# APPLICATIONS OF PMUSIMULATOR IN PDC TESTING

Philip M. Kersey

Thesis submitted to the faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science

In

Electrical Engineering

Virgilio A. Centeno, Chair

Jaime De La Reelopez

James S. Thorp

April 27, 2012

Blacksburg, Virginia

**Keywords:** Phasor Measurement Unit, Phasor Data Concentrator, Simulator

# Applications of PMUSimulator in PDC Testing

Philip M. Kersey

## Abstract

With the development of the power grid into an automated system, phasor measurement units and phasor data concentrators are essential for real time control of the system. PMUs are time synchronized throughout the power system and take sample measurements in very small windows of time. Phasor Data Concentrators accept PMU data and time align the data so that a snapshot of the power system can be viewed in real time. It is unfeasible to possess enough real PMUs to thoroughly test PDCs, thus a Real Time PMU Simulator is desired.

It is possible to implement a UNIX based PMU simulator that can emulate the behavior of real PMUs, while also allowing the user to alter the Synchrophasor data to test the response of a PDC. GPS is used to synchronize a UNIX machine to UTC time to match that of a real PMU. In this way, the PMU simulator will accurately behave as a PMU. This PMU data can be sent to PDCs to test the response of the device. To test extremes of the PDC, alterations were made to the PMU software to send irregular data to a PDC. The results conclude that the open source iPDC software is capable of being used for latency testing, sending late data frames, as well as sending corrupted data. The PMU simulator proved to be successful in the area of PDC testing. The purpose of this thesis is to demonstrate how the iPDC software can be implemented to test PDC's.

## Table of Contents

## Table of Figures

## List of Tables

# Chapter 1: Introduction

## 1.1 PMU History

In 1965, the North-Eastern power grid experienced a blackout that revealed the need to monitor the whole power system in real-time. With the limited capabilities of the time, a method was devised that used a sequential measurement scan with a non-linear state estimation scheme to obtain a rough sketch of the power system state. This method advanced the awareness of the state of the power system, but the information provided was not sufficient to significantly reduce the probability of blackouts.

In most power systems, techniques for state estimation which were developed in the 1970's involve measuring active and reactive power at a substation and sending that information to a central location to process all of the data. Due to the fact that the scan time for all of the measurements taken varied from seconds to minutes, the estimates of the system state inherit some limitations due to a large scanning window.

The state estimation setup assumes that all of the measurements are taken at the exact same time and sent to the centralized location instantaneously. In the 1970's, there was no way to determine precisely when a measurement was recorded so the central processing would have to assume the measurements were taken at the same time, even if they were taken seconds apart.

With IEDs and faster communications, the size of the scanning window has been reduced and it is reflected in faster and better updates from state estimators but this has reduced not eliminated the limitations due to the size of the scanning window. The solution to this problem is to have a way to time synchronize all substations so that a device can take a measurement and timestamp the data. Additionally, if the devices are time synchronized, it becomes desirable to decrease the sampling window to what is allowed by the communication channels. The data can

then be sent to a central location where the timestamps are aligned. This solution was made possible by the availability of affordable GPS receivers.

### 1.2 PMU Time Synchronization

In the early 1980's, communication technology improved and the first GPS satellites were launched into orbit, which made it possible to have accurate time synchronized devices connected to GPS receivers. This led to the development of Phasor Measurement Units (PMUs), which were developed at Virginia Tech for wide area monitoring through a common time reference. When the first PMUs were developed, there were not many GPS satellites in orbit so a GPS receiver would lose synchronization with a satellite quite often. As a result, GPS receivers needed extremely accurate crystal oscillators to sustain time when a satellite was not in view, which resulted in a very high price for GPS receivers. Now that the full constellation of satellites is installed, GPS receivers do not require expensive crystals and their cost have reduced drastically. In the case that the PMU GPS synchronization is lost, the PMU can also set a bit signaling that the GPS clock is out of synchronization, which would mean that the incoming measurements are inaccurate.

All Synchrophasor measurements are time tagged with the Coordinated Universal Time (UTC) time, via GPS. All GPS receivers generate a very accurate pulse (+-100 nanoseconds) every second that is used by PMUs to generate very precise sampling pulses. PMUs provide a more exact view of the system because samples are taken and time stamped in a microsecond sampling window and although data from multiple units can only be available and time aligned in the order of milliseconds the time stamp included with the measurements eliminates the errors introduced in the state estimator by the size of the scanning window.

### 1.3 Phasors

PMUs use data samples to compute phasors of the sinusoidal voltage and current measurements in real time and then time tagged them with the UTC time provided by the GPS receiver. The current Synchrophasor standard sets the timetag as the time that the computed phasor represents, which is typically around the middle of the sample window. Phasors are computed and transmitted in a data frame at a specific data or frame rate in the units of frames per second.

PMU data is sent to a central location to be processed. At this central location, a device called a Phasor Data Concentrator (PDC) accepts and time aligns data from several PMUs. After the PMUs data is aligned, the PDC will send an output data stream to another PDC or computer to be analyzed. With increasing numbers of PMUs added to the power system, the number of PDCs also increases. It then becomes imperative to test the behavior of these PDC's in order to understand how they will operate in stressed or unusual conditions.

PMUs can be used for better state estimation, protection schemes, and control systems. In recent years, the increasing computer technology has allowed most power system measurements, controls, and protection to be completed by computers that can take advantage of the real time information provided by PMUs.

### 1.4 IEEE C37.118.2-2011 Standard

PMUs use a Synchrophasor standard called IEEE C37.118.2-2011, which was developed to enhance the previous standard, IEEE C37.118-2005. The IEEE C37.118-2005 standard was a revision of the IEEE 1334-1995 standard that was the first effort to ensure PMUs from different manufacturers could be used on a common application. The C37.118.1-2011 presents the way to test the accuracy of existing PMUs. The C37.118.2 standard determines how the PMU information is communicated.

Synchrophasor messages, in the C37.118.2-2011 Standard, consist of a configuration, header, and command frame, as well as multiple data frames. Transmission of these frames occurs in the following way: A user who wants the Synchrophasor messages will send a command frame to the PMU or PDC requesting the data. The configuration and header frame are sent from the PMU to the user wanting to view the information. The configuration frame is machine readable and it describes the specifics of what is going to be sent from the PMU or PDC including calibration factors, number of phasors, analogs, as well as digital channels. The header frame is human readable and supplied by the user. After the configuration frame is sent, the PMU begins to create data frames by computing phasors and time stamping the measurements and then sending those data frames to the user to be interpreted.

### 1.5 The Need for a Real Time PMUSimulator

In a laboratory environment, due to the high cost of PMU's, it is extremely inefficient to possess as many PMUs as necessary to test the full functionality of PDCs. A solution was devised by implementing an open source Real Time PMUSimulator.

This thesis presents a method to use a PMU simulator to take the place of physical PMUs or enhanced the number of PMU messages for the purpose of testing phasor data concentrators. A true Real Time PMU simulator must provide not only valid PMU data frames, but these data frames must be synchronized to UTC just as a real PMU data. Effectively, a timetag is generated by the Real Time PMU simulator, which matches the timetag of a PMU at any instance in time within a few microseconds.

### 1.6 Thesis Content

This PMU simulator can be used for latency testing or programmed for sending late data frames or generating invalid data frame checksums. A real PMU is incapable of generating incorrect data on demand, as it will always attempt to function correctly. However, a PMU

simulator can be programmed to function however desired in order to test the response of a PDC. Chapter 1 provides a short history and introduction of the primary concepts of PMUs and PDCs and their relation to the power system. Chapter 2 presents how an existing PMU simulator was adapted to be used as the base for the Real Time PMU Simulator. Chapter 3 describes how the time synchronization was achieved to allow a Real Time "time tag" for the PMU simulator. Chapter 4 presents the results obtained and compares them to the responses obtained from commercial PMUs. Chapter 5 presents the conclusions and future work.

## Chapter 2: Simulating A PMU

A Phasor Data Concentrator (PDC) is a device that collects PMU data, time aligns the data, and sends it to a computer or another PDC. To properly test a PDC for latency, the maximum number of PMU inputs must be applied to the device. For most PDC's, the maximum number of PMUs is large and it is usually not available in a laboratory environment. The need of a large number of PMUs can be supplied by PMU simulators that provide data in the correct format and with a GPS sync valid timetag.

### 2.1 iPDC/PMUSimulator

In 2011, an open source PMU simulator was released on ipdc.codeplex.com. The software PMUSimulator is part of a larger package called iPDC, which also includes a PDC as well as a DBServer. It was created by Kedar Khandeparkar and Nitesh Pandit at the Department of Computer Engineering and IT College of Engineering in India. [13]

The iPDC software is compatible with the IEEE C37.118-2005 Standard and uses multithreaded programming. It also supports both Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) communication protocols. It has a Graphical User Interface to set up the system which makes it easy to use. The software PMUSimulator is open source which means that anyone can see exactly how the code is written and how it operates. If the user wants to change particular settings, he can alter the C code to function however it is desired. [13]

PMUSimulator works exactly like an actual PMU but without having to calculate the phasor measurements from an actual power system signal. While PMUSimulator is running, it receives a command frame from a client declaring that it wants the PMU to begin transmitting data. In response, PMUSimulator sends the configuration frame informing the client of all of the necessary information needed to interpret the data frames such as ID Code and number of phasor, analog, and digital channels. When using PMUSimulator as a server to iPDC, iPDC can

send a message to PMUSimulator commanding it to turn on data transmission. At this point, PMUSimulator receives the command and begins sending data frames to the client.

## 2.2. iPDC Version History

### 2.2.1 iPDC Version 1.0

The first version of iPDC was released in August 2011 and it allowed for randomly generated data to be sent via IEEE C37.118 messages. This version also allowed for multiple instances of PMUSimulator at the same time which allowed for at least twenty outputs to a PDC from a single computer. The only limit on the number of instances of PMUSimulator is the processing speed of the computer that it is running on. On an Intel Pentium 4 processor, running twenty instances of PMUSimulator was possible, but within an amount of time the computer (Intel Pentium 4) would lock up. Even with these limitations, it was a good starting point for a PMUSimulator. After initial testing, it was discovered that there were many limits to the original software. The first version had a checksum calculation error when more certain combinations of phasors, analogs, and digitals were used. Having more than three analog channels or adding any digital channels resulted in a checksum error. Checksum errors also occurred when updating the configuration frame or having a combination of analog and phasor values together. The creators of the program were notified of these problems.

### 2.2.2 iPDC Version 1.0.1

The creators of the program fixed the problem very shortly after they were notified of the errors and they upgraded PMUSimulator to version 1.0.1. This version had the same features of the original PMUSimulator without the errors. In this version, it was possible to send up to twenty phasors, twenty analogs, as well as digital signals on one instance of PMUSimulator.

### 2.2.3 iPDC Version 1.2

In PMUSimulator Version 1.2, which was released in late 2011, reading Comma-separated values (CSV) files with necessary data frame information became possible. The user could create a specific configuration frame that would match the specifications of the data file to be read. If the user created a configuration frame that did not match the layout of the CSV file, then an error would occur within the program. However, during testing, it was noticed that when reading from a CSV file, there were some inconsistencies between the CSV file and the final destination, which was PMU Connection Tester in this setup. The frequency displayed on PMU Connection Tester would read 60 Hz below what the CSV read. For instance, if the data file had a frequency of 60.1 Hz for a given data frame, the frequency on PMU Connection Tester would read 0.1Hz. A way to counteract this problem was to simply edit the CSV file and add 60 to all of the frequency values so that the end result would be correct. Another problem was that the phase angle for the phasors all lagged by 180 degrees. In order to counteract this problem, the user could edit the CSV file and add 180 degrees, or pi radians, to the value of the angle to obtain the accurate end result. Since this program is open source, the user could also change the code so that these changes would be made automatically. Another problem arose on some computers, particularly a 64 bit Asus computer, received a "Segmentation Fault" when trying to install the program. On the 32-bit systems the error did not occur. Again, the creators of the software were notified of the problems encountered and they fixed the problems and released iPDC Version 1.2.1, which at the time of this thesis the most current version of the software.

### 2.2.4 iPDC Version 1.2.1

In PMUSimulator Version 1.2.1, the Segmentation Fault was corrected as well as the wrong phase angles and frequencies read by the end user when reading CSV files. The format of a CSV file must be as follows:

“Timestamp, Phasor 1 magnitude, Phasor 1 angle(radians), Phasor 2 magnitude, Phasor 2 angle, …, Phasor N magnitude, Phasor N angle, Frequency, df/dT, Analog 1, Analog 2, …, Analog M”

The maximum number of phasors and analogs for PMUSimulator is 20 for each channel, as in the previous versions. In this version of the software, the user can create and save a Configuration Frame whereas in previous versions, the user would have to create a new configuration frame for every new instance of PMUSimulator. This saves time on the user not having to create a configuration frame every time a change is made.

### 2.2.5 iPDC (PDC) Versions

There has only been one release of the actual iPDC, which is a Phasor Data Concentrator. Most of the testing was done using only PMUSimulator in conjunction with either real PDC’s or PMU Connection Tester. However, it is often desirable to be able to send Synchrophasor data to a machine running Ubuntu or any UNIX based operating system due to the time accuracy associated with the operating system. Using standard hardware such as switches, configuring a UNIX based system to receive data and configuration frames is no easy task. PMUSimulator as well as most PMU configurations require a command frame from a client before they will start sending data. The problem is sending a command frame to a PMU without a PDC to generate it. Fortunately, iPDC can be used to send command frames to PMUs and PMUSimulator so that the PMUs will begin sending data and configuration frames to a UNIX based computer for data frame timing tests. This can be implemented regardless of hardware specific to a given network.

After receiving hubs, which send out all the data they receive; it is possible to connect a Windows computer running PMU Connection Tester to a UNIX machine, through a hub, so that all the data frames that go to the windows computer will also go to the UNIX machine automatically. This is a hardware solution to the problem of receiving Synchrophasor data on a UNIX machine.

One of the current problems with iPDC is that when it outputs a phasor data stream consisting of more than one PMU, the data stream is corrupt. Instead of sending data for all the PMUs, it leaves off the data of the last PMU. In PMU Connection Tester, this error can be seen as the program expects a SYNC byte (AA XX) and receives the checksum of the first incoming PMU to iPDC.

The corrupted iPDC PMUSimulator software solved the problem of generating or reading phasor data, however, for a Real Time PMUSimulator the unit must also synchronize the computer system time to the actual UTC time so that PMUSimulator can have a valid Real Time "time-tag". The PMUSimulator must behave as a real PMU does by sending the data frames at the right time and rate. The next section will describe how a UNIX machine can be synchronized to the right time, which will allow accurate PMUSimulator time stamping.

## Chapter 3: UNIX Time Synchronization

In Windows operating systems, the time resolution is approximately one millisecond for most applications. The default time resolution for Windows 7 is 15.6 milliseconds, which is nearly the time between 2 frames sending at 60 frames per second for a PMU. One millisecond accuracy is far too inaccurate for simulating Synchrophasor data in a Windows platform. [19]

However, UNIX based operating systems, which PMUSimulator was designed for, allow for time resolutions on the order of microseconds. FreeBSD is a UNIX based non GUI operating system that can obtain time accuracies below 1 microsecond. Ubuntu is a UNIX based GUI operating system that has a time resolution of approximately 1-50 microseconds. Ubuntu is a good choice because of its ease of use and high time resolution. Ubuntu does not automatically synchronize to within a few microseconds of the actual time. The user must use a type of time protocol in conjunction with a GPS clock to achieve time synchronization. The most well-known types of time protocols are Network Time Protocol (NTP) and Precision Time Protocol (PTP). PTP tends to yield more accurate results, but NTP is very easy to implement and the limit on the accuracy is the operating system itself, not the specific type of protocol. For instance, PTP could be used on a Windows operating system but the time resolution would still be only 1 millisecond. The same is true for Ubuntu, the limit is not NTP or PTP, the limit is the operating system itself. For the proposed application, NTP was selected in conjunction with a GPS 18x LVC receiver.

### 3.1 Network Time Protocol (NTP)

NTP synchronizes computer clocks to time references and was created by Professor David L. Mills from the University of Delaware [1]. Network Time Protocol uses the UTC (Universal Time Coordinated) time as the standard. The UTC time is estimated by various organizations and the actual time is derived from a combination of these estimates. Since PMUs

use UTC time, PMUSimulator can have the same timestamp as real PMUs. An NTP server can be created on a network that enables any other computers on the network to synchronize to that server, which allows for multiple computers running Real Time PMUSimulators at the same time.

Even if a computer had an extremely accurate clock, if it is not updated regularly, the time will drift with respect to other clocks. The resolution, the smallest increment of time possible on that particular clock, is limited by the actual hardware of the computer. However, there is time needed for the system clock to interpret the data. This time, called precision, is the smallest possible amount of time a program needs to increment its clock. When multiple measurements of the system offset are measured, the values will vary, this is known as jitter. Accuracy is a measurement of how close the system clock is to the UTC time. Hardware clocks always have some frequency error. Over time, NTP will take measures to account for the frequency error but the frequency can be slightly changed by temperature and other affects.
In the area of Phasor Measurement Units, time synchronization is critical to having an accurate system wide view of the grid. The most important aspect of a Synchrophasor message is precise time stamping. A PMU inserts a time stamp on every created phasor so that a snapshot of the system can be viewed. The Real Time PMUSimulator program obtains the time from the system clock, which is time synchronized by NTP, and places that information into the data frame. The time stamp in the Real Time PMUSimulator data will inherit the resolution, precision, jitter, and any other characteristics of the computer time. The NTP time matches PMU time, which allows the PMU simulator to generate a nearly identical timestamp as the real PMU.

NTP is configured by the user to accept certain NTP servers to synchronize to. These servers can be on the internet, a local network, or a reference GPS. If multiple servers are used,

NTP can use a combination of all of the servers to more accurately determine the time. NTP automatically adjusts the system time, but the adjustments are very small and it can take hours for NTP to converge on the actual time. NTP requires a network or internet connection in order to connect to NTP servers. If for some reason a UNIX machine running NTP is disconnected from the network that has the servers it is using, NTP can use past measurements in order to predict what the current and future times will be.

There are thousands of NTP servers available on the internet, most of which are very inaccurate. Virginia Tech maintains four NTP servers that are much more accurate than servers found on the internet. These Virginia Tech servers are physically closer, which reduces delay, and they are maintained properly. For this thesis, Virginia Tech's NTP servers and a Garmin GPS 18x LVC clock were used as the NTP servers.

A reference clock is a very accurate clock that produces a time that NTP can use. With the advent of GPS in recent years, GPS clocks are a cheap and fairly accurate type of reference clock. With NTP, reference clocks are called Stratum 0 servers, which indicate the most accurate type of server. The computer synchronizing to a GPS Stratum 0 server will become a Stratum 1 server if another computer tries to synchronize to that computer. The higher the stratum number, the less accurate the server is. NTP records statistical information such as offset, frequency error, and jitter so that the computer can determine which references are most accurate.

NTP can utilize Pulse Per Second (PPS) synchronization, which uses an extremely accurate external clock that sends a pulse to the UNIX machine every second. The computer tries to synchronize with these pulses and make corrections every second. It is possible for specific programs to use PPS signals but NTP uses PPS directly with the kernel of the operating system, which is much more accurate since the kernel clock can be synchronized to an accurate PPS

signal. These very accurate PPS signals can be generated by GPS receivers.

NTP determines a polling interval for each particular server that minimizes all unwanted noise. The steady state accuracy is determined by the server being used as well as the speed and reliability of the network. NTP determines the stability of the clock and uses that information to define how often it should check the server so that the clock does not drift. When a server is polled often, the clock is updated frequently but subject to more jitter. When a server is not polled often, there are large errors at updates.

In order to have reasonable accuracy, at least three NTP servers should be used. Using only one server may encounter the problem that the particular clock is inaccurate. With three or more servers, NTP can more accurately estimate the accurate time. NTP will evaluate the error of all of the servers used and estimate the accurate time. NTP will determine which are the most accurate clocks and not use any clocks that are inaccurate.

### 3.1.1 Garmin GPS18x LVC

For this project, a Garmin GPS18x LVC receiver was used for NTP time synchronization. The Garmin GPS18x LVC will lock on to several satellites to obtain an accurate location, movement, and time. This GPS receiver outputs National Marine Electronics Association (NMEA) 0183 data. It features a very accurate PPS output, which can be used with NTP to tune UNIX machine clocks for more accurate timing. In order for this device to work with a computer, the user must physically connect the outputs of the GPS module to both a serial port as well as a USB port in the computer, as shown in Figure 1. The serial port transfers all of the pertinent GPS information as well as the PPS Signal and the USB cable is used to power the device. According to the GPS specifications, the PPS edge is accurate to +- 1 microsecond. The PPS width can range from 20ms to 980ms in increments of 20ms. A default width of 200ms was used.

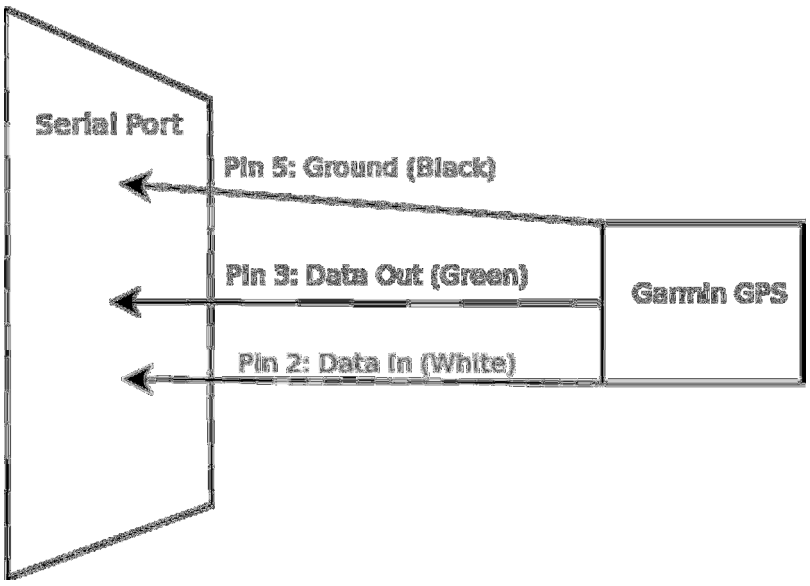


Figure 1 Garmin GPS to Serial Port [16]

The Garmin 18x LVC needs some slight modification in order to function most accurately. By the default settings, the Garmin 18x LVC GPS will output much more data than what is required for the Real Time PMUSimulator application, such as exact location. In the application of time synchronization, the only important information is the PPS functionality as well as the time data provided by the GPS. When the GPS is sending the Stratum 1 server many unnecessary messages, increased latency and inaccuracy will result. Garmin offers configuration software that the user can specify which messages are needed and which are not. For the time synchronization, only the time information was used. In FreeBSD, firmware version 3.60 of the Garmin GPS caused a 1 second offset to what the value should be. After upgrading to firmware version 3.70, the offset disappeared with good results. In Ubuntu, both firmware version 3.60 and 3.70 worked for time synchronization.

## 3.2 NTP Results

For this project, two NTP configurations were tested. Both configurations used the Garmin 18x LVC GPS receiver as the Stratum 0 server and Virginia Tech's NTP servers as the

Stratum 2 Servers. First, Ubuntu was used as a Stratum 1 server, with other Ubuntu based computers acting as Stratum 2 servers. Second, a more accurate system involved using two FreeBSD computers as the Stratum 1 servers, and Ubuntu computers running as Stratum 2 servers.

### 3.2.1 Ubuntu Stratum Server

Figures 2 to 3 show the time offset with all operating systems on Ubuntu. The Stratum 1 server is running Ubuntu and the Stratum 2 servers, also running Ubuntu, receive their time from the Stratum 1 server.

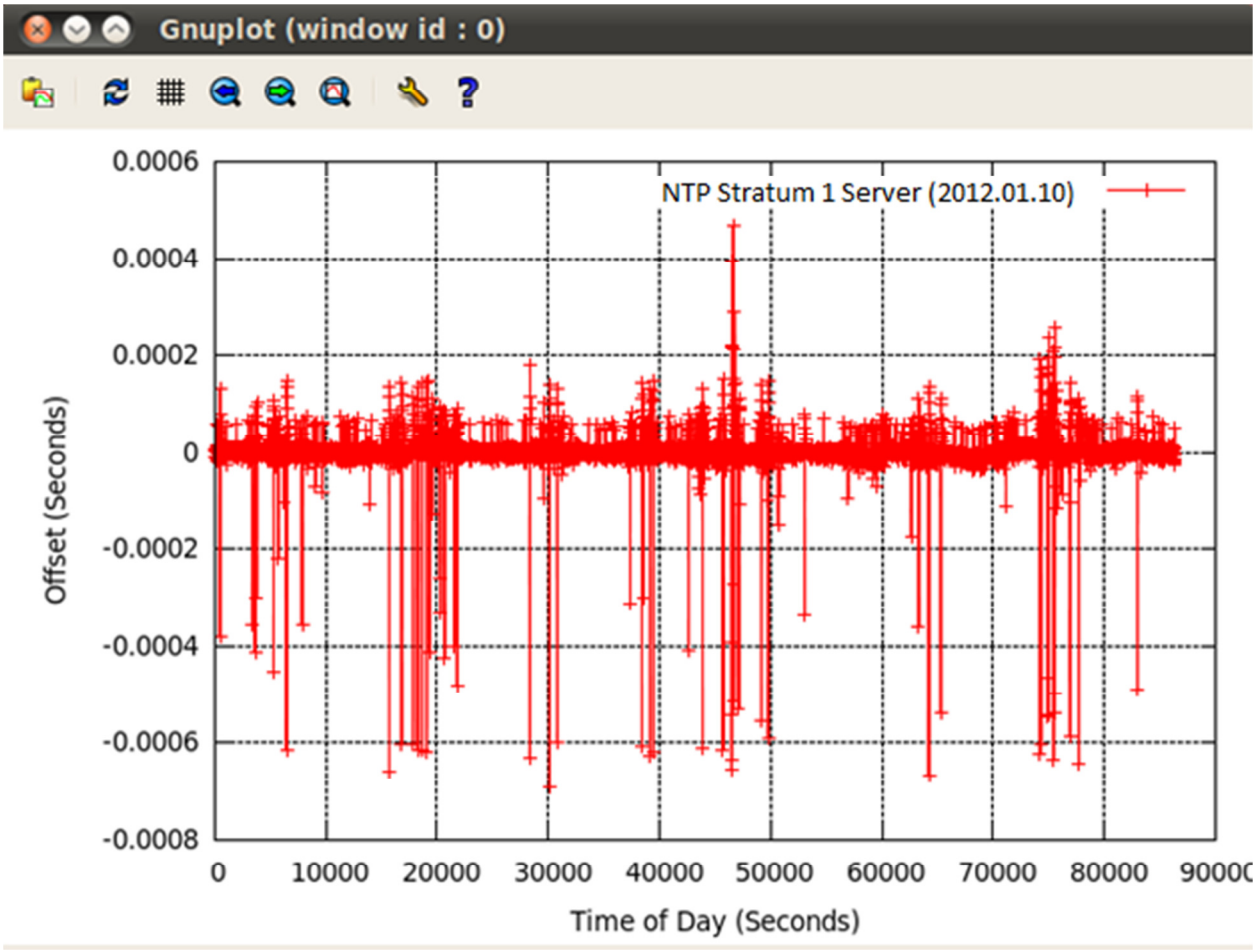


Figure 2 NTP Stratum 1 Ubuntu Server

Figure 2 shows the offset with respect to the time of day of an Ubuntu system synchronized via GPS using NTP. The offset is very close to zero throughout the day and it is obvious that Ubuntu is much more accurate than the 1 millisecond that is possible with Windows operating Systems.

For Figure 2, the mean was 0.112302 microseconds, a variance of 68.0201 microseconds and a Standard Deviation of 8.24743 microseconds. This Stratum 1 server is closer than 20 microseconds to the actual time for the vast majority of the time.

Using NTP, it also becomes desirable to synchronize other UNIX machines to the Stratum 1 server that was just created. These Stratum 2 servers will not be as accurate as the

Stratum 1 server because no client can have a more accurate clock than its server. Figure 3 shows a plot of the offset of the Stratum 2 server running Ubuntu that is synchronized to a Stratum 1 server also running Ubuntu.

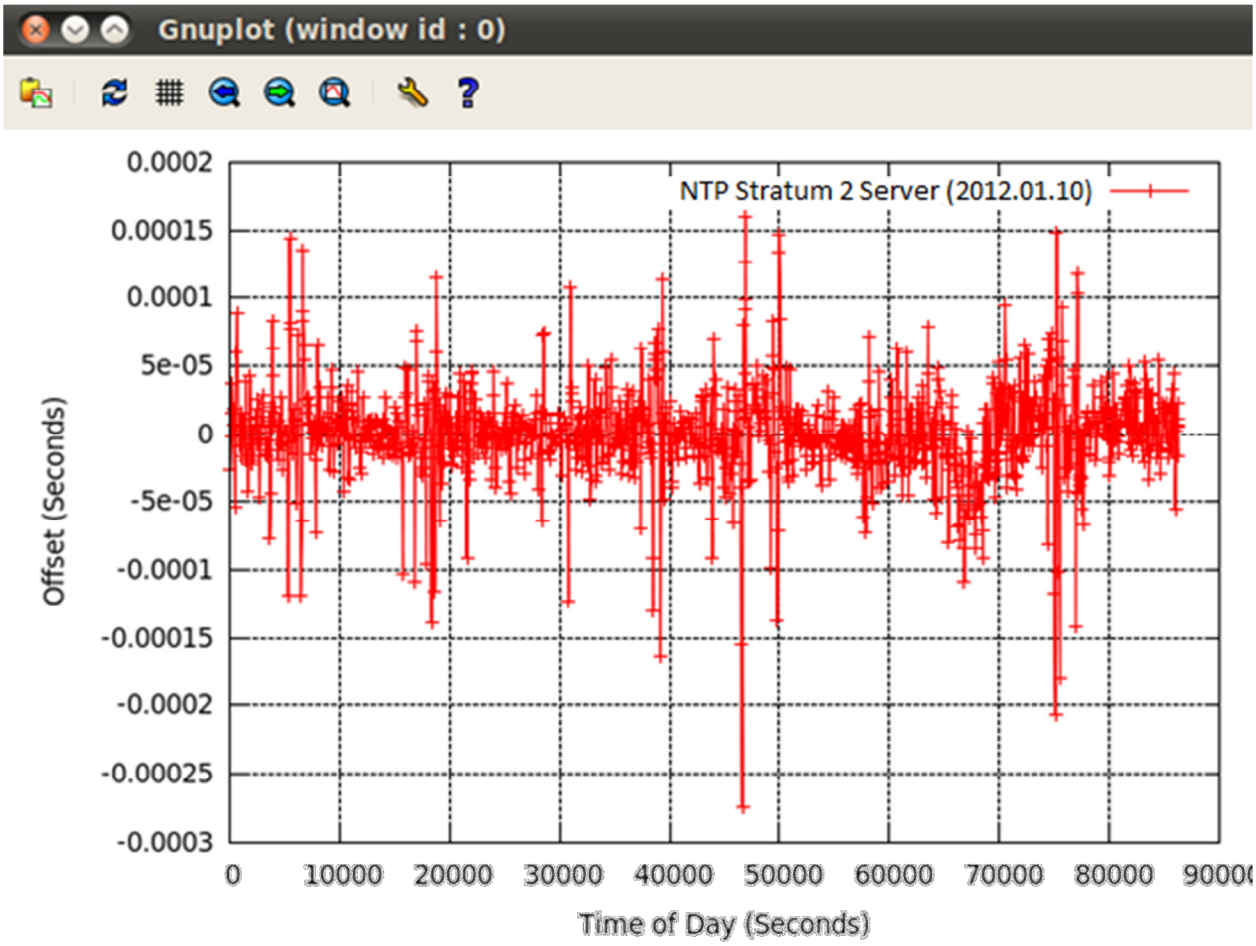


Figure 3 NTP Stratum 2 Server Ubuntu

For this particular Stratum 2 Server, the mean was 0.708176 microseconds, the variance was 1048.84 microseconds, and the standard deviation was 32.3858 microseconds. The other Stratum 2 servers yielded similar results.

### 3.2.2 FreeBSD Stratum Server

#### *3.4.2.1 FreeBSD Advantages*

Since Ubuntu can only have a UTC time accuracy of approximately 30 microseconds, it becomes desirable to use FreeBSD as a Stratum 1 Server instead of Ubuntu. FreeBSD can obtain much higher resolutions than any graphically based operating system can. Typically, FreeBSD can be within one microsecond of the actual time. There are multiple computers that need to synchronize their times, so if all of the Stratum 2 servers were linked to a Stratum 1 server running Ubuntu, the results would be good, but not as accurate as possible. With synchronizing multiple computers, configuring a Stratum 1 server to run on FreeBSD is the optimal solution. FreeBSD is an operating system that has no graphical user interface, which is similar to using a Windows PC with only command prompt. It is a difficult operating system to use but yields the best time resolution. The Operating System requires the user to manually do many tasks that graphical user interface (GUI) based systems do automatically. For instance, the user must configure the serial port to accept GPS NMEA messages.

#### *3.4.2.2 FreeBSD Setup*

For connecting FreeBSD to a Garmin 18x LVC GPS, first, the FreeBSD operating system must be installed. During installation, the user must designate the partitions and setup the network connections. After the installation is completed, the user must recompile the kernel to allow the system clock to synchronize to a PPS signal. The kernel connects the hardware to the software at a lower level than the application layer. There is a “GENERIC” kernel that FreeBSD offers, but also a “PPS” kernel. The user must add the line “options PPS_SYNC” to the kernel file and then build and install the kernel. It takes 2-3 hours to recompile the kernel, but it is necessary to take advantage of the GPS PPS signal. Next, the user must enable the serial port to accept GPS signals. At this point, the user can view and edit the NTP configuration file. The

default editor in FreeBSD is “ee”. So, to open the file the user must type “ee /etc/ntp.conf”. This will open the editor that shows the contents of NTP configuration file. This file contains the NTP information. The user must specify a directory for the statistics, which are useful for determining how accurate a certain computer is. This file also contains a list of all of the servers that the user wants to use. For this setup, two of Virginia Tech’s NTP servers were used and one server was the Stratum 0 Garmin 18x LVC GPS. The user must program the kernel to use PPS Synchronization by asserting “flag3 1”. Once the servers are added, the user must apply the changes by restarting NTP. As with Ubuntu, it takes a while for the offset to converge to its final value. Fortunately, although it takes NTP a long time to synchronize initially, the clock stats are recorded so that once the system clock is synchronized to the PPS signal; NTP can determine what clock adjustments to use to synchronize faster. The time offset of a GPS synchronized FreeBSD system was recorded for one day and plotted, as shown in Figure 4.

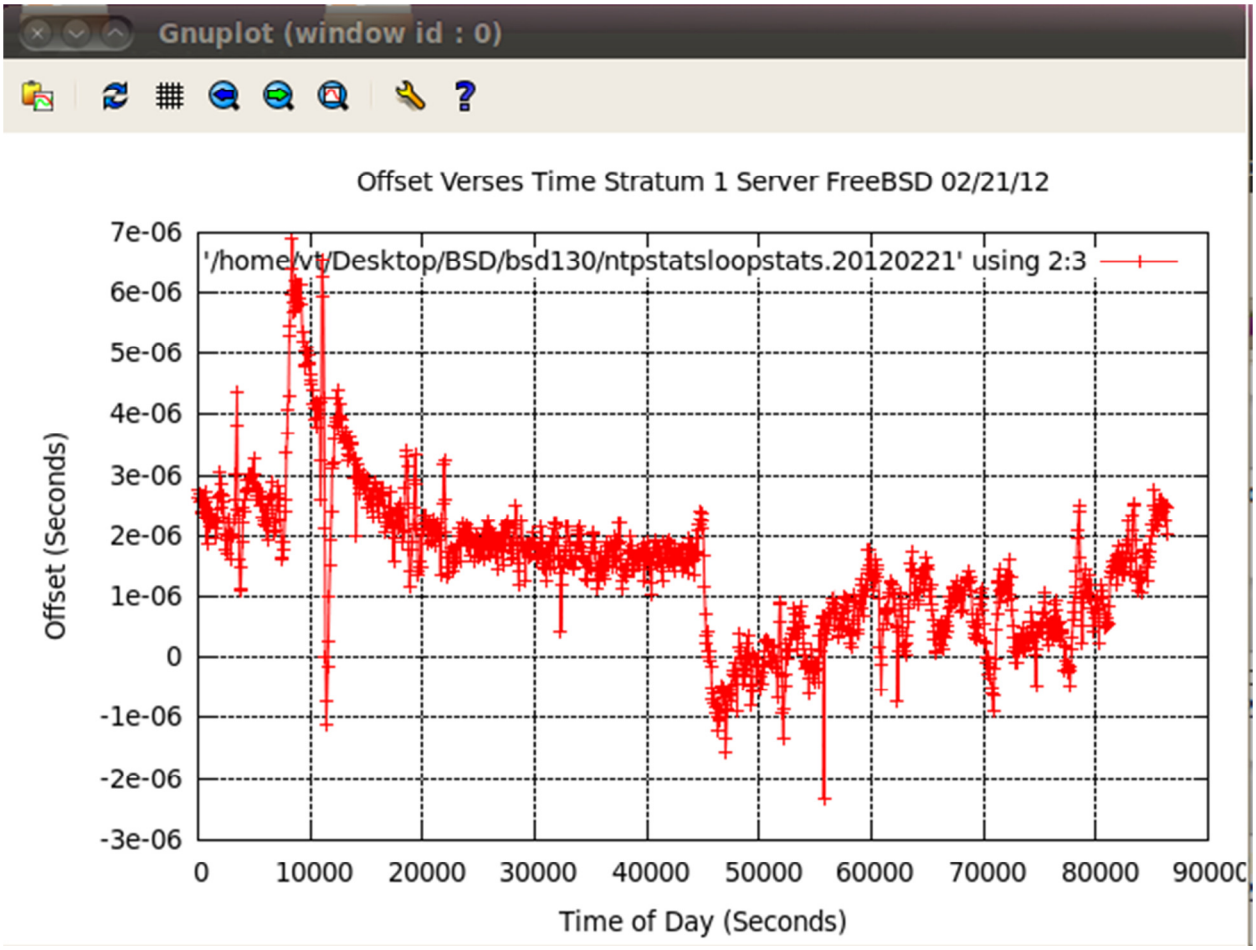


Figure 4 NTP Stratum 1 Server FreeBSD (1)

For this computer running FreeBSD, the offset had an average of 1.50173 microseconds, a variance of 51.6142 microseconds, and a standard deviation of 7.1843 microseconds. A second FreeBSD operating system was synchronized to GPS, and yielded similar results, as shown in Figure 5.

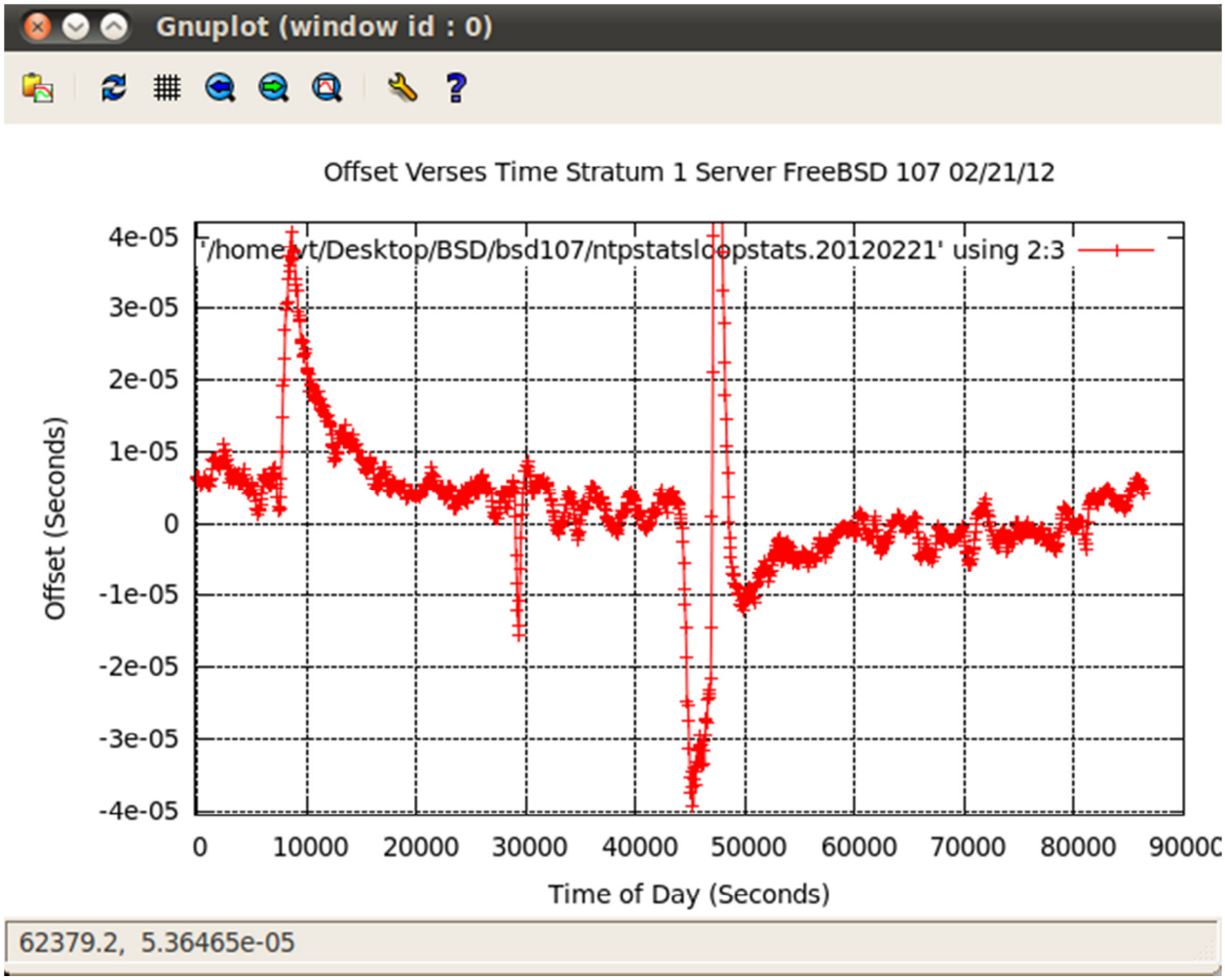


Figure 5 NTP Stratum 1 Server FreeBSD (2)

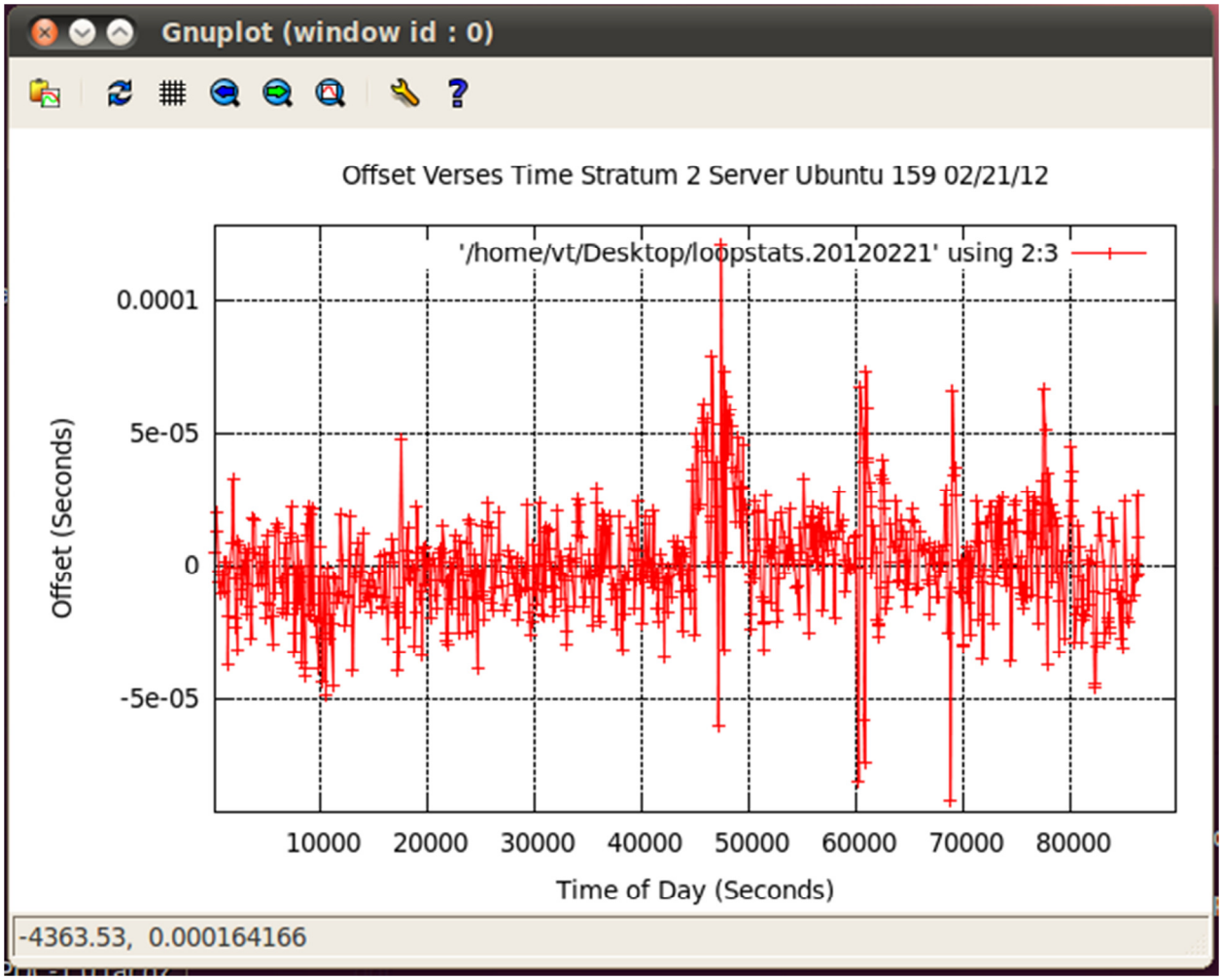


Figure 6 NTP Stratum 2 Ubuntu (BSD Server)

Figure 6 shows the offset of a Stratum 2 server running Ubuntu using the FreeBSD Stratum 1 server as its primary server. Table 1 summarizes the two NTP configurations that were tested.

| **Server Type** | Mean (us) | Standard Deviation (us) | Variance (us) |
|---|---|---|---|
| FreeBSD Stratum 1 | 1.50173 | 7.1843 | 51.6142 |
| Ubuntu Stratum 2 (BSD Source) | 0.921278 | 17.288 | 298.873 |
| | | | |
| Ubuntu Stratum 1 | 0.112302 | 8.24743 | 68.0201 |
| Ubuntu Stratum 2 (Ubuntu Source) | 0.708176 | 32.3858 | 1048.84 |

**Table 1 NTP Server Summary**

From Table 1, it is clear that FreeBSD is the most accurate operating system for time synchronization. However, for the purpose of using Real Time PMUSimulator, Ubuntu is the necessary operating system. So, to allow multiple Ubuntu systems to synchronize at the same time over a network, FreeBSD is used as the Stratum 1 server while Ubuntu will be the Stratum 2 server. Since there were two Garmin GPS's available, two FreeBSD systems were used in conjunction with Virginia Tech's NTP servers, as shown in Figure 7.

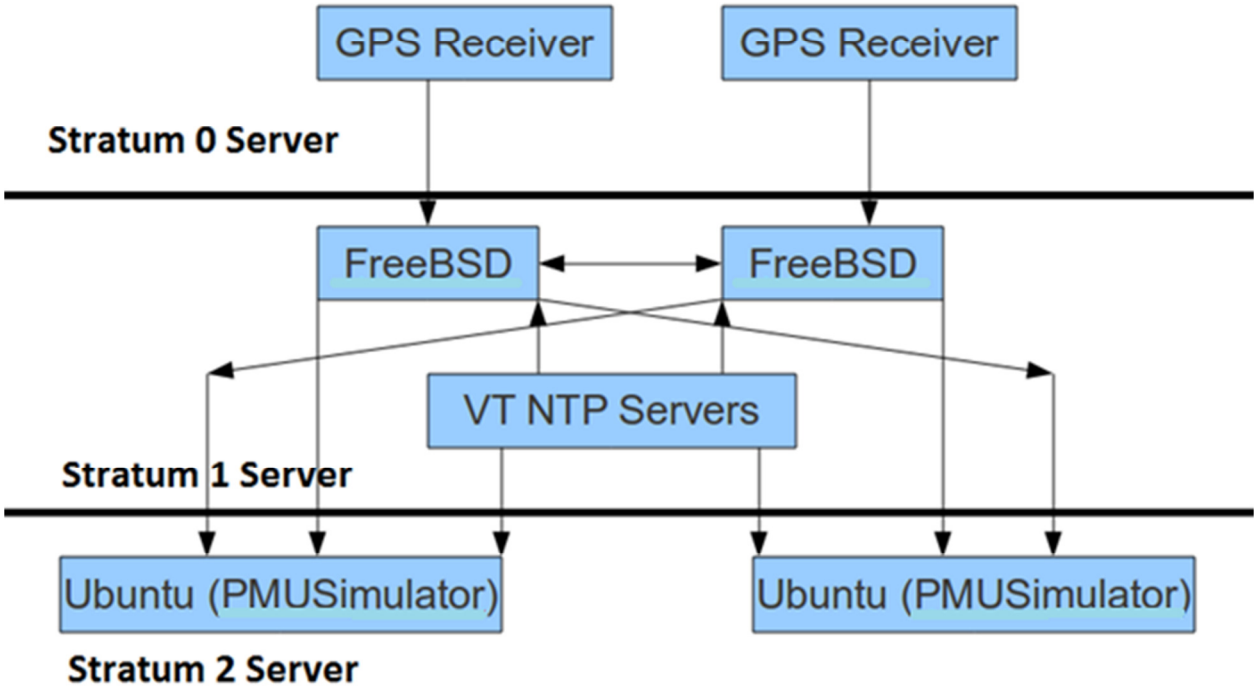


Figure 7 NTP Layout

Real PMUs have resolutions on the order of a few hundred nanoseconds. However, a PMU simulator does not need to compute a phasor as a PMU does. A Real Time PMU simulator only needs to generate a data frame by reading or generating the phasor value directly with no need for any complicated math. A real PMU samples over a window and with those samples sets the time stamp somewhere in the middle of the window which can lead to inaccuracies. A PMU simulator obtains a time stamp from the operating system and immediately sends out the data frame. Though the time resolution is far from a real PMU, there is not as much computation necessary so the simulator is reasonable.

This chapter has explained how NTP can be used to allow a Real Time PMUSimulator to be synchronized to UTC time and thereby generating identical timestamps as real PMUs. At this point, the Real Time PMUSimulator software can generate and send PMU data frames with

precise timestamps. The next chapter will discuss how Real Time PMUSimulator can be used to test PDC's where normal PMUs cannot.

## Chapter 4: PMUSimulator Applications in PDC Testing

This chapter will discuss how the Real Time PMUSimulator can be used to test PDC processing latency, late data frames, and bad data frame checksums.

### 4.1 Latency

The latency of a network and PDC processing time is very important. On most computer applications, due to the relatively slow nature of the human-machine interface, any latency in the network is typically very small and rarely noticeable. So, many delays cannot be perceived by the end user. However, in very precise timing applications such as phasor measurement units, small delays that would not affect most applications would have an impact on PMUs. Latency testing is also important because they help determine the limitation of protection and control applications of PMUs. There is an unavoidable delay from the time a PMU produces a phasor to the time the PDC receives the data. There is also additional delay from the time required by the PDC to processes the data, this additional delay is addressed in this chapter. The PDC's main task is to align by time stamp the data received from many PMUs. Some PDC's can be programmed to also do some type of data manipulation for local applications of the PMU data. This processing also takes time to complete and should be studied extensively to know what kind of delays to expect in the case of a contingency and their effect on the specific applications.

There are many methods of determining the latency of different nodes of a network. All of them include recording and comparing the time on all of the packets that are received on a specific device. All information that is transferred over a network is transferred via packets. Whether browsing the internet or receiving Synchrophasor measurements on a computer, all of the data transferred is in a packet form. When a person downloads anything from the internet, the download is separated into packets and transferred by a protocol, for example TCP or UDP. When a packet arrives to a computer, the computer determines that the packet is from a specific

protocol and can begin assembling the packets back together the way it was before it was separated into packets. A PMU sends Synchrophasor messages over a network in the form of packets over a specified protocol. There are commercial and open source programs that can record the exact time that a packet is received by a computer**.** Wireshark is a free program that can determine the arrival and departure time of all packets on the network that the computer encounters. Any packet that the computer encounters, Wireshark records the information that exists in the packet. Wireshark can run on both UNIX and Windows operating systems and allows for packet filtering and decoding of network packets.

When using a computer program such as Wireshark to read time information, the time the program records can only be as accurate as the computer system that it is running on. If the computer system clock is only accurate to one second, the program cannot time stamp the packet any more accurately than that since the program receives the time from the system clock. Therefore, it is important to run Wireshark or any similar program on an operating system that has an accurate time for the system time. For these studies, Wireshark was run on an Ubuntu-UNIX operating system for an accurate system clock as well as its graphical platform for ease of use. The output PDC data stream was also sent to the UNIX computer running Wireshark. In this setup, the PMU, PDC, and UNIX computer were connected by an Ethernet cable into a hub. This way, the PMU data will arrive at the computer at the exact time that it arrives at the PDC. The PDC will process the PMU data and output a data stream to the UNIX computer where Wireshark will capture the data, as shown in Figure 8.

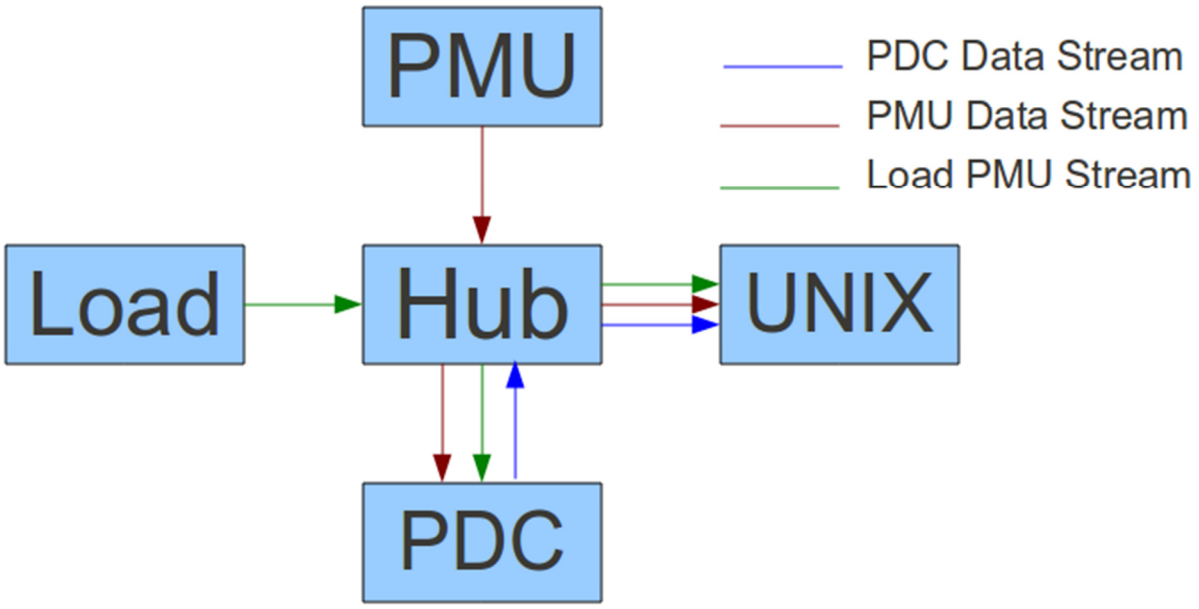


Figure 8 PMU/PDC Latency Connections

After the data is captured using Wireshark on a UNIX system, both the PDC output and direct PMU data were decoded as Synchrophasors, using a built in Wireshark function, and a Matlab program the arrival time difference between the direct PMU data and the PDC data is computed. These tests assume that separate PMU data from similar PMUs through similar communication channels and for a given timestamp should arrive at the PDC within a very small time window. Results shown in this chapter prove that even real PMUs do not send the frames at exactly the same time. However, it is useful to demonstrate the applicability of using highly controlled Real Time PMUSimulators for latency testing.

### 4.1.1 PMUSimulator verses Commercial PMUs

Whenever two commercial PMUs send data to a PDC, not only are the frame rates very specific and accurate, the time that they will actually send the packets should also be very close to the data timetag. Figure 9 shows the arrival times of two separate Commercial Schweitzer

Engineering Laboratories (SEL) PMUs to a client computer. Both PMUs were running at 60 frames per second for approximately 50 seconds.

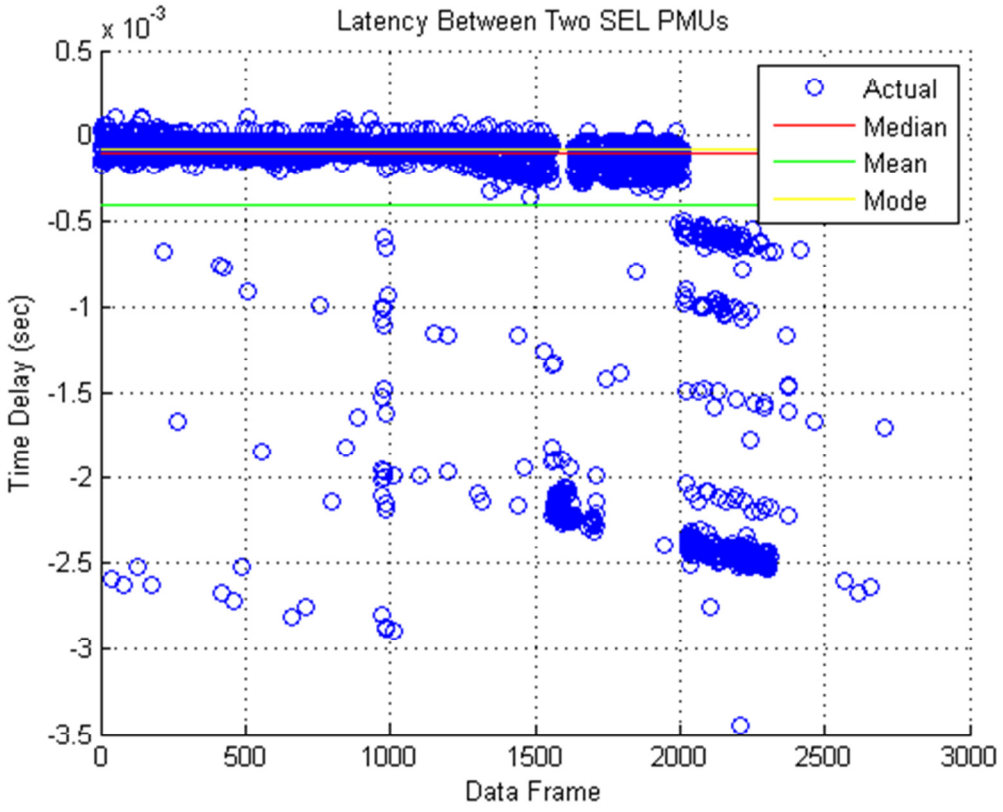


Figure 9 Arrival Time Delay Between 2 SEL PMUs

For a given timestamp, the data from one PMU will arrive, while the data from the second PMU will arrive a few hundred microseconds behind. Specifically, the data from two PMUs in Figure 9 arrived on average within 409 microseconds of each other. Essentially, the packets containing Synchrophasor data from two separate PMUs arrive at close to the same time. The differences in those times are within the realm of error of the network delays, PMU delays and the limits of the testing software in this case Wireshark. The accuracy in testing PMU data delay is essential for testing PDC's because the PDC expects the packets containing Synchrophasor data to arrive within a given time window correlated to the timestamp. Since the

processing latency of a PDC is on the order of hundreds of microseconds, if the delay introduced by the testing system packet time is any greater, PDC correct operation could not be determined.

In PMUSimulator Version 1.2.1, the data frames are sent at regular intervals, yielding a close to exact number of frames per second, however, the frames may not be sent at the same time as a real PMU would. A real PMU takes a sample of measurements triggered by a time synchronized sampling pulse, computes the phasor and uses the completion of the computation to trigger the time the data frame is sent. The data will arrive at the PDC very closely to the data timestamp time. However, PMUSimulator does not reference an exact time to send the data frame so it may send the data any number of milliseconds after the real PMU data was sent as shown in Figure 9. Since the frame rate for PMUSimulator is not precisely 60, the offset of sent packets from the real PMU and PMUSimulator drift in a random way, as shown in Figure 10.

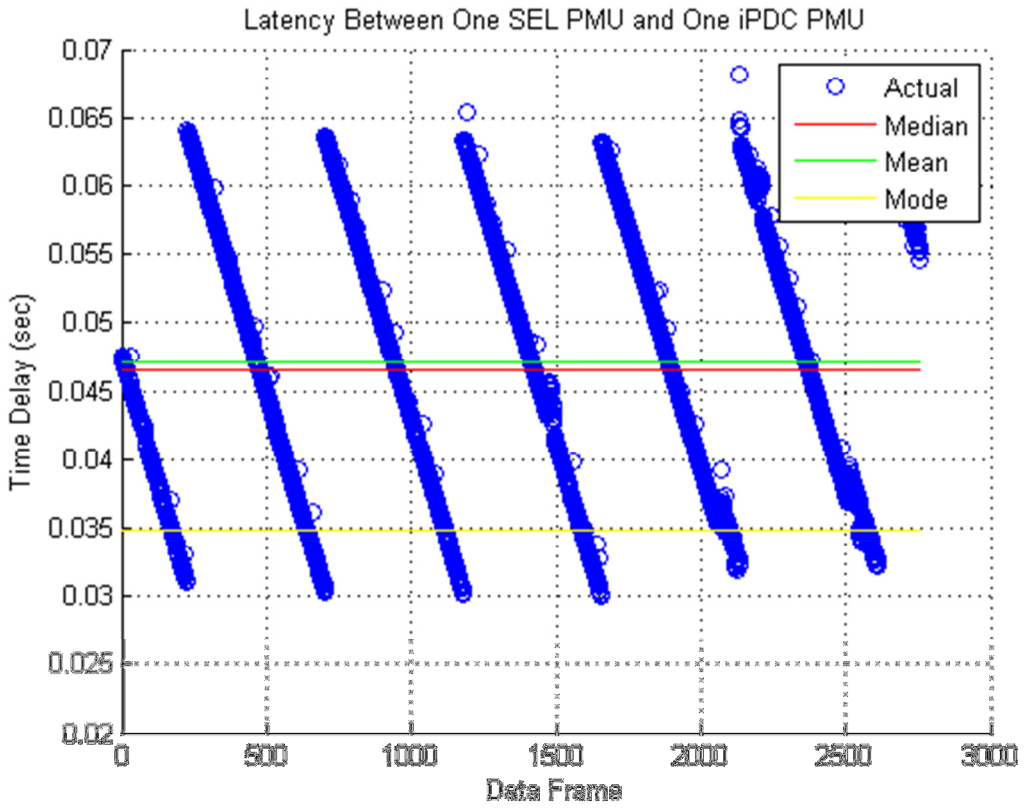


Figure 10 Arrival Time Delay Between iPDC and SEL PMU

Even with two PMUSimulators running on the same computer, the sending times will drift in time, as shown in Figure 11. The reason for this is that the computer running the program will not distribute the processing time precisely evenly.

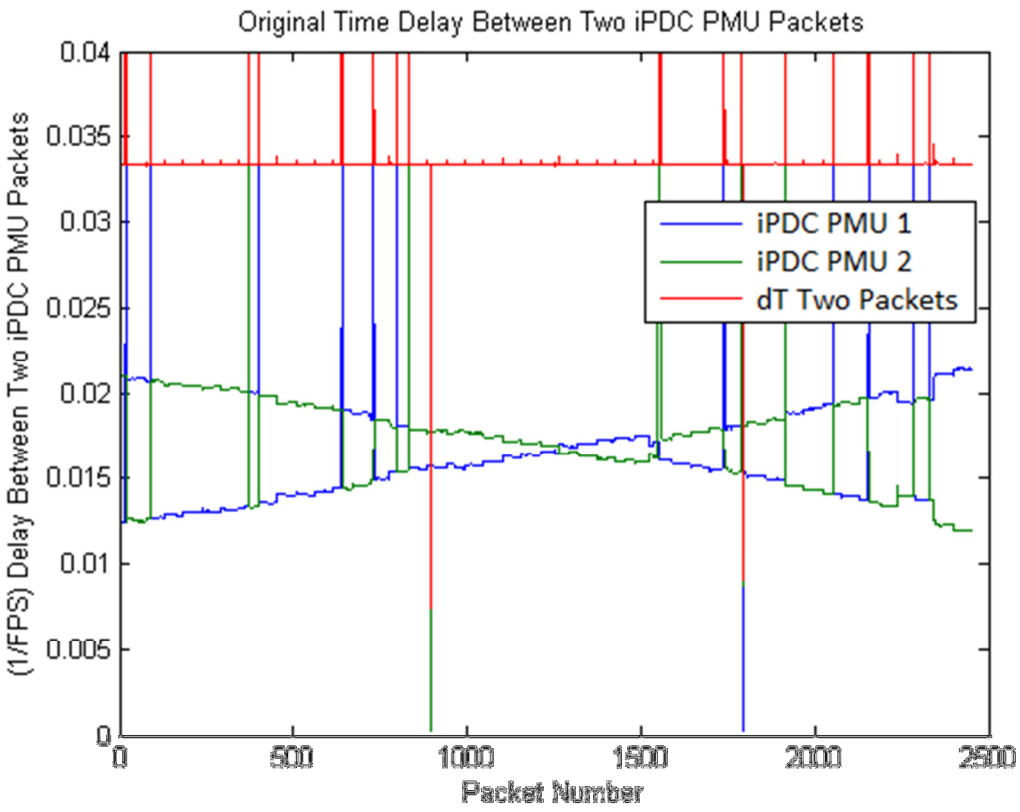


Figure 11 Arrival Time Delay Between 2 iPDC PMU's

To demonstrate the point that it is only possible to determine PDC processing latency with an enhanced PMUSimulator, a test was carried out to determine the latency of only SEL commercial PMUs and only PMUSimulators. Since the SEL PMUs send their data frames at nearly the same time, as shown in Figure 8, the processing delay is expected to be reasonably consistent. For example, the PDC will receive the UDP packets at approximately the same time, so the PDC will be able to align the time stamps in a small amount of time. With PMUSimulator, since the exact time that the data frames are sent is unknown and changed in time, the packet arrival time for a given time stamp will be vastly different. So, if the processing latency is computed in the same way that the SEL PMUs is, the latency will be higher and less consistent. The following figures show the processing latency of the SEL-3378 PDC with 1, 2, 3, and 4 incoming commercial SEL PMUs.

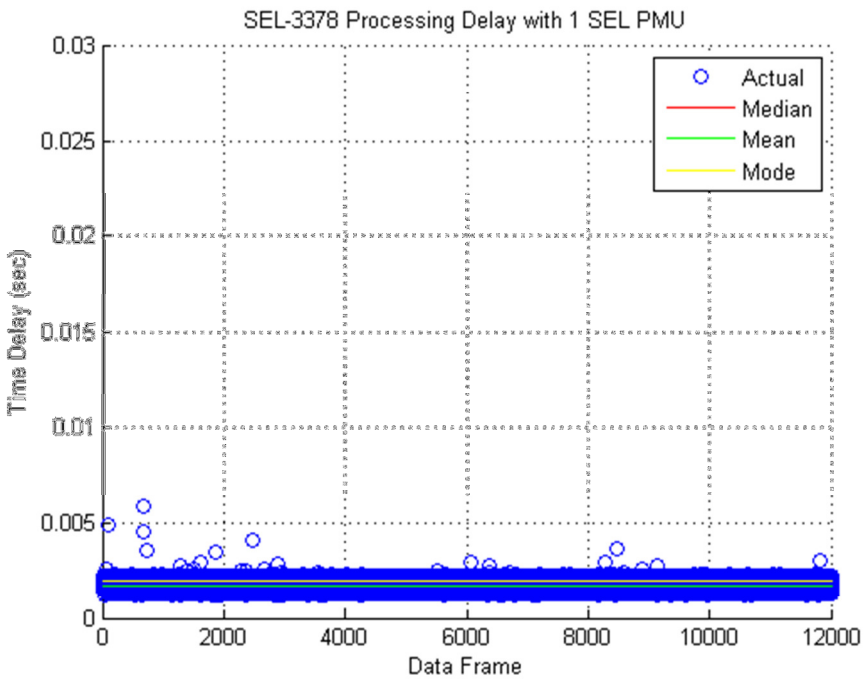


Figure 12 SEL-3378 Processing Delay with 1 SEL PMU

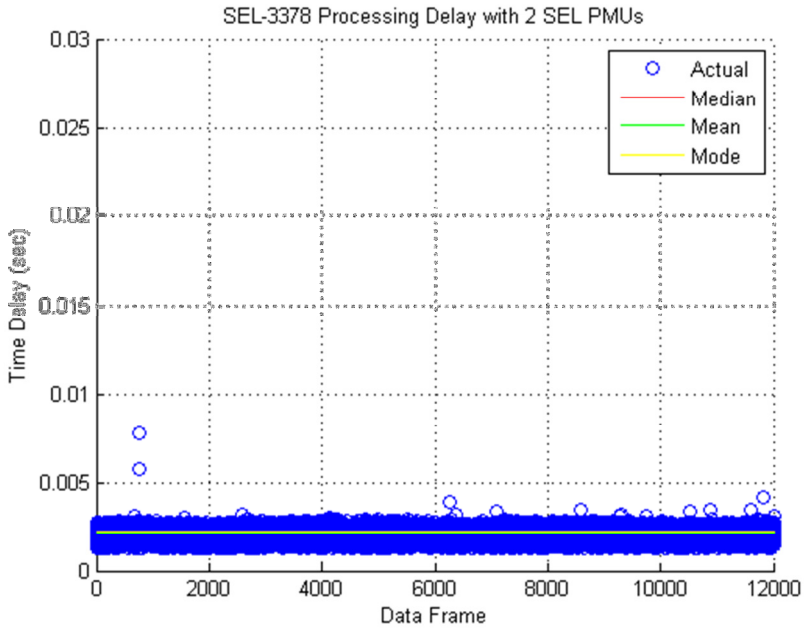


Figure 13 SEL-3378 Processing Delay with 2 SEL PMUs

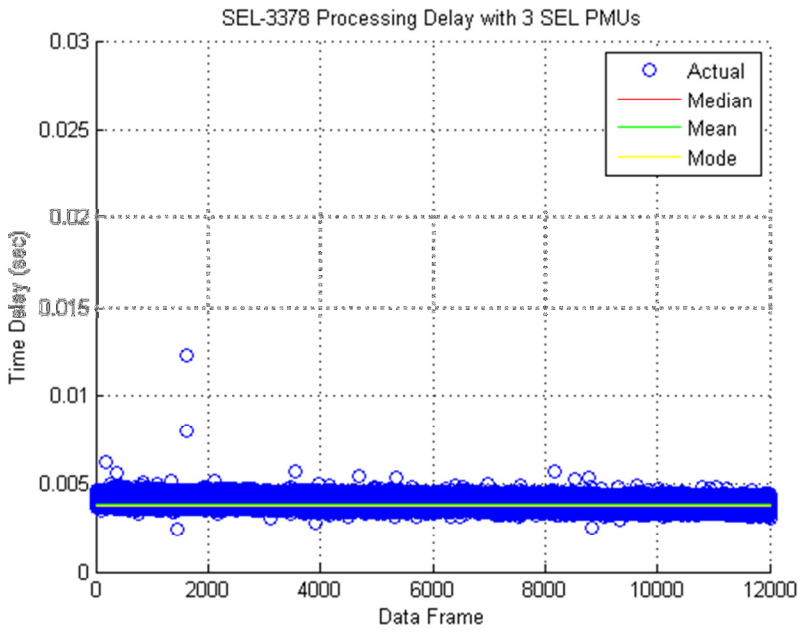


Figure 14 SEL-3378 Processing Delay with 3 SEL PMUs

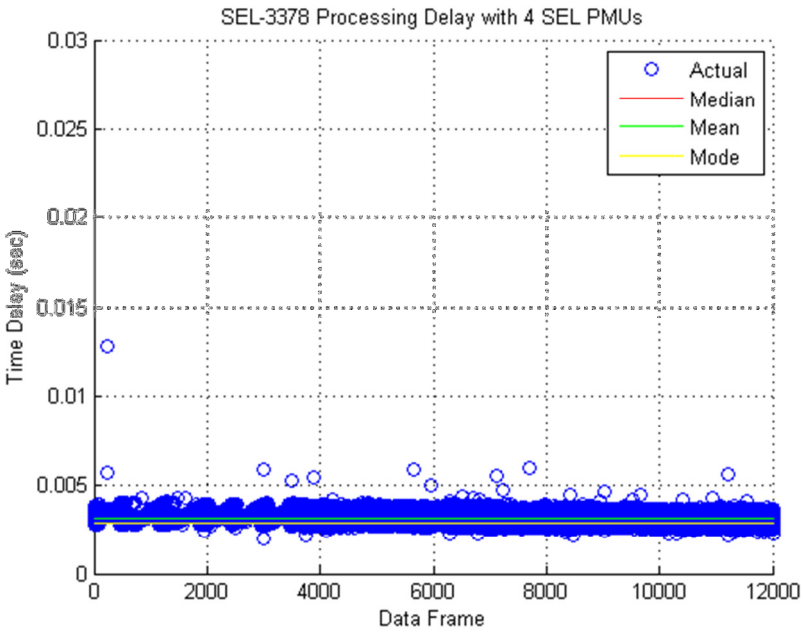


Figure 15 SEL-3378 Processing Delay with 4 SEL PMUs

The figures above show that the PDC will receive the PMU packets at fairly regular intervals, because it can align all time stamps within a few milliseconds and the latency time is consistent. In the case of PMUSimulator, the packets of different PMUs are not sent at a similar time, which results in the PDC receiving corresponding timestamps at random times. For one PMUSimulator input to the SEL-3378 PDC, the PDC will receive the data frame, and immediately send an output. Thus, for one PMUSimulator input, the latency should be consistent with that as a real PMU as shown in Figure 16.

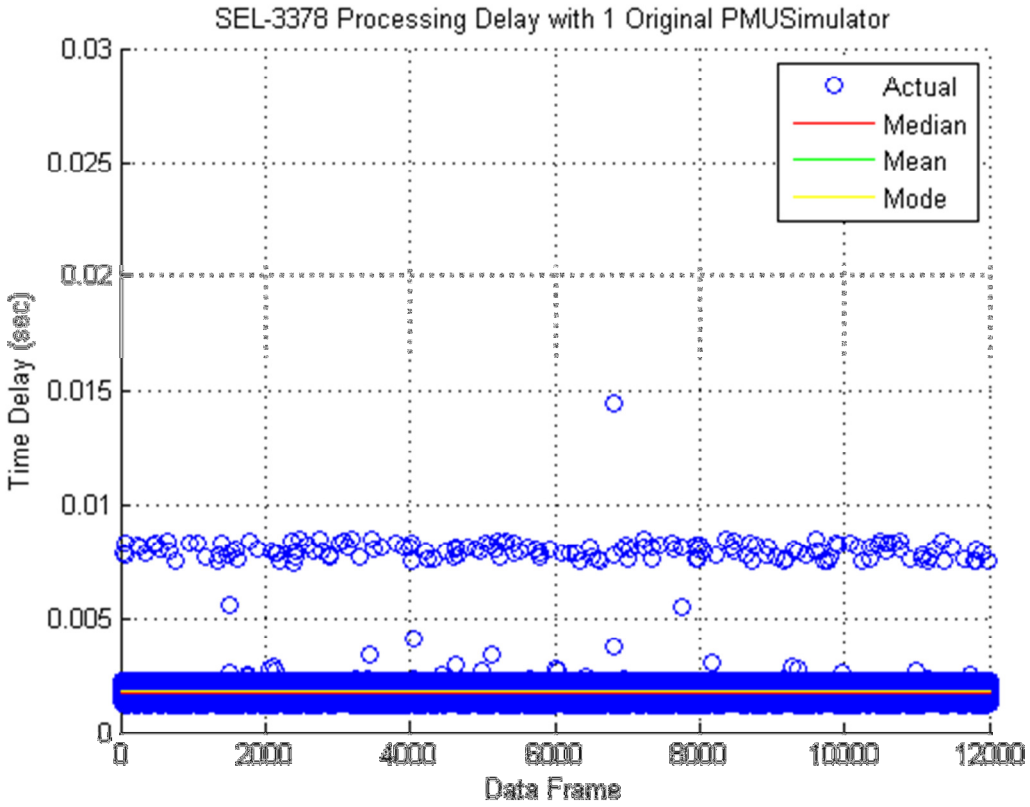


Figure 16 SEL-3378 Processing Delay with 1 iPDC PMU

In the figure above, it can be inferred that the SEL 3378 will send its output data frame as soon as it receives all of the input data frames for a given time stamp. This allows the latency test to appear consistent, and it is. However, as more than one PMUSimulator is added, the inconsistent result is shown in Figure 17.

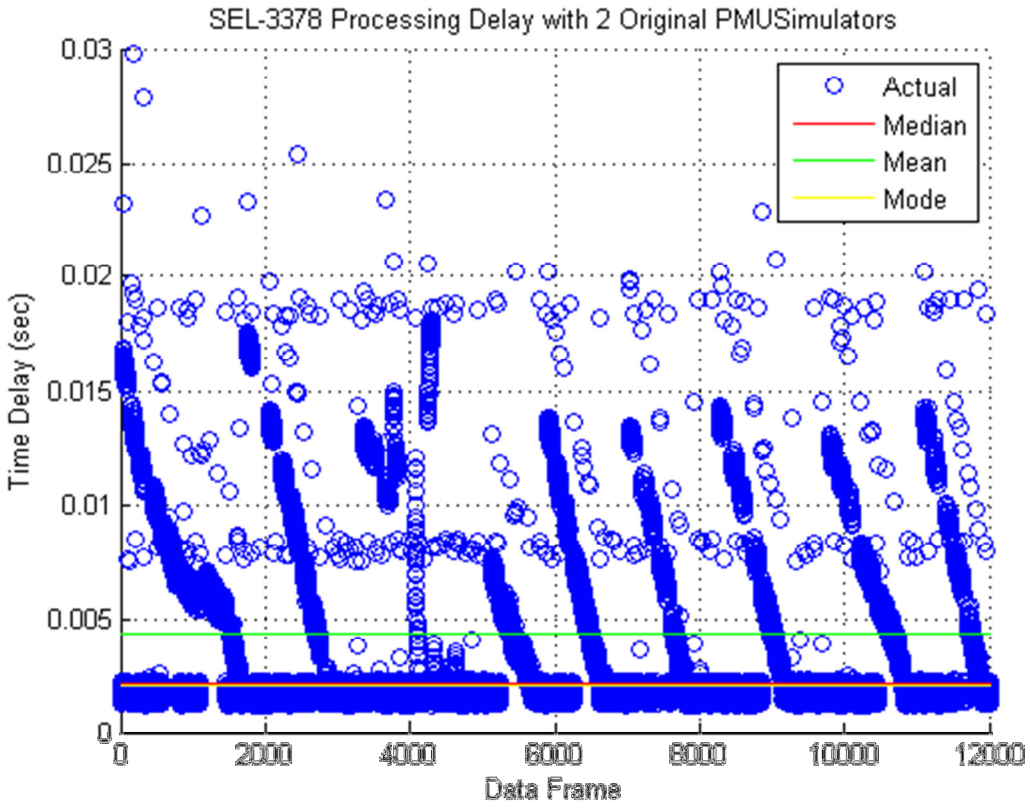


Figure 17 SEL-3378 Processing Delay with 2 iPDC PMUs

It is evident that with different PMU sending times the average latency will appear to be higher than the actual processing latency of the PDC. As even more PMUSimulators are added to the PDC, the random sending times of the data frames increases, which decrease the accuracy of the latency test. Figure 18 and 19 demonstrate that as PMUSimulators are added, the latency test that was devised becomes useless.

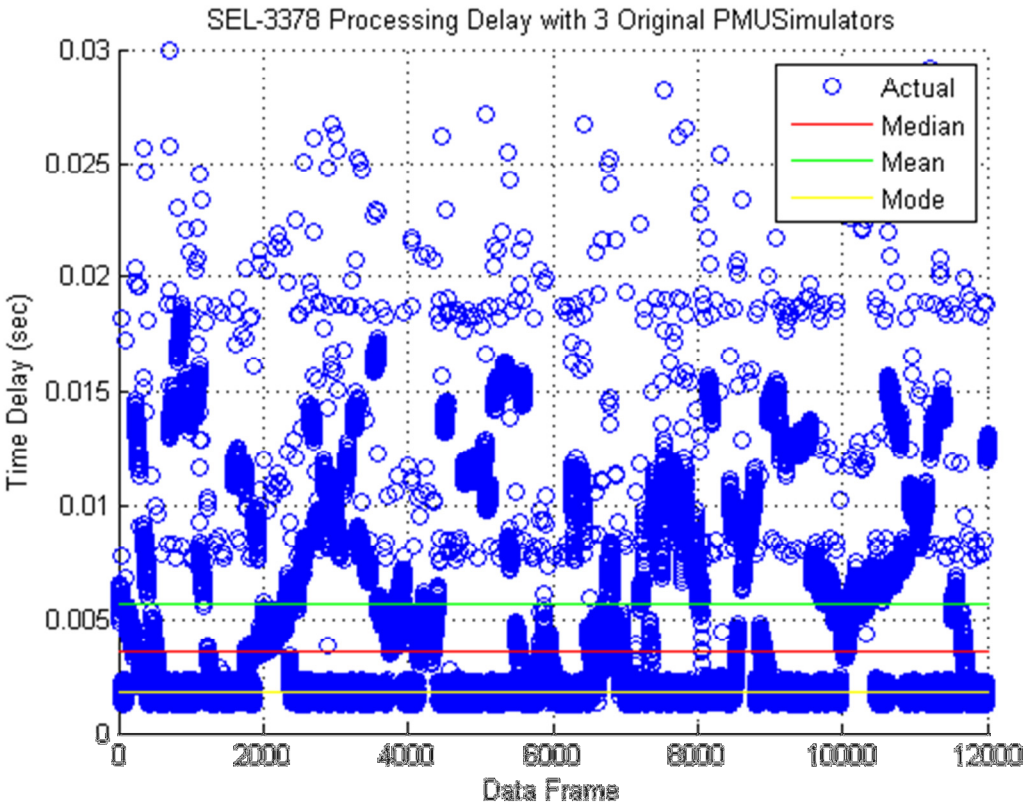


Figure 18 SEL-3378 Processing Delay with 3 iPDC PMUs

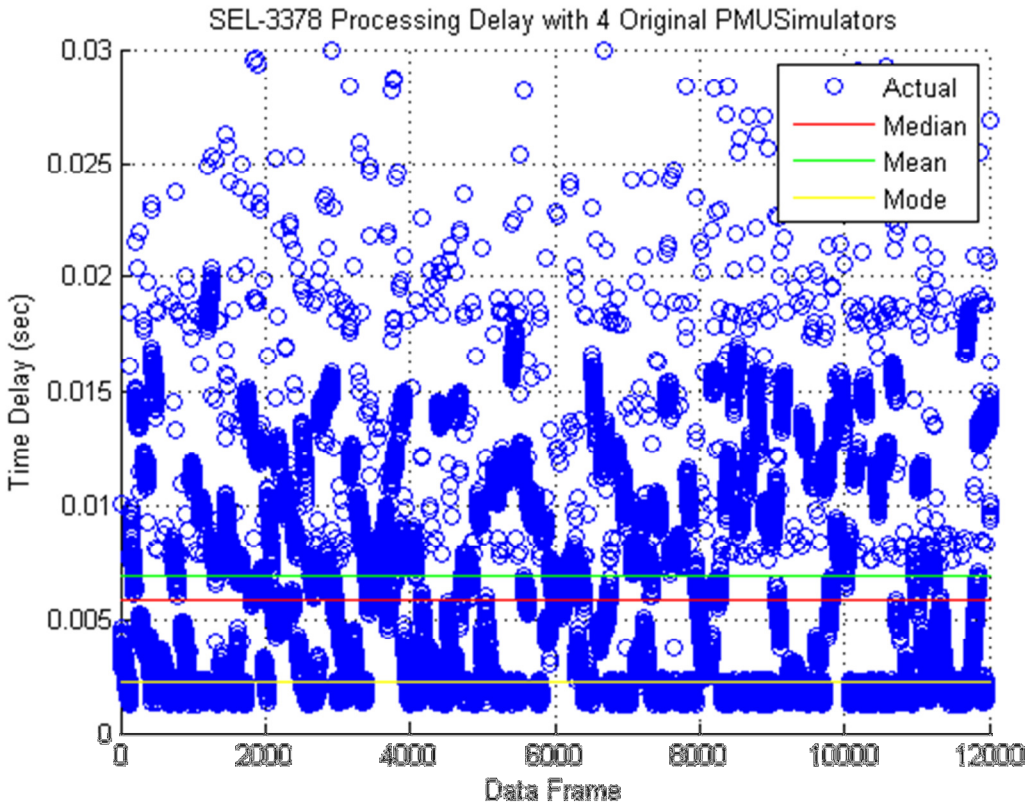


Figure 19 SEL-3378 Processing Delay with 4 iPDC PMUs

The figures above show that the original PMUSimulator setup is not applicable for PDC latency testing. The same number of input PMUs, whether PMUSimulator or SEL PMUs, should yield the same latency time, but it does not, meaning that either a new latency test must be devised or the source code of PMUSimulator must be changed.

## 4.2 Frame Rate Problems

In order for PMUSimulator to be used for PDC latency testing, the software must generate a steady stream of data frames as a real PMU does. However, a frame rate problem arises in PMUSimulator. When a 60 messages per second frame rate is desired from the PMUSimulator, a data frame must be sent every 1/60 seconds or 16.666 milliseconds. The original code constantly generated a data frame, sent it, waited for 16.666 milliseconds, and then repeated the process, as shown in Figure 20.

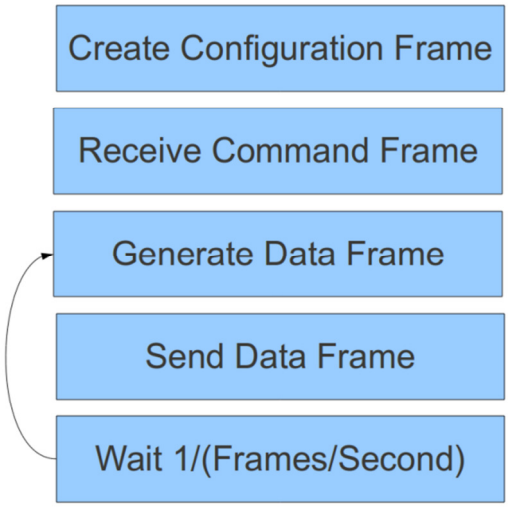


Figure 20 PMUSimulator Data Frame Simulation

However, this setup assumes that generating the data frame and sending it is completed instantly. In reality, however, generating the data frame and sending it can take anywhere from 200 to 800 microseconds depending on the speed of the computer it is being used on. The device receiving the PMU data frames receives a frame approximately every 17.000 milliseconds, which results in an approximate frame rate of 58-59 frames per second. When using the original PMUSimulator to send data from a CSV file, as time progresses, the frame rate starts to deviate from the original setup. For example, If one PMU is reading from a file at 58 frames per second and another at 60 frames per second, five seconds into the simulation the first PMU will be behind by ten frames, or approximately 160 milliseconds from the other PMU. As a result, it becomes difficult to accurately recreate a past recorded situation accurately. Another problem with an inaccurately low frame rate is the way that a PDC will interpret the data. A PDC expects exactly 60 frames in every second. If the PDC expects a data frame with a certain timestamp and does not receive it, the PDC will wait a preset time, anywhere from 50-200 milliseconds, on that data frame and discard the data for that frame if it does not arrive. Lost data is never desirable;

however, the larger problem arises in latency testing. If PMUSimulator is used to send data to a PDC to emulate a load on it, it is very undesirable to have the PMUSimulator to cause unnecessary delays by not sending the data frame packets on time. In that case, the delays will represent erroneousness data frames from the simulator and not the actual processing time of the PDC. Fortunately, PMUSimulator is open source so the user can edit the code. Various solutions were devised by altering the "udp_send_data" function in the PMUSimulator source code.

### 4.2.1 Hardcoded Usleep Subtraction

The first modification was to simply hardcode in a subtraction in the wait (usleep) time to cause the PMUSimulator software to send data frames faster. After some trial and error, the user can zone in on what value works best for the specific computer. The fallback of this method is that when the load on the UNIX system is increased, the time to generate the data frame will also increase, resulting in an inaccurate frame rate. Specifically, when more than one PMUSimulator is running, the processing time will be slower and it becomes nearly impossible to manually determine the correct correction values for all of the running PMUSimulators. Another downfall is that all computers will have different processing speeds so each user would have to zone in on the correction value, which is quite time consuming.

### 4.2.2 Automated Hardcoding Adjustment

Another solution was made to the code in the "Generate Data Frame" section in the file ServerFunction.c. The added code executes a command to timestamp before and after generating and sending 600 data frames. If it takes more time than expected to send 600 data frames, the code will signal the computer to start sending the data frames faster for the next 600 data frames and the process repeats until the frame rate is sending precisely 60 frames per second. If it takes less time than expected to send the data frames, the program will start sending the data frames slower. By displaying the amount of time that the program waits between sending data frames,

the user can see the exact amount of time that is needed for the particular system to send the right amount of data frames. Once this value is determined, the user can initialize the program to start sending the faster to allow the receiver to receive PMU data frames at the correct rate of 60 frames per second.

For this method, it is good to obtain an average over 600 data frames instead to obtain an average of what the actual frame rate is. It takes some time to execute the "gettimeofday" function as well as making the corrections. So, taking an average over a long window seems to be a good alternative. On the negative side, it takes a very long time for PMUSimulator to zone in on the right correction value. The smaller the increments to the correction value, the longer it takes to zone in on the right value, but will results in the closest value possible once it converges. On the other hand, when increments to the correction value are larger, the final convergence varies more but converges quicker. Overall, this solution will yield better results, but it takes much time to converge to the correct frame rate.

### 4.2.3 Running Average Frame Rate

The second best solution that was implemented involved computing a running average. This code would take the time at the beginning and ending of the send data loop and compute the correction time for every frame. This is similar to finding the average over 600 frames, but in this case the correction is added every frame. There is a memory of the correction so that the previous correction value is considered in all subsequent calculations. This technique can be used to close in on the exact 60 frames per second value quickly and use small increments in the correction value to arrive at a stable result. Using this technique, in time, the program will begin sending frames very closely to 60 frames per second. However, the same problem arises in that it takes a long time to converge to the correct sending rate. The figures below compare this autocorrecting program to the original frame rate.  All of the statistics of these plots are listed in

Table 2, which is shown after the plots. In Figures 21 and 22, PMUSimulator was sending data frames containing 3 phasors with the original and autocorrecting PMUSimulator settings respectively. Figures 23 and 24 show the original and autocorrecting frame rate for 10 phasors respectively. Finally, Figures 25 and 26 show the original and autocorrecting frame rate for 20 phasors respectively.

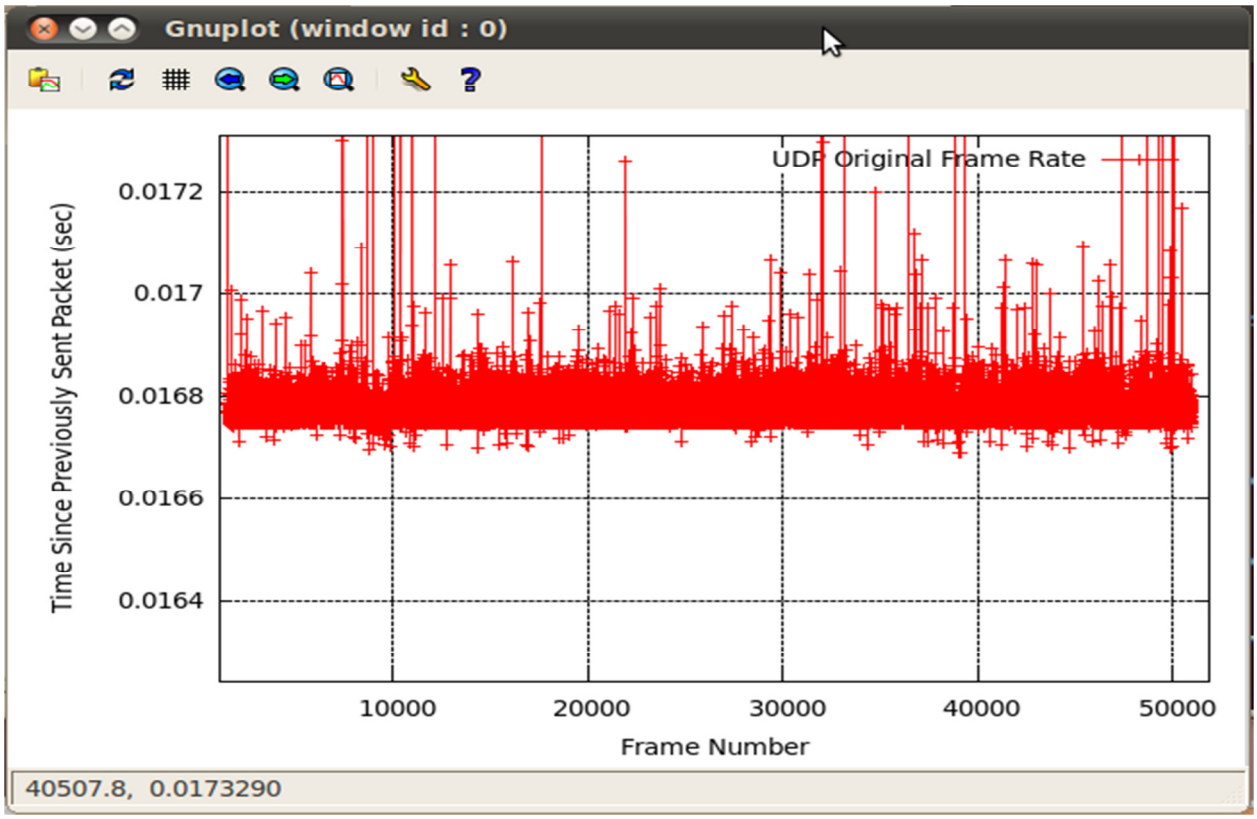


Figure 21 PMUSimulator 60 FPS measured Data Arrival for 3 Phasors

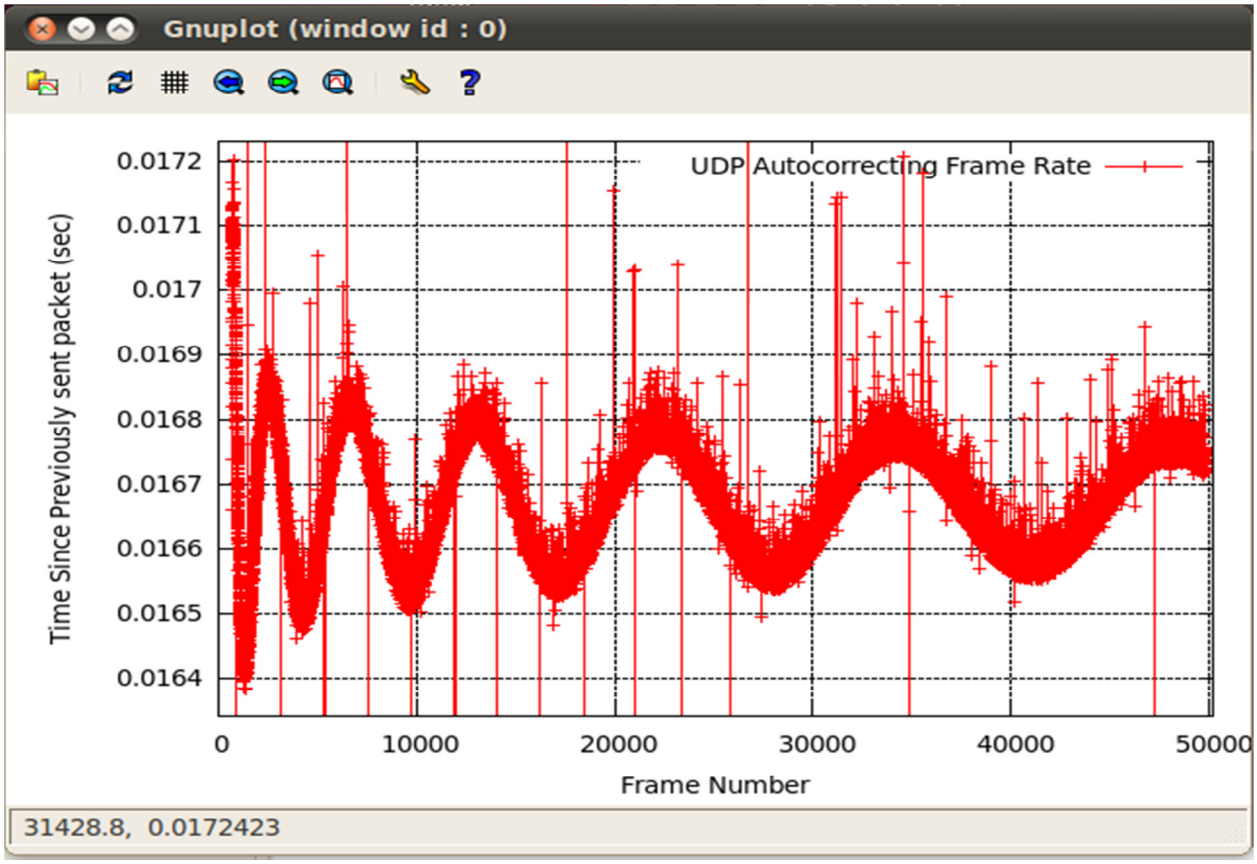


Figure 22 Autocorrecting PMUSimulator 60 FPS measured Data Arrival for 3 Phasors

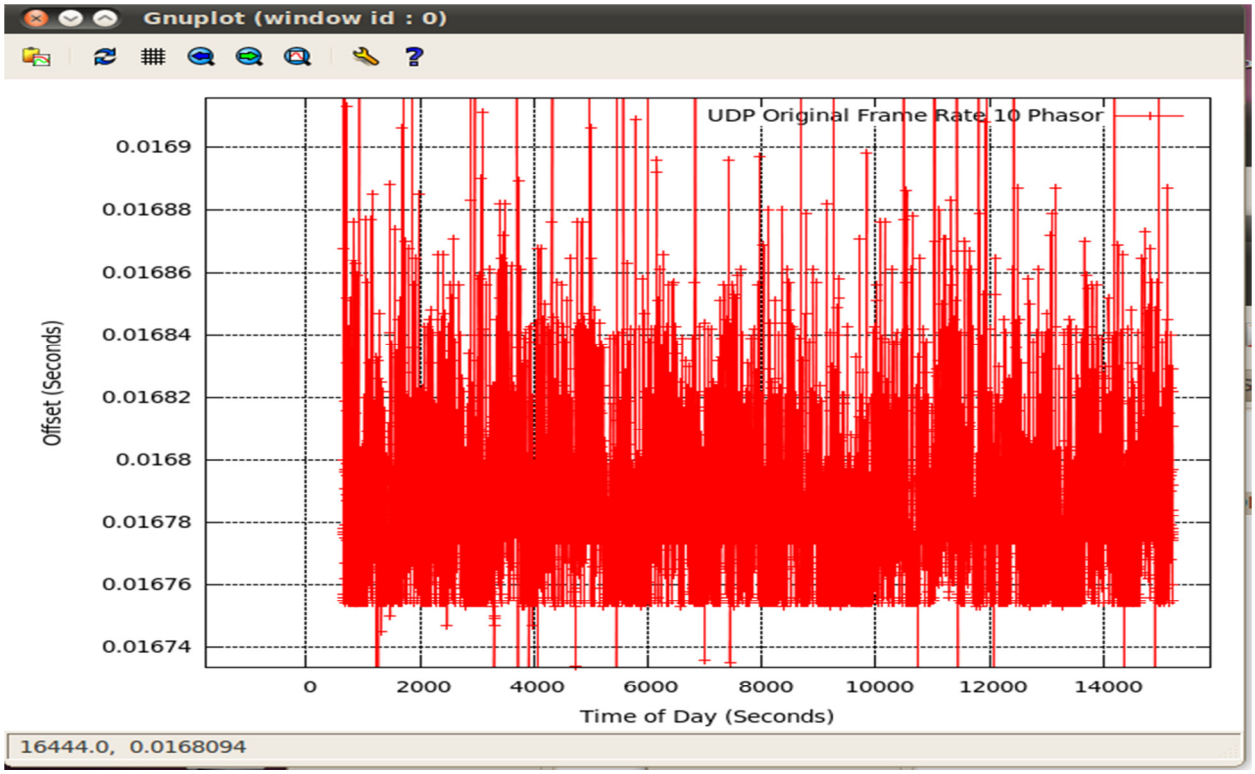


Figure 23 UDP Original Frame Rate 10 Phasors

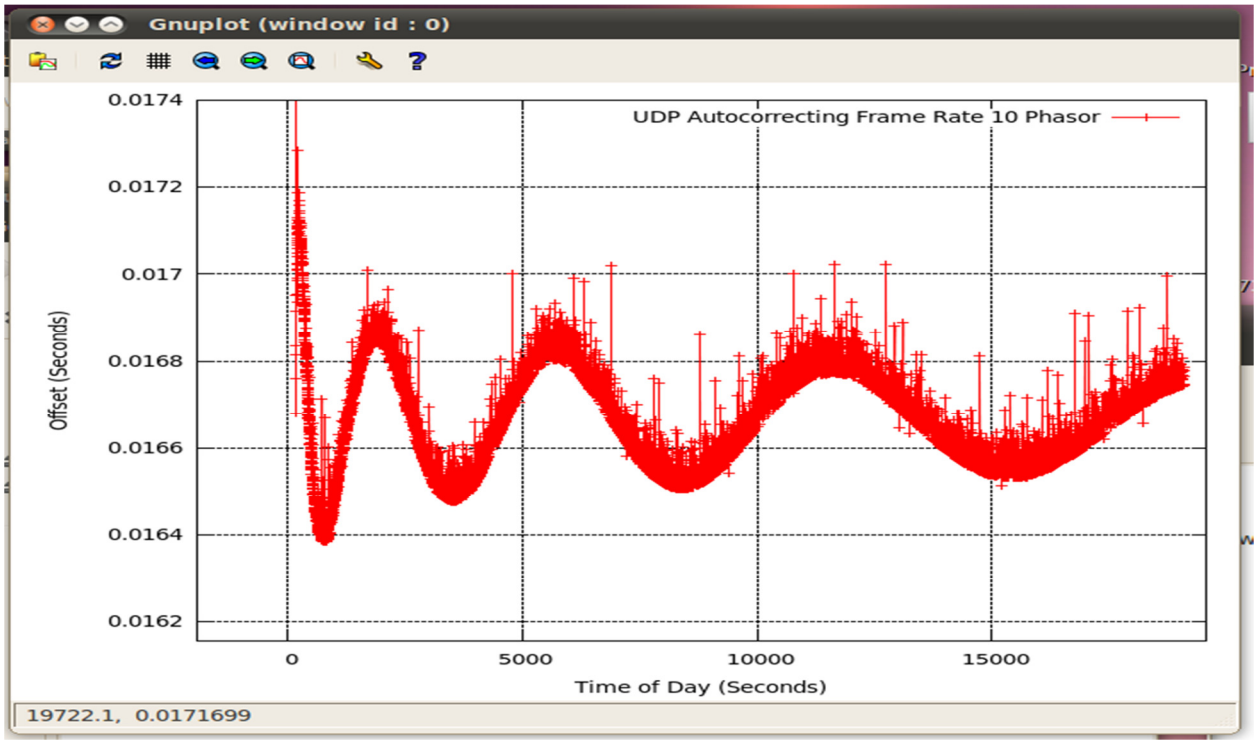


Figure 24 Autocorrecting PMUSimulator 60 FPS measured Data Arrival 10 Phasors

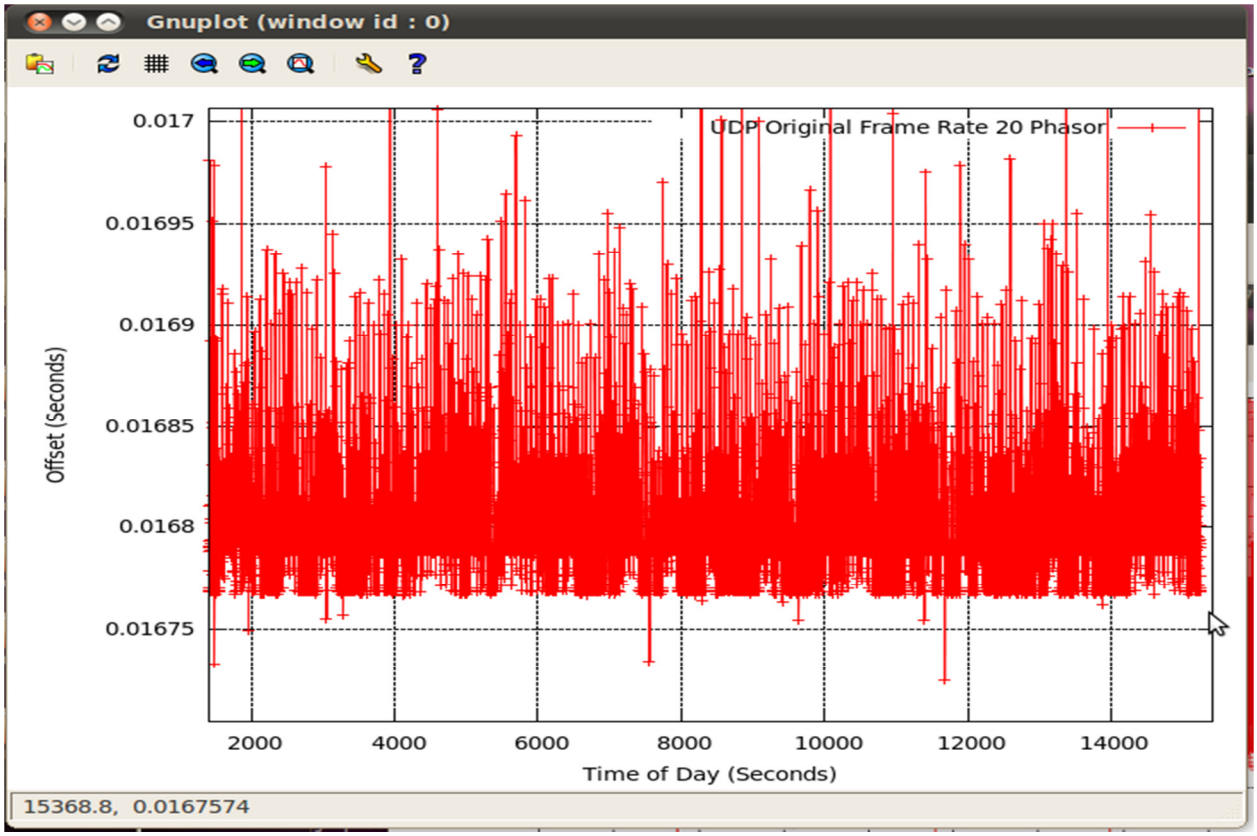


Figure 25 UDP Original Frame Rate 20 Phasors

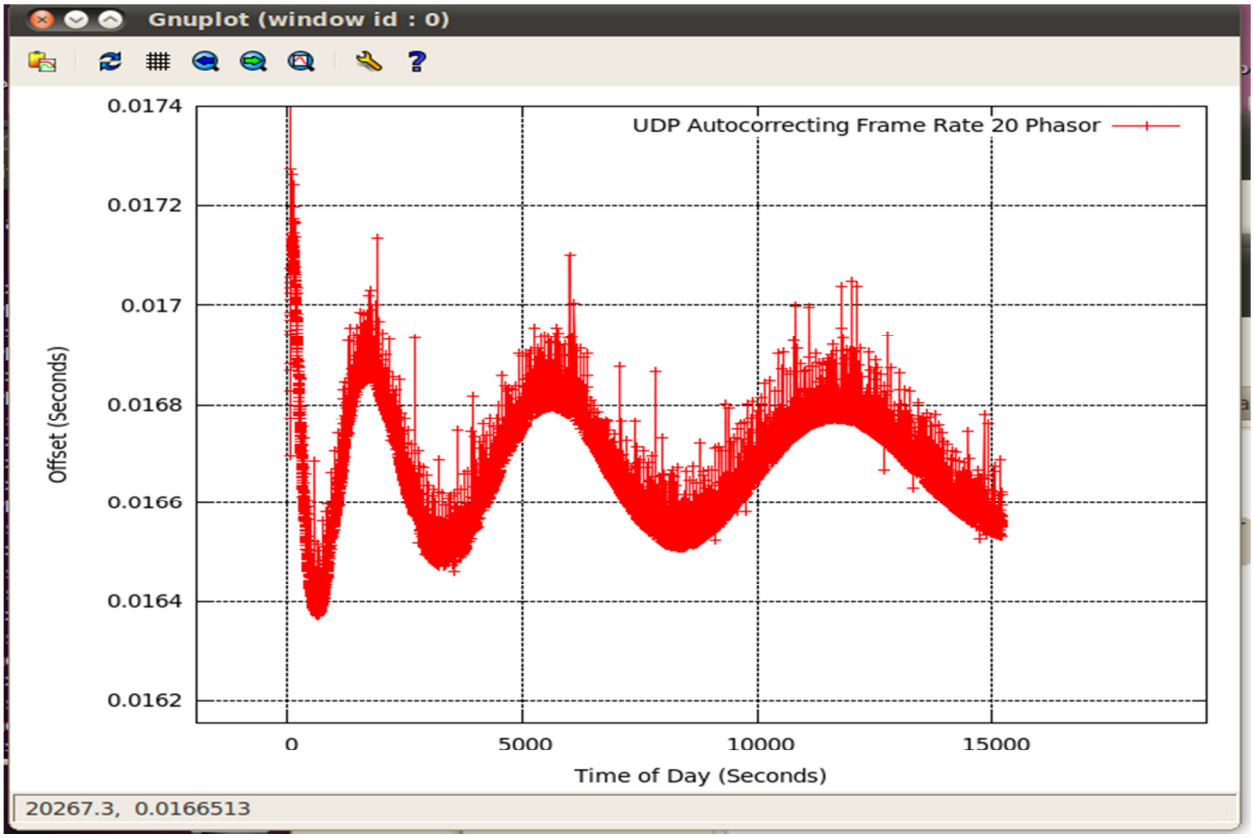


Figure 26 Autocorrecting PMUSimulator 60 FPS measured Data Arrival for 20 Phasors

From the plots above, it is evident that at any given period of time, the frame rate may be inaccurate. As time progresses, the figures show that the frame rate will eventually converge to the desired value. However, this convergence time is on the order of minutes, which is far too long for Simulating PMUs.

| | Original | | |
|---|---|---|---|
| **Phasors** | **Mean(ms)** | **Standard Deviation(ms)** | **Variance(ms)** |
| **3** | 16.7717 | 82.1156 | 6742.97 |
| **10** | 16.7779 | 47.3354 | 2240.64 |
| **20** | 16.7935 | 47.2467 | 2232.25 |

| | Autocorrecting | | |
|---|---|---|---|
| **Phasors** | **Mean(ms)** | **Standard Deviation(ms)** | **Variance(ms)** |
| **3** | 16.6593 | 56.0122 | 3137.37 |
| **10** | 16.6631 | 48.2174 | 2324.92 |
| **20** | 16.6691 | 47.5728 | 2263.17 |

Table 2 Original Vs Autocorrecting Frame Rate

The table above shows that the average frame rate over a long period of time is more accurate for the autocorrecting PMU than the original PMUSimulator. The table also shows that there is no noticeable difference in the standard deviation of the autocorrecting and the original software. Thus, autocorrecting is a solution for the average frame rate, but a very rough solution at best.

#### 4.2.3.1 Autocorrecting: 1 Commercial SEL PMU and 1 iPDC PMUSimulator:

When the technique of computing a running average is used, as described in section 4.2.3, the offset in sending time becomes even worse than the original setup as shown below.

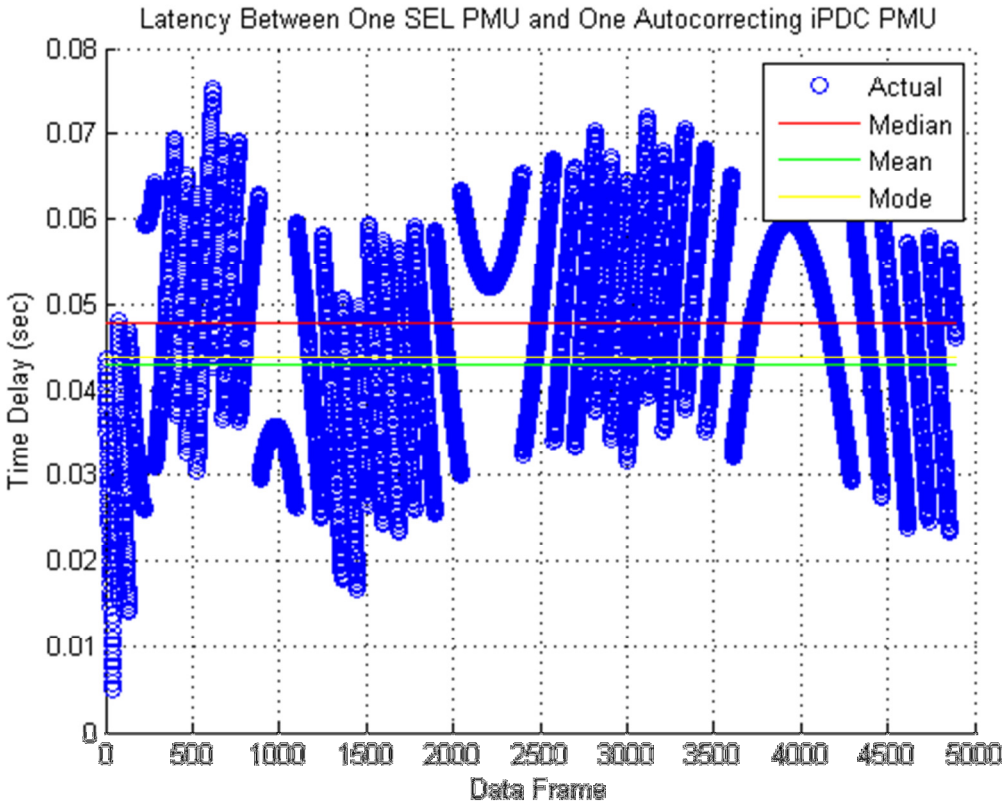


Figure 27 Arrival Time Delay Between Autocorrecting iPDC PMU and SEL PMU

#### 4.2.3.2 Autocorrecting: 2 iPDC PMUSimulators

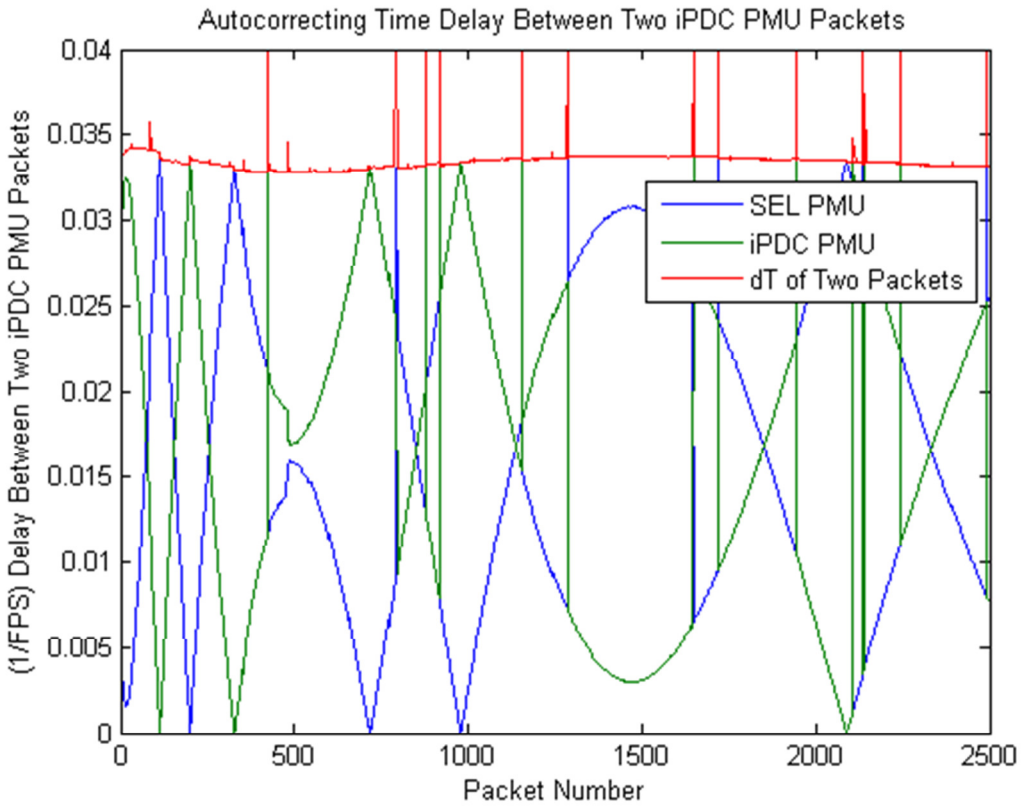


Figure 28 Arrival Time Delay Between 2 Autocorrecting iPDC PMUs

### 4.2.4 Frame Rate/Sending Time Solution

A Partial solution was devised by changing the fundamental way PMUSimulator sends data frames. Instead of beginning to send data at any time, the program can wait until the next second and begin after the next PPS. The program can generate the data frame, send it, and then calculate the time via ntp gettime() function and determine how many microseconds until the next interval when the packet should be sent. So, after the data frame is sent, and the time until the next sending interval is calculated, the program will use the usleep() command to wait the

specified time and generate the next data frame at that particular time, as shown in Figure 29.

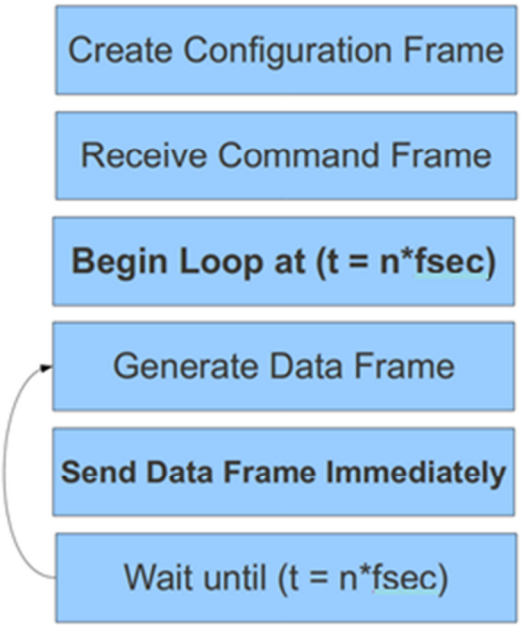


**Figure 29 Updated PMUSimulator Data Frame Generation**

Implementing this setup yielded a steady frame rate with respect to a SEL PMU as shown in Figure 30.

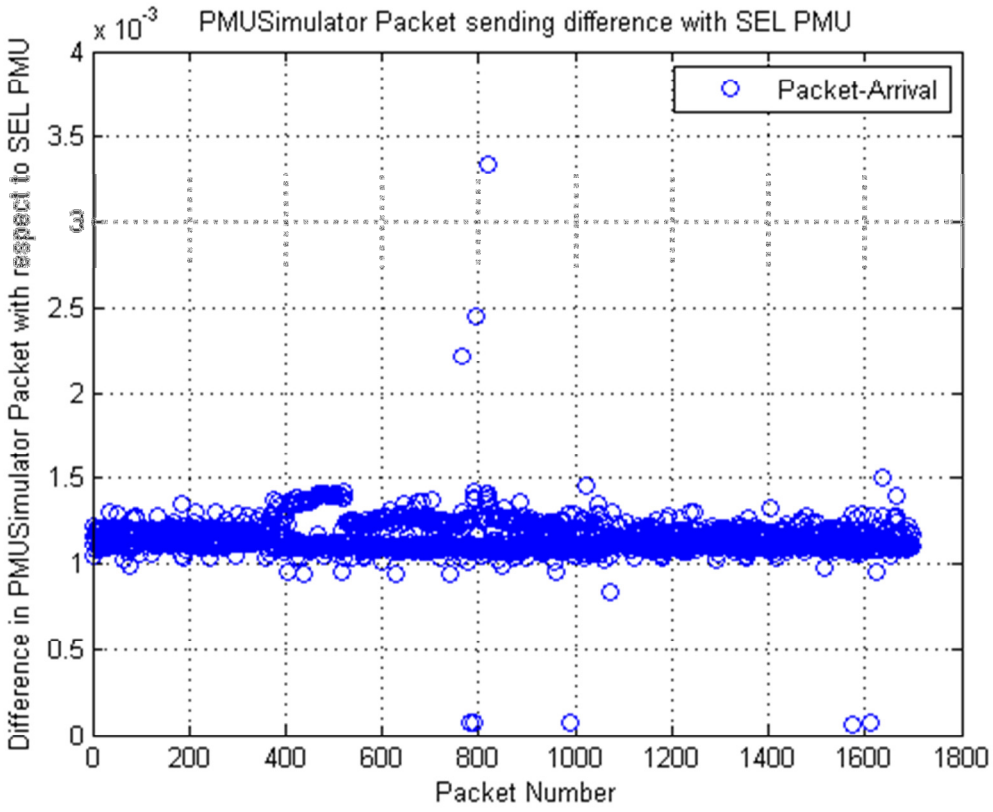


Figure 30 Updated PMUSimulator Packet Arrival Time Verses SEL PMU Arrival Time

With sending the data frames at specific points in time rather than focusing on the amount of frames per second, the drift between the SEL PDC and iPDC is diminished. The median offset between the SEL PDC and Real Time PMUSimulator was 1.124 milliseconds, which is acceptable for most PDC testing applications. The software can be programmed to begin generating and sending data frames at the next one second mark. The program then tries to send the ith data frame when time t = i*fsec.

By altering the sending time to send at specific times, the processing latency test applied should yield similar results as the SEL PMUs did. Figures 31 and 32 show the SEL-3378 processing latency of one and two Real Time PMUSimulators respectively. These two figures show that the packets are received by the PDC at consistent times as the SEL PMUs did.

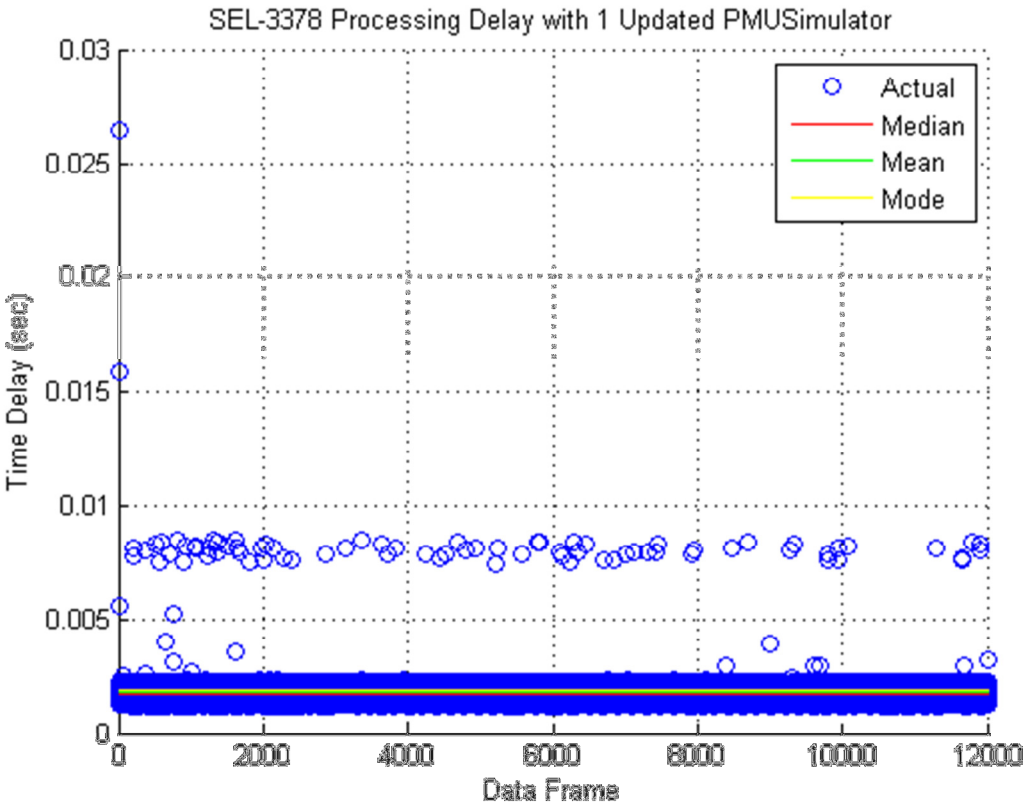


Figure 31 SEL-3378 Processing Delay: 1 Updated PMUSimulator

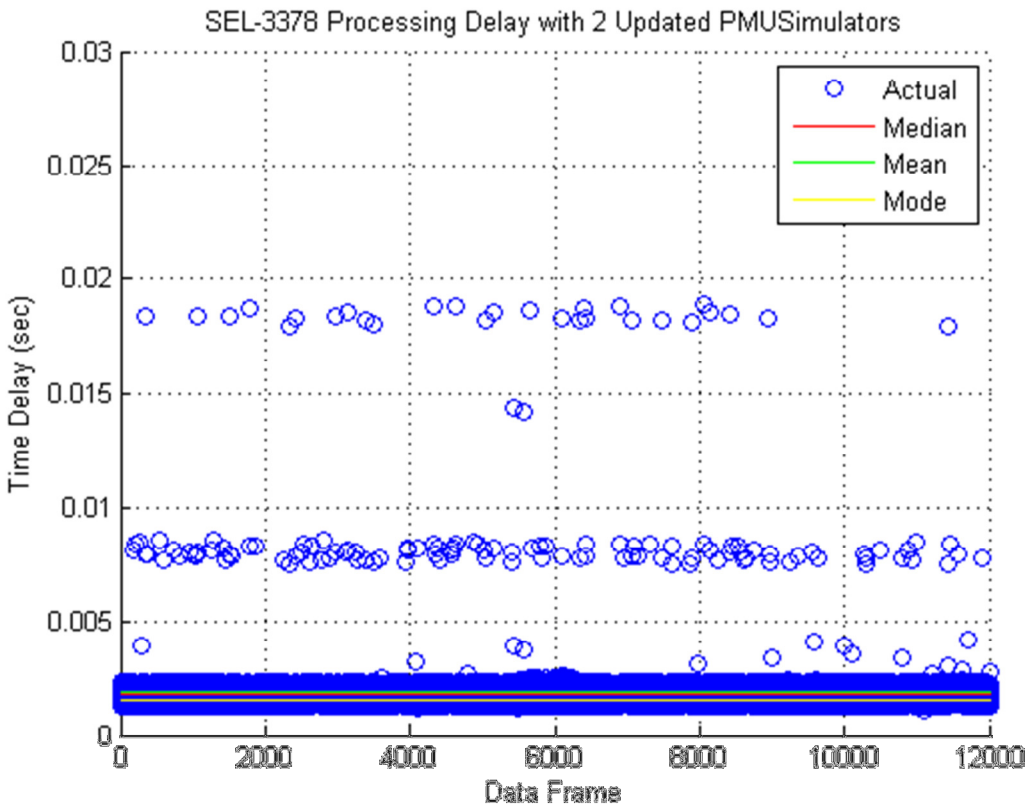


Figure 32 SEL-3378 Processing Delay: 2 Updated PMUSimulators

Figures 33 and 34 show that as more Real Time PMU Simulators are added to the inputs of the SEL-3378 PDC, the majority of the packets still arrive at the same point of time. However, at some points in time, the program will send inaccurate time stamps, which are represented by corresponding latencies of approximately zero seconds.

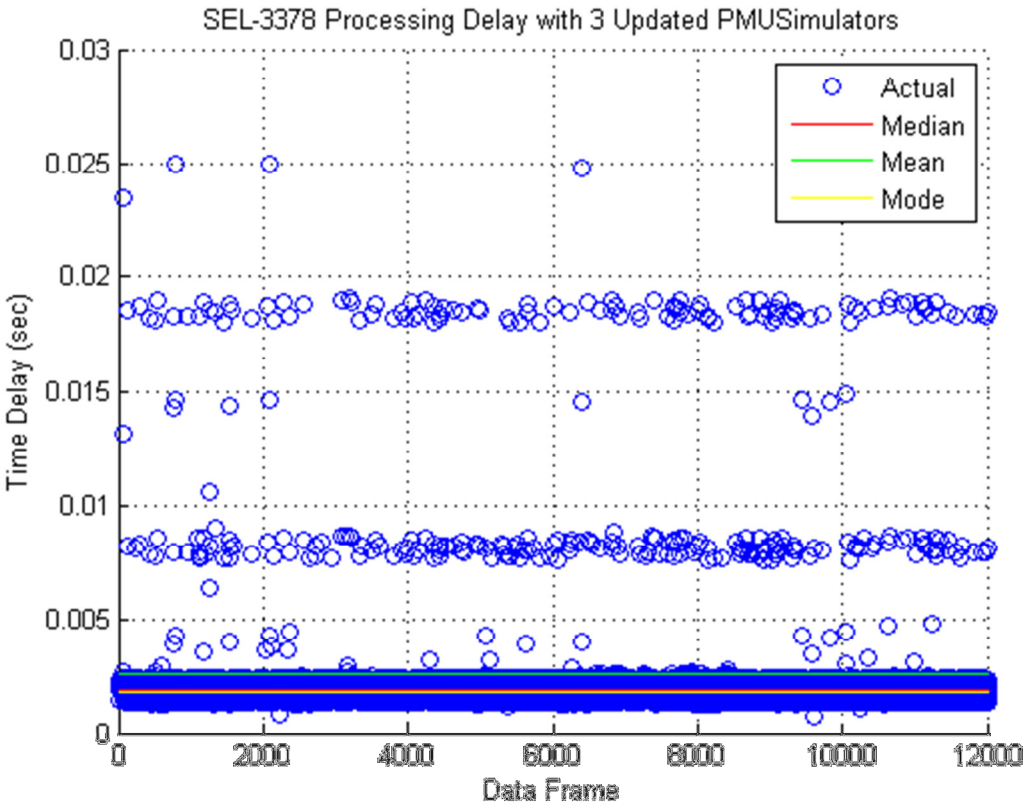


Figure 33 SEL-3378 Processing Delay: 3 Updated PMUSimulators

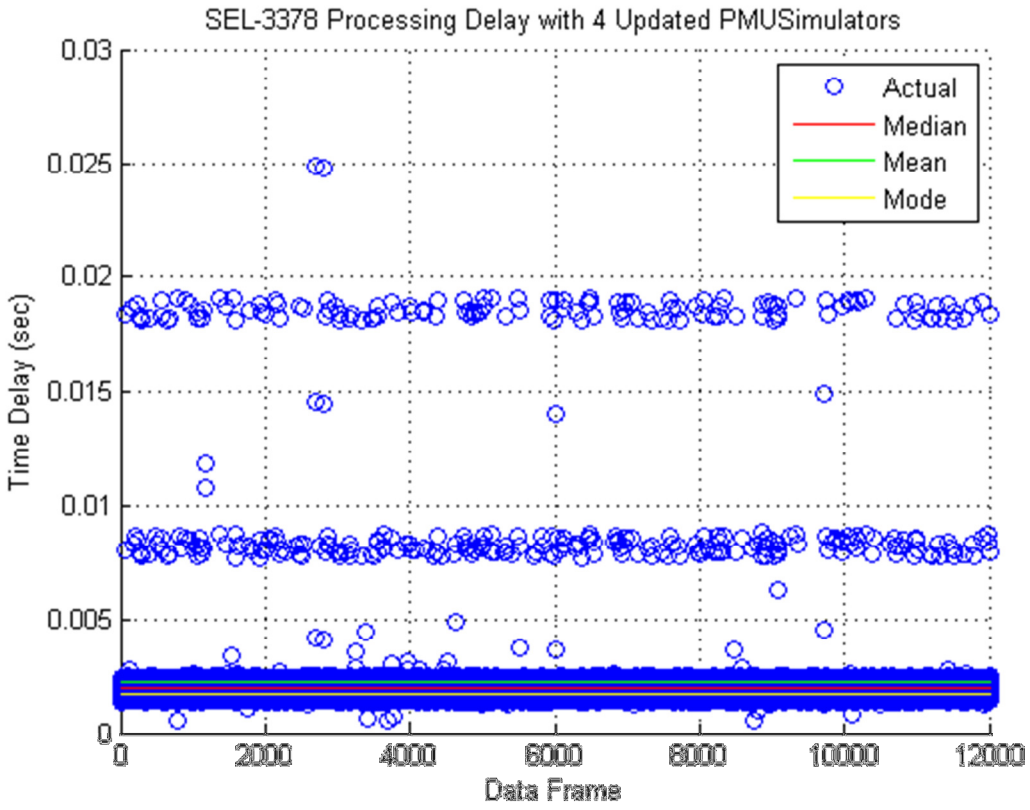


Figure 34 SEL-3378 Processing Delay: 4 Updated PMUSimulators

In order to determine the actual processing latency of the PDC, the latencies less than ten microseconds were removed from the corresponding array to calculate the mean, median, and mode accurately. Table 3 summarizes the results of the three different latency testing systems.

| Median Time Delay (s) | | | |
|---|---|---|---|
| **PMUs** | **iPDC** | **SEL** | **Real Time PMU Simulator** |
| 1 | 0.001734 | 0.001752 | 0.001739 |
| 2 | 0.002157 | 0.002126 | 0.001801 |
| 3 | 0.003514 | 0.003838 | 0.001900 |
| 4 | 0.005799 | 0.003149 | 0.001957 |

Table 3 Median Time Delay: iPDC, SEL, Updated

Table 3 shows that the Real Time PMU Simulators yield lower latency times than even the SEL PMUs. This means that the sending time of the Real Time PMUSimulator is more consistent than real SEL PMUs. Thus, the assumption that SEL PMUs send their data frames at exactly the same time is invalid. Figure 34 shows the latency computed for SEL, iPDC, and updated PMUSimulator.

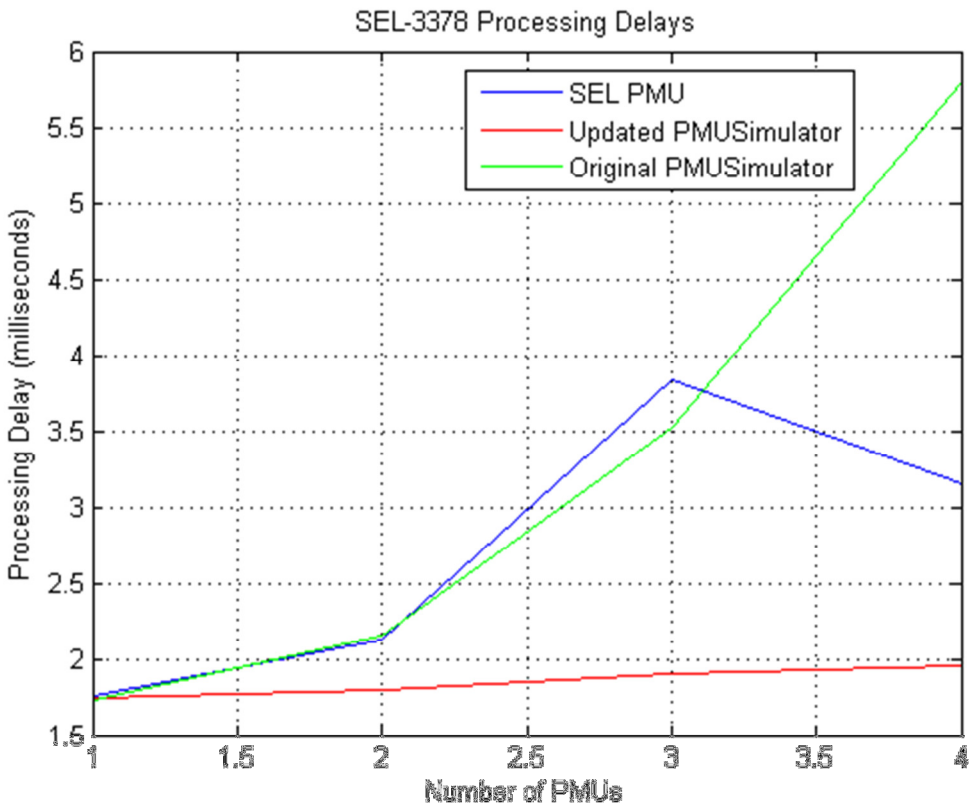


Figure 34 Median Latency Time Vs. Number of PMUs

## 4.3 Sending a Packet Late

A PDC has a specific waiting time for receiving a packet. The waiting time is the programmed time the PDC will wait to receive a PMU message before discarding the PMU data from a particular input channel for that time stamp. For example, if the waiting time for a PDC is 200 milliseconds, the PDC will wait 200 milliseconds from the arrival of the first data packet or from a fixed GPS time before discarding the data of the missing PMU. Consequently, it becomes important to test the programmed wait time in order to determine if the PDC will actually collect the information in those data frames when the PMU message arrives shortly before the cutoff time.

It is possible to modify PMUSimulator to simulate such a delay. The PMUSimulator will generate data frames at a constant rate and keep track of the packet number within the second by

the variable “I”. When the frame rate is 60 frames per second, the variable “I” will range from 0-59. Typically, PMUSimulator will generate the data frame, send it, and wait 1/(Frames per Second) and repeat the process. However, to simulate a late data frame, the user can insert a line of code that asserts, “if i=59 then wait x milliseconds”, where x is the integer value of the number of milliseconds desired. In this way, every second a late data frame will be sent to the corresponding PDC. The figure below shows the late packet delay that was captured from Wireshark.

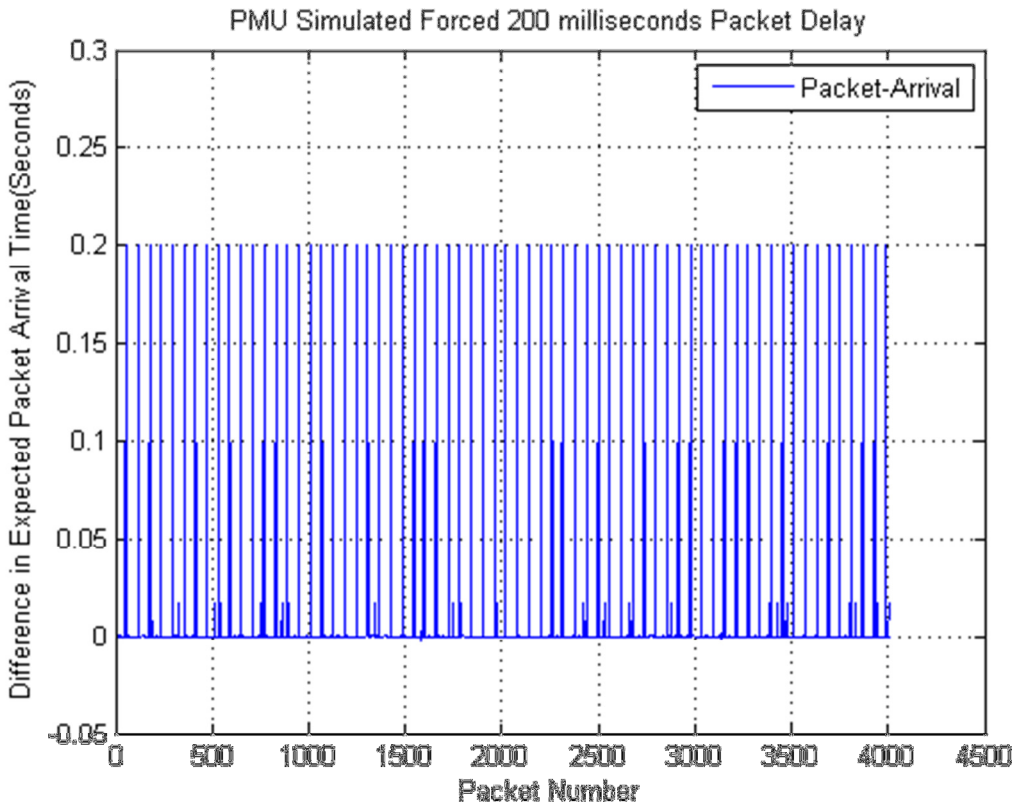


Figure 35 PMUSimulator Late Data Frame

In the figure above, “0” on the Y-axis denotes the expected arrival time of the data frame. Thus, the figure shows that on regular intervals a packet will arrive 200 milliseconds late to the destined PDC. Within the late data frames, the minimum difference in expected packet time was shown to be 200.2 milliseconds, while the maximum was 200.4 milliseconds. The waiting time

for every PDC can be different so the end user needs to calibrate the exact time delay in the code to obtain the desired results. Of course, these packet delay times are with respect to the PMUSimulator itself. In order for this to work properly, the data frames must be sent at specific times that the PDC expects. For instance if Real Time PMUSimulator sends a frame later or earlier than a real PMU would, this test will be more difficult to calibrate.

### 4.4 Bad Data Frame Checksum

On some occasions, a data stream from a PMU will become corrupted in which case the checksum received by the PDC will be invalid. The checksum is used by the computers to verify that the information it is receiving is correct. For testing purposes, it becomes desirable to introduce Synchrophasor data frames with an incorrect checksum in a Real Time PMU simulator. To alter PMUSimulator to complete such a task, the ServerFunction.c file in PMUSimulator has a commented section called "Calculate and insert the checksum value in data frame until now". In the original program, this section of code correctly computes the checksum. The user can simply change the number of shifted bits to introduce an incorrect checksum. The code original reads "df_chk >> 8 …..", which shifts the bit registers 8 times. If the "8" is changed to any other number, then the data frame checksum will be incorrect, which is desired for testing. To prove that the data frame checksum generated is in fact incorrect, the Real Time PMUSimulator was connected to PMU Connection Tester, as shown in Figure 36.

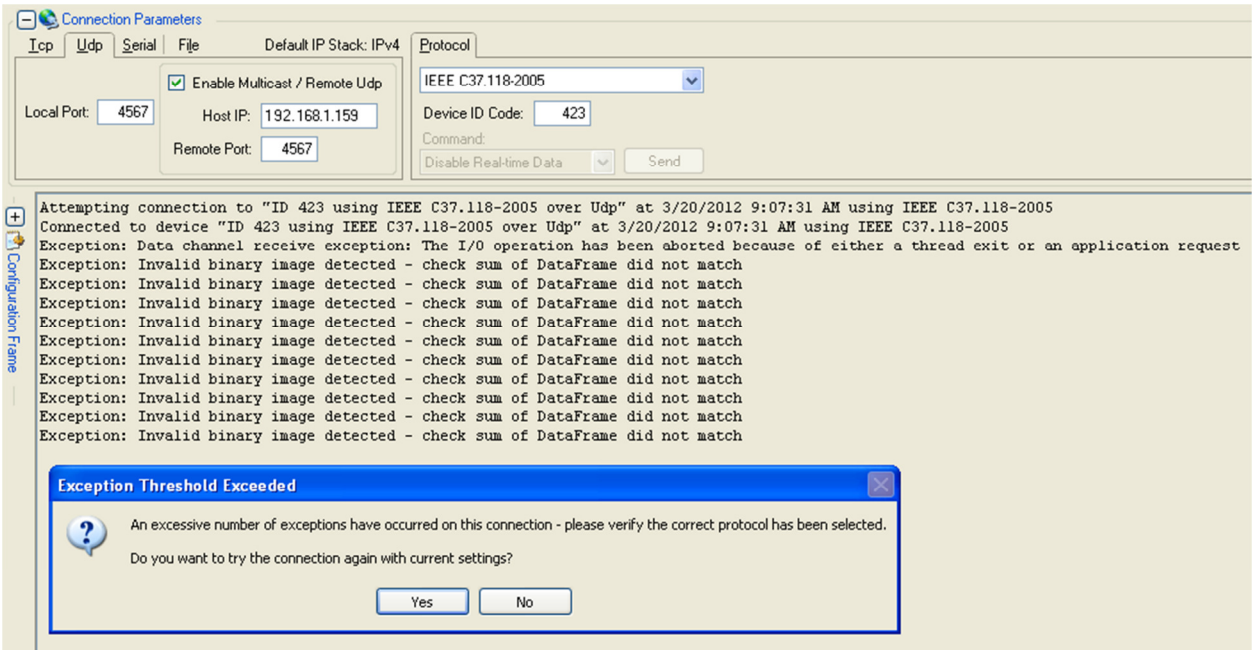


Figure 36 PMU Connection Tester Invalid Checksum

In the figure above, it is proved that the data frame checksum of the connected PMUSimulator is incorrect. For PDC's that accept ordinary UDP or TCP connections, this bad data frame test can be easily performed.

### 4.4.1 SEL-3378 Bad Data Frame Checksum

When a PDC receives a data frame with an incorrect checksum, the data should be flagged as corrupt and all of the measurement data of the frame should be set to zero. The figure below shows a PMU data frame with an incorrect checksum dissected by Wireshark. Wireshark detects that the checksum for the frame is wrong, as shown in Figure 37.

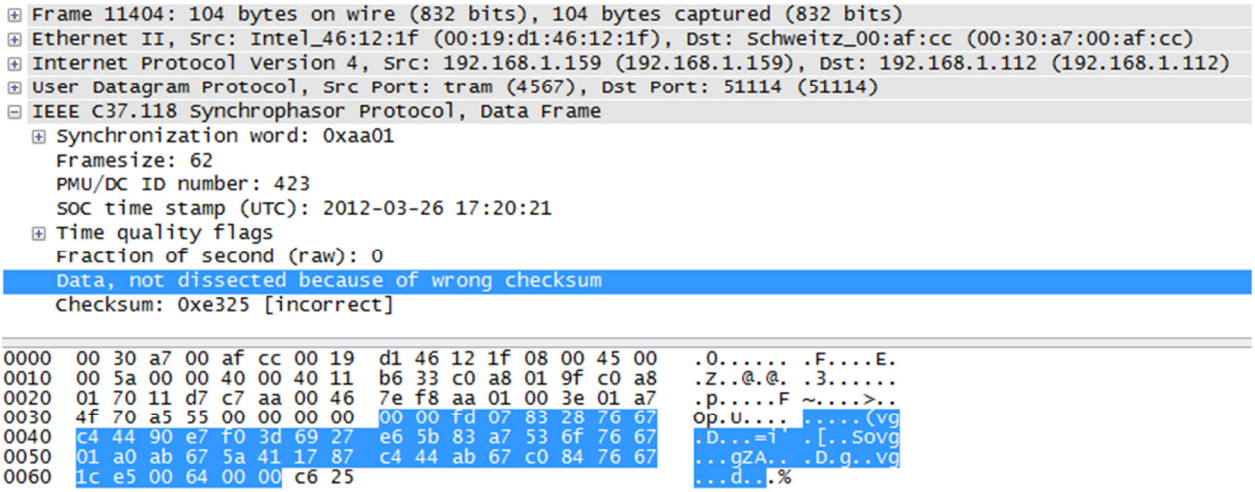


Figure 37 Wireshark Invalid Checksum

The data for the PDC data frame corresponding to the bad PMU data frame is shown in Figure 38.

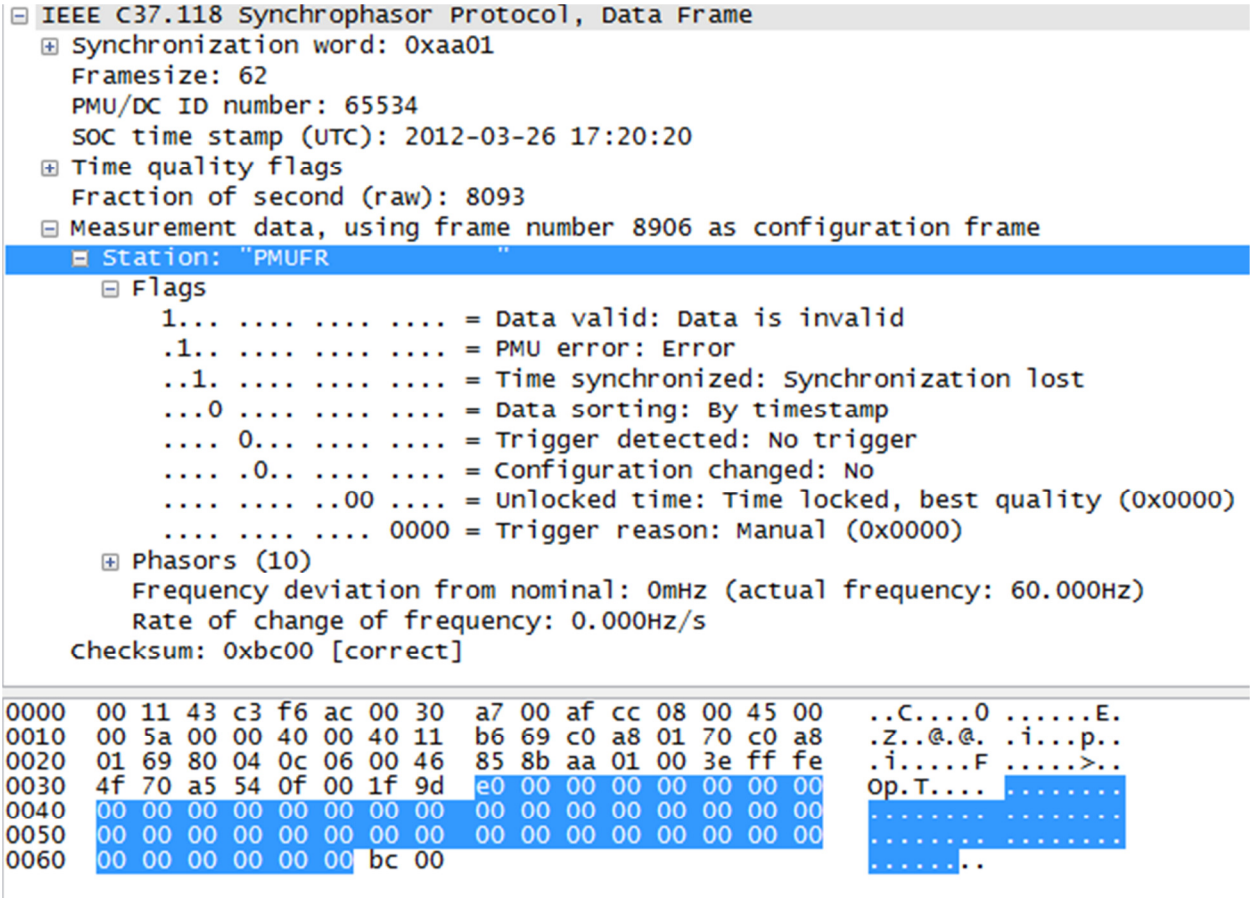


Figure 38 Wireshark Invalid Checksum (PDC data)

The PDC data frame sets the "PMU Error" bit to let the end user know that this frame is corrupted. The figure also shows that all of the information regarding the PMU is set to 0 by the PDC, where the original PMU data is shown in Figure 37.

To properly test a PDC, it is desired to have test equipment that can send bad data at known time intervals to evaluate the response of the PDC. Real Time PMUSimulator can be programmed to send and incorrect checksum to test the correct operation of a PDC. In order to find the bad data easily when using Wireshark to evaluate the response, Real Time PMUSimulator was programmed to send a bad checksum only when the fraction of second was 0, otherwise the correct checksum was generated. This reduced the confusion of finding PDC frames corresponding to PMU frames.

The Real Time PMUSimulator software sends data frames with a time base of 1000000. This means that the FSEC values will be calculated by 1000000*(Frame Number/Frame Rate). The first data frame at the turn of the second should have an FSEC value of 0. The second data frame should have an FSEC value of 16666. The sixtieth data frame would have an FSEC value of 1000000, but at that point the SOC value increases by one making the FSEC value 0 again.

When PMU data frames have bad checksums, the PDC output fraction of second time is also corrupted. It was observed that all of the FSEC timestamps were less than 10000, where they should be between 0 and 1000000, as explained earlier.

### 4.4.2 SEL-3373 Bad Data frame Checksum

When Real Time PMUSimulator sent a bad data frame checksum to the SEL-3373 over TCP connection, the results obtained were different than expected. When the SEL-3373 receives bad data, it does not send any data as an output. If the checksum is not correct, the SEL-3373 will treat the packet as if it was random information on the network. The figure below shows that frame number 365 has a “wrong checksum”. In the rest of the Wireshark file, the PDC output data will appear directly after the input data. However, the figure below shows that the PDC does not send any data whatsoever as an output when the input data is corrupted. There is no corresponding PDC timestamp for the bad data anywhere in the capture file. This incorrect operation of the SEL-3373 in the lab could not be observed without the ability to generate bad data at known intervals.

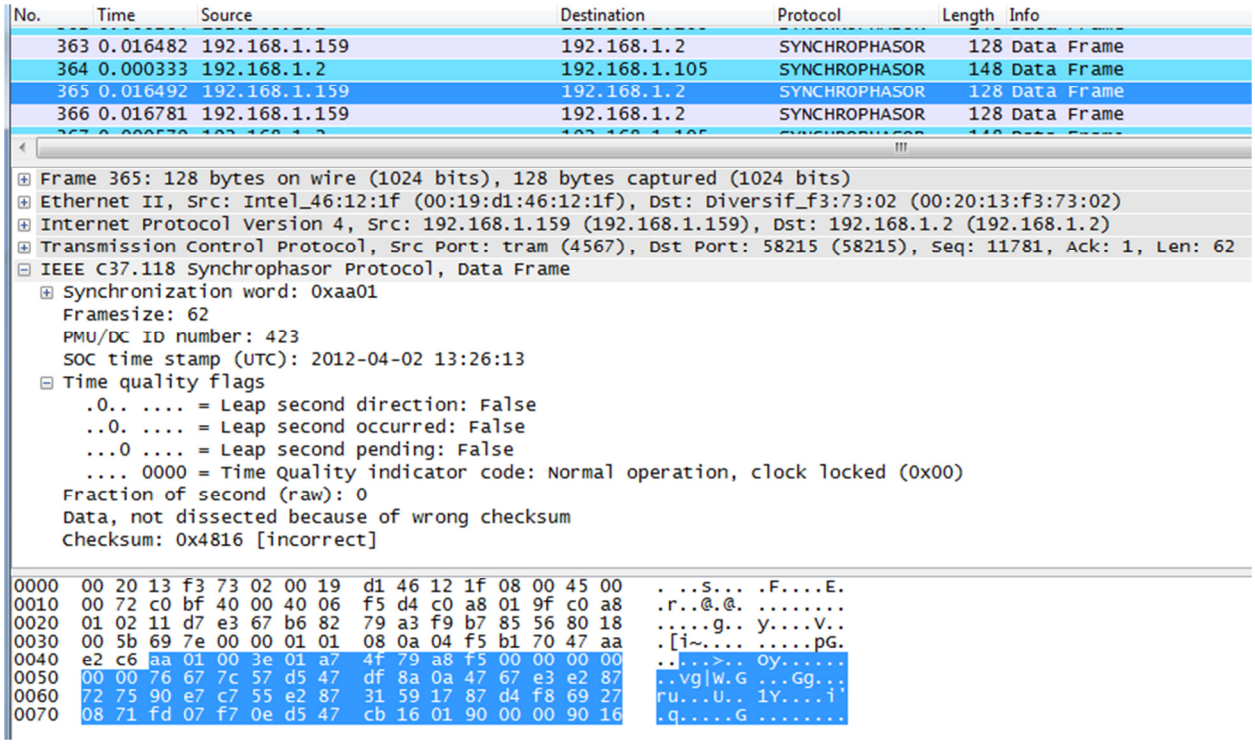


Figure 39 Invalid Checksum Wireshark SEL-3373

## Chapter 5: Conclusions and Future Work

The use of an open source PMU simulator as the base for the developed Real Time PMUSimulator tool is described in Chapter 3. The original PMUSimulator software produced a slower frame rate than the desired frame rate, lacked synchronization to a precise time signal, and did not have the ability for data integrity checks. As available, PMUSimulator could not be used for PDC testing or synchronized with other PMUSimulators.

The developed Real Time PMUSimulator is implemented on a UNIX system with NTP time synchronization using a Garmin 18x LVC GPS receiver. With NTP, the UNIX system synchronized to the actual time within the range of tens of milliseconds, which is not as accurate as a PMU, but sufficient for simulations of correct reporting rates from a PMU required to evaluate PDC operation.

The original PMUSimulator software produced a slower non time synchronized frame rate. Several solutions were tested for this problem before completely correcting it by sending data frames at specific intervals in time instead of random points in time. We have used an open source software, iPDC, and enhanced it to accurately simulate a real time PMU.

The aim of this work was to develop a tool to test the correct operation and limitations of commercial PDC and other devices associated with multiple PMU data collection. Three characteristics of the PDC needed to be evaluated specifically: data delay with a large number of PMUs, PMU data wait window operation, and bad data handling. Three tests were successfully created, implemented, and used to evaluate the operation of a PDC with the enhanced Real Time PMUSimulator:

a) Latency testing was successfully performed using the "Sending Time Solution", which yielded more accurate results than the current method of using commercial PMUs because the Sending Time Solution sends the data frames at a specified time on the order of microseconds.

b) A method was designed to send a data frame packet late to a PDC to test the actual waiting time of the PDC, which can be used to determine the accuracy of the programmed waiting window.

c) An invalid data frame checksum test can be generated once a second and sent to the PDC's for evaluation.

In addition to the developed tests, the device and its synchronization procedures will allow for the operation of multiple Real Time PMUSimulators to test PDCs under maximum data load. The developed Real Time PMUSimulator and the developed procedures were successfully applied to a SEL-3378 and a SEL-3373 PDC.

## Future Work

Since PMUSimulator can generate Synchrophasor data frames by reading CSV files, it is possible to recreate a time synchronized PMU event player that can use data from a simulation program like PSS/E or PSLF to evaluate the operation of a PMU enhanced protection scheme or to evaluate the effect of the PMU communication network in control or protection operations. PSS/E can be programmed to simulate certain conditions in a given power system. The software can record the magnitude, angle, and frequency at the rates produced by PMUs in a CSV file. With minor alterations to the CSV file, several Real Time PMUSimulators can play back the

simulated measurements at a fixed time and send the output to a PDC, in order to determine the response of the PDC to a simulated event.

It is also necessary to fully develop the wait time PDC test as presented in section 4.3. This thesis has shown that it is possible for the Real Time PMU simulator to send a late data frame to a PDC. However, there is currently no valid technique for mining through the data and determining the absolute latest wait time a PDC will undergo. It is possible to check each late frame individually, but not the overall results.

Finally, additional testing is needed in any new versions of the iPDC software package, as more bugs in the software are likely to be found as the developed Real Time PMUSimulator is used for more diverse operations.

## References


1. NTP: The Network Time Protocol. http://*www.ntp.org*. June 3, 2011

2. Phadke, A.G., Thorp, J. S. , *History and Applications of Phasor Measurements.* IEEE, pp.331-335, 2006.

3. Phadke, A. G., *Synchronized Phasor Measurements~A Historical Overview.* IEEE, pp. 476-479, 2002.

4. Wireshark Network Protocol Analysis software, *http://www.wireshark.org/.*

5. *IEEE Standard for Synchrophasors for Power Systems.* IEEE Standard C37.118-2005 (Revision of IEEE Std. 1344-1995), 22 March 2006.

6. Thomas, M. K. *Implementation of the Security-Dependability Adaptive Voting Scheme*, in ECE 2011, Virginia Polytechnic Institute and State University: Blacksburg.

7. Schweitzer Engineering Laboratories. *SEL-3378 Synchrophasor Vector Processor Instruction Manual,* 2010.

8. Schweitzer Engineering Laboratories. *SEL-421 Relay Protection and Automation System Instruction Manual,* 2011.

9. Schweitzer Engineering Laboratories, *http://www.selinc.com.*

10. PMU Connection Tester, *http://www.pmuconncectiontester.codeplex.com*.

11. Wireshark. Wireshark Foundation, *http://www.wireshark.org.*

12. Schweitzer Engineering Laboratories. *SEL-3373 Station Phasor Data Concentrator Instruction Manual, 2011.*

13. iPDC/PMUSimulator, http://ipdc.codeplex.com.

14. Quint, R. D. Practical. *Implementation of a Security-Dependability Adaptive Voting Scheme Using Decision Trees,* in ECE, 2011. Virginia Polytechnic Institute and State University: Blacksburg.

15. Dekhane, K. S. *The Virginia Tech Phasor Data Concentrator Analysis and Testing System*, in ECE, 2011. Virginia Polytechnic Institute and State University: Blacksburg.

16. Garmin GPS 18x LVC http://www8.garmin.com/support/agree.jsp?id=4055.

17. FreeBSD. The FreeBSD Project, http://www.freebsd.org/.

18. Ubuntu, http://www.ubuntu.com.

19. Timers, Timer Resolution, and Development of Effi. Microsoft. http://www.microsoft.com/whdc/system/pnppwr/powermgmt/Timer-Resolution.mspx

**Definitions and Acronyms**

**CSV –** Comma Separated Value

**FSEC –** Fraction of a second.

**GPS –** Global Positioning System. A satellite based system for providing position and time.

**GUI –**Graphical User Interface

**IEEE C37.118 –** A standard used to define the particular formats that can be transmitted and interpreted by all PMUs.

**iPDC –** Open Source Phasor Data Concentrator

**Matlab –** Commercial numerical environment and programming language.

**NMEA –**National Marine Electronics Association

**NTP –** Network Time Protocol.

**PDC –** Phasor Data Concentrator. A device that collects phasor data and discrete event data from PMUs and other PDCs and transmit data to other applications.

**PMU –** Phasor Measurement Unit. A device that samples analog voltage and current data in synchronism with a GPS-clock.

**PPS –** Pulse-Per-Second. A signal consisting of a train of square pulses occurring at a frequency of 1 Hz, with the rising edge synchronized with UTC seconds.

**PTP –** Picture Transfer Protocol

**SEL –**Schweitzer Engineering Laboratories

**TCP –** Transmission Control Protocol. Packets of data transferred over a network

**UDP –** User Datagram Protocol

**UNIX –** A computer operating system

**UTC –** Coordinated Universal Time. It respresents the time-of-day at the earth's prime meridian.