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

Jeffrey Shafer, Jesse Conn, Cameron Lucas, James Caffery, Klaus Schug

61st International Instrumentation Symposium

May 12, 2015

Data Acquisition & Control Systems

Industrial Scale

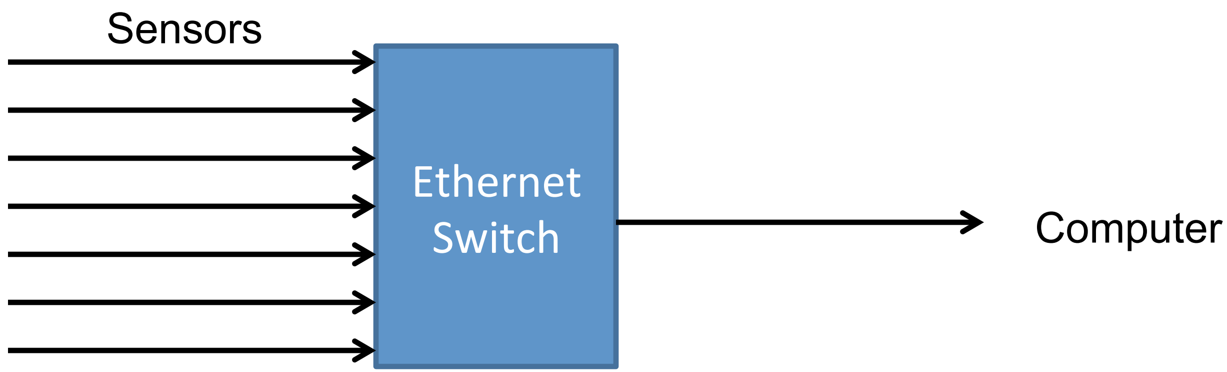
Data Acquisition & Control Systems

Problem Statement

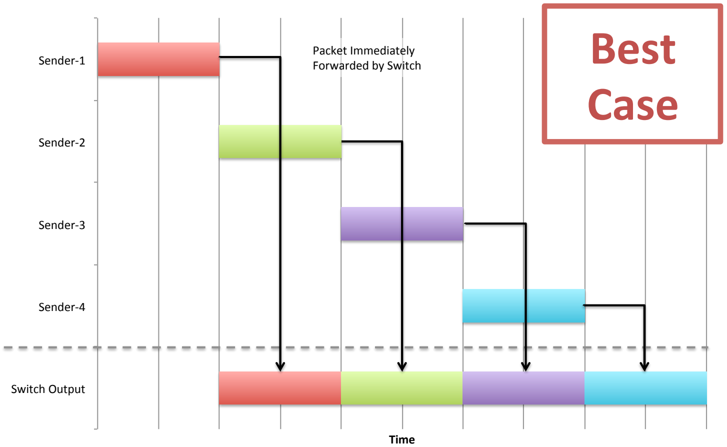
- **Data Acquisition and Control Systems**
  - Built from many data producers (sensors), data consumers (computers), and **networks**
- Requirements
  - **Synchronized time** across data network for correlation of events from different data producers
  - **Deterministic & low latency** end-to-end data flow; enabling real-time processing for control system
- *Examined the networking and data consumer (computer) side of the problem*
- *How to implement requirements in* <u>*cost-effective*</u> *manner?*

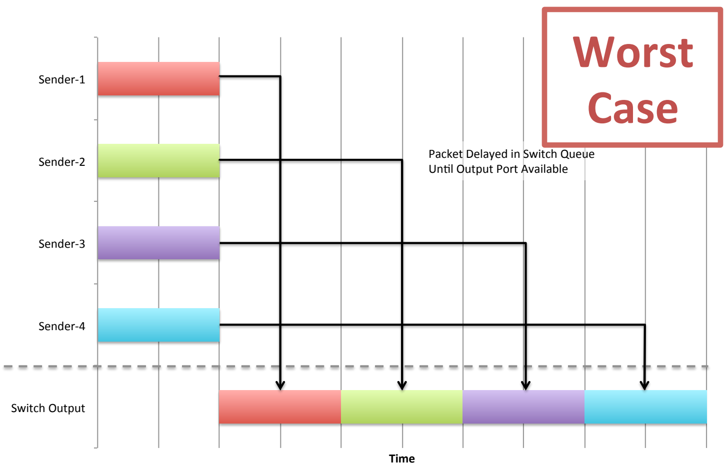
Observation

- Gigabit Ethernet is low latency…
  - 1kB payload: 12μs at 1GbE and 1.2μs at 10GbE
- … *provided* that **congestion** (**queuing**) are engineered out of the system



Observation





- **Best case**: staggered message transmission avoids queuing delay
- **Worst case**: synchronized message transmission maximizes queuing delays at Ethernet switch

Option 1 – Specialized Software

- **Real-Time Operating System (RTOS)**
- **Pros**: Predictable / bounded delays in software enable high precision systems
- **Cons**:
  - Proprietary, or
  - Expensive, or
  - Require specialized development skills, or
  - Incompatible with existing applications, or
  - *All of the above*
- Not all applications require the highest levels of performance that RTOS can provide

Option 2 – Specialized Hardware

- **General-purpose computers** running Microsoft Windows and custom applications
- **Data distribution**
  - Ethernet NIC with **RDMA** capabilities
    - **Pros:** Minimal latency (wire speed)
    - **Cons:** Proprietary, expensive, new programming API
  - **Reflective memory** network
    - **Pros**: Predictable latency
    - **Cons**: Proprietary, expensive, bandwidth constrained, new API
- **Time synchronization**
  - Use **hardware time cards** and **separate network** (GPS, IRIG, …)
  - **Pros**: Dedicated hardware provides high performance
  - **Cons**: Proprietary, expensive, second network to maintain

Design Challenge

- ***Can we build a data acquisition and control system with the following inexpensive components?***
  - Commodity operating systems (Microsoft Windows)
  - Commodity hardware
  - Commodity networks (specifically, a single Ethernet network for both time synchronization and test data)
- **Challenge**: Commodity technologies are optimized for high bandwidth, not low latency
  - True for both software and hardware
- Desired benefits
  - **Lower upfront and long-term cost (↓$)**
  - **Faster technology curve** for performance upgrades

NEW SOFTWARE-ONLY SOLUTIONS

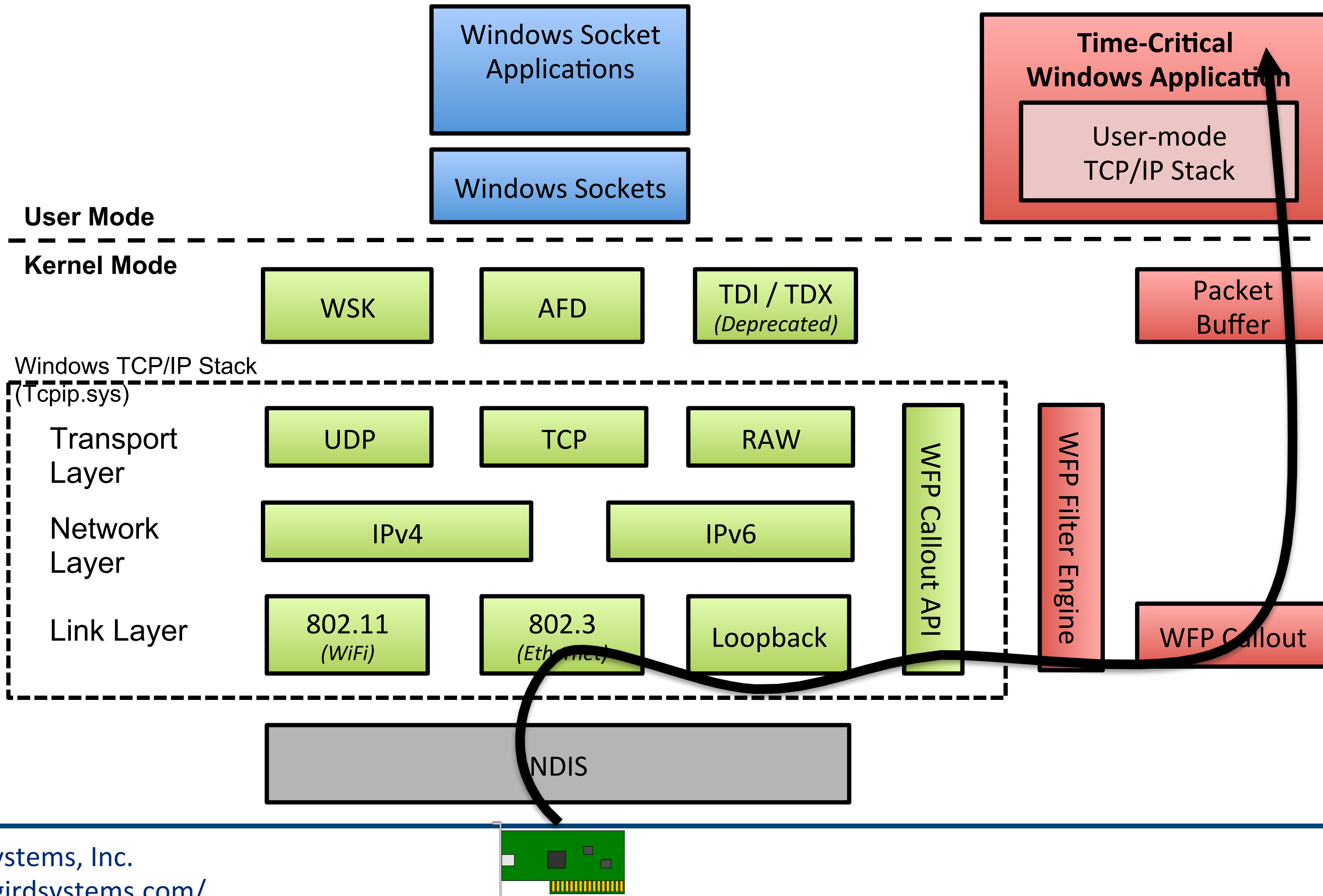
Solution 1: Software Time Sync Service

- **Software-only** Windows service provides **simple API** for application use ↑*Capability*
  - Common time source for all applications
- **Accurate** to ±10μs of central NTP server on LAN
- **Eliminates** need for **separate time network** ↓$
- **Eliminates** need for **hardware timecard** ↓$
- **Faster to obtain** than timecard ↑*Performance*
  - Does not require OS or hardware involvement
  - No context switch (and associated delay)

Software Time Sync Service

- Clock uses processor Time-Stamp Counter (TSC)
  - **Highest-resolution time** with low-latency access in user-mode without context switch ↑*Performance*
- Time sync service
  - Generates NTP packets with TSC timestamps
  - Selects "high quality" NTP responses
  - Rate synchronization / offset synchronization
- Custom *Windows Filtering Platform* kernel driver
  - Time-stamp packets in kernel-mode
  - Reduces round-trip time and removes largest sources of software latency & jitter) ↑*Performance*

Solution 2: Windows Network Stack Bypass



Windows Network Stack Bypass

- **Fast-path design** for performance-critical apps
  - Bypass majority of Windows network stack while preserving compatibility with all hardware NICs
  - **Software-only solution ↓$**
- **Transparently accelerates unmodified Windows applications** that use Winsock networking functions
  - No source code modifications needed
  - Only accelerates desired applications – other apps continue to use Windows TCP/IP stack ↑***Performance & Capability***
- Administrator access not required to run applications

User-Mode TCP/IP Stack

- Simplified design (only IPv4, IPv6, TCP, UDP)
- Built from open-source LWIP project
  - Portable w/ no OS dependencies, quick-start to project
- Added performance optimizations targeting modern x86-64 computer systems
- Integrated with Windows network stack
  - Shares same IP address and MAC address
  - Non-performance-critical tasks: DHCP, ARP, DNS, routing
    - Occur once per connection or at initialization
  - Consistent behavior with Windows simplifies administration

Windows Kernel Driver

- Bypasses latency/jitter sources in Windows TCP/IP stack (OSI model layers transport and above) ↑*Performance*
- Diverts **time-critical** *inbound* IP packets to user-mode stack
  – IP packets are diverted ONLY for connections the user-mode stack registers with the kernel driver
- Provides low-level injection point into Windows TCP/IP stack for *outbound* IP packets
- Implemented using *Windows Filtering Platform*
  – Callout driver is **NIC agnostic** (no custom hardware requirements) ↓$

Winsock Replacement DLLs

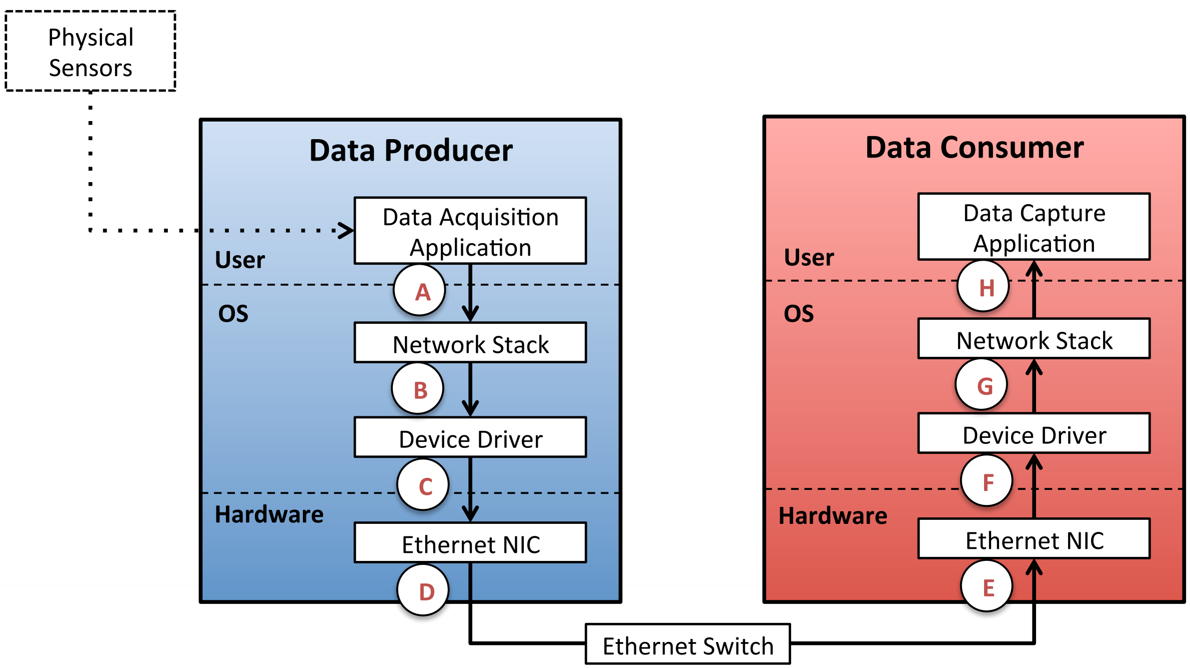
- Facilitates transparent compatibility with existing application binaries
- Built replacement DLLs for
  - `ws2_32.dll`
  - `wsock32.dll`
  - `mswsock.dll`

  (place DLLs in directory of application desiring to use user-mode stack)
- Provides same API as genuine Winsock DLLs, but redirects calls into user-mode stack
- Long list of *potential* API functions
  - Current implementation supports standard Berkeley socket API
  - Additional API functions added as needed for application compatibility

TESTING

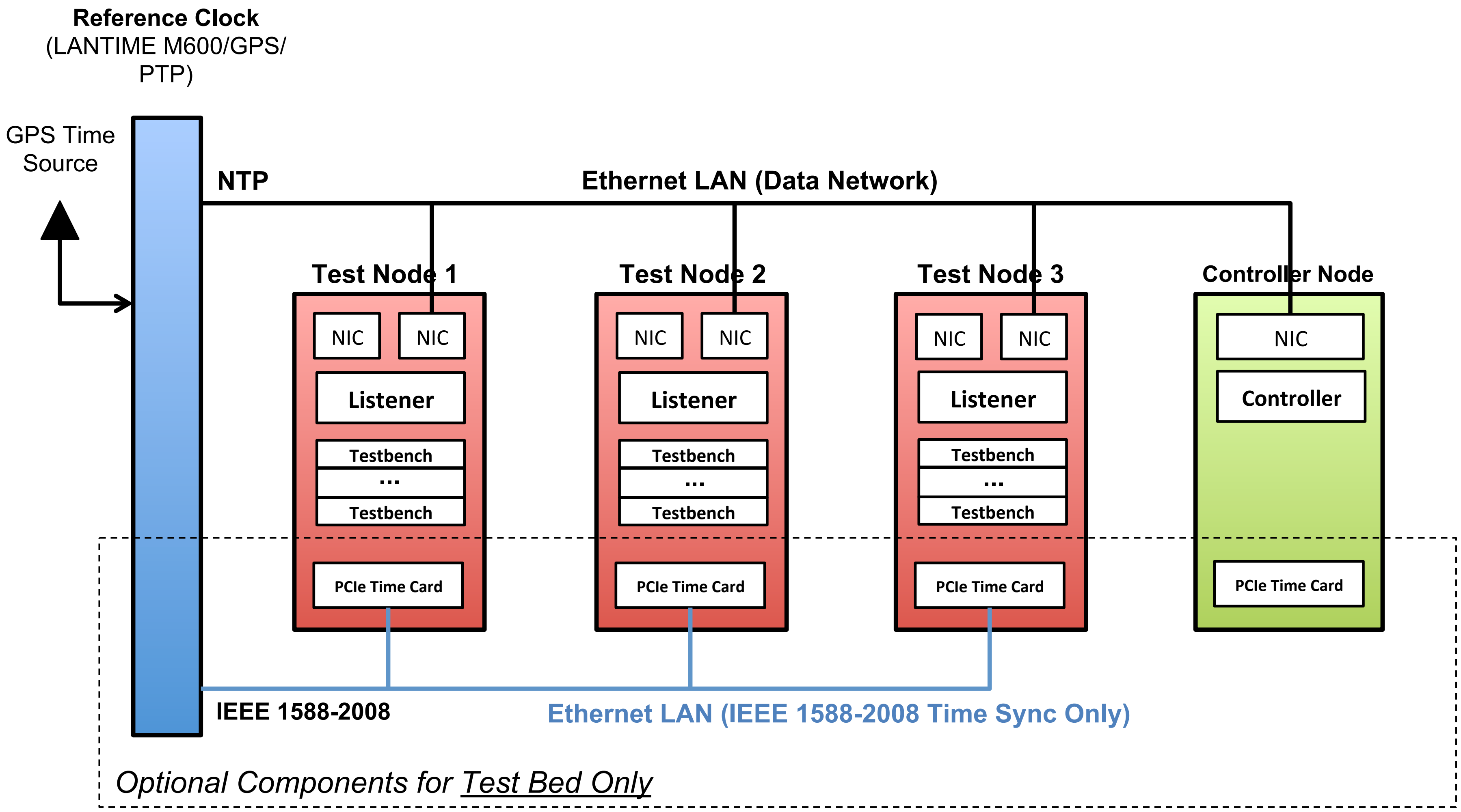
Test Bed

- Measures **end-to-end latency** from a producer computer to a consumer computer
  - **Application-level** to **application-level**
  - All (software/hardware) latency sources included

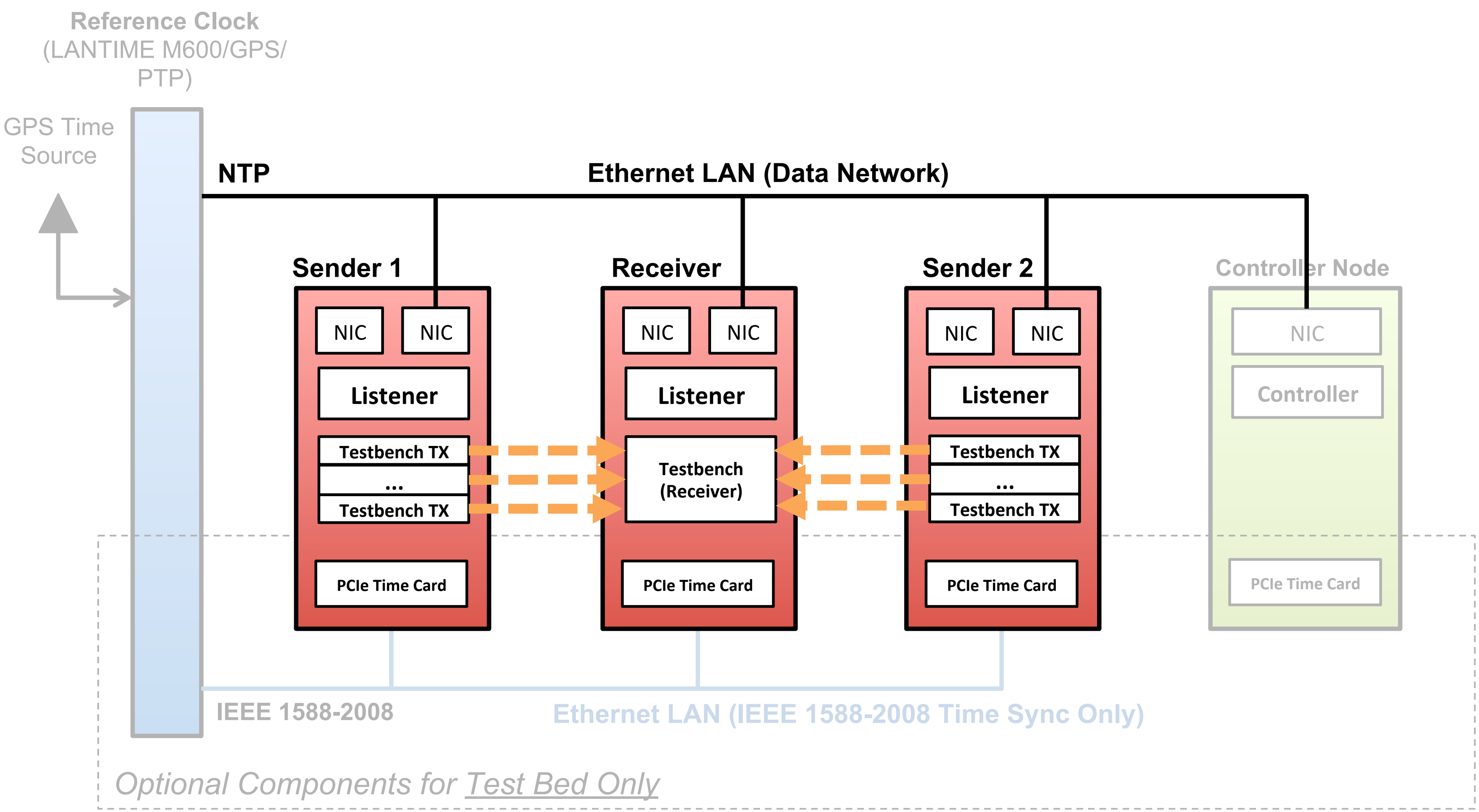


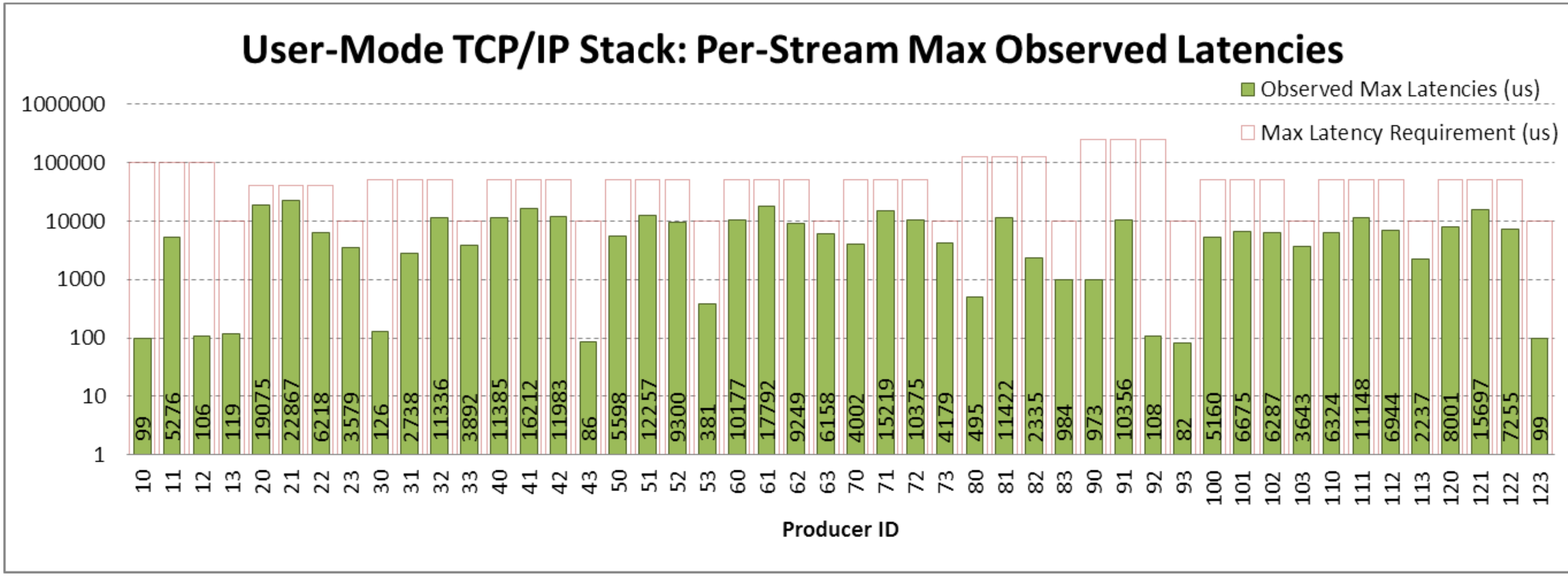
- Emulates *client-specific* **data pattern**
  - TCP and UDP
  - 48 types of messages
  - Date size varied from 100 bytes – 300kB
  - Data sampling rates varied from 10ms-100ms
  - Latency target varied from 10ms-100ms

Test Bed Architecture



Test Bed Architecture



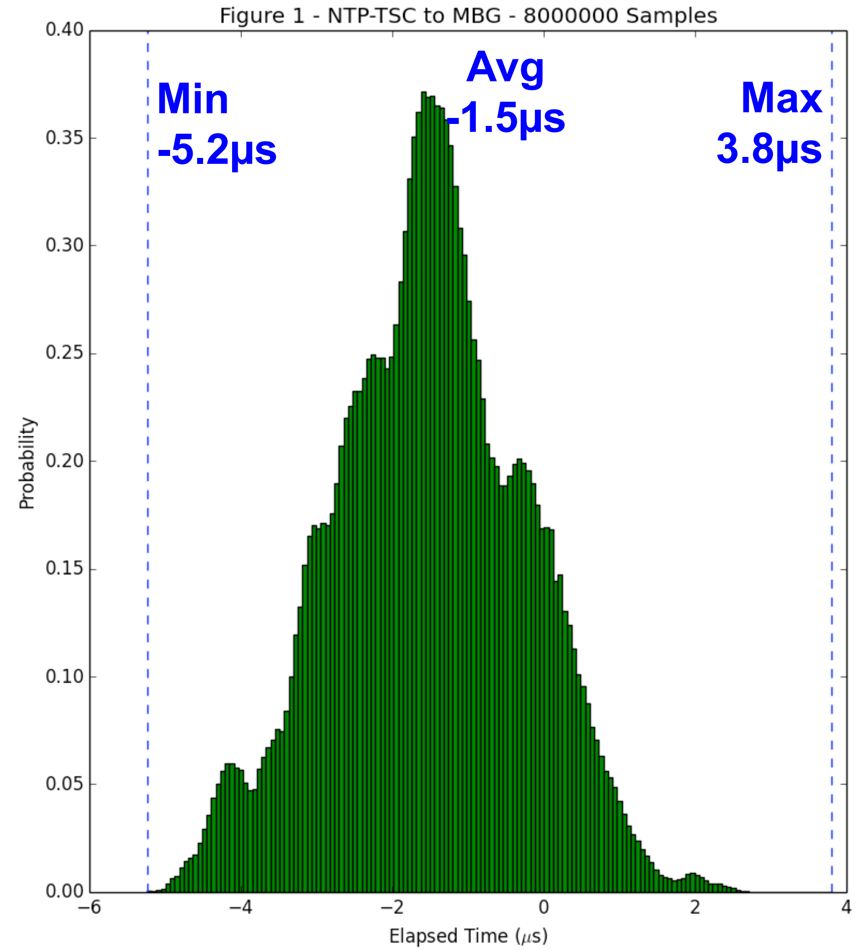


- Maximum latency observed - vs- client requirements
- All latency goals met

- Comparison: Windows had lower average latency but higher maximum latency for certain message types

Network Time Synchronization

- Back-to-back timestamps
  1. Software service (NTP over Ethernet)
  2. Hardware timecard (IEEE 1588, experimental control)
- 8 million samples, 4 hours
- Best performance when service synchronizes every 30s (could delay to every 4 minutes and remain < 10μs)
- Comparison: Traditional NTP performance on LAN is within 1ms (best case)



Summary

- Two software-only components for Microsoft Windows
- **Time Sync Service**
  - Low-latency access to time-stamps ±10μs of NTP server
  - Based on CPU Time Stamp Counter (TSC)
  - Eliminates need for hardware time-cards and separate network
- **Low-Latency User-mode TCP/IP Stack**
  - Transparently accelerates **unmodified applications**
  - Integrated with Windows to simplify administration
  - Kernel driver bypasses significant software sources of latency and jitter in Windows network stack

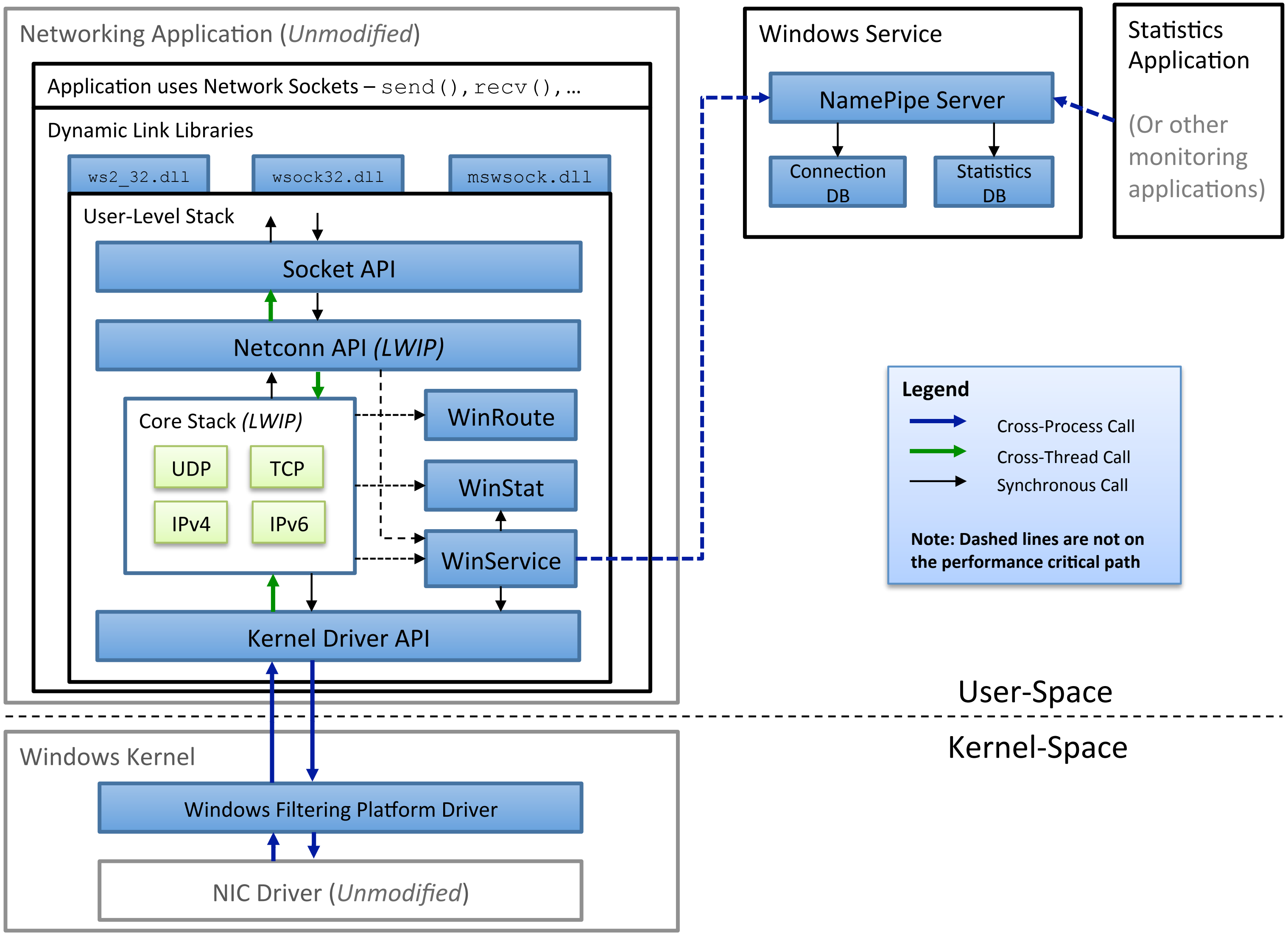
SUPPLEMENTAL SLIDES

GIRD SYSTEMS, INC.

UNIVERSITY OF THE PACIFIC

Client-Specific Data Pattern

	Message Type 0 (TCP)			Message Type 1 (TCP)			Message Type 2 (UDP Multicast)			Message Type 3 (Low-latency TCP)		
#	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**	0.10	10	**10**
3	0.75	100	**50**	1	100	**50**	2.5	100	**50**	0.10	100	**10**
4	8	100	**50**	8	100	**50**	3.2	100	**50**	0.10	100	**10**
5	4	100	**50**	4	100	**50**	1.6	100	**50**	0.10	100	**10**
6	20	100	**50**	20	100	**50**	4	100	**50**	0.10	100	**10**
7	10	100	**50**	10	100	**50**	2	100	**50**	0.10	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**	0.10	100	**10**
11	0.25	100	**50**	0.25	100	**50**	0.5	100	**50**	0.10	100	**10**
12	150	100	**50**	150	100	**50**	4	100	**50**	0.10	100	**10**


Networking Application (*Unmodified*)
Application uses Network Sockets – send(), recv(), ...
Dynamic Link Libraries
ws2_32.dll
wsock32.dll
mswsock.dll
User-Level Stack
Socket API
Netconn API *(LWIP)*
Core Stack *(LWIP)*
UDP
TCP
IPv4
IPv6
WinRoute
WinStat
WinService
Kernel Driver API
Windows Service
NamePipe Server
Connection DB
Statistics DB
Statistics Application
(Or other monitoring applications)
Legend
Cross-Process Call
Cross-Thread Call
Synchronous Call
Note: Dashed lines are not on the performance critical path
User-Space
Kernel-Space
Windows Kernel
Windows Filtering Platform Driver
NIC Driver (*Unmodified*)