1. **Introduction**

Over the last few decades, advancements in technology have allowed for small low-powered devices that can accomplish a multitude of tasks. It is now commonplace for these devices to contain radios and to be members of distributed systems over complex networks. These devices also contain sensors that collect real-time data about their environments and stream this data over the network. These networks are referred to as wireless sensor networks (WSNs).

WSNs have applications in a multitude of areas including military intelligence, environmental monitoring, medical treatment, home automation, and inventory control. In many of these applications, power is a resource under hard constraints, and power management is of utmost importance. As tasks become increasingly complex, dealing with real-time events and streams of data, these same networks have the hard constraint of accuracy as well. Time synchronization plays a large part in supporting this accuracy. Because of this, it is crucial that the power characteristics of time synchronization mechanisms are analyzed and optimized in order to make them as efficient as possible.

In this project, we analyze the power characteristics of a popular time synchronization algorithm for WSNs called the Flooding Time Synchronization Protocol (FTSP). Our goal is to identify important power/accuracy tradeoffs that might lead to future work in optimizing the FTSP algorithm to improve power consumption. In section two we give a brief explanation of how the FTSP algorithm works. In section three we explain our experiment, including our method and the results. In section four we analyze our findings, and finally in section five we identify future work that could build on top of what we have done.

2. **FTSP Algorithm**

The flooding time synchronization protocol (FTSP) provides 2 microsecond clock synchronization accuracy. In theory, it requires “low communication bandwidth” and is able to handle failures in the network. The algorithm is described and tested in [1]. As taken from this paper, we give a more brief description of how it works in this section.

FTSP synchronizes the clock of a message sender to one or more receivers. The RBS and TPSN time-synch algorithms suffer from uncertainties associated with network communication. MAC layer time-stamping eliminates these uncertainties and the errors they create. Adjustments for clock-drift make the solution highly accurate. FTSP uses linear regression to handle clock-drift among the motes in the sensor network.

The algorithm selects a root node that has the global time every other node synchronizes to. The job of the root node is to broadcast the global time at regular intervals (the time-synch period). Actually, every mote also broadcasts its estimation of the global time once per time-synch period. The receivers are able to gain an accuracy of 1.4 microseconds between motes.

The time-stamp accuracy is skewed by the clock-drift between motes. These drifts are as large as 40 microseconds. To compensate, FTSP estimates the drift between a sender clock and a receiver clock. This offset should change linearly. To estimate the drift between clocks, FTSP uses linear regression.
3. Our Experiment

3.1 Hardware Platform

We used six Crossbow MICAz motes in our experiment.[2] These devices are ideal for low-power wireless sensor networks. Each device has a 2.4GHz Chipcon CC2420 radio that is IEEE 802.15.4 compliant, with a data rate of 250kbps. The MICAz also has 128KB of program flash memory, and 512KB of measurement flash memory. Lastly, the devices have a serial port that is used for programming, and attaching compatible sensor boards. The advertised battery life for the motes supplied by two AA batteries is greater than two years.

3.2 Software Platform

TinyOS 2.1 was used to develop the code that ran during our experiment. TinyOS is an open source event based operating environment designed specifically for embedded sensor networks. It is designed to support concurrency intensive operations with minimal hardware requirements.[3] TinyOS is written in and uses the NesC programming language. NesC is an extension of the C programming language with additional language features supporting components and concurrency. For the experiment, parts of the TinyOS library were modified in order to provide for more accurate timing measurements.

3.3 Method

Since the bulk of power consumption by FTSP is due to communication with other nodes, the experiment focused on measuring power consumption of the radio. The radio draws approximately the same amount of current while it stays in one state. Based on the MICAz specifications, the radio draws 19.7 mA in receive mode, 11 mA in transmit mode, and .02 mA in idle mode. If the time the radio spends in each mode is measured, then the total current draw is known for the radio.

A software-based method was used for measurement. A component was designed to take timestamp data and reliably send it over the serial port to a PC that was connected to the mote. The component uses an in-memory buffer to initially record the timestamp information, and then issues a write task to TinyOS. The write tasks are handled by TinyOS at a non-critical time. This avoids synchronously sending packets over the serial port, which might affect the characteristics of FTSP during the experiment.

In order to collect the most accurate measurements possible TinyOS needed to be modified. Based on the radio state diagram (see appendix A), we identified the control pins responsible for transitioning the radio from one state to another. Timestamps were collected at the lowest software level possible. In the hardware specific strobe interfaces, timestamps were collected as soon as the control pins for the radio were strobe. Timestamps were collected at millisecond precision. A Java application running on the connected PC was used to receive the timestamp packets sent over the serial port.

The voltage across the radio was taken from the CC2420 specifications documents. The CC2420 includes a voltage regulator that provides a constant 1.8 V power supply to the radio. The regulator is connected to an unregulated 2.1 to 3.6 V power supply. In our experiment we assume that the regulator is efficient, and the 1.8 V can be used to determine the voltage across the radio. In the worst case, our results are off by at most a range of 1.5 V.
Six motes were used during the experiments. For the first experiment, we ran the FTSP implementation provided by TinyOS along with our component. FTSP was ran five times for five minutes, varying the synchronization periods each run. Synchronization periods of 1 second, 5 seconds, 10 seconds, 30 seconds and 60 seconds were used. For the second experiment, we ran the FTSP implementation along with a component that measured the accuracy of FTSP. For each mote in the network, this component broadcasts the current local and global time, network id, and whether the global timestamp is valid. From these statistics, synchronization error and other accuracy statistics can be calculated. The second experiment ran for the same synchronization periods as the first.

4. Results

4.1 Power Consumption

To us, the most obvious parameter that should affect the power consumption of FTSP is the time-synch period. For example, doubling the time-synch period from 1 second to 2 seconds should essentially halve the number of messages being sent by the algorithm. This insight led us to believe that, while not perfectly true in practice, halving the messages sent would also halve the power consumption.

Figure 1 shows that the above assumption is false. In fact, using typical values the time-synch period [1] shows that the power consumption varies very little.

4.2 Time Per Radio State

The results from the previous paragraph exist for good reason. Table 1 gives a significant indication of what the radio use is during our experiments. The radio never shuts off and spends most of the experiment in receive state. There is an order of magnitude difference between the time the radio is in transmit mode and the time the radio is in receive mode. Figure 2 attempts to represent graphically the time spent in receive vs. the time spent in transmit.
### Table 1.

<table>
<thead>
<tr>
<th>Time Synch Period (In seconds)</th>
<th>Radio Control States (Time In Each State In Milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Idle</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>30</td>
<td>1</td>
</tr>
<tr>
<td>60</td>
<td>1</td>
</tr>
</tbody>
</table>

### 4.3 FTSP Accuracy

While we made no significant impact on the power consumption by varying the time-sync period, we did see expected results for the FTSP accuracy. When the time-sync period is increased, the time-sync accuracy drops among the motes. Additionally, longer periods also mean the network requires a greater length of time for all motes to be synched. See appendix B for corresponding graphs.

### 5. Analysis

Our data suggests that power consumption might not be affected by FTSP. This however is not true. Because each mote in the network broadcasts a message once per sync-period and broadcasts occur at random times, every mote must have its radio in receive mode when it is not transmitting. As a result, nodes spend a tremendous amount of time in receive mode.

Therefore, there is great opportunity to reduce power consumption by putting the radio into sleep mode where it is currently in receive mode. We would like to give a theoretical lower bound for how much energy could be saved. Figure X shows how many messages should be sent and how many messages should be received for each of our experiments. Consider the case for which the time-sync period is 1 second. Over a five minutes period one mote will transmit 300 messages. Each mote will also receive 900 messages from the three other motes in the network. Naively assuming transmits and receives take 1 millisecond (about what we timed one transmit to take), the radio needs to be on for only 1200 milliseconds out of 300000. The potential energy saving is over 99%. Our FTSP algorithm would end up using .1 joule over 5 minutes instead of 10 joules. The real time penalty for switching between radio states, for receiving, and for transmitting is needed to be explicitly accurate.
An algebraic analysis is probably more useful for abstractly describing the potential energy savings. If the time-sync period is $t$, the frequency of transmission, $f$, for each mote is $(1/t)$ hz. The number messages received by each mote is $(n-1)*f$, where $n$ is the number of motes in the network. Let the transmit time be $t_x$, the receive time be $t_x$, the state switch cost to be $c$ (for all states). The radio needs to be turned on for a minimum of $(2f*c)(t_x*c + (n-1)r_x)$ milliseconds. Define the maximum time spent in receive mode, $RX_{max}$, as $(1000-2f*c*t_x)$ and the minimum time spent in receive mode, $RX_{min}$, as $(2f*c*r_x)$. The maximum power savings is given by $(RX_{max}-RX_{min}/1000)*100$.

6. Future Work

The obvious next step is to come up with a way to put each node's radio to sleep in a coordinated fashion. With this approach, radio time spent in receive mode is reduced and, if sleep cycles are coordinated in an intelligent manner, no accuracy is lost. There has been a large amount of research done on this topic to draw from.[4, 5] We will take this work, and use it to improve the usage of FTSP in TinyOS. For example, a simple solution is to have each mote estimate when it will receive a message from the other motes. Then timers can be set to put the radio in receive mode at these times. Unfortunately motes enter and leave the network at random times, thus the times at which individual motes broadcast their messages is not statically determined. We believe that it would be possible for each mote to dynamically construct a representation of the network traffic created by FTSP. It can use this representation to duty cycle the radio and conserve energy.

7. References


