Network Acceleration and Time Synchronization for Data Acquisition Systems using Commodity Networks and Operating Systems

Jeffrey Shafer[a], Jesse Conn[b], Cameron Lucas[a], James Caffery[b], Klaus Schug[c]

[a] *University of the Pacific, 3601 Pacific Avenue, Stockton, CA 95211*
[b] *GIRD Systems, Inc., 310 Terrace Avenue, Cincinnati, Ohio 45220*
[c] *Aerospace Testing Alliance (ATA), Arnold Engineering Development Complex, AEDC/ATA/IT&S, 1103 Avenue B, Arnold AFB, TN 37389-1200*

Abstract

A user-mode network stack was designed to reduce the network communication latency and jitter of data acquisition and analysis programs. The resulting system has sub-millisecond I/O latency for small message sizes while running on the Microsoft Windows operating system without any custom hardware requirements or application software modifications. Existing applications are transparently redirected to use this custom network stack through the use of replacement dynamic link libraries that intercept standard networking API calls. A software device driver redirects incoming packets into the user-mode stack, allowing accelerated data (specifically, TCP and UDP for IPv4 and IPv6) to bypass the Windows data path. In addition to the user-mode network stack, a network-based software clock with microsecond resolution at the user-mode software level was also produced. This synchronization solution is compatible with Microsoft Windows and the NTPv4 standard, requires no hardware support, and allows clock synchronization to run over a commodity Ethernet network shared with application data, thereby significantly reducing the cost of deployment. The architecture includes a centralized time service, an API that allows access to this time source from all user-mode applications without the delay of a system call, and a device driver that identifies time synchronization packets in the network stack and bypasses key sources of delay in the operating system.

Keywords: Data acquisition system, Network stack, Time synchronization, Windows OS, Winsock, TCP, UDP, NTP

1. Introduction

High precision and low cost test data acquisition and analysis systems are of wide interest in both commercial and military circles today. These systems depend on timely delivery of data samples from myriad sensors to control, analysis, and logging applications, in order to allow those applications to meet their own strict time deadlines. Further, they also require accurate time synchronization in order to correlate individual data samples across multiple sensors. Current systems employ proprietary and expensive hardware communication systems (such as reflective memory) that can guarantee communication at the user application level within a deterministic time [1], and use external methods for time synchronization (such as GPS or IRIG) [2, 3, 4]. The proprietary nature of the data and time distribution systems increase maintenance costs and force costly upgrades as the data acquisition system scales. This paper describes a method to replace proprietary systems with a more flexible and cost-friendly Ethernet-based system using non real-time computer operating systems (specifically, Microsoft Windows) running on commodity computer hardware. On this hardware, network communication is provided with sub-millisecond latency and low jitter, and network time synchronization is provided with microsecond-level accuracy.

The challenge of coupling a non real-time operating system (*e.g.,* Microsoft Windows) with Ethernet as a data transport mechanism is that such a system introduces random and significant latencies in the transport of data samples. These delays may be caused by the network itself, due to packet loss and queuing delay at Ethernet switches [5]. In a data acquisition system, however, it is reasonable to assume that the network itself is overprovisioned, and thus packet loss and queuing delay at the Ethernet switches is minimal. Thus, the delay is often due to the way the Windows operating system handles network traffic. Network stream bandwidth is prioritized over per-message latency. Variable and random latencies exist from the network interface, up through the Windows network stack, to the user application. These latencies can be of the order of tens to hundreds of milliseconds, thus rendering real-time processing of data across the network unviable [6, 7]. Controlling and reducing these communication delays is essential for data acquisition systems.

In addition to data communication, time synchroniza-

Email addresses: jshafer@pacific.edu (Jeffrey Shafer), JConn@girdsystems.com (Jesse Conn), JCaffery@girdsystems.com (James Caffery), klaus.schug.ctr@us.af.mil (Klaus Schug)

[1] For more information, visit http://girdsystems.com/

tion is also essential in a data capture system. To enable reliable post-test data analysis, individual data samples must be correlated across different sensors. This requires time-stamps across various processing computers to be highly synchronized, which is typically accomplished using sources like GPS and IRIG. These existing methods have a variety of drawbacks, however. Both require specialized receiver cards to be installed at each node, and both require extensive physical wiring: GPS to reach antennas with clear satellite reception, and IRIG to reach a local master clock. This wiring is separate from the data communication network.

Instead of hardware-based GPS and IRIG time synchronization techniques, time can also be synchronized over the same commodity Ethernet network through protocols such as NTPv4 (Network Time Protocol, version 4) and IEEE 1588-2008 (also called the Precision Time Protocol or PTP). NTP provides a portable end-to-end software-only solution. However, its algorithms are negatively impacted by network delay and jitter at both the Ethernet switch level (due to congestion) and the host level (due to delays in moving network data between the NIC, OS, and NTP application). As such, the clock accuracy is only on the level of a few milliseconds for existing systems running NTP. IEEE 1588 cancels out network delay and jitter to achieve accuracy in the sub-microsecond range. However, this is made possible only through the use of dedicated hardware implemented inside high-end Ethernet NICs and switches. Further, both of these existing synchronization solutions fail to address the application-specific requirements of a data acquisition system. Specifically, the requirement is not to have accurate time on a hardware NIC or time card. Rather, the requirement is to have accurate time inside the user-mode application responsible for sampling experimental data and sending it across the network.

The remainder of this paper describes a method to build a data acquisition system using an Ethernet network and Microsoft Windows OS. Section 2 introduces a user-mode network stack that bypasses key sources of latency and jitter inherent in the Windows network stack. Section 3 describes a method of time synchronization that can run on generic Ethernet networks and provide highly accurate time at the user application level. Combining these solutions allows Ethernet networks to replace proprietary hardware in a prototype data acquisition system, as described and tested in Sections 4 and 5.

2. User-Mode Network Stack Architecture

The user-mode network stack was designed to reduce the network communication latency of software programs running on the Microsoft Windows operating system. This "fast-path" solution for TCP/UDP IPv4/IPv6 communication is implemented entirely in software and runs on commodity computer hardware. Programs that require accelerated network communication are transparently redirected to use this custom network stack through the use of replacement dynamic link libraries that intercept standard networking API calls. No program modification is required. A custom device driver redirects incoming packets into the user-mode stack, allowing accelerated data to bypass the Windows data path. Accelerated programs can co-exist with non-accelerated programs on the same computer system. Applications accelerated with this user-mode stack have communications with sufficiently low latency and jitter to enable them to be used for data acquisition and control systems.

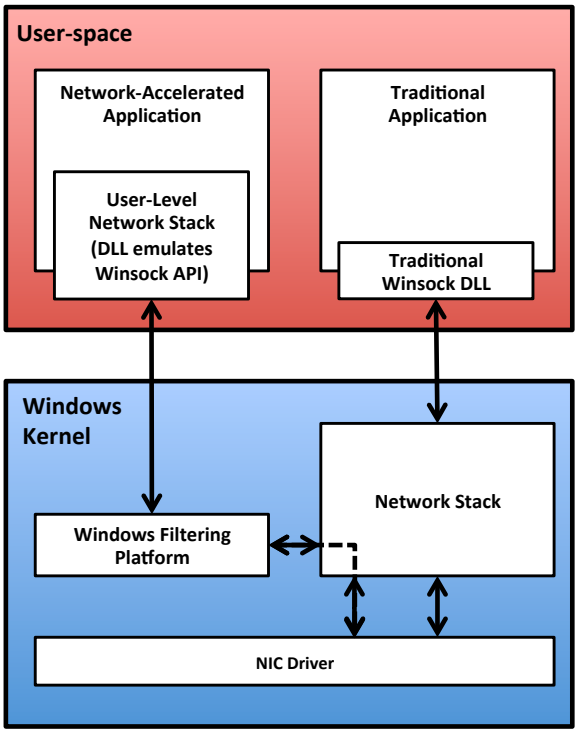

Figure 1: Architecture Comparison – User-Mode Network Stack versus traditional Winsock Network Stack

An overview of the network architecture is shown in Figure 1. An accelerated application using the user-mode stack is shown on the left half of the figure, and a traditional "Winsock"-based application is shown on the right. The "Winsock" (Windows Socket) API provides network functions for the Microsoft Windows platform, and all applications still use this traditional API. In the accelerated architecture, however, this API is implemented not by the real Microsoft-provided Winsock layer and network stack, but by the new user-mode network stack. This new stack communicates with a custom kernel driver (following the Windows Filtering Platform standard), allowing packets to be injected at a low level in the operating system, thereby bypassing most layers of the Microsoft network stack.

The key functional components of the architecture include a user-mode network stack, replacement dynamic link libraries, a Windows service, and a kernel driver. Figure 2 shows the detailed system architecture, including all key components and the communication between them, at

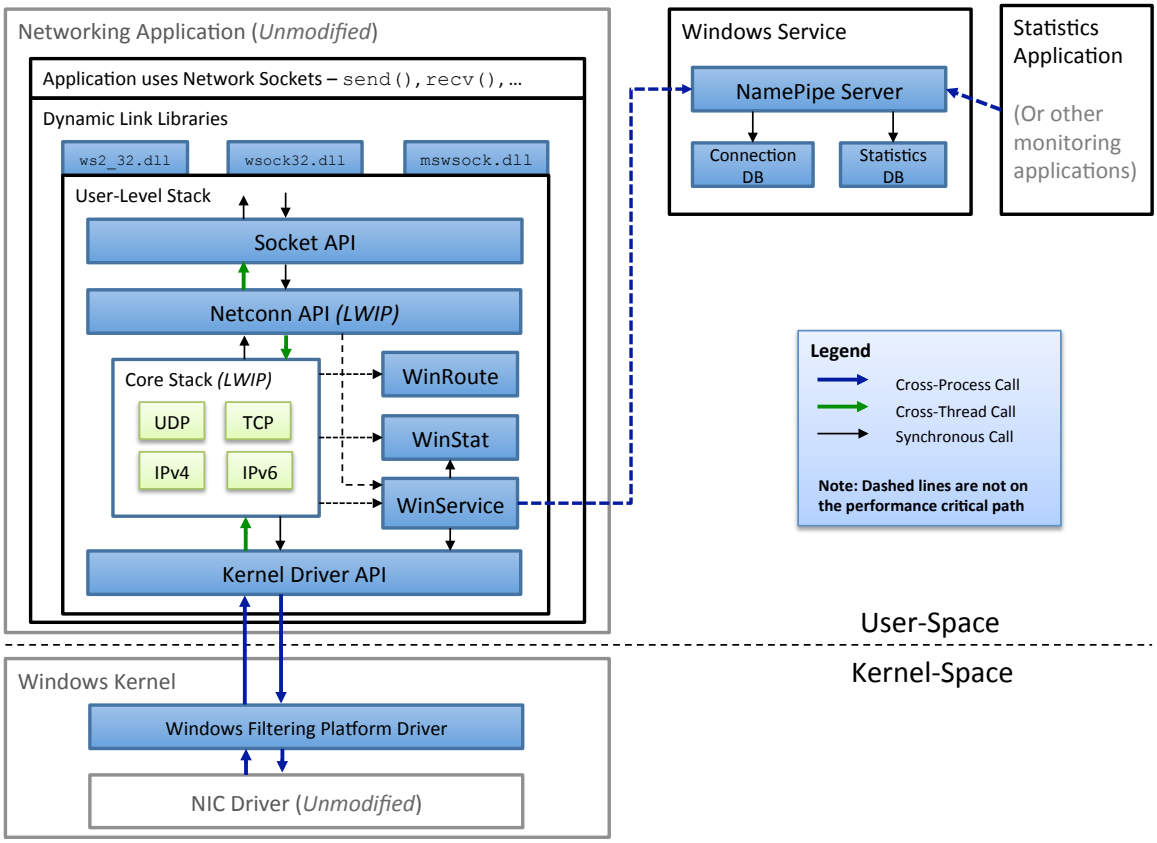

Figure 2: User-Mode Network Stack Architecture

both the user and kernel levels. These components are explained in the subsequent sections.

2.1. User-mode Network Stack

The user-mode network stack provides fast-path functionality for accelerated applications. As shown in Figure 2, the user-mode stack provides a "Socket API" layer that is a close replica of the Winsock API that is used by Windows applications. Implementation focused on the "Berkeley-style" socket functions, but also encompassed some of the important Winsock 2.0 API. Below the Socket API layer is the Netconn API layer. This layer represents internal stack functions and is tightly coupled with the core stack architecture, which is responsible for TCP and UDP processing with either IPv4 or IPv6 packets.

Elements from the Lightweight IP (LWIP) [8] project were used as the core of the user-mode network stack. LWIP provides, out of the box, a full network stack solution, including link layer (Ethernet), network layer (IP), and transport layer (TCP, UDP) protocols. In addition, it supports higher-layer protocols such as DNS (for domain name resolution), DHCP (for network address autoconfiguration), ARP (for resolving Ethernet addresses), and more. Running the full LWIP stack produces an independent network entity on the Windows host system, similar in effect to running a virtualized operating system on top of a host operating system. While this design is useful for embedded devices where no operating system is present, it is unnecessary when running a user-mode stack on top of Windows. In fact, it greatly complicates system administration, because a single computer now has two IP addresses, two MAC addresses, two DNS clients, two DHCP clients, and more. Further, these LWIP variants are hidden from the standard Windows management tools, potentially causing user confusion.

To simplify the overall system architecture, LWIP was modified to tightly couple with the host Windows operating system in a variety of places that are not on the performance critical path. This is the "WinRoute" box shown in Figure 2. For example, instead of initiating and processing its own ARP requests, DNS lookups, DHCP autoconfiguration, and more, LWIP defers to Windows by directly calling the relevant Windows APIs. Routing, specifically the selection of the best outgoing NIC and source IP address to reach a given destination, is also performed by Windows. It is possible to run all of these operations in Windows while still maintaining low latency because these tasks occur infrequently. DHCP configuration is done at system initialization and is not specifically relevant to the target application, while DNS lookups, ARP lookups, and routing are done when a socket is opened and a connection established. As such, LWIP does not request Windows services on a per-packet basis. The user-mode stack shares the same IP and MAC addresses as the Windows stack,

simplifying administration and configuration.

The network stack core also keeps a set of internal performance counters of key packet events, as labeled by the "WinStat" box in Figure 2. Statistics include counters for the number of IP, UDP, and TCP packets processed, number of currently active TCP connections, number of TCP retransmissions, and other counters that are typically reported by a system `netstat` application.

2.2. Replacement Dynamic Link Libraries (DLL)

One important design goal is to transparently accelerate network operations for selected applications without modifying application source code. Indeed, for some legacy applications, source code may not be available, or the application may not be easily recompiled.

To accomplish this, the user-mode network stack preserves key elements of the existing Winsock API that provides network functions for the Microsoft Windows platform. Programs do not implement the Winsock API directly. Rather, they link against Microsoft-provided DLLs such as `ws2_32`, `wsock32`, and `mswsock`. When a program is started, the necessary DLLs are retrieved from disk and loaded into memory in the program's address space.

As shown in the user-mode network stack architecture (Figure 2), the Winsock DLLs are replaced in memory – for just the targeted program, not the entire computer system – with custom DLLs. These custom DLLs export the same function calls as the Microsoft API. Internally, however, they forward calls into the user-mode network stack, thereby allowing an application to be transparently redirected. Not every function in the Winsock API has been implemented. Implementation focused on classic "Berkeley-style" functions that are widely used, with proprietary Winsock-only functions being implemented on an as-needed basis for specific application support.

2.3. Windows Service

The background Windows service provided with the user-mode stack facilitates better integration with non-accelerated programs running on the same computer. For example, it ensures that non-accelerated and accelerated programs cannot both reserve the same network ports at the same time. This is critically important because both the Windows stack and the user-mode stack share the same MAC addresses and IP addresses. In addition to coordinating port reservations, the Windows service also provides access to the internal stack performance counters for debugging and performance analysis purposes. A small standalone program has been provided to read these values, and a generic interface allows other third-party programs to access the same statistics.

2.4. Filter Driver

This Kernel-Mode Driver Framework (KMDF) Windows Filtering Platform (WFP) [9] filter driver ensures that packets destined for the user-mode network stack are not seen by the standard Windows network stack, and vice versa. The driver intercepts packets at the lowest level of the WFP Filter Engine, thereby bypassing much of the traditional Windows stack. Figure 2 depicts the WFP kernel-mode filter driver architecture and how it fits into the replacement user-mode TCP/IP stack.

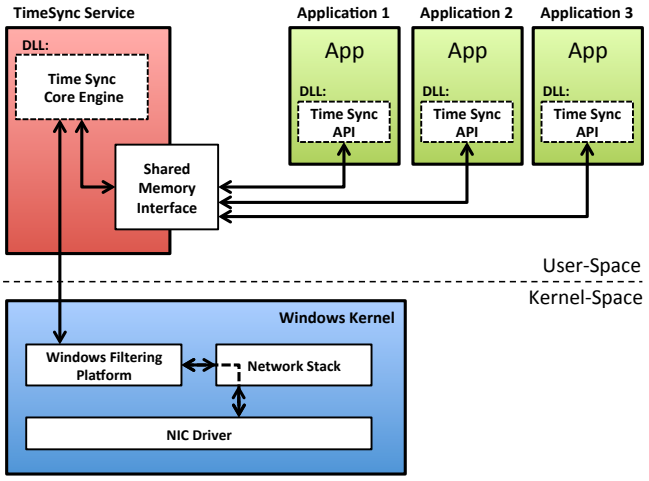

Figure 3: Time Synchronization Architecture

3. Time Synchronization Service Architecture

A network-based software clock with microsecond resolution at the user-mode software level was also produced to aid in data acquisition systems. Application programs using this system can obtain a network-synchronized timestamp with extremely low latency without requiring either expensive hardware timecards or even a relatively slow or jitter-prone context switch into the operating system. This system is compatible with Microsoft Windows and the NTPv4 standard, requires no hardware support[2], and allows clock synchronization to run over a commodity Ethernet network shared with application data, thereby significantly reducing the cost of deployment. The network-based software clock provides timestamps within 10 microseconds of the network reference clock.

The key functional components of the time synchronization system include a central Windows time service, an API that allows access to this unified time source from all user-mode applications without the time delay of a system call, and a Windows device driver that identifies time synchronization packets in the network stack and bypasses key sources of delay in the operating system. The interaction of these components is shown in Figure 3.

3.1. Time Synchronization Service

All key time synchronization algorithms have been centralized into a single service that will provide a common

[2]CPU support for the Time Stamp Counter (TSC) is required and provided in all modern Intel and AMD 64-bit processors

Table 1: User-Mode Stack Testbed Data Pattern and Per-Message Maximum Allowed Latency

	Message Type 0 (TCP)			Message Type 1 (TCP)			Message Type 2 (UDP)			Message Type 3 (Low-latency TCP)		
Data Producer	**Size (KB)**	**Freq (ms)**	**Latency (ms)**	**Size (KB)**	**Freq (ms)**	**Latency (ms)**	**Size (KB)**	**Freq (ms)**	**Latency (ms)**	**Size (KB)**	**Freq (ms)**	**Latency (ms)**
1	0.5	200	**100**	0.5	200	**100**	1	200	**100**	0.01	200	**10**
2	320	80	**40**	640	80	**40**	4	80	**40**	0.1	10	**10**
3	0.75	100	**50**	1	100	**50**	2.5	100	**50**	0.1	100	**10**
4	8	100	**50**	8	100	**50**	3.2	100	**50**	0.1	100	**10**
5	4	100	**50**	4	100	**50**	1.6	100	**50**	0.1	100	**10**
6	20	100	**50**	20	100	**50**	4	100	**50**	0.1	100	**10**
7	10	100	**50**	10	100	**50**	2	100	**50**	0.1	100	**10**
8	0.2	250	**125**	0.2	250	**125**	0.4	250	**125**	0.01	250	**10**
9	0.4	500	**250**	0.4	500	**250**	0.8	500	**250**	0.01	500	**10**
10	0.4	100	**50**	0.4	100	**50**	0.8	100	**50**	0.1	100	**10**
11	0.25	100	**50**	0.25	100	**50**	0.5	100	**50**	0.1	100	**10**
12	150	100	**50**	150	100	**50**	4	100	**50**	0.1	100	**10**

time source for all applications on a computer. The operation of the time synchronization system is as follows. The processor Time Stamp Counter (TSC) is employed for internal timekeeping, as it is the most stable clock available in modern computers [10]. Further, it is accessible with a single processor instruction and does not require a context switch into the operating system. The TSC only provides an incrementing counter, however. To convert that counter to a time-stamp, two additional pieces of information are needed: the offset between the counter and some known reference time, and the rate at which the TSC increments. Both of these pieces of information can be obtained by sending a series of modified NTPv4 queries to a network time server. The modification is that instead of sending request messages with NTP-style time-stamps, the request messages contain raw TSC values instead. As the NTP server simply echoes the request time-stamp back to the requester for processing, this modification is transparent to the server. Filtering algorithms select time-stamps with the lowest network latency, and this information is used to save a tuple of the NTP reference clock (wall time), local TSC counter value, and the local TSC tick rate. When a new time-stamp is required, a new TSC value is obtained from the processor, and used to calculate an offset to the previously saved reference time. Synchronization can continue periodically to account for any drift in the TSC. The resulting data is stored in a single-producer, multiple-consumer data structure in a shared memory location accessible to other applications running on the same computer. Client applications access this data by means of an API, described next.

3.2. Time Synchronization API

An Application Programmer Interface (API) is provided that allows any application to obtain time-stamps by loading a DLL and making a simple function call. Behind the scenes, the API interfaces via shared memory with the time service, obtains the last known reference time (global clock time plus the local processor TSC value and tick rate), offsets the clock time based on the current TSC value, and returns the time-stamp in NTP format to the user. This API is accessible directly from user-mode applications with no latency and jitter-inducing system call required.

3.3. Filter Driver

A Windows Filtering Platform driver improves the synchronization accuracy of the system by bypassing part of the operating system and the socket interface to the user application, thereby reducing latency and jitter. Time-stamps (specifically, TSC values) are added to outgoing NTP request packets late in the network stack processing, and time-stamps of received NTP reply packets are recorded early in the network stack processing. This improves the overall quality of the synchronization solution.

4. User-Mode Stack Testbed

To test the effectiveness of the developed software at delivering low end-to-end network delay at the software application program interface level, a simulated data acquisition and processing system was created in the laboratory, and modeled after real data acquisition systems in use today. In this abstracted view, multiple senders transmit streams of data – at varying data rates and message sizes – to a single receiver over a commodity Ethernet network. The latency of each individual message is measured by means of a hardware timecard synchronized to a master timeserver via IEEE 1588-2008.

The specifications of the data pattern produced by the experimental system are shown in Table 1. In this data pattern, there are 12 data producers. Each producer sends out four different types of messages using either TCP or UDP, representing data flows from different applications on the same producer. The maximum allowable latency for each message type, from each data producer, is shown

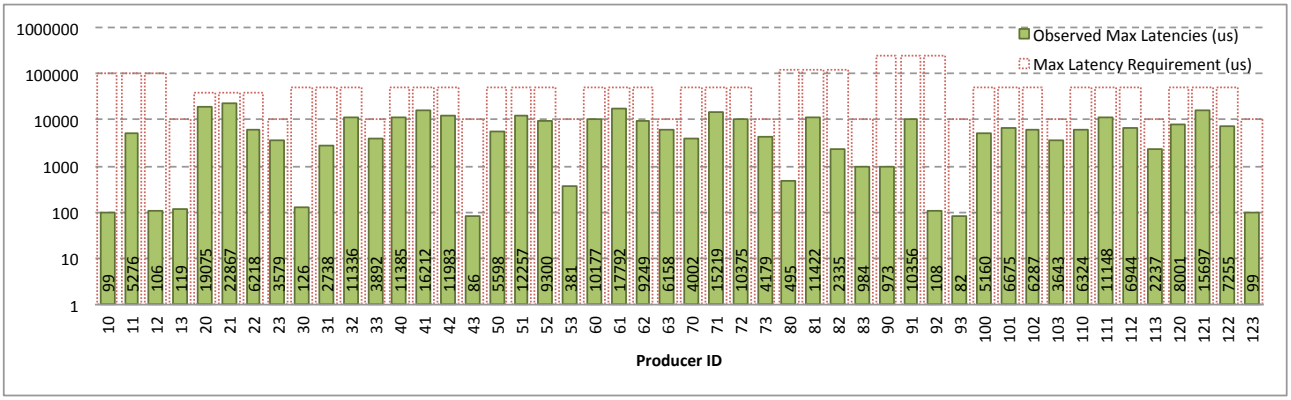

Figure 4: Per-Stream Maximum Observed Latencies for User-Mode Stack

in the table. Because there are 12 transmitters, there can be no more than 12 simultaneous transmissions hitting a single Ethernet switch with a highest load of using the largest packet from each of the 12 senders. The data consumer (receiver) would therefore see all transmissions, but in a serialized fashion, coming out of the Ethernet switch.

To uniquely identify per-stream metrics and map stream performance to the corresponding requirements, the producer streams have been encoded to a unique identifier. Each of the Data Producers numbered one through twelve is responsible for four socket streams and have been given base zero number mappings. This number is appended to the Data Producer number to create a unique identifier referred to as the "Producer ID". For example, the ID for "Producer 3 – Message Type 1, TCP Stream" is 31.

Figure 4 shows the performance of the user-mode stack when measured with the testbed. The stack is running on a single computer and receiving messages from the 12 data producer streams. The elapsed message latency is measured, and the maximum latencies seen for each individual message stream are recorded. The user-mode stack meets the network latency targets under conditions that simulate a realistic data acquisition and control system with many parallel data flows. Additional performance optimization is planned in order to improve the system further.

5. Time Synchronization Testbed

A lightweight test program was written to measure the performance of the time synchronization system. This program is run on a computer with a modern Intel Core i5-class CPU, and sends NTP requests via gigabit Ethernet to a hardware Meinberg time server that is synchronized against GPS. For measurement purposes only, a Meinberg PCIe hardware timecard was also installed in the test computer and configured to synchronize over a separate Ethernet network via IEEE 1588-2008 with the same hardware time server. Published documentation on this configuration indicates that the timecard provides a reference time accurate to within ±20ns of the network time server.

Table 2: Synchronization Performance at Varying Sync Intervals

Sync Rate (sec)	**Iterations**	**API Time to Ref. Time**		
		Min (μs)	**Max** (μs)	**Avg** (μs)
30	8 million	-5.23	3.80	-1.52
30	32 million	-6.65	4.11	-2.25
60	4 million	-9.33	5.75	-2.06
120	4 million	-8.54	1.86	-3.34
240	4 million	-8.95	12.21	-2.87
480	4 million	-34.88	27.88	-5.58
960	4 million	-3.86	39.39	11.16
No sync	32 million	-4.37	1700.99	596.03

The test program obtains a time-stamp first from the time service via the API, then from the Meinberg timecard, and then sleeps for a small random delay. This cycle then repeats for a configurable number of iterations. A "positive" offset between the API time and reference time provided by the timecard represents the anticipated time sequence, where time should be increasing between the first and second time-stamps obtained. A "negative" time represents the opposite ordering.

5.1. Results

Table 2 first shows the performance of the time service in a test for 8 million measurement iterations, with the time service resynchronizing with the time server every 30 seconds. In this standard configuration, performance was excellent. The minimum recorded difference between the API time and the hardware reference time was -5.23μs (indicating the API time-stamp was later in time despite being obtained earlier in software), and the maximum difference was 3.80μs (indicating that the API time-stamp was earlier in time, as expected from program design). The average difference was -1.52μs. Figure 5 shows a histogram of this same configuration, detailing the elapsed time between the first API time-stamp and the Meinberg timecard for 8 million measurements. The dashed blue vertical lines represent the minimum and maximum time offset recorded in the data set.

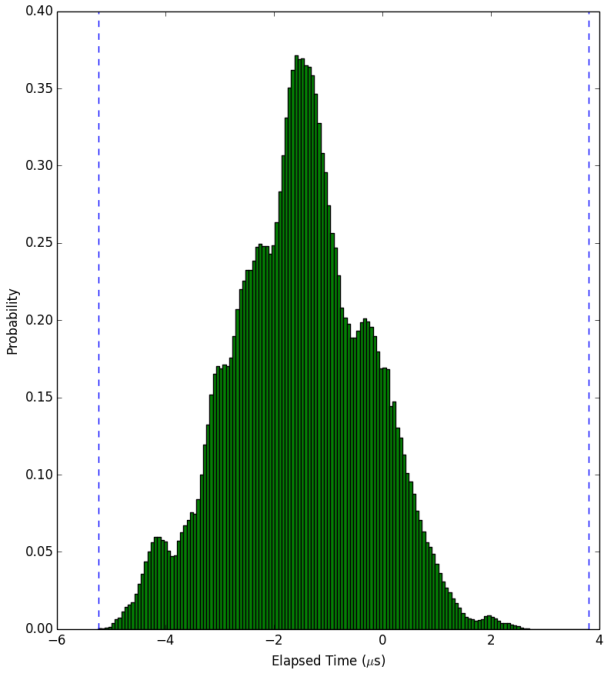

Figure 5: Histogram of Elapsed Time Between Time Service API and Hardware Timecard

Table 2 also shows the performance of the time service in two long-running tests that lasted for 4+ hours, totaling 32 million iterations. In the first test, the time service synchronized with the network time server every 30 seconds. In this standard configuration, performance was excellent. The minimum recorded difference between the API time and the hardware reference time was -6.65μs, the maximum difference was 4.11μs, and the average difference was -2.25μs. Other configurations in the table show the impact of reducing the synchronization interval from 30 seconds to 60, 120, 240, 480, and 960 seconds, respectively. As shown, the default synchronization interval of 30 seconds is slightly aggressive, and a relaxed interval of 120 or even 240 seconds between synchronization attempts would still provide an excellent quality of results within $\pm$12μs of the reference time.

In the final configuration labeled "No Sync", the time service synchronized with the network time server for the first 5 minutes of the test. At that point, the service was paused, and the test continued for the remaining 4+ hours with no further software communication with the reference time source. (The reference hardware timecard continued to synchronize with the time server). This configuration places significant pressure on the quality of the TSC rate measurement, which is used to convert the absolute difference in TSC counter values into measurement of elapsed time. Here, the minimum recorded difference between the API time and the hardware reference time was -4.37μs, but the maximum difference was 1700.99μs, which is a clock skew of approximately 0.1 microseconds per second during test operation. This shows the importance of providing a consistently-available network time source for the synchronization application, and also an area of potential future work in refining the TSC rate estimate to reduce skew in free-running operation.

6. Related Work

Several commercial products exist that provide real-time capabilities to Microsoft Windows, including networking, by using a partitioning approach. Examples include IntervalZero's RTX64 for Windows [11, 12] and TenAsys' INtime for Windows [13]. In these systems, a custom hard real-time operating system (RTOS) runs alongside the Windows kernel and conventional software stack, reserving one or more CPU cores for its exclusive use. Applications requiring real-time service make network and system API calls not into the regular Windows kernel, but into facsimile APIs provided by the RTOS instead. Other applications continue to use the Windows kernel. Applications may need to be modified to take advantage of RTOS capabilities. In addition, these architectures require a dedicated network interface for use exclusively by the RTOS, and thus hardware support is limited to devices supported by the RTOS manufacturer [14].

Microsoft has provided additional programming APIs in recent years to reduce the latency of network communication in Windows for applications such as high-frequency trading. One such API is the Registered I/O (RIO) framework added in Windows 8 / Server 2012 [15]. This API requires programs to be modified, sometimes significantly, in order to take advantage of the new capabilities.

User-level networking has a lengthy history in the high-performance computing field. The VIA (Virtual Interface Architecture) and its successor, InfiniBand, allow applications secure, zero-copy access to the network interface card for communication. The goal is to remove the operating system from the communication path, and reduce context switching, data-copy, and protocol overheads. As a result of these architectural changes, applications must use a new programming API for communication. To provide a smoother transition for legacy software, the SOVIA (Sockets Over VIA) [16] and Sockets Direct Protocol (SDP) [17] for Infiniband allowed existing socket-based applications to use the new underlying network architecture by providing a user-mode translation layer.

The processor Time Stamp Counter (TSC) has previously been identified as a highly stable oscillator with a resolution equal to the underlying CPU clock rate and drift of under 0.1 parts per million [10]. The TSCclock [18] and TSC-RCSP [19] systems provides similar LAN clock synchronization performance to our architecture – under 10μs – albeit for the Linux or FreeBSD operating systems, not Windows.

7. Conclusions

This paper presented a method for building data acquisition systems using flexible and cost-friendly Ethernet networks and non real-time computer operating systems (OS) such as Microsoft Windows. To remedy the random and significant latencies that such an architecture can produce, a new user-mode network stack and network-based software clock were developed, thus enabling real-time processing across multiple computers.

The network-based software clock provides time-stamps, at the software API level of any Windows program, that are within 10μs of a centralized NTP server on the local area network. The software-only design eliminates the need for dedicated hardware timecards or separate time synchronization networks, both of which reduce installation cost and complexity.

The user-mode network stack reduces the network latency for unmodified Windows applications that use common Winsock networking functions and standard IPv4 / IPv6 and TCP/ UDP network protocols. A kernel driver bypasses latency/jitter in Windows network stack by diverting packets to user-mode software. When tested against a complex data pattern that represents different customer systems in use today, all latency targets were met, and small messages were delivered with sub-millisecond latency. The user-mode network stack is a software-only solution, which minimizes deployment costs and complexity. It can transparently accelerate unmodified applications that use common Winsock networking functions without the need to modify application source code, which also minimizes deployment cost. Only the desired applications are accelerated, while other non-performance-critical applications continue to use the existing Windows network stack, providing deployment flexibility. The network stack shares the same network addresses as Windows and requires only the standard administrator tools to configure, simplifying deployment. Administrator access is not required to run the accelerated network stack, simplifying security considerations.

References

[1] G. Patterson, Data Validation in the AEDC Engine Test Facility (Feb 2010).
URL http://oai.dtic.mil/oai/oai?verb=getRecord&metadataPrefix=html&identifier=ADA514616

[2] IEEE standard for a precision clock synchronization protocol for networked measurement and control systems (IEEE 1588-2008), IEC 61588:2009(E)doi:10.1109/IEEESTD.2009.4839002.

[3] R. Zarick, M. Hagen, R. Bartos, Transparent clocks vs. enterprise ethernet switches, in: Proceedings of the 2011 International IEEE Symposium on Precision Clock Synchronization for Measurement Control and Communication, ISPCS '11, 2011. doi:10.1109/ISPCS.2011.6070161.

[4] D. Ingram, P. Schaub, D. Campbell, R. Taylor, Evaluation of precision time synchronisation methods for substation applications, in: Proceedings of the 2012 International IEEE Symposium on Precision Clock Synchronization for Measurement Control and Communication, ISPCS '12, 2012. doi:10.1109/ISPCS.2012.6336630.

[5] J. Loeser, H. Haertig, Low-latency hard real-time communication over switched Ethernet, in: Proceedings of the 16th Euromicro Conference on Real-Time Systems, ECRTS '04, 2004. doi:10.1109/EMRTS.2004.1310992.

[6] S. Debbarma, A. Das, Empirical measuring IPv4/IPv6 network performance on Microsoft Windows operating systems, in: Third International Conference on Advances in Computing and Communications, ICACC '13, 2013, pp. 393–395. doi:10.1109/ICACC.2013.83.

[7] D. Terhell, Windows and Real-Time, in: The NT Insider, OSR Open Systems Resources, Inc, 2014.
URL https://www.osr.com/nt-insider/2014-issue3/windows-real-time/

[8] A. Dunkels, lwIP - A Lightweight TCP/IP stack (accessed February 1, 2015).
URL http://savannah.nongnu.org/projects/lwip/

[9] Microsoft, Windows Filtering Platform (WFP) (accessed February 1, 2015).
URL https://msdn.microsoft.com/

[10] A. Pásztor, D. Veitch, PC based precision timing without GPS, in: Proceedings of the 2002 ACM International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '02, ACM, New York, NY, USA, 2002. doi:10.1145/511334.511336.

[11] IntervalZero, RTX64: A Powerful, Flexible Platform for Industrial Vision Control Systems (2014 - accessed February 1, 2015).
URL http://intervalzero.com/assets/08-14Motion-Directed-Vision-White-Paper.pdf

[12] IntervalZero, Transforming 64-Bit Windows to Deliver Software-Only Real-Time Performance (2014 - accessed February 1, 2015).
URL http://intervalzero.com/assets/2014-04-01-RTX64-White-Paper.pdf

[13] TenAsys, INtime for Windows (accessed February 1, 2015).
URL http://www.tenasys.com/tenasys-products/intime-rtos-family/intime-for-windows

[14] TenAsys, Real-time Networking (accessed February 1, 2015).
URL http://www.tenasys.com/tenasys-products/intime-technology/networking

[15] Microsoft, Registered Input/Output (RIO) API Extensions (2012 - accessed February 1, 2015).
URL https://technet.microsoft.com/en-us/library/hh997032.aspx

[16] J.-S. Kim, K. Kim, S.-I. Jung, SOVIA: A user-level sockets layer over virtual interface architecture, in: Proceedings of the 3rd IEEE International Conference on Cluster Computing, CLUSTER '01, IEEE Computer Society, Washington, DC, USA, 2001.

[17] P. Balaji, S. Narravula, K. Vaidyanathan, S. Krishnamoorthy, J. Wu, D. Panda, Sockets direct protocol over InfiniBand in clusters: is it beneficial?, in: Proceedings of the 2004 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS '04, 2004. doi:10.1109/ISPASS.2004.1291353.

[18] J. Ridoux, D. Veitch, Ten microseconds over LAN, for free, IEEE Transactions on Instrumentation and Measurement Vol 58 (Issue 6). doi:10.1109/TIM.2009.2013653.

[19] G.-S. Tian, Y.-C. Tian, C. Fidge, Precise relative clock synchronization for distributed control using TSC registers, Journal of Network and Computer Applications Vol 44. doi:10.1016/j.jnca.2014.04.013.