On Systems Integration: Tuning the Performance of a Commercial TCP Implementation

D. Leon Guerrero
Network Solutions Inc.
Herndon, Virginia

Ophir Frieder*
Dept. of Computer Science
George Mason University
Fairfax, Virginia

Abstract

We describe common TCP implementation pitfalls and provide novel solutions to solving these deficiencies. The performance enhancements described herein are implemented in Network Solutions' OPEN-Link TCP product. The results demonstrate significant performance improvements over the prior release. The techniques described improve overall network performance, while in some instances, also reduce CPU demands. The majority of the improvements are incorporated into the TCP Window Management. Although our study is focused specifically on the TCP protocol, the lessons learned are well suited to other network protocols.

1.0 Introduction

One of the more commonly deployed network protocol is the Transmission Control Protocol (TCP), Internet Protocol (IP). Although the protocol specification for TCP is publicly available, incompatibilities between implementations still exist. Some of these compatibility problems are immediately noticeable, as error messages may be printed or the system may become non-functional. However, more subtle incompatibilities may also result in poor network performance.

Our experimental study describes common TCP implementation pitfalls and provides solutions to solving these deficiencies. The TCP enhancements described herein are implemented in Network Solutions' OPEN-Link™[1] TCP product. The results demonstrate significant performance improvements over the prior release. In certain areas, the performance improved by 600%. Section 2.1 provides detailed comparisons between OPEN-Link and other vendor implementations. The comparisons include our original implementation as well as the enhanced high performance version.

*This work was partially funded by a grant from the National Science Foundation under Grant Number CCR-9109804

[1]OPEN-Link is a trademark of Network Solutions Inc.

The early TCP implementations were designed to operate over the ARPAnet[14]. By its very nature, the ARPAnet is a low-speed, high-delay, wide area network. ARPAnet's limited network bandwidth disguised some performance pitfalls of TCP implementations. In today's demanding high speed Local Area Networks (LAN) however, TCP must operate in environments that require connectivity between desk-top computer upwards to supercomputers.

Many TCP implementations solve this problem by providing configurational parameters that may be manually adjusted to fine tune performance. Configurational parameters alone are not adequate; they only work well for limited number of instances. Instead, adjustments and adaptation to real-time scenario requires intelligent, dynamic calculations. To be effective, these dynamic calculations must be performed with minimal latency and without the expense of significant CPU overhead.

We describe techniques to improve overall network performance, while in some instances, also reduce CPU demands. The majority of the improvements are incorporated into the TCP Window Management. Additionally, we describe checksumming algorithms as well as header prediction techniques. Although our study is focused specifically on the TCP protocol, the lessons learned are well suited to other network protocols.

2.0 Common Pitfalls

Measurements examining interpacket latency and byte throughput rate are crucial to the evaluation of TCP performance. The interpacket delay introduced by the protocol layers provide needed information to reveal where the bottlenecks occur. To acquire performance measurements, we used an EX-5000™[2] LAN analyzer. The EX-5000 provides time stamped packet traces necessary to analyze acknowledgement delays and interpacket latency.

To isolate the problem areas, we conducted a series of tests to measure the throughput rate at various

[2]EX-5000 is a trademark of Novell Inc.

0-8186-2697-6/92 $03.00 © 1992 IEEE

DOCKET ALARM
Find authenticated court documents without watermarks at docketalarm.com.

network layers. We examined three distinct layers: the ethernet layer, (includes the operating system overhead), the IP layer (connectionless) and the TCP layer (connection oriented with flow control). The tests are designed to continuously send datagrams at each protocol layer. The datagram size, initially set to 64 bytes (minimum ethernet frame size) is increased 10 bytes after each test suite until the datagram size reached 1512 bytes (maximum ethernet frame size). The purpose for this exercise is twofold; to determine the optimum packet size and to identify the protocol layer where bottlenecks may occur. Figure-1 and Figure-2 show the graphs of the packet and byte throughput rates, respectively, when varying the packet size.

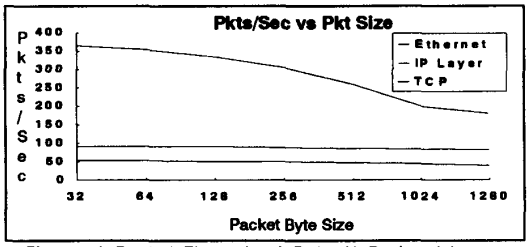


Figure-1 Packet Throughput Rate At Protocol Layers

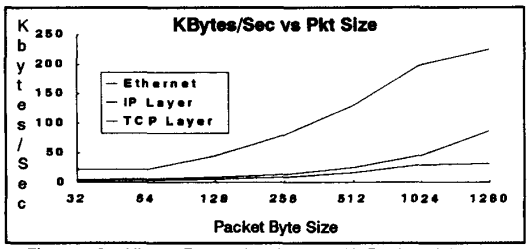


Figure-2 Kbyte Throughput Rate At Protocol Layers

By examining the graphs in Figure-1 and Figure-2, it appears that the TCP and IP layers introduce significant performance degradation. In Figure-1, the TCP and IP curves remain relatively flat, compared to the Ethernet curve which depicts a direct correlation between packet size, packet rate, and byte rate. This observation implies that regardless of the packet size, our TCP implementation requires the same amount of processing time. Intuitively, the packet size must affect the packet throughput rate since operations such as checksumming, buffer moves and CRC calculation must operate on larger buffers. Close observation of the Ethernet curve in Figure-1 validates this assumption.

In our implementation, the ethernet Network Interface Module (NIM), TCP, and IP, are three separate processes communicating via interprocess communication (IPC). Suspecting the operating system overhead to be the limiting factor, we devised a modification which combined the NIM, TCP, and IP into one process. The results of these modifications are shown in Figure-3 and Figure-4.

In Figure-3 and Figure-4, the IP curve shows improved throughput rates which coincides with the Ethernet curve. However, the TCP curve showing a performance increase, remains relatively flat compared to the Ethernet and IP curves.

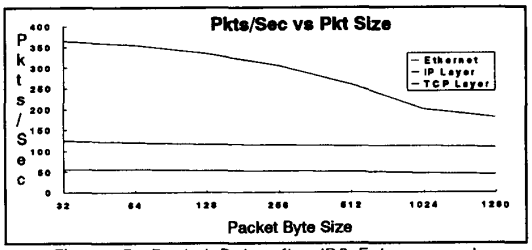


Figure-3 Packet Rate after IPC Enhancement

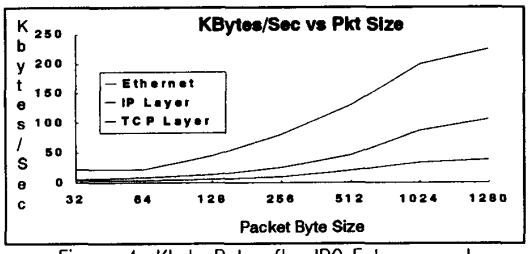


Figure-4 Kbyte Rate after IPC Enhancement

By examining the EX-5000 traces closely, several problems were diagnosed. The first occurred when communicating with a UNIX workstation. Independent of the datagram sizes, TCP eventually transmits only small datagrams (32-128 data bytes) resulting in the UNIX acknowledgements behaving erratically.

As Clark[1] points out, many TCP implementors blame excessive processing overhead associated with TCP for poor network performance. Like many protocol implementors, we followed the TCP specifications verbatim. Our study confirms the claim and concludes that the performance degradation is often attributed to the poor protocol implementation instead of the protocol overhead processing or protocol architecture. Pitfalls in many TCP software reside in the acknowledgement processing. Some of these pitfalls include sending multiple acknowledgement packets that can be similarly accomplished by delaying the acknowledgement, compacting the information, and then sending one acknowledgement packet.

TCP has a subtle trap attributed to its nature of being a byte sequence architecture. A phenomenon known as the Silly Window Syndrome (SWS) [2] can result in poor performance in many TCP implementations. The

DOCKET ALARM
Find authenticated court documents without watermarks at docketalarm.com.

SWS phenomenon occurs between two dissimilar flow control implementations causing data transmission to thrash much like disk I/O on a badly fragmented disk. This occurrence results in data being badly fragmented and inefficiently transmitted in small packet sizes.

2.1 Performance comparisons

Comparisons of other vendor products as well as our implementation and the enhanced high performance version are presented in Table-1. Table-2 lists the platforms which were used for these benchmarks.

	Excelan	OL/CCUR R2-0	OL/CCUR R2-1	OL/ MVS	PC/ TCP	SCO	SUN	WIN/ TCP
Excelan	n/a	5	62	68	45	50	50	68
OL/CCUR R2-0	5	20	5	n/a	5	5	5	5
OL/CCUR R2-1	62	5	65	n/a	68	68	70	70
OL/MVS	68	n/a	n/a	190	68	n/a	250	n/a
PC/TCP	45	5	68	68	45	45	45	68
SCO	50	5	68	n/a	45	n/a	96	n/a
SUN	68	5	65	250	45	96	250	84
WIN/TCP	68	5	70	n/a	68	n/a	84	n/a

2.5 MByte file transferred in binary mode. Measurements are KBytes/Second.

Table-1 Performance Benchmarks

TCP/IP PRODUCT	PLATFORM	MAIN CPU	Operating SYSTEM	MIPS	ETHERNET
Excelan	Compaq-286	Intel 286/12	PC-DOS	7.7*	Lance
OPEN-Link CCUR	CCUR 3210	Proprietary	OS/32	1	Interlan
OPEN-Link MVS	IBM 3090	Proprietary	MVS	72	Interlan
PC/TCP	PC Clone-386	Intel 386/16	PC-DOS	12.4*	WD8003e
RTU/TCP	CCUR 6000	MC 68000	UNIX	4	Lance
SCO	AST-486	Intel 486/33	SCO UNIX	72*	WD8003e
SUN	SUN Sparc 1	Proprietary	SUN-OS 4.0	15	SUN
WIN/TCP	VAX 8250	Proprietary	VMS	3.0	DEC

Million Instructions per Second base on manufacturer specification, unless otherwise denoted.
* Denotes Compute Index using Norton Utilities

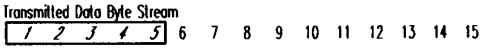
Table-2 Platform Specifications

3.0 TCP flow control

TCP has been well studied both in practice and in the literature. For brevity, we forgo a detailed description of TCP, referring the reader to the two volume series by Comer[2] and RFC 793 and 1122. This study extends the study by Clark[13] by introducing techniques that optimize frame-fill and avoid unnecessary fragmentation. To better understand our efforts, a very brief description of TCP is provided.

TCP provides a connection oriented transport service. Data delivery is guaranteed in a byte sequenced order. TCP implements the positive acknowledgement technique using acknowledgement and sequence numbers to synchronize datagram delivery. Similarly, the sliding window technique is used to manage the data flow control.

TCP flow control implements the sliding window technique. During the connection setup, each endpoint advertises its maximum receive window. The receive window restricts the number of unacknowledged bytes that may be in transit at any given time. The TCP sequence numbers and acknowledgement numbers are used to manage the byte order and flow control. The acknowledgment number indicates the last data byte received by the sender of the packet. Similarly, the sequence number indicates the sequence in the data stream of the first byte in the packet. Figure-5 illustrates an example of data transmission.



TCP Client State	Direction	Packet Type	Seq No.	Ack No	Data Len	Window	TCP Server State
ESTABLISHED	-->	DATA	101	901	5	5	ESTABLISHED
ESTABLISHED	<--	ACK	901	106	0	0	ESTABLISHED
ESTABLISHED	<--	ACK	901	106	0	5	ESTABLISHED

Figure-5 Acknowledgement Strategy & Flow Control

Figure-5 shows that the constraint which throttles the flow of data is the receive window (advertised as 5 bytes). The sender (TCP client) sends 5 bytes of data and fills the entire receive window. The receiver (TCP server) then replies with an acknowledgement that it has received 5 data bytes along with an updated receive window. The TCP receiver's buffer is full as this is reflected in the receive window becoming zero.

To form the acknowledgement number the receiver adds the data length to the sequence number. Since the data are still in the server's TCP receive buffer, the server replies with a zero receive window to indicate that its receive window is full and that it is unable to receive additional data momentarily.

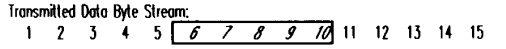
At some later time, after the TCP receiver delivers the data to the recipient, (an application program such as file transfer), the TCP receiver sends an updated window to the TCP sender. In this example, data are being transmitted only in one direction (client to server). Therefore, the server's sequence number, (which is the counterpart of the client's acknowledgment number), remains unchanged.

When the TCP receiver finally delivers the data to the application, the TCP receive buffer becomes available to receive additional data. Upon receiving the window update, the sender's window then slides based on the acknowledgement number and receive window. Figure-6 illustrates the sliding window concept.

In Figure-6, the sender's send window advances from bytes 1-5 to bytes 6-10. For simplicity of our discussion, a receive window of 5 bytes is selected.

DOCKET ALARM Find authenticated court documents without watermarks at docketalarm.com.

Realistically the receive window must be sufficiently large to accommodate multiple packets to allow a continuous transmission. For example, if ethernet is used, (maximum frame size 1512 bytes), and it takes 4 packets to provide a continuous transmission stream then the receive window would be 6048 bytes.



TCP Client State	Direction	Packet Type	Seq No.	Ack No	Data Len	Window	TCP Server State
ESTABLISHED	-->	DATA	106	901	5	5	ESTABLISHED
ESTABLISHED	<--	ACK	901	111	0	0	ESTABLISHED
ESTABLISHED	<--	ACK	901	111	0	5	ESTABLISHED

Figure-6 TCP Sliding Window Concept

We revisit the TCP acknowledgment strategy in section 5 and describe approaches to reducing CPU overhead by implementing efficient algorithms.

4.0 Round trip time (RTT) calculations

The design of proper strategies for computing Round Trip Time (RTT) calculations is a highly debated topic. The basic concept behind RTT is to calculate the amount of time it takes a packet to be transmitted and acknowledged. While many studies have been published on RTT, most conclude that an efficient RTT approximation is one that provides the calculation within the specified amount of time and with an accuracy needed for the particular application[8,9].

Our requirements mandate an RTT computation of at least every 16ms and must provide an accuracy within 30ms. These requirements were derived by carefully examining the performance of the SPARC-1™[3] workstation using an EX-5000 analyzer. The EX-5000 analyzer provides a timestamped packet trace with an accuracy of 500 microseconds. The EX-5000 traces show two important facts. First, data packets are transmitted at a burst rate, (restricted by the TCP receive window size), with an interpacket spacing less than 500 microseconds, (EX-5000 timestamp limitation). It can be assumed that the interpacket spacing is not less than 9.6 microseconds[4]. Second, the acknowledgement turnaround occurred within 16ms after the end of the packet burst, (hence 16ms compute time). Given these known parameters, we then derived the accuracy requirements of 30ms, (actually 32ms, but system clock limitations restrict us to 30ms), which is influenced by two samples. At this high frequency, RTT computation must use minimal CPU resources.

RTT provides the cornerstone upon which our acknowledgement strategy and back-off algorithms are based. We maintain two variables associated with RTT. A rough estimate on a per-packet basis is referred to as simply RTT. A second variable, Smoothed Round Trip Time (SRTT) is maintained using a Decay Smoothing Algorithm[13]. SRTT computation is necessary to compensate for latency abnormalities associated with the chaotic network behavior. SRTT provides a safety feature by restricting the fluctuation within a predetermined range.

Our SRTT approximation is a modification of Karn's[9] approximation. It is our experience that using integer, instead of floating point, arithmetic, allows faster computation and less CPU overhead. Although floating point arithmetic yields more accurate results, given our limited precision requirement, the CPU overhead does not justify its use.

The RTT and SRTT computations used in our implementation are shown in Figure-7.

```
RTT = Ack_Time - Dispatch_Time
SRTT= (RTT/Delta_W)+(SRTT*(Delta_W-1)/Delta_W)
```

Figure-7 Round Trip Time Computations

RTT is a rough approximation. Ack_Time is the timestamp taken when the TCP sender receives the acknowledgement, and Dispatch_Time is the timestamp when the TCP sender queued the datagram onto a device independent interface. Our implementation provides a hardware independent interface. RTT includes the operating system overhead.

SRTT is a weighted average of the previous SRTT and the latest RTT. The weight factor Delta_W determines variance for the approximation. "Choosing a large Delta_W makes the weighted average immune to changes that last for a short time (e.g., single segment that encounters a long delay). Choosing small Delta_W makes the weighted average respond to changes in delay very quickly."[2, Page-145]

Our goal in computing SRTT is to restrict the fluctuation to within the 25% range. This value was selected because the amount of buffer space allocated per-connection is set to 4096 bytes, (memory constraints to support 32 simultaneous sessions). This yields 3-4 packets per burst. Assuming that the receiving TCP can accommodate the burst, the weight factor Delta_W must then be 8. This yields a 12.5 percent variance on a per-packet basis. It is necessary to acquire at least 2 samples to interpolate; using this axiom the 12.5% computation then yields a 25% variance.

All timestamp variables in our TCP implementation are kept in milliseconds since midnight. One drawback to this technique is that a special case for midnight roll over must be taken into account. Since the operating system timer service for this function is fast, we opted to use this technique.

[3]SPARC is a trademark of Sun Microsystems Inc.

DOCKET ALARM
Find authenticated court documents without watermarks at docketalarm.com.

Given an estimation of SRTT that will guarantee our approximation outlined, we present the enhanced acknowledgment strategies to increase the overall network performance.

5.0 Silly Window Syndrome (SWS)

Flow control is an essential requirement for any network protocol. Regulating the flow of data ensures high reliability and efficient data delivery when implemented properly. When implemented poorly, network performance severely degrades!

SWS occurrence causes poor TCP performance. The SWS problem is often attributed to a sender transmitting faster than the receiver can process the input data. SWS is identified by noticing that the transmission consists of mostly small packets when larger packets are supported. The transmission often exhibits thrashing much like I/O on a badly fragmented disk.

SWS causes the actual user data, (data from applications such as file transfer), to be transmitted over an excessive number of fragmented packets. Because each packet requires protocol processing as well as operating system overhead, occurrence of SWS results in excessive overhead processing.

To properly describe fragmentation, we must clarify our use of three terms: *datagram, TCP segment*, and *packet*. The term *datagram* will imply the block of data that the application program, e.g., file transfer, requests to be transmitted. The term *packet* will imply the actual ethernet frame transmitted on the media and is recorded by the LAN analyzer. The term *TCP segment* will imply the data portion, (excludes TCP, IP, and Ethernet headers), of a packet. In our context, fragmentation does not imply the IP fragmentation as outlined in [11], but instead, describes the relationship between a datagram and packet. Specifically, datagrams may be fragmented over two or more packets.

Figures 8-9 illustrate EX-5000 analyzer traces recorded during a file transfer session. A 2.5 megabyte image file was transferred in binary mode between OPEN-Link, (old version before performance enhancements), and a Unix workstation. The EX-5000 traces are presented to describe SWS, inefficient acknowledgment, and excessive packet fragmentation. To describe SWS using the EX-5000 traces in Figures 8-9, we categorize each packet as: *data, ack* or *ack+winup* packets.

Our premise, as in [1], is that bulk data transfers such as file transfer is the area of performance interest. Such applications often exhibit data transmitted only in one direction. As a result, the acknowledgement number and receive window remain unchanged in the data packets. The relevant TCP parameters that affect the data packet are sequence number, data length, and send window size.

The sequence number in each data packet specifies the data byte offset relative to the Initial Sequence Number (ISN) for the first data byte in the data packet. The ISN in our example is 99, therefore, in Figure-8, data byte 1 in pkt-1 must be associated with sequence number 100. As data are transmitted, the sequence number maintains the accumulative data byte offset. The sequence number for pkt-2 must then be 1552 as shown in Figure-8. This is derived by adding the sequence number of pkt-1 (100) and the data byte length (1452 bytes) of pkt-1.

The maximum number of data bytes that may be transmitted in each packet is restricted by the maximum TCP segment size (1452 bytes) or the amount of send window bytes available, the lesser of either. The send window is derived by subtracting the amount of unacknowledged data from TCP receiver's available window. For example, in Figure-8, when pkt-1 was transmitted, there were no unacknowledged data, therefore the send window is 4096 bytes. Unlike pkt-1, when pkt-2 was transmitted, pkt-1 remains unacknowledged. Therefore, the send window for pkt-2 (2644 bytes) must reflect the unacknowledged data from pkt-1 (1452 bytes).

Packet No.	Packet Type	Seq No.	Ack No	Data Len	Send Window	Receive Window
1	DATA	100	---	1452	4096	---
2	DATA	1552	---	1452	2644	---
3	DATA (fragment)	3004	---	1192	1192(full)	---
4	*ack(1-2)*	---	*3004*	*0*	---	*1192*
5	*ack+winup(1-2)*	---	*3004*	*0*	---	*4096*

Figure-8 EX-5000 SWS Data Packet Traces

As data are transmitted the send window decreases until window updates are received. Pkt-3 (1192 bytes) in Figure-8 must be fragmented because there are only 1192 bytes available in the send window. After transmitting pkt-3, the TCP sender must refrain from sending more data since the receive window has reached zero. This is denoted with the notation *"full"* in the *"Send Window"* column. Transmission of the remaining 260 bytes must be postponed until the TCP receiver returns a window update.

The acknowledgement and window update packets, are denoted as *ack* and *ack+winup*, respectively, in Figure-8. The relevant TCP parameters that affect the ack and ack+winup packets are the acknowledgment number and the receive window.

The acknowledgement number in the ack-pkts specifies the last data byte received. The ack number is derived by adding the data packet's sequence number and the data byte length. The ack and ack+winup packets also reflect the current available receive

DOCKET ALARM Find authenticated court documents without watermarks at docketalarm.com.

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.