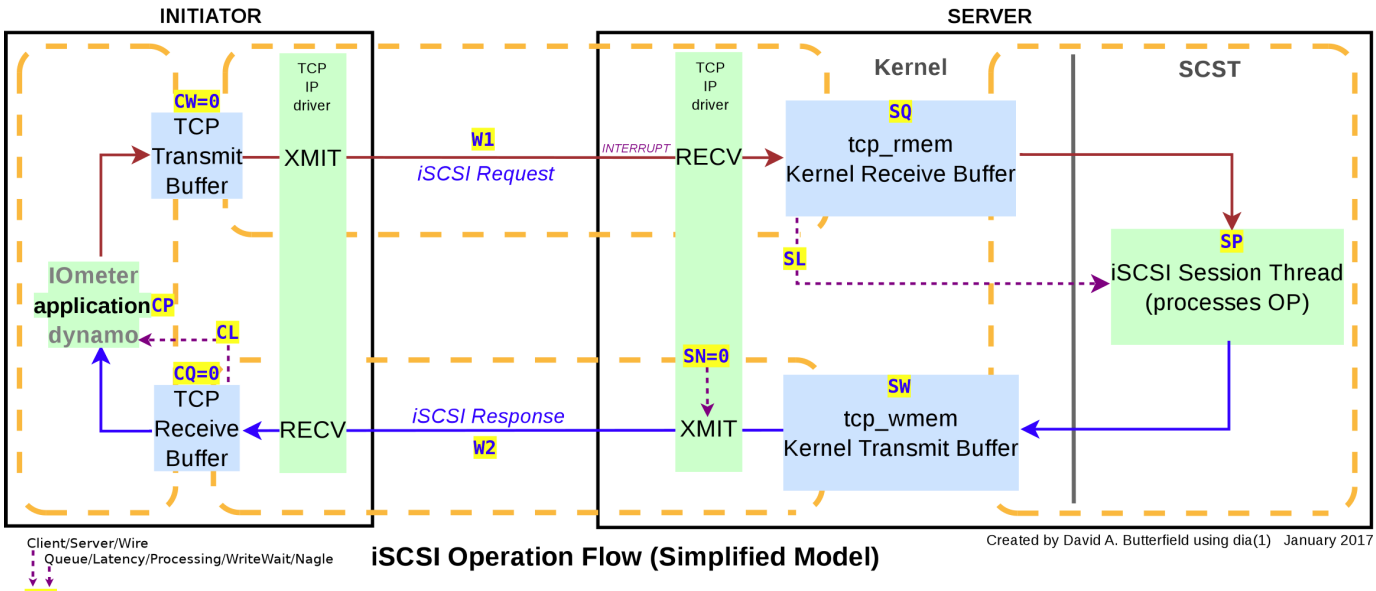
iSCSI-SCST Usermode Read Performance Model

David A. Butterfield — February 2017

We can model the system as four queues *[blue boxes]* serviced by four concurrent q-processes *[dashed orange boxes]* arranged in a loop as shown, with iSCSI operations flowing clockwise around the circuit. At any given instant there are exactly QD (Queue Depth) operations concurrently active at various stages in the pipeline from the Initiator to the Server and back. When an operation Response arrives at the Initiator it immediately issues a successor Request to the Server, thus maintaining QD outstanding operations in the pipeline.

For any given I/O size, each q-process is modeled to consume some constant amount of time { W1, SP, W2, or CP } to transition an operation from its input queue to its output queue. On the other hand, the time an operation spends waiting on those queues for service { CW, SQ, SW, or CQ } varies widely from zero up, with the average of their sum tending to rise linearly with Queue Depth.

Because the four q-processes run concurrently, above a Queue Depth of 1 the average time interval between completing operations back to the Initiator is shorter than the sum of the times each of the q-processes consumes for an operation. In fact, it is not the sum, but the *maximum* q-processing time of pipelined q-processors that bounds the minimum possible time between operations exiting the pipeline.



iSCSI Operation Flow (Simplified Model)

CW	Time interval from the Initiator application writing OP Request into TCP Transmit buffer until start of transmission
W1	Network transmission time of Request on the wire and interrupt latency until Request becomes available to the Session Thread in tcp_rmem
SQ	Time interval that an available OP Request spends waiting in tcp_rmem to be read and serviced by the Session Thread
SL	Time delay from the time Request becomes available in a previously-empty tcp_rmem until the time its sleeping Session Thread resumes executing
SP	Time for the Session Thread to process an OP on a CPU and append the Response to tcp_wmem
SW	Time interval OP Response is waiting in tcp_wmem for transmission
SN	Time delay from tcp_wmem transition to non-empty until transmission of first Response begins
W2	Network transmission time of Response on the wire and into Initiator's Receive buffer available to be read by Initiator application
CQ	Time interval that an available OP Response spends waiting in TCP Receive Buffer to be read and serviced by the Initiator application
CL	Time delay from the time Response becomes available in Client's previously-empty TCP Receive Buffer until sleeping application resumes executing
CP	Time for the Initiator application to process OP Response and write a successor OP Request into its TCP transmit buffer

Define

IOPS	Input/Output operations per second
IOSIZE	I/O operation data size (bytes of application data)
QD	Queue Depth (number of operations concurrently active)
BW	Network Bandwidth (raw maximum bytes per second)
MTU	Maximum Transmission Unit — largest network packet excluding MAC/PHY fields
MAC	Size of MAC and PHY layer overhead (PRE + SOF + MAC + CRC + GAP = 38 bytes)
TCPIP	Size of TCP and IP headers (assuming TCP timestamps, 52 bytes)
H	Size of an iSCSI protocol header, a Read Request, and a Write Response (48 bytes)
RTT	Total Round-Trip Time for an iSCSI operation (including time on queues) *[distinct from RTT measured at the TCP layer]*
RTTmin	Minimum possible RTT if operations spend no time waiting in queues (CW = SQ = SW = CQ = 0)
TPT	Throughput Time — average time interval between OP completions

Model Assumptions

- Single iSCSI session over a single (full duplex) dedicated Ethernet cable
- Session Thread processes OPs serially on one dedicated CPU
- Zero media access time
- No iSCSI socket Writer Thread is active for the session
- No OP coalescing or "autocorking"
- No IP fragmentation
- Read workload, so W1 < W2
- Workload is homogeneous; one I/O size in a test run; a series of OPs consume similarly

Additional Assumptions for Simplified Model

- Ignore TCP ACK-related issues and assume TCP_NODELAY is set, so (SN = 0)
- Requests spend no time waiting to start Initiator transmit to Server (CW = 0)

Identities

RTT = CW + W1 + SQ + SP + SW + W2 + CQ + CP
RTTmin = W1 + SP + W2 + CP *[no queue time]*
IOPS = QD / RTT
TPT = 1 / IOPS = RTT / QD

Note carefully the difference between RTT and TPT

Round-Trip Time (RTT) is the time interval it takes for the average individual iSCSI operation issued by the application to traverse the Request/Response circuit, and complete with the application receiving the Response and issuing a successor Request.

Throughput Time (TPT) is the average time interval between operation completions, corresponding to (1 / IOPS).

It should be easy to see that when Queue Depth is 1 (one Request active going around the circuit at a time), TPT = RTT.
More generally, TPT = RTT / QD, or said differently, QD = RTT / TPT It is helpful to have some intuition about this.

> "***If OPs exit a pipeline ten times as often as the time it takes an individual OP to get through that same pipeline, then there must be ten of them in there marching through at any one time.***"

Now we can start to play with some math to explore relationships between variables. For example: we know that the raw network bandwidth must put an upper bound on maximum possible IOPS at a given I/O size. Converting raw network bandwidth into equivalent IOPS and substituting our identity (QD / RTT) for IOPS:

```
    IOPS <= BW / (IOSIZE + H)              [(OPs per second) cannot exceed (max bytes per second / bytes per OP)]
QD / RTT <= BW / (IOSIZE + H)              [Substitute (QD / RT) for IOPS]
     RTT >= QD / (BW / (IOSIZE + H))
     RTT >= QD * (IOSIZE + H) / BW     and     QD < (RTT * BW) / (IOSIZE + H)
```

So a necessary condition **to keep response time RTT below an interval TI** is to set Queue Depth **QD < (TI * BW) / (IOSIZE + H)**.

Network bandwidth efficiency at the TCP layer is: (transmitted data size) / (transmitted data size + TCPIP + MAC)
The maximum transmitted data size is: (MTU - TCPIP)
So the maximum network bandwidth efficiency is: (MTU - TCPIP) / (MTU + MAC) = (MTU - 52) / (MTU + 38)

When MTU = 1500 that yields 94.15% maximum effective bandwidth.
When MTU = 9000 that yields 99.00% maximum effective bandwidth; so at MTU=9000:

We can compute a lower bound on the value of W2 for a given I/O size as: **W2 >= (IOSIZE + H) / (0.99 * BW)**
We can similarly bound W1 using the size of a Read Request (H): **W1 >= H / (0.99 * BW)** *[full MTU frame packing]*
At MTU=9000 on a 1 Gigabit network that's a lower limit of about 388 nanoseconds to transmit Read Requests
If each Request goes in its own packet, the lower limit on transmit time rises to: W1 = (H + TCPIP + MAC) / (BW)
which on a 1 Gb network is about 1.1 microseconds.

Network Bound or CPU Bound

The maximum possible IOPS corresponds to the minimum possible TPT (average time interval between OP completions). The minimum possible TPT is determined by the bottleneck in the system of queues serviced by the q-processes in the pipeline — that q-processing step taking longest to process its portion of a full round-trip operation (with other, shorter steps overlapping execution on their portions of other operations in the pipeline).

So TPTmin is given by max(W1, SP, W2, CP), which becomes max(SP, W2) (under an additional assumption that our test load Client is faster than our Server, CP < SP). Consider separately two Read I/O cases (W2 > SP), where network Response transmission time W2 dominates TPT; and (SP > W2), where OP CPU time exceeds network reply time *[small I/O sizes only, at 1 Gb]*.

(W2 > SP) Approaches Network Bound on the Response Path

[Referring to the diagram] When W2 > SP the Session Thread processes Read Requests faster than the network transmits the Responses; so it is (practically) inevitable that the Session Thread must sleep, and it will incur scheduling latency SL each time it wakes.

- OP Responses tend to accumulate in tcp_wmem awaiting transmission to Initiator
- When Response path is network bound (qdepth sufficiently high)
  - tcp_wmem (as extended by the network driver xmit buffers it supplies) is never empty
  - Reply transmit is always active, network utilization 100% in the Response direction
  - Session Thread CPU utilization averages (SP / W2)
  - Session Thread completes the next OP before transmit of the previous OP completes
  - tcp_wmem begins to fill because Session Thread adds faster than network can drain
  - TPT ≅ W2 (the longest step in the pipeline); so maximum IOPS ≅ (1 / W2)

Notice the cyclic and bursty nature of CPU usage when the session is network bound — the Session Thread runs BUSY at 100% CPU for a while, then goes IDLE for a minimum of time SL, then repeats that cycle; while the network remains fully saturated by the Server's XMIT logic drawing from the BACKLOG of Responses in tcp_wmem.

To remain network bound, there must always be Response data ready to be transmitted; so tcp_wmem must never become empty. So **before the Session Thread sleeps it must supply tcp_wmem with enough BACKLOG of Responses to keep the network busy transmitting until the Session Thread wakes up** to process another OP and replenish tcp_wmem by appending the OP's Response.

If not enough BACKLOG is supplied, the network will exhaust tcp_wmem and go IDLE before the Session Thread wakes up and appends more Responses — losing bandwidth opportunity (and network-bound status). **An *increased Session Thread scheduling latency* requires an increased BACKLOG to keep the network busy during the increased time interval that the Session Thread is IDLE.**

To remain network bound, **Queue Depth must be large enough to permit accumulation of the necessary BACKLOG. An *increased minimum Queue Depth* is required to permit stocking enough BACKLOG in tcp_wmem to cover an increased scheduling latency SL.** (Below this Queue Depth the IOPS curves are still "ramping up" toward network bound.)

Referring to the diagram: After a Response *completes* transmission to the Initiator TCP receive buffer, the time interval until its successor Request arrives in tcp_rmem is (CL + RTTmin - W2 - SP). In our case where the Session Thread has gone IDLE, after a Request arrives in tcp_rmem an additional scheduling latency SL is incurred before the Session Thread resumes processing, followed by the time SP to

process the Request and make the Response available in tcp_wmem to the Server's XMIT logic. That sums to (CL + RTTmin - W2 + SL)

To keep the network from going IDLE, all of that has to happen in less time than it takes for the network to entirely drain the BACKLOG of responses from tcp_wmem. Consider a "first" OP *completing* Response transmission, just as the Session Thread is writing its Response to the "last" OP into tcp_wmem and going to sleep, leaving tcp_wmem holding a BACKLOG of (QD - 1) Responses.

The network will take time (QD - 1) * W2 to drain the BACKLOG of Responses from tcp_wmem — to keep the network 100% busy we must ensure new Response data appends to it before then. As explained above, the Session Thread will append the Response to the successor Request of the "first" OP into tcp_wmem at time delta (CL + RTTmin - W2 + SL) after the "first" Response *completes* transmission to the Initiator. That completion time is also the starting time of the drain of the other (QD - 1) Responses from tcp_wmem. So **to stay network bound** we need to maintain the condition:

```
W2 * (QD - 1)  >=   CL + RTTmin - W2 + SL
 W2 * QD - W2  >=   CL + RTTmin - W2 + SL
      W2 * QD  >=   CL + RTTmin + SL
           QD  >=  (CL + RTTmin + SL) / W2                    [1]
```

This quantifies the idea that an increased scheduling latency SL requires an increased Queue Depth QD to stay network bound.

(SP > W2) Approaches CPU Bound on the Session Thread *(small I/O sizes only, at 1 Gb)*

When (SP > W2) the network transmits Responses faster than the Session Thread generates them; so each message completes Response transmission before another Response becomes available to transmit; so the network goes IDLE between transmitting each Response.

- OP Requests tend to accumulate in tcp_rmem awaiting Session Thread processing
- Responses are always written to an empty tcp_wmem, so SW = 0
- When Session Thread is CPU bound (qdepth sufficiently high)
  - tcp_rmem is never empty when Session Thread looks there for another OP
  - Session Thread never sleeps, CPU utilization 100%
  - Network utilization averages (W2 / SP)
  - Scheduling latency SL is never incurred (no sleeps, so no wakeups)
  - TPT ≅ SP (the longest step in the pipeline); so maximum IOPS ≅ (1 / SP)

To remain CPU bound and avoid the scheduling latency SL, tcp_rmem must always have an operation Request available to be read when the Session Thread looks there for one.

If we are CPU-bound on the Server then we can assume CP < SP (or we would be CPU-bound on the Initiator instead). Also, CQ = CL because W2 < CP < SP and SW = 0, so Initiator's TCP Receive Buffer is always empty when a new Response arrives.

Referring again to the diagram, the time for an operation to traverse the circuit from the Server's tcp_wmem, through the Initiator, and (send its successor OP) back into tcp_rmem is (CL + RTTmin - SP).

Consider a boundary case at an instant when the Server's NIC has just completed transmitting the last Response from tcp_wmem and gone IDLE, and the Session Thread completes processing the first new OP, writing it into tcp_wmem slightly too late for it to transmit before the network went IDLE. At this instant one Response, just-arrived into tcp_wmem, is about to start transmitting, and (QD - 1) Requests are queued in tcp_rmem (or will be soon, long before tcp_rmem is exhausted).

The Session Thread will take time (QD - 1) * SP to drain the BACKLOG of Requests from tcp_rmem — to keep the CPU 100% busy we must ensure that a new Request message arrives before then. That next message will be the successor Request to the Response the Session Thread just wrote into tcp_wmem, and from a paragraph above that will arrive at time (CL + RTTmin - SP) after starting to transmit the Response, which in this case occurs immediately upon being written into tcp_wmem (because SW = 0).

The time the Response was written is also the starting time of the Session Thread's drain of the remaining (QD - 1) Requests from tcp_rmem. So to keep the Session Thread from going IDLE and **remain CPU bound** we need to maintain the condition:

```
SP * (QD - 1)  >=  CL + RTTmin - SP
 SP * QD - SP  >=  CL + RTTmin - SP
      SP * QD  >=  CL + RTTmin
           QD  >= (CL + RTTmin) / SP                          [2]
```

Nagle's Algorithm (Transmission Delay SN > 0)

Now we extend the model so that it does not rely on the "additional assumptions" stipulated below the previous diagram.

John Nagle's algorithm for TCP (RFC 896) seeks to improve network efficiency by causing a TCP protocol implementation to defer sending any partial packet out a TCP connection until all of the previously-transmitted packets on that connection have been acknowledged. The delayed transmission allows multiple small outgoing messages to be combined into a single network transmission, usually improving efficiency of the network and of CPUs at both ends of the connection. But it also introduces a transmission latency which can adversely affect throughput for some types of workloads, including low queue-depth iSCSI workloads.

To avoid this latency the standard SCST implementation disables Nagle's algorithm by setting TCP_NODELAY on iSCSI sockets. Here we discuss **behavior when Nagle's algorithm is instead enabled**:

Nagle's algorithm modifies the behavior of the performance model under a CPU-bound workload from that expected when running with TCP_NODELAY set: when the Session Thread adds a small-I/O Read Response to tcp_wmem, there may be a delay SN before the network transmission interval W2 begins. The delay may persist until there is a full packet of Response messages accumulated (from multiple Response writes by the ongoing Session Thread); or the delay may end before that time due to TCP protocol operation.

When Nagle's algorithm introduces a Response transmission delay SN, that delay ***becomes part of the time interval*** through which the CPU-bound Session Thread must be able to remain busy — solely on the BACKLOG of Requests already queued in tcp_rmem — to avoid the Session Thread going IDLE (and subsequently incurring the scheduling latency).

Consider again the boundary case where the Session Thread wrote a (smaller than MTU) Response into tcp_wmem slightly too late (immediately after network transmission went IDLE): with Nagle's algorithm enabled the transmission of that Response will not begin until after the Nagle transmission delay SN has concluded.

To remain CPU bound, the Session Thread's BACKLOG of work in tcp_rmem must be large enough to keep it busy processing until another Request appears to replenish tcp_rmem, which with Nagle's algorithm enabled is (CL + RTTmin - SP + SN) after the "first" Response was written into tcp_wmem. So to remain CPU bound with Nagle's algorithm enabled the BACKLOG in tcp_rmem must satisfy

```
SP * BACKLOG  >=  CL + RTTmin - SP + SN
     BACKLOG  >= (CL + RTTmin + SN) / SP - 1                              [3]
```

When deriving [2] above, the imagined scenario positioned one OP in tcp_wmem and the remaining (QD - 1) OPs in tcp_rmem, giving BACKLOG = QD - 1. When SN is zero (Nagle's algorithm disabled) the inequality [3] reduces to the same as inequality [2], as expected.

Comparing [3] with [2]: when Nagle's algorithm is enabled on a CPU-bound workload, the minimum tcp_rmem BACKLOG to remain CPU bound **increases by (SN / SP) outstanding operations** from the minimum BACKLOG required in TCP_NODELAY mode.

When CPU bound, (SN / SP) can be no greater than the number of Responses in a full-size network packet (MTU - TCPIP) / (IOSIZE + H), about 16 for 512-byte Read operations at MTU=9000 (discussed more below).

The next table shows, for small Queue Depths 1 to 5, the observed percentage of BUSY-IDLE cycles ending having processed various numbers of operations in the cycle (ending busy_count). The workload was 512 Byte Read from /dev/zero.

At Queue Depths two and three, TCP_NODELAY (on the left) maintains the entire QD OPs together, arriving at the Server's tcp_rmem together in 100% of the cycles; whereas with Nagle's algorithm enabled (on the right) we can see at Queue Depth 2, 100% of the OPs arrive as singletons, implying that the QD is always split into two cycles of one OP each. This means there are two SL scheduling latency delays for every QD OPs processed instead of only one.

At Queue Depth three Nagle again always splits, into a group of two and a singleton. At Queue Depth five Nagle usually splits 1-4, but sometimes splits 2-3. At low Queue Depths (above 1) Nagle *always* splits the QD, with 0% of cycles reaching all the way to QD busy_count.

TCP_NODELAY also sees splits, but not as early or as strongly. The additional start latency in the Nagle pipeline makes it easier for the QD OPs in the circuit to split, with some ops getting caught behind the Nagle start latency and not able to make it back around the circuit to tcp_rmem in time before the Server thread sleeps.

% of Cycles Ending With		**TCP_NODELAY**					**NAGLE**				
Busy Count -->		**1**	**2**	**3**	**4**	**5**	**1**	**2**	**3**	**4**	**5**
Queue	**1**	**100%**					**100%**				
Depth	**2**	**0%**	**100%**				**100%**	**0%**			
	3	**0%**	**0%**	**100%**			**50%**	**50%**	**0%**		
	4	**8%**	**8%**	**8%**	**76%**		**30%**	**40%**	**30%**	**0%**	
	5	**15%**	**12%**	**12%**	**15%**	**46%**	**43%**	**7%**	**7%**	**43%**	**0%**

What's directly relevant for avoiding the Server thread sleeping is the BACKLOG in tcp_rmem; the Queue Depth is an upper bound on the BACKLOG, but the BACKLOG is lower than that when the QD OPs are split into different queues waiting for start latencies to expire. So the increase in Queue Depth needed to compensate for adding the Nagle delay into the pipeline is actually more than the (SN / SP) expected from comparing [3] and [2] above.

The unfortunate interactions between the Nagle delay and scheduling latency tend to reduce IOPS by nearly half (as compared to running with TCP_NODELAY set) until Queue Depth becomes high enough to make TCP_NODELAY CPU bound. Then the TCP_NODELAY IOPS no longer increases with Queue Depth (IOPS curve flattens). However the Nagle-mode IOPS continues to increase with Queue Depth beyond that, eventually reaching the crossover point where 100% Nagle-mode performance becomes superior to 100% TCP_NODELAY performance, and at small I/O sizes continuing on to a peak IOPS significantly higher than the TCP_NODELAY peak.

Notice that whenever Nagle's algorithm is active, CPU-bound workloads see a cyclic / bursty effect "dual" to the one observed with network-bound loads; but here the CPU usage remains fully saturated and it is network transmission that toggles between 0% and 100%.

More Complexity (CW > 0)

The situation is still quite a bit more complicated than all that due to multiple factors, the first-order ones being: *(1)* knowing when a Nagle Delay will end; and *(2)* the fact that Nagle's algorithm affects not only the Server's Response messages, but also the Initiator's Request messages.

However when Queue Depth is large enough modeling these effects can be much simplified. When the Response path is *network bound* the Nagle delay on the Server side has no impact because those scenarios have an ample supply of data in tcp_wmem awaiting transmission, so there is no "partial packet" to delay for, and Nagle delay SN is never incurred, effectively zero.

For a *CPU-bound* workload: Recalling that a Nagle delay ends when a queued partial packet fills to the MTU size, an upper bound on SN is the time it takes the Session Thread to write enough Responses to finish filling a partial packet after the first Response has been written into it. When CPU bound that would be (per-OP processing time) * (the number of additional Responses needed to finish filling the packet), which after the first one gives SN <= SP * ((MTU - TCPIP) / (IOSIZE + H) - 1)

The Server's Nagle delay SN could be shorter than that if the ACK for the previous reply packet arrives before the Session Thread processes enough OPs to entirely fill the outgoing MTU size; in such cases the partial packet would be allowed out to the network short of the full MTU size.

For this to be possible requires that SP * (MTU - TCPIP) / (IOSIZE + H) > CL + RTTmin - SP so that the Session Thread's time to fill a whole packet exceeds the time for the previous Response's ACK to arrive (piggyback on the successor Request).

```
SP *  (MTU - TCPIP) / (IOSIZE + H)      >  CL + RTTmin - SP
SP * ((MTU - TCPIP) / (IOSIZE + H) + 1) >  CL + RTTmin
                                      SP > (CL + RTTmin) / ((MTU - TCPIP) / (IOSIZE + H) + 1)
```

In this case the reduced upper bound for the Nagle delay is SN <= CL + RTTmin - 2*SP. (The second subtracted SP accounts for the interval between writing the Response that fills the Response packet and ultimately triggers the ACK, and writing the following response which marks the start of the Nagle delay period SN.)

Quantum Mechanics

Accurate understanding of Read behavior at Queue Depths smaller than (MTU - TCPIP) / (IOSIZE + H) — the number of Responses that can fit in a network packet — requires a more detailed "counting" analysis following the movement and detailed timing of individual Requests, Replies, and Acknowledgements through the components seen in the OP flow diagram below; as opposed to the more "probabilistic" approach relying on averages used in the analysis above. It is at this "small" level that the details of acknowledgement timing, elaborated in the next few paragraphs, become essential to the analysis.

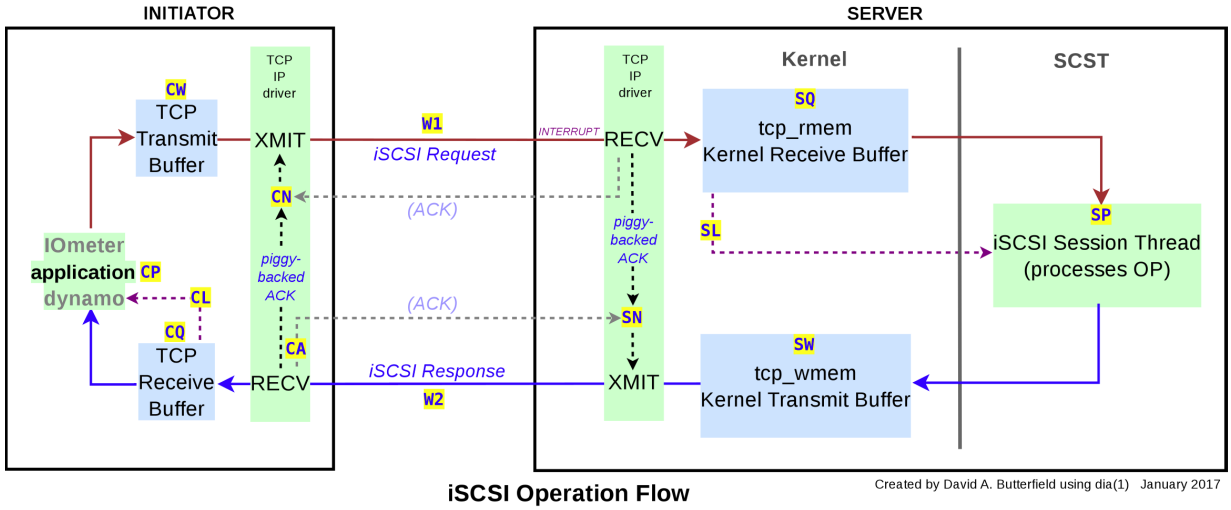
A Nagle delay can be ended by an arriving TCP protocol acknowledgement (ACK) of received data; so to use this model predictively it is helpful to have some understanding of how TCP decides when to send one.

A TCP connection has two connection partners, one at each endpoint, which communicate with each other through the connection. Each connection represents two, logically-independent streams of data bytes between the connection partners, one stream in each direction.

Each stream's data is partitioned into discrete packets; each packet is individually transported across the network as a unit. (The partitioning can occur at multiple layers of protocol; here it suffices to refer to them all together as "packets", omitting more precise specification of "segments", "frames", etc.) Each packet comprises protocol header information and (usually) data being transported.

Every data packet in each of these (unidirectional) TCP streams is identified by a sequence number in its TCP header, representing the byte number (in its stream) of the first byte of data in the packet. An acknowledgement of received data is communicated by returning to the sender of a stream the sequence number of the first UNreceived byte in the stream. Under this scheme TCP ACKs are cumulative, and multiple received packets can be acknowledged using a single ACK.

The ACK for a TCP data packet that has arrived through a connection travels back through the connection in a special field in the header of a regular TCP data packet going in the opposite direction. Every transmitted data packet carries an ACK for its opposing stream with it in this special field. If a data sender has received no additional data since its last transmission, the value of the transmitted ACK field will be the same as in its previous transmission — again, the (unchanged in that case) number of the first unreceived byte in the stream.



iSCSI Operation Flow

CW	Time interval from the Initiator application writing OP Request into TCP Transmit buffer until start of transmission
CN	Time delay from Initiator Transmit Buffer transition to non-empty until transmission of first Request begins
W1	Network transmission time of Request on the wire and interrupt latency until Request becomes available to the Session Thread in tcp_rmem
SQ	Time interval that an available OP Request spends waiting in tcp_rmem to be read and serviced by the Session Thread
SL	Time delay from the time Request becomes available in a previously-empty tcp_rmem until the time its sleeping Session Thread resumes executing
SP	Time for the Session Thread to process an OP on a CPU and append the Response to tcp_wmem
SW	Time interval OP Response is waiting in tcp_wmem for transmission
SN	Time delay from tcp_wmem transition to non-empty until transmission of first Response begins
W2	Network transmission time of Response on the wire and into Initiator's Receive buffer available to be read by Initiator application
CA	Delayed ACK latency -- time from when an ACK becomes owed until the time it is transmitted
CQ	Time interval that an available OP Response spends waiting in TCP Receive Buffer to be read and serviced by the Initiator application
CL	Time delay from the time Response becomes available in Initiator's prreviously-empty TCP Receive Buffer until sleeping application resumes executing
CP	Time for the Initiator application to process OP Response and write a successor OP Request into its TCP transmit buffer

An ACK becomes owed when a data receiver successfully receives its next expected TCP packet, so that the sequence number of its first unreceived byte has changed. Its next packet to the sender through the connection will deliver the owed ACK. After an ACK becomes owed, the TCP protocol specification allows an implementation to delay transmission of the ACK for up to 500 milliseconds, holding on to it for a "short" time in hopes of sending it more efficiently than in a network packet all by itself (Linux uses 40 ms). An ACK must also be sent (sooner than that) whenever two full packets worth of data have arrived on the connection since the last ACK was sent. (Linux appears to send an ACK once for every two incoming packets, regardless of their size.)

If any data is transmitted across the connection while an ACK is still owed, the TCP header encapsulating the data will carry the ACK with it, and that ACK will no longer be owed. In this (preferred) case, sending the ACK costs no additional network bandwidth.

If no data goes that direction through the TCP connection before an owed ACK must be sent, the TCP implementation constructs a zero-length data packet to carry the ACK across the connection to the connection partner. In this case there is some network cost to send the extra packet (though it is going out over a connection having no data traffic). There can still be advantage to some delay in this case, because additional ACKs might become owed in the meantime, and all be collapsed into a single message specifying the highest one.

Notice that Nagle's algorithm can affect ACK timing, because ACKs travel on regular data packets and Nagle's algorithm can increase the average time interval between those. So Nagle's algorithm and ACK timing each affects the other — more complexity.

In this iSCSI Operation Flow diagram, "*piggybacked ACK*" refers to the (preferred) case where an ACK "rides piggyback" in the header of a nonempty data packet in the opposing stream that was being transmitted anyway; "*(ACK)*" refers to the case where a zero-length data message was specially sent for the purpose of carrying the acknowledgement.

Pathology

Some non-optimal socket usage is recognizable by extremely low throughput when Queue Depth is less than the number of Responses required to fill a network packet up to its MTU size. This occurs by bringing delayed acknowledgements into the mix of unfortunate interaction with Nagle's algorithm. With the Linux Initiator the ACK delay timeout is 40 milliseconds, so if this low-Queue-Depth interaction is happening anywhere in the circuit, IOPS cannot exceed (25 * QD).

This type of behavior should not occur with well-considered socket usage. Specifically usermode SCST avoids such behavior by:

- Whenever the Session Thread is going IDLE with Nagle's algorithm enabled, it flushes any pending reply data out to the network.
- A socket write of a fragment of an iSCSI Response message specifies the MSG_MORE flag unless it is the final fragment of the Response message, in which case it does not.

The MSG_MORE flag on a socket write tells the kernel that "*the caller has more data to send*", with the implicit assumption that this data will be written in a *timely* way. The kernel can use this information to optimize network and CPU efficiency by waiting for the additional data to be written and combining it with the data from the MSG_MORE write into a single network transmission. The assumption is that holding back transmission of the bytes written with MSG_MORE will not introduce *undue* latency for the application because the additional data will be written in a *timely* way. (Since each application controls its own use of MSG_MORE, it can implement its own tradeoff between the notions of *timely* and *undue*.)

When Nagle's algorithm is enabled, proper use of MSG_MORE (or another suitable solution) can be far more important than a simple linear efficiency gain. Suppose the Server writes the Response for a 512-Byte iSCSI Read using a pair of send(2) system calls, one for the 48-Byte iSCSI header and one for the 512 data bytes. Consider the simplest case where Queue Depth is 1.

In the (very likely) event that the Server has no ACKs outstanding, **without** the MSG_MORE flag the iSCSI header (written in the first send call) will begin transmission immediately upon being written; but the data that goes with it (written in the second send call very shortly after the first) gets stuck behind the Nagle delay because it must wait for the ACK of the earlier packet that carried the iSCSI header, before Nagle's algorithm will allow the second (partial) packet out with the 512 bytes of data.

Meanwhile the Initiator receives the 48 bytes of iSCSI Response header, but this is not enough to process the Response because it also needs the data; so no operation completes, so no successor Request is issued, so no data packet travels from the Initiator to the Server to carry the ACK for the header packet (piggyback), so the Server cannot transmit the packet containing the data.

This temporary deadlock is broken by the (40 millisecond) delayed ACK timer expiring on the Initiator — at that time the ACK for the iSCSI header packet is transmitted to the Server, which can then transmit the data portion of the Response. This Queue Depth 1 scenario has RTT over 40 milliseconds; so according to our identity we have IOPS = QD / RTT = 1/(40ms) = 25 IOPS at maximum. (This would be two IOPS with the maximum allowable ACK delay of 500 ms.)

Using MSG_MORE when writing any but the final fragment of a Response message avoids any partial packet transmissions until the final (non-MSG_MORE) fragment is written. That ensures at least one full Response arrives at the Initiator to generate a successor Request to carry the awaited ACK piggyback to the Server, without having to wait for the 40 millisecond timer to expire.

When Queue Depth is larger than the number of Responses that can fit in a packet this problem does not occur, because the Server can process enough operations to fill the first reply packet and can send it without having to wait for an ACK; and the arrival of that packet results in the Initiator issuing successor Requests which can carry the ACK back to the Server before it *does* become needed (for TCP windowing).

> (Although the most severe manifestation of the problem disappears with Queue Depth greater than the number of Responses in a full packet, I think it takes somewhat more than that number to fully avoid the problem because of the splitting of the Queue Depth into separate BACKLOGS by Nagle's algorithm and the scheduling latency.)

When using MSG_MORE, careful attention should be given to proper use. The MSG_MORE flag should be treated as a "promise" rather than a "hint", because if the application fails to follow through with more activity to the socket, the data written with MSG_MORE can be held back from transmission for a long time, up to 200 milliseconds, before finally being sent. (The same thing can happen with misuse of TCP_CORK.) The kernel waits for the expected additional data so that it can combine it with what it was already given. This condition is *not* unwedged by an arriving ACK; it is distinct from the Nagle mechanism. In such a case throughput can reduce to 1/(200ms) = 5 IOPS.

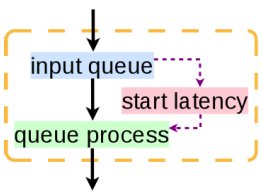
Abstracted Pipeline Stages

Before adding any more detail to the pipeline model, let's step back and look at what we already have, observing similarities between the pipeline stages. The hope is to clarify thinking by abstracting some implementation details into their essential identities and behaviors.

The Client/Server operation pipeline can be modeled as a ring of connected stages, each stage of the pipeline having an input queue and a queue process, as shown, with operations flowing through the pipeline in the direction of the solid arrows. At any particular moment a queue process *[green box]* may be BUSY or IDLE.

When a queue process is BUSY, it runs continuously in a loop, removing operations from its input queue one-by-one in sequence, processing them, and appending the results to the input queue of the next stage in the pipeline. When a queue process (having exhausted its input queue) finds no more operations awaiting processing, it becomes IDLE. When a new operation arrives at the empty input queue of an IDLE queue process, the queue process is transitioned back to BUSY state and resumes processing operations.

There is a time delay from when the first new operation arrives into an empty input queue until the time its IDLE queue process returns to BUSY state and resumes processing. This delay applies only when the queue process is IDLE and the queue is empty. Once the queue process is BUSY it processes each operation in turn without further delay, until it looks for more work in its input queue and finds none.



Each stage of the pipeline is characterized, for any particular workload, by two parameters:

(1) The time it takes the queue process to remove one operation from its input queue, process it, and append the result to the input queue of the next stage of the pipeline *[green box]*.

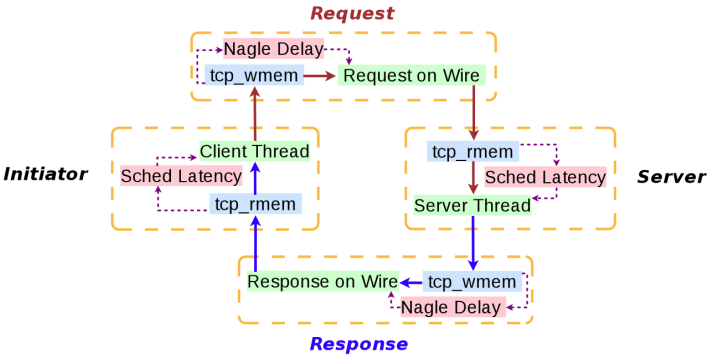
(2) When a queue process is IDLE, the time delay between the first operation arriving to its (empty) input queue and the queue process starting to process that first operation *[pink box]*.

Time spent on the input queue *[blue box]* is not a parameter; it is determined by global system behavior, strongly influenced by the local stage start latency.

(1) represents the minimum time necessary for an operation to process through the stage of the pipeline — the transit time through the stage if no time is spent waiting on the input queue (i.e. zero time elapses between the OP becoming available in the input queue and the queue process starting to process the OP). This is modeled as constant from OP to OP for a given I/O size and hardware configuration. The *sum* of all of these (green boxes) in the pipeline represents a lower bound on the Round-Trip Time for an operation (RTTmin). The maximum IOPS attainable by the system is the reciprocal of the *maximum* time consumed by any *one* of the green boxes in the pipeline.

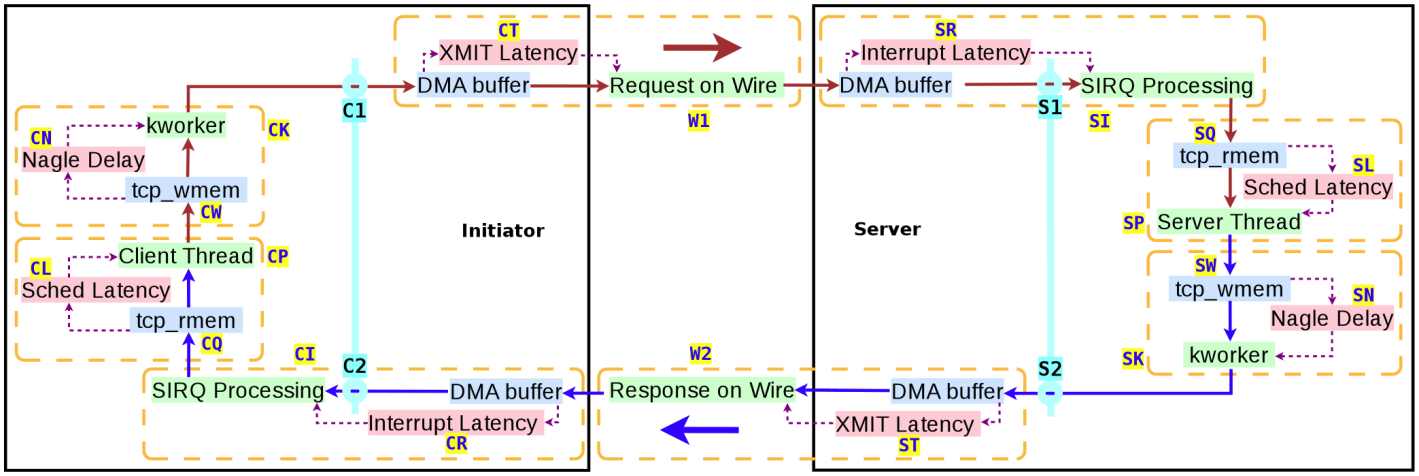
(2) represents "start delays" incurred when a queue process returns to BUSY after an IDLE period. It is important to realize that these start latencies do not affect the peak IOPS attainable by the system — what they affect is how large the queue depth must be to *reach* the peak IOPS.

The performance model described earlier can be viewed as four of these pipeline stages arranged in a ring, as shown here. This depiction removes most details of implementation modules and focuses on their similar essential queuing behavior.



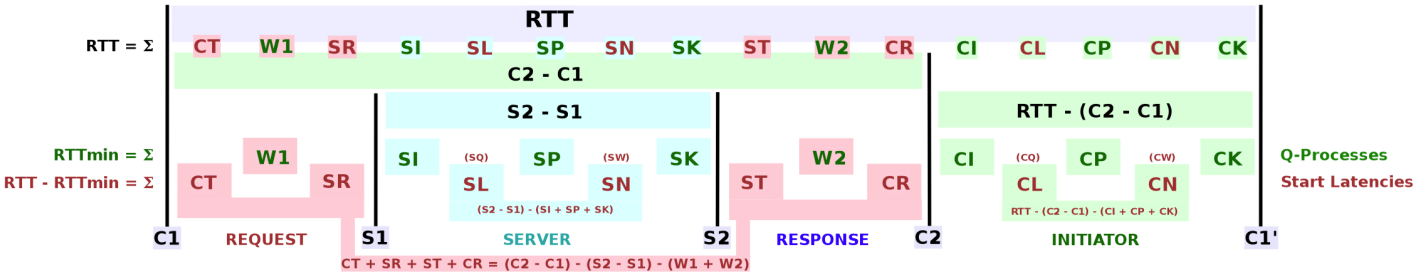
A more accurate model appears below. This extends the number of modeled queues and processes from four to eight, taking interrupt and transmission latencies into account in both directions. This model enables accurate latency and Queue Depth calculations; and independent CPU assignments for the SIRQ handler and the Server Thread. The two vertical cyan lines indicate **tcpdump timestamp capture points.**

When the Server Thread is pinned to the same CPU as its network SIRQ handler, the times taken by those two stages should be added together and treated as one stage (because they run serially). The corresponding CPU-bound limit on IOPS when they are pinned to the same CPU is 1 / (SI + SP) *[green, diagram upper right]*.



The vertical cyan timestamp-capture lines divide the iSCSI operation Round-Trip Time RTT into three disjoint subsets: Server processing (S2 - S1); Client processing (RTT - (C2 - C1)); and communication between the Server and the Client, in both directions combined ((C2 - C1) - (S2 - S1)).

This diagram breaks down Round-Trip Time (RTT) into segments, shown as equivalent to differences between tcpdump timestamps captured as each OP passes through points C1, S1, S2, and C2.



To further break down RTT of a CPU-bound workload into smaller segments, we combine measurements taken at a CPU-bound Queue Depth with measurements of the same I/O size at Queue Depth 1. (This also works for network-bound workloads.)

Queue Depth 1 is relatively simple to analyze because every applicable start latency is incurred every time passing through every stage of the pipeline (because a stage is always IDLE when the OP arrives, given only one OP in the circuit). On the other hand, there are never any delays due to other OPs queued ahead. So when QD=1 we know how much time is spent waiting on each queue:

```
SQ = SL;        SW = SN;        CQ = CL;        CW = CN;
```

and the measured Round Trip Time at QD=1 is exactly the sum of RTTmin and all the applicable latencies. (When QD=1 the CN and SN Nagle latencies are inapplicable because both sides are always up-to-date with ACKs when data becomes ready to transmit.)

We observe IOmeter IOPS and use a top(1) display to observe and record separate measurements of the %CPU usage of each of the various threads and SIRQ handlers during the execution of a CPU-bound workload. The %CPU observations are easily converted into CPU-microseconds per operation (CPU fraction divided by IOPS), giving values for SI, SP, SK, CI, CP, and CK.

It has been observed and we assume that the average time consumed for an operation by the queue-processing step in each stage of the pipeline remains constant for a given I/O size, independent of Queue Depth. So we can apply the per-OP CPU-time results from the CPU bound run in the analysis of the latencies of the Round-Trip Time in the QD=1 run. (The CPU numbers at QD=1 are too small to measure precisely enough in the noise of the rest of the system.)

The values for W1 and W2 can be computed based on the network bandwidth, MTU size, and I/O size, as described earlier. Those along with the six values for CPU usage from the CPU-bound run give us enough to calculate RTTmin (their sum), and TPTmin (their maximum). These queue-processing times are represented in the above diagram by the eight components in green font in the "RTTmin/Q-Processes" row. The calculated TPTmin should approximate the reciprocal of the IOPS observed during the run being analyzed.

From the breakdown of RTT shown above it is apparent that when running at Queue Depth 1, the sum of all the start latencies is (RTT - RTTmin). Subtracting the CPU-bound RTTmin from the QD=1 RTT gives us the sum of all the QD=1 start latencies:

```
(RTT - RTTmin) = (CT + SR) + (SL + SN) + (ST + CR) + (CL + CN)
```

Based on the breakdown table we can calculate some subsets of these latencies by combining values measured at the CPU-bound Queue Depth and at Queue Depth 1:

```
(CL + CN) = RTT - (C2 - C1) - (CI + CP + CK)
(SL + SN) =       (S2 - S1) - (SI + SP + SK)
```

At Queue Depth 1 we can assume that CN is zero because the Initiator is always up-to-date on ACKs when it wants to send a Request message — an ACK always arrives to the Initiator carried piggyback on the reply packet that resulted in the issuance of the new Request. The Server sees a similar experience, so also SN = 0 when QD=1. So when QD=1 we have

```
CL = RTT - (C2 - C1) - (CI + CP + CK)
SL =       (S2 - S1) - (SI + SP + SK)
```

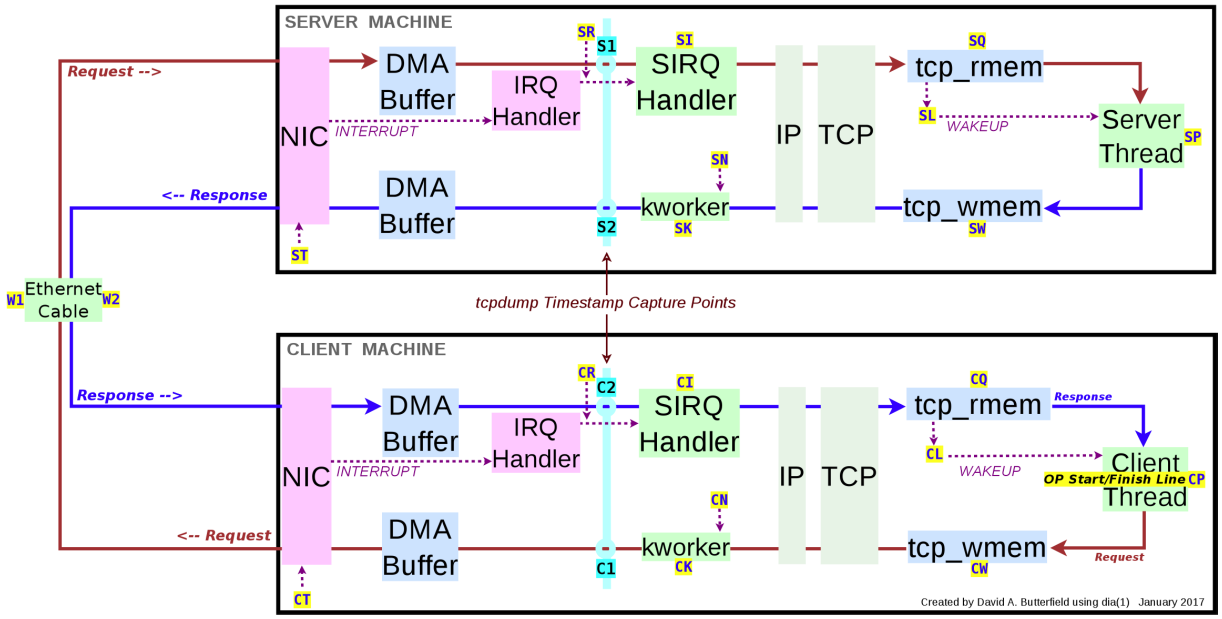
From the RTT breakdown table we can calculate the combined transmission and receive latencies as

```
(CT + SR) + (ST + CR) = (C2 - C1) - (S2 - S1) - (W1 + W2)
```

Note that we cannot compute (CT + SR) separately from (ST + CR) because we cannot validly compare Server timestamps S1 or S2 with Client timestamps C1 or C2. Because the clocks on the Client and the Server are not synchronized, Server timestamps can only be compared with other Server timestamps (and likewise for Client timestamps).

Now back to the implementation view with the new pipeline stages added. This diagram is specific to Linux Initiators, since it depicts the Linux networking stack in the Client machine. (The Client machine has been relocated under the Server to fit better into the diagram space.)

Network Client-Server Connection -- Breakdown of Operation Round-Trip Time



Notes:

- The diagram is not really specific to SCST or even to iSCSI, but applies more generally to network Client/Server models.
- Receive DMA Buffer may physically coincide with tcp_rmem, but data is not available to the receiving thread until SIRQ handler runs.
- Time intervals are calculated from timestamp data collected by tcpdump each time an operation crosses a capture point (vertical cyan lines).
- Ideally tcpdump timestamps would reflect when packets arrived or departed on the wire itself, but they actually span a wider interval of time whose precise limits are not clear and probably varies by driver. I've drawn the (vertical cyan) lines at "assumed" positions at the time the SIRQ handler begins to process an incoming packet.

Revised Identities

```
            Round-Trip Time RTT    = CT + W1 + SR + SI + SQ + SP + SW + SK + ST + W2 + CR + CI + CQ + CP + CW + CK
      Minimum Round-Trip Time RTTmin =    + W1      + SI      + SP      + SK      + W2      + CI      + CP      + CK

Network interrupt on different CPU from Server Thread:  Minimum Throughput Time TPTmin = max(W1,  SI,  SP,  SK,  W2,  CI,  CP,  CK)
Network interrupt on the same CPU as Server Thread:     Minimum Throughput Time TPTmin = max(W1,  SI + SP,  SK,  W2,  CI,  CP,  CK)
Uniprocessor Server (kworker on same CPU also):         Minimum Throughput Time TPTmin = max(W1,  SI + SP + SK,  W2,  CI,  CP,  CK)
```

OP Start/Finish Line

CW Time interval from the Client Thread writing OP Request into tcp_wmem Transmit buffer until kworker begins to process it for transmission
CN Time delay from tcp_wmem transition to non-empty until kworker begins to process the first Request out of it
CK kworker processing time from tcp_wmem until OP Request crosses Client's tcpdump Transmit capture point

CT Client machine latency to transition from Transmit-IDLE to begin a series of one or more transmissions
W1 Network transmission time of Request on the wire DMA Buffer to DMA Buffer (calculated from bandwidth and packet overhead)

SR Server machine latency between receiving a Request from the wire and reaching Server's Receive capture point (including interrupt latency)
SI Time the Server software interrupt handler processes Request from Receive capture point until available in tcp_rmem to be read by Server Thread

SQ Time interval that an available OP Request spends waiting in tcp_rmem to be read and serviced by the Server Thread
SL Time delay from the time Request becomes available in a previously-empty tcp_rmem until the time its sleeping Server Thread resumes executing
SP Time for the Server Thread to process an OP on a CPU and append the Response to tcp_wmem

SW Time interval OP Response is waiting in tcp_wmem for kworker to begin processing it for transmission
SN Time delay from tcp_wmem transition to non-empty until kworker begins to process the first Response out of it
SK kworker processing time from tcp_wmem until OP Response crosses Server's tcpdump Transmit capture point

ST Server machine latency to transition from Transmit-IDLE to begin a series of one or more transmissions
W2 Network transmission time of Response on the wire DMA buffer to DMA buffer (calculated from bandwidth and packet overhead)

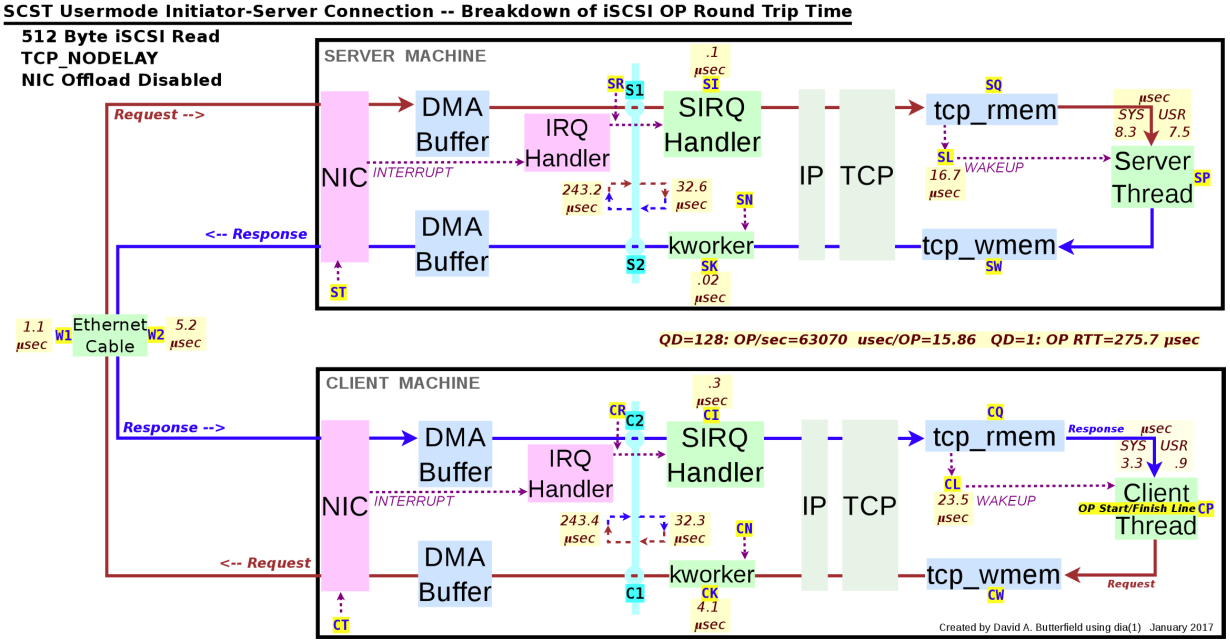
CR Client machine latency between receiving a Response from the wire and reaching Client's Receive capture point (including interrupt latency)
CI Time the Client software interrupt handler processes Response from Receive capture point until available in tcp_rmem to be read by Client Thread

CQ Time interval that an available OP Response spends waiting in tcp_rmem to be read and serviced by the Client Thread
CL Time delay from the time Response becomes available in a previously-empty tcp_rmem until Client Thread resumes executing
CP Time for the Client to process OP Response and write a successor OP Request into the Client's TCP transmit buffer

OP Start/Finish Line

Next is the same diagram again with numbers filled in for 512 Byte iSCSI Reads from /dev/zero with TCP_NODELAY set.

SCST Usermode Initiator-Server Connection -- Breakdown of iSCSI OP Round Trip Time



Notes:

- Measurements taken with Server sockets set to TCP_NODELAY.
- NIC segment offload features disabled, so network packets directly reflect I/O sizes issued by the application layers.
- Workload is 512 Byte Random Read from /dev/zero using a single Initiator over a single 1 Gb connection.

The queue-processing time numbers in the above diagram are from measurements taken during the CPU bound run. The following are the averages of many thousands of subtracted tcpdump timestamps accumulated during the run at Queue Depth 1:

```
                    (  RTT  ) = 275.7 µsec          [Round-Trip Time]
                    (C2 - C1) = 243.4 µsec
                    (S2 - S1) =  32.6 µsec          [Server Time]
```

Other numbers were calculated by applying the earlier formulas to those measurements:

```
                                        SN = CN = 0                          [ACKs always up-to-date at QD=1]
512 Byte TCP_NODELAY Read          (W1 + W2) =  6.3 µsec                     [Wire Time both directions]
                       RTT - (C2 - C1) = 275.7 - 243.4 = 32.3 µsec         [Initiator Time]

                  CL = RTT - (C2 - C1) - (CI + CP + CK)  =  32.3 -  8.6      =  23.7 µsec
                  SL =       (S2 - S1) - (SI + SP + SK)  =  32.6 - 15.9      =  16.7 µsec
(CT + SR) + (ST + CR) = (C2 - C1) - (S2 - S1) - (W1 + W2) = 243.4 - 32.6 - 6.3 = 204.5 µsec
```

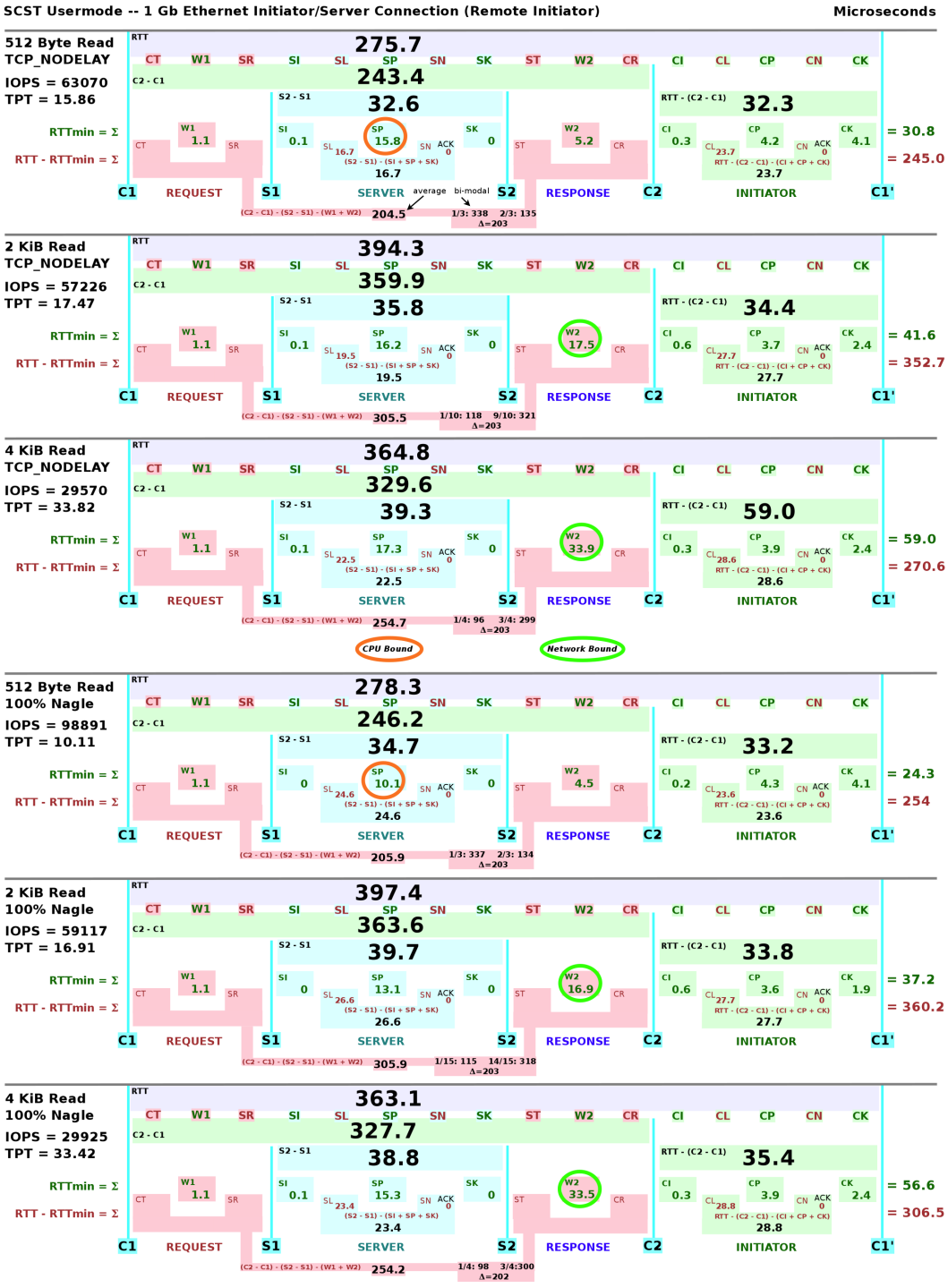
The actual time on the 1 Gb wire for a 512 Byte Read can be computed as about 1.1 microseconds per OP for the 48-Byte Request and about 5.2 microseconds per OP for the 560-Byte Response; those sum to 6.3 microseconds of wire time for both directions combined. The timestamps indicate 210.8 microseconds "out on the network" between tcpdump capture points for both directions combined.

The 204.5 microsecond difference between those is latency unaccounted for somewhere between the tcpdump capture points and the wire, combining both directions on both machines. This is the time represented by the sum (CT + SR + ST + CR).

The colorful table summarizes the breakdown of iSCSI Read Round-Trip Time at three I/O sizes { 512, 2048, 4096 } Bytes. The upper three stanzas show results with TCP_NODELAY set; the lower three show results with Nagle's algorithm enabled 100% of the time. The "bottleneck" queue-processing stage in each case is circled, either the Server Thread CPU time (for CPU bound I/O sizes), or the network Response transmission time (for network bound I/O sizes). 2048-Byte Reads were network bound in these test runs either with or without TCP_NODELAY set.

The TPT listed on the left side under the IOPS is computed as (1/IOPS). This should approximate the circled bottleneck stage queue-processing time. Sometimes a component is "known to be equal" to something else for one of these reasons:

Condition	Equality	Reason
TCP_NODELAY (Server)	SN = 0	Nagle's algorithm is disabled
Queue Depth 1	CW = CN = 0	Initiator received ACK-update piggyback on the Response that triggered transmit of Request
Queue Depth 1	SW = SN = 0	Server received ACK-update piggyback on the Request that triggered transmit of Response
Queue Depth 1	CQ = CL	OPs arrive one at a time; no OPs queued ahead
Queue Depth 1	SQ = SL	OPs arrive one at a time; no OPs queued ahead
Network Bound	SN = 0	Ample supply of Responses available to fill up "partial packets"
Network Bound	ST = CR = 0	Reply network always busy; start latency not incurred
CPU Bound	SL = 0	Session Thread always busy; start latency not incurred
CPU bound	CQ = CL	Responses arrive at the Initiator one at a time; Initiator faster than Server (CP < SP)
CPU bound	SW = SN	tcp_wmem is always empty when written (because SP > W2)

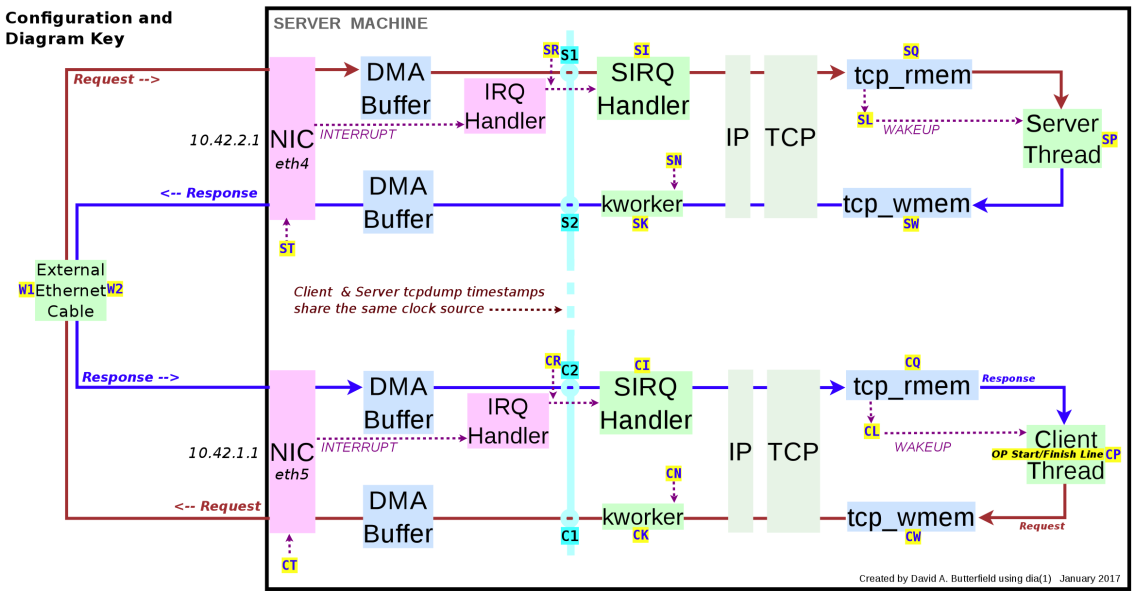


This is a case where averages may not tell enough of the story. The averages for the network Request and Response segments of RTT conceal an underlying bi-modal behavior. For example, the 512-Byte TCP_NODELAY numbers (top stanza in RTT Breakdown table above) show an average of 204.5 microseconds for total transmission and receive latencies (pink); but examining the fine print what is really happening is: 1/3 of the time the latency is 338 microseconds, and 2/3 of the time it is only 135 microseconds.

In the above analysis we could only calculate the sum of the two directions combined, due to lack of clock synchronization between the Server and Client machines. The next configuration facilitates calculation of (CT + SR) separately from (ST + CR) by giving the Client and the Server the same clock source, by running them on the same machine. The next diagram is very similar to one above.

The Linux networking layer is inclined to bypass the physical interface when a destination IP address is local to the machine, so there is a little setup using Network Address Translation (NAT) to get packets to transit over the physical wire for timing purposes.

Server Machine with Local Client Configured to Use External 1 Gb Ethernet Link



Notes:

Running both Client and Server on the same machine sidesteps the clock synchronization problem, enabling calculation of unidirectional pipeline segments through the low-level network and interrupt layers.

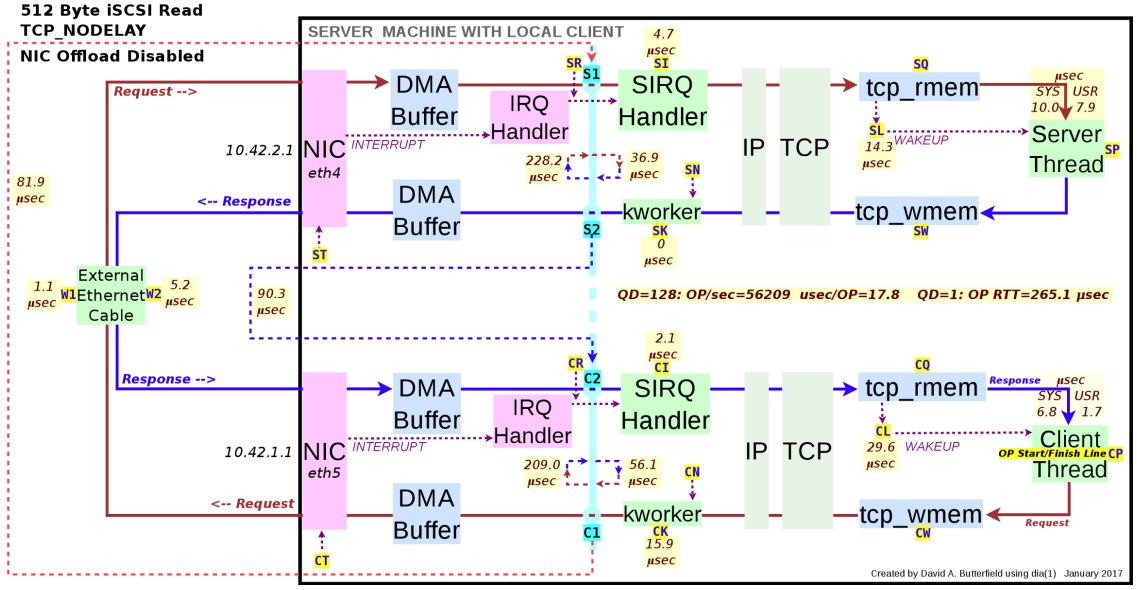
NAT rules facilitate routing traffic through the external cable: client connects to "fake" 10.42.2.100; route and ARP entries set for that address force traffic out eth5; also vice versa

```
DST 10.42.2.100: SNAT 10.42.1.0/24 -> 10.42.1.100
                      SRC eth4:  DNAT 10.42.2.100 -> 10.42.2.1
DST 10.42.1.100: SNAT 10.42.2.0/24 -> 10.42.2.100
                      SRC eth5:  DNAT 10.42.1.100 -> 10.42.1.1
```

To allow iSCSI Initiator to find Server portal for login, set "ifconfig eth4:0 10.42.2.100 up". After login, "ifconfig eth4:0 down" to start routing traffic

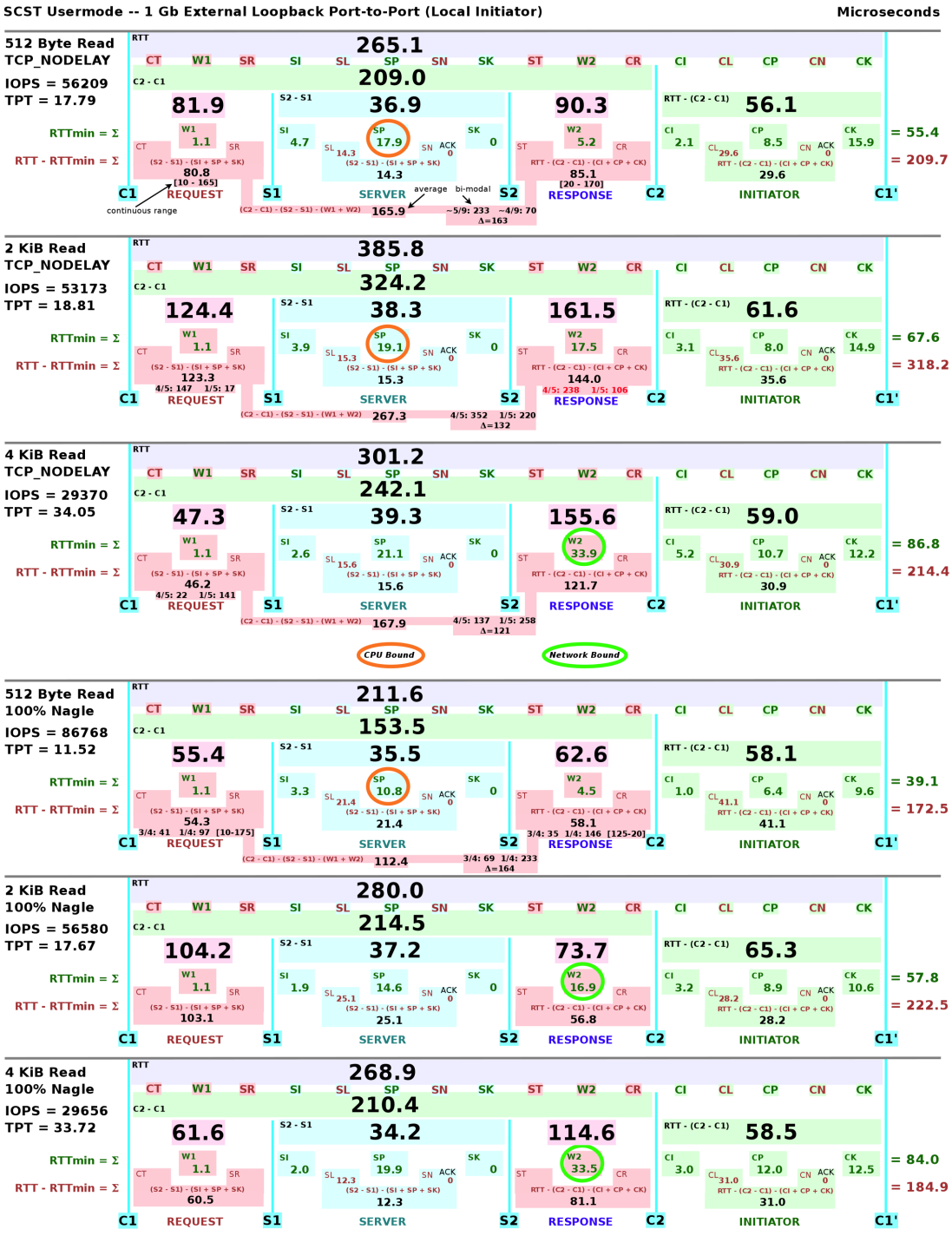
Here's the same diagram with timing numbers filled in from a test with the Client running on the same machine as the Server, communicating through a pair of ethernet ports and an external cable. As in the earlier "numbers" diagram, workload was 512 Byte Read operations from /dev/zero with TCP_NODELAY set, with runs at QD=1 and QD=128.

SCST Usermode Server with Local Initiator Configured to Use External 1 Gb Ethernet Link



Under a CPU-bound workload the Session Thread takes about 13% longer to process an operation when running in this "loopback" configuration, as compared with using a separate client; so here we have 56,200 IOPS instead of the 63,000 IOPS seen in the earlier diagram depicting results of the run with a separate client machine. The difference in IOPS is entirely attributable to the Session Thread, with about 80% of the increase in SYS time, 20% in USR time. This performance decrease only occurs with CPU-bound workloads. I speculate that this is due to contention for memory bandwidth during block-copy of incoming and/or outgoing socket data.

This is just like an earlier diagram, but with numbers from test runs in the **external loopback configuration** rather than using a separate Client machine. The 2048-Byte TCP_NODELAY case has moved from network-bound to CPU-bound due to the few percent lower performance pushing it over the line. The CPU time also increases with Nagle's algorithm enabled (lower three stanzas), but only about half as much, and not enough to put the CPU time longer than the Response transmission time.



The final few charts plot time intervals measured for a few thousand individual samples in series taken under these workloads at Queue Depth 1:

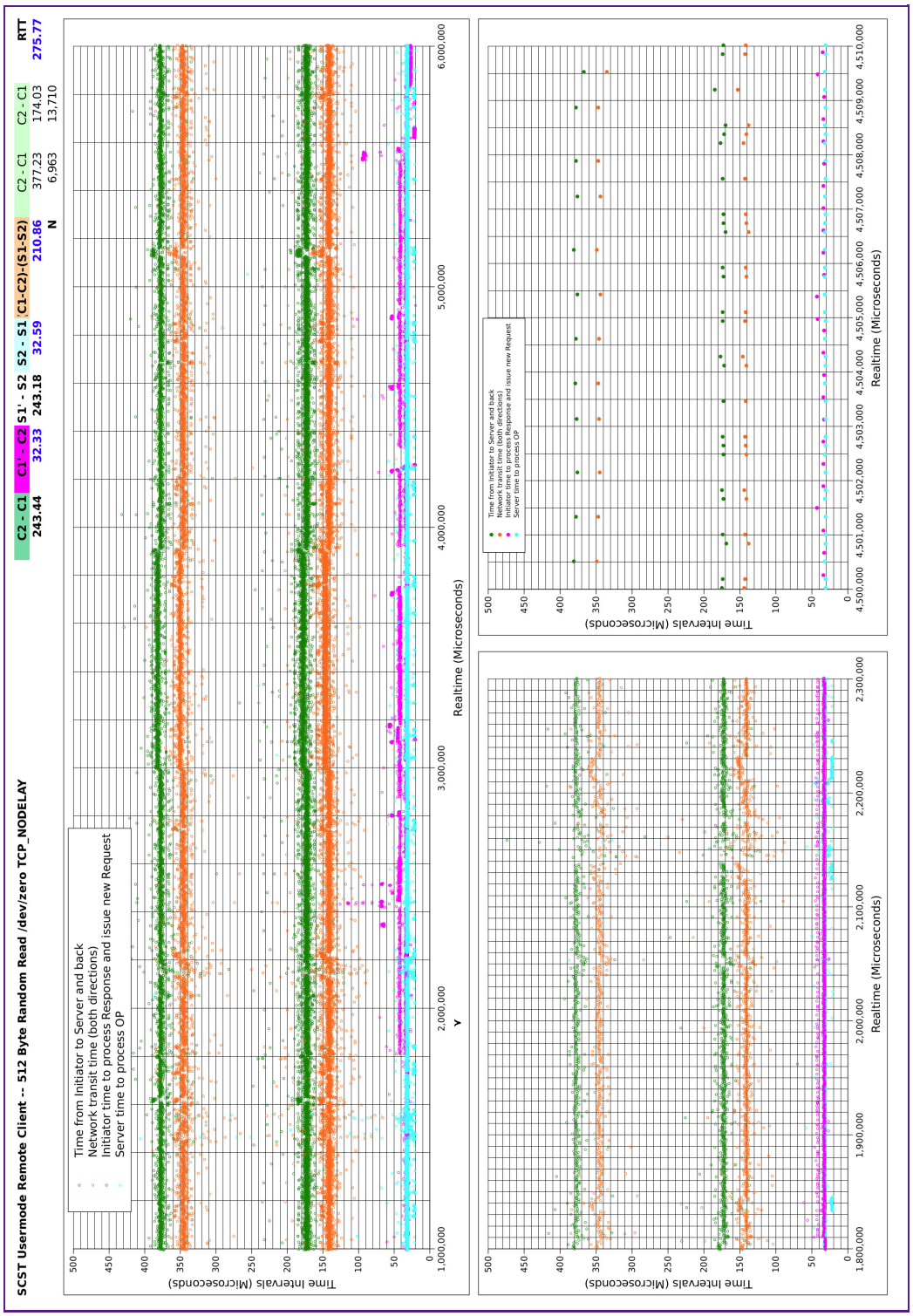
- Remote Initiator 512 Byte TCP_NODELAY
- Local Initiator 4096 Byte TCP_NODELAY
- Local Initiator 4096 Byte 100% Nagle
- Local Initiator 512 Byte TCP_NODELAY
- Local Initiator 512 Byte 100% Nagle

Realtime proceeds along the X axis, while the Y axis denotes time intervals taken to complete various stages of the pipeline for an individual operation. The charts on a page represent different time scales, with individual operation timing patterns visible in the last chart(s) of each page. A green dot denotes "time outside the Client"; cyan denotes "time inside the Server"; orange is the difference between those two; magenta is time inside the Client.

This first chart starts with the **remote Initiator** (not looped back) **512 Byte Random Read with TCP_NODELAY** set; the pattern is fairly regular over time. Looking at the closeup view, there is one higher-latency operation for every two lower-latency operations. The orange dots denote "time outside of both the Server and the Client": the network transmission time and latency in both directions. The difference between the two rows of orange dots represents the difference in network time between the faster and the slower OPs. Subtracting shows 203 microseconds higher time for 1/3 of the OPs, spaced (about) evenly among the faster OPs. All the difference is outside the tcpdump capture points — the Server and Client processing times per OP (cyan and magenta) are fairly constant.

The other Remote Initiator plots (not included) look very similar to this one, with a few percent variation in the latency values from size to size. This was true for both TCP_NODELAY and with Nagle's algorithm enabled, with the two showing similar latencies at each I/O size measured.

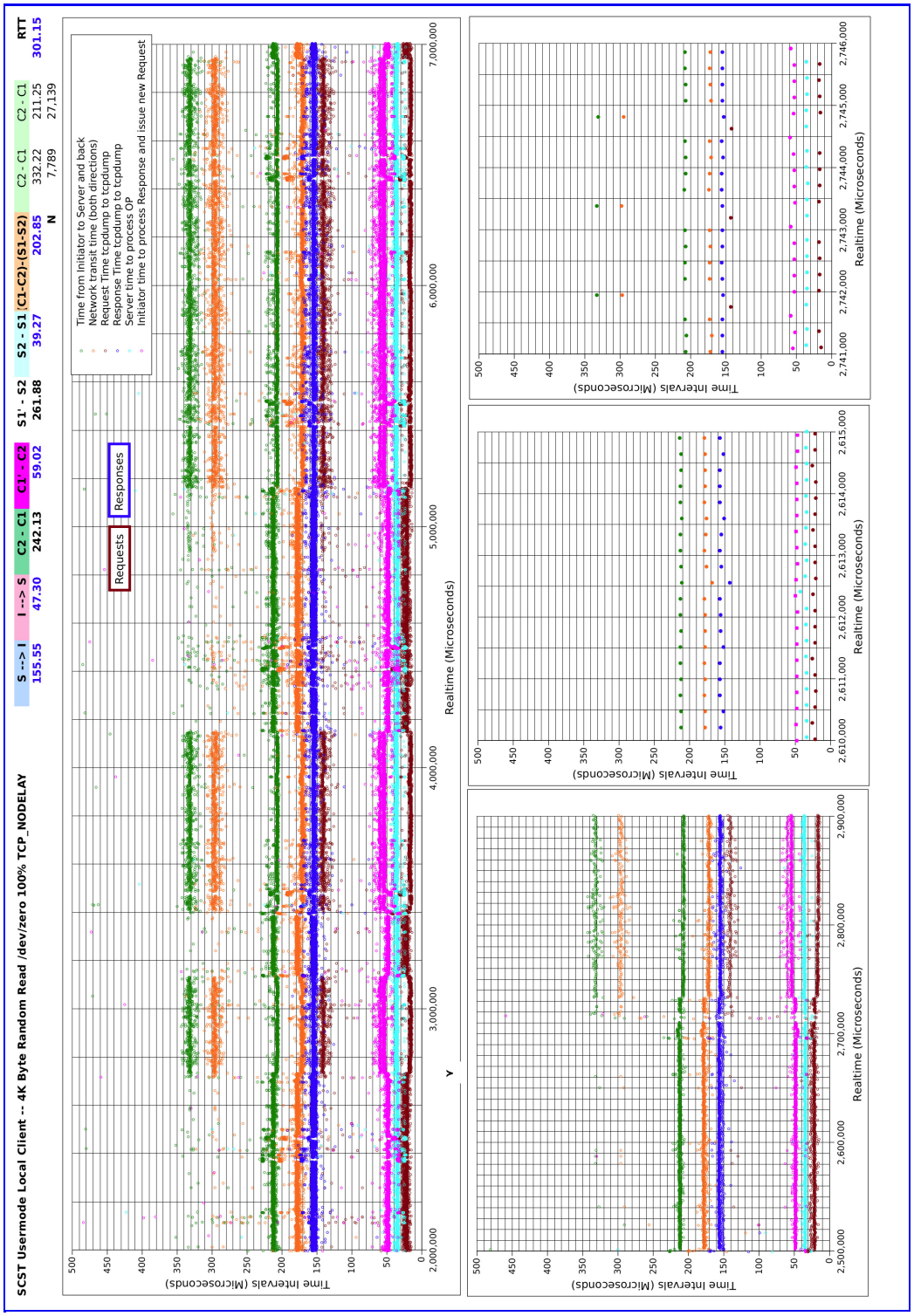
But in all three sizes for both TCP_NODELAY and Nagle mode, although the absolute values vary, **the difference between the higher-latency OPs and the lower-latency OPs is always about 203 microseconds** (when the Initiator is on a different machine).



The rest of the charts show data from runs with the **Initiator running on the same machine** as the Server. With the synchronized clocks between Initiator and Server we can add two more sets of dots: red for Request transmission (wire time plus transmit and receive latencies) and blue for Response transmission. (The values of red and blue should sum to the value of orange.)

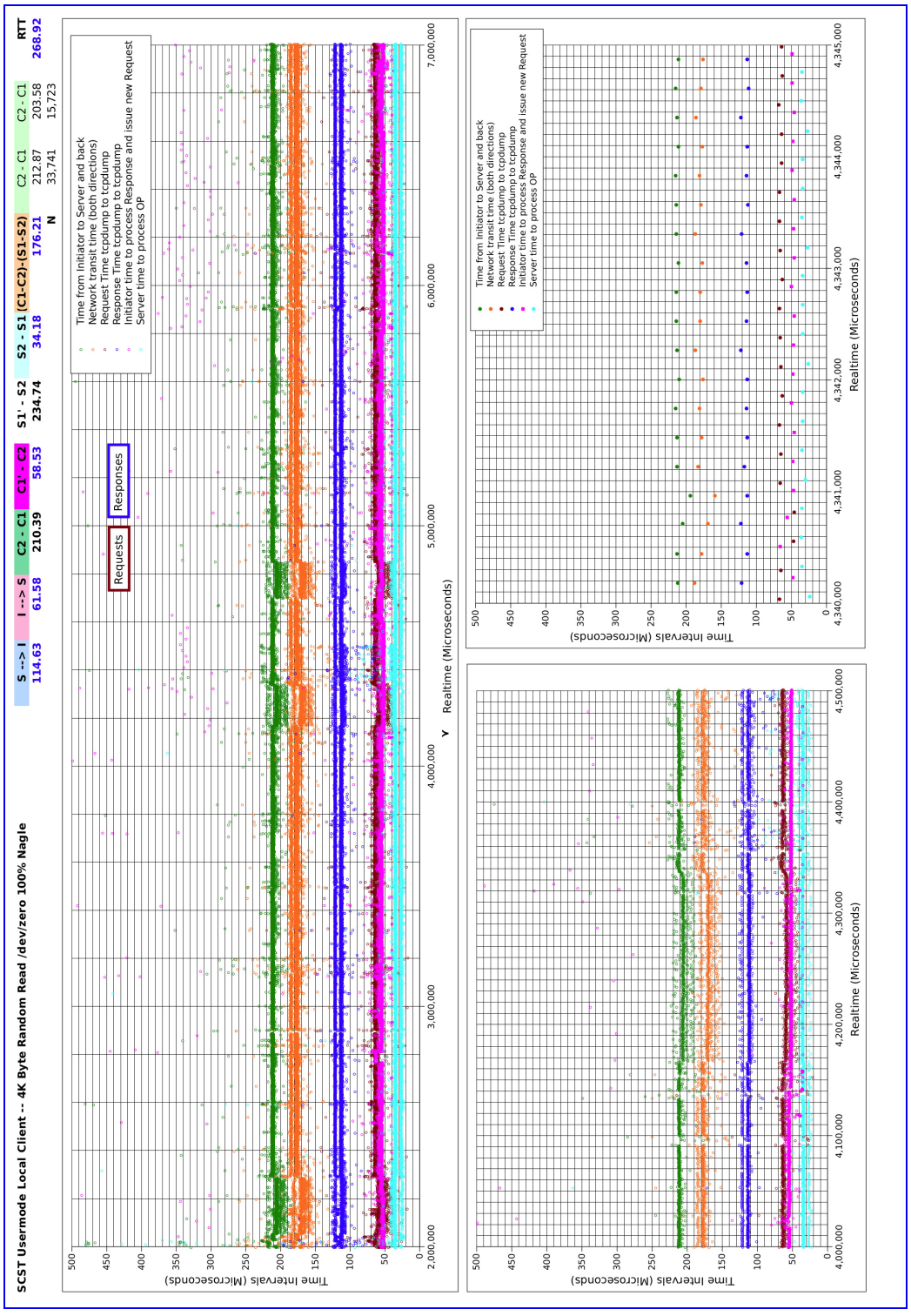
The first local-Initiator chart is for **4096-Byte Reads with TCP_NODELAY set**. There are two modes apparent in the 5 seconds of realtime shown in the top plot: one mode with all ops completing after about the same amount of time; and another mode (with split orange lines) where one OP out of every five takes about 119 microseconds longer. The detailed view shows a regular repeating pattern of every fifth operation taking longer, not simply an "average" of 1/5. Here again all the extra time is attributable to transmission and receive latencies, as implied by the split orange line, and the (more-or-less) constant Server (cyan) and Initiator (magenta) lines.

The addition of separate Request (red) and Response (blue) transmission times lets us see it is the Request component that is bi-modal in this case, switching between about 22 microseconds and 141 microseconds; while the Response time stays constant.



This chart shows **4096-Byte Reads**, same as the previous chart except **with Nagle's algorithm enabled** (100% of the time). Here we do not see the second mode with 1/5 of the Requests experiencing a higher latency, as in the previous chart for TCP_NODELAY.

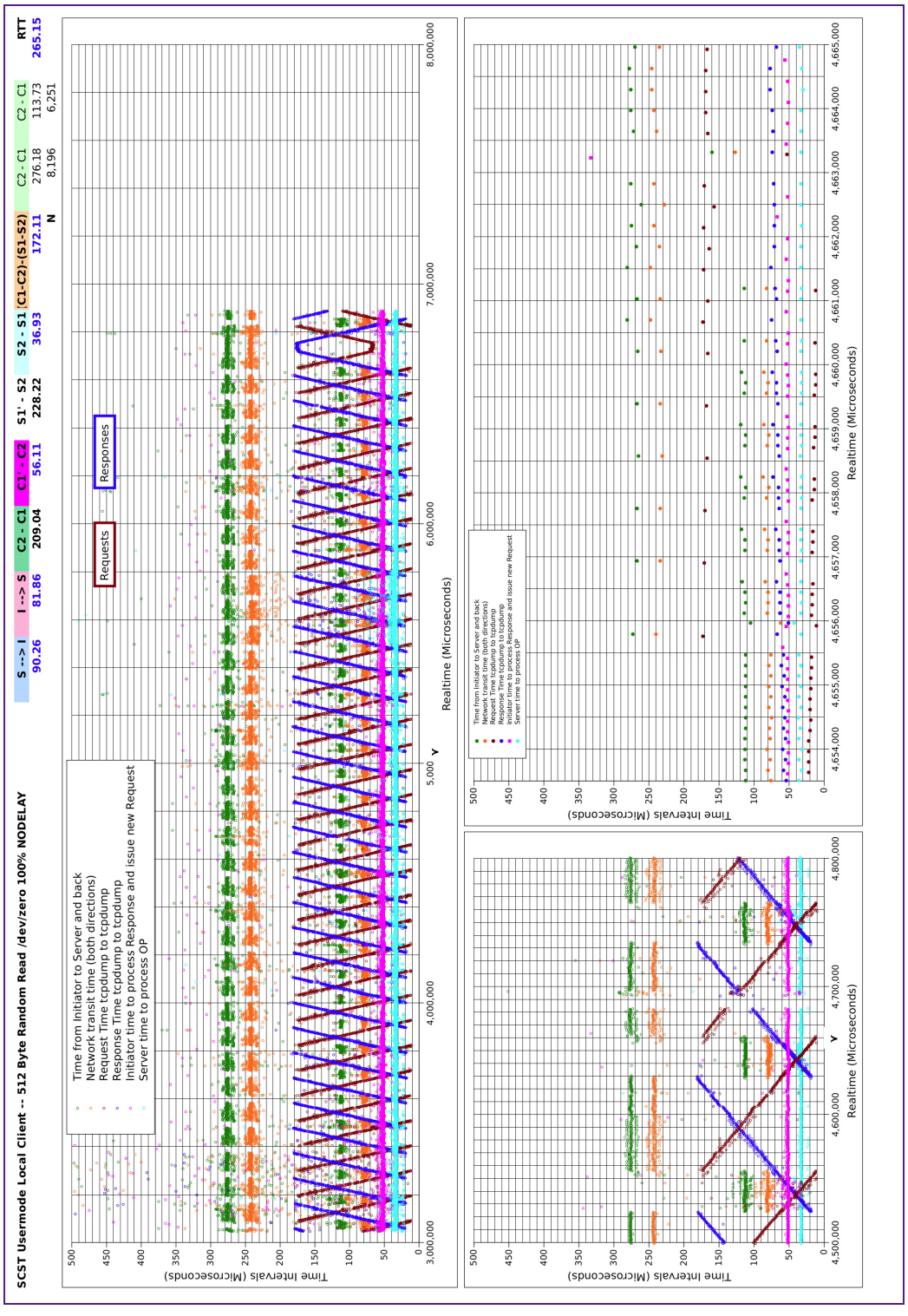
Notice that although the total time here is about the same as with TCP_NODELAY in its better mode, the Request latency has shifted up, and the Response latency has shifted down by about the same amount, bringing them closer to each other.



At **512 Byte Read size** an unexpected pattern emerges, seen both in TCP_NODELAY mode (here) and with Nagle's algorithm enabled (next). The split orange lines indicate bi-modal total network latency time in both directions combined. But in the earlier case the underlying component of the bi-modal total network latency was bi-modal *Request* latency, with Response latency constant.

Here we have something entirely different: continuous ramping *down* of Request latency (red) and ramping *up* of Response latency (blue), together maintaining a constant *sum* (orange); punctuated by periodic jumps of each latency back to its starting value about every 105 MILLIseconds. But the Request latency jumps *up* about 30 milliseconds *after* the Response latency jumps *down* (bottom-left view), leading to their sum (orange) alternating between a "high" constant for 75 ms and a "low" constant for 30 ms in a repeating 105 ms cycle.

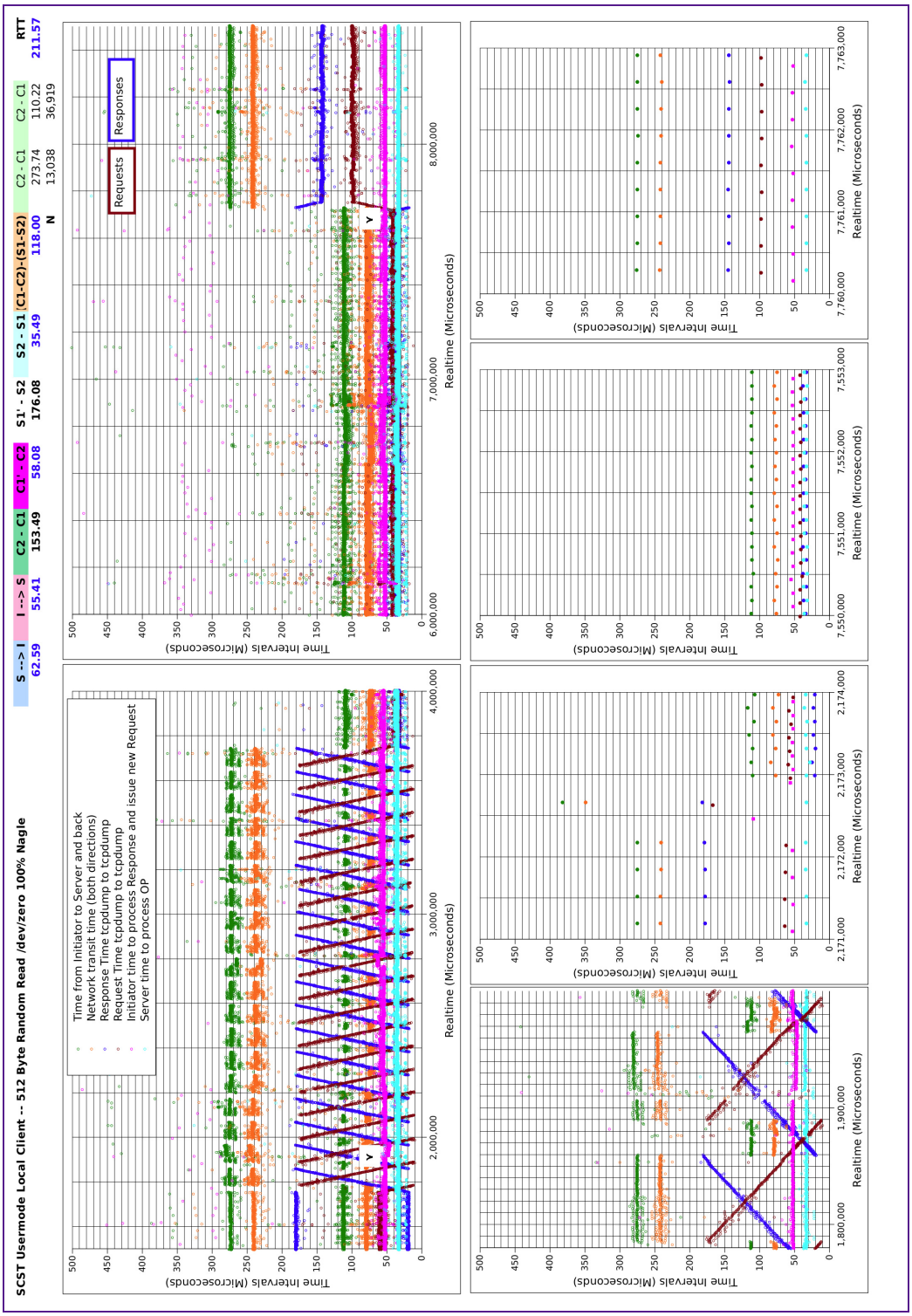
Notice just before the end at the right side of the top plot, the pattern suddenly reverses so that it is Request latency that is ramping up and Response latency ramping down the last couple of cycles. (None of this madness would be visible looking at averages, which probably contributes to their popularity of use.)



Next is the same 512-Byte I/O size with Nagle's algorithm enabled, where we can see the same pattern during part of the running time. Other patterns also appear, switching between modes with widely varying total network latency every few seconds.

Keeping in mind these are plots at Queue Depth 1, the two directions of Request and Response transmission are never concurrent, so there should be no conflict happening between them. Yet the charts show a clear correlation between Request latency and Response latency. I don't think I have enough information to tell whether this is a resource conflict occurring only because the Initiator is running on the same machine as the Server, or if this is something that might occur even with a separate Initiator machine.

Possibly network transmit and receive interrupts interfere with each other and sometimes delay processing of an incoming packet or transmission of a pending one. I would not expect this to occur with a separate Initiator machine; and I haven't come up with a plausible "Request/Response interference" mechanism for a separate Initiator. On the other hand we do see some kind of unexplained bi-modal behavior back on the plot page for the Remote Initiator... but that does not appear to be the same pattern as here.



This pattern also has me suspecting the possibility of some sort of sliding clock skew in the times fed to tcpdump originating on different CPUs, in which case the "ramping up and down" of Request and Response latencies might be a problem with the measurement and not actually happening the way it appears. But the difference between the extremes is 164 microseconds, which seems like an awful lot of skew for a microsecond clock. This remains to be investigated further.

NAT, ARP, and Route Settings For External Loopback Configuration

```
# eth4  10.42.2.1   Server
# eth5  10.42.1.1   Initiator

##### Initiator (10.42.1.1) --> Server (10.42.2.1)
route add 10.42.2.100 eth5              # to transmit to 10.42.2.100, go out through eth5
arp -i eth5 -Ds 10.42.2.100 eth4        # to transmit to 10.42.2.100 via eth5, transmit to the eth4 MAC address

iptables -t nat -A POSTROUTING -d 10.42.2.100 -s 10.42.1.0/24 -j SNAT --to-source 10.42.1.100
                                        # on transmit to 10.42.2.100 from 10.42.1.1, pretend to be 10.42.1.100
iptables -t nat -A PREROUTING  -d 10.42.2.100 -i eth4      -j DNAT --to-destination 10.42.2.1
                                        # on receive for 10.42.2.100 at eth4, reroute to 10.42.2.1

##### Server (10.42.2.1) --> Initiator (10.42.1.1)
route add 10.42.1.100 eth4              # to transmit to 10.42.1.100, go out through eth4
arp -i eth4 -Ds 10.42.1.100 eth5        # to transmit to 10.42.1.100 via eth4, transmit to the eth5 MAC address

iptables -t nat -A POSTROUTING -d 10.42.1.100 -s 10.42.2.0/24 -j SNAT --to-source 10.42.2.100
                                        # on transmit to 10.42.1.100 from 10.42.2.1, pretend to be 10.42.2.100
iptables -t nat -A PREROUTING  -d 10.42.1.100 -i eth5      -j DNAT --to-destination 10.42.1.1
                                        # on receive for 10.42.1.100 at eth5, reroute to 10.42.1.1

ifconfig eth4:0 10.42.2.100 up          # let Server think it should advertise this address for login

set -v                                  # see what we've got
iptables -t nat -L -n -v
arp -n -v
route -n -v
ifconfig eth4
ifconfig eth5
# XXX  This takes two or three minutes to start working -- probably TCP timeout and iSCSI auto-relogin
##########  ifconfig eth4:0 10.42.2.100 down     # start routing traffic through external cable after login  ##########
```

```
iptables -t nat -L -n -v
Chain PREROUTING (policy ACCEPT 0 packets, 0 bytes)
 pkts bytes target     prot opt in     out     source               destination
    0     0 DNAT       all  --  eth4   *       0.0.0.0/0            10.42.2.100          to:10.42.2.1
    0     0 DNAT       all  --  eth5   *       0.0.0.0/0            10.42.1.100          to:10.42.1.1

Chain INPUT (policy ACCEPT 0 packets, 0 bytes)
 pkts bytes target     prot opt in     out     source               destination

Chain OUTPUT (policy ACCEPT 0 packets, 0 bytes)
 pkts bytes target     prot opt in     out     source               destination

Chain POSTROUTING (policy ACCEPT 0 packets, 0 bytes)
 pkts bytes target     prot opt in     out     source               destination
    0     0 SNAT       all  --  *      *       10.42.1.0/24         10.42.2.100          to:10.42.1.100
    0     0 SNAT       all  --  *      *       10.42.2.0/24         10.42.1.100          to:10.42.2.100

arp -n -v
Address                  HWtype  HWaddress           Flags Mask            Iface
10.42.2.100              ether   00:15:17:93:90:03   CM                    eth5
10.42.1.100              ether   00:15:17:93:90:02   CM                    eth4
Entries: 12     Skipped: 0      Found: 12

route -n -v
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
10.42.1.0       0.0.0.0         255.255.255.0   U     1      0        0 eth5
10.42.1.100     0.0.0.0         255.255.255.255 UH    0      0        0 eth4
10.42.2.0       0.0.0.0         255.255.255.0   U     1      0        0 eth4
10.42.2.100     0.0.0.0         255.255.255.255 UH    0      0        0 eth5

ifconfig eth4
eth4      Link encap:Ethernet  HWaddr 00:15:17:93:90:03
          inet addr:10.42.2.1  Bcast:10.42.2.255  Mask:255.255.255.0

ifconfig eth5
eth5      Link encap:Ethernet  HWaddr 00:15:17:93:90:02
          inet addr:10.42.1.1  Bcast:10.42.1.255  Mask:255.255.255.0

# XXX  This takes two or three minutes to start working -- probably TCP timeout and iSCSI auto-relogin
##########  ifconfig eth4:0 10.42.2.100 down     # start routing traffic through external cable after login  ##########
```