EasyChair Preprint

№ 2335

Reliable Timestamping in Windows

Kyle Rebello

January 8, 2020

Reliable Timestamping in Windows

Kyle Rebello
Northrop Grumman
Orlando, FL
Kyle.Rebello@ngc.com

ABSTRACT: The Northrop Grumman Corporation - Mobility Air Forces (MAF) Distributed Mission Operations (DMO) team located Orlando, FL participates in Air Refueling Events by interacting and testing other simulators across the country using our interactive test tool. The test tool along with other products and services we provide are built and developed using Windows machines. The MAF DMO team currently has to support Red Hat Linux environments for the test tool mentioned above because of Windows' inability to natively adhere to the MAF DMO Time Synchronization standards. There is currently a financial benefit in being able to use the test tool on Windows instead of Linux, due to the current yearly Red Hat Linux license and maintenance cost of supporting multiple production environments.

This paper discusses how we plan to transition our application from Linux to Windows allowing us to create the Windows only production environment we desire. This paper provides solutions to the following problems that occur when using Windows instead of Linux. The first problem is getting Windows to sync to a GPS time source within 100 microseconds as described in our MAF DMO Time Synchronization Standards. The next problem is getting Windows to grab the system time, put it in a PDU and send it quick enough so that the system time we grabbed is still within the 100 microsecond time range when sent. Finally, this paper discusses how we plan to implement the solutions found to achieve the Windows only production environment throughout our office.

1. Background

Over the course of many years we have been developing and using a Graphical User Interface (GUI) application called EZ-Fly, which is used to simulate an entity involved in an air refueling event. EZ-Fly has the ability to simulate all three entities involved in an air refueling event which are Tanker, Receiver, and Boom. EZ-Fly accomplishes this by sending DIS network traffic specific to the entity it is simulating, along with receiving DIS network traffic from the other entities in the air refueling event. The sending and receiving of DIS network traffic allows EZ-Fly to participate in the air refueling event as if it were a real simulator. EZ-Fly is a cross platform application which means it can be built and run on both Windows and Linux. While the code for EZ-Fly is developed, built and tested on Windows machines, EZ-Fly is run on Linux machines during the air refueling test events. The goal is to be able to run an air refueling test event using EZ-Fly on Windows. If this is successful then we would be able to switch to the Windows only environment we are striving towards. In order to run EZ-Fly on Windows during a test event we have to solve a few Windows specific problems.

It is important to note that EZ-Fly interacts with flight simulators that are located in multiple time zones across the world. To be able to accurately and successfully participate in an event with live simulators around the world the simulation must adhere to some common time synchronization rules. The first rule is that we must sync and stay synced to within 100 microseconds of a GPS-locked time source. This is important because all of the entities in a test event send Entity State PDUs which contain a timestamp along with location and speed information. Dead Reckoning is used to determine the position of entities in the battle space. If a simulator is not properly synced to a GPS time source then the Dead Reckoning calculations can produce erroneous results which can invalidate the test event. The next rule is that a computer participating in an event needs to achieve sync to within 100 microseconds of the GPS time source within 20 minutes of a cold boot. This is important because if for some reason a computer needs to be restarted prior to or during an event, we can ensure that the computer can be reintroduced to the test event as quickly as possible. In order to satisfy the above rules, a current guideline is that the computer must poll the GPS time source at least once every 16 seconds. If the computer can stay within 100 microseconds of its GPS time source with a higher poll number then this guideline can be relaxed appropriately.

After we can prove that our Windows machine is synced to within 100 microseconds of its GPS time source then we can begin the next step of getting EZ-Fly to send out its time information accurately enough to stay within 100 microsecond threshold. When EZ-Fly is started it is configured with a default callback time of 60Hz, which means it will call a particular function 60 times per second. The function that it calls takes in two parameters, the first being a static value of 0.016667 called frame time, and the second parameter being the callback time which is the system time at which the function was called. The frame time is what is added to our perfect timestamp value each frame and then put into the PDU to be sent out to the other simulators. The reason our perfect timestamp is statically incremented is because we want to make sure that the time we send out is periodic within numerical precision. In order for this time stamping to work properly we need to make sure that the function is getting called at as close to 60Hz as possible. To call the callback function at 60Hz, the periodic task thread execution behavior is regulated.

Linux is able to sync to a GPS time source within 100 microseconds and is also able to call a callback method at the desired frequency, but Windows natively cannot accomplish either of these tasks. This means that Windows has different scheduling priorities than Linux. Windows is used by millions of users and is the most used operating system globally. Windows focuses on user experience operations where Linux focuses more on lower level processes. We will be using our Windows and Linux machines located in our lab to try and get the Windows machine to behave with respect to timing more like our Linux machine. Since we know that the Linux machine performs well during air refueling test events, we will frequently be comparing results produced on the Windows machine to results produced on the Linux machine.

2. Syncing to a GPS Time Source

Windows natively comes with its own time service called Windows Time Service (Win32tm), which can be modified slightly and has had accuracy improvements over the recent years. We decided to try this solution first as it would be the easiest and quickest to implement. Using the Registry Editor, the user can change Windows Time Service settings such as setting a GPS ntp time source along with a min and max poll interval. The poll interval just tells the application the min and max amount of time it can send a poll packet to try and sync the system clock to the provided time source. A problem that immediately introduced itself was that the smallest min poll we could provide was 32 seconds which was double the value that we are supposed to poll at. As stated previously, this requirement is more of a guideline meaning that as long as we are synced to within 100 microseconds we can poll at whatever value we determine is necessary. The w32tm command with the stripchart argument was used to monitor the local time for a few minutes. The stripchart argument requires the user to enter an IP address of a time server to compare against, then it compares the local system time to the time servers system time and outputs the difference between the two times. The closest the times got to each other was 250-300 microseconds apart with most of the results up around 1 millisecond. Changing priority levels of the Windows Time Service thread and modifying other variables in the Registry Editor did not produce different results. Windows Time Service has come a long way but it still is not quite as precise as we need it to be, so we decided to look for another solution.

Since the native Windows Time Service did not work, the next step was to find a third party solution. After doing some research and talking to coworkers the application called Meinberg kept popping up. This is a free software that implements the Linux ntp library to work in Windows. When Meinberg is installed it disables the Windows Time Service and disables it from starting when the computer is rebooted and instead starts and runs itself when the computer is rebooted. Meinberg has an ntp.conf file that gives the user the same control over the ntp settings that they would have in Linux. The application allows the user to set a GPS time source along with any min and max poll desired. To satisfy the requirement of polling the GPS time source at least once every 16 seconds, the min poll was set to 8 seconds and the max poll was set to 16 seconds. The final thing before starting the application was to turn iburst mode on which enables faster clock synchronization. After starting the Meinberg application the time difference was again monitored using the w32tm command with the stripchart argument. Within a couple minutes the w32tm command was showing times within 10 microseconds which was well within our MAF DMO Time Synchronization Standards. The computer was then rebooted to see how long it took to return to being within 100 microseconds of the GPS time source and it impressively took under 10 minutes. With these results we proved that Meinberg was able to sync a Windows machine to a GPS time source and satisfy the MAF DMO Time Synchronization Standards. Unfortunately, after all the testing was completed we found out that Meinberg was not allowable per our Risk Management Framework. It is worth mentioning that Meinberg is the best solution, so if it can be used it should be used.

After Meinberg had been shot down we located another third party time synchronization client called PresenTense which was listed on our approved product list. This software is produced by orolia (formerly Spectracom) which is the brand of our GPS time servers. This software is not free however, and would require us to purchase three IP-specific licenses that would cost $1,200 each or about $3,600 per year. PresenTense does however provide a free 30-day trial license that we downloaded and began testing with. After PresenTense is installed it does not disable Windows Time Service like Meinberg did, so this has to be manually disabled through either the PresenTense GUI or the Services tab located in Task Manager. The user also should manually disable Windows Time Service from starting at reboot because time syncing will use Windows Time Service by default and prevent PresenTense from running properly. The PresenTense GUI can be a little difficult to use but it did allow similar control over the ntp settings as Meinberg did. It does allow the user to set iburst mode along with setting the service priority to high while also allowing the user to set a GPS time source along with a min and max poll. One problem is that while the min poll can be set to 16 seconds, the max poll could only be set to 32 seconds which would not meet our polling requirement. After starting the PresenTense software and running the w32tm command with the stripchart argument for a few minutes we observed something interesting. While PresenTense was polling at 16 seconds we were within our 100 microsecond target range, but after a few minutes the polling rate switched to 32 seconds and we were only occasionally within our 100 microsecond target range. After an email to the PresenTense support team explaining the above situation and they did respond with a solution. It turns out that after the application is started the user can go into the Registry Editor and manually change the min and max poll values to both be 16 seconds. After doing some local testing with both min and max poll set to 16 seconds, PresenTense produced most of its times within the 100 microsecond target range. PresenTense was then installed in the test lab on one of the Windows machines and configured with all of the previously mentioned configuration settings along with using the Registry Editor

to change the min and max poll values. The w32tm command with the stripchart argument was again used but this time the output was sent to a file so the results could be graphed.

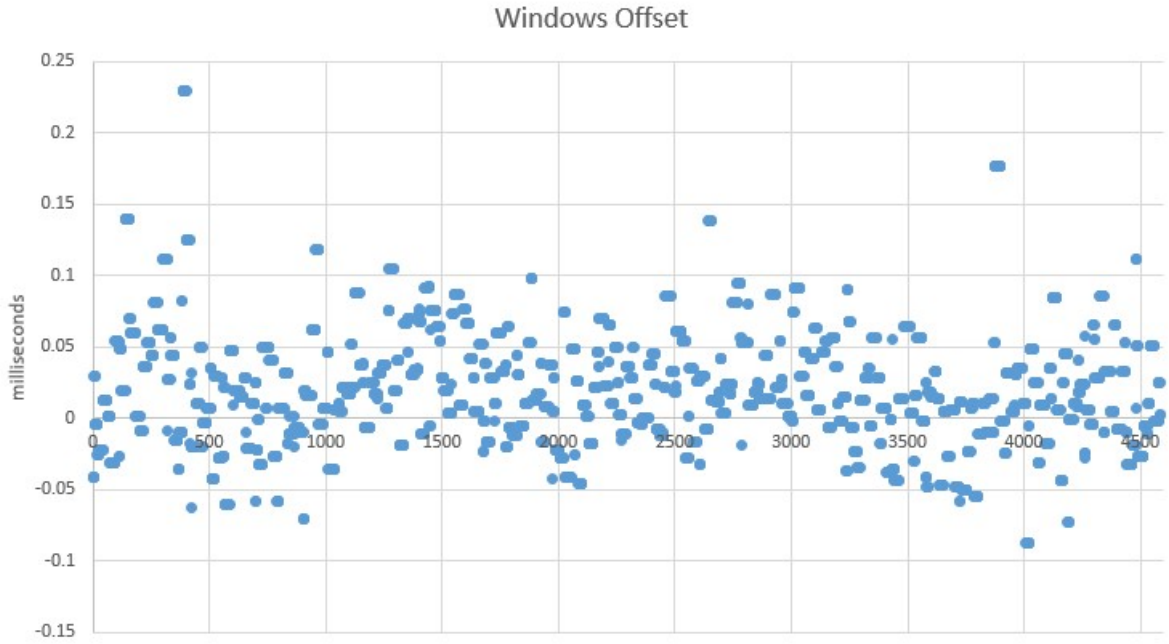

Figure 2.1

The graph in Figure 2.1 shows that 98% of the time we were within our 100 microsecond target range (millisecond values 0.1 and -0.1) which initially seemed like a great result. The next step to determine if this was a good result or not was to compare it to one of the Linux machines. Linux does not have the w32tm command so the Linux equivalent ntpdate command with argument –q was used and additionally configured to output every second to completely mimic the w32tm command and then sent the output to a file so the results could again be graphed.

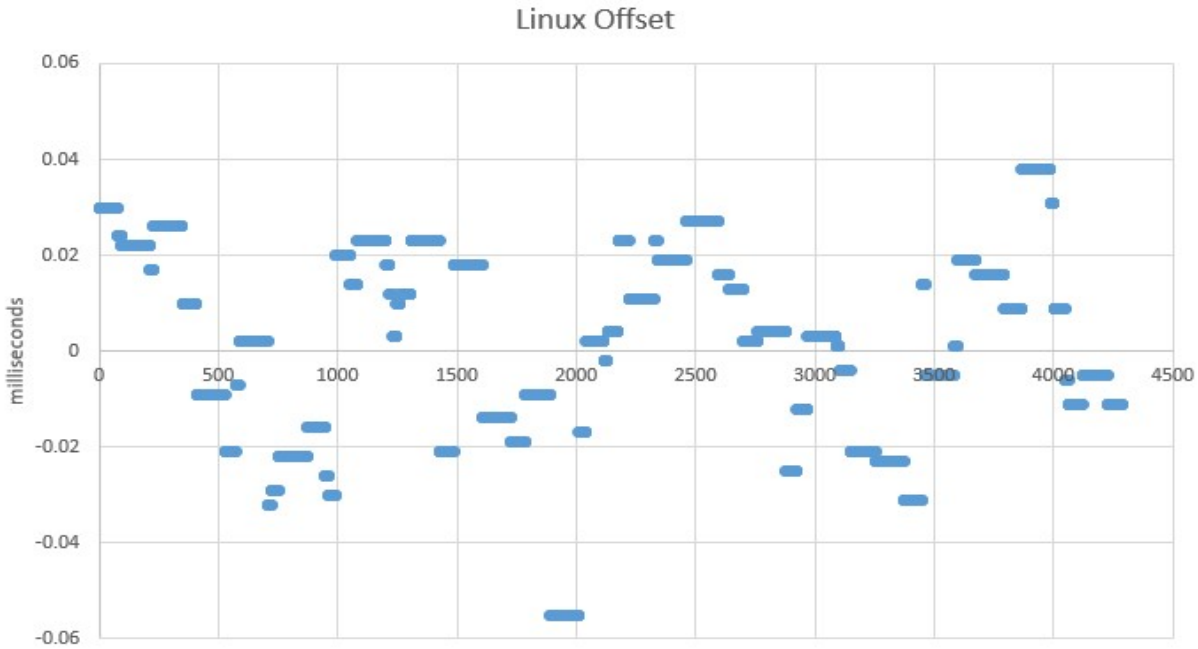

Figure 2.2

The graph in Figure 2.2 shows that 100% of the time the Linux machine was within the 100 microsecond target range with values significantly more accurate than the values seen on the Windows machine. Some investigation was needed to determine the cause of the large gap between the two data sets. What has not been mentioned before is that w32tm for Windows and ntpdate for Linux produce two additional outputs besides the time difference between itself and a time server, they output ntp jitter and network latency. For reference, the ntp jitter is the variance in latency on the network and the network latency is the round-trip time it takes the packet to get to its destination and return to sender. The w32tm and ntpdate commands were run with their respective arguments on their respective machines and this time the network latency and jitter values were output to a file to be graphed.

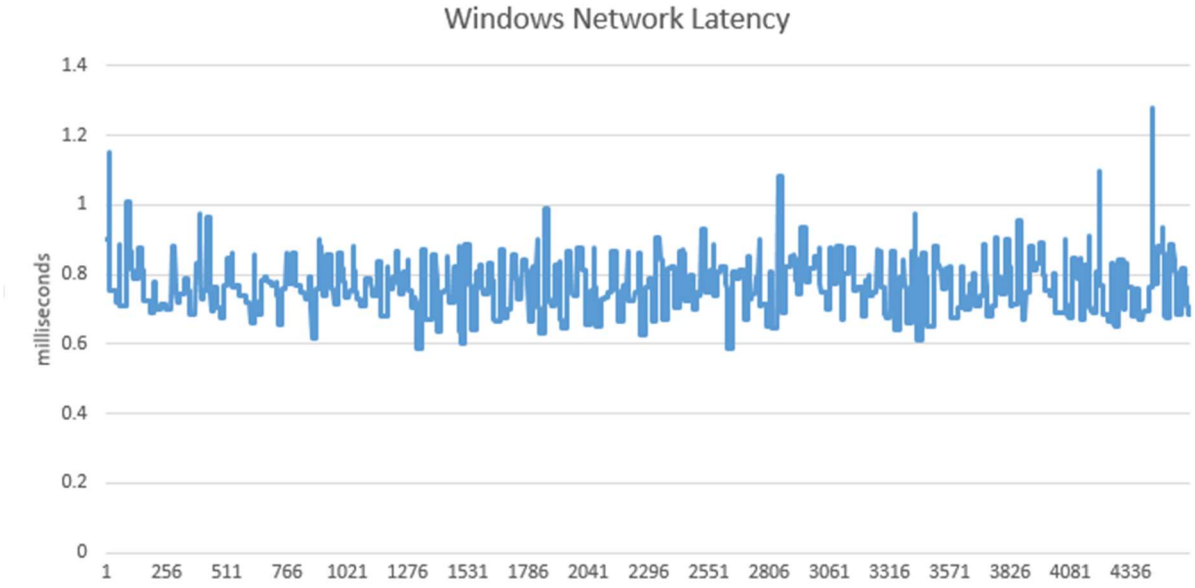

Figure 2.3

The graph in Figure 2.3 shows the network latency of the Windows machine fluctuates between 0.58 and 1.3 milliseconds with the biggest spike being 0.515 milliseconds (515 microseconds). Not pictured are the results of the Windows jitter which varied between 0.05 and 0.8 milliseconds. When visually comparing Figure 2.3 to Figure 2.1 the data seemed to support the hypothesis that whenever we had a spike in the network latency we jumped out of the 100 microsecond target range.

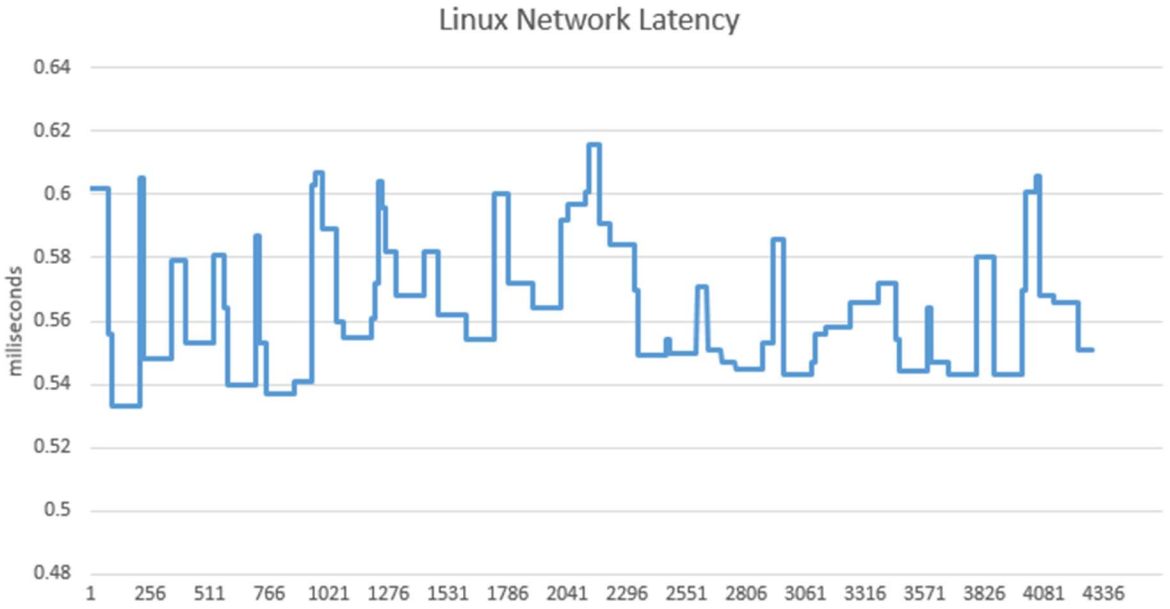

Figure 2.4

The graph in Figure 2.4 shows the network latency of the Linux machine fluctuates between 0.53 and 0.62 milliseconds with the biggest spike being 0.062 milliseconds (62 microseconds). Not pictured are the results of the Linux jitter which varied between 0.005 and 0.105 milliseconds. Further research is needed to determine why the latency spikes are occurring and why the Windows latency is much higher than the Linux latency since both are roughly the same distance from the GPS time server (give or take a foot or two) and have the same number of hops to the server.

During our testing in the lab we discovered that PresenTense takes roughly two hours to get consistently inside the 100 microsecond target range. This discovery means that PresenTense does not meet our second requirement stating that a machine needs to be synced to within 100 microseconds within 20 minutes after startup. After talking to the test team they believe that this should not be a problem. They mentioned that after the switch to Windows only they would have 12 machines capable of participating in a refueling test event, meaning if one machine went down they could simply switch to another machine. With the reboot sync and latency issue still unsolved, we did however decide that PresenTense getting us within 100 microseconds of our GPS time source 98% percent of the time would be good enough to mark this requirement as complete and move on to the next phase of testing.

3. Sending Accurate Time Information

Now that we have shown that Windows can accurately sync to a GPS time source, we need to prove that EZ-Fly can send out its time information accurately enough to stay within 100 microsecond threshold. To isolate the functionality that sends out the time information, a new project was created with a modified version of EZ-Fly. This modified version of EZ-Fly simply contained a thread with a callback function that by default is called at 60Hz (60 times per second). The callback function is called with two values, a callback time which is a time since the epoch and frame time which increments each frame by the unchanging value of 0.01666667 (1/60). An additional change to this callback function included some logic to calculate and then store how long it took between calls to the callback function. Then upon shutdown of the modified version of EZ-Fly the stored values are output to a file so they can be graphed. Since the callback function is expected to be called at 60Hz the time difference between calls should be very close to 0.01666667 (1/60). To normalize the data on the graph the value 0.01666667 was subtracted from each of the time differences meaning the values on the graph should be within 100 microseconds of zero. The modified EZ-Fly application was then built for Windows and brought back to the air refueling test lab and installed on one of the Windows machines we will call Test PC3. Test PC3 was also loaded with PresenTense and synced to the GPS time source with 98% accuracy as described in the previous section. The modified EZ-Fly application was started and ran for about 30 seconds producing the following graph.

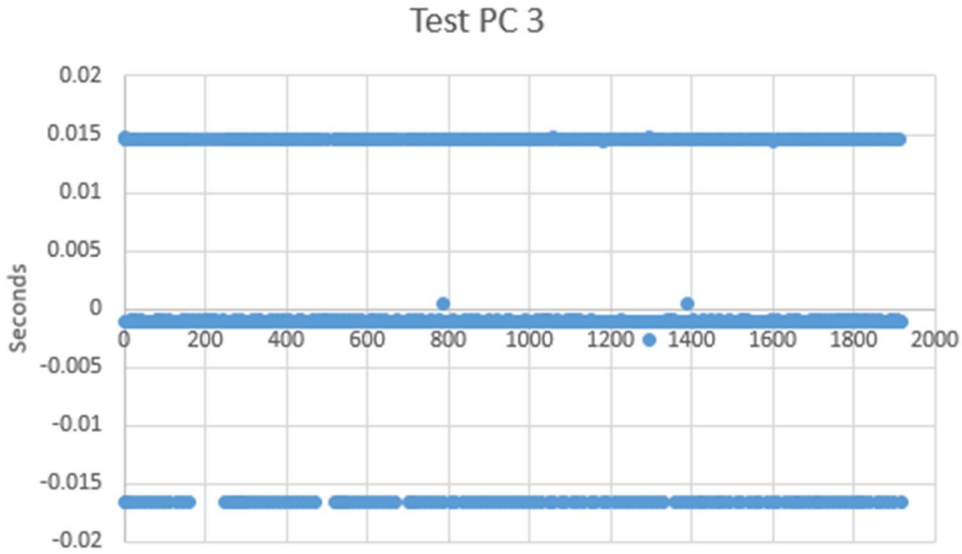

Figure 3.1

The ideal result would be all points within 100 microseconds of zero meaning the thread function was called at 60Hz or 60 times a second. The graph in Figure 3.1 was not expected at all and was a bit tricky to analyze. The data showed that 80% of the points were within 1 millisecond of 0 with 13.5% at positive 15 milliseconds and 6.5% at negative 16.667

milliseconds. To get these three distinct lines knowing that we are subtracting 0.0166667 from the time difference between calls means that the actual time between calls was either 0.031, 0.015, or 0 seconds. One thing to note about the thread is that it has some intelligence in that it changes its sleep time to try and accurately call the thread function at the expected time. These results show that we were missing frames and then trying to make up for it by not sleeping during some frames. After adjusting some visual effect settings and starting the modified EZ-Fly application as an administrator with high priority the test was ran again. The changes had no effect on the output and what was interesting is that when setting the thread priority to low, the modified EZ-Fly application produced the same result as when the thread was in normal and high priority. A few setting tweaks were made with no change in the graphed results, so the next step was to determine if this is how the graph would look on Linux. Since the modified version of EZ-Fly is cross platform, it was built for Linux and loaded onto one of the Linux machines in the lab. The Linux machine uses its native ntp for time synchronization and is synced to the GPS time source with 100% accuracy as shown in the previous section. The modified EZ-Fly application was then started and run for about 30 seconds producing the following graph.

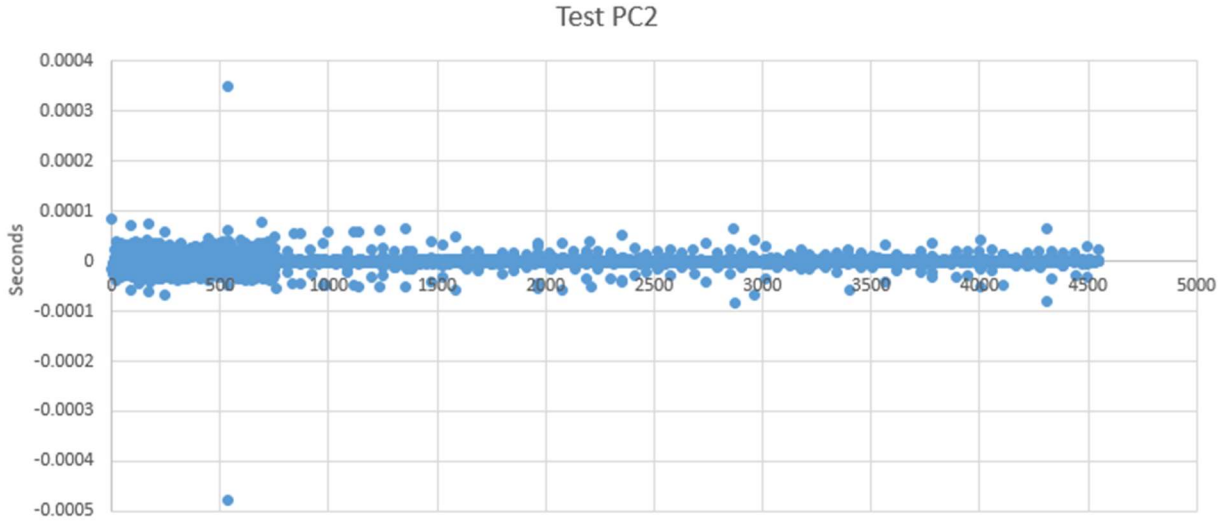

Figure 3.2

The graph in Figure 3.2 is about as ideal as you can get with all but two points 100 microseconds away from zero meaning that the thread function was called at 60Hz or 60 times a second. Now the concern was how to get Windows to produce a result similar to this since the closest Windows got was 10 times worse at 1 millisecond away from zero.

The initial thought was that maybe the time functions for the Windows version of EZ-Fly needed to be updated as we were using the older timespec and time_t structs along with c++99. Visual Studio 2019 was used to create a new project and it replicated the thread and callback function used before. After running the new project and verifying it behaved exactly like the original EZ-Fly code, the old time functionality was upgraded to the chrono library which contains things like steady clock, time points, and durations. After running tests on the Windows machine in the lab, it turned out that updating to the chrono library did not change the output at all. Another possible solution to try is the Windows multimedia library that when paired with a Query Performance Counter can be used to accurately keep track of time intervals at a sub microsecond resolution. The Visual Studio 2019 project mentioned previously was updated to use the windows multimedia library. Changes were also made to how the thread slept to try and increase the accuracy of the callback times. Essentially, the thread calculates how many milliseconds it should sleep by using the Query Performance Counter to determine the number of counts until the next frame converted into milliseconds. The thread then sleeps for 90% of the millisecond value, then wakes up and sits in a while loop for the remaining 10%. The while loop again uses the counts from Query Performance Counter with sub microsecond accuracy to determine when to break out of the loop. Once out of the loop the thread calls the callback function and repeats the process until execution is terminated. The callback function that is called keeps track of the previous system time that the method was called and logs the time difference between the current system time and the previous system time so it can be graphed. This new application was then built and brought back to the lab to test on Test PC 3 with the following result.

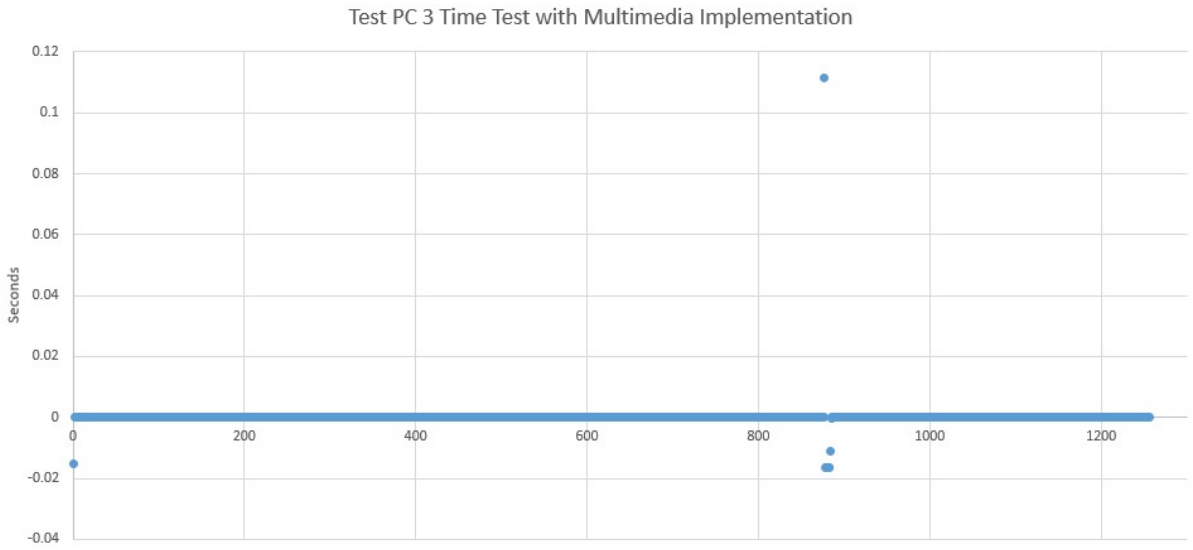

Figure 3.3

Initially the graph in Figure 3.3 looks pretty good even though we cannot clearly see what the values around zero are. There are a few rogue points that are believed to be from the system clock updating which it does every 16 seconds. Further testing will need to be done to determine if the spike will cause problems. To see what the values around zero where the graph was zoomed in around the zero on the y-axis and produced the following graph.

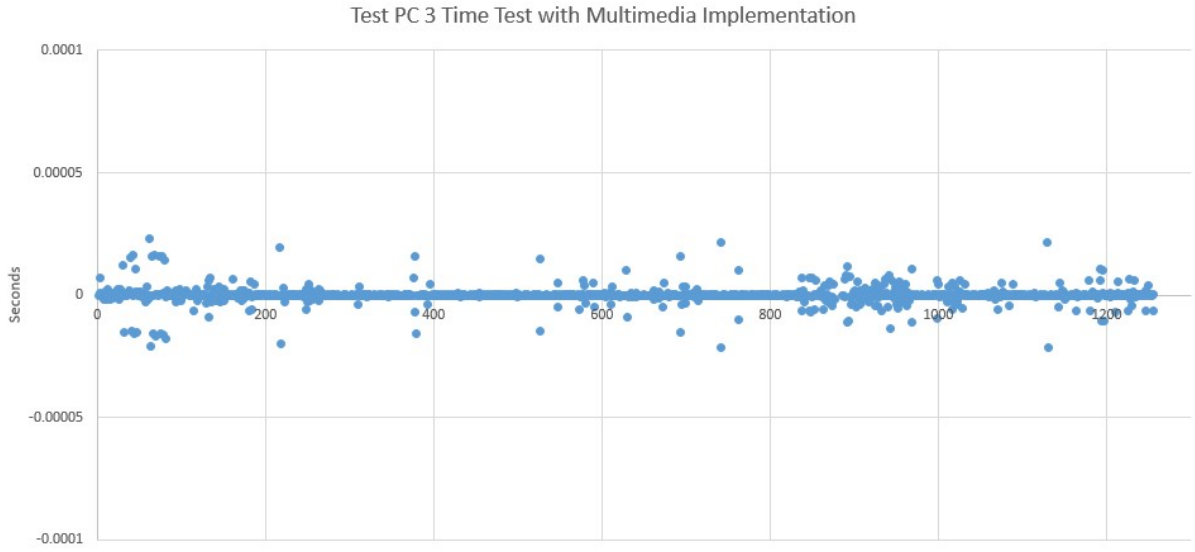

Figure 3.4

Notice that in Figure 3.4 the y-axis has a max of 0.0001 (100 microseconds) and a min -0.0001 (-100 microseconds). All pictured points are well within the 100 microsecond range and are actually under 20 microsecond away from zero, proving that the callback function was called every 1/60 of a second with +/-20 microsecond accuracy. Interestingly, these results are actually better than the result we observed on the Linux Test machine. After showing these results to the proper team leads and management it was determined that we had enough evidence to make the switch from Linux to Windows.

4. Conclusion

In the previous sections the data has shown that Windows has the ability to successfully sync to within 100 microseconds of a GPS time source and can accurately call a callback function to allow for proper timestamps to be sent out. After getting the go ahead to proceed with converting EZ-Fly to Windows we established the following next steps. The first big step is converting EZ-Fly to Visual Studio 2019 and upgrading our c++99 version to c++17. After upgrading, we then have to get the project to build by updating all depreciated c++99 functions and verifying the application runs as expected by running our regression test suite. The next step is implementing the multimedia time functions and verifying we can reproduce the values in Figure 3.4. We then will have to purchase a PresenTense license and install on one of the Windows computers in the lab. After the install is complete and we get the Windows machine to sync to the GPS time source, we then can load the new version of EZ-Fly onto it. The final and most important step is getting the new Windows version of EZ-Fly to participate in a live air refueling test event to benchmark its performance and compare to the performance of the Linux version. If the Windows version of EZ-Fly is able to successfully complete a live air refueling test event we then can begin the site transition from Linux to Windows. We would be able to cancel our renewal of the Linux license fees and then reimage the 9 Linux test machines in the lab to all be Windows operating systems. This would also mean that during future test events we would have 12 Windows machines to switch between if one goes down or is having issues. While we still need to solve the Windows network jitter issue and PresenTense taking longer than 20 minutes to properly sync to the GPS time source, we have a few tests and fixes in mind that could mitigate the negative effects these issues may cause. We also don't know for certain that the aforementioned issues will even cause problems. We will document our test event with visual observations along with capturing the PDU packets using Wireshark to determine the success of the test. The steps and process provided in this document will hopefully provide ideas to parties interested in using a Windows environment to participate in a simulated test event with strict time synchronization requirements.

5. References

[1] Win32 Apps: Windows Time Service Tools and Settings, https://docs.microsoft.com/en-us/windows-server/networking/windows-time-service/windows-time-service-tools-and-settings

[2] Microsoft: W32tm, https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2012-r2-and-2012/ff799054(v%3Dws.11)

[3] Meinberg Funkuhren GmbH & Co KG.: Meinberg, https://www.meinbergglobal.com/english/sw

[4] Bytefusion: PresenTense Time Client, http://www.bytefusion.com/products/ntm/ptnt/whatispresentense.htm

[5] Linux Man Page: Ntpdate(8), https://linux.die.net/man/8/ntpdate

[6] Cppreference.com: Date and Time Utilities, https://en.cppreference.com/w/cpp/chrono

[7] Win32 Apps: Windows Multimedia, https://docs.microsoft.com/en-us/windows/win32/multimedia/windows-multimedia-start-page

[8] Win32 Apps: Queryperformancecounter Function, https://docs.microsoft.com/en-us/windows/win32/api/profileapi/nf-profileapi-queryperformancecounter

Author Biography

KYLE REBELLO is a Software Engineer for the Northrop Grumman Corporation – MAFDMO and the PACAF Live Virtual Constructive (LVC) programs. He has a B.S. in Computer Science from the University of Central Florida and is currently pursuing a M.S. in Computer Science with a focus on Artificial Intelligence from the University of Central Florida. Kyle has over 6 years of experience developing mission-critical software for the Air Force.