OUTLIER IDENTIFICATION IN SENSOR NETWORK CLOCK SYNCHRONIZATION

A Paper
Submitted to the Graduate Faculty
of the
North Dakota State University
of Agriculture and Applied Sciences

By

William Davis Voorhees

In Partial Fulfillment of the Requirements
for the Degree of
MASTER OF SCIENCE

Major Department:
Computer Science

May 2010

Fargo, North Dakota

North Dakota State University
Graduate School

Title

OUTLIER IDENTIFICATION IN SENSOR NETWORK

CLOCK SYNCHRONIZATION

By

WILLIAM DAVIS VOORHEES

The Supervisory Committee certifies that this ***disquisition*** complies with North Dakota State University's regulations and meets the accepted standards for the degree of

MASTER OF SCIENCE

SUPERVISORY COMMITTEE:

ABSTRACT

Voorhees, William Davis, M.S., Department of Computer Science, College of Science & Mathematics, North Dakota State University, May 2010. Outlier Identification in Sensor Network Clock Synchronization. Major Professor: Dr. Kendall Nygard.

We first present a survey of clock synchronization research that describes the problem and various challenges to implementing an efficient protocol for clock synchronization in sensor networks. We then present several changes to an existing protocol that we believe will enhance the accuracy. These changes are implemented in a software simulation and a large experiment is conducted that involves several runs of the simulation using various parameters for error bounds and outlier detection. We present our results and show that modifying the outlier detection range used by many existing algorithms can improve network clock accuracy and provide a stabilized average error.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

Sensor networks have been a hot topic in computer science research for the last several years. The concept behind a sensor network is to use many low power, radio connected nodes to form a large network capable of sensing various environmental variables across a large area. Each sensor node is battery powered and has limited processing power and memory. This makes developing control schemes for sensor networks very challenging. Approaches used in traditional computer networks do not translate well to this resource constrained platform.

Each sensor node is designed to be very cheap to manufacture. This allows several hundred or even thousands of nodes to be deployed into an area for relatively minimal cost. Each node is equipped with a low power radio that allows the nodes to share information. Generally, the information collected by a network gets relayed to a base station, or sink, that is capable of sending the information via high power radio or satellite to a staffed command station.

There are several outstanding issues in regards to sensor networks. Network topology and routing are fundamental problems that have yet to be solved in a general sense. Several mission specific solutions exist, but no over arching scheme has yet been developed. In the case of military and some civilian applications, security becomes extremely important. Due to the resource constraints of the average sensor node, standard encryption schemes cannot be used. This is an area of active research.

Finally, the last major issue faced by sensor network engineers is clock synchronization, the topic of this paper. Protocols in use on standard computer networks to keep local clocks synchronized (such as NTP (Mills, 1994, [1])) do not translate well to sensor networks with severely limited processing and storage. There are also several extra sources of synchronization error in a sensor network that do not exist in a standard computer network. Most of this error comes from the radio links used to create the network. In addition, most computer networks can sustain substantial error with no ill effects to higher level applications. In the case of sensor networks, large errors can render the network unusable. The extra precision demanded by a sensor network compared to a traditional computer network makes the task of clock synchronization substantially more difficult.

In the following section, we will discuss the current state of sensor network clock synchronization research. Several papers have been presented in the last few years that represent promising approaches to solving the problem of clock synchronization. In Section 3, we will discuss one of these papers in detail and propose several modifications to the scheme presented. These modifications will be designed to lower the total clock synchronization error between two nodes in the network. Section 4 will present a simulation designed to test our changes compared to the original protocol. Data from several simulation runs will be presented and the effectiveness of our protocol changes analyzed. We conclude the paper in Section 5.

2. PROBLEM REVIEW

Clock synchronization in sensor networks is one of the most important support services to the nodes. Accurate time synchronization has a direct impact on many applications: military target tracking relies on accurately determining the difference between when two nodes sense the same target, counter-sniper systems use sound triangulation to pinpoint the source of a noise (Girod, 2001, [2]), MAC layer protocols such as TDMA rely on nodes being synchronized to a global network time (Claesso, 2001, [3]), removing duplicate events sensed by multiple sensors necessitates proper ordering of events (Intanagonwiwat, 2000, [4]), and power management schemes that rely on being able to accurately schedule radio time also require accurate global network time (Gu, 2005, [5]). It is obvious that a robust, scalable network time synchronization protocol is essential to efficient operation of nearly any sensor network.

Several sources of error exist in a sensor network that are detrimental to clocks synchronization. These sources of error come mostly from the radio and MAC scheme, but several exist on the node itself. These error sources were first described by Kopetz and Ochsenreiter (Kopetz, 1987, [6]) and later extended in (Maróti, 2004, [7]).

1. *Send Time* - the time required to build the message to be sent and forward it to the transmitter. Also included is the time required for the transmitter to make the MAC layer access request. This delay can be hundreds of milliseconds.

2. *Access Time* - the delay while waiting for access to the radio channel used for transmission. This delay is heavily influenced by the network traffic and can be as high as several seconds. The access time is the most variable time and presents the single largest source of error in clock synchronization.

3. *Transmission Time* - the time required to transmit the entire message. This can be tens of milliseconds, but can be easily determined using the length of the message and the speed of the radio.

4. *Propagation Time* - the time it takes for the message to go from the sender to the receiver over the air. This time is determined solely by the distance between the nodes, and in cases where the nodes are 300m apart or less, is under 1μs. This value comes from the simple fact that radio waves propagate at the speed of light (300,000,000 m/s, or 300m/μs).

5. *Reception Time* - the time required for the receiver to get the message from the air. This time is identical to the transmission time and is determined by the message length and radio speed.

6. *Receive Time* - time needed to process a received message and notify higher level applications. This time is very similar to the send time.

7. *Interrupt Handling Time* - the time required for the radio chip to signal the microcontroller that there is a message incoming. This delay is normally on the order of at most a few microseconds, but in situations where interrupts are disabled on the node, the delay can become very large, potentially into the many seconds range.

8. *Encoding Time* - this is the delay incurred by the radio chip translating the message into a radio signal to be transmitted. This time runs on the order of hundreds of microseconds, but it does not vary widely from transmission to transmission and can be easily calculated.

9. *Decoding Time* - this time is similar to the encoding time, but deals with the reception of the message and translating it from a radio signal to bits on the radio controller. The time is mostly stable, can be calculated, and runs on the order of hundreds of microseconds. However, signal strength variations can lead to small amounts of jitter (change in the decoding time from transmission to transmission).

2.1. Existing Solutions

There have been many proposed algorithms over the years to solve the problem of network clock synchronization, such as (Maróti, 2004, [7]), (PalChaudhuri, 2004, [8]), (Sommer, 2008, [9]), and (Yoon, 2007, [10]). Two schemes in particular stand out in the research. These are Reference Broadcast Synchronization (RBS) (Elson, 2002, [11]) and

Timing-sync Protocol for Sensor Networks (TPSN) (Ganeriwal, 2003, [12]). These two methodologies are most often presented as solutions for the clock sync problem. In all schemes, the goal is to provide network clock synchronization with minimum error. Some systems require errors on the order of just a few microseconds, so this is the goal of most researchers.

RBS relies on the fact that the radio medium is inherently broadcast. Using a source node and a broadcast message, RBS aims to synchronize two receivers with one another. The source node sends a broadcast message that both of the receiving nodes hear. Each node exchanges the time it received the broadcast with the other node. The relative offsets of the nodes are calculated via linear regression. This scheme allows for synchronization with an error of 11μs on Berkeley Motes. This is an error factor of approximately 5.5 given the Berkley Mote clock resolution of 2μs.

Because RBS does not rely on time stamping at the sender, the errors from send time, access time, transmission time, and encoding time are all eliminated. Since access time is the single largest source of error in the network, eliminating it makes the RBS scheme quite accurate. RBS also has the advantage that it does not rely on low level access to the hardware like some other protocols. This makes it ideal for platforms that do not provide such access.

Several derivatives of RBS have been developed that attempt to reduce the total error including (PalChaudhuri, 2004, [8]) and (Ganeriwal, 2003, [12]). TPSN is the most famous and widely studied of these derivative works. TPSN uses the basic scheme of RBS, but adds a methodology for organizing the network nodes and synchronizing them in a

more efficient fashion. TPSN also uses 2-way message exchange to eliminate the unknown propagation error in RBS. TPSN starts by forming a spanning tree of the network. Nodes then synchronize to their parent. Using Mica motes for simulation, TPSN achieved 16.9μs error. With a clock resolution of .25μs, this is an error factor of 67.6. TPSN relies on MAC layer access to the radio chip. This is a feature of Mica motes that will be discussed in Section 3.

2.2. Other Considerations

All of the research mentioned has assumed a homogeneous sensor network. That is to say, a network where all nodes are of the same design and have the same processing, storage, and power resources. Heterogeneous networks have motes of varying capabilities. Most commonly, there are many lower powered motes and several higher power motes with increased processing, storage, and energy resources. In the case of TPSN, the spanning tree protocol used would migrate to a heterogeneous network quite easily by simply stating that all root nodes in the TPSN protocol are replaced by the high power nodes.

There are few clock synchronization algorithms tailored to heterogeneous networks. This is mostly due to the fact that the benefits of a heterogeneous network come in the form of increased power and computation of some nodes. Clock synchronization does not rely heavily on computation and increased power does not become a significant resource in many cases. This is due to the fact that radio

transmissions are symmetric: they take the same amount of power to send as to receiver, so the only time extra power becomes important is when one node transmits different messages to many. Broadcast synchronization schemes do not garner any benefit from the heterogeneous design.

Heterogeneous networks have a distinct advantage in that there can be many clocks in the network that are always accurate. Each of the high power nodes can conceivably have a GPS unit on board. With the increased battery life and overall higher cost of the node, the addition of a GPS unit is not unreasonable. GPS uses satellite signals to determine the location of the GPS unit. Many satellites are constantly beaming their current time to Earth. A GPS unit uses the small differences in times from different satellites to determine its location. The accuracy of the time reported by a GPS satellite signal is significantly less than 1μs and is often on the order of 2-3 nanoseconds. This gives a GPS enabled node an extremely reliable source of time.

The nine sources of error described above assume that the messages being broadcast are readily accessible. Some sensor network designers are very concerned with security (such as networks with military applications). One potential way to safe guard transmissions is to encrypt all data sent and received. Most high level encryption schemes in use today are not practical on the very constrained node platform. RSA encryption/decryption would slow the network to the point where it is unusable. Some schemes have been proposed that use lightweight encryption, reducing the time delay when sending or receiving messages (Traynor, 2006, [13]).

It might be of concern to researchers attempting to achieve high accuracy clock synchronization that network security researchers want to introduce another level of delay. If the timestamp packet needs to be encrypted before being sent, then there will be a new source of error that depending on the encryption scheme, may be quite large. We discuss this problem in section 3.2.

3. BASE SCHEME

This section begins with essential background information for our scheme. We then address other issues that were mentioned in section 2. We describe the basic scheme we are using as a base and explain the modifications that we believe will improve the performance of the protocol.

3.1. Scheme Background

The TPSN scheme relies on having direct MAC level access to time stamp synchronization packets just as they go to the air or are received from the air. With this level of access, nearly all of the delay in the network is eliminated. Send time and access time are not an issue since the synchronization packet can be time stamped *after* MAC access has been confirmed and the message has begun transmission. The transmission time is deterministic and therefore can be calculated and removed as a source of error with minimal effort. The propagation time is still a source of error, but since most nodes only have a radio range of about 150m (Crossbow, 2009, [14]), the maximum error less than 1μs.

The reception time is identical to the transmission time and is also deterministic. With low level MAC access, the received synchronization packet can be time stamped as soon as it begins to be received, so the receive time and interrupt handling time are on the order of 1 clock tick, which in the case of the Mica2 mote is .25μs. With this direct access to the MAC layer, the only sources of error in a synchronization exchange are the

propagation time, encoding time, and decoding time. The only one of these that exhibits any significant jitter is the decoding time and is therefore the largest source of error.

Not all sensor nodes allow this low level access. The Mica and Mica2 motes are pioneers of this technology. The node design that allows this kind of access was described by Hill and Culler in (Hill, 2001, [15]). The architecture presented described a shift away from specialized processing boards on the node. Instead of having an onboard processor for a specific communication scheme, the higher application layer handles the communication protocol. This is possible only when the application layer has direct access to the communication hardware.

The low level access is made possible by hardware accelerators. The naive solution to providing the higher application layer with direct hardware access is to simply tie the two components directly together on the node. This, however, results in a massive decrease in node performance. To get around this limitation, hardware accelerators are added alongside the link between application processing and the radio. See Figure 1 taken from (Hill, 2001, [15]) Figure 4, page 7. The application layer still has direct access to the hardware, but communication protocols can run through the hardware accelerators giving the protocol a significant boost in performance that nearly equals the performance of a traditional node with specialized hardware (Hill, 2001, [15]).

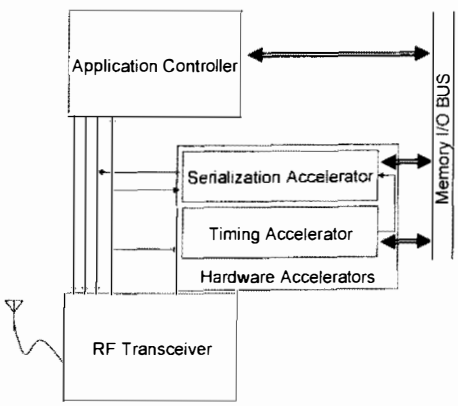

Figure 1: Hardware Accelerators Architecture

Using this architecture, it is possible to timestamp outgoing packets once they begin transmitting and also to sense an incoming packet and make a note of its arrival within the .25μs clock granularity. Section 5.2 of (Hill, 2001, [15]) discusses a time synchronization protocol using this architecture that synchronizes two nodes to within 2μs of each other. They list the sources of error as raw radio transmission (+/- 1μs), arriving signal capture delay equal to the granularity of the node clock (+/- .25μs), and the time required for the node to process the packet and update its clock offset (+/- .625μs).

3.2. Other Considerations

We briefly mentioned the effect encryption might have on clock synchronization in section 2. Now that we have seen how low level access to the radio hardware can be used for high accuracy synchronization, the problem of encryption becomes even more important. If we want to timestamp a packet as it is being sent, we need to be able to generate the stamp and add it into the packet very quickly. All the encryption schemes presented for sensor networks thus far would take far too long to generate an encrypted stamp (Gura, 2004, [16]).

We must consider when it becomes necessary to encrypt synchronization packets. Security of this level only becomes important when there is an adversary attacking the network. It is easy to imagine a malicious node sending bad time packets in an attempt to desynchronize the clocks in the network. In section 3.4, we will discuss rejection of outliers - error data points that are much larger than any other collected. An attacker

attempting to skew the global network clock would need to introduce large errors for any significant desynchronization. Throwing out larger than normal data points would not allow this bad data to effect the overall network synchronization. If we can prevent attacks against clock synchronization without encryption, then we can safely send timestamps in cleartext.

This type of interaction between security and clock synchronization will no doubt become a topic of intense research over the coming years. While large errors will not be allowed to change the global network time, it may be possible to introduce small errors that shift the synchronization by small amounts over a long period of time, resulting in desynchronization of the clocks. More research is needed to accurately determine possible threats to network synchronization.

3.3. Base Scheme Description

A direct derivative work of (Hill, 2001, [15]) was developed by Kusy and Culler (Kusy, 2006, [17]). This paper used the lower error of a hardware accelerated node as a basis for a full time synchronization protocol based on a primitive called Elapsed Time on Arrival (ETA). ETA is a time stamping primitive that is used in their primary protocols: Routed Integrated Time Synchronization (RITS) and Rapid Time Synchronization (RATS). The ETA protocol is designed for node to node synchronization. RITS extends ETA to multihop networks by having all nodes synchronize to a root node. RATS reverses RITS and has the root node initiate synchronization to all nodes in the network.

In a heterogeneous network, the root node used in both RITS and RATS is simply replaced with the higher power nodes of the network. This results in multiple root nodes each with a group of child nodes that is a subset of the entire network. The advantage to this approach is that the total number of hops from root to the edge of the network (or in this case, the edge of the group with the high powered node as the root) is reduced, and therefore so is the total clock error.

RATS was implemented on the Mica2 mote using the TinyOS node operating system. It was tested on 60 nodes arranged in a 5x12 grid. The root node transmitted synchronization pulses every 2 seconds for the first 10 seconds then every 30 seconds for the remainder of the test. The global time was queried from each node every 5 seconds for the first 2 minutes and then every 23 seconds thereafter. For each sample, the absolute and average absolute error was calculated as the difference between the queried node time and the root time. The experiment was run for 6 hours. The maximum error was 26μs and the average error was 2.7μs. A similar protocol, FTSP (Maróti, 2004, [7]), was able to achieve slightly higher accuracy (2.3μs average error), but took 150 times longer to converge (all nodes synchronized).

The ETA primitive and RATS implementation provide very high accuracy time synchronization without large computational expenses or power demands. Using the high accuracy of the clocks to create tightly bounded transmission windows in schemes such as TDMA may indeed more than pay for the periodic updates required by the protocol. There are a few aspects of the scheme that should be investigated, and that is the purpose of this paper.

3.4. Changes to RATS

RATS relies on the hardware accelerator architecture for its accuracy. The clock synchronization in (Hill, 2001, [15]) had three sources of error: raw radio transmission (+/- 1μs), arriving signal capture delay (+/- .25μs), and packet processing (+/- .625μs). This resulted in an accuracy of +/- 2μs. We would argue that the total error in this scheme will always skew to the positive side and the error is therefore +2μs at maximum. This gives a total error range of 2μs instead of 4μs. We will use the simple example below to illustrate the argument.

At time 100μs, the root node creates a synchronization packet with the numeric payload 100 to represent the time the packet was created. Best case scenario, the packet leaves the node immediately, resulting in 0 error. Using the established error bounds from (Hill, 2001, [15]), the radio transmission may be delayed 1μs. We can see from Figure 2 that so far we have error from 0μs to 1μs. Propagation delay is assumed to be negligible. The packet is received by the destination node. If the packet is timed right, it comes in just as the high accuracy clock ticks, so the message is time stamped perfectly, resulting in 0 error. If the packet just misses the clock tick, it is stamped at the next tick .25μs later. Or total error now ranges from 0μs to 1.25μs. Finally, there is some delay

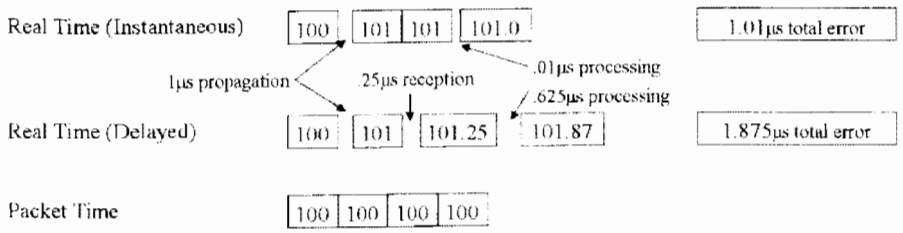

Figure 2: Transmission Sources of Error

incurred by the processing of the packet. This delay will almost assuredly be greater than 0, but no lower bound is given. The maximum delay might be .625μs. Our total error is therefore between 0μs (realistically slightly higher) and 1.875μs. This error only leans on the positive side. In Section 4 we present our simulation of the proposed changes to ETA and RATS. We run the simulation using both the +/-2μs error assumption and our +2μs assumption.

One other aspect of ETA and RATS requires analysis. Many of the proposed time synchronization algorithms rely on linear regression to provide accurate synchronization between synchronization events. Using linear regression in this way was first described in (Elson, 2002, [11]). Linear regression is necessary because clocks will drift apart from each other over time. This is a consequence of using inexpensive crystals as a frequency source. Each clock crystal will vibrate by very slightly different amounts. Crystals will also change their frequencies over time due to natural aging. Even temperature and vibration can cause changes in the crystal vibration.

Since these crystals are responsible for keeping a clock ticking uniformly, the clocks in the network will drift because of imprecise ticks between one another. It would technically be possible to keep all clocks in sync by simply broadcasting synchronization packets continuously. This does not make much sense from a practical standpoint because all the network would be capable of is keeping an accurate global time. By using linear regression, we can extrapolate the drift between two clocks and compensate for it once we have a few data points. In papers such as (Maróti, 2004, [7]), the data points for linear regression are tuples of the root node time and the local node time.

While not explicitly stated, the charts from (Elson, 2002, [11]) indicate that the standard size for the regression data table is 25 points. These 25 readings are used to calculate the linear regression variables *a* and *b* (slope and y-intercept, respectively) that are used for extrapolation. When a new data point is added, the oldest data point is removed. This gives a rolling window of data points that keeps the linear regression calculation current with the state of the clocks. This is important because as the crystals in the clocks age, the drift will change as well. If we were to graph the effects of this type of drift, it would not be a straight line. The drift might be 1μs per second at the start of the network, but natural crystal aging may bring the drift to much higher values (such as 100μs per second) at time goes on.

We could keep all data points ever collected, but this would essentially force us to fit a straight line to a non-linear pattern of clock drift change. By using only the most recent data points, we can approximate the drift curve with ever changing lines. Figure 3 illustrates this by showing continuous recalculation of linear regression parameters. By using multiple linear regressions (red lines), we more closely approximate the curved drift of the node clock (black line). In addition, storing all data points collected would

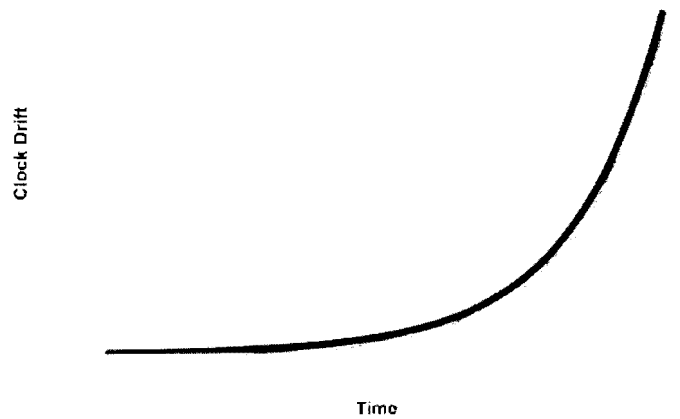

Figure 3: Curve Approximation with Multiple Linear Regression

eventually take up too much memory on the node, leaving no room for actual sensing data. This is obviously undesirable.

This linear regression method also requires some methodology for removing outliers. It is possible that some data points will be wildly inaccurate. This may be because of significant network congestion, natural forces interfering with communication, or even attacks against the network by adversaries. These outliers would only serve to skew the linear regression best fit line away from the idea regression. As such, any new data point determined to be an outlier is simply dropped.

The traditional methodology used by (Elson, 2002, [11]), (Maróti, 2004, [7]), and (Kusy, 2006, [17]) is to not include data points whose error is greater than 3 standard deviations from the mean error. The mean error is found by averaging the difference of each point's local node time component and the calculated local node time using the regression variables and the collected root time. As an example, assume that we have the data point (100,104) (local node time, root node time) and variables $a=1.025$ and $b=3.42$. Our calculated local node time would be 1.025(100) + 3.42 = 105.92 and this would give us an error of 105.92 - 104 = 1.92 for this data point. The error for each point is calculated and averaged. The standard deviation (σ) for the errors is also computed.

The new data point goes through the same process using the current regression variables. The error of this new data point is compared against the mean error. If the new data point's error is greater than 3 standard deviations from the mean (positive or minus), the data point is dropped. If the data point falls within this range, the oldest point is removed, the new point is added, and the regression variables are recalculated.

We are interested in what happens when the range of acceptable values changes. Using a more stringent outlier check, such as reducing the range from +/-3σ to +/-2σ may result in a more accurate line. As part of our simulation, we examine the effects of changing the outlier criteria with 2σ, 3σ (baseline), and 4σ.

4. SIMULATION

In this section we will describe our simulation software. We will discuss the goals of the experiment, how the program was developed, what assumptions we are using for our experiment, and the results after running the simulation. The simulation and experimental parameters are based on the ETA and RATS protocols described in (Kusy, 2006, [17]) and the error bounds from (Hill, 2001, [15]). We implement the ETA protocol in software and emulate a network that tests the 1-hop characteristics of RATS. RATS is designed and was tested in (Kusy, 2006, [17]) as a multi-hop protocol. We use a baseline simulation directly mimicking RATS as our performance baseline in the single hop case.

4.1. Goals and Assumptions

The purpose of this simulation is twofold. First, we wish to see the effects of our argument for using a +2μs error bound instead of the +/-2μs bound in (Hill, 2001, [15]). We contend that with this adjusted error bound, we should see errors that float toward the positive overall. This will of course be offset to some extent by node drift rates that may skew positive or negative. In the interest of completeness, we will run the simulation with both +/-2μs and +2μs for all other variables.

The second aspect of the simulation is to test changes to the outlier criteria used by the protocol. Outliers are traditionally defined as +/-3σ from the mean. We wish to see the effects of changing this range, so we will run tests with +/-2σ, +/-3σ (our baseline), and +/-4σ. We expect that a tighter bound on the data points will result in a better line,

but there may be unexpected side effects. Restricting the data points may result in such a narrow band that all subsequent data points are thrown out. If this is the case, our +/-4σ experiment should show slightly better performance.

Additionally, we will also test the differences between using an unbounded number of data points for the linear regression versus using a static number of data points. We will use the standard 25 data points when we limit the size of the data set. We fully expect the limited number of data points to provide better performance than all data points (see section 3.4).

With 2 different variables to test of the error bounds, 3 different outlier criteria, and 2 different bounds for the number of data points, we will have a total of 12 runs of the simulation. For all experiments we use a simulated 11 node network with one node being the root node. We will run each experiment for 2.5 hours using a synchronization period of 30 seconds and a sample period of 23 seconds. This will give us 391 samples for each of the 10 child nodes for a total of 3910 data points per run.

4.2. Simulation Design

The simulation software was written in the Python programming language and run on the Windows XP Profession 64bit Edition operating system using Python 3.1.1. The computer used has a 2.4Ghz Intel Core 2 Duo processor and 6GB of DDR2 RAM. The simulation program consists of three main parts that utilize object oriented principles. See Appendix A for source code.

The first part of the program is a node time class. This class creates an integer count timer that is constantly incrementing by one. The class takes in one argument that defines the drift of the node's clock. This is important to simulate because clock drift is a major factor when correcting for error. We do not take into account changing drift rates for our simulated clock due to the fact that our simulation times are not long enough for clock drift rate change to significantly affect our results. It would be a simple matter to add this functionality for further experimentation. The drift rate is used to periodically change the node time to reflect the variations in counting that occur in a crystal based clock.

The node time class keeps track of the current node time with three variables: *micro*, *second*, and *minute*. The *micro* variable stores the integer count and represents microseconds. Every 1,000,000 microseconds, we rollover the *micro* variable to zero and increment the *second* variable. Every 60 seconds we roll over the *second* variable and increment the *minute* variable. We split our time into these smaller chunks to avoid large integer calculations that would result from simply having a counter that starts at zero and keeps growing.

The second part of the simulation software is the node class. This class is used to create a simulated node that can communicate with the root node and handle linear regression calculations. Each node that is created takes in arguments for the node's ID, the drift rate, how many data points it can keep in memory, and what the error bounds are (refer to our previous discussion of +/-2μs versus +2μs error). When the node is created, it sets up several internal variables. The *regression* variable is a dictionary that

stores node Ids and synchronizations messages. This dictionary holds all the data that is periodically used to calculate linear regression information.

For those that are unfamiliar with Python, a dictionary is essentially an associative array, or an array indexed with arbitrary keys instead of numbers. In our case, we use the node ID of node being synched with as a key, then store all the information for that particular node ID in an array in the dictionary. This allows us to keep track of any inter-node communication with ease. For our simulation we focused on the node to root connection only, effectively limiting the simulation to 1 hop scenarios. It would be a relatively minor change to allow nodes to synchronize with any other node, creating a multi-hop simulation. We hope to explore this option in the future.

The node class has functions for several different jobs. The main job is to keep the counter going and periodically synchronize with the root node. The counter for each node runs in a separate thread to achieve as close to parallel execution of clock timing and synchronization as possible. The node class is also responsible for introducing the errors that are common in a real wireless network. It uses the errors for transmission time, receive time, and interrupt handling time defined in (Hill, 2001, [15]) as the basis for the errors that are introduced. The goal of the linear regression component is to eliminate these errors.

Finally we have the control program. Figure 4 graphically represents how the control program interacts with the node class. This program uses the classes we have discussed thus far to actually run the simulation. The control program acts as the root node for all communication. The same node time class that each node uses for its clock is used here for the root node's clock. The root node does not have any clock drift. We assume that it is a much higher quality clock than in the other nodes.

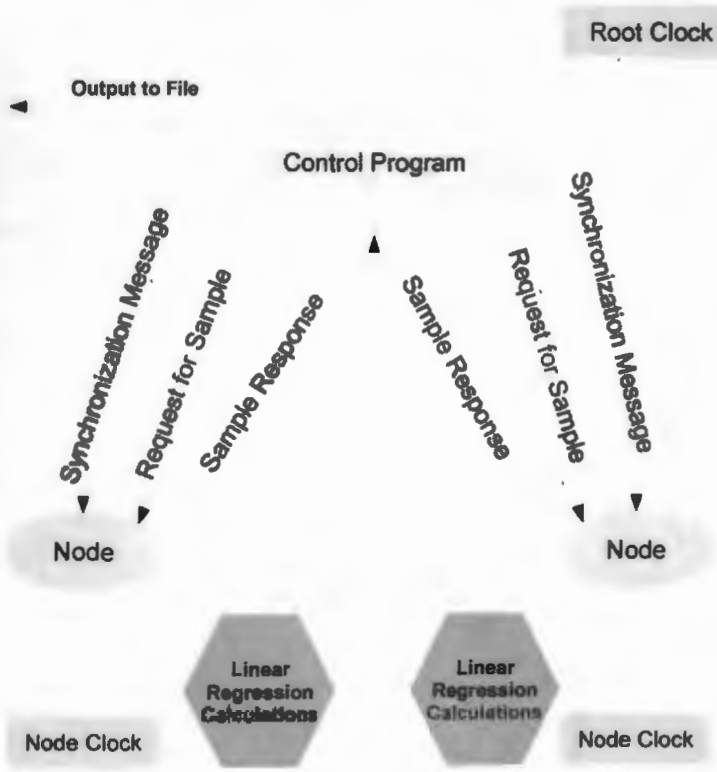

Figure 4: Simulation Program Architecture

This control program has variables for all the testable items we wanted to research. The *dev_points* variable stores the criteria for eliminating outliers. For our tests this had the values 2, 3, or 4. The *error* variable controls if we use +/-2µs or +2µs. This is a boolean where 1 means +2µs and 0 means +/-2µs. We also have the variables *run_time*, *sync_interval*, and *sample_interval*, for the total run time, synchronization period, and sample period, respectively. The synchronization period defines how often the nodes

synchronize with the root and the sample period defines how often we get the nodes' current time and the root time for offline analysis.

The main method starts by setting up the root clock and starting a timer for how long the program should run. After that is set up, the file operations for storing the sample data from the nodes are created. The program then builds as many nodes as are called for by the *number_nodes* variable. We default to 10 nodes in all tests, but this number could be increased easily. Each node is assigned a random drift value between 0 ppm and 100 ppm (Kusy, 2006, [17]). This metric determines how much the clock drift changes. In this case each node drifts by a random value between 0 and 100 μs per second.

Once all the preliminary data is ready, the root clock starts, node clocks start, and the control program goes into a loop where it periodically takes sample data from each node and synchronizes with the nodes. Figure 4 shows the synchronization message from control to node, the request for a sample from control to node, and the sample request response from node to control. When data is collected, the Node ID, Error, Node Time, and Root Time are written to a comma separated value file. The error is simply the root time minus the time linear regression would predict for the root. Each node has its own linear regression parameters. The node takes its current time, uses linear regression to determine what it thinks the root time should be, finds the error by subtracting this expected time from the actual reported root time (no error is introduced into this root time when it is sent to the node), and then reports this data back to the control program.

Once the *run_time* variable has been reached, the program takes one last sample, then shutdowns all the nodes, closes the data file, and ends the program. Using the control program, we would run multiple simulations back to back by just modifying the variables as needed in our main method calls.

4.3. Results

After running the program 12 times for 2.5 hours for each run and collecting our data, we imported the data into Excel. The first 3 data points for each node were removed. This is because the linear regression calculation we used requires a minimum of three data points. After these initial points were removed, the remaining errors were changed to their absolute value. We calculated the average error across all nodes for the entire run, the maximum error across all nodes, the minimum error across all nodes, and the percentage of data points below the average. In addition, we broke the results down by individual nodes. For each node we found the average error, maximum error, minimum error, and percentage of data points below the average.

The average error is the best "single number" metric to evaluate the performance of the network. A small average error is beneficial because it means that the network on a whole is synchronized. There are bound to be instances where a node is completely out of sync, and that cannot be avoided, so we try to minimize the overall error. The maximum error gives us an idea of how bad the network can get. If we see maximum errors that are orders of magnitude larger than our average errors, we know that at some

point the network was extremely out of sync. Eliminating the source of such errors can greatly increase overall performance.

The percent under average metric gives us insight into the distribution of error points. Having many nodes under the average would indicate that there are a few very high error points and then many more error points that are below average. In such cases, we may find that a few network hiccups have ruined the overall metric when in fact the network was relatively well synchronized for the majority of the time. Conversely, we may see that many data points are above the average. Such a case would lead us to believe that the network performed poorly most of the time, but managed to be well synchronized at a few points. Having around 50% under average means that we have a roughly normal distribution, which may mean that the network is not prone to high or low points.

We also applied a correcting factor to convert our simulation results into numbers more indicative of a sensor network. We did this by dividing the root clock's time at the end of the run by the number of microseconds the simulation lasted. As an example, our baseline criteria run (3σ, errors of $+/-2\mu$, 25 data point limit) had a final root time of 668759150 and ran for 9000000000 microseconds. Dividing these gives us 0.0743. This number is how many microseconds were simulated per "real" microsecond. We multiplied all our absolute errors by this factor. This gives us final results that more closely match sensor network clock error times. The one negative effect of this methodology is that our synchronizations time and sample time of 30 and 23 seconds,

respectively, does not perfectly match the 30 second synchronization and 23 second sample times used in (Kusy, 2006, [17]).

Table 1 presents a simulation performance summary. The table is divided into two sections. The first section shows the runs that used unlimited data points for linear regression calculations. The second section shows the runs limited to 25 data points. In both cases, we present data for 2σ, 3σ, and 4σ outlier rejection criteria and +/-2μs and +2μs error bounds. We show the average error for each run, the maximum error for each run, and the percentage of data points under the average.

Table 1: Simulation Performance Results Summary

Unlimited Data Points						
	Average Error		Maximum Error		% Under Average	
	"+/-Error"	"+Error"	"+/-Error"	"+Error"	"+/-Error"	"+Error"
2σ	29.49062	50.97794	180.0265	315.1288	62.24%	72.14%
3σ	24.26215	26.18788	101.9434	124.9978	62.19%	65.80%
4σ	32.13119	22.77619	283.9537	120.2946	76.47%	63.69%

25 Data Points						
	Average Error		Maximum Error		% Under Average	
	"+/-Error"	"+Error"	"+/-Error"	"+Error"	"+/-Error"	"+Error"
2σ	9.101664	8.368255	73.56993	72.35982	57.86%	59.36%
3σ	11.35326	8.473533	124.4635	37.19237	68.04%	58.53%
4σ	8.078223	10.13886	72.39269	111.9795	64.30%	66.24%

By doing a general comparison between the 25 Data Point sets and Unlimited Data Point sets presented in Table 1, we immediate see that runs using the bounded 25 point set perform better than the unlimited data point runs. For all outlier variables, the 25

data point runs were at a minimum 2 times as accurate (10.13μs vs 22.78μs for 4σ using our modified error bounds) and at the high end were 6 times better (8.37μs vs 50.98μs) for 2σ using our modified error bounds). Maximum errors were also lower in 5 out of 6 instances. The exception is that the 3σ run using the error bounds from (Kusy, 2006, [17]) showed very slightly lower maximum with unlimited data points. This is especially interesting since the 25 data point limitation with 3σ and (Kusy, 2006, [17])'s error bounds is the baseline.

We also notice some interesting results with the percentage of data points under the average. While both the unlimited and 25 data point bounded runs consistently achieve greater than 50% of data points under the average, the unlimited data point runs almost always have more data points under the average than the limited data set. While interesting, this fact is misleading. Even though there are more data points under the average, the average also sits much higher as an absolute value for the unlimited data sets, so the raw average error is still lower for the bounded data set runs.

We were surprised to see that there is little difference overall between a +/-2μs error bound and a +2μs bound. In some cases our bound shows improved performance, but in others the +/-2μs bound performs better. Compare the 4σ average error in the second part of Table 1. Our error bounds ran slightly more than 2μs slower than the +/-2μs error bound. We believe that the linear regression method used in the protocol serves to remove the differences between the two bounds. Linear regression is used to compensate for clock drift which can be 100μs per second. Since synchronizations only occurred every 30 seconds, synchronization errors only introduce 8μs or 4μs of error per

node per minute depending on which bound is used. Both of these are insignificant to the error introduced by clock drift. More generally, we could argue that as long as the total error introduced by the sending node, propagation, and receiving node is much less than the drift of the node's clock, the synchronization error becomes unimportant compared to the natural clock drift.

Based solely on the data from Table 1, it would appear that the 4σ, +/-2μs, 25 data point method performs the best with an average error of 8.08μs, a maximum error of 72.4μs (2σ, +2μs, 25 data points is very slightly lower and 3σ, +2μs, 25 data points is significantly lower), and 64% of data points under average. Another aspect of any clock synchronization protocol that needs to be considered is consistency. A protocol that has an average error of 8μs is a good start, but if it ranges from 0μs error to 16μs, it could present problems at times when the error is very high. A protocol that has a 9μs or 10μs average error but only ranges from 8μs to 11μs might be more worthwhile since the maximum error is significantly less even with slightly higher average error.

We decided to examine the overall stability of each run by graphing the average network error over time. Each time data was collected, we got 10 data points, one for each node. For each 10 data points, we averaged the absolute values. We then graphed the average network error over time and applied a trend line to the data. The idea is that a trend line can approximate the change in error in the network over a long period of time. The ideal trend line would have a negative slope, meaning that as time went on, the network became more and more synchronized with smaller error values. Zero slope would also be an acceptable result. At long as the error remains constant over time, we

can at least know that the errors will not spiral out of control. A positive slope is by far the worst result, as it implies that as time goes on the network becomes more and more inaccurate. The following section shows our plots and trend lines.

4.3.1. Plotted Data

First let us examine the +/-2μs, unlimited data point charts. Figure 5 shows the 2σ simulation results.

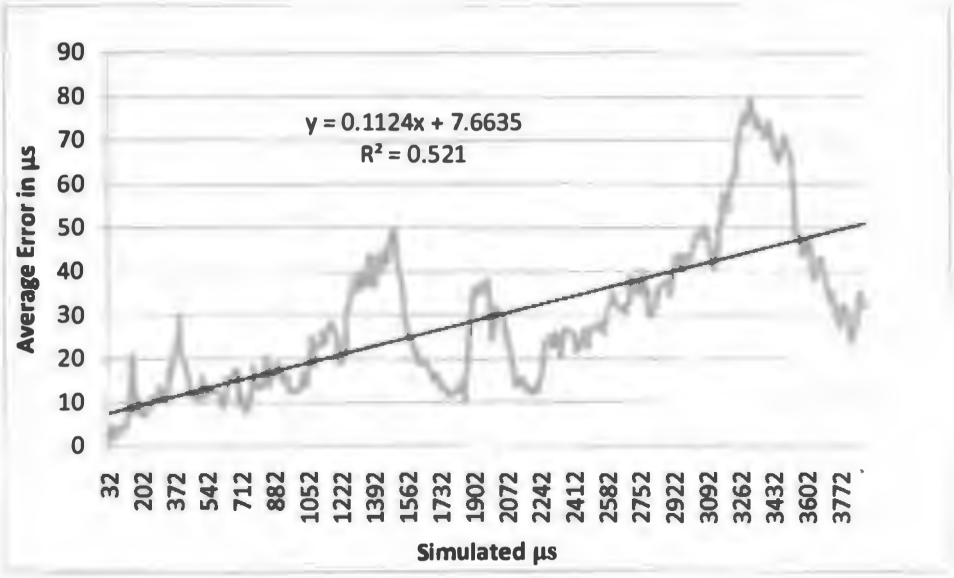

Figure 5: Average Error over Time (2σ, +/-2μs, unlm.)

The trend from the 2σ run is obviously not ideal. The average error grows over time. Figure 6 shows the 3σ simulation results.

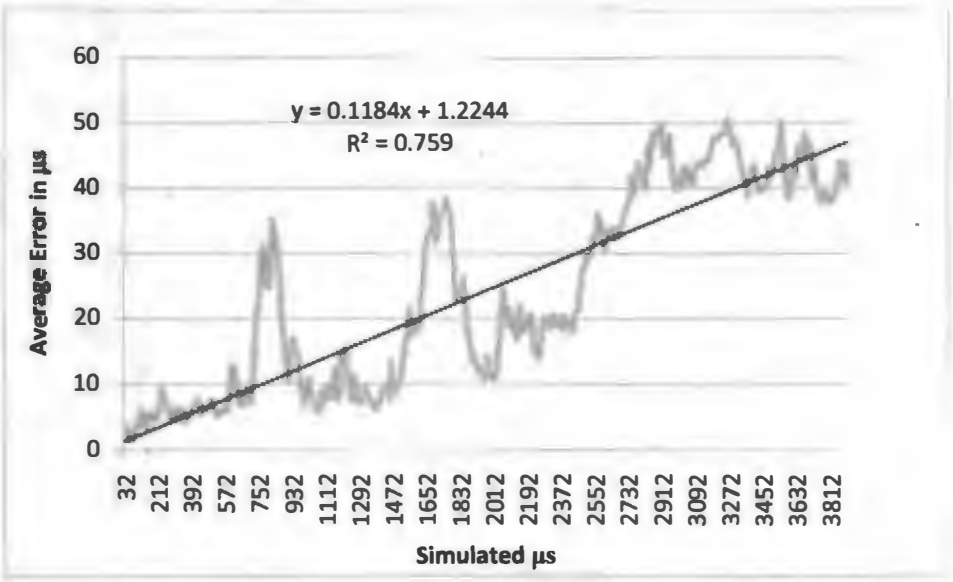

Figure 6: Average Error over Time (3σ, +/-2µs, unlm.)

Like Figure 5, we again see massive growth in average network error as the simulation runs. This trend line is very slightly worse than that of the 2σ run. Finally, figure 7 shows the 4σ simulation results.

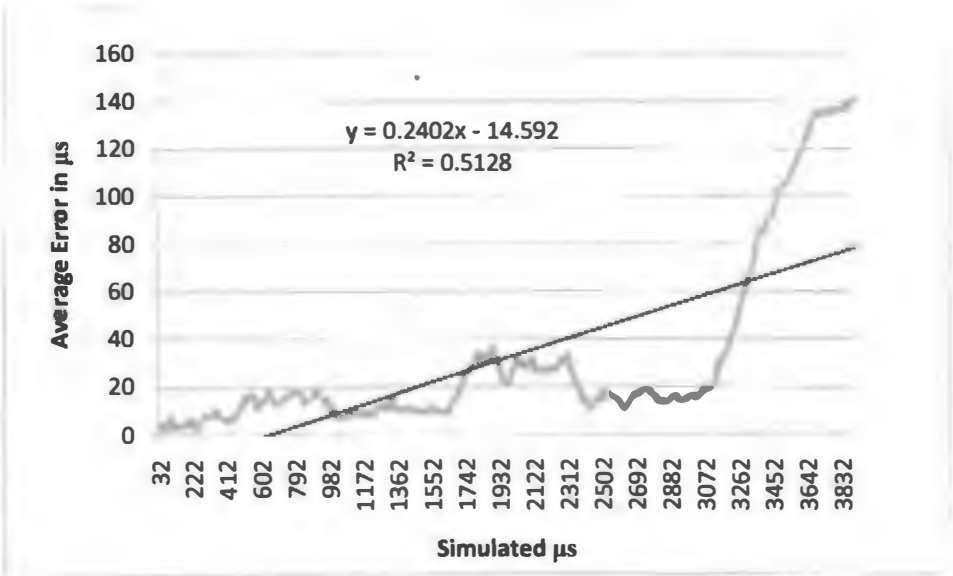

Figure 7: Average Error over Time (4σ, +/-2µs, unlm.)

We can see from Figures 5-7 that in each instance the trend line has a significant upward slope. This indicates that each run experienced ever growing average error in the network. While the R^2 values do not necessarily indicate a strong correlation with the data, we would argue that it is still a valid guide. The data has large variations that make accurate trend lines difficult to generate, but we are only interested in general trends, not specific trends. In each case, it is apparent that over time the networks will become more and more unsynchronized. These results would indicate that using an unbounded number of data points results in compounding errors and ever increasing error levels.

We will now look at the +2μs unlimited data point charts. Figure 8 shows the 2σ results.

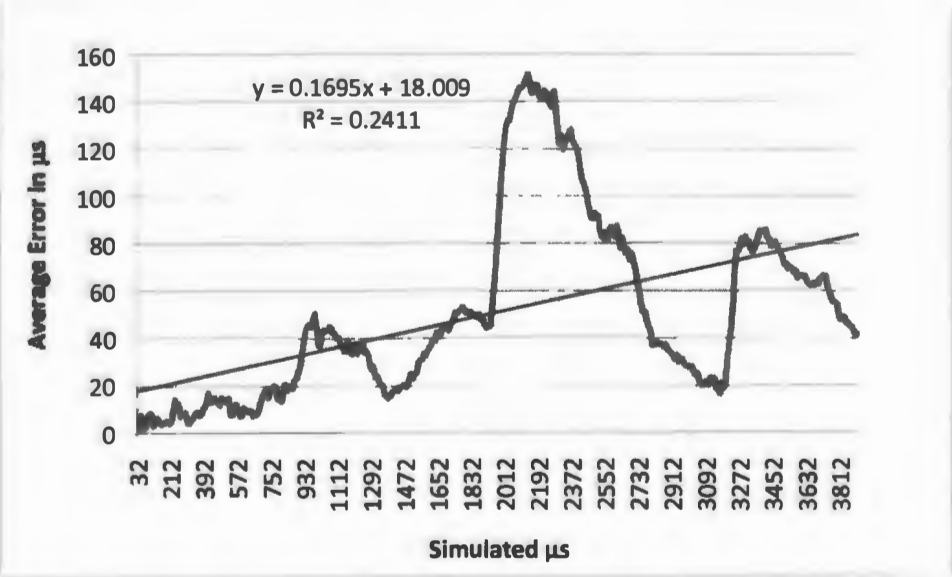

Figure 8: Average Error over Time (2σ, +2μs, unlm.)

Just like in the last set of graphs, we see a strong upward slope to the trend line. Figure 9 shows the 3σ results.

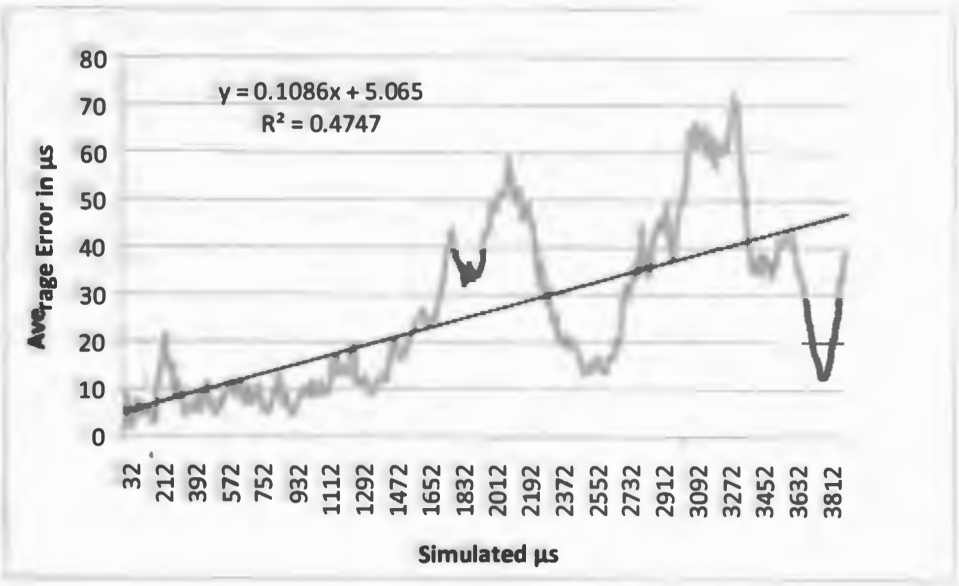

Figure 9: Average Error over Time (3σ, +2µs, unlm.)

Again, strong upward trend. Figure 10 shows the 4σ simulation results.

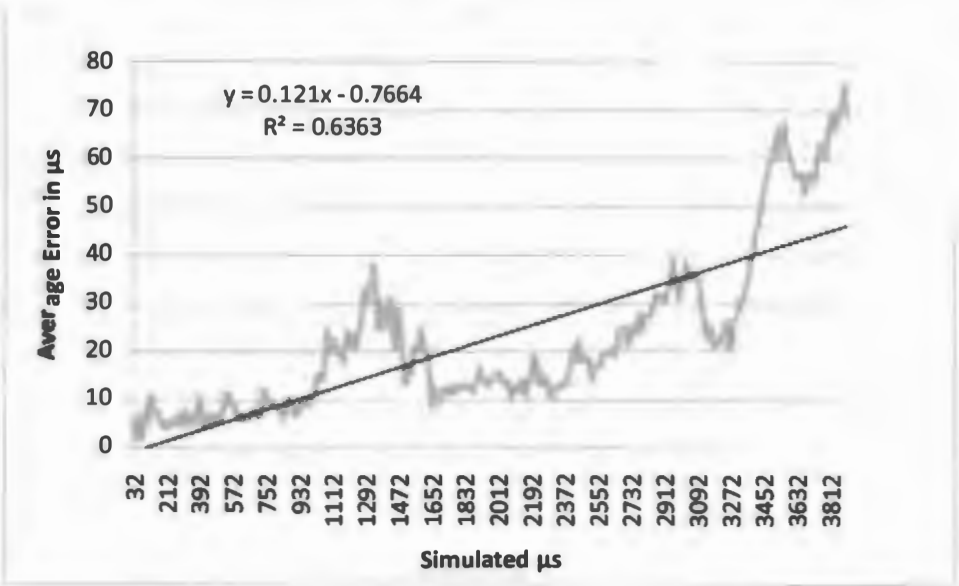

Figure 10: Average Error over Time (4σ, +2µs, unlm.)

We can see that Figures 8-10 for the +2μs error assumption show the same trends as the +/-2μs runs. The R^2 values are slightly higher than the preceding charts. These charts do not surprise us, since the summary data in Table 1 quite clearly shows that the unlimited data set runs had significantly worse performance than the 25 data point runs.

We will now switch gears to the more interesting and accurate 25 data point runs. The following charts show the +/-2μs, 25 data point runs. We start with the 2σ run.

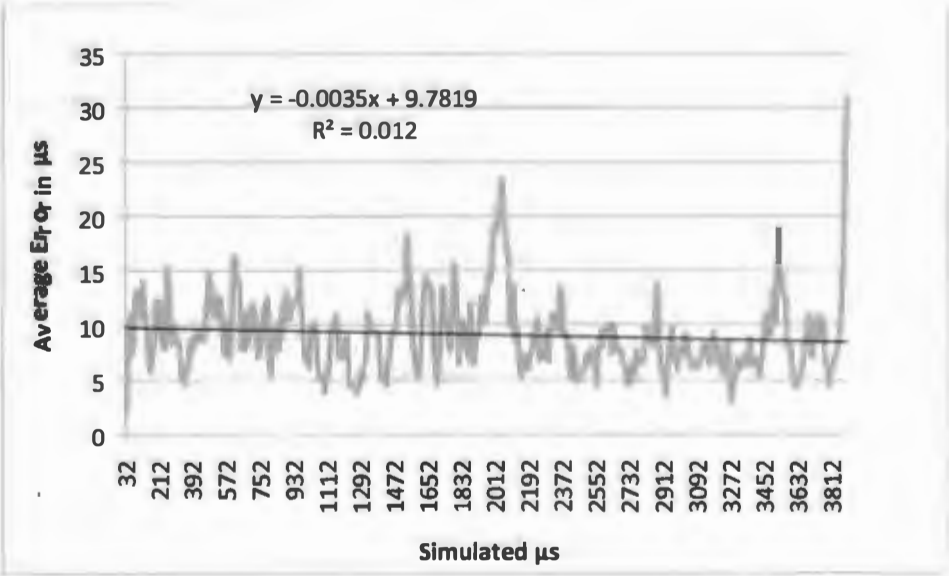

Figure 11: Average Error over Time (2σ, +/-2μs, 25 pts.)

This is our first interesting result. Our trend line appears to have a slight downward trend to it. This is confirmed by the slope of the linear regression expression. This trend indicates that as the simulation progressed, the average error in the network decreased. There is a large spike at the end of the simulation that is of interest. Figure 12 shows the 3σ simulation.

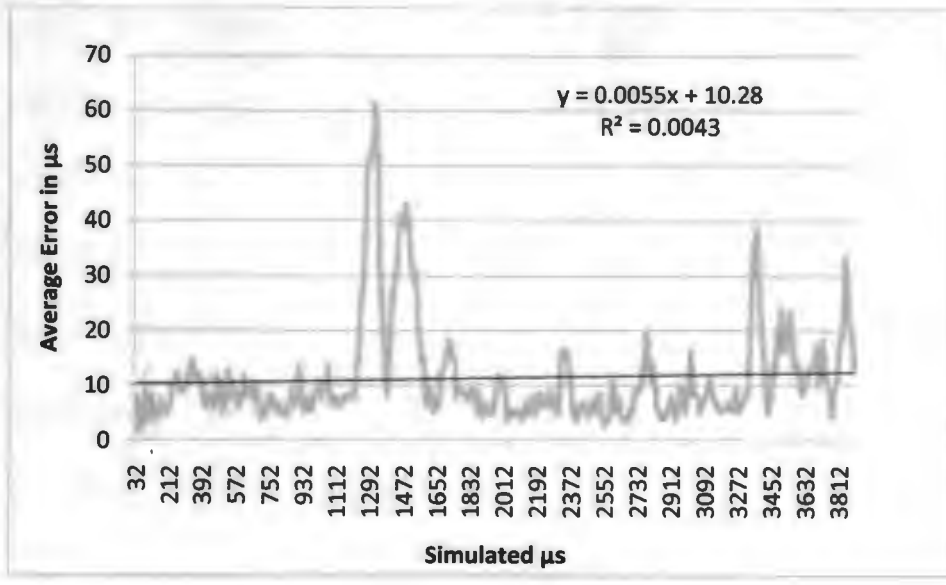

Figure 12: Average Error over Time (3σ, +/-2µs, 25 pts.)

Another interesting result. Here the trend line has a very slight upward trend, but is still much more level than any of the unlimited data point runs. Lastly, figure 13 shows the 4σ, +/-2µs, 25 data point run.

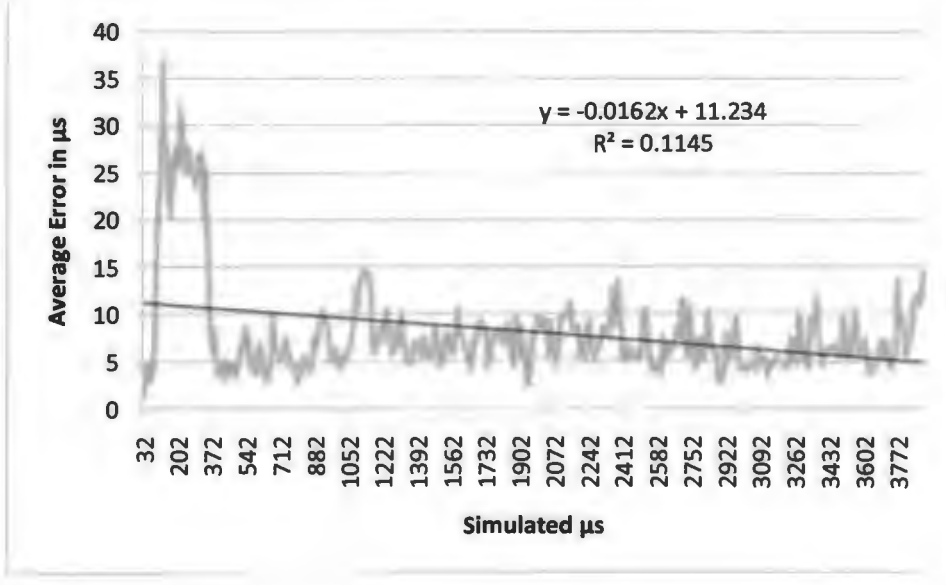

Figure 13: Average Error over Time (4σ, +/-2µs, 25 pts.)

These charts are particularly interesting. Each has a very low R^2 value, but each also shows either a negative trend (Figures 11 and 13) or a very small positive trend (Figure 12). This would indicate that these runs were very stable and over time will have very little error growth, or actually improve the average error. This is exactly the behavior any clock synchronization protocol should display. Figure 13 shows that the 4σ run had the highest R^2 value and also showed a negative trend. This simulation would indicate that using a large error range for outlier detection improves overall protocol performance. This is in direct contradiction to our predictions. We believe it is possible that being too restrictive on outlier detection leads to all new data points being considered outliers and being thrown out. This would be a good argument for widening the outlier detection range. However, we do not believe this is the explanation for our data, since if that was the case, we would expect to see very poor performance in the 2σ run. More study is required for us to fully understand this phenomenon.

It should also be noted that for the 4σ run, the large spike in error at the beginning of Figure 13 greatly throws off the linear regression. If we remove this spike, we would get a much flatter line with a minimum of variation.

Finally, we present the +2μs, 25 data point plots. Figure 14 shows the 2σ simulation run.

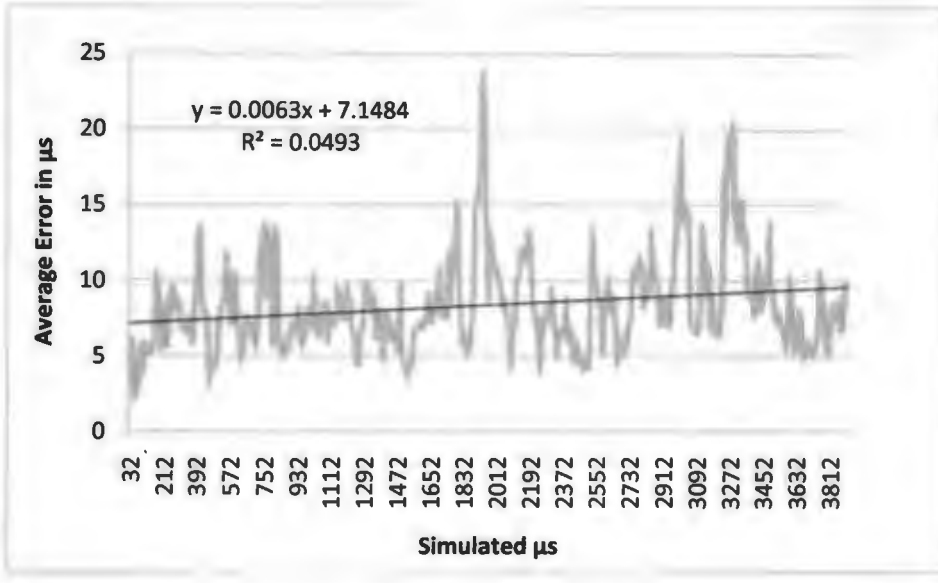

Figure 14: Average Error over Time (2σ, +2µs, 25 pts.)

This graph is very similar to other 25 data point graphs. We see a very slight upward trend in average error here. Figure 15 shows the 3σ simulation data.

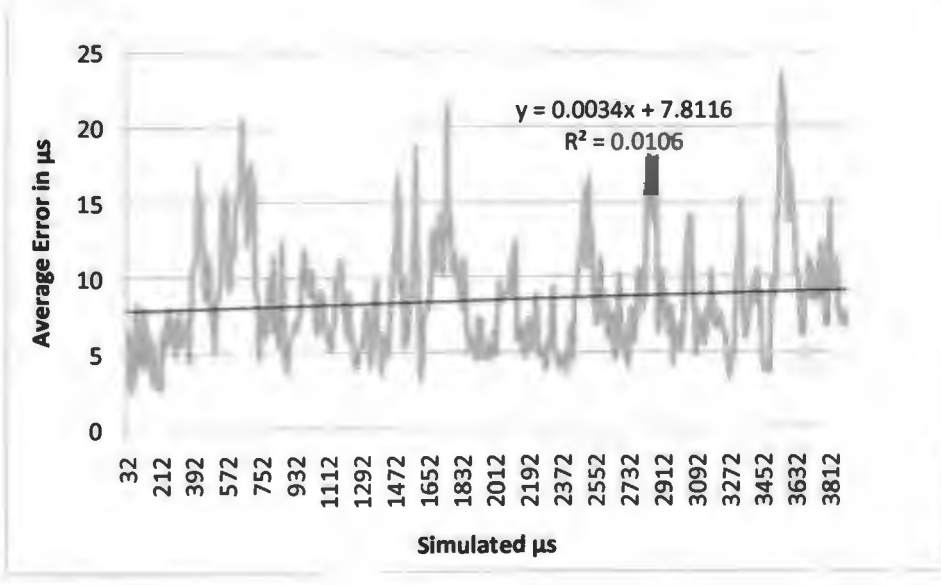

Figure 15: Average Error over Time (3σ, +2µs, 25 pts.)

Again, a very slight upward trend is observed. This is not significantly different from the 2σ run in Figure 14. Figure 16 shows the 4σ simulation run.

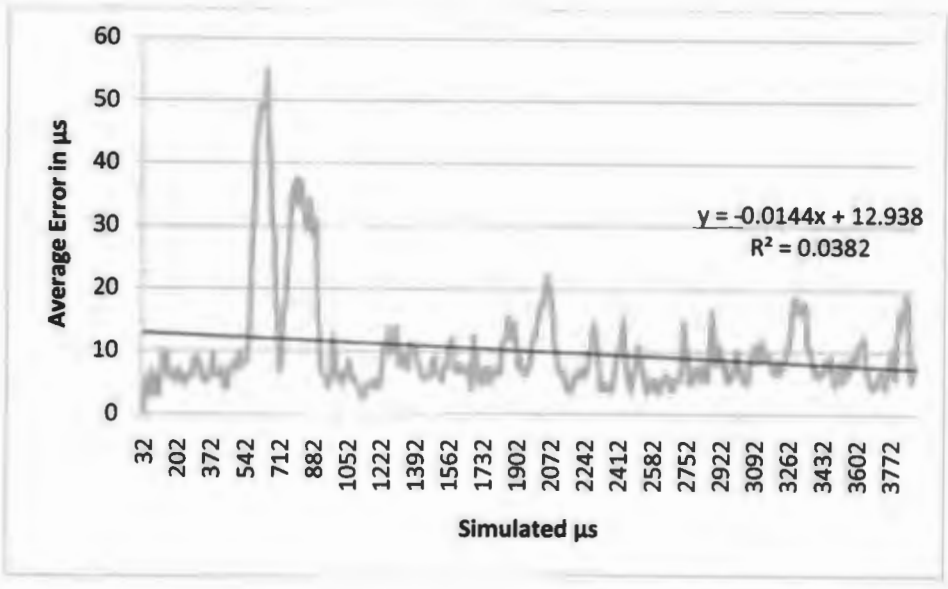

Figure 16: Average Error over Time (4σ, +2µs, 25 pts.)

Both the 2σ (Figure 14) and 3σ (Figure 15) charts are uninteresting. They mimic previous charts. The 4σ (Figure 16) chart, however is rather intriguing. It shows a negative trend and has a large spike much like the 4σ, +/-2µs chart. Since we have gotten this result a second time, the previously stated theory that using a 4σ outlier range improves performance gains extra credibility. This run has slightly worse performance than the +/-2µs run, but the trend is the same.

Our simulation results show that limiting the number of data points used in linear regression drastically improves the effectiveness of the method. This makes sense from a conceptual standpoint because the drift of each clock in the network changes over time.

Using all past data mixes old drift rates with new drift rates while eliminating old data makes sure that the linear regression takes in to account only the newest drift rates.

The experiment also revealed that there is no substantial difference between the +/-2μs error presented in (Hill, 2001, [15]) and our assumption of +2μs. This is due to the linear regression. Such a small difference in error is compensated for with the linear regression. The total clock drift also dwarfs the small difference between +/-2μ and +2μs.

Finally, our simulation showed that using an outlier range of 4σ yields better performance than the traditional 3σ range and also a 2σ range. Using the 4σ range resulted in small error and an overall negative trend in the average network time error. This would imply that over time the network error stabilizes and may actually reduce. The exact reason for this has yet to be determined and will be the topic of future work.

5. CONCLUSION AND FUTURE WORK

We began by presenting a brief review of the problem of sensor network clock synchronization and the challenges involved with creating a suitable synchronization protocol. We discussed in detail the Estimated Time on Arrival (ETA) primitive and how it relies on hardware accelerators to eliminate some of the largest sources of error in clock synchronization. We explained the RITS and RATS protocols built on the ETA primitive. Changes to the protocols were presented. We focused on three main items.

First, we wanted to see what happened if we used an unlimited number of data points when doing linear regression for clock synchronization. Most schemes use 25 data points. We believed that the unbounded number of points would lead to poor performance due to the fact that we would be trying to fit linear data to a curved clock skew line resulting from the aging of the physical clock crystals.

Second, we took issue with the ETA primitive's assumption of a maximum clock error of +/-2μs. We argued that the clock error would skew to the positive side, essentially limiting the total clock error to +2μs. Finally, we wanted to test the outlier elimination criteria. The traditional approach is to throw out new data points that are greater than three standard deviations from the mean. We decided to test the effects of using two standard deviations and four standard deviations. We predicted that using a more strict guideline for elimination (two standard deviations) would result in a more accurate linear regression calculation.

We created a simulation using the Python programming language to mimic the behavior of a ten node network where each node synchronizes its clock to a root node. The program used integer counting to simulation clock ticks. Each node was assigned a clock drift value that was taken into account. We tested +/-2μs and +2μs error bounds for each of the 2σ, 3σ, and 4σ cases using both unlimited data points and 25 data points resulting in 12 different combinations. Data that was collected was imported into Excel for analysis.

The differences between 25 data points and unlimited data points were not surprising. The 25 data point runs performed markedly better than the unlimited runs. The differences between the +/-2μ and +2μs were surprising. There was nearly no difference between the two error bound criteria. We believe this is a result of the linear regression smoothing out the rather minor difference between the two.

The results of the outlier detection were the most surprising. We discovered that the 4σ run had the best performance, followed by the 2σ run, then the 3σ run. We cannot even say that higher error bounds result in better performance because the 2σ run performed better than the 3σ run. Isolating the source of this effect and finding the optimal error bound is a goal of future work.

Our results are interesting because many existing solutions rely on linear regression in some form. We have shown that using a 3σ outlier detection criteria does not necessarily result in the optimal clock synchronization. In fact, the 3σ criteria performed worse than either 2σ or 4σ. The 3σ run had average error of about 11μs while the 4σ run had average error of about 8μs, showing a considerable, but perhaps not

dramatic, increase in accuracy. More importantly though, the 4σ run showed a definite negative trend over time while the 3σ run had a very slight positive trend.

Developing a methodology for calculating the optimal criteria for outlier detection in a linear regression based approach is extremely important, since the entire protocol ultimately relies on the accuracy of these calculations. We hope to conduct further research on this topic to better understand how outlier detection effects overall performance. There is clearly some very interesting relationship between outlier identification and linear regression that directly effects performance of the network.

In this paper we have only looked at linear regression. Other avenues of future research will hopefully include identification of alternative methods for keeping clocks synchronized between time synchronization messages. We will look at the performance impact of these alternative methods, the effects of outlier identification on the performance of a clock synchronization scheme based on the alternative methods, and the computation required by these other methods.

REFERENCES CITED

1. **D. Mills,** *"The Network Time Protocol"* in *Global States and Time in Distributed Systems*, 1994.

2. **L. Girod and D. Estrin,** *"Robust range estimation using acoustic and multimodal sensing"* in *Proceedings of the IEEE/RSJ Internation Conference on Intelligent Robots and Systems*, 2001.

3. **V. Claesso, H. Lonn, and N. Suri,** *"Efficient TDMA Synchronization for Distributed Embedded Systems"* in *20th Symposium on Reliable Distributed Systems*, Pittsburgh, PA, 2001, pp. 198-200.

4. **C. Intanagonwiwat, R. Govindan, and D. Estrin,** *"A scalable and robust communication paradigm for sensor networks"* in *Proceedings of the Sixth Annual International Conference on Mobile Computing and Networking*, Boston, MA, 2000, pp. 56-67.

5. **L. Gu and J. A. Stankovic,** *"Radio-Triggered Wake-Up for Wireless Sensor Networks"* , vol. 29, 2005, pp. 157-182.

6. **H. Kopetz and W. Ochsenreiter,** *"Clock Synchronization in Distributed Real-Time Systems"* in *IEEE Transactions on Computers*, 1987, pp. 933-939.

7. **M. Maróti, B. Kusy, G. Simon, and Á. Lédeczi,** *"The Flooding Time Synchronization Protocol"* in *SenSys*, Baltimore, MD, 2004.

8. **S. PalChaudhuri, A. K. Saha, and D. Johnson,** *"Adaptive Clock Synchronization in Sensor Networks"* in *IPSN*, Berkeley, CA, 2004, pp. 340-348.

9. **P. Sommer and R. Wattenhofer,** *"Symmetric Clock Synchronization in Sensor Networks"* in *REALWSN*, Glasgow, United Kingdom, 2008.

10. **S. Yoon, C. Veerarittiphan, and M. Sichitiu,** *"Tiny-Sync: Tight Time Synchronization for Wireless Sensor Networks," ACM Transactiosn on Sensor Networks*, 2007.

11. **J. Elson, L. Girod, and D. Estrin,** *"Fine-Grained Network Time Synchronization using Reference Broadcasts"* in *Proceedings of the Fifth Symposium on Operating Systems*

Design and Implementation, Boston, MA, 2002.

12. **S. Ganeriwal, R. Kumar, and M. Srivastava,** *"Timing-sync Protocol for Sensor Networks"* in *SenSys*, Los Angeles, CA, 2003, pp. 138-149.

13. **P. Traynor, R. Kumar, H. Bin Saad, G. Cao, and T. La Porta,** *"LIGER: Implementing Efficient Hybrid Security Mechanisms for Heterogeneous Sensor Networks"* in *MobiSys*, Uppsala, Sweden, 2006.

14. Mica2 Datasheet, Crossbow Technology. Retrieved Dec., 2009 [Online]. http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/MICA2_Datasheet.pdf

15. **J. Hill and D. Culler,** *"A wireless embedded sensor architecture for system-level optimization"* , 2001.

16. **N. Gura, A. Patel, A. Wander, H. Eberle, and S. C. Shantz,** *"Comparing Elliptic Curve Cryptography and RSA on 8-bit CPUs"* , 2004.

17. **B. Kusy, P. l. Dutta, P. Levis, M. Maróti, Á. Lédeczi, and D. Culler,** *"Elapsed Time on Arrival: A simple and versatile primitive for canoncial time synchronization services,"* vol. 2, no. 1, 2006.

APPENDIX A

This appendix presents the source code used in the Python simulations. The first section is the node time class that handled the simulated clock for each node. The second section is the node class that creates and runs a simulated node. The third section is the control code that creates the simulated nodes, passes in the simulation parameters, controls synchronization, and collects data.

APPENDIX A.1. Node Time Class

node_time.py

```
"""
Node Time Class

This class defines a time object for use by a network node.  The time object stores:

microseconds (int)
seconds (int)
minutes (int)

drift (int) stores drift value (ppm)

The class supports these operations:

run(self) - clock counts up by 1 continuously until stopped
        drift is added in when appropriate
"""

import threading

class NodeTime:

  global stop

  """A time object"""
  def __init__(self,drift):
    self.micro = 0
    self.second = 0
    self.minute = 0
    self.drift = drift
```

```
"""Primary run method"""
def run(self):

  # Set stop variable to 1 (i.e. run)
  self.stop = 1

  # Determine when to drift
  if(self.drift == 0):
    # Never drift
    drift_every_micro = 1000000

    # self.micro max is 999999
    # 999999 mod 1000000 = 1, so the if test in the
    # while loop is never true
  else:
    drift_every_micro = round(1000000/self.drift)

  while 1 and self.stop:
    #print("Time is " + str(self.micro))
    if(self.micro % drift_every_micro == 0):
      # The current time is a multiple of the drift_every_micro value
      # Drift the clock by 1 (an extra tick does this)
      # Increment clock
      self.micro = self.micro + 1

    # Regardless of drift, increment clock by 1
    self.micro = self.micro + 1

    # Check if increment causes micro rollover
    if(self.micro >= 1000000):
      self.micro = self.micro - 1000000
      self.second = self.second + 1

    # Check if increment causes second rollover
    if(self.second >= 60):
      self.second = self.second - 60
      self.minute = self.minute + 1
```

APPENDIX A.2. Network Node Class

node.py

```
"""
Network Node Class

This class defines a network node.  The network node includes the following information:

Node ID (int)
Node Time (NodeTime object)
Regression Table (dict)
Drift (int - ppm)
Current regression data (dict)
Std dev bounds (int)
Error bounds (int)

A node has the following operations:

give_time() - Return current node time
sync_time() - Syncs clock with root node
end() - Ends the node simulation
give_regressions() - Hands off regression informaiton
sample() - hands off regression info and current time
add_regress() - Adds regression info to node dictionary
lin_reg() - Calculates linear regression variables for node
linreg() - Does linear regression
std_dev() - Calculates standard deviation
minTomac() - Orders a list from min to max using absolute values

Regression information is stored in a dictionary with remote node ID as the key and a list
of tuples of tuples for the regression data points (e.g. 1:[ ((3,4),(5,6)) , ((8,11),(9,12)) ] )
"""
from node_time import *

import math, threading, random

class Node:

  """A network node"""
  def __init__(self,node_ID,node_drift,points,error):
    self.time = NodeTime(node_drift)
    self.ID = node_ID
```

```
        self.regression = {}
        self.a = {}
        self.b = {}
        self.RR = {}
        self.dPoints = points
        self.error = error

        # Build clock thread
        clock_run = threading.Thread(target=self.time.run, name='clck_run', args=[])
        # Start clock thread
        clock_run.start()


    """Hand off current node time"""
    def give_time(self):
        return self.time.micro,self.time.second,self.time.minute



    """Synchronize time with root"""
    def sync_time(self,root_time):
        # Record current node time
        local_time = (self.time.micro,self.time.second,self.time.minute)

        # Establish limits of error
        transmission_error_limit = 1
        receive_error_limit = .25
        sync_error_limit = .625

        # Establish error for this transmission
        # random.uniform returns a random floating point number
        # Decimal values rounded out in a later step
        if(self.error):
            transmission_error = random.uniform(0,transmission_error_limit)
            receive_error = random.uniform(0,receive_error_limit)
            sync_error = random.uniform(0,sync_error_limit)
        else:
            transmission_error = random.uniform(-
transmission_error_limit,transmission_error_limit)
            receive_error = random.uniform(-receive_error_limit,receive_error_limit)
            sync_error = random.uniform(-sync_error_limit,sync_error_limit)
```

```
# Decompose root_time
root_micro = root_time[0]
root_second = root_time[1]
root_minute = root_time[2]

# Subtract out error
# This simulates the clock running while the sync completes.
# In essence, it acts as though the message was sent several cycles ago.
new_root_micro = root_micro - transmission_error - receive_error - sync_error

# Our simulated clock has less precision, so round error to nearest int
new_root_micro = round(new_root_micro)


#print("Base: ",new_root_micro)


# Check to see if time units need to be rolled back
if(new_root_micro < 0):

   if(root_second == 0):
      # Roll back minute
      root_minute = root_minute - 1
      root_second = 59
   else:
      root_second = root_second - 1

   root_micro = 1000000 - abs(new_root_micro)


#print("Corrected: ",new_root_micro)


# Redefine root_time
root_time = (root_micro,root_second,root_minute)

# Create entry in regression table
self.add_regress(local_time,root_time)

"""
   ####Notes on the lin_reg function####
```

The lin_reg function has a term where a value is divided by the length of the input list minus 2

value/(len(list)-2)

Therefore, the input list must have at least 3 values before the lin_reg function can be called. The conditional below enforces this.

This works in the context of the simulation because the more data points we use, the better the linear regression. So having a minimum number of data points doesn't hurt the performance, especially when the minimum is just 3.

"""

```
    # Make sure there is data to lin reg
    # regression[0] references the root node in the regression dictionary
    # regression[0] is a list
    # lin_reg(0) does linear regression for this node and node 0 (the root node)
    if(len(self.regression[0]) >= 3):
        # Perform linear regression
        self.lin_reg(0)

    # Finally, return regression info to control node
    return (local_time,root_time,self.ID)


"""Stop the clock"""
def end(self):
    self.time.stop = 0
    return self.time.micro,self.time.second,self.time.minute


"""Hand off regression info"""
def give_regressions(self):
    return self.regression


"""Hand off regression info and current time"""
def sample(self,node):

    # Make sure data exists for a and b
    if(len(self.a) == 0):
```

```
    a_ret = 0
    b_ret = 0
  else:
    a_ret = self.a[node]
    b_ret = self.b[node]
  #print("A: ",a_ret)
  #print("B: ",b_ret)
  return self.time.micro, self.time.second, self.time.minute,a_ret,b_ret


"""
These are helper functions for the other functions
"""


"""Add data point to regression table for node"""
def add_regress(self,local_time,remote_time):


  """ Version with a maximum of 25 data points for regression"""

  if(len(self.regression) > 0):
    current = self.regression[0]
    """Remove this conditional for unlimited length"""
    if(len(current) == 25):
      # Pop oldest off
      current.pop(0)
    """End remove"""
  else:
    current = []
  current.append( (local_time,remote_time) )
  self.regression[0] = current



"""Calculate regression variables"""
def lin_reg(self,node):
  self.a[node] = 0
  self.b[node] = 0

  # Build x and y lists from regression dictionary
  rootTimes = []
```

```
nodeTimes = []

# Pull out info for "node" from regression list
for regress in self.regression[node]:
   # At this point 'regress' references a tuple in 'regression'
   # indexed to 'node'
   node_time = regress[0]
   root_time = regress[1]

   # node_time and root_time are both tuple of (micro,second,minute)
   # Convert to ints
   temp_sec = node_time[1] * 1000000 # Line shortener variable
   temp_min = node_time[2] * 60000000 # Line shortener variable
   node_time = node_time[0] + temp_sec + temp_min

   temp_sec = root_time[1] * 1000000 # Line shortener variable
   temp_min = root_time[2] * 60000000 # Line shortener variable
   root_time = root_time[0] + temp_sec + temp_min

   # Add data points to x and y lists
   nodeTimes.append(node_time)
   rootTimes.append(root_time)

# Get regression info
temp_a,temp_b,temp_RR = self.linreg(nodeTimes,rootTimes)

# Calc projected node_time
projected = []
for t in range(len(nodeTimes)):
   projected.append(temp_a*nodeTimes[t] + temp_b)

# Build list of actual-calculated
diffs = []
for u in range(len(rootTimes)):
   diffs.append(rootTimes[u] - projected[u])

# Get std dev differences
std_dev = self.std_dev(diffs)

# Get mean
mean = sum(diffs)/len(diffs)

# Throw out outliers if more than X (dev_points) deviations from mean
```

```
        dev_points = self.dPoints
        list_min = 5

        # Order list min to max
        # Gets ordered list and original list
        diffs,org_diffs = self.minTomax(diffs)

        # Remove anything greater than dev_points deviations from the mean,
        # provided that the list is at least list_min long when completed

        i = len(diffs)-1
        while i >=0 and len(diffs) > list_min:
            # Check for dev_points over mean, remove
            if( (diffs[i] > (mean + dev_points * std_dev)) or
                (diffs[i] < (mean - dev_points * std_dev)) ):
                    # Remove from lists
                    # Find index of data point in diffs in the org_diffs list
                    # org_diffs and rootTimes have synced indexes
                    rootTimes.pop(org_diffs.index(diffs[i]))
                    nodeTimes.pop(org_diffs.index(diffs[i]))

            i = i - 1

        # Calc linear regression using purified list
        self.a[node],self.b[node],self.RR[node] = self.linreg(nodeTimes,rootTimes)


    """
    Linear regression function provided by:
    http://www.answermysearches.com/how-to-do-a-simple-linear-regression-in-
python/124/

    Modified to be Python 3 compliant
    """
    def linreg(self,X, Y):

        """
        Summary
            Linear regression of y = ax + b
        Usage
            real, real, real = linreg(list, list)
        Returns coefficients to the regression line "y=ax+b" from x[] and y[], and R^2 Value
        """
```

```
    if len(X) != len(Y):  raise ValueError('unequal length')
    N = len(X)
    Sx = Sy = Sxx = Syy = Sxy = 0.0
    #for x, y in map(None, X, Y):
    for x,y in map(lambda *a:a, X, Y):
        Sx = Sx + x
        Sy = Sy + y
        Sxx = Sxx + x*x
        Syy = Syy + y*y
        Sxy = Sxy + x*y
    det = Sxx * N - Sx * Sx
    a, b = (Sxy * N - Sy * Sx)/det, (Sxx * Sy - Sx * Sxy)/det
    meanerror = residual = 0.0
    for x, y in map(lambda *a:a, X, Y):
        meanerror = meanerror + (y - Sy/N)**2
        residual = residual + (y - a * x - b)**2
    RR = 1 - residual/meanerror
    #RR = 0
    ss = residual / (N-2)
    Var_a, Var_b = ss * N / det, ss * Sxx / det
    #print "y=ax+b"
    #print "N= %d" % N
    #print "a= %g \\pm t_{%d;\\alpha/2} %g" % (a, N-2, sqrt(Var_a))
    #print "b= %g \\pm t_{%d;\\alpha/2} %g" % (b, N-2, sqrt(Var_b))
    #print "R^2= %g" % RR
    #print "s^2= %g" % ss

    #print("A: ",a)
    #print("B: ",b)
    return a, b, RR



"""
Function that returns the standard deviation of a list of numbers
"""
def std_dev(self,nums):
    mean = sum(nums)/len(nums)
    temp = []
    for n in range(len(nums)):
        temp.append( (nums[n] - mean) ** 2 )

    std_dev = ( (sum(temp)/len(nums)) ** (1.0/2.0) )
```

```
    return std_dev


  """
  Function that orders a list from minimum to maximum (absolute vals)
  Returns ordered list and original list
  """
  def minTomax(self,in_list):
    out_list = in_list

    swaps = 1
    i = 0
    while swaps > 0:
      swaps = 0
      for i in range(len(out_list)-1):
        if(abs(out_list[i]) > abs(out_list[i+1])):
          # Swap items
          temp1 = out_list[i]
          temp2 = out_list[i+1]

          out_list[i] = temp2
          out_list[i+1] = temp1

          swaps += 1

    return out_list,in_list
```

APPENDIX A.3. Control Program

control.py

```
"""
Program Flow Control

This controls the simulation and exports data to CSV files

root_micro (int)
root_second (int)
root_minute (int)
samples (int)
```

```
syncs (int)
nodes (dict)

main() - Creates nodes and runs simulation
add_tick() - Ticks the root node's clock
"""

from node import *
from node_time import *
import time, threading



"""
Primary test function
"""
def
main(number_of_nodes,run_time,sync_interval,sample_interval,dev_points,error,filena
me):

  # Clock
  root_micro = 0
  root_second = 0
  root_minute = 0

  # Stats
  samples = 0
  syncs = 0

  # Set up root clock
  node_clock = NodeTime(0)
  root_clock = threading.Thread(target=node_clock.run, name='root_clock', args=[])

  # Startup output
  print("Program running...")
  print("Start time: ",time.ctime(),"\nRun time: ", run_time, " minute(s)\n")

  # List of nodes
  nodes = []

  # Create data file
  data_file = open(filename, 'w')
```

```
    # Put header in file
    write_string = "Node ID,Error,Node Time,Root Time\n"

    # Write header to file
    data_file.write(write_string)

    # Start root clock
    root_clock.start()

    # Build nodes
    i = 0
    while i < number_of_nodes:
        # Random node drift 0-100 ms
        drift = random.randint(0,100)
        new_node = Node(i,drift,dev_points,error)
        nodes.append(new_node)
        i += 1

    # Start the clock
    start_time = time.time()
    end_time = start_time + (run_time*60)


    """
    Optionally, we could run a set number of cycles

    run_time = 100
    j = 0
    while j < run_time:
        #do stuff
        j += 1

    """

    # Start the simulation

    sync_number = 1
    sample_number = 1
    while time.time() < end_time:

        # Tick root clock
        #root_micro,root_second,root_minute =
add_tick(root_micro,root_second,root_minute)
```

```
# Sample data points from nodes
if(int(time.time()) >= (int(start_time) + sample_interval * sample_number)):
    print("Sampling data points")
    samples += 1

    sample_number += 1

    # For each node, get the sample data, calc error, write to file
    for node in nodes:

        # Get sample data
        # Gets data from node for root (node 0)
        node_id = 0
        """Make node IDs start at 1, Excel doesn't like node ID 0 for some reason"""
        micro,second,minute,a,b = node.sample(node_id)

        # Get time as ints for regression
        temp_sec = node_clock.second*1000000 #Temp variable to shorten line
        temp_min = node_clock.minute*60000000 #Temp variable to shorten line
        root_time_int = node_clock.micro + temp_sec + temp_min

        temp_sec = second * 1000000 #Temp variable to shorten line
        temp_min = minute * 60000000 #Temp variable to shorten line
        node_time_int = micro + temp_sec + temp_min

        # Regression time
        reg_time = round(node_time_int*a + b)

        #print("Node: ",node_time_int)
        #print("Reg: ",reg_time)
        #print("Regressions: ",node.regression)

        #print("A: ",node.a)
        #print("B: ",node.b)
        #print("Reg: ",node.regression)


        # Error
        error = root_time_int - reg_time

        # Write string
        write_string = str(node.ID) + "," + str(error) + \
```

```
            "," + str(node_time_int) + "," + str(root_time_int) + "\n"

            # Write node ID and error to file
            data_file.write(write_string)

        # Sync node clocks to root
        if(int(time.time()) >= (int(start_time) + sync_interval * sync_number)):
            print("Syncing node clocks to root")
            syncs += 1

            sync_number += 1

            # Sync the clocks
            for node in nodes:
                # Build root time
                root_time = (node_clock.micro,node_clock.second,node_clock.minute)

                # Sync
                node.sync_time(root_time)



    # Explicitally end nodes
    for node in nodes:
        node.end()

    """End of simulation"""

    # End root clock
    node_clock.stop = 0

    print("Program complete")
    print("Samples: ",samples)
    print("Syncs: ",syncs)


def add_tick(micro,second,minute):
    micro += 1

    # Check if increment causes micro rollover
    if(micro >= 1000000):
        micro = micro - 1000000
        second = second + 1
```

```
    # Check if increment causes second rollover
    if(second >= 60):
        second = second - 60
        minute = minute + 1

    return micro,second,minute



# Program variables

"""Basic Testing
number_of_nodes = 10

#run_time = 150 # In Minutes
#sync_interval = 30 # In Seconds
#sample_interval = 23 # In Seconds

run_time = 2 # In Minutes
sync_interval = 15 # In Seconds
sample_interval = 7 # In Seconds

dev_points = 3
error = 0 # 1 means 0 to high, 0 means -high to high
"""


"""Baseline
number_of_nodes = 10

run_time = 150 # In Minutes
sync_interval = 30 # In Seconds
sample_interval = 23 # In Seconds

dev_points = 3
error = 0 # 1 means 0 to high, 0 means -high to high
"""

"""My version"""
number_of_nodes = 10

run_time = 150 # In Minutes
```

```
sync_interval = 30 # In Seconds
sample_interval = 23 # In Seconds

#dev_points = 3
#error = 1 # 1 means 0 to high, 0 means -high to high


# Test runs
#main(number_of_nodes,run_time,sync_interval,sample_interval,2,0,"2std neg high to
high.csv")
main(number_of_nodes,run_time,sync_interval,sample_interval,2,1,"2std 0 to
high(25lim).csv")
#main(number_of_nodes,run_time,sync_interval,sample_interval,3,0,"3std neg high to
high.csv")
main(number_of_nodes,run_time,sync_interval,sample_interval,3,1,"3std 0 to high
(25lim).csv")
#main(number_of_nodes,run_time,sync_interval,sample_interval,4,0,"4std neg high to
high.csv")
main(number_of_nodes,run_time,sync_interval,sample_interval,4,1,"4std 0 to high
(25lim).csv")
```