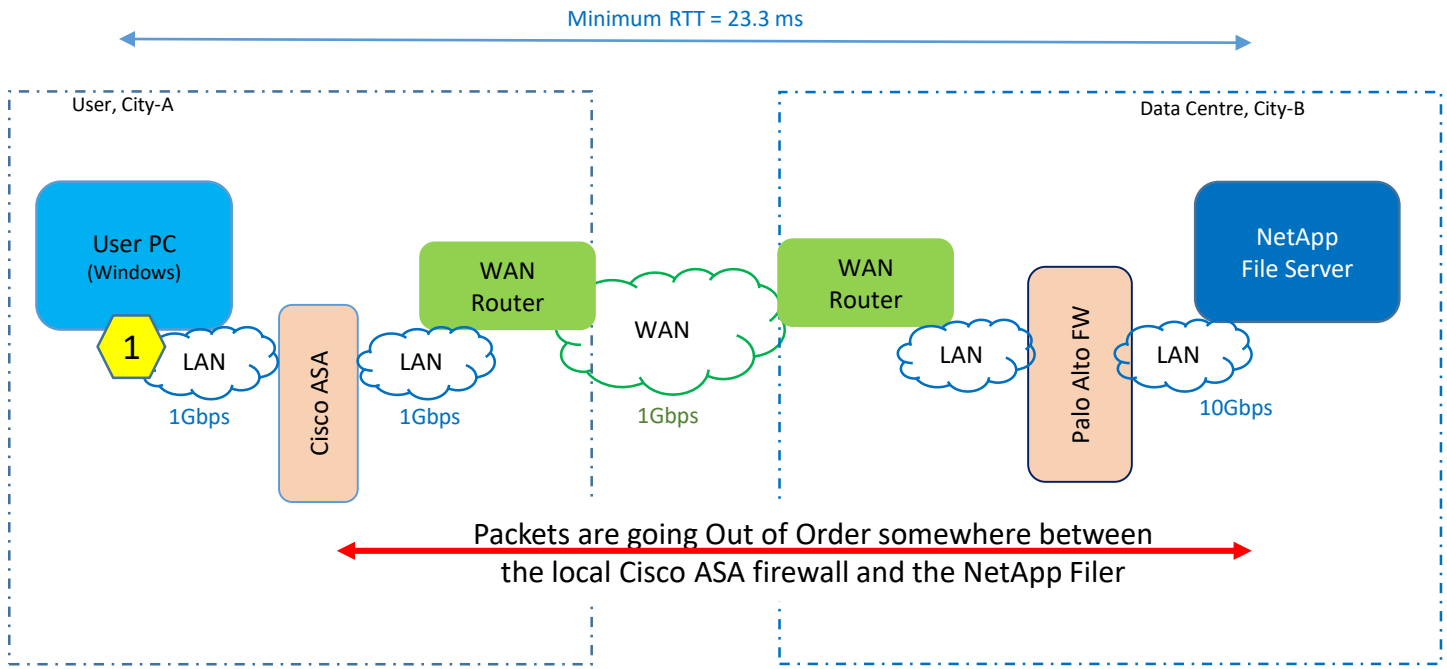
PC-to-NetApp “Root Cause”

Root Cause: Out of Order (OOO) packets are triggering SACKs, which in turn trigger unnecessary packet retransmissions.

Due to the retransmissions, the connection goes into “Congestion Avoidance Mode” where the sender halves its Transmit Window (often multiple times) and then ramps it back up in a linear manner.

We can also determine that the OOO events occur on the WAN side of the local ASA firewall.

The OOO events occur more often during the “H to C” file transfers, that is, when the large data flow is from the NetApp to the PC.



Packets from the NetApp arrive at the ASA “Out of Order”

ASA buffers them and issues SACKs back to the NetApp

The original “missing” packets arrive at the ASA “Late”

ASA releases all the buffered and “late” packets in the correct order

Client ACKs all the received data

NetApp unnecessarily retransmits packets (responding to ASA’s SACKs), ASA sends them through.

Client sends D-SACKs due to the duplicated packets

D-SACKs trigger more unnecessarily retransmitted packets from the NetApp

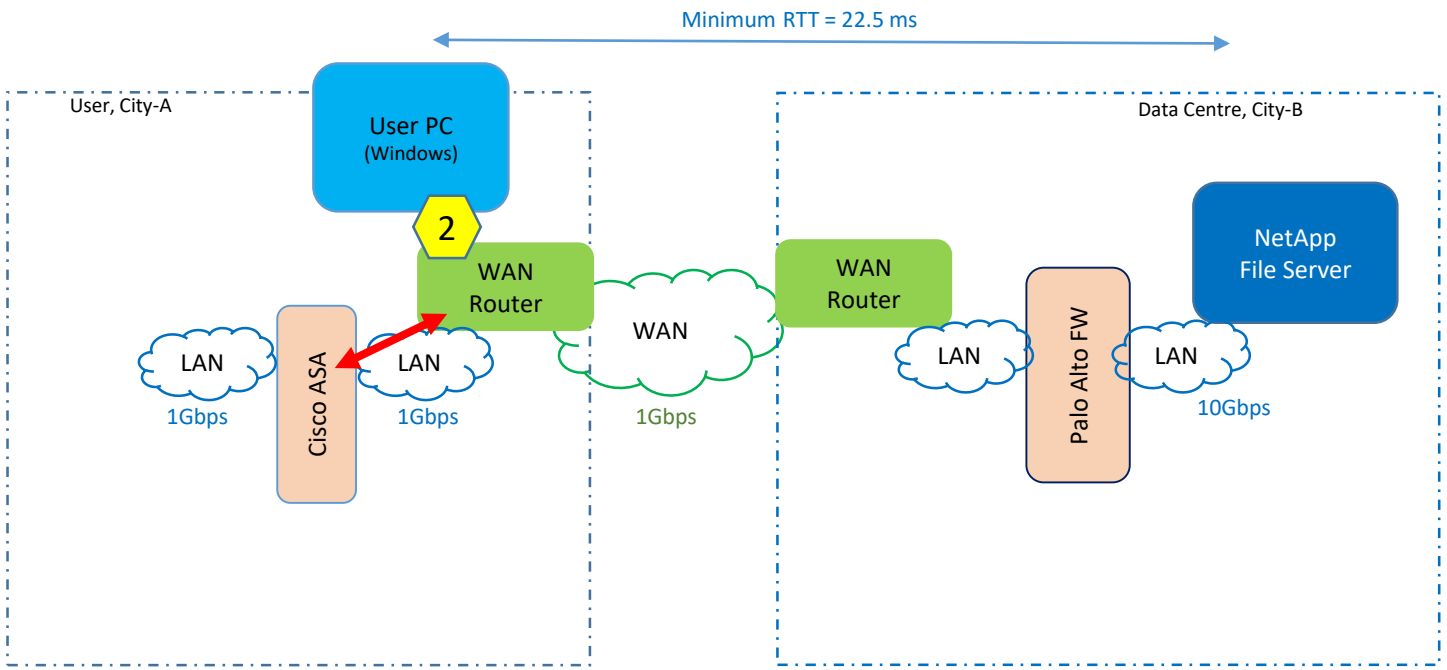
PC-to-NetApp – Subsequent Testing

More Tests: Laptops were taken to different sites and connected directly to the remote office WAN routers.

The same “C to H” and “H to C” file transfers to/from the NetApp were performed.

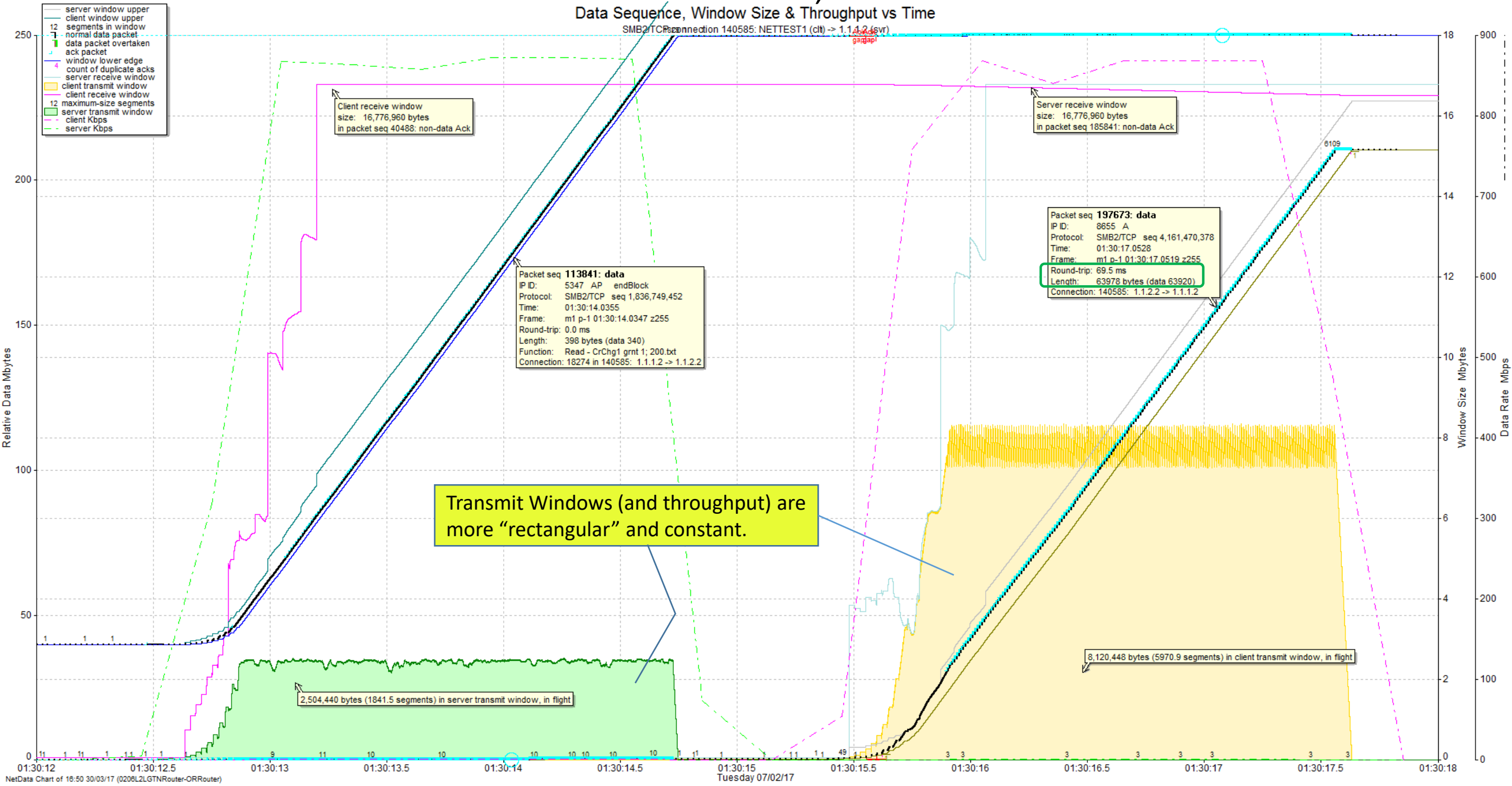
The next slides plot the flows achieved during these tests.

No OOO events were seen during these tests and throughput both ways was as good as it could be.



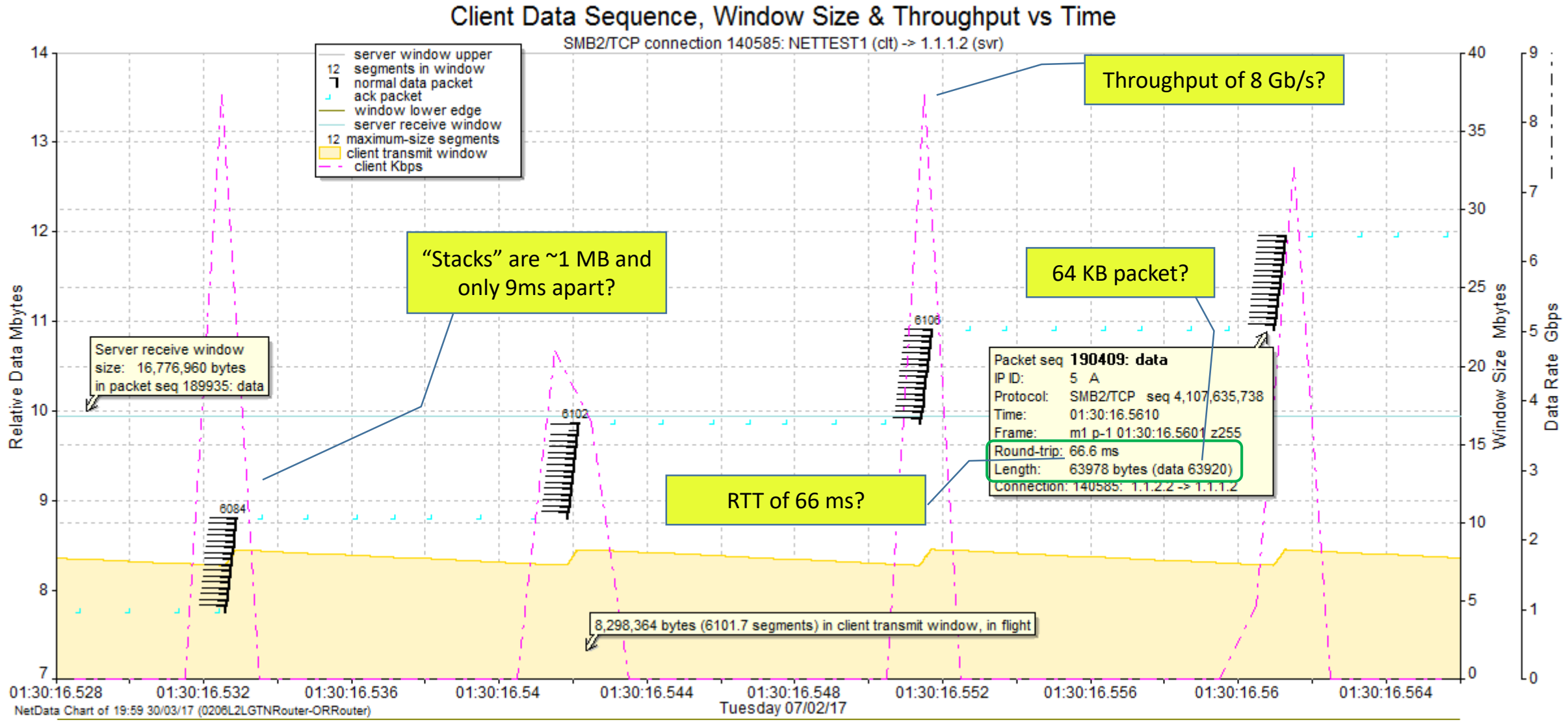
Therefore, the packets must be going Out of Order somewhere between the local Cisco ASA firewall and the local WAN router!!

“H to C” & “C to H” – Clean, Full Flows – 2.5 Secs



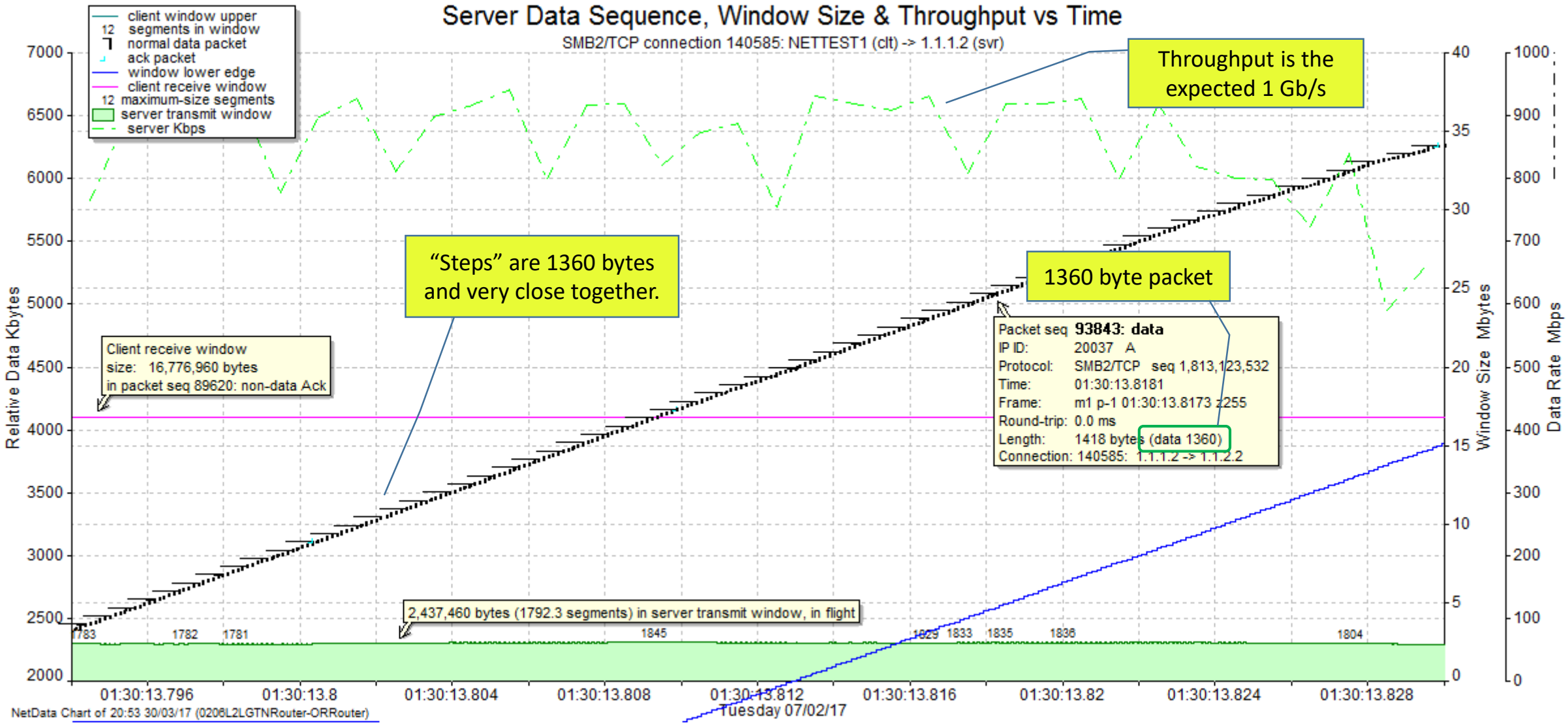
“C to H” – Capture Artefacts

Some things seem “odd” here? These are the “artefacts” of running the packet capture inside a PC or server. We are seeing the “packets” as they pass between the TCP layer to the IP layer inside the machine. It is IP’s job to chop the data into the MSS sized packets that end up on the wire. Data can flow internally from TCP to IP much faster than the wire speed. At this stage in the flow, the transmit window is just being “topped up” 1 MB at a time.
Lastly, if IP has 8 MB of data waiting in its Transmit Buffer waiting to get out on the 1 Gb/s wire, it will take 66ms for the whole 8 MB to leave the local machine.

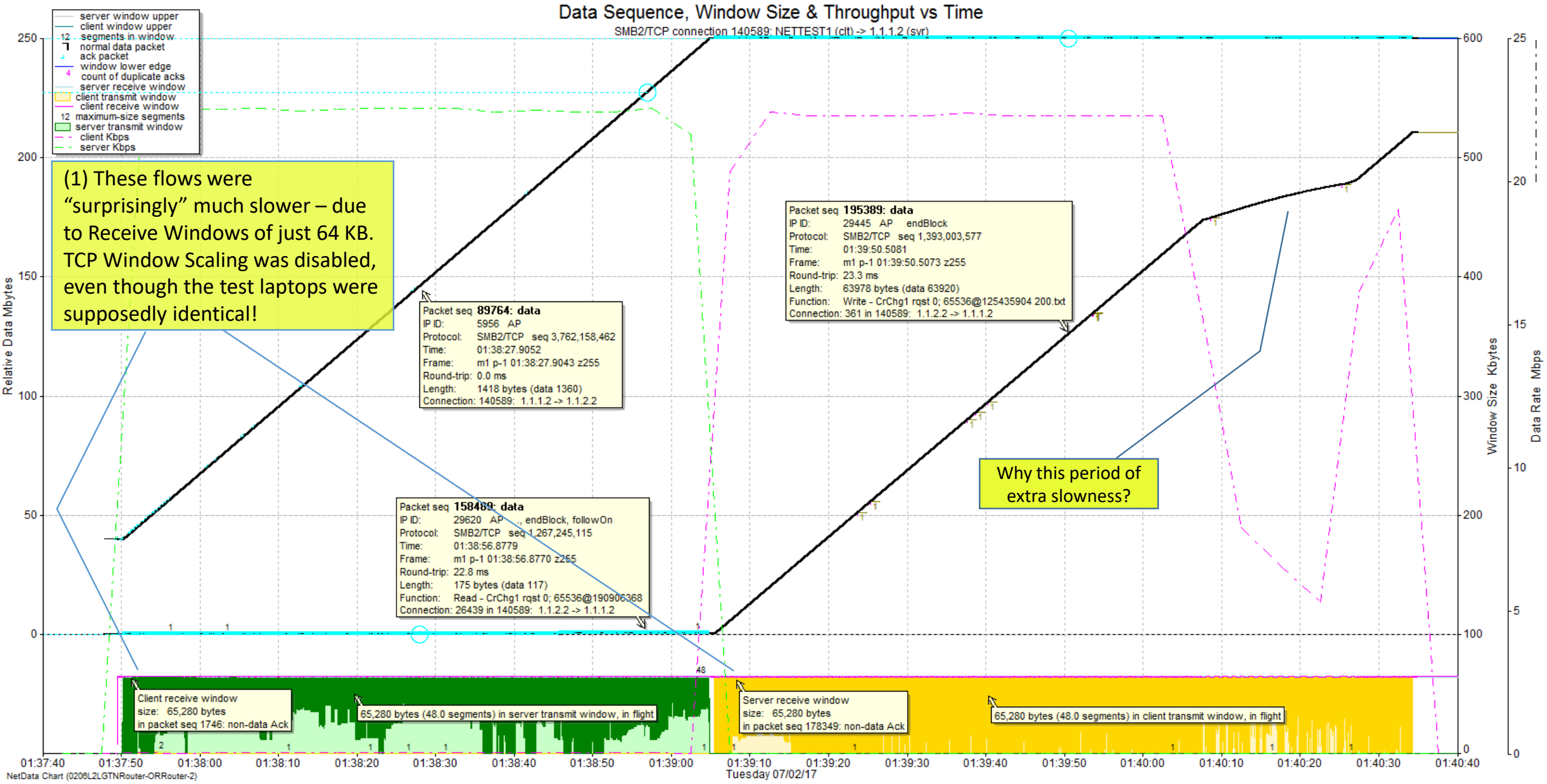


“H to C” – Same Timeframe

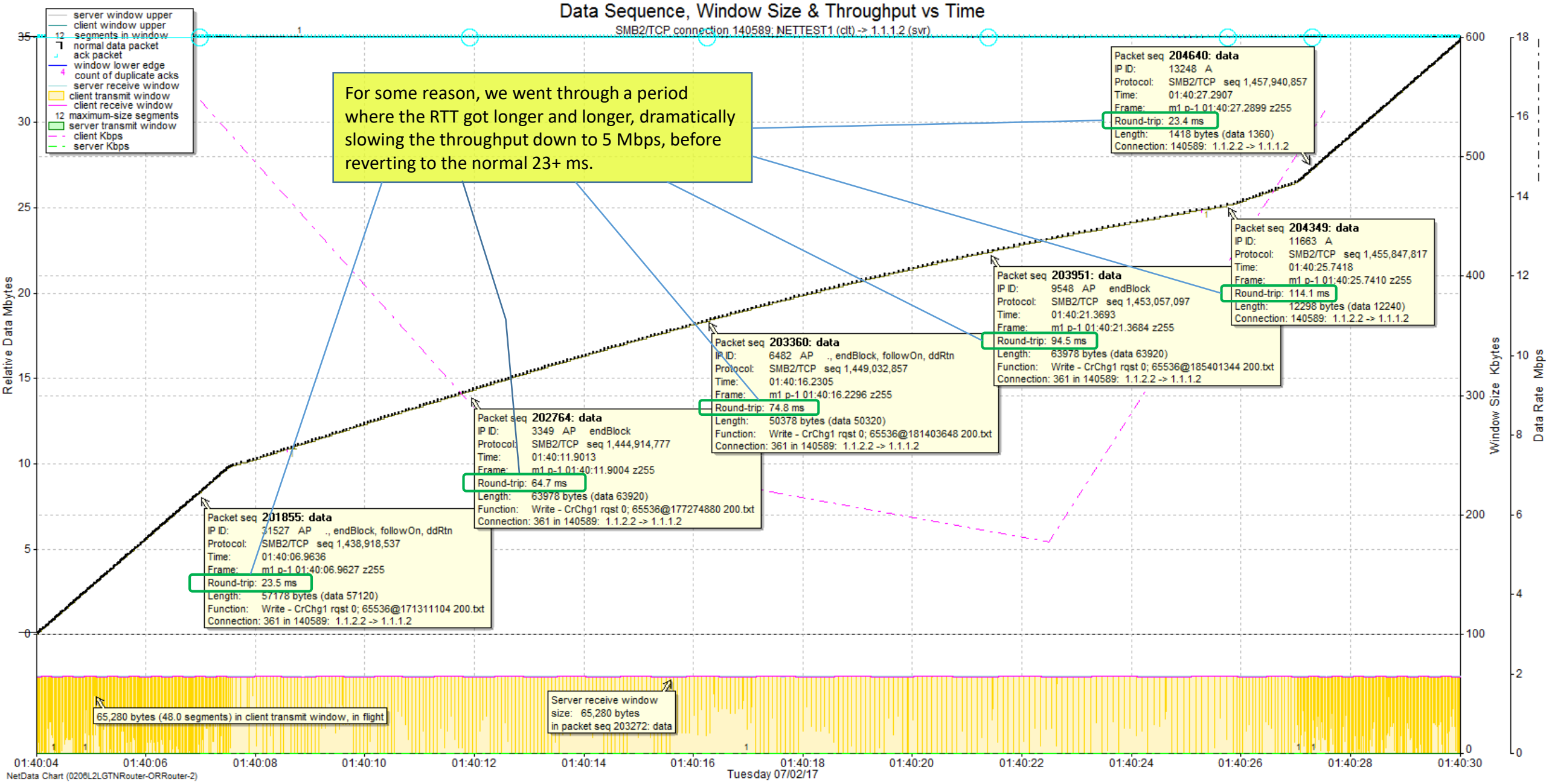
In the flow from the server, we see all the “packets” as they have arrived from the wire.
The timeframe here is about the same as the last slide (~36 ms) and we same about the same data volume in the period (~4 MB) but it looks much smoother because each vertical “step” is now just 1360 bytes. The server’s Transmit Window of 2.5 MB is enough to keep the 1 GB * 23 ms path full.



“H to C” & “C to H” – One Location – 75/90 Secs?



“C to H” – Zoom-in to Extra Slow Period



TCP/IP Protocol Terminology

Nearly All Done?

“Acknowledgement (ACK)” “Duplicate ACK” “Delayed ACK” “Partial ACK”
“Fast Retransmission” “Retransmission Timeout (RTO)”
“Round Trip Time (RTT)” “Bytes In Flight” “Bandwidth Delay Product (BDP)”
“Receive Window” “Transmit Window” “Window Scaling” “Long, Fat Network (LFN)”
“Slow Start” mode “Congestion Avoidance” mode

“3-Way Handshake” “Maximum Segment Size (MSS)”
“Selective Acknowledgement (SACK)” “Duplicate SACK”

“Out of Order (OOO)”

“Window Closure” aka “Zero Window”

SMB2 Transactions

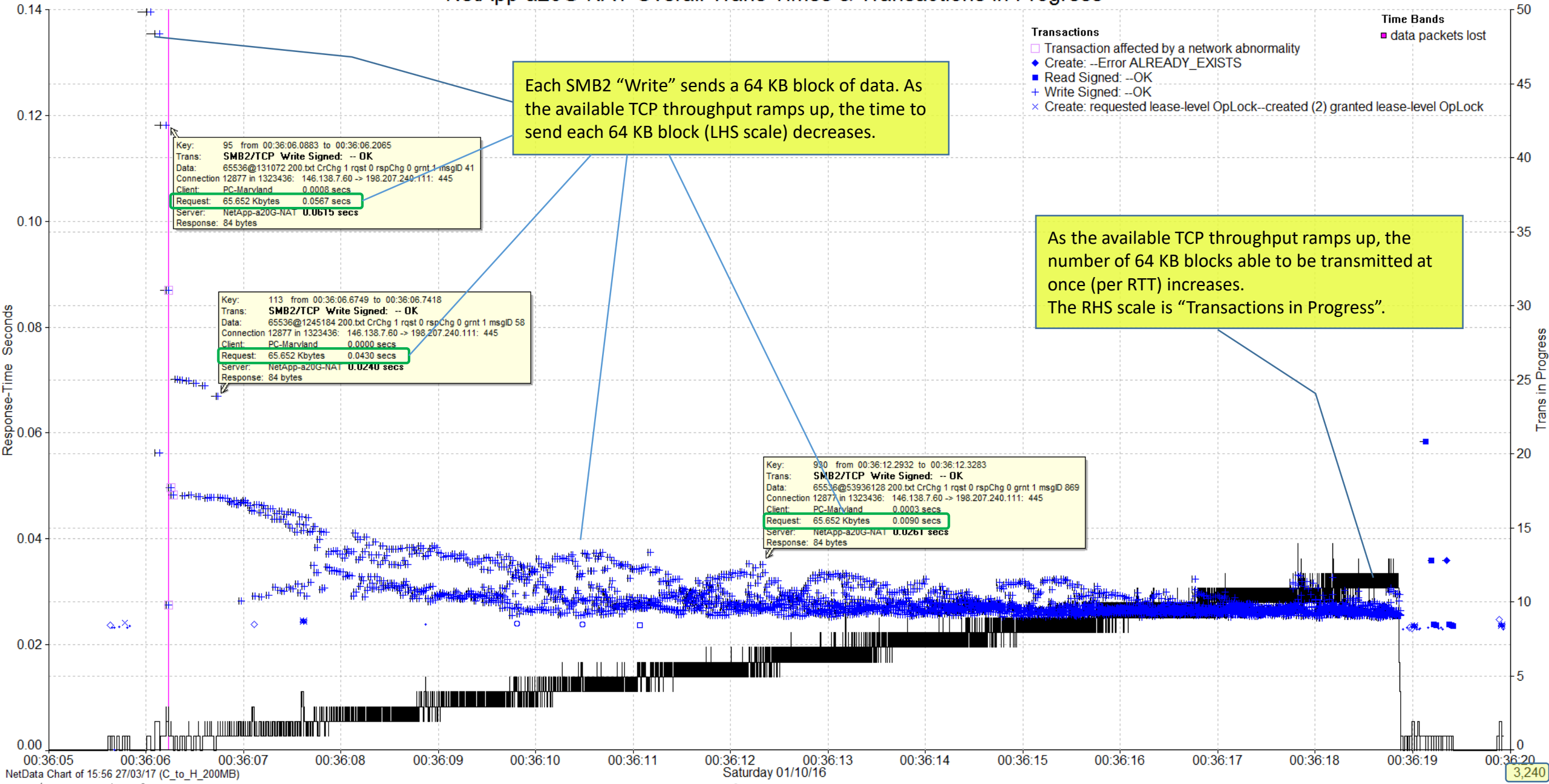
The following slides show the transaction Performance Charts for both the "C to H" and "H to C" file transfer tests.

The timing detail of each SMB2 transaction is displayed.

The reason for the different "SRT" times from Wireshark will be explained. We'll see that the NetApp Filer is not at fault in any way.

Do we have time to discuss them?

“C to H” - SMB2 Transactions

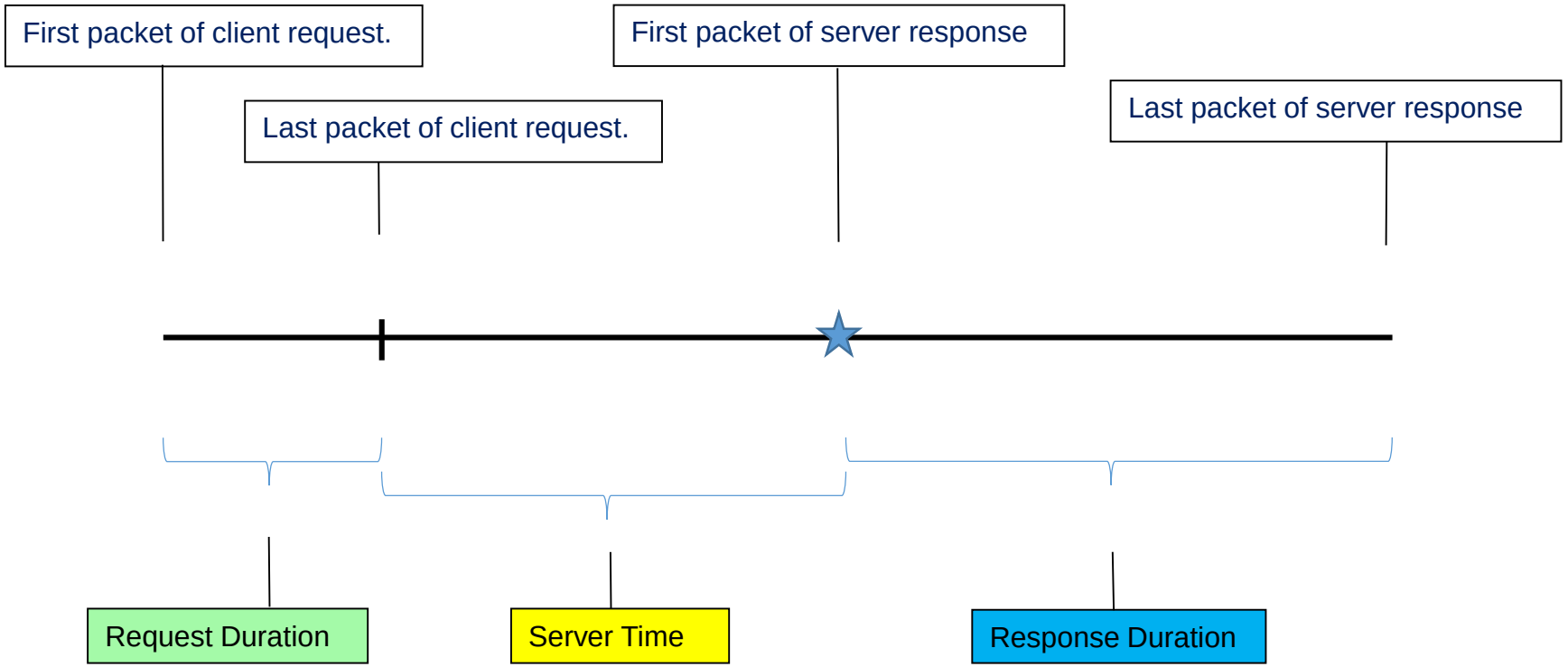


Understanding Performance Chart Transaction Symbols

NetData measures and reproduces individual application layer transactions. The “Performance Chart” plots each transaction as a horizontal line that represents the overall transaction time. Symbols within the line indicate timings of sub-components of the transaction. The symbol colour represents the server (or client) and symbol shape represents the transaction type.

The X-axis is time-of-day and the Y-axis is transaction time. A transaction is plotted at a height that represents the overall transaction time (full length of the horizontal line).

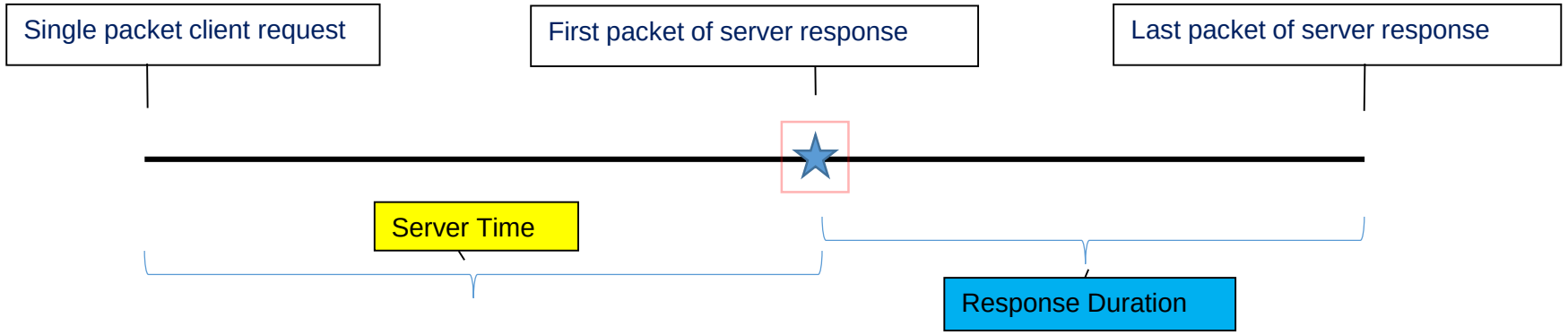
The green, yellow and blue colours are used consistently whenever these times are represented on the various charts.



Understanding Performance Chart Transaction Symbols

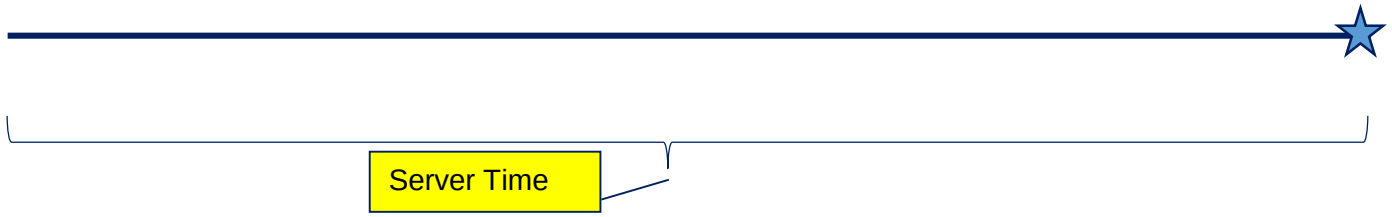
The length of the horizontal line before the coloured symbol represents “Server Time”.

Transactions that have been affected by a “network abnormality” get a pink square around the symbol.

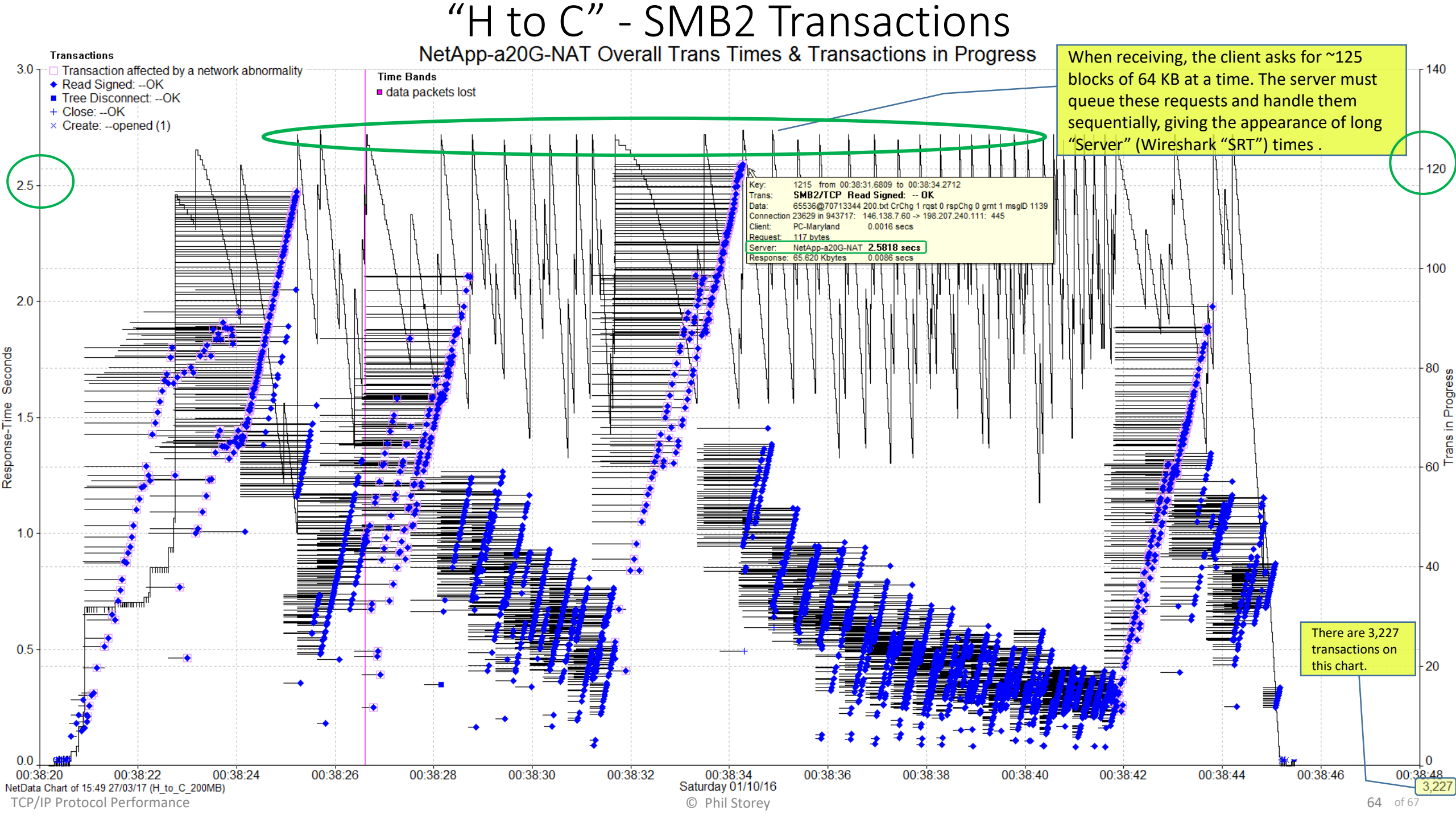


Extremely common.

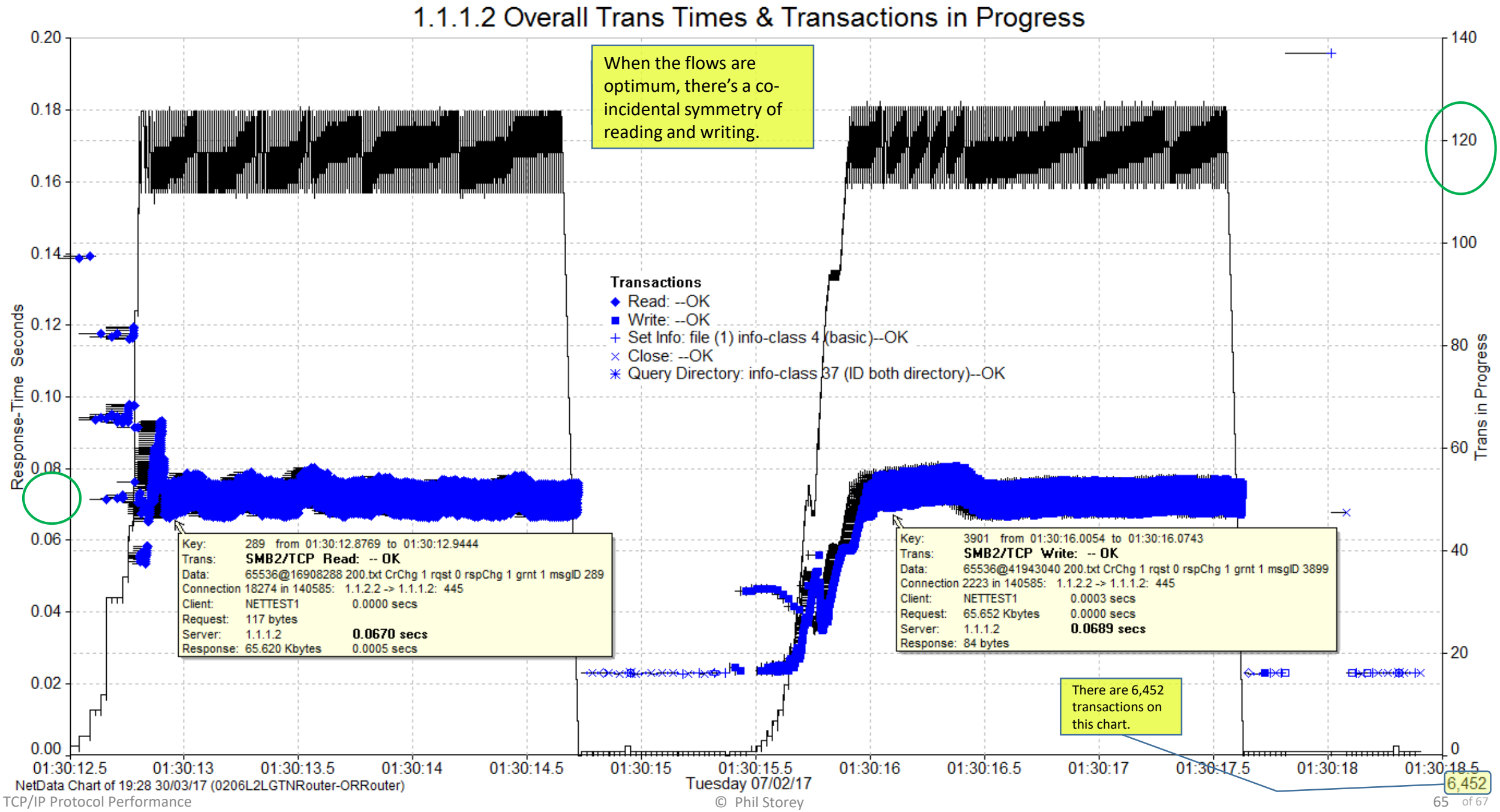
If the server response is also small (or relatively fast), then the coloured symbol will be at the very end of the line.
In this case, the whole transaction duration appears to be “Server Time”.



“H to C” - SMB2 Transactions



“H to C” & “C to H” – Clean SMB2 Transactions



Receive Window Closures

The following slide is just a nice example of a file transfer flow that contains several Receive Window “closures” (or what Wireshark calls “Zero Window” events).

The flow was captured at the receiving end of a low speed WAN that had WAN Accelerators deployed at each end.

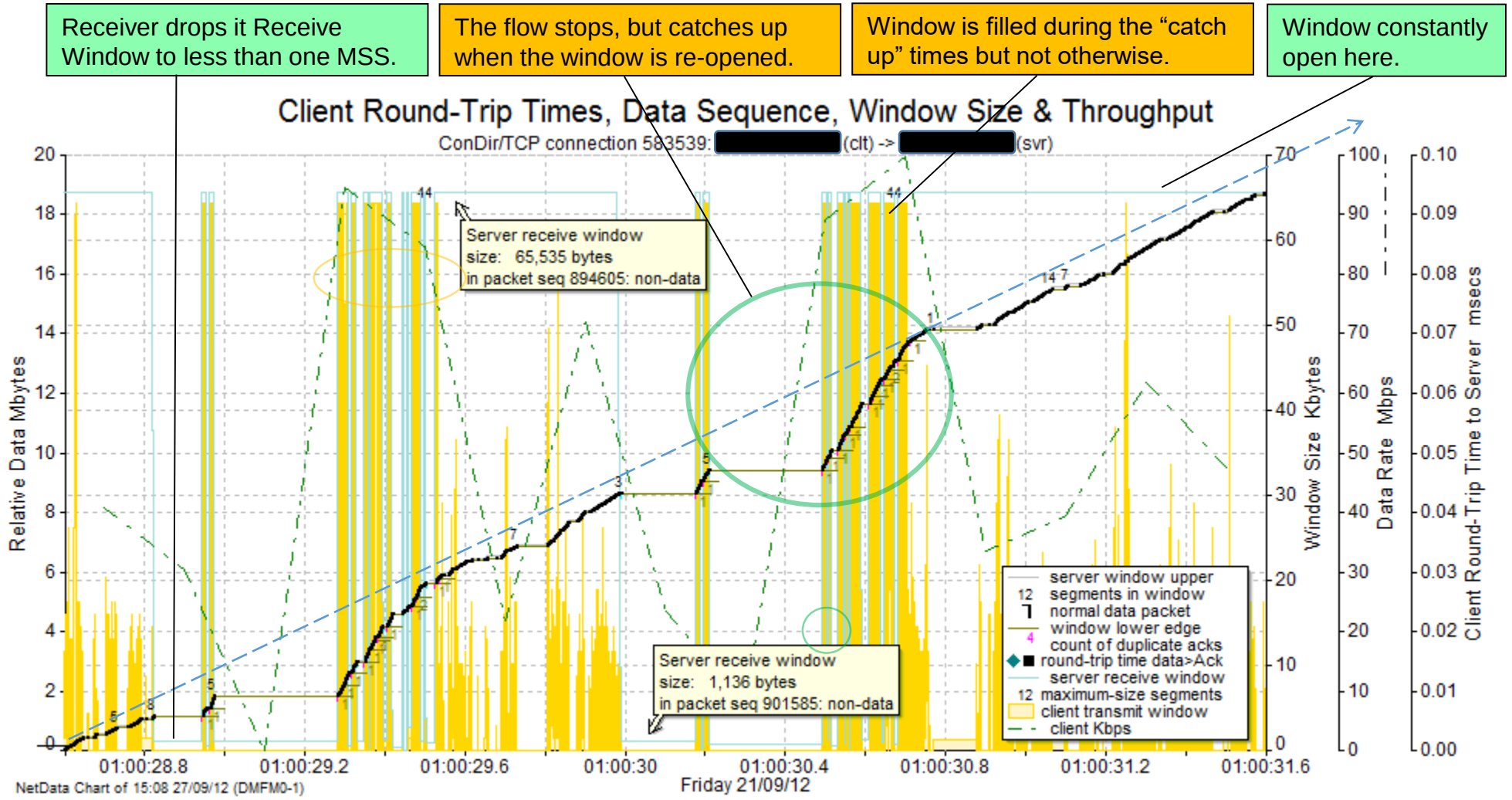
The throughput being achieved by the WAN link was only about half of the expected capability.

The vendor who managed the WAN link and WAN Accelerator devices was claiming that the receiver’s many window closures were the cause of the poor throughput performance (the WAN Accelerators logged such things). Therefore, they wouldn’t investigate the poor WAN throughput until we made the window closures go away.

Were they correct?

Receive Window Closed/Opened

In this example, the Receive Window is closed (dropped to less than one MSS size), the sender stops transmitting but “catches up” once the Receive Window is re-opened (to 64 KB). The window closures do not, therefore, affect the overall file transfer throughput or time. A local WAN Accelerator did not pass the window closures through to the sending side – but rather, let packets continue to flow over the low speed WAN, to be buffered in the local WAN accelerator and released at local LAN speed when the Receive Window was re-opened. Thus, the overall time for the file transfer (throughput) was not affected by the window closures.



TCP/IP Protocol Terminology

All Done!!

“Acknowledgement (ACK)” “Duplicate ACK” “Delayed ACK” “Partial ACK”
“Fast Retransmission” “Retransmission Timeout (RTO)”
“Round Trip Time (RTT)” “Bytes In Flight” “Bandwidth Delay Product (BDP)”
“Receive Window” “Transmit Window” “Window Scaling” “Long, Fat Network (LFN)”
“Slow Start” mode “Congestion Avoidance” mode

“3-Way Handshake” “Maximum Segment Size (MSS)”
“Selective Acknowledgement (SACK)” “Duplicate SACK”

“Out of Order (OOO)”

“Window Closure” aka “Zero Window”

Phil Storey

Phil@NetworkDetective.com.au

www.NetworkDetective.com.au

au.linkedin.com/in/philipstorey3

@PhilStorey24

www.youtube.com/c/NetworkDetective

ask.wireshark.org: @philst