12-6-2018 10:00 AM

Current Implementation of the Flooding Time Synchronization Protocol in Wireless Sensor Networks

Asma Khalil
The University of Western Ontario

Supervisor
McIsaac, Kenneth A.
The University of Western Ontario Joint Supervisor
Wang, Xianbin
The University of Western Ontario

Graduate Program in Electrical and Computer Engineering
A thesis submitted in partial fulfillment of the requirements for the degree in Master of Engineering Science

Abstract

Time synchronization is an issue that affects data accuracy within wireless sensor networks (WSNs). This issue is due to the complex nature of the wireless medium and can be mitigated with accurate time synchronization. This research focuses on the Flooding Time Synchronization Protocol (FTSP) since it is considered as the gold standard for accuracy in WSNs. FTSP minimizes the synchronization error by executing an algorithm that creates a unified time for the network reporting micro-second accuracy. Most synchronization protocols use the FTSP implementation as a benchmark for comparison. The current and only FTSP implementation runs on the TinyOS platform and is fully available online on GitHub. However, this implementation contains flaws that make micro-second accuracy impossible. This study reports a complete FTSP implementation that achieves micro-second accuracy after applying modifications to the current implementation. The new implementation provides a new standard to be used by future researches as a benchmark.

Keywords: Wireless sensor networks, synchronization protocols, FTSP, implementation, TinyOS, time synchronization.

Acknowlegements

I would first like to thank my supervisors Dr. X. Wang and Dr. K. McIsaac for their guidance, support and reassuring words throughout my Masters degree. In addition, I would like to thank my “trash-lab” mates who helped me get through the last few difficult months of my degree with coffee, baked goods and genuine friendship. I would also like to thank my dear friend and colleague Madison who was always willing to help and whom without I would not have made it this far. Moreover, I would like to thank my fiance and family for their unwavering love and support through all of the ups and downs. Finally, I would like to thank the beautiful people who work at the University of Western Ontario from the electrical engineering office to the electrical shop who have stood by my side since the very beginning, this thesis truly was a joint effort.

Contents

List of Figures

List of Tables

List of Appendices

Chapter 1

Introduction

During the last decade, the majority of engineering applications and processes have migrated from wired systems to networks of nodes that are wirelessly connected. This change is largely due to the wide range of merits that wireless communication offers such as versatility and scalablity. These properties allow the end nodes to be easily added/removed in addition to accommodating the deployment of nodes in areas that are difficult for humans to reach. WSN's are used in a wide range of applications most of which utilize the mobile nature of the network by placing the sensor nodes on moving parts. This ad-hoc setup allows for many different network arrangements that can be made within wireless sensor networks, however, along with these advantages come added complexities. The unreliable nature of the wireless medium makes it difficult to provide accurate timing information in situations where time-stamp precision is critical. Time synchronization is used to obtain the exact time that the sensor sends/receives data or to calculate the current time of the sensor relative to the rest of the network. The need for time synchronization is largely because in most WSN applications, data is only as accurate as the time-stamp associated with it. To elaborate with an example, a health monitoring application developed by R.A. Bloomfield et al. [1], proposed a knee measurement system which utilizes wireless sensor nodes placed at different locations on a patient's knee to obtain information on the health of the joint. The test involves basic movements of the joint for the course of an hour. In order to reconstruct the correct alignment of the joint after the experiment is completed, one must ensure that the gathered data is modeled from all the sensors at the same time. Otherwise it would not be an accurate depiction of the exercise making the

data unreliable. In scenarios like these and many more, time synchronization becomes a challenge. More specifically, a synchronization protocol is needed in order to ensure the timely arrival of packets with their corresponding correct time-stamps. Synchronization protocols are algorithms that attempt to bridge the gap between the time the packet was sent and the time it was received. This calculation done by estimating either the sent or received time to match the other. For example, in the case of the flooding time synchronization protocol (FTSP), the sender will embed its own time as the global time and the receiver will take its received time as the local time. The protocol then works to estimate the global time from the local time by subtracting calculated estimates of the errors that a packet encounters during wireless transmission and reception.

This study shows the steps taken to provide a practical implementation of the flooding time synchronization protocol. The original FTSP paper by Maroti et al. [2] is used as a basis for this implementation. The available code is studied and it was discovered that there were many discrepancies between the current implementation [3] and the characteristics of the FTSP that were detailed in the original paper. Chapter 2 provides some background on the need of synchronization protocols and the currently used protocols in WSN's. Section 2.2.2 delivers an overview of the most widely used wireless synchronization protocols including the flooding time synchronization protocol. Finally, an argument regarding the integrity of the current implementation [3] is presented and backed by a number of recent studies. Chapter 3 explains the current implementation [3] and the areas where it is lacking. Chapter 3 also includes a fully functional FTSP solution equipped with a detailed explanation of the modifications done. Chapter 4 compares the results of this implementation to what was reported by Maroti et al. [2] and Chapter 5 concludes the study.

1.1 Contributions

The contribution that this thesis achieves is quite important for the field of wireless sensor networks. The work done in this study is motivated by the absence of a complete FTSP implementation. In addition, research has shown that the results reported in the original 2004 FTSP paper are still being used as a benchmark for new and upcoming synchronization pro-

tocols. This work provides a new implementation that will be available for future researchers as the new benchmark for the FTSP. Implementation flaws in the current implementation are addressed and corrected in this thesis and a new hardware calibration step is added to achieve micro-second accuracy. The comprehensive research conducted on the challenging time synchronization problem in this thesis extends the value of the work done to reach a larger audience. Researchers, working with WSN applications which require micro-second level accuracy, will now have a fully functional FTSP implementation with easily reproducible results at their disposal.

Chapter 2

Literature Review

2.1 Wireless Sensor Networks

With the increase in technological advancements, more emphasis has been put on enhancing the wireless efficiency of current systems. Specifically speaking, wireless sensor networks have become the basis under which most new applications are built upon. A wireless sensor network, as the name suggests, is a collection of sensors, stationary or mobile, that are placed in various locations connected together over a wireless medium. Applications of wireless sensors extend from smart home networks, area surveillance to military operations and remote sensing [4]. They are usually placed in areas of interest where measurements are to be taken over a period of time. Initially, WSNs were used to sense and send physical and/or environmental data in order to monitor a certain behavior, for example, their use in smart home monitoring and vegetable greenhouse monitoring [5]. Recent advancements in WSNs have created an added complexity to the networks. Their uses now extend to military target tracking and surveillance, natural disaster relief, bio medical health monitoring, object and behavior tracking and automation and hazardous environment exploration [6,7] to name a few; see Figure 2.1 for more WSN applications.

Furthermore, wireless sensor networks are being utilized in recent medical advancements in order to monitor a patient's health profile remotely by checking the physiological data of the patient [8]. In the military, WSNs are mostly used for detection of any kind of danger or threat such as sensing chemical or nuclear attacks and alerting the appropriate channels.

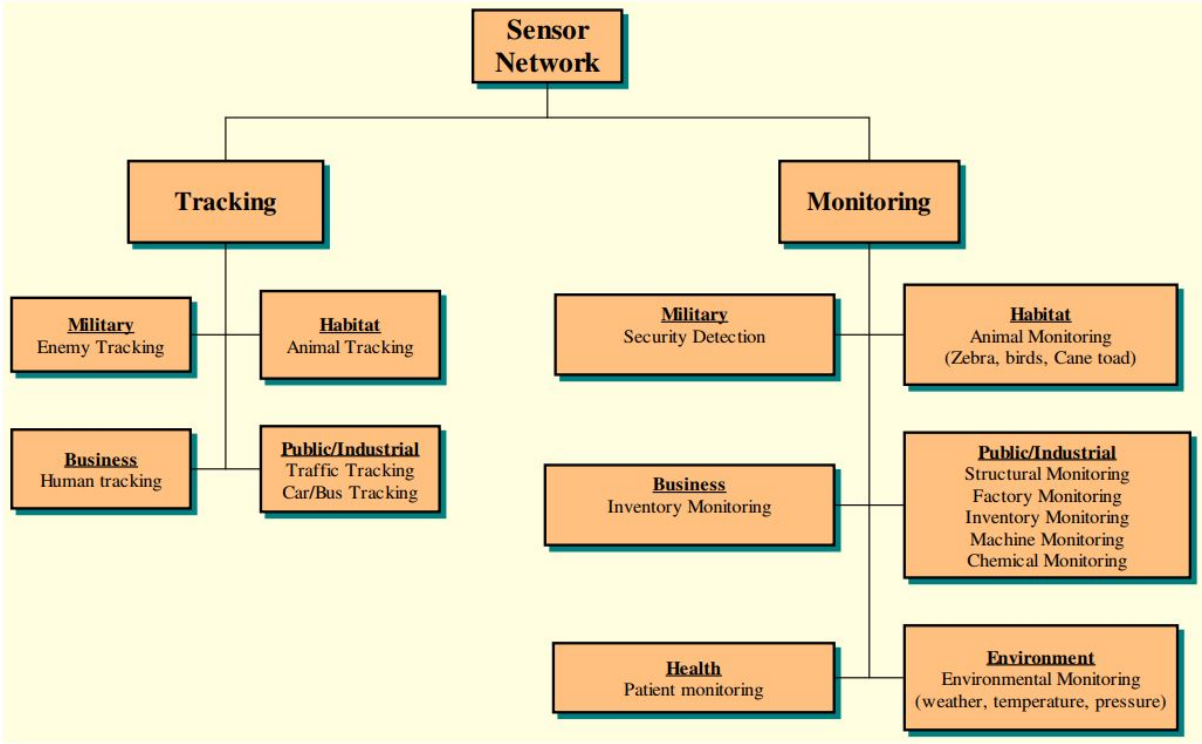

Figure 2.1: Applications of Wireless Sensor Networks [6]

Sensing may also be in the form of visual detection of foreign air crafts for national security purposes. As for a wireless sensor network's role in natural disasters, they are mainly used for prediction purposes by monitoring the environment and making forecasts based on calculations from sensor readings. This kind of WSN application is often used for forest fire monitoring, earthquake detection and gathering data to learn more about certain ecosystems. More common and everyday uses of wireless sensor networks are sensors which monitor public areas such as malls and create alerts if the security is compromised. In addition, some parking lots employ sensors which can help detect empty parking spaces in order to reduce traffic congestion [9]. Finally, a more complex yet very useful application of wireless sensors is in interplanetary exploration and high energy physics [10].

The end nodes of WSNs are characteristically low power devices since they usually consist of one or more sensors, a processor, memory, a power supply, a radio, and an actuator if needed. WSNs have little or no infrastructure, therefore they can be classified into two types: structured and unstructured. Unstructured WSNs are deployed in a location of interest and left unattended

to perform monitoring and reporting functions. This type of wireless sensor network has the advantage of being scalable in size since the nodes are deployed in an ad hoc manner on the field. Node mobility provides the option of sensor deployment in locations that are hard for humans to reach. However, this characteristic causes fault detection and troubleshooting to become more difficult. Structured wireless sensor networks are systems of sensors which are organized in a pre-planned setting. Although this can reduce uncertainty by having easier access and lower management cost, the amount of nodes that can be used is limited [6].

One of the main challenges of wireless sensor networks is energy consumption. This issue is due to the large number of processes that sensor nodes need to perform regularly within their limited lifetime. Another more debated issue that arises is time synchronization which is done to ensure that all of the sensor nodes have a common global time [11]. Due to the nature of events taking place in a wireless sensor network, timing is of utmost importance. The usefulness and validity of the data received is dependent on the time it was received. One of the simplest examples of time-synchronization is implementing power-saving techniques for the sensor nodes. Since these algorithms would require the nodes to switch their radios on/off depending on a certain time schedule, accurate timing must be established among all nodes in the network for these techniques to work [12].

The unreliability of the wireless medium poses a major threat network security in WSNs. The susceptible nature of the wireless communication medium makes it accessible to any device within the vicinity. This lack of security allows an intruder to easily intercept the signal within the network and make malicious changes [13]. Finally, although the scalability of WSNs is considered an advantage, it also introduces stringent constraints on the network that need to be satisfied in order to realize that characteristic [14].

2.2 The Time Synchronization Problem

Time synchronization was initially an issue that was faced by wired networks way before wireless networks and has been studied thoroughly from that angle. The GPS (Global Positioning System), aimed to provide accurate location and timing information for nodes to solve that issue for wired networks. However, it was revealed that the system was not power efficient and

not widely available thereby motivating the development of software based time synchronization protocols such as NTP (Network Time Protocol)[15]. These protocols will ensure that the tasks are ordered and processes are time-stamped by a simple call to the kernel. This procedure confirms that the source of time for all the sensors in the network emerge from the same clock thereby eliminating any ambiguity that is otherwise faced with wireless networks. GPS coupled with NTP displayed great results for time synchronization in wired networks which were in the order of a few microseconds [16].

Due to the nature of wireless sensor networks, it is not possible to replicate the same solution for the time synchronization problem. The limited hardware and computing capabilities of wireless sensor networks in addition to network instability introduced by the wireless medium require the creation of a solution unique to WSNs [17]. Furthermore, the characteristics of wireless sensor nodes as standalone devices impose added complexity. The nodes are each equipped with sensors, their own physical clock and a processor. This set-up introduces a variable variance to the system making it difficult to choose a common time [18]. As a result, time-synchronization protocols are designed in order to minimize the error caused by these uncertainties. In the next section, the different synchronization protocols will be discussed in addition to their role in reducing uncertainty in wireless sensor networks.

2.2.1 Time Synchronization in Wireless Sensor Networks

Synchronization protocols can be classified in many ways, for the purposes of this study, they will be classified by their main features. The most common kinds of synchronization protocols are listed below:

- Internal synchronization vs external synchronization:
  Internal synchronization does not have a global time therefore the aim of the protocol is to minimize the difference between the nodes. With external synchronization, there is a global time such as UTC (Universal Time Controller) that is available and used by the nodes [18].

- Master-slave vs peer-to-peer synchronization:
  Master-slave synchronization is one where there is one clock (master node clock) that all

the other nodes synchronize their clock/time to. With peer-peer synchronization, nodes can communicate directly with each other and by that removing the risk of the master node failing [18].

- Sender-to-receiver vs receiver-to-receiver synchronization:
  Sender to receiver will calculate the delay based on the difference in time-stamps between the sender and receiver in addition to the time it takes to propagate. However, receiver to receiver synchronization eliminates the sender role and thereby all nodes will only operate as receiving nodes. The time-stamps are calculated based on the difference in times between two receivers when they get the same message [19].

- Clock correction vs un-tethered clocks:
  Due to the difference in characteristics of hardware clocks and crystal oscillators between nodes, the clock drift issue arises. Some synchronization protocols will perform a certain clock-correction mechanism to account for that problem [18].

- Probabilistic vs deterministic synchronization:
  This refers to the way that the clock offset is measured. With probabilistic synchronization, the maximum clock offset is calculated based on a probabilistic guarantee with a known probability of failure. Deterministic synchronization protocols will have an upper and lower bound set for the clock offset.[18]

- MAC-layer-based approach vs non-MAC-layer-based approach:
  Some synchronization protocols are implemented at the MAC layer and are tightly linked to the access scheme used in order to perform time-synchronization making it more accurate however not modular. In other protocols which are not at the MAC layer, although they will not offer the same accuracy as those who are but have the advantage of being modular [10].

In addition to the attributes discussed, the synchronization protocols are also characterized by the kinds of errors they help minimize. As shown in Figure 2.2, the main errors faced by wireless sensor networks are a result of radio message delays. Table 2.1 summarizes the delays that significantly affect radio message delivery.

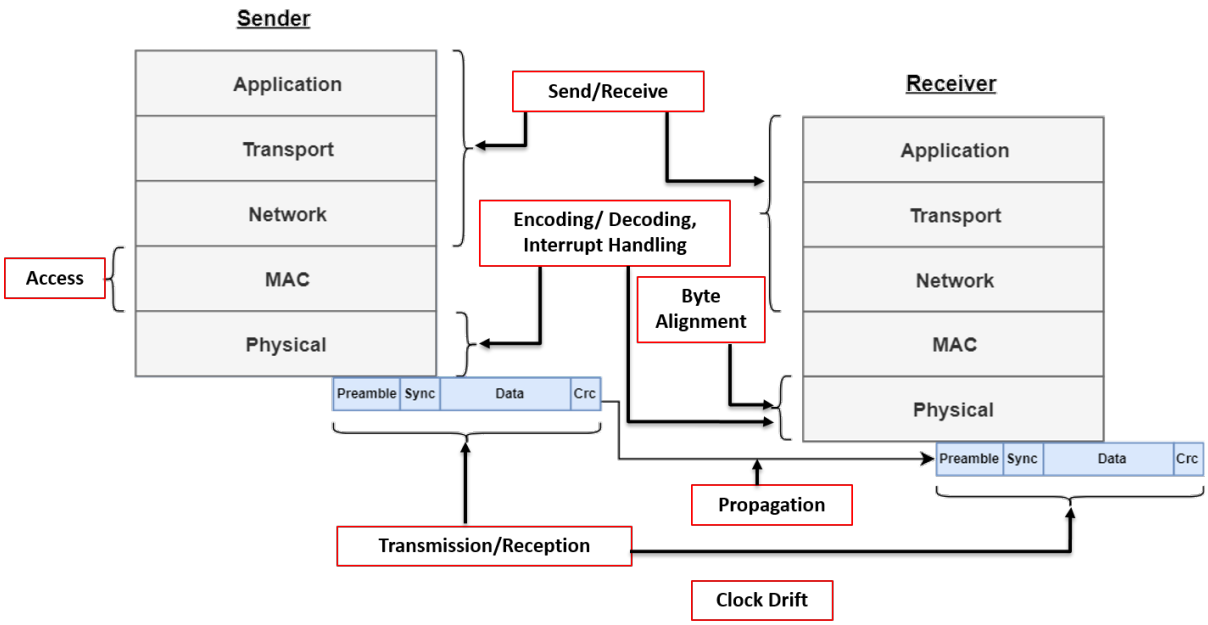

Figure 2.2: Sources of Radio Message Delay

2.2.2 Time Synchronization Protocols for Wireless Sensor Networks

Accurate time synchronization is one of the most debated issues faced by wireless sensor networks; therefore, there are many protocols that hope to bridge this gap. In this section, various kinds of protocols that are being used to synchronize the times of the nodes within a wireless network will be discussed. In addition, a comparison will be made using the attributes discussed in Section 2.2.1. The initial comparison of the most common time synchronization protocols for WSNs is done in Tables 2.2 and 2.3 [20, 21]. A more detailed explanation on each synchronization protocol along with a brief overview of the algorithms used is also provided in this chapter. Finally, an in-depth study on the Flooding Time Synchronization Protocol (FTSP) is presented.

Reference Broadcast Time Synchronization (RBS)

The reference broadcast time synchronization protocol is a receiver-receiver based deterministic protocol that offers high energy conservation due to its post-facto synchronization [28]. Post-facto synchronization means that the synchronization of the nodes is only done when necessary. In addition, RBS protocol uses the broadcasting feature of wireless communications, this gives an added advantage since a broadcast means the message can reach a number

Table 2.1: Overview of Radio Message Errors

Radio Message Delay	**Comments**	**Error Magnitude for Mica2**
Send/Receive	Time for the assembly of the packet to be sent and signalling to MAC layer/ Time for the processing of received message and signalling to receiving application.	0 - 100 ms
Access	Time for packet to access the channel.	10 - 500 ms
Transmission/ Reception	Time for packet to be sent from the first to last bit/ Time for the packet to be received from the first to last bit.	10 - 20 ms
Propagation	Time for the packet to travel through air.	<1 μs
Interrupt Handling	Time incurred from sections in the code disabling interrupts which are raised from the radio chip to indicate message transmission or reception to the microprocessor.	5 - 30 μs
Encoding/Decoding	Time to transform binary data to electromagnetic waves/ Time to transform electromagnetic waves to binary data.	100 - 200 μs
Byte Alignment	Delay incurred from bits being received in incorrect order.	0 - 400 μs
Clock Drift	Delay due to different characteristics of hardware oscillators making them have different frequencies.	>40 μs

of receivers at almost the same time. Similarly to the flooding time synchronization protocol (FTSP), RBS also exploits this physical property of the wireless medium [18].

Although RBS does not employ MAC layer time-stamping, it can obtain fairly good accuracy because of its receiver-receiver feature. This attribute will allow the elimination of non-deterministic errors that arise from radio message delivery, such as send and access time, through reducing the critical path. Figure 2.3 shows how the critical path is reduced by choosing receiver-receiver based synchronization. As illustrated in Figure 2.3, since both receivers will receive the broadcast message, the difference between their local times can then be taken in order to estimate the clock offset and correct their times accordingly. Even though RBS can provide good accuracy, it does not account for any clock correction among the nodes since

Table 2.2: Comparison of Synchronization protocols

Protocol	**Master-to-slave/ Peer-to-peer**	**Internal/ External**	**Sender-receiver/ Receiver-receiver**
RBS [22]	Peer-to-peer	Both	Receiver-receiver
TPSN [23]	Master-to-slave	Both	Sender-receiver
FTSP [2]	Master-to-slave	Both	Sender-receiver
DMTS [24]	Master-to-slave	Both	Sender-receiver
Romer's protocol [25]	Peer-to-peer	Internal	Sender-receiver
Mock et Al. [26]	Master-to-slave	Internal	Receiver-receiver
Sichitiu and Veerarittiphans [27]	Peer-to-peer	Internal	Sender-receiver

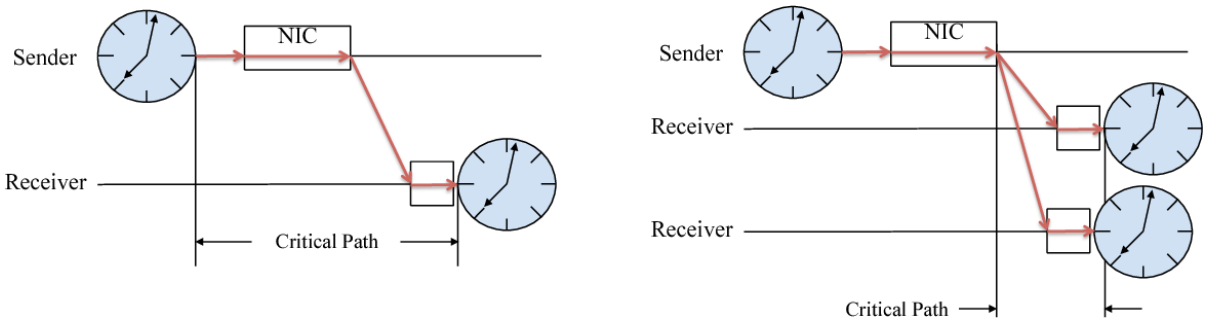

Figure 2.3: Critical path comparison, the sender-receiver critical path (left) and the receiver-receiver critical path (right) [22]

there is no global time to synchronize to. Although this may reduce cost, it comes at an expense to the precision of the protocol [18].

Timing-Sync Protocol for Sensor Networks (TPSN)

The timing-sync protocol for sensor networks is a protocol that improved upon the RBS protocol's time-stamping weakness. This was achieved by performing MAC layer time-stamping which resulted in the synchronization error measured for TPSN being reduced by over half than that of RBS, refer to Table 2.3. TPSN takes a more common sender-receiver approach with message exchange offering easier handshaking between nodes in order to achieve network-wide synchronization.

Table 2.3: Comparison of Synchronization protocols Cont'd

Protocol	**Clock Correction**	**Probabilistic/ Deterministic**	**MAC layer/ Standard**	**Sync Error (μs)**
RBS [22]	No	Deterministic	Standard	29.1
TPSN [23]	Yes	Deterministic	MAC layer	16.9
FTSP [2]	Yes	Deterministic	MAC layer	1.48
DMTS [24]	No	Deterministic	Standard	32
Romer's protocol [25]	No	Deterministic	Standard	200
Mock et Al. [26]	Yes	Deterministic	MAC layer	300
Sichitiu and Veerarittiphans [27]	Yes	Deterministic	MAC layer	3000

Prior to the start of this protocol the authors have assumed that the network has a hierarchical structure consisting of nodes which are each assigned a certain level (i.e. level i, level $(i + 1)$, ..). This protocol has two main phases:

- Level Discovery phase.
- Synchronization phase.

In the discovery state, a root node is selected and given a label: level 0. The newly elected root node will then start the synchronization phase by synchronizing each level i node with a level $(i - 1)$ node until they are all synchronized to the level 0 node; the root node [23].

Delay Measurement Time Synchronization Protocol (DMTS)

This sender-receiver algorithm elects a master node which transmits the same synchronization message to all the receivers at the same time. The master's time is taken as the global time and all the receivers will calculate their respective delays from this global time. They will then set their local time to be the global time plus the delay calculated. The delay that is calculated will take care of some radio message errors, however, the delay caused by the transmission time, send/receive times and access time remain. This protocol deals with those two errors in the following ways:

- Sender processing time and access time: The protocol will only take a time-stamp when a clear channel is detected to remove the error caused by those two delays.

- Transmission time: This protocol divides the transmission times into two separate times since the transmit speeds may be different for each: preamble/start symbols transmission time and the data transmission time. The delay is then calculated from the speeds of each part of transmission [24].

Romer's Protocol

Romer's protocol was created for time synchronization in Ad Hoc Networks. AD Hoc networks describe networks in which their nodes are mobile and prone to changes. The protocol does not work by synchronizing clocks, however, it works on synchronizing time stamps of local clocks through time transformation. Time synchronization is done by first embedding a time stamp in the message being sent, and when it is received, the receiver will transform the sender's time-stamp to the Coordinated Universal Time (UTC) and finally to the local time of the receiver. This procedure gives a lower and upper bound on the time stamp in addition to the real-time. The time taken from the generation of a time stamp at the sender to the reception of it at the receiver node. The lower and upper bounds of this time are calculated and transformed to the time of the receiver. The receiver, now equipped with the upper and lower bounds will subtract this transformed time from the time of arrival which is taken at the reception by the receiver's local clock [25].

Mock et Al.

This protocol utilizes the master-to-slave mechanism and extends the IEEE 802.11 standard where a chosen master will send out a "high-priority" message for all the other "slave" nodes to synchronize their virtual clocks to. In addition, this protocol is similar to RBS since it too uses the property of the wireless medium and by that reduces the critical path in the same way. This is summarized by the following steps:

- The master prepares an "indication message" at time t_1 and broadcasts it at time t_2.

- All of the nodes (master and slave) then receive the "indication message" and record their local time stamp at reception.
- The master then sends its time stamp for the last indication message to all the slaves.
- The slave nodes then correct their local clocks based on the difference between the received time stamp and the local time stamp.

Mock et al. defined a unique rate-based correction algorithm that provides continuous time synchronization for applications where message loss can't be tolerated. Initially, a maximum value is set on the number of messages that can be lost during transmission, this is labeled as "OD" or omission degree. The number of time stamp values "n" needed to ensure that not more than "OD" number of messages are consecutively lost is then calculated, this is given by: $n = OD + 1$. The protocol then includes the last calculated "n" number of synchronization messages within the current synchronization message. This procedure will allow the receiver to synchronize its clock to the master even if a current message loss was detected [26].

Sichitiu and Veerarittiphans Protocol

This protocol works with both the Mini-sync and Tiny-sync algorithms in order to achieve deterministic clock synchronization. It is characterized by low complexity and computational power in addition to operating with limited resources. Tiny-sync and Mini-sync are different in the amount of resources they use; Tiny-sync using considerably less resources than Mini-sync. They do however share a lot of common features such as the deterministic nature of calculating the clock offset and their tolerance to message losses. The way the algorithms calculate the clock offset is by extending the set-valued estimation method. Sichitiu and Veerarittiphans protocol modifies the set-valued estimation method slightly and relates the processors and local time of the nodes in a network to each other using the linear equations below:

$$\begin{aligned} t_1(t) &= a_{12}t_2(t) + b_{12}, \\ a_{12} &= a_1 - a_2, \\ b_{12} &= b_1 - b_2 \end{aligned} \tag{2.1}$$

Where $t_1(t)$ and $t_(2)$ are functions of the local clock of nodes 1 and 2 respectively and t is the UTC. Variables a_i and b_i are the clock drift and offset of the i^{th} node. These equations provide a data collection algorithm to get data points (at least two) which are then used for time synchronization. The main three data points needed to set the clock drift and offset are governed by the inequalities below:

$$\begin{aligned} t_o(t) &< a_{12}t_b(t) + b_{12} \\ t_r(t) &> a_{12}t_b(t) + b_{12} \end{aligned} \tag{2.2}$$

Where t_o is a probe message sent from node 1 to node 2 and t_b is the same probe message time-stamped at the receiver and returned to the sender (node 1) who then records the time of reception of this message as t_r.

Flooding Time Synchronization Protocol (FTSP)

The Flooding Time Synchronization Protocol (FTSP) is a sender-receiver protocol created by Maroti et al. that achieves micro-second accuracy in multi-hop networks. FTSP's robustness and level of precision is considered to be the best when compared to other available protocols for sensor networks and has been used for time synchronization in a counter sniper application [29]. Table 2.3 shows how FTSP's synchronization precision surpasses all the other synchronization protocols described with an average reported time synchronization error of a single-hop case being 1.48 μs. FTSP differentiates itself from other protocols with its ability to create a synchronization point with only one broadcast message and its promise to remove interrupt jitter through several techniques such as multiple time-stamping [2, 30].

This protocol uses the flooding or broadcasting of the synchronization message (which includes the root's time otherwise called the global time) to all nodes and the receiving node would generate its own local time thereby creating a global-local time pair. This difference in the times is therefore called the offset for that pair and since the timestamp is embedded in the synchronization message, overhead is reduced by achieving time synchronization though one message transmission. Most of the errors that arise from radio message delivery can be eliminated at the MAC layer time stamping; however, interrupt handling time, encoding/decoding times, clock drift and byte alignment are all sources of error that this protocol aims

to reduce. Both the interrupt handling time and the encoding/decoding time are dramatically reduced through the multiple time stamping technique described in the next section. The byte alignment time is calculated using the SYNC bytes and linear regression is used to reduce clock drift as explained in the sections below.

FTSP also extends its protocol to network-wide time synchronization through multi-hop synchronization using a unique node ID. This feature allows the network to be dynamic and the synchronization root node to be re-elected whenever needed. The mechanism with which this protocol elects/re-elects the root is as follows:

1. When a node waits for "ROOT TIMEOUT" number of seconds without receiving a synchronization message, it declares itself to be the root node, this ensures that there is at least one root in the network after "ROOT TIMEOUT" number of seconds.

2. In order to ensure that there is only one root node in the network (adhering to the master-slave dynamic),when a node receives a message with a root ID smaller than its own, it updates the nodes root ID with the one that was just received.

3. Finally, all the nodes with higher root ID will give up their status to the ones with lower root ID until the lower remains and only one root in the network remains.

4. Every node will then be synchronized to the global time of the node one level higher that itself [2, 30].

Multiple Time-stamping : The Flooding Time Synchronization Protocol credits its multiple time stamping technique taken at both the sender and receiver for its micro-second accuracy. Maroti et al. [2, 30] documented that multiple time-stamps (≈ 6 according to their calculations) are to be taken at each byte boundary after the SYNC bytes have been sent. The format of the packets is shown in Figure 2.4. The aforementioned time stamps will then be normalized by subtracting a certain delta from them which corresponds to the nominal byte transmission time after which they will be minimized and finally averaged. A more detailed explanation of this procedure is described below. As indicated in Figure 2.5, t_1 to t_6 are to be minimized by choosing the minimum of each two normalized time-stamps starting with t'_6 being equal to t_6. Maroti et al. defines the BYTE TIME as the time it takes to transmit a byte and calculates it based on

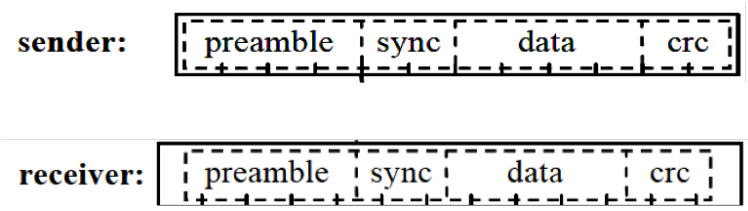

Figure 2.4: Packet format

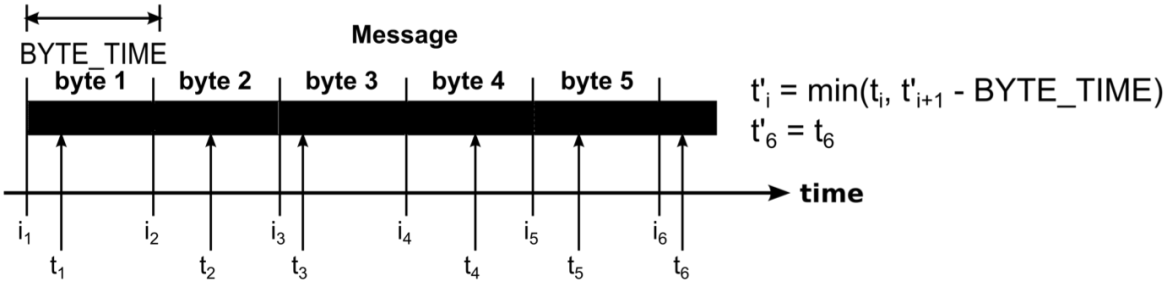

Figure 2.5: Multiple Time-Stamping Technique

the transfer rate of the hardware. The transfer rate indicated in [2] is 38.4 kbps for the Mica2 hardware used and since each cycle will transmit 16 bits (address bits followed by the 8 data bits)[31] this computes a delta of 417 μs [32]. However, in the case of the Mica2, this research shows that this delta value does not hold due to hardware clock instability, Chapter 3 provides more insight on this issue. To elaborate on the time-stamping procedure, see the equations below.

$$\begin{aligned} t'_5 &= min(t_5, t'_6 - \Delta) \\ t'_4 &= min(t_4, t'_5 - \Delta) \\ t'_3 &= min(t_3, t'_4 - \Delta) \\ t'_2 &= min(t_2, t'_3 - \Delta) \\ t'_1 &= min(t_1, t'_2 - \Delta) \end{aligned} \tag{2.3}$$

The average of these time stamps is then calculated as follows:

$$t'_{avg} = \frac{\sum_{i=1}^{6} t'_i}{6} \tag{2.4}$$

This final timestamp is then further corrected with the byte alignment and clock drift approximation calculations and then embedded in the same packet.

Byte Alignment: Byte alignment errors arise from the difference in the order of the sent and received bytes. FTSP combats this at the receiver by utilizing the synchronization (sync) bytes in order to indicate the start of the data received. Once the preamble bytes are received in the listen state, the protocol moves into the synchronization (sync) state where it compares the incoming bytes to the sync bytes and stores the number of bits it took until the sync bytes are received, this is called the offset. This value is obtained from the speed of the radio. For example, at a data rate of 19.2 kbps, the time delay is calculated for a bit offset of 0 to 7 bits to be 0 to 365 μs respectively [2], see Table 2.4. These values are then used as a look-up table to provide the delay incurred for the bit offset calculated at the receiver.

Table 2.4: Byte Alignment Time Delay at 19.2 kbps

Bit Offset	**Corresponding Time in μs**
0	0
1	52.1
2	104.3
3	156.4
4	208.5
5	260.6
6	312.7
7	364.8

Clock Drift: The implementation of the protocol [3] was done on the Mica2 [33] hardware motes and because different sensor motes have different clocks, errors resulting from clock drift are inevitable. FTSP uses linear regression and the method of least squares to calculate clock drift and correct the timestamp accordingly. The method of least squares is defined by the following steps:

A number of ordered pairs (x_i, y_i) must be obtained to calculate the mean, refer to Equation

2.5.

$$\overline{X} = \frac{\sum_{i=1}^{n} x_i}{n} \qquad \overline{Y} = \frac{\sum_{i=1}^{n} y_i}{n} \tag{2.5}$$

The next step is to calculate the slope of the line of best fit using these calculated averages for any new data point. This is used to forecast the upcoming data points and by that reduce uncertainty. The equation to calculate the slope using this method is shown below:

$$m = \frac{\sum_{i=1}^{n}(x_i - \overline{X})(y_i - \overline{Y})}{\sum_{i=1}^{n}(x_i - \overline{X})^2} \tag{2.6}$$

The authors of the paper have indicated that they achieved good results using only 8 data points. This means that every 8 packets received, the linear regression and method of least squares is used to recalculate the slope and then use it as a multiple to help predict the upcoming data. In the case of this protocol, the data points were (time, offset) where offset is the difference between the local and global times and time is the current local time of the mote. They then set the slope to be equal to the clock skew and used it to transform the local time to a global time. This was the mechanism through which FTSP eliminated clock drift. Although the effect of clock drift might be slower than other phenomena, the Mica2 oscillators introduce drifts of up to 40 μs per second [2, 30].

FTSP in Recent Studies

According to a recent study made by D. Djenouri and M. Bagaa [34] on synchronization protocols, it was argued that an implementation which truly follows the FTSP guidelines was not currently available . Furthermore, new and upcoming synchronization protocols are still comparing their efficiency to the values that were reported in the original FTSP paper. For example, Glossy [35] is a synchronization protocol designed to flood the network with the goal of implicit time synchronization. In the 2011 paper explaining the Glossy protocol, the authors and creators of the protocol compared it to FTSP. They mentioned the synchronization error (in the microsecond rage) obtained in Maroti et al.'s paper and used it to show how their protocol compares to that without re-creating the implementation.

Average TimeSync (ATS) [36] is another synchronization protocol created in 2011 which also uses FTSP as a benchmark in addition to calling it "the defacto standard for time synchronization in WSN". In their testing they have indicated that they did use the widely available FTSP TinyOS [37] implementation in addition to testing both protocols in a 3x3 WSN grid with a synchronization period T = 60 s. Since this study argues that the available online code is not a true translation of the flooding time synchronization protocol, the relevancy of the previous comparison might be affected.

In 2011, Thomas Kunz and Ereth MCKnight-MacNeil [38] implemented the clock sampling mutual network synchronization (CS-MNS) algorithm in TinyOS on a hardware similar to the Mica2 and compared its performance to that of the FTSP. It was found that the FTSP performed rather poorly than otherwise reported in the original paper. Their final results show that the CS-MNS algorithm had a final synchronization error of 31 μs compared to 61 μs for FTSP. While the authors of [39] attempted to use the TinyOS 2.x [3] implementation to recreate the results in order to compare with their own protocol, they found that while using the default parameter settings, the available code FTSP was unable to synchronize the nodes in the network. The faults they have uncovered in the implementation lead to them modifying the code in order to obtain results which could be comparable to their own. This variation in the FTSP code is due to its shortcomings and results in possibly different adaptations of the protocol which may not be a true translation of the algorithm of FTSP.

The Energy-Balanced time Synchronization protocol (EBS) described in [40] uses FTSP as a benchmark when analyzing the efficiency of the synchronization protocol. The novel time synchronization protocol introduced in [41] is implemented in TinyOS and resulted in test results up to 40% better than those of FTSP. In an attempt to improve the FTSP, the authors in [42] suggest that their changes to the protocol improves battery life by reducing the number of sent and received frames by 20%. The testing they have done was via simulation on OMNeT++, which is an object-oriented modular discrete event network simulator and not on actual hardware.

P.A Sommer [43] indicated a much deliberated issue that affects the accuracy of the current FTSP implementation. Due to the nature of the synchronization protocol, the nodes in FTSP will always synchronize with the node that is one level up; however, the way the protocol was

implemented currently adds room for error. This error would occur because of a resetting of the linear regression process in the case that the difference between the actual received time and the calculated received time exceeds a certain limit. Since the next node will now not receive a synchronization message from this node that is currently resetting and attempting to re-establish synchronization, it will declare itself as the root and by that degrade the performance of the network. P.A Sommer explains how the current implementation does not account for errors that result from this scenario which have proven to be detrimental to the accuracy of the FTSP. In order to ensure that this error is not exhibited in their implementation, the authors had to fix the root to one specific node and tested the protocol. The average one-hop synchronization error recorded after this modification was 9.04 μs.

F. Wang et al. [44] created the Extensible time synchronization protocol (ETSP) and used FTSP as a benchmark to compare their protocol's performance. The authors implemented both protocols on the SCSC-RFA1 sensor node from Shandong Computer Science Center which has similar features to the Mica2 [45]. With their FTSP implementation, they achieved a time synchronization error that ranged from -2 to 5 ticks whereas they reported an average error of around 1 tick for ETSP for a single-hop network. Since this sensor node uses a 32 kHz oscillator, a tick in this setting was defined as 31.25 μs [45] giving us a reported error of -62.5 to 156.3 μs for the FTSP. These results contradict the precision claims of the original FTSP paper resulting in poor time synchronization performance when tested by F. Wang et al. causing them to rule in favor of ETSP.

Furthermore, a recent paper claims that the FTSP is in fact a low accuracy algorithm that is only useful for short-term applications [46]. This claim contradicts the definition of the FTSP; it has been advertised as a precise synchronization protocol that achieves micro-second range accuracy. However both [46] and [47] challenge that claim. G. Huang et al. [46] report that using the FTSP with a one minute re-synchronization rate achieves a 90 μs error. They then suggest that FTSP is suitable for low accuracy applications such as surveillance whereas the authors of the FTSP paper have indicated that it has been used in a sensor network-based counter-sniper system [28]. There appears to be a huge disconnect between the claims of the authors of the FTSP paper and the current implementation of the protocol: is it or is it not the high-precision protocol that achieves micro-second accuracy? The more well versed in the

time-synchronization community seem to side with the latter on that argument and doubt the accuracy of FTSP whereas others creating their own protocols still use it as the bench mark for time-synchronization.

In the 2017 paper, F. Gong et al. [48] present a new way of measuring the performance of the FTSP while comparing it with a real test-bed FTSP implementation. The protocol is tested on a TelosB hardware unit using the TinyOS version 2.1.2 [3]. As explained in this research, the TinyOS version 2.1.2 implementation does not implement the true function of the flooding time synchronization protocol and therefore their results might not be a true representation of the protocol. L.Li et al. [49] used a different approach to examine the protocol performance, they obtained results using the network simulator software ns3 (version 3.26). This paper presents a new protocol which synchronizes time-stamps from the receiver to a reference time in a reactive fashion called on demand timestamp synchronization framework (OTSF). The creators of OTSF compare the performance of their novel protocol to that of the FTSP through ns3 simulations. Their test results vary with the average sleep interval of the radio, this value ranges from a minimum of 5 s to a maximum of 25 s. Their mean squared error for the synchronization error of the FTSP simulation showed a best case scenario of 110 μs at the minimum average sleep interval of 5 s.

The temperature-compensated Kalman based distributed synchronization protocol (TKDS) is proposed in [50] and tested against FTSP. The authors vary the effective delay in wireless transmission and observe how that changes the reported skew and offset values. J. Wang et al. [51] defined the maximum single-hop delay (effective delay) to be equal to the cycle length of the transmitter. In the case of [50], they set the default value to be 100 μs and offset and skew values for FTSP were measured accordingly. At that effective delay, the reported FTSP skew error was 12 parts per million (which is a unit used to measure the timing accuracy of the crystals in clock oscillators [52]) and an offset of 16 μs. Since the experiment ran for 10 mins or 600 seconds, then the skew would be $\frac{12}{10^6} \times 600 = 7.2$ ms. The summation of both gives us a rather poor result for the Flooding time synchronization protocol. In [53], a modified version of FTSP, FTSP+, is proposed and implemented with TinyOS 2.1.2. The authors aimed to improve upon the modularity of the FTSP protocol by eliminating MAC layer time-stamping and instead performing the time-stamps at the application layer while compensating for the

stack delay. The accuracy of the FTSP+ was then determined by showing the results of their implementation on hardware similar to the Mica2. It was noted that this 2016 paper used the values reported by Maroti et al. in the 2004 paper as a benchmark for comparison of their proposed protocol's synchronization error results.

This argument raises the question: If the results in the original paper that was published in 2004 are not repeatable, how is FTSP still being used as a benchmark for all the new and upcoming synchronization protocols? The fact remains that there isn't a single complete FTSP implementation to be used by others highlighting a need for a fully functional widely available implementation to ensure a fair comparison.

Chapter 3

TinyOS FTSP Implementation

3.1 Current Implementation

The implementation currently being set as the official code for the Flooding Time Synchronization Protocol, which is found on GitHub [3], does not translate the algorithm described by the flooding time synchronization protocol. One of the first issues noticed was the use of a millisecond clock in the current implementation for time-stamping purposes. In addition, the implementation does not perform the multiple time-stamping technique which the FTSP is characterized by. Moreover, seeing as the time-stamps must be taken after each byte transmission or reception, the byte-wise radio must employ its byte-wise radio chip (CC1000 from Texas Instruments). According to TEP 133 (TinyOS Enhancement Proposals number 133) [54], which talks about packet-level time synchronization, the time-stamping approach used is not one intended for the FTSP. The approach they have used is explained below:

- Sender: time-stamp taken at start of transmission and at the end, the delta of the two times is embedded in the message.
- Receiver: a local time-stamp is taken at receiver and the delta that was embedded in the transmitted packet is subtracted from it to obtain the time of the receiver with reference to the transmitter.

Although the previous procedure outlines a time-stamping mechanism, it does not correspond to the one required to carry out the FTSP. To be able to perform this protocol as instructed, the

following values are required:

- Sender: time-stamps of first six bytes sent after the transmission of SYNC bytes
- Receiver: time-stamps of first six bytes received after the reception of SYNC bytes

In the code segments below, the current implementation of FTSP appears to use a single time-stamp and by that confirming that multiple time-stamping does not take place. The programming language used for the FTSP implementation is nesC (nc). NesC is an extension to the C programming language created for event-driven programming on the TinyOs platform [55]. This code is taken from the TimeSyncP.nc file in the tinyos-release/tos/lib/ftsp/ repository found on GitHub [3].

```
1
2  task void sendMsg()
3      {
4          uint32_t localTime, globalTime;
5
6
7          globalTime = localTime = call GlobalTime.getLocalTime();
8          call GlobalTime.local2Global(&globalTime);
9
10         if( numEntries < ENTRY_SEND_LIMIT && outgoingMsg->rootID !=
11         TOS_NODE_ID ){
12             ++heartBeats;
13             state &= ~STATE_SENDING;
14         }
15         else if( call Send.send(AM_BROADCAST_ADDR, &outgoingMsgBuffer,
16         TIMESYNCMSG_LEN, localTime ) != SUCCESS ){
17             state &= ~STATE_SENDING;
18             signal TimeSyncNotify.msg_sent();
19         }
20     }
```

This code segment describes what happens when a message is to be sent. A time-stamp is taken when the call GlobalTime.getLocalTime() is made and that is only done once per packet.

The interface GlobalTime is defined for this protocol and specifically the call mentioned has the function of returning the local time of the mote.

In order to find out if the byte-wise time-stamping was implemented, the interrupt which would be used to trigger the time-stamping event was found in the original code for the CC1000 radio. Specifically, the function below which is found in the file CC1000SendReceiveP.nc seems to do that job:

```
void sendNextByte() {
  call HplCC1000Spi.writeByte(nextTxByte);
  count++;
}
```

HplCC1000Spi.writeByte(nextTxByte) will write the byte ”nextTxByte” to the CC1000 bus thereby allowing us to take a time-stamp at around the same time that happens. After careful research and inspection of the FTSP code available, it was confirmed that this function was never used to create byte-wise time-stamps thereby making it impossible for the protocol to work as specified.

3.1.1 Problem Formulation

Concerns regarding the feasibility of micro-second precision synchronization have been raised on the TinyOS - Help archived online forum [56]. Responses from the authors of the FTSP paper stated that the paper uses an older TinyOS 1.x-based implementation that offers microsecond-precision time-stamping on the Mica2 hardware. However, that implementation has been phased out and in the current implementation [3], there does not exist a Hardware Interface Layer (HIL) component providing micro second granularity. Since the protocol promises microsecond accuracy, it would be impossible to obtain that kind of accuracy using a millisecond clock to record time-stamps. The code snippet below taken from the current FTSP implementation shows that the TMilli clock is the current denomination chosen for the time-stamp:

```
module TestFtspC
{
```

```
uses
{
    interface GlobalTime<TMilli>;
    interface TimeSyncInfo;
    interface Receive;
    interface AMSend;
    interface Packet;
    interface Leds;
    interface PacketTimeStamp<TMilli,uint32_t>;
    interface Boot;
    interface SplitControl as RadioControl;
}
```

The above FTSP test code is taken from GitHub [3]: *tinyos−release/apps/tests/TestFtsp/Ftsp/TestFtspC.nc*. As previously stated, both the GlobalTime interface and the PacketTimestamp interface appear to use a TMilli clock. In addition to that, line 71 in the file TestFTSPC.nc (Appendix A), clearly shows only one timestamp being taken at the receiver: $uint32\_t\ rxTimestamp = callPacketTimeStamp.timestamp(msgPtr)$. As for the sender timestamp, that is embedded in the packet during the transmission of the packet. Current implementation embeds a single timestamp from the sender taken after the SYNC bytes have been sent, however that value is not used at the receiver, the TestFTSP application found online estimates the global time from the local time (received time) using the command: `call GlobalTime.local2Global(&rxTimestamp).` This function is defined in the file *tinyos − release/tos/lib/ftsp/TimeSyncP.nc* as:

```
async command error_t GlobalTime.local2Global(uint32_t *time)
{
    *time += offsetAverage +
    (int32_t)(skew * (int32_t)(*time - localAverage));
    return is_synced();
```

```
}
```

The variables shown in this function only take into account the clock drift calculations when estimating the global time. The CalculateConversion() function will convert the local time to the global time and is called once from within the processMsg() function. ProcessMsg() is a task that is posted when a packet is received (in the Receive.receive function). The code described fails to perform multiple time-stamping thereby eliminating the possibility of carrying out the FTSP as initially defined.

3.2 Complete FTSP Implementation

3.2.1 Specifications and Software Setup

The authors of the flooding time synchronization protocol tested the protocol on the Mica2 wireless mote from crossbow that runs the TinyOS open source platform. The two main components of the mica2 chip are the Atmega128 Atmel micro-processor chip and the Texas Instruments CC1000 byte-wise radio. TinyOS uses the nesC programming language for event-driven component-based programming that is widely used in embedded systems [55]. Furthermore, the code is compiled from within the command line interface using the make tool which is supported by an extensive make system defined for TinyOS which is to be prompted from within the application directory of the desired code. In addition, a specific "tinyos-tool-chain" is needed which contains all the packages used in order to compile the code and program the hardware (also referred to as a mote). This tool-chain is available on the online GitHub [3] repository: tinyos/tinyos-release version 2.1.2.

It was concluded that the best option was to run TinyOS on the Linux kernel since it is the most recently supported kernel. For the purpose of this research a virtual machine was created which ran Linux, Ubuntu 14.04 since it was found that the TinyOS tool-chain worked best on that edition. Avr-dude is the compiler used for this hardware and the packages which make up the tinyos-tool-chain are shown below.

- avr-binutils-tinyos.
- avr-gcc-tinyos.

- avr-libc-tinyos.
- avr-optional-tinyos.
- avr-tinyos-base.
- avrdude-tinyos.

In addition, tinyos-tools and the nesC library must also be downloaded and installed. Finally, since PC-mote communications are java-based, it is vital to install the Java Development Kit for mote programming.

The Mica2 radio is rated for 900 MHz however it has a range of frequencies it can be set to in the make file of each application. It is crucial to set all of the frequencies of the motes to be the same to allow for radio communication through the following command: `CFLAGS += -DCC1K_DEF_FREQ=900000000`. Since this application depends on radio communications from specific nodes, it is important to set the node ID of the mote that can be done at the time of programming. The MIB510 programming board is used to program the Mica2 through the command `make install.0x0002 mica2 mib510,dev/ttyUSB0` where 0x0002 is the nodeID and /dev/ttyUSB0 is the port through which the programmer is connected to. Prior to downloading the program on the mote, it must first be compiled offline using the "make mica2" command and manually debugged. Testing and debugging are done online using the hardware since there isn't an integrated development environment for the Mica2 hardware.

3.2.2 Mica2 Hardware Calibration

Due to the hardware limitations of the Mica2, this step was added to obtain micro-second accuracy for the synchronization error values. More details on the nature of this limitation are described in the next section under implementation difficulties. The values that need to be re-calculated for each Mica2 unit are the byte transmission time (the time it takes to transmit a byte) and the bit offset time. In order to obtain these values the following procedure was carried out:

- Record the time-stamps taken after the sync bytes have been sent/received and calculate the difference between each consecutive time-stamp, this will be the byte transmission

time taken for further calculations. The byte transmission time is calculated for the sender and the receiver as slight differences were noticed that might affect data accuracy.

- The calculated byte transmission time is divided by 8 to get a single bit transmission time. This value is used to calculate the bit offset time from the bit offset at the receiver. For a bit offset of 0 the bit offset time is 0 μs, for a bit offset of 1 the bit offset time is 1$\times$ (single bit transmission time), for a bit offset of 2 the bit offset time is 2$\times$ (single bit transmission time) and so on until a bit offset of 7. This calculation will populate the bit offset correction array with the new values that better match the clock of the Mica2 hardware being used. These newly calculated constants will be the values used for the byte alignment calculations.

3.2.3 Description of Implementation Code

The Flooding Time Synchronization promised to minimize interrupt and software jitter through its novel multiple time-stamping technique. This research and investigation confirmed that this technique has not been part of the current online code repository containing the implementation of the protocol. Consequently, the implementation presented in the following sections alters the available code by taking into account the FTSP algorithm and the shortcomings described earlier to perform the desired outcome. The base files used for this are listed below:

- `TestFtspC.nc`.
- `CC1000SendReceiveP.nc`.
- `CC1000CsmaRadioC.nc`.

The files below were created in order to utilize the micro-second clock:

- `TimeSyncMicroC.nc`.
- `MuxAlarmMicro32_.nc`.
- `MuxAlarmMicro32.nc`.
- `MuxAlarmMicro16_.nc`.

- MuxAlarmMicro16.nc.
- AlarmCounterMicroP.nc.
- HilTimerMicroC.nc.
- TimerMicroC.nc.
- TimerMicroP.nc.

Micro-Second Clock

Given that the HIL component providing TMicro time does not exist, the 32 kHz code was used as a base for a TMicro implementation. In addition to changing the wiring to use the newly created TMicro HIL component, all the precision tags used for the interfaces which are part of the time-stamping process were changed to [TMicro].

Starting with the TestFtspC.nc file, the precision tags in the GlobalTime interface and PacketTimeStamp interfaces were changed from from TMilli to TMicro. In addition, similar changes had to be done in the CC1000SendReceiveP.nc file which describes the logic behind the sending and receiving functions of the CC1000 radio. Currently both the PacketTimeStamp and LocalTime interfaces in CC1000SendReceiveP.nc are defined for TMilli and T32khz but not TMicro. In order to use TMicro precision for those interfaces they must be defined in the CC1000SendReceiveP.nc file. The following definitions were developed:

- `async command bool PacketTimeStampMicro.isValid(message_t* msg).` This function will return a boolean value to indicate whether the timestamp of the message (msg) is valid or not by performing a check. The same check is done on all denominations (TMilli, T32kHz, TMicro) since the over the air value is always 32 kHz.
- `async command uint32_t PacketTimeStampMicro.timestamp(message_t* msg).` Since the value of the timestamp is always in the 32khz precision, in order to translate it to Micro it must be shifted. In the approach specified, this timestamp will calculate the offset in 32 kHz, shift it by 5 (offset >> 5) and add it to the current local time in TMicro.

- `async command void PacketTimeStampMicro.clear(message_t* msg)`
  Removes current Time-stamp.

- `async command void PacketTimeStampMicro.set(message_t* msg, uint32_t value)`
  Using the same mechanism of calculating the offset of the timestamp and adding it to the local time, the setting is done the same way in TMicro then translated to 32 kHz before being inserted in the metadata of the packet.

The configuration file CC1000CsmaRadioC.nc was also changed by providing the wiring for the PacketTimeStampMicro interface to the CC1000SendReceiveP.nc. In addition, the TimeSyncMicroC.nc file had to be created in order to be able to use the defined FTSP TimeSyncP component with the micro-second granularity. TimeSyncC.nc was used as a template and the precision was changed to TMicro for all.

However, after careful examination of the ftsp code found in tos/lib/ftsp, changing the precision of the current time-stamping code still does not reflect the flooding time synchronization protocol algorithm therefore the packet time-stamping has to be done from scratch. This issue is explained further in the multiple time-stamping code section. Nevertheless, obtaining the Local Time in TMicro is needed in order to carry on with the time-stamping. That step is done by creating the following components:

```
components CounterMicro32C, new CounterToLocalTimeC(TMicro) as
CounterToLocalTimeMicroC;
```

After which they are then wired to give us LocalTimeMicro for the CC1000SendReceiveP.nc file:

```
CounterToLocalTimeMicroC.Counter -> CounterMicro32C;
SendReceive.LocalTimeMicro -> CounterToLocalTimeMicroC;
```

The HilTimerMicroC file created describes a microsecond timer for the Mica2 that is built upon a hardware timer. The interfaces provided are `Init`, `Timer<TMicro> as TimerMicro[uint8_t num]` and `LocalTime<TMicro>`. The implementation is shown below:

```
implementation {
```

```
2
3   enum {
4     TIMER_COUNT = uniqueCount(UQ_TIMER_MICRO)
5   };
6
7   components AlarmCounterMicroP, new AlarmToTimerC(TMicro),
8     new VirtualizeTimerC(TMicro, TIMER_COUNT),
9     new CounterToLocalTimeC(TMicro);
10
11  Init = AlarmCounterMicroP;
12
13  TimerMicro = VirtualizeTimerC;
14  VirtualizeTimerC.TimerFrom -> AlarmToTimerC;
15  AlarmToTimerC.Alarm -> AlarmCounterMicroP;
16
17  LocalTime = CounterToLocalTimeC;
18  CounterToLocalTimeC.Counter -> AlarmCounterMicroP;
19 }
```

TEP 102 [54] describes how the timers are chosen, it states that "a new timer is allocated using unique(*UQ_TIMER_MILLI*) to obtain a new unique timer number," has been changed to *UQ_TIMER_MICRO* for this study. They explained how the timer will then be "used to index the TimerMilli parameterised interface." In the case of this thesis, it is the TimerMicro interface. The header file Timer.h defines *UQ_TIMER_MICRO* which has been used in the newly created TimerMicroC and HilTimerMicroC files. TimerMicroC.nc and TimerMicroP.nc files were both created.

Since the HilTimerMicroC depends on an AlarmCounterMicroP component, it had to be created. The AlarmCounterMicroP.nc file configures the hardware timer for use as the Mica2's microsecond timer with the AlarmCounterMicroC.nc being the wiring file created for it.

Finally, a TimerMicroP.nc file was created outlining the configuration of virtualized microsecond timers which auto-wire the timer implementation (TimerC) to the boot sequence and export the various Timer interfaces. TimerMicroC.nc was then added to create the abstraction for this timer in the form of a component which is instantiated in order to give an independent microsecond granularity timer as per the recommendations of TEP 102 [54].

Multiple Time-stamping Code

The multiple time-stamping is done on a byte-by-byte basis at the sender and receiver in the CC1000SendReceiveP.nc file. The function used to obtain the time is the LocalTimeMicro.get() that was discussed earlier. A counter was created in order to hold the number of bytes sent after the sync bytes have been sent, lets call that txcount. Similarly the receiver also had a counter which held the number of bytes received after the sync byte has been received, this is labelled rxcount. These respective tx/rx counters are set after the SYNC byte is transmitted/received and the number of bytes sent after the SYNC byte is then calculated using a difference of the local rx/tx count (which remains constant throughout the packet) and the packet count (which is incremented after each byte transmission or reception throughout the packet). Both the packet counter and the local tx/rx counters are reset for a new packet. Using this difference, the first six bytes that are sent/received are time-stamped and their time-stamps are recorded. The values of the time-stamps are then used in order to obtain t′, which as explained in Equation (2.3), is the minimum of the last received timestamp minus the byte transmission time and the current received timestamp. The byte transmission time for the hardware was calculated using the procedure detailed in Section 3.2.2, this produced a value of 378 ticks for the sender and 384 for the receiver. The time-stamps are recorded first and then compared to each other starting with the last timestamp until the first one with the results of that comparison being held in a new variable. The average of these minimized normalized time-stamps is then taken and embedded into the sender/receiver before the packet is finished transmitting/receiving. The pseudo code used for the data transmission is shown below:

```
1  void txData() {
2
3      sendNextByte();
4
5      if (nextTxByte == SYNC_BYTE2) {
6        // SYNC_WORD has just been sent
7      txCount = count;
8    }
9
10 if ((count - txCount) <  6){
```

```
11    t[(count - txCount)] = call LocalTimeMicro.get();
12  }
13  if ((count - txCount) ==  6){
14
15    tp[5]=t[5];
16
17    for (i = 4; i >= 0; i--) {
18          tp[i] = min((tp[i+1]-delta),t[i]);
19          total = total + tp[i];
20    }
21    total = total +tp[5];
22
23      avg = (total/6);
24   }
25  }
```

The time-stamping code at the receiver follows the same algorithm of the transmitter with the exception of the setting of time-stamp values after the calculations are done. The transmitter will embed the timestamp in the radio count message and the receiver will set its local calculated time-stamp in the metadata of the message using `getMetadata(rxBufPtr)->timestamp = Received Timestamp`. Since the location of the calculated timestamp is at a lower level (happening in CC1000SendReceiveP.nc), the higher level application file (TestFtspC.nc) is then able to access this value through the function call:
PacketTimeStamp.timestamp(*msgPtr*). The current setup of the applications has RadioCountToLeds being programmed as the sender (root) node and TestFtsp being the receiver node (which will be synchronizing itself to the root). Therefore the txData function will differ from the sender to receiver. The receiving node will have the job of calculating the received multiple time-stamps, combining them into one and also performing the clock drift calculations on the past 8 packet Time-stamps in order to correct its own "Local Time" while the sender keeps its own time as the "Global Time". The receiver will then relay all of the data in a radio message to a Base Station node which is connected to the PC via UART.

Clock-Drift Code

The current clock drift code that is available is implemented in the TimeSyncP file. A version of that implementation was programmed on the receiver node. The same logic that was used at the original FTSP implementation was followed and implemented in the TestFtspC.nc application file. The packet-level time-stamps were recorded and a counter kept track of the packets received in order to perform a linear regression on the last 8 time-stamps received. At each sending interval the skew is recalculated and applied to the local time to obtain a global time estimate. The estimated global time is then taken by performing the following operation using the calculated skew value:

$$localtime = localtime + offsetAverage + skew(localtime - localAverage) \tag{3.1}$$

The localtime is the receivers time at the reception, the offsetAverage is the average of the difference between the global and local times and the skew is the slope calculated with linear regression as explained in Equation (2.6).

Byte Alignment Code

The available implementation of the byte alignment correction has the following values:

```
From the file CC1000SendReceiveP.nc:
....
#ifdef PLATFORM_MICA2
  // estimated calibration, 19.2 Kbps data, Manchester Encoding,
  // time in jiffies (32768 Hz)
  static const int8_t BIT_CORRECTION[8] = { 27, 28, 30, 32, 34,
  36, 38, 40 };
....
```

This array is then used to obtain the delay in jiffies, as stated in the comment above the array declaration. Each jiffy is equal to about 30.5 μs. Table 3.1 shows a translation of these numbers for each bit offset. Comparing the values from Table 3.1 with those in Table 2.4, many discrepancies are noted and therefore do not follow the concept that Maroti et al. [2] described in their paper. For this implementation, the hardware calibration was done to get

Table 3.1: Current FTSP Code Byte Alignment Time Delay

Bit Offset	Corresponding Time in μs
0	824.0
1	854.0
2	915.0
3	976.0
4	1037.0
5	1098.0
6	1159.0
7	1220.0

the byte transmission time and ultimately the delay corresponding to each bit offset. Hardware calibration of the Mica2 used in this study produced the following bit correction array:

```
....
#ifdef PLATFORM_MICA2
  // estimated calibration
  // time in ticks (1 tick = 0.95 microsecond)
  static const int8_t BIT_CORRECTION[8] = { 0, 48, 96, 144, 192,
  240, 288, 336};
....
```

3.2.4 Implementation Difficulties

Due to the nature of the wireless medium, packet loss was a problem that was visible in the testing and results. In order to ensure that packet loss was minimized, acknowledgments were used to ensure reliability. This function returns a boolean true or false if an acknowledgement was received after which it was programmed to re-transmit if it had failed. In addition, the way the code was setup only allowed a message to be sent after it has been received and from within the receiving function. The Isforme() function was also used at the receiver to ensure that the node is not bombarded with any other packets which could have otherwise caused it to drop the packets addressed to the node. The NodeID of each mote is specified at the time

of the program download on the hardware and this number was then hard-coded in the sending function. In addition, the interval that the packets were sent was increased from the 250 ms in the original RadioCounttoLeds file to 1 s. Applying the previous changes helped reduce the packet loss to 0%.

When reporting the averaged and minimized time-stamps at the sender and receiver, the overflow issue was faced. Keeping in mind that the time-stamps are `uint_32` types, adding six of them may cause an overflow in the "total" variable which holds the sum of the time-stamps. Overflow will cause the value of the variable to default to an unknown number giving inaccurate readings. When trying to overcome this problem, one of the solutions was to change the size of the total variable to `uint_64` while keeping the average variable as 32-bit number. However, doing so did not work with the code as the radio was unable to handle a variable of that size even when casting the final time-stamp to type `uint_32`.

A workaround that was attempted was having a 64-bit total and choosing the lowest 32 bits of the average then setting those to the final time-stamp `uint_32` type variable instead of type-casting it. It seemed that it was a size issue, when programming the RadioCounttoLEDs mote having more than one 64 bit variable was too much for it to handle. Consequently, in order to be able to keep all the variables a `uint_32` type, a certain offset was subtracted from the time-stamps if an overflow was suspected. In order to know if an overflow is suspected a check is done after collecting all six time-stamps on the last timestamp and the overflow flag is updated to true. See the updated pseudo code below for more details:

```
1  void txData() {
2
3      sendNextByte();
4
5      if (nextTxByte == SYNC_BYTE2) {
6        // SYNC_WORD has just been sent
7      myCount = count;
8    }
9
10 if ((count - myCount) <  6){
11   t[(count - myCount)] = call LocalTimeMicro.get();
12 }
```

```
13 if ((count - myCount) ==  6){
14
15   tp[5]=t[5];
16
17 if (tp[5] > y){
18 //where y is the maximum value a uint_32 variable may hold in this case
19 //to prevent overflow during the summation of the total time-stamps
20   overflow = TRUE;
21   set tp[5] = tp[5] - x;
22   //where x is a known value that will be added back after the average
23   //is calculated to prevent overflow
24 }
25 else {
26   overflow = FALSE;
27 }
28   for (i = 4; i >= 0; i--) {
29     if (overflow == TRUE) {
30         tp[i] = min((tp[i+1]-delta),(t[i]- x));
31         total = total + tp[i];
32         }
33     else if (overflow == FALSE) {
34         tp[i] = min((tp[i+1]-delta),t[i]);
35         total = total + tp[i];
36     }
37   }
38   total = total +tp[5];
39
40   if (overflow == TRUE){
41     avg = (total/6) + x;
42   }
43   else if (overflow == FALSE){
44     avg = (total/6);
45   }
46  }
47 }
```

Another problem faced was visible in the synchronization error that was calculated. The

data obtained displayed a periodic delay of unknown origin every 256 packets that disrupted the results and caused inaccuracies. Initially, it was thought to be a delay caused by the atomic sections in the code disabling interrupts. After further investigation, this delay appeared to be the result of the residual function of the pre-existing code. The reason behind this error was found out to be caused by the beaconing rate of FTSP which was set by the `PFLAGS+=-DTIMESYNC_RATE=3` command in the Makefile.

Finally, a more complicated timing issue was discovered much later in the study due to its finer effect on the results. This problem was one that required expert in depth hardware understanding of the Mica2 hardware which included the ATmega128L and the CC1000 data-sheets [31, 57]. Initially the problem started by measuring the "byte transmission time" of the Mica2 device. In [32], Maroti indicates that the Mica2 byte time is 417 μs, however, when measured, the data indicated a much lower value. This discovery was made by investigating the six time-stamps taken at the sender and receiver prior to computing the averaged single time-stamp. The time between each consecutive time-stamp, which corresponds to the time after a byte of data is sent or received, was calculated to be around 365 μs. A constant difference of 52 μs indicated that the number specified by the creator of the FTSP as around 417 μs was merely just one obtained specifically for their own hardware and is not universal to all. This discovery adds to the argument that the results shown by the authors of FTSP paper are not repeatable since they were tailor-made for a specific hardware unit. Even more so, research into the nature of the TinyOS platform revealed that there exists a hardware limitation in the Mica2 device. TEP 102 [54] states that the Mica2 hardware running at 7.37 MHz is unable to produce an exact binary micro-second timer and so the closest it can achieve is an accuracy of 7.37MHz/8. This sparked an interest in discovering why that is the case and so more research in the code provided, CC1000 radio data-sheet and ATmega128L micro-processor data-sheet was carried out. The outcome indicated that the Mica2's external crystal is rated at 7.3728 MHz, however, the micro-processor was programmed to run at 8 MHz with their reasoning being "rather than introduce a plethora of precisions, we believe it is often best to pick the existing precision closest to what can be provided, along with appropriate documentation" taken from TEP 102 [54]. The authors therefore have shown that only up to a 92% accuracy was possible with the current timers and alarms that TinyOS employs that directly affects the results obtained since

FTSP claims micro-second accuracy. The former statement indicates that the use of TinyOS to obtain micro-second accuracy is impossible unless certain adjustments were made to the results based on each specific hardware. This limitation is the reason for the extra calibration step added in the implementation. Therefore, in order for the FTSP to provide micro-second accuracy one must not simply just follow the steps in the original paper, rather, use the method outlined in this study to ensure that the Mica2 is properly calibrated.

Implementing the FTSP as described in the original paper without calibration resulted in an average synchronization error of around 13 μs. Applying the modifications described in this thesis reduced the error to about 1.4 μs clustered about a mean of 5 μs. In order to realize single micro-second accuracy precision, a few constants had to be re-calculated from the raw data obtained from the Mica2 hardware used. Using the hardware calibration procedure, the byte transmission time was recalculated to be 365 μs and the bit offset time ranged from 0 μs for a bit offset of 0 to 320 μs for a bit offset of 7.

Chapter 4

Protocol Testing and Results

4.1 Test-bed setup

In this section both the original test- carried out by Maroti et al. and the tests done in this study are presented. The results obtained from the tests are then compared to to those reported in the original FTSP paper.

4.1.1 FTSP original test-bed

In order to ensure that the protocol is performing as predicted, accurate and controlled testing is required. Before creating s test setup, the original FTSP paper test-bed was first examined. The parameters used and results obtained by Maroti et al. are summarized in Table 4.1.

As mentioned in Table 4.1, Maroti et al. [2] performed an offline linear regression in order to compensate for the clock offset and skew. They have included a histogram of their error values and it showed that the majority of their error values were within the 1 μs range, see Figure 4.9. They then used this offline regression as a base for their online implementation. The FTSP paper indicated that the sender node was programmed to communicate its local time to the receiver node as the global time, the receiver then computes the deviation of the received value (global time) from its own local time at reception as the offset. The skew is then calculated as the sum of the offsets divided by the sum of the local times. Due to memory constraints of the Mica2 hardware, the authors programmed the receiver to perform a linear

Table 4.1: Original FTSP paper test parameters and results [2]

	Original FTSP paper
Hardware	4 units of Mica2 motes
Total experiment time	10 minutes
Packet sending period	5 seconds
Distance between nodes	Insignificant as the propagation delay is less than 1μs for up to 300 meters
Calculated values	Clock offset and skew calculated offline using linear regression
Average synchronization error calculation	The absolute value of the difference between the global and local time-stamps
Reported average synchronization error	1.4 μs for offline Linear regression 1.48 μs for online Linear regression
Reported maximum synchronization error	4.2 μs for offline Linear regression 6.48 μs for online Linear regression

regression on the data from the last received 8 packets. At a synchronization interval of 30 s, they obtained an average absolute error of 1.48 μs and a maximum absolute error of 6.48 μs. This thesis implements and tests the FTSP in order to reproduce the results reported by Maroti et al [2].

4.1.2 Recreated FTSP test-bed

The FTSP test application that is currently available on the online GitHub repository does not provide an accurate test scenario for the protocol as previously discussed. The TestFtspC.nc and CC1000SendReceiveP.nc files had to be modified to test the new implementation. Due to limited resources, only 3 Mica2 nodes could be procured for the experiment. The sender node was programmed with the RadioCounttoLedsC.nc app as the readme.txt file that the FTSP test scenario suggests. However, the multiple time-stamping was implemented on a much lower level, namely in the CC1000SendReceiveP.nc file, in order to get byte-wise granularity. The

test conducted in this study is programmed so that the sender node will perform the multiple time-stamping and embed its averaged, normalized time into the message. As this message is

Table 4.2: Original vs recreated FTSP test parameters and results

	Original FTSP paper	**New FTSP test-bed**
Hardware	4 units of Mica2 motes	3 units of Mica2 motes
Total experiment time	10 minutes	10 + minutes
Packet sending period	5 seconds	1 second, 5 seconds
Distance between nodes	Insignificant as the propagation delay is less than 1μs for up to 300 meters	
Calculated values	Clock offset and skew calculated with linear regression both offline and online using the last 8 data points	
Average synchronization error calculation	The average of the absolute value of the difference between the global and local time-stamps	
Average absolute synchronization error with offline linear regression	1.4 μs	6.5 μs
Average absolute synchronization error with online linear regression	1.48 μs	4.4 μs
Maximum synchronization error with offline linear regression	4.2 μs	43 μs
Maximum synchronization error with online linear regression	6.48 μs	9 μs

being received, the receiver node will perform multiple time-stamping at the byte-wise level and apply the byte-alignment correction then report its own averaged normalized local time-stamp in addition to the global time-stamp received. The receiver, which was programmed with TestFtspC.nc already has code which will extract the information from the received message. This code is used to get the global time. The receiver was then programmed to perform an online linear regression on the last 8 data points and correct the local time accordingly. This is

in line with the test that the authors of the FTSP paper carried out in order to show the effect of computing the linear regression online. However due to limited resources, it was not possible to run the test for 18 hours as the original paper had suggested. Instead, the same sending period was used and tested for about 10-15 minutes to be consistent with the rest of the tests. Table 4.2 shows a comparison of the test done in this thesis and the original FTSP test. Initially, the sending period was set to 1 second, however, in order to recreate the results as accurately as possible, a 5 second sending interval was chosen.

4.1.3 Test Results

In this section the progression of the effect that the error reducing techniques have on the data is displayed. Figure 4.1 visualizes the four main sources of error that affected the data.

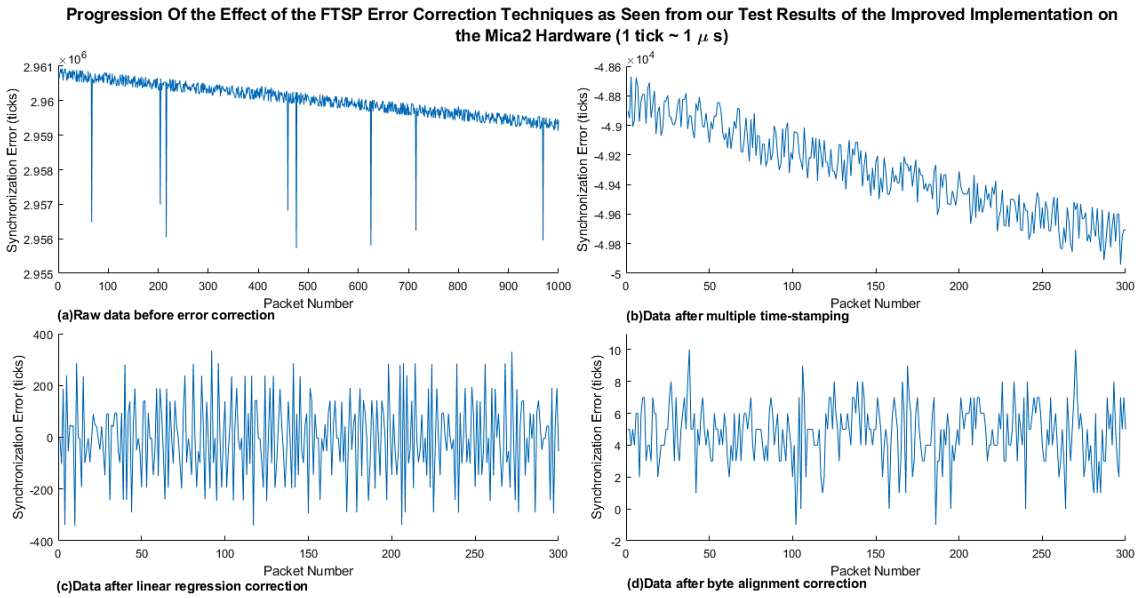

Figure 4.1: (a) The raw data with no error correction, (b) Multiple time-stamping, normalization and minimization to remove interrupt and encoding/decoding errors, (c) Linear regression to remove clock drift errors, (d) Bit offset delay at the receiver to compensate for byte alignment errors.

Raw data

In this section the difference between the sender time and the receiver time is displayed. Figure 4.2 shows the major errors that the FTSP aims to eliminate. The first error is represented by the random large dips which correspond to interrupt jitter. In addition, the slow effect of the clock drift caused an error that impacts the output with a negative slope. Bit offset errors are not quite visible in Figure 4.2, but can be seen clearly in Figures 4.4 and 4.6. This error is responsible for the high frequency errors that the plot exhibits.

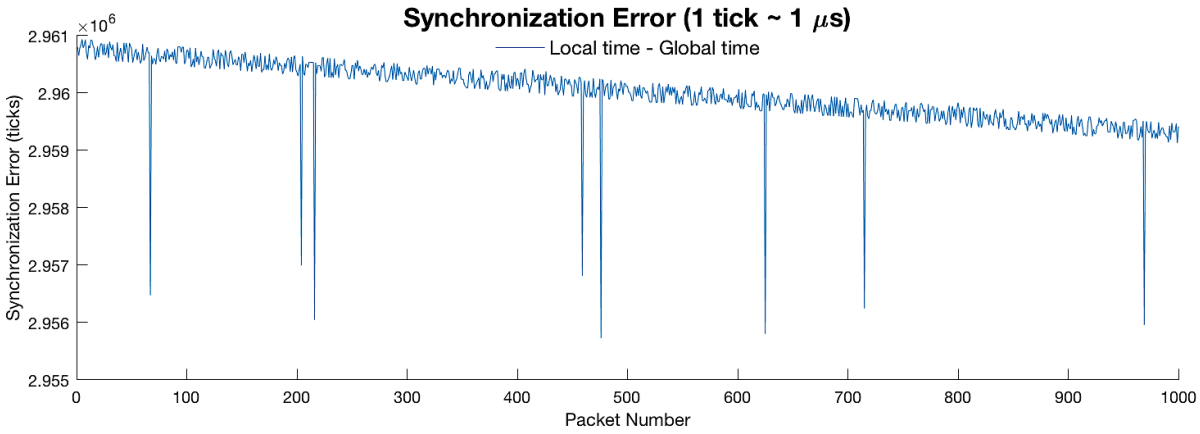

Figure 4.2: Raw data without error correction, synchronization error is the difference between the sender and receiver times

Data after multiple time-stamping

In this section the multiple time-stamping technique is applied and its effects are explained. Figure 4.3 shows that the large errors have now disappeared as predicted. This behavior is expected since the multiple time-stamping technique uses averaging to get rid of the outliers. Further correction which included the normalization and minimization have also helped remove any encoding/decoding errors. However, the slow impact of the clock drift is visible from the negative slope that the data displays.

Data after multiple time-stamping and linear regression

In this section, the effects of both the multiple time-stamping and linear regression on the data are studied. Figure 4.4 indicates that the data is now free from the negative slope that was

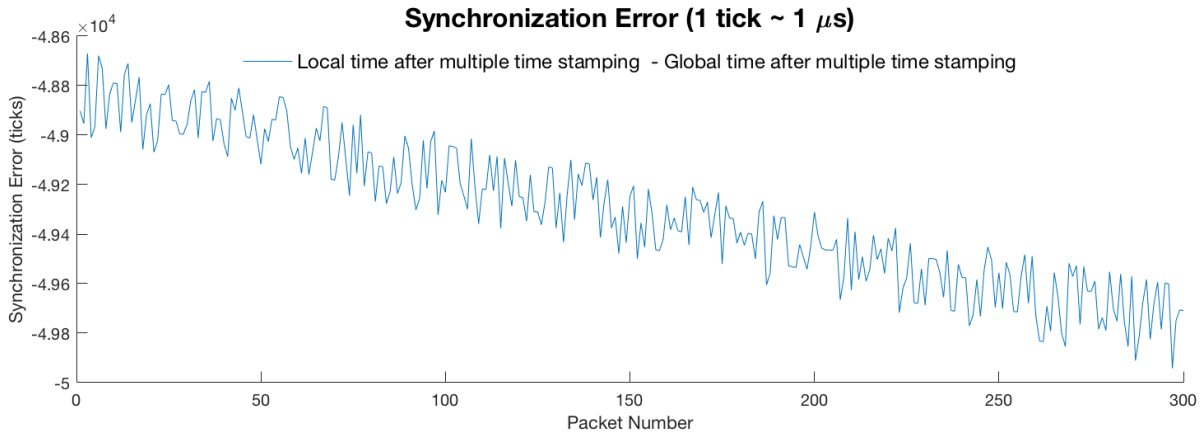

Figure 4.3: Data after multiple time-stamping, offset error is the difference between the sender's corrected time (Global time) and the receivers corrected time (Local time) both after multiple time-stamping

brought about from the clock drift through linear regression correction - least squares method. The range of the sync error is shown to be around -300 < sync error < 300 μs in Figure 4.4. This range is still unacceptable for the FTSP and so the byte alignment correction must be applied. Due to the nature of online linear regression, the initial few minutes of the data appear to fluctuate before stabilizing. In order for online linear regression to be effective it would take a few cycles for the hardware to secure enough data to perform the linear regression properly. This initial inaccurate data is shown in the first part of the graph in Figure 4.5, however, it quickly corrects itself after a few cycles as seen in the plot. The initial adjustment period is removed for data analysis purposes and presented in Figure 4.4.

Data after multiple time-stamping, linear regression and byte alignment correction

The plot in Figure 4.6 shows the dramatic decrease that the byte alignment correction implementation had on the data. Using the calibration step in this study, the synchronization error was reduced to a mean of 4.5 ticks with a standard deviation of 1.9 ticks, 1 tick is approximately 1 μs. The average maximum error calculated is 9 ticks throughout the 10 tests carried out. All the results are available in Tables 4.3 and 4.4 in the next section.

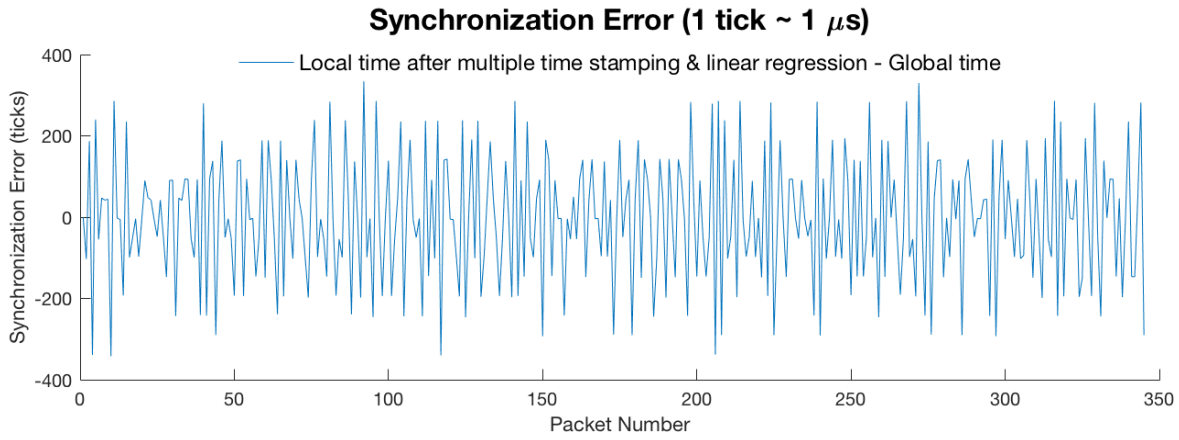

Figure 4.4: Data after multiple time-stamping and linear regression, offset error is the difference between the Global time (sender) and the Estimated Global time which is the local time after correction (receiver)

4.1.4 Result Analysis

In an attempt to keep the tests as similar as possible, the linear regression was implemented online as suggested by the second test in the original paper. An offline regression for the same values was also calculated. Testing was carried out multiple times and the data from 10 tests was recorded to ensure reliability, Tables 4.3 and 4.4 detail the results obtained. The nature of the data resulting from the tests conducted is studied and a histogram for both the offline and online linear regression implementations has been created. This result is also compared to the available plot from the original FTSP paper. Figure 4.7 represents the online linear regression implementation histogram distribution. The results show that the statistical mean is offset by around 4.5 ticks, however, the standard deviation shows that most of the values are within 1.9 ticks of the mean. If this offset is compensated for, a synchronization error of 1.3 μs can be obtained with the implementation procedure outlined in this thesis.

Furthermore, the results from this study show a rather interesting phenomena in terms of the linear regression calculation. The results of the online linear regression performed much better than those calculated with an offline linear regression. A visualization of both results is shown in Figure 4.8. Although this is not the typical outcome one would expect, it agrees with the argument made throughout this study. More specifically, due to the aforementioned

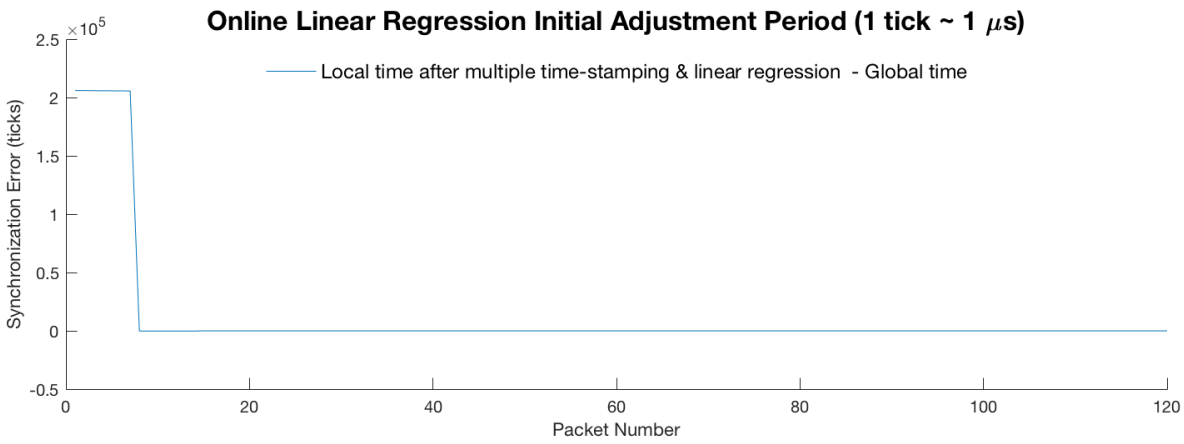

Figure 4.5: Synchronization error behavior during the online linear regression initial adjustment period.

hardware clock limitations described in detail in Chapter 3 and then compensated for with the extra calibration step, it is likely that the offline computation would perform rather poorly. The instability of the hardware clocks is better compensated for with an online linear regression that continuously recalculates the skew every 8 packets. In comparison, an offline linear regression assumes that the clocks are stable throughout the experiment, this assumption does not hold in the case of the Mica2 hardware.

The flooding time synchronization protocol implementation currently available does not take into account the hardware limitations of the Mica2 (device it was tested on) but rather, the authors present their results as ideal without having to do any extra steps and crediting the success to the merits of the FTSP protocol. In reality, the resulting synchronization error obtained without the calibration step that has been added was 13 μs with a very large standard deviation. After performing the Mica2 calibration, the average synchronization error was reduced to a mean of 4.3 μs with a standard deviation of 1.8 μs. The results show that the hardware limitations imposed by the Mica2 hardware can be taken care of with the proper procedure. On the contrary, the available implementation exhibits non-repeatability of the results reported by the authors of the FTSP, in addition, no insight is offered as to how they have achieved a synchronization accuracy of 1.4 μs. To summarize, this implementation produced results which closely match those produced in the original FTSP paper while at the same time being easily reproducible.

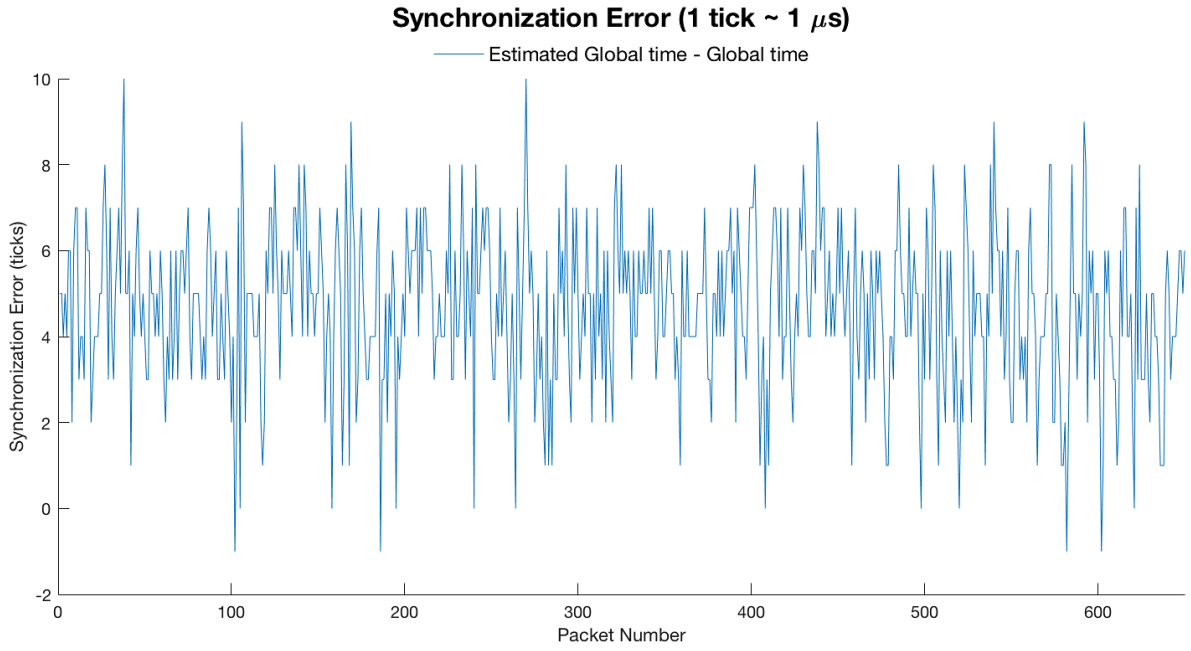

Figure 4.6: Data after multiple time-stamping, linear regression and byte alignment correction, offset error is the difference between the Global time (sender) and the Estimated Global time which is the local time after correction (receiver).

Table 4.3: FTSP implementation test results in ticks (1 tick = 0.95 μs)

	Test 1	**Test 2**	**Test 3**	**Test 4**	**Test 5**	**Test 6**	**Test 7**	**Test 8**	**Test 9**	**Test 10**
Mean	4.7	4.9	4.6	4.8	4.7	4.3	4.3	2.4	5.1	5.5
Std Dev	1.7	1.9	1.9	1.6	1.8	2.1	2.0	2.3	1.7	1.8
Min Error	-1	-1	0	0	0	-1	0	-4	0	1
Max Error	10	9	10	8	9	9	9	7	9	10

Table 4.4: FTSP implementation average test results in ticks (1 tick = 0.95 μs)

	Average Synchronization Error
Mean Error	4.5
Standard Deviation	1.9
Minimum Error	-0.6
Maximum Error	9

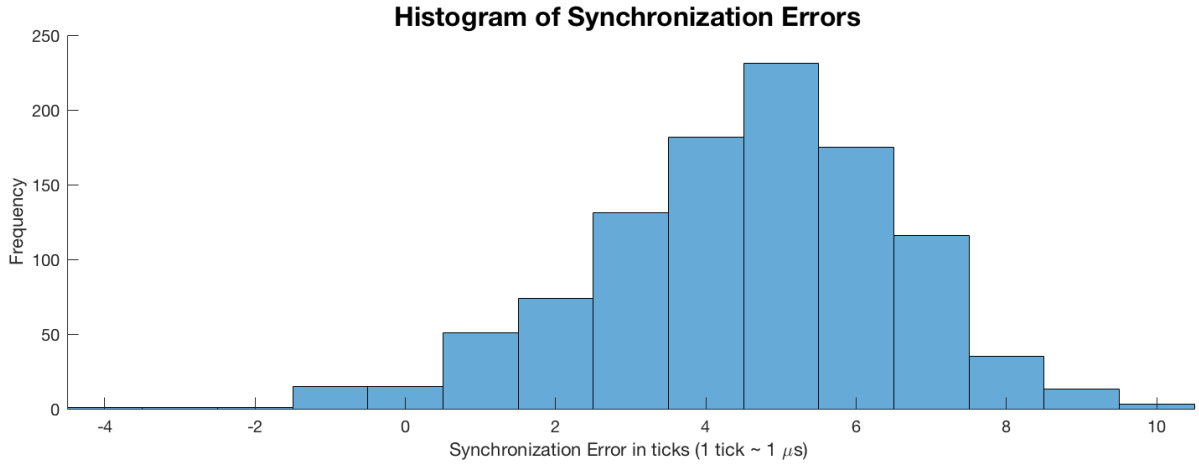

Figure 4.7: Histogram of synchronization errors after online linear regression, mean = 4.4 μs, standard deviation = 1.8 μs.

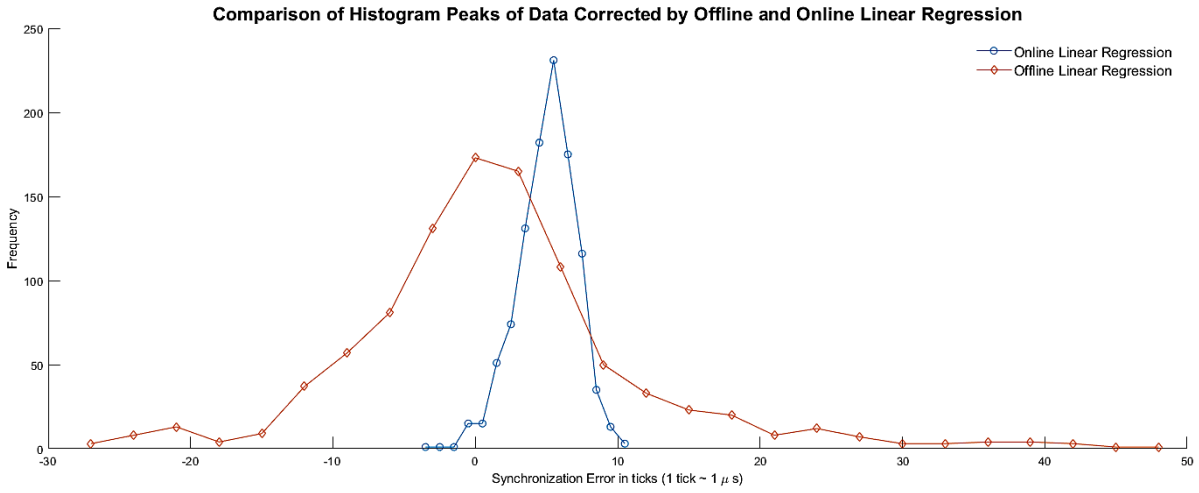

Figure 4.8: Comparison of synchronization errors between online and offline linear regression, the offline linear regression produced the following results: mean = 0 μs, standard deviation = 9.4 μs.

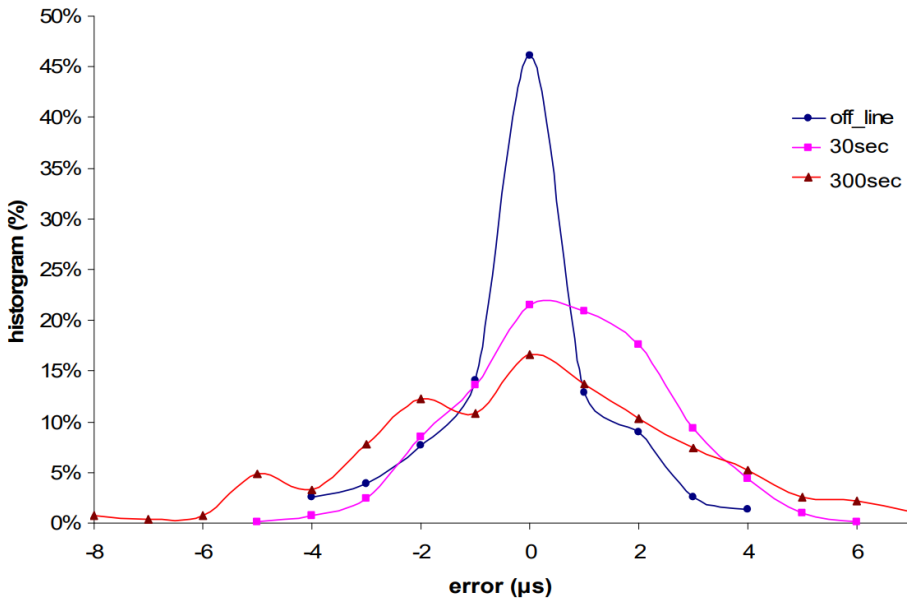

Figure 4.9: The FTSP original paper plot showing the error distribution of the offline linear regression (offline LR) and online linear regression (30s and 300s) both of which refer to the test's time synchronization interval [2].

Chapter 5

Conclusion

Synchronization protocols in wireless sensor networks must take into account the unpredictable nature of the wireless medium and provide accurate timing information while being reliable at the same time. Maroti et al. [2] indicated that the main sources of time synchronization error stemmed from software jitter resulting from the disabling of interrupts and encoding/decoding errors. To further increase the accuracy of the protocol, they compensated for the clock drift phenomenon that affects the local clocks of the hardware and also corrected the byte alignment at the receiver nodes. The tests they conducted were on 4 units of the Mica2 hardware nodes and produced an absolute average synchronization error of 1.4 μs. After a thorough study of the methods used for error correcting that the FTSP employs, this study implements the protocol in an attempt to recreate the reported results. The multiple time-stamping technique was implemented to compensate for interrupt jitter and encoding/decoding errors. This technique takes six time-stamps and normalizes them by subtracting the time it takes to transmit a byte and minimizing these values, an average is then taken and a single time-stamp is reported. At the receiver end, the bit offset is calculated and the received time-stamps are further corrected by adjusting the byte-alignment according to the bit offset calculated. For the clock drift, online linear regression was performed on the last 8 data points in order to obtain the clock skew which was taken as the slope calculated. The skew is then used as a multiplier to calculate the estimated global time at the receiver from the local time and the time offset. The previous techniques have been implemented as previously stated in this study, however, extra calibration steps had to be done in order to obtain results similar to those reported in the original FTSP

paper.

FTSP has been labeled as the "gold standard" of synchronization protocols by many; however, there hasn't been an implementation which produces an accuracy within the micro-second range until now. The available FTSP implementation lacks the main attribute which the protocol heavily advertises: its novel multiple time-stamping technique. Other major errors with the implementation included incorrect byte alignment code and their use of a milli-second clock to obtain values which were supposed to be within micro-second accuracy. Finally, the current FTSP implementation did not indicate nor compensate for the hardware limitation imposed by the Mica2. Therefore, the absence of proper implementation documentation for the FTSP motivated this study to provide a complete protocol implementation with easily reproducible results. Testing of the new implementation was carried out on 3 units of the Mica2 hardware and the global and local times were calculated and reported to the base-station node that ran a java application to display the results in real-time on the laptop. Data was then taken and statistical operations were carried out on it to better understand the nature of the output. As indicated previously, testing for the FTSP was done many times and the results of 10 of these tests (Tables 4.3, 4.4) were taken as the main data set. At first it was noticed how the error changed with each correction technique being applied to the data and ensured that the results were in line with theoretical predictions.

The averaging and minimization of time-stamps eliminated the large errors as would be expected from that technique. The added calibration procedure coupled with the byte-alignment process served to decrease the magnitude of the high frequency errors as seen in Figure 4.6, this effect is the result of bit offset correction at the receiver. Finally, the linear regression that was implemented at the packet level removed the negative slope which is otherwise known as the clock skew. In online linear regression correction, it is typical to have an initial adjustment period since the node needs some time to gather enough data before beginning the correction. This slight limitation that affected the first few cycles of the data is indicated in the results and visible in Figure 4.5. Although the error minimizing techniques did perform as expected, the results appeared to be shifted by a constant factor of 5. The average synchronization error was calculated to be 4.4 μs with a standard deviation of 1.8 μs for online linear regression and an average error of 0 μs with a standard deviation of 9.4 μs for offline linear regression.

Interestingly, results showed that the online linear regression out-performed the offline linear regression which can be explained by the hardware limitations imposed by the Mica2 clock. This unusual behavior did go to show that the implementation presented in this study is able to take into account the instability of the hardware clock in a way that an offline linear regression would not be able to. The resulting error values which clustered around a mean of 4.4 μs all exhibited the same offset and if accounted for could be reduced to the value of 1.4 μs that [2] reported.

5.1 Contributions

As previously stated in Chapter 2, FTSP's reported micro-second accuracy synchronization error has been referred to by many new and upcoming synchronization protocols. However, some recent papers have expressed some suspicion regarding the ability to recreate the results that were obtained by the authors of the original FTSP report. The goal of this study was to first test this protocol as detailed in the literature and provide a fully functional FTSP implementation for comparison purposes. The testing done throughout this study uncovered that following the online implementation resulted in a synchronization error within the range of tens of micro-seconds and not around 1 μs like the original paper suggested. This mismatch launched a search into the nature of the hardware used for this experiment. It was discovered that the Mica2 hardware used by Maroti et al. for the FTSP implementation to obtain a synchronization error of 1.4 μs had a hardware limitation. Not only did the available implementation not follow the flooding time synchronization protocol, but there was an extra step that had to be implemented in order to realize such high time synchronization accuracy on the Mica2 hardware. This work provides a new implementation that will be available for future researchers as the new benchmark for the FTSP. Implementation flaws in the current code are corrected in this study and a new hardware calibration step is developed and presented. This new implementation has achieved micro-second accuracy when testing on Mica2 hardware nodes. In order for others to use this implementation, the hardware calibration process must be communicated to the wider community before publishing the code on an open source software such as GitHub.

5.2 Future Work

In an attempt to understand how the authors of the FTSP were able to get an absolute synchronization error of 1.4 μs in a wireless network configuration, the protocol presented in Maroti et. al's paper was carefully followed. Following the steps indicated in [2], the minimum synchronization error that could be obtained was around 13 μs and so more research was done in order to match the value reported. Research indicated that the hardware calibration step was a key component in obtaining such high accuracy. That helped narrow down the synchronization error to a mean of 4.4 μs with a standard deviation of 1.8 μs. Furthermore, a constant offset of 5 was visible in the test results, which if compensated for, would bring down the average synchronization error to 1.3 μs. At the moment, the cause of this offset is not understood, however, that research will constitute much of the future work for this study. Future work includes using a different approach than the hardware calibration to achieve micro-second accuracy. This approach would be to create a true micro-second timer using the available virtualized timers in the Mica2 as a base. However, this would require in-depth knowledge of the hardware and more research regarding the hardware limitation of the Mica2.

In order to gain more insight on the sources of error, it would be beneficial to conduct a detailed study of the other sources of error that plague synchronization within wireless sensor networks. In addition, future research includes studying the effect of increasing the distance between nodes on the propagation delay and how that will affect the time synchronization error. Other approaches that could be taken in the future would be to procure more resources, such as Mica2 hardware units, and run tests for longer periods of time in order to extract more information about the effect of the protocol on the data. For a more modern take on the FTSP, an adaptation of the FTSP implementation could be created to run on a field programmable gate array (FPGA) and by that making the protocol more accessible. Furthermore, a more advanced yet costly option would be to build an embedded system which uses a micro-controller and a byte-wise radio transceiver such as the CC1000 radio [31] from Texas instruments [58] since the multiple time-stamping depends on byte-wise radio transmission/reception. Ultimately, in addition to providing an implementation for the FTSP, this study investigated the state of available wireless synchronization protocol implementations and shed light to invaluable insight for

research and development within wireless sensor networks.

Bibliography

[1] R. A. Bloomfield, M. C. Fennema, K. McIsaac and M. Teeter, "Proposal and Validation of a Knee Measurement System for Patients with Osteoarthritis," in *IEEE Transactions on Biomedical Engineering*. doi: 10.1109/TBME.2018.2837620.

[2] M. Maroti, B. Kusy, G. Simon, and . Ldeczi, "The flooding time synchronization protocol, in *Proc. of the 2nd international conference on Embedded networked sensor systems (SenSys '04)*, Baltimore, MD, 2004, pp. 39–49.

[3] B. Kusy, J. Sallai, M. Maroti, "tinyos-release," github.com, 2008. [Online], Available: `https://github.com/tinyos/tinyos-release`. [Accessed: June 4, 2018].

[4] H. Aboelfotoh, E. Elmallah, and H. Hassanein, "On The Reliability of Wireless Sensor Networks, in *2006 IEEE International Conference on Communications*, Istanbul, 2006, pp. 3455–3460.

[5] M. Srbinovska, V. Dimcev and C. Gavrovski, "Energy Consumption Estimation of Wireless Sensor Networks in Greenhouse Crop Production. in *IEEE EUROCON 2017 -17th International Conference on Smart Technologies*, Ohrid, 2017, pp. 870–875.

[6] J. Yick, B. Mukherjee and D. Ghosal, "Wireless Sensor Network Survey. in *Computer Networks*, vol. 52, no. 12, pp. 2292-2330, Aug. 2008.

[7] D. Hamdan, O. Aktouf and I. Parissis, "Test and diagnosis of wireless sensor networks applications. in *2013 World Congress on Computer and Information Technology (WCCIT)*, Sousse, 2013, pp. 1–7.

[8] C. R. Baker, K. Armijo, S. Belka *et al.*, "Wireless Sensor Networks for Home Health Care," in *21st International Conference on Advanced Information Networking and Applications Workshops (AINAW'07)*, Niagara Falls, Ont, 2007, pp. 832–837.

[9] A. Hilmani, A. Maizate, and L. Hassouni, "Designing and Managing a Smart Parking System Using Wireless Sensor Networks, in *Journal of Sensor and Actuator Networks*, vol. 7, no. 2, p. 24, Jun. 2018.

[10] I.-K. Rhee, J. Lee, J. Kim, E. Serpedin, and Y.-C. Wu, "Clock Synchronization in Wireless Sensor Networks: An Overview, in *Sensors*, vol. 9, no. 1, pp. 56-85, Jun. 2009.

[11] Willig, A. "Wireless Sensor Networks: Concept, Challenges and Approaches. in *Elektrotechnik Und Informationstechnik*, vol. 123, no. 6, pp. 224-231, Jun. 2006.

[12] S. M. Lasassmeh and J. M. Conrad, "Time synchronization in wireless sensor networks: A survey, in *Proc. of the IEEE SoutheastCon 2010(SoutheastCon)*, Concord, NC, 2010, pp. 242–245.

[13] Ru. Singh, J. Singh and Ra. Singh, "SECURITY CHALLENGES IN WIRELESS SENSOR NETWORKS" in *IRACST - International Journal of Computer Science and Information Technology & Security (IJCSITS)*, vol. 6, no. 3, pp. 1-5, May–Jun. 2016.

[14] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam and E. Cayirci, "A survey on sensor networks," in *IEEE Communications Magazine*, vol. 40, no. 8, pp. 102–114, Aug. 2002.

[15] D. L. Mills, "Internet time synchronization: the network time protocol," in *IEEE Transactions on Communications,* vol. 39, no. 10, pp. 1482–1493, Oct. 1991.

[16] N. Bulusu and S. Jha, *Wireless Sensor Networks: A Systems Perspective*, Artech House: Norwood MA, USA, 2005.

[17] J. N. Al-Karaki and A. E. Kamal, "Routing techniques in wireless sensor networks: a survey," in *IEEE Wireless Communications*, vol. 11, no. 6, pp. 6–28, Dec. 2004.

[18] B. Sundararaman, U. Buy and A. Kshemkalyani , "Clock Synchronization for Wireless Sensor Networks: a Survey. in *Ad Hoc Networks*, vol. 3, no. 3, pp. 281-323, May 2005.

[19] A. B. Kulakli and K. Erciyes, “Time synchronization algorithms based on Timing-sync Protocol in Wireless Sensor Networks,” in *2008 23rd International Symposium on Computer and Information Sciences*, Istanbul, 2008, pp. 1–5.

[20] F. Wang, Y.-Q. Qin, and H.-D. Lei, “Review of Time Synchronization in Wireless Sensor Networks, in *2017 International Conference on Computer, Electronics and Communication Engineering (CECE 2017)*, Sanya, 2017, pp. 299–305.

[21] P. Kaur and Abhilasha, “TIME SYNCHRONIZATION IN WIRELESS SENSOR NETWORKS: A REVIEW, in *International Journal of Computer Engineering and Applications*, vol. 9, no. 6, pp. 138-145, Jun. 2015.

[22] J. Elson, L. Girod, and D. Estrin. “Fine-Grained Network Time Synchronization using Reference Broadcasts.” in *Proc. Fifth Symposium on Operating Systems Design and Implementation (OSDI 2002)*, vol. 36, no. SI, pp. 147-163, Dec. 2002.

[23] S. Ganeriwal, R. Kumar, and M. B. Srivastava, “Timing-sync protocol for sensor networks, in *Proc. of the first international conference on Embedded networked sensor systems - SenSys 03*, Los Angeles, CA, 2003, pp. 138–149.

[24] S. Ping, “Delay Measurement Time Synchronization for Wireless Sensor Networks,” *Intel Research Berkeley Lab, IRB-TR-03-013,* June 2003.

[25] K. Romer, “Time Synchronization in Ad Hoc Networks,” *Proc. ACM Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc 01)*, Long Beach, CA, 2001, pp. 173-182.

[26] M. Mock, R. Frings, E. Nett, and S. Trikaliotis, “Continuous Clock Synchronization in Wireless Real-time Applications,” *Proc. 19th IEEE Symposium on Reliable Distributed Systems SRDS-2000,* Nurnberg, 2000, pp. 125-133.

[27] M. L. Sichitiu and C. Veerarittiphan, “Simple, accurate time synchronization for wireless sensor networks,” in *2003 IEEE Wireless Communications and Networking*, New Orleans, LA, 2003, pp. 1266–1273 vol.2.

[28] J. Elson and D. Estrin, "Time synchronization for wireless sensor networks," in *Proc. 15th International Parallel and Distributed Processing Symposium (IPDPS 2001)*, San Francisco, CA, 2001, pp. 1965–1970.

[29] Gy. Simon, M. Maroti, A. Ledeczi *et al.*, "Sensor network-based countersniper system," in *Proc. of of the 2nd international conference on Embedded networked sensor systems (SenSys '04),* Baltimore, MD, 2004, pp. 1-12.

[30] M. Maroti, B. Kusy, G. Simon, and A. Ledeczi, "Robust multi-hop time synchronization in sensor networks," in *Proc. of the International Conference on Wireless Networks (ICWN'04),* Las Vegas, NV, 2004, pp. 454–460.

[31] Texas Instruments, "Single Chip Very Low Power RF Transceiver," CC1000 datasheet, Feb. 2007.

[32] M. Maroti, "Clocks, Time Stamping and Time Synchronization," in *TinyOS Technology Exchange,* Berkeley, USA, Feb 2004.

[33] P. Ballal and F. Lewis, Class Lecture, Topic: "Introduction to Crossbow Mica2 Sensors," Automation & Robotics Research Institute, University of Texas at Arlington. Arlington, TX, 2007

[34] D. Djenouri and M. Bagaa, "Synchronization Protocols and Implementation Issues in Wireless Sensor Networks: A Review," in *IEEE Systems Journal,* vol. 10, no. 2, pp. 617–627, Jun. 2016.

[35] F. Ferrari, M. Zimmerling, L. Thiele, and O. Saukh, "Efficient network flooding and time synchronization with glossy, in *Proc. IPSN,* Chicago, IL, 2011, pp. 73-84.

[36] L. Schenato and F. Fiorentin, "Average TimeSynch: A consensus-based protocol for clock synchronization in wireless sensor networks, in *Automatica,* vol. 47, no. 9, pp. 1878-1886, Sep. 2011.

[37] D. Gay, P. Levis and R.v. Behren, "The nesC language: a holistic approach to networked embedded systems," in *Proceedings of the PLDI '03,* San Diego, CA, 2003, pp. 1–11.

[38] T. Kunz and E. McKnight-MacNeil, “Implementing clock synchronization in WSN: CS-MNS vs. FTSP,” in *2011 IEEE 7th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob),* Wuhan, 2011, pp. 157–164.

[39] C. Lenzen , P. Sommer , R. Wattenhofer, “Optimal clock synchronization in networks,” in *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems,* Berkeley, CA, Nov. 2009, pp. 225–238.

[40] X. Hu, B. Wang and X. Hu, “A Novel Energy-Balanced Time Synchronization Protocol in Wireless Sensor Networks for Bridge Structure Health Monitoring,” in *2010 2nd International Workshop on Database Technology and Applications,* Wuhan, 2010, pp. 1-5.

[41] R. Sugihara and R. K. Gupta, “Clock synchronization with Deterministic Accuracy Guarantee, in *Wireless Sensor Networks - 8th European Conference (EWSN),* Bonn, 2011, pp. 130–146.

[42] N. Xu, X. Zhang, Q. Wang, J. Liang, G. Pan and M. Zhang, “An Improved Flooding Time Synchronization Protocol for Industrial Wireless Networks,” in *2009 International Conference on Embedded Software and Systems,* Zhejiang, 2009, pp. 524–529.

[43] P.A. Sommer, “Wireless embedded systems. time, location, and applications,” PhD thesis, ETH ZURICH, 2011.

[44] F. Wang, X. Wu, Y. Wang, P. Zeng, and Y. Xiao, “Extensible time synchronization protocol for wireless sensor networks, in *International Journal of Sensor Networks,* vol. 23, no. 1, pp. 29–39, 2017.

[45] F. Wang, X. Wu, Y. Pang, C. Yu, Y. Hu and X. Liu, “A time synchronization method of Wireless Sensor Networks based on the simulated annealing algorithm,” in *The 26th Chinese Control and Decision Conference (2014 CCDC),* Changsha, 2014, pp. 870–875.

[46] G. Huang, A. Y. Zomaya, F. C. Delicato and P. F. Pires, “Long term and large scale time synchronization in wireless sensor networks,” in *Computer Communications,* vol. 37, pp. 77–91, Jan. 2014.

[47] Q. M. Chaudhari and E. Serpedin, "Clock estimation for long-term synchronization in wireless sensor networks with exponential delays," in *EURASIP Journal on Advances in Signal Processing,* vol. 2008, Article No. 27. pp. 1–6, Jan. 2008,

[48] F. Gong and M. L. Sichitiu, "On the Accuracy of Pairwise Time Synchronization," in *IEEE Transactions on Wireless Communications,* vol. 16, no. 4, pp. 2664–2677, Apr. 2017.

[49] L. Li, J. Yang and Y. Mao, "Local clock-based timestamp synchronization for multihop ad hoc and sensor networks," in *2017 11th International Conference on Signal Processing and Communication Systems (ICSPCS),* Surfers Paradise, QLD, 2017, pp. 1–6.

[50] F. Gong and M. L. Sichitiu, "Temperature compensated Kalman distributed clock synchronization," in *Ad Hoc Networks,* vol. 62, no. C, pp. 88–100, Jul. 2017.

[51] J. Wang, W. Dong, Z. Cao and Y. Liu, "On the Delay Performance in a Large-Scale Wireless Sensor Network: Measurement, Analysis, and Implications," in *IEEE/ACM Transactions on Networking,* vol. 23, no. 1, pp. 186–197, Feb. 2015.

[52] T. Schmid , Z. Charbiwala , J. Friedman , Y.H. Cho and M.B. Srivastava, "Exploiting manufacturing variations for compensating environment-induced clock drift in time synchronization," in *Proc. ACM SIGMETRICS,* Annapolis, MD, 2008, pp. 97-108 .

[53] H. D. Soares, R. P. de Oliveira Guerra and C. V. N. de Albuquerque, "Ftsp+: A mac timestamp independent flooding time synchronization protocol." in *XXXIV Simpsio Brasileiro de Redes de Computadores e Sistemas Distribudos-SBRC. Sociedade Brasileira de Computaao,* Salvador, Bahia, 2016, pp. 820–832.

[54] P. Levis, "TinyOS Extension Proposals," github.com, 2006. [Online], Available: `https://github.com/tinyos/tinyos-release/tree/tinyos-2_1_2/doc/txt`. [Accessed: June 30, 2018].

[55] "Master nesc repository" github.com. [Online], Available: `https://github.com/tinyos/nesc`. [Accessed: August 6, 2018].

[56] J. Sallai, "Why microsecond accuracy in FTSP with telosb," [Tinyos-help]:http://mail.millennium.berkeley.edu/pipermail/tinyos-help/2013-September/057604.html, Sept. 2013. [Accessed: April 13, 2018].

[57] Microchip, "8-bit Atmel Microcontroller with 128KBytes In-System Programmable Flash," ATmega128 datasheet, 2011.

[58] Analog, Embedded Processing, Semiconductor Company, Texas Instruments - TI.com, TI.com. [Online]. Available: http://www.ti.com/. [Accessed: 16 Sep, 2018].

Appendix A

```
/*
 * Copyright (c) 2002, Vanderbilt University
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 *
 * - Redistributions of source code must retain the above copyright
 *   notice, this list of conditions and the following disclaimer.
 * - Redistributions in binary form must reproduce the above copyright
 *   notice, this list of conditions and the following disclaimer in the
 *   documentation and/or other materials provided with the
 *   distribution.
 * - Neither the name of the copyright holders nor the names of
 *   its contributors may be used to endorse or promote products derived
 *   from this software without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
 * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
 * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
 * FOR A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL
 * THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT,
 * INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
 * (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
 * SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
```

```
27  * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
28  * STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
29  * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
30  * OF THE POSSIBILITY OF SUCH DAMAGE.
31  *
32  * @author: Miklos Maroti, Brano Kusy (kusy@isis.vanderbilt.edu)
33  * Ported to T2: 3/17/08 by Brano Kusy (branislav.kusy@gmail.com)
34  */
35
36 #include "TestFtsp.h"
37 #include "RadioCountToLeds.h"
38
39 module TestFtspC
40 {
41     uses
42     {
43         interface GlobalTime<TMilli>;
44         interface TimeSyncInfo;
45         interface Receive;
46         interface AMSend;
47         interface Packet;
48         interface Leds;
49         interface PacketTimeStamp<TMilli,uint32_t>;
50         interface Boot;
51         interface SplitControl as RadioControl;
52     }
53 }
54
55 implementation
56 {
57     message_t msg;
58     bool locked = FALSE;
59
60     event void Boot.booted() {
61         call RadioControl.start();
62     }
```

```
63
64  event message_t* Receive.receive(message_t* msgPtr, void* payload,
    uint8_t len)
65  {
66      call Leds.led0Toggle();
67      if (!locked && call PacketTimeStamp.isValid(msgPtr)) {
68          radio_count_msg_t* rcm = (radio_count_msg_t*)call Packet.
    getPayload(msgPtr, sizeof(radio_count_msg_t));
69          test_ftsp_msg_t* report = (test_ftsp_msg_t*)call Packet.
    getPayload(&msg, sizeof(test_ftsp_msg_t));
70
71          uint32_t rxTimestamp = call PacketTimeStamp.timestamp(msgPtr);
72
73          report->src_addr = TOS_NODE_ID;
74          report->counter = rcm->counter;
75          report->local_rx_timestamp = rxTimestamp;
76          report->is_synced = call GlobalTime.local2Global(&rxTimestamp);
77          report->global_rx_timestamp = rxTimestamp;
78          report->skew_times_1000000 = (uint32_t)call TimeSyncInfo.
    getSkew()*1000000UL;
79          report->ftsp_root_addr = call TimeSyncInfo.getRootID();
80          report->ftsp_seq = call TimeSyncInfo.getSeqNum();
81          report->ftsp_table_entries = call TimeSyncInfo.getNumEntries();
82
83          if (call AMSend.send(AM_BROADCAST_ADDR, &msg, sizeof(
    test_ftsp_msg_t)) == SUCCESS) {
84            locked = TRUE;
85          }
86      }
87
88      return msgPtr;
89  }
90
91  event void AMSend.sendDone(message_t* ptr, error_t success) {
92      locked = FALSE;
93      return;
```

```
94      }
95
96      event void RadioControl.startDone(error_t err) {}
97      event void RadioControl.stopDone(error_t error){}
98  }
```

Curriculum Vitae

Name:	Asma Khalil
Post-Secondary Education and Degrees:	The University of Western Ontario London, ON 2009 - 2013 B.E.Sc. University of Western Ontario London, ON 2017 - 2018 M.E.Sc.
Honours and Awards:	Steinmetz-Woonton (Merit) Scholarship 2011
Related Work Experience:	Teaching Assistant The University of Western Ontario 2017 - 2018