Troubleshooting TCP Unidirectional Data Transfer Throughput

Packet Trace Analysis Using Wireshark - September 8, 2023

VMware, Inc. 3401 Hillview Avenue Palo Alto CA 94304 USA Tel 877-486-9273 Fax 650-427-5001 www.vmware.com

Table of Contents

Audience

This paper is for developers, advanced admins, and tech support specialists who want to troubleshoot TCP data transfer throughput issues that can affect the performance of customers' or their own VMware vSphere® environments.

Introduction

Data transfer over TCP is very common in vSphere environments. Examples include storage traffic between the VMware ESXi™ host and an NFS or iSCSI datastore, and various forms of vMotion traffic between vSphere datastores.

We have observed that even extremely infrequent TCP issues could have an outsized impact on overall transfer throughput. For example, in our experiments with ESXi NFS read traffic from an NFS datastore, a seemingly minor 0.02% packet loss resulted in an unexpected 35% decrease in NFS read throughput [1].

In this paper, we describe a methodology for identifying TCP issues that are commonly responsible for poor transfer throughput. We capture the network traffic of a data transfer into a packet trace file for offline analysis. This packet trace is analyzed for signatures of common TCP issues that may have a significant impact on transfer throughput.

The TCP issues considered include packet loss and retransmission, long pauses due to TCP timers, and Bandwidth Delay Product (BDP) issues. We use Wireshark to perform the analysis and provide a Wireshark profile to simplify the analysis workflow. We describe a systematic approach to identify common TCP issues with significant transfer throughput impact, and recommend engineers troubleshooting data transfer throughput performance to include this methodology as a standard part of their workflow.

We assume our readers to be familiar with the relevant TCP concepts described in this paper and have a good working knowledge of Wireshark. For additional information about these topics, refer to the SharkFest Retrospective [2] page of recent SharkFest Conferences.

Things to Know Before Starting

Notation used

- S: Sender
- R: Receiver

About unidirectional data transfer

Consider unidirectional data transfer from sender S to receiver R over TCP.

Network traffic consists of the following:

- Data segments: S → R
- ACK, duplicate ACK: R → S
- Retransmit: S → R
- TCP zero window: R → S

Wireshark Profile: TcpTransferTput

In this paper, we use the profile TcpTransferTput (a zip file) to simplify identifying many TCP issues that impact data transfer throughput. You can find a detailed description of the profile in the appendix section "Wireshark Profile: TcpTransferTput."

To use this profile:

1. Download the profile (zip file) at https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/techpaper/performance/TcpTransferTput.zip
2. In Wireshark:
    a. **Edit → Configuration Profiles → Import → from zip file**
    b. **Edit → Configuration Profiles → TcpTransferTput**

Performance Anomalies

In this section, we describe signatures of performance anomalies that have a significant impact on data transfer throughput.

Classes of performance anomalies for data transfer throughput

There are three classes of performance anomalies for data transfer throughput:

- **Packet Loss, Out of Order:** This class includes packet loss, out of order, duplicate ACKs, and fast and slow retransmissions. Wireshark issues TCP warnings about these anomalies.
- **Long Pause, High Latency:** This class includes TCP timer-related anomalies, such as delayed ACK or Nagle delay. These anomalies typically consist of long repeated pauses of the same duration.
- **Bandwidth Delay Product:** This class consists of anomalies regarding the interaction of the number of bytes sent but not yet acknowledged, receive window size, and TCP send and receive buffer sizes. There is no packet loss, no retransmission, and no long pause.

Table 1 summarizes the characteristics of these classes of performance anomalies.

Class	Characteristics	Highlight
Packet Loss, Out of Order	Retransmit (fast, slow)	Wireshark TCP warnings
Long Pause, High Latency	TCP timers, network latency, storage latency	Timer-based
Bandwidth Delay Product	Bytes in flight, receive window, Slow Start	No Wireshark warnings, no long pauses

Table 1. Classes of performance anomalies

Identification of performance anomalies using Wireshark

The following table shows various methods of identifying performance anomalies using Wireshark: Expert Info, Display Filter, Tcptrace graph, and I/O graphs (from the TcpTransferTput profile).

Class	Detail	Wireshark: Column (Packet List pane)	Wireshark: Display Filter (Time in seconds)	Wireshark: Feature	Info
Packet Loss, Out of Order	DUP ACK		tcp.analysis.duplicate_ack	Expert Info • DUP ACK • Out of order • Fast retransmit • Retransmit	Require trailing segment(s)
	3x DUP ACK		tcp.analysis.duplicate_ack_num >= 3		Require 3+ trailing segments
	Out of order		tcp.analysis.out_of_order		
	Fast retransmit		tcp.analysis.fast_retransmission		
	Slow		tcp.analysis.retransmission		

Class	Detail	Wireshark: Column (Packet List pane)	Wireshark: Display Filter (Time in seconds)	Wireshark: Feature	Info
	retransmit (RTO)		tcp.analysis.flags		
Long Pause, High Latency	Inter-frame time	frame.time_delta_displayed	(frame.time_delta > 0.001)	Tcptrace graph • Long pause	
	TCP RTO Timer	frame.time_delta_displayed	(frame.time_delta > 1)		
	TCP Delayed ACK Timer	frame.time_delta_displayed	(frame.time_delta > 0.1)		
	Network iRTT latency	tcp.analysis.initial_rtt	(tcp.analysis.initial_rtt > 0.01)		TCP 3-way handshake
	Network ACK RTT latency	tcp.analysis.ack_rtt	(tcp.analysis.ack_rtt > 0.01)		
	Storage NFS Call-Reply latency	rpc.time	(rpc.time > 0.01)		
	Storage iSCSI Request-Response latency	scsi.time	(scsi.time > 0.01)		
Bandwidth Delay Product	Receiver window size (scaled)	tcp.window_size		Tcptrace graph • Bytes In Flight • Receive window I/O graphs (TcpTransferTput profile) Obs Point: Closer to Sender	from receiver
	Sender BytesInFlight	tcp.analysis.bytes_in_flight			from sender
			tcp.analysis.zero_window		from receiver
			tcp.analysis.window_full		from sender
			tcp.analysis.zero_window_probe		from sender

Table 2. Methods of identifying performance anomalies using Wireshark

> **Note:** The time values in Display Filter expressions here (for example, `(rpc.time > 0.01)`) are for illustration only. You should use values that are reasonable for your environment.

Regarding long pauses:

- TCP timers are typically in some round number of milliseconds (for example: 10ms, 200ms), so they are easy to identify.

- Known interactions leading to long pauses (not an exhaustive list):
  - Nagle (delayed send) interacts with Delayed ACK. See the explanation by Hansang Bae: Wireshark Tutorial of TCP Nagle and Delayed ACK Interaction [3].
  - Slow Start interacts with Delayed ACK. Refer to NFS Read: Slow Start vs Delayed ACK [1].

  These deadlocks are eventually broken by typically slow TCP timers, resulting in poor throughput.

Class	**Characteristics**	**Signature Identification using Wireshark**
Packet Loss, Out of Order	Retransmit (fast, slow)	Expert Info: DUP ACK, out of order, fast retransmit, retransmit Display Filter: from TcpTransferTput profile
Long Pause, High Latency	TCP timers, network latency, storage latency	Tcptrace graph: long pause Display Filter: from TcpTransferTput profile
Bandwidth Delay Product	Bytes in flight, receive window, Slow Start	Tcptrace graph: bytes in flight, receive window I/O Graphs: from TcpTransferTput profile Observation Point: Closer to Sender

Table 3. Summary of signature identification

Wireshark Analysis

The following sections step through the Wireshark analysis methodology we use.

High-level workflow

Use Wireshark to identify performance anomaly signatures by following this workflow (we'll describe the details in the following sections):

1. Note packet capture conditions and capture drops.
2. Note packet trace file stats and TCP connections.
3. Export the selected TCP connection to a packet trace file.
4. Identify performance anomalies using signatures.

Step 1: Note packet capture conditions and capture drops

In this step, we review key information about the packet capture process. You will use this information to estimate whether the packet trace file is an accurate representation of the network traffic you are analyzing.

Review the following:

1. Packet capture details

- Where were packets captured?
  - On a network endpoint (for example: on an ESXi vmknic using `tcpdump-uw` or `pktcap-uw`)
  - Via an inline network tap on the physical network
  - Via the mirror/SPAN port of a physical network switch
- Was a hardware packet capture system used?
- Observation point: Was the capture performed closer to the sender or the receiver on the network?

2. Experiment conditions

- Was the performance anomaly being investigated observed during capture?
- What was the time of day of the anomaly?

3. Capture drops

- Capture drops are packets in the traffic being analyzed but are not included in the packet trace file, due to limitations of the packet capture process. Capture drops are common in endpoint captures and could make packet trace analysis significantly more challenging.
- You can identify capture drops in a packet trace file using Wireshark as follows:
  - In Wireshark, open the packet trace file. For example: `xyz.pcapng.gz`

  Use the Display Filter
  `(tcp.analysis.lost_segment || tcp.analysis.ack_lost_segment)`

  which is predefined as the Display Filter button **TcpCapDrops** in the TcpTransferTput profile.

For more information about packet capture, refer to our paper ESX IP Storage Troubleshooting Best Practice: Packet Capture and Analysis at 10G [3].

Step 2: Note packet trace file stats and TCP connections

In this step, we examine the number of bytes, number of packets, and duration of the packet trace file.

1. In Wireshark, open the packet trace file. For example: `xyz.pcapng.gz`
2. Go to **Statistics** → **Capture File Properties** and note the following:
   - **Time: First Packet** (Time of Day)
   - **Statistics: Packets**, **Time Span** (duration), **Bytes**

For each TCP connection in the file, examine similar statistics and identify the data transfer direction.

1. In Wireshark, go to **Statistics → Conversations → TCP**.

   Per connection:

   a. Note the **Packets**, **Bytes**, and **Duration**.

   b. Identify the data transfer direction (A → B or B → A), which will have higher bytes than the other direction.

Step 3: Export the selected TCP connection to a packet trace file

In this step, we select a TCP connection of interest based on, say, the endpoint IPs and ports, or the number of packets or bytes. We then export all the packets of this TCP connection to a new packet trace file. Finally, we perform further analysis of this selected TCP connection on the new packet trace file. This is an optional but recommended step—it simplifies some of the subsequent analysis steps in Wireshark. If we have multiple TCP connections of interest, we need to export each connection to a separate packet trace file for further analysis.

1. Select the TCP connection of interest with the desired endpoint IPs and ports.

> **Note:** If you identify multiple connections of interest, you will need to select (and export) each connection separately.

2. Select all frames of one TCP connection of interest. There are two ways to go about this:

   Method 1

   a. In the Wireshark Packet List pane, select one frame from the TCP connection of interest.

   b. Use the Display Filter `(tcp.stream == ${tcp.stream})` which is predefined as the Display Filter button **TcpStream** in the TcpTransferTput profile.

   c. If the Packet List pane is sorted by some column other than the frame.number (No.) column—the leftmost column—revert to sort by the frame.number column.

   Method 2

   a. In Wireshark, go to **Statistics → Conversations → TCP** and select one connection.

   b. Select **Apply as Filter → Selected → Filter on stream id**. The Packet List pane now shows only frames of the selected TCP connection.

3. Export the displayed frames to a new packet trace file by selecting **File → Export Specified Packets**.

   - The file name should be in the form `xyz.connection=??.pcapng.gz`
   - Save as type: **Wireshark/... pcapng**
   - Compressed with gzip: **Y**
   - Select **All packets**.
   - Select **Displayed**.

Step 4: Identify performance anomalies using signatures

In this step, we use different Wireshark features to identify various performance anomaly signatures. The features used include:

- Display Filter
- Tcptrace graph
- I/O graphs

The use of these features will be described in the following sections.

> **Note:** We assume that the packet trace file contains packets from one TCP connection.

Use Display Filter Signatures

Here, we use predefined display filters (from the TcpTransferTput profile) to identify performance anomaly signatures.

1. Iterate over the predefined display filter buttons to identify performance anomaly signatures.
2. If some frame (F) matches a signature, analyze the TCP connection frame F belongs to. Examine frame F and successively earlier frames in the TCP connection until you discover the underlying issue.
3. For each performance anomaly identified in a TCP connection, determine if the frequency and severity of the anomaly together results in a significant impact on transfer throughput.

   Example: An instance of packet loss followed by fast retransmit does not always lead to a significant decrease in transfer throughput.

Table 4 shows performance anomaly signatures and the corresponding display filters.

Class	Display Filter Button	Display Filter Expression	Direction
Packet Loss, Out of Order	TcpDupAck3+	tcp.analysis.duplicate_ack_num >= 3	R→S
	TcpRetrans(F/S)	tcp.analysis.fast_retransmission || tcp.analysis.retransmission	S→R
	TcpOO	tcp.analysis.out_of_order	S→R
Long Pause, High Latency	Pause=1+ms	frame.time_delta > 0.001	(any)
	Pause=10+ms	frame.time_delta > 0.01	(any)
	Pause=100+ms	frame.time_delta > 0.1	(any)
Bandwidth Delay Product	TcpWinFull	tcp.analysis.window_full	S→R
	TcpZeroWin	tcp.analysis.zero_window	R→S

Table 4. Performance anomaly signatures and their corresponding display filters

Example: For the anomaly signature TCP triple duplicate ACK (which would trigger a TCP fast retransmit), you will use the following analysis steps:

1. Use the Display Filter `(tcp.analysis.duplicate_ack_num >= 3)` which is predefined as the Display Filter button **TcpDupAck3+** in the TcpTransferTput profile.
2. Select one matching frame (F) in the **Packet List** pane.
3. Click the Display Filter `(tcp.stream == ${tcp.stream})` which is predefined as the Display Filter button **TcpStream**.

   This shows all frames of the TCP connection containing frame F.
4. Investigate: Examine earlier frames in the current TCP connection, beginning with frame F.

Use Tcptrace graph signatures

Here, you will use the Wireshark Tcptrace graph to identify signatures of two classes of performance anomalies:

- Long pause (of multi-millisecond duration) between frames, typically due to TCP timers
- Bandwidth Delay Product (BDP) issues

Refer to the appendix section "Wireshark: Tcptrace graph" for an overview.

There are three different signatures to look for: one for long pauses due to TCP timers, and two for BDP issues.

Signature 1: Long pauses due to TCP timers

To identify these pauses, inspect the Tcptrace graph of the connection. The characteristics to look for include:

- Recurring pauses of identical duration.
- TCP timer values are typically in a round number of milliseconds (for example: 1ms, 10ms, 200ms).

To use the Tcptrace graph to identify long pauses:

1. In Wireshark, go to **Statistics → TCP Stream Graphs → Time sequence** (tcptrace).
2. **Switch Direction** if necessary so that the Tcptrace graph is in the data transfer direction. See the IPX:portX → IPY:portY line at the top of the Tcptrace graph.
3. Look for long, recurring pauses of identical duration between the frames. These appear as relatively long horizontal line segments in the Tcptrace graph, as shown in Figure 1.

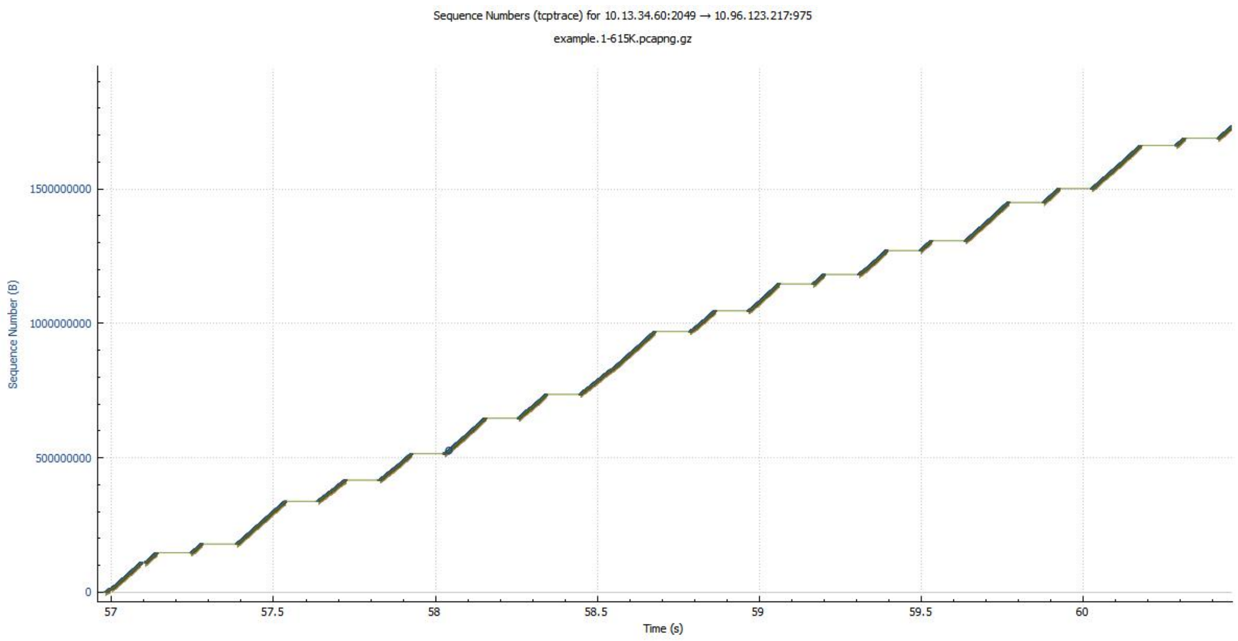

Figure 1. An example of a Tcptrace graph that indicates long pauses

Signature 2: BDP receiver bottleneck

TCP segments frequently fill up the receive window. That is, the "I"-beams come close to the upper line, as shown in Figure 2. This suggests a receiver bottleneck.

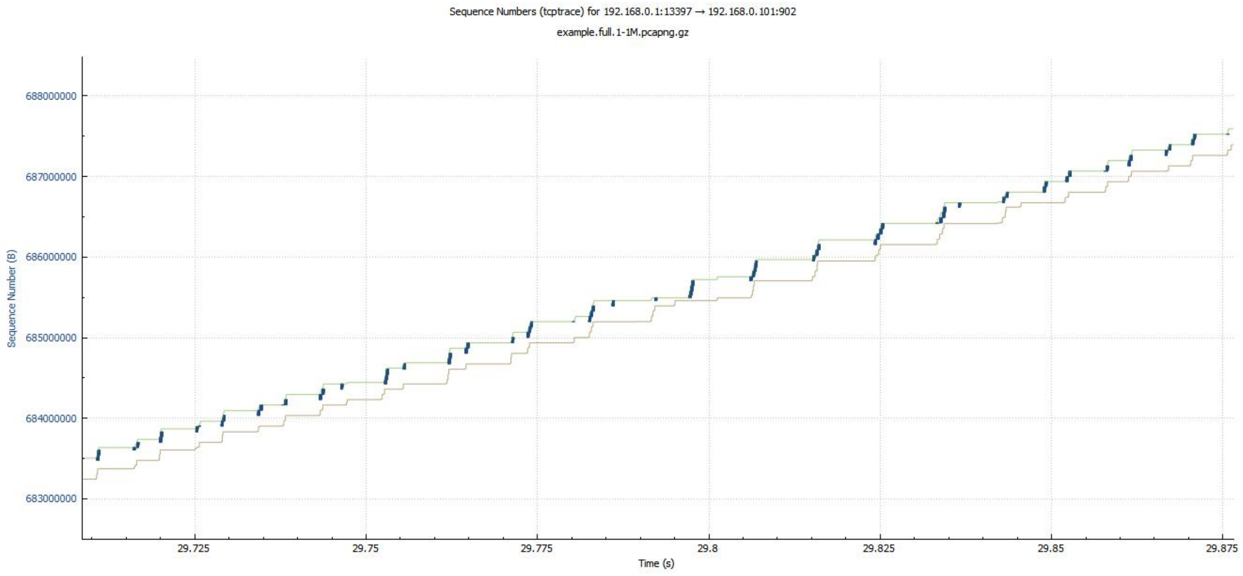

Figure 2. Example of a Bandwidth Display Product receive bottleneck graph

Signature 3: BDP sender bottleneck

TCP segments rarely fill up the receive window, (that is, the "I"-beams are nowhere close to the upper line). This suggests a sender bottleneck.

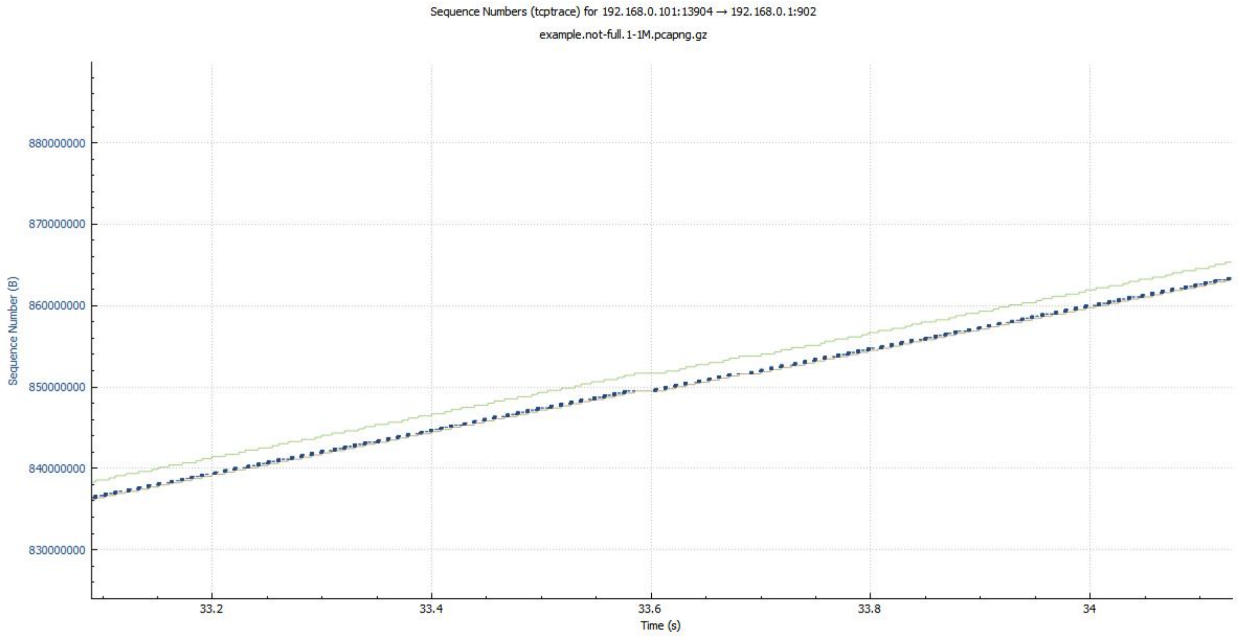

Figure 3. Example of a Bandwith Display Product sender bottleneck

To use the Tcptrace graph to identify BDP issues:

1. In Wireshark, go to **Statistics → TCP Stream Graphs → Time sequence** (tcptrace).
2. **Switch Direction** if necessary, so the Tcptrace graph is in the correct data transfer direction.
3. Determine visually if the TCP segments fill up the receive window.

Use I/O graph signatures

In this section, we use predefined I/O graphs (from the TcpTransferTput profile) to identify signatures of Bandwidth Delay Product–related performance anomalies.

> **Note:** This section is not an exhaustive treatment of BDP. In particular, the following will not be covered:
>
> BDP = Bandwidth * RoundTripTime
>
> Comparing SendBufferSize to BDP
>
> Comparing ReceiveBufferSize to BDP

Table 5 describes the predefined I/O Graphs from the TcpTransTput profile.

Graph	Metric	Direction	Color	Info
BytesInFlightMax	tcp.analysis.bytes_in_flight	S→R	Green	Bytes sent but not acknowledged
CalcWinSizeMax	tcp.window_size	R→S	Red	Scaled receive window size
TcpWinFull	tcp.analysis.window_full	S→R	Blue	Wireshark thinks sender has filled advertised window
TcpZeroWin	tcp.analysis.zero_window	R→S	Purple	Receiver announces receive window is zero

Table 5. Predefined I/O Graphs from the TcpTransTput profile

The TcpTransTput profile predefines 2 sets (X, Y) of 4 I/O graphs each, for the two potential directions of data transfer, respectively.

- Set X: BytesInFlightMaxX, CalcWinSizeMaxX, TcpWinFullX, TcpZeroWinX
- Set Y: BytesInFlightMaxY, CalcWinSizeMaxY, TcpWinFullY, TcpZeroWinY
- **Statistics → I/O Graphs**
  - Enable the 4 graphs of set X (or set Y) as appropriate
  - Interval: 1ms
  - Log scale: (Yes: easier to observe non-zero TcpWinFull* or TcpZeroWin*)
  - Automatic Update: Yes

Signature 4: Receiver may be the bottleneck if any of the following is observed:

- BytesInFlightMax is frequently close to the max of CalcWinSizeMax.
- TcpWinFull is non-zero
- TcpZeroWin is non-zero

Example A: Receiver bottleneck

Figure 4 shows an example I/O graph of linear scale with a receiver bottleneck.

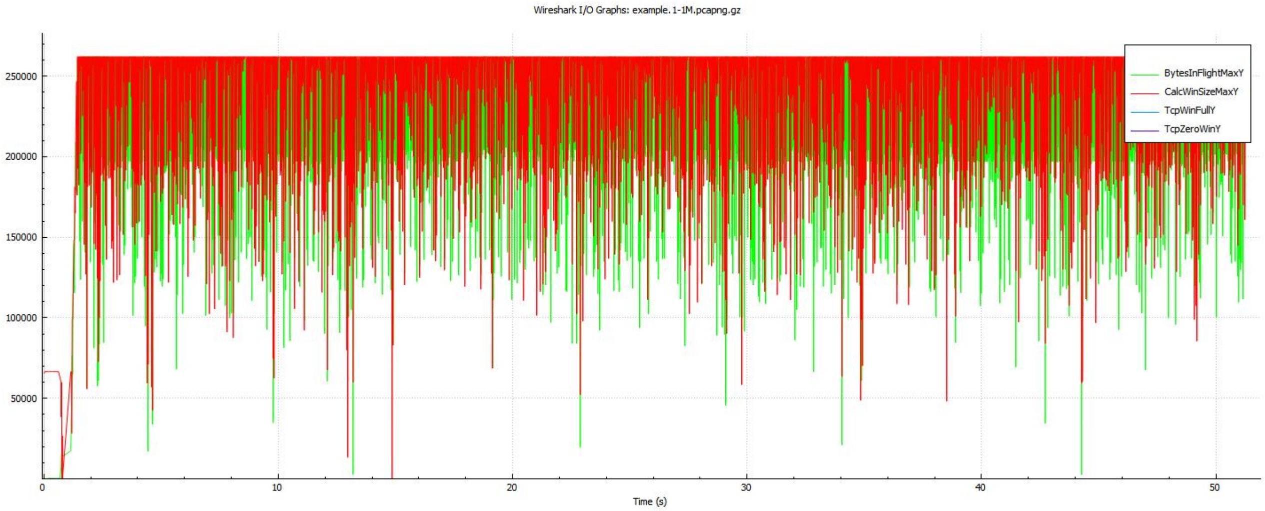

Figure 4. Example I/O graph showing receiver bottleneck (linear scale)

Figure 5 shows an example I/O graph of log scale with a receiver bottleneck. Note the non-zero values of TcpWinFull and TcpZeroWin at the bottom.

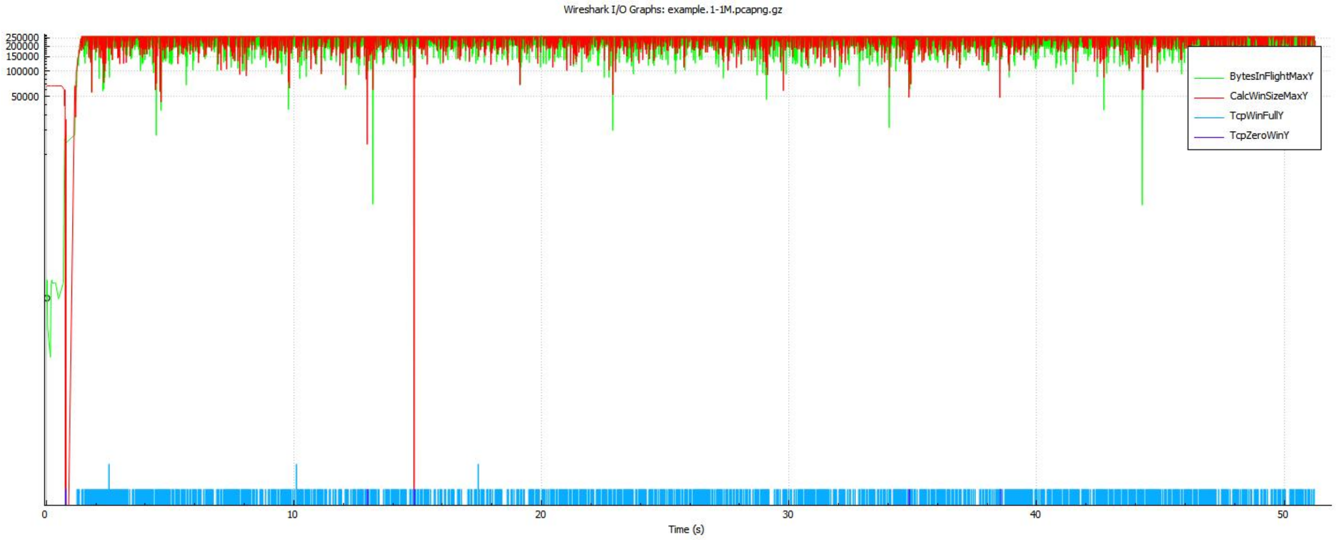

Figure 5. Example I/O graph showing receiver bottleneck (log scale)

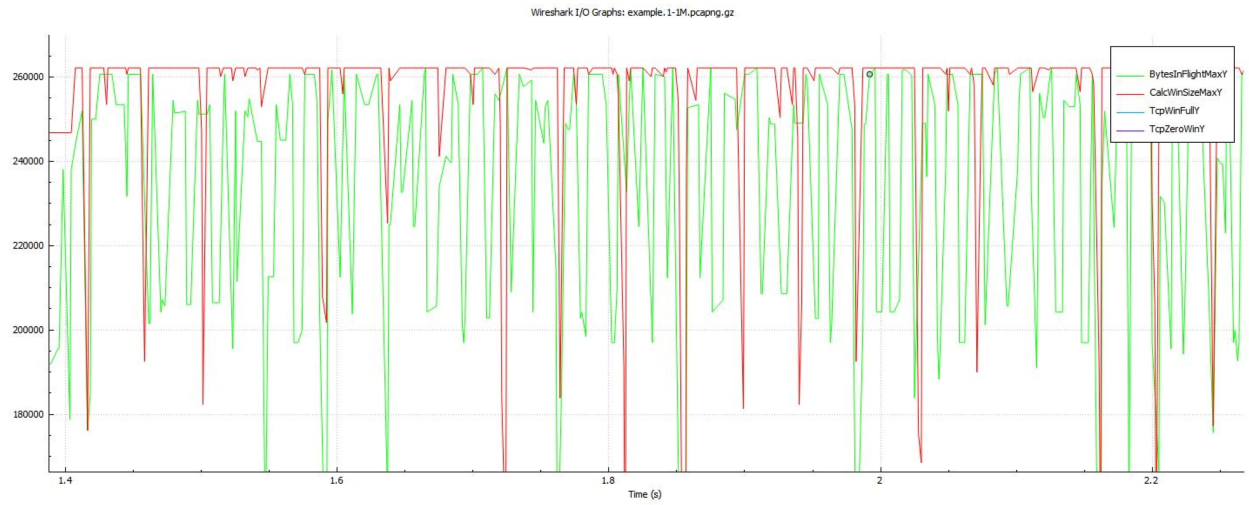

Figure 6. Example I/O graph showing receiver bottleneck (linear scale, zoomed in)

From figures 4-6, we can make the following observations:

- BytesInFlightMax is frequently very close to CalcWinSizeMax.
- TcpWinFull is frequently non-zero.

This indicates a receiver bottleneck.

Signature 5: Sender may be the bottleneck if the following is observed:

- BytesInFlightMax is always significantly lower than CalcWinSizeMax.

Example B: Sender bottleneck

Figure 7 shows an example I/O graph of linear scale with a sender bottleneck.

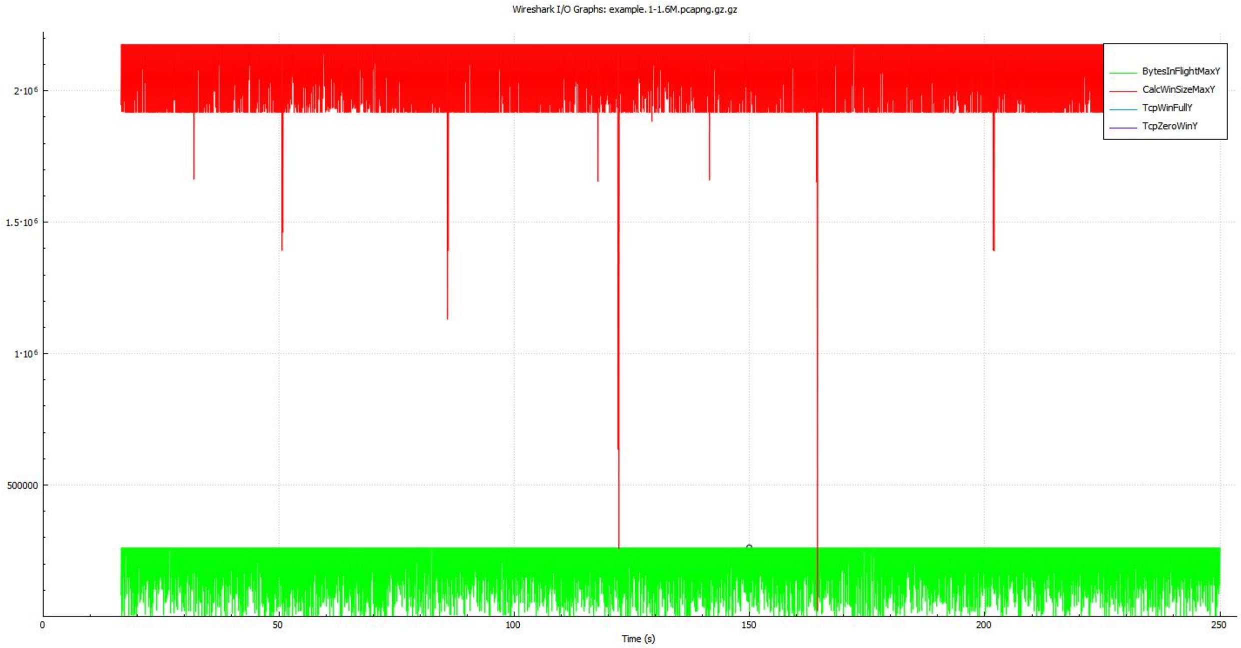

Figure 7. Example I/O graph showing sender bottleneck (linear scale)

Figure 8 shows an example I/O graph of log scale with a sender bottleneck.

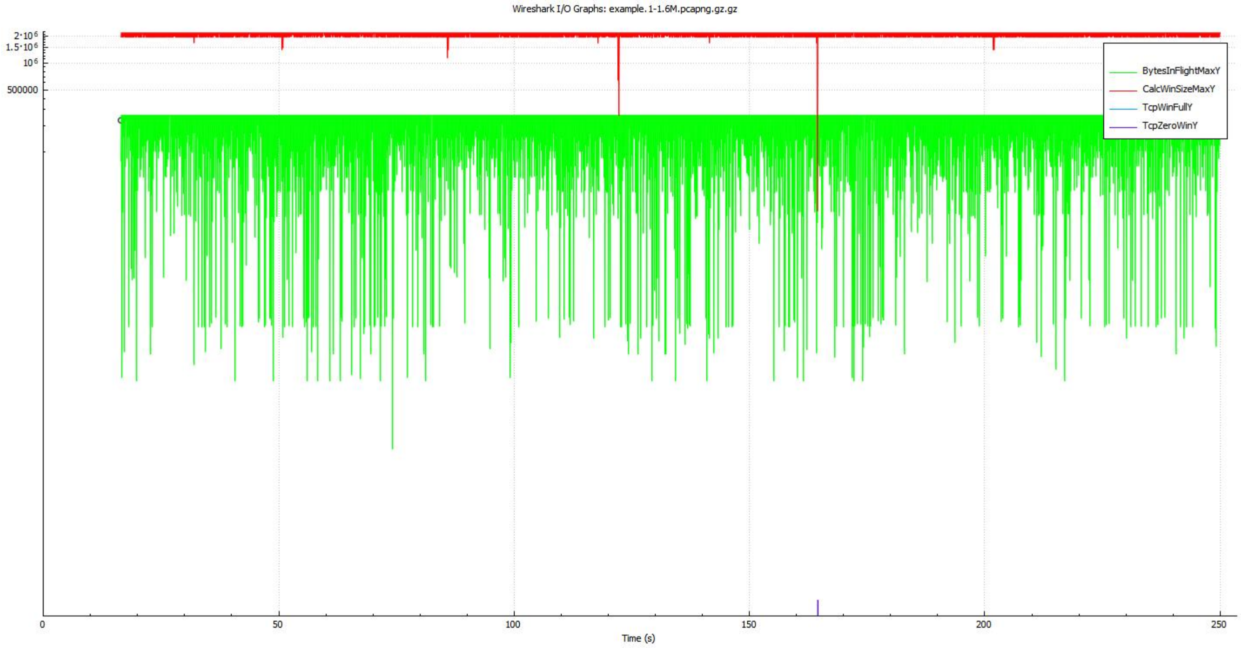

Figure 8. Example I/O graph showing sender bottleneck (log scale)

In figures 7 and 8, we can make the following observation:

- BytesInFlightMax is never close to CalcWinSizeMax

This indicates a sender bottleneck.

Conclusion

We have developed a methodology for identifying TCP issues that commonly lead to poor unidirectional data transfer throughput. We captured the network traffic of the data transfer into a packet trace file for offline analysis. We then analyzed this packet trace to detect signatures of common TCP performance anomalies that have a significant impact on transfer throughput. The TCP issues we considered include packet loss and retransmission, long pauses caused by TCP timers, and Bandwidth Delay Product (BDP) issues. We performed the analysis using Wireshark and provided a Wireshark profile to simplify the analysis workflow. This methodology offers a systematic approach to identify common TCP issues that have a significant impact on transfer throughput. We encourage engineers troubleshooting data transfer throughput performance to incorporate this methodology as a standard part of their workflow.

Appendix

Wireshark profile: TcpTransferTput

We provide the Wireshark profile TcpTransferTput, which provides predefined Display Filters and I/O graphs that help you identify TCP performance anomaly signatures that impact data transfer throughput.

To use this profile:

1. Download the profile (zip file) [4]

   https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/techpaper/performance/TcpTransferTput.zip

2. In Wireshark: **Edit → Configuration Profiles → Import → from zip file**.

3. In Wireshark: **Edit → Configuration Profiles → TcpTransferTput**

High level of using the TcpTransferTput profile:

1. Identify TCP performance anomaly signatures that impact data transfer throughput.
2. Display Filters: Predefined filters for performance anomaly signatures.
3. I/O Graphs: Predefined graphs for Bandwidth Delay Product anomaly signatures.

> **Note:** We assume that the packet trace file contains only one TCP stream.

Packet List pane: Columns

The following columns are selected to simplify various TCP analysis:

- **DeltaTime:** The time between the current frame and the previous frame in the same TCP connection.
- **DeltaTimeDisplayed:** The time between the current frame and the previous frame is displayed in this pane.
  - This is useful if some Display Filter is in use and not all frames are displayed.
  - **Example:** DisplayFilter: `(tcp.analysis.duplicate_ack_num>=3)||(tcp.analysis.fast_retransmission)`
- **Seq:** Sequence number
- **Ack:** ACK number
- **TcpLen:** Payload size
- **CalcWinSize:** Scaled receive window size
- **BytesInFlight:** Bytes sent but not acknowledged

Packet List pane: Display Filters predefined (buttons)

The following Display Filters are predefined to simplify anomaly signature identification.

Button	**Filter Expression**
TcpDupAck3+	(tcp.analysis.duplicate_ack_num >= 3)
TcpRetrans(F/S)	(tcp.analysis.retransmission || tcp.analysis.fast_retransmission)
TcpOO	(tcp.analysis.out_of_order)
Pause=1+ms	(frame.time_delta > 0.001)
Pause=10+ms	(frame.time_delta > 0.01)
Pause=100+ms	(frame.time_delta > 0.1)
TcpWinFull	(tcp.analysis.window_full)
TcpZeroWin	(tcp.analysis.zero_window)
TcpStream	(tcp.stream == ${tcp.stream})
TcpCapDrops	(tcp.analysis.lost_segment || tcp.analysis.ack_lost_segment)

Table 6. Display Filter buttons and their corresponding filter expressions

I/O graphs

I/O graphs are predefined for Bandwidth Delay Product–related anomaly signature identification.

Graph	DisplayFilter	Style	YAxis	YField
BytesInFlightMaxX	tcp.srcport < tcp.dstport)	Line	MAX(Y Field)	tcp.analysis.bytes_in_flight
CalcWinSizeMaxX	tcp.srcport >= tcp.dstport)	Line	MAX(Y Field)	tcp.window_size
TcpWinFullX	tcp.srcport < tcp.dstport)	Line	COUNT FRAMES(Y Field)	tcp.analysis.window_full
TcpZeroWinX	tcp.srcport >= tcp.dstport)	Line	COUNT FRAMES(Y Field)	tcp.analysis.zero_window
BytesInFlightMaxY	tcp.srcport >= tcp.dstport)	Line	MAX(Y Field)	tcp.analysis.bytes_in_flight
CalcWinSizeMaxY	tcp.srcport < tcp.dstport)	Line	MAX(Y Field)	tcp.window_size
TcpWinFullY	tcp.srcport >= tcp.dstport)	Line	COUNT FRAMES(Y Field)	tcp.analysis.window_full
TcpZeroWinY	tcp.srcport < tcp.dstport)	Line	COUNT FRAMES(Y Field)	tcp.analysis.zero_window

Table 7. I/O graphs predefined in TcpTransferTput profile

The TcpTransTput Wireshark profile predefines 2 sets (X, Y) of 4 I/O graphs each, for the two potential directions of data transfer respectively.

- Set X: BytesInFlightMaxX, CalcWinSizeMaxX, TcpWinFullX, TcpZeroWinX
- Set Y: BytesInFlightMaxY, CalcWinSizeMaxY, TcpWinFullY, TcpZeroWinY
- Data transfer direction (srcport => dstport)
  - Set X: srcport < dstport (for example: 789 => 2049)
  - Set Y: srcport >= dstport (for example: 2049 => 789)
- In Wireshark, go to **Statistics → I/O Graphs**
  - Enable the 4 graphs of set X (or set Y) as appropriate.
  - Interval: 1ms
  - Log scale: (Yes: easier to observe non-zero TcpWinFull* or TcpZeroWin*)
  - Automatic Update: Yes

Wireshark: Tcptrace graph

The following table shows the key features of Wireshark's Tcptrace graph.

Graph Element	Info
Vertical axis	Sequence number
Upper line (green)	Highest sequence number sender is allowed to send
Lower line (brown)	ACK number
Tiny "I"-beams between lower and upper lines (dark blue)	TCP segments
Vertical distance between lower and upper lines	Scaled receive window size

Table 8. Key features of Tcptrace graph in Wireshark

Example: Identification using Display Filter signature

In this section, we describe the detailed steps for the identification and analysis of a Display Filter anomaly signature related to a packet loss and TCP fast retransmission episode. The key steps are:

1. Use the Display Filter signature to identify TCP fast retransmission.
2. Identify key frames of interest for a packet loss and fast retransmission episode.
   - Lost segment (if found in the packet trace file)
   - The 3rd (triple) duplicate ACK (which would trigger TCP fast retransmission on the sender)
3. Compute the delta time between the triple duplicate ACK and fast retransmit, and compare the delta time with the initial Round Trip Time of the connection.

Highlights

- TCP
  - Client: 172.16.64.15:29053
  - Server: 13.32.207.59:443
- Data Transfer
  - Sender (S): 13.32.207.59:443
  - Receiver (R): 172.16.64.15:29053
- Packet capture
  - Observation Point is closer to receiver R (from TCP 3-way handshake)

Packet trace analysis (example)

Identify key frames of interest

> **Assumption:** The packet trace file contains only one TCP connection.

```
  Frame  S/R   Details        Comments

Step 1: Signature for Fast/Slow Retransmit
   Display Filter: TcpFastRetrans(F/S): (tcp.analysis.retransmission || tcp.analysis.fast_retransmission)

   1921  S->R  Seq=1572235,Len=1440 [TCP Fast Retransmission]
   3646  S->R  Seq=2964715,Len=1440 [TCP Fast Retransmission]
   5323  S->R  Seq=4637995,Len=1440 [TCP Fast Retransmission]

   # 1921 is fast retransmission

   # Action: select Frame=1921 for further analysis


Step 2: DupAck#3
   Display Filter: (tcp.ack == 1572235) && (tcp.analysis.duplicate_ack_num == 3)

   1541  R->S  Ack=1572235    [DupAck 1531#3]

   # 1541 is DupAck#3


Step 3: DupAck#1
   Display Filter: (tcp.ack == 1572235) && (tcp.analysis.duplicate_ack_num == 1)

   1534  R->S  Ack=1572235    [DupAck 1531#1]

   # 1534 is DupAck#1


Step 4: Ack
   Display Filter: (tcp.ack == 1572235) && (!tcp.analysis.duplicate_ack)

   1531  R->S  Ack=1572235    [ACK]
   1539  R->S  Ack=1572235    [ACK][TCP Window Update]
   1688  R->S  Ack=1572235    [ACK][TCP Window Update]

   # 1531 is Ack


Step 5: Lost Segment
   Display Filter: (tcp.seq == 1572235)

   1233  S->R  Seq=1572235,Len=1440
   1921  S->R  Seq=1572235,Len=1440  [TCP Fast Retransmission]

   # 1233 is lost segment
   # 1921 is Fast Retransmission

   # Note: we do not always observe the lost segment in the packet trace file.
   # Sender (S) => Observation Point (OP) => Receiver (R)
   # a. If segment was lost between S/OP: segment not in packet trace file
   # b. If segment was lost between OP/R: segment in packet trace file
```

```
Step 6: Receiver caught up
   Display Filter: (tcp.ack > 1572235)

   2115  R->S  Ack=1576555    [R caught up]

   # Lost segment ACK'd


Step 7: Summary: Key frames of interest

      1  R->S  Syn            [SYN]
      2  S->R  Syn/Ack        [SYN, ACK]
      3  R->S  Ack            [ACK]

   1233  S->R  Seq=1572235,Len=1440 [Lost segment: not arrive at R] *

   1531  R->S  Ack=1572235    [ACK]
   1534  R->S  Ack=1572235    [DupAck 1531#1]
   1541  R->S  Ack=1572235    [DupAck 1531#3] *

   1920  R->S  Ack=1572235    [DupAck 1531#30]
   1921  S->R  Seq=1572235,Len=1440 [TCP Fast Retransmission] *
   1922  R->S  Ack=1572235    [DupAck 1531#31]

   2114  R->S  Ack=1572235    [DupAck 1531#51]
   2115  R->S  Ack=1576555    [R caught up]

  # Note: Not all frames are shown.
```

Compare time from DupAck#3 to fast retransmit with iRTT

```
  Frame  Time  S/R   Details        Comments

Step 8: Initial Round Trip Time (iRTT) of TCP 3-way handshake

      1  0.000 R->S  Syn            [SYN]
      2  0.015 S->R  Syn/Ack        [SYN, ACK]
      3  0.016 R->S  Ack            [ACK]

      # iRTT: 160ms (0.016s)
      # This example:
      # + TCP client is Receiver
      # + TCP server is Sender


Step 9: Set time reference at DupAck#3 (Frame=1541)

   1541  *REF* R->S  Ack=1572235    [DupAck 1531#3]

   1920  0.018 R->S  Ack=1572235    [DupAck 1531#30]
   1921  0.019 S->R  Seq=1572235,Len=1440 [TCP Fast Retransmission]
   1922  0.019 R->S  Ack=1572235    [DupAck 1531#31]

   2114  0.024 R->S  Ack=1572235    [DupAck 1531#51]
   2115  0.024 R->S  Ack=1576555    [R caught up]

   # DeltaTime: DupAck#3 (Frame=1541)..Fast Retransmission (Frame=1921): 19ms (0.019s)
   # Important: Unset time reference after this analysis!!!


Delta time between DupAck#3 and Fast Retransmission is 19ms,
which is comparable with initial Round Trip Time of 16ms for the connection.
Frame=1921 is indeed a fast retransmission.
```

Key frames of interest: details

```
No.     Time           DeltaTime   Source        Destination    SrcPort DstPort Seq        Ack
TCPLen     CalcWinSize BytesInFlight Info

      1 0.000000000    0.000000000  172.16.64.15  13.32.207.59  29053   443     0          0          0
64240                     [SYN]
      2 0.015451449    0.015451449  13.32.207.59  172.16.64.15  443     29053   0          1          0
65535                     [SYN, ACK]
      3 0.016156258    0.000704809  172.16.64.15  13.32.207.59  29053   443     1          1          0
263424                    [ACK]

   1233 0.187764965    0.000012204  13.32.207.59  172.16.64.15  443     29053   1572235    2772
1440       68096       391680        [ACK]

   1531 0.194847383    0.000025576  172.16.64.15  13.32.207.59  29053   443     2772       1572235    0
4013056                   [ACK]
   1532 0.194856671    0.000009288  13.32.207.59  172.16.64.15  443     29053   1959595    2772
1440       68096       375840        [ACK]
   1533 0.194868882    0.000012211  13.32.207.59  172.16.64.15  443     29053   1961035    2772
1440       68096       377280
   1534 0.194869650    0.000000768  172.16.64.15  13.32.207.59  29053   443     2772       1572235    0
4013056                   [TCP Dup ACK 1531#1]
   1535 0.194881086    0.000011436  13.32.207.59  172.16.64.15  443     29053   1962475    2772
1440       68096       364320        [ACK]
   1536 0.194893282    0.000012196  13.32.207.59  172.16.64.15  443     29053   1963915    2772
1440       68096       365760        [ACK]
   1537 0.194905506    0.000012224  13.32.207.59  172.16.64.15  443     29053   1965355    2772
1440       68096       367200        [ACK]
   1538 0.194958353    0.000052847  172.16.64.15  13.32.207.59  29053   443     2772       1572235    0
4013056                   [TCP Dup ACK 1531#2]
   1539 0.194959122    0.000000769  172.16.64.15  13.32.207.59  29053   443     2772       1572235    0
4200448                   [TCP Window Update]
   1540 0.194974901    0.000015779  13.32.207.59  172.16.64.15  443     29053   1966795    2772
1440       68096       354240        [ACK]
   1541 0.194985089    0.000010188  172.16.64.15  13.32.207.59  29053   443     2772       1572235    0
4200448                   [TCP Dup ACK 1531#3]

   1920 0.213974920    0.001065628  172.16.64.15  13.32.207.59  29053   443     2772       1572235    0
4230656                   [TCP Dup ACK 1531#30]
   1921 0.214040303    0.000065383  13.32.207.59  172.16.64.15  443     29053   1572235    2772
1440       68096       478080        [TCP Fast Retransmission]
   1922 0.214116890    0.000076587  172.16.64.15  13.32.207.59  29053   443     2772       1572235    0
4230656                   [TCP Dup ACK 1531#31]

   2114 0.219340445    0.000012552  172.16.64.15  13.32.207.59  29053   443     2772       1572235    0
4230656                   [TCP Dup ACK 1531#51]
   2115 0.219375412    0.000034967  172.16.64.15  13.32.207.59  29053   443     2772       1576555    0
4230656                   [ACK]
```

Note: Packet trace file provided by www.bettydubois.com.

References

[1] Kinson Ho, "ESXi NFS Read Performance: TCP Interaction between Slow Start and Delayed Acknowledgement," May 2020. https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/techpaper/performance/esxi7-nfs-read-perf.pdf

[2] Wireshark, "Sharkfest Retrospective," 2022. https://sharkfestus.wireshark.org/retrospective

[3] Hansang Bae, "Wireshark Tutorial of TCP Nagle and Delayed Ack interaction," January 2013. https://www.youtube.com/watch?v=adDC5T-RzR4

[4] Kinson Ho, "ESX IP Storage Troubleshooting Best Practice: Packet Capture and Analysis at 10G," December 2017. https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/techpaper/performance/ESX-IP-storage-troubleshooting.pdf

[5] Kinson Ho, "Wireshark Profile: TcpTransferTput," August 2023. https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/techpaper/performance/TcpTransferTput.zip

About the Author

Kinson Ho is a staff engineer in Performance Engineering at VMware focusing on vSphere networked storage performance, including NFS, iSCSI, and vSAN File Services performance. He has extensive experience with the use of packet capture and analysis for performance troubleshooting and optimization on high-speed networks.

Acknowledgments

The author sincerely thanks Vikas Madhusudana, John McSweeney, Murali Krishnamurthy, Krishna Yenduri, and Nitish Kumar for their very thoughtful comments about the paper. A big thank you is extended to Julie Brodeur for superb technical editing that greatly improved the readability of the paper at incredibly fast turnaround times.

Special thanks are due to Wireshark super experts Sake Blok, Jasper Bongertz, and Betty Dubois for their highly insightful feedback about the performance anomaly signatures described in this paper. Betty Dubois provided the packet trace file used in one of the examples and explained some of the fine points about TCP fast retransmission.