Ben Axelrod
4/16/2006
CS 7210 Final Project

Reference Broadcast Time Averaging

Introduction

Time synchronization is important for all distributed systems. Wireless sensor networks often rely on synchronized clocks for data fusion and synchronized behavior. However, because of the requirements of these low-power and low-cost devices, clock skews are often very large. This paper presents a simple variant of the Reference-Broadcast Synchronization scheme [4] geared towards implementation on the GNATs [9]. This new scheme, called Reference Broadcast Time Averaging is simple enough to run on simple sensor networks such as the GNATs. While it ultimately failed to be tested on the GNATs due to hardware limitations, the algorithm was tested in simulation with encouraging results. Fast convergence times were observed for fully connected networks.

Approach

Algorithm Overview

Many time synchronization schemes employ a centralized server in which all of the nodes synchronize with. Cristian's algorithm is one such scheme. Other methods measure round trip delay of the messages such as NTP [8]. Both of these techniques are sub-optimal for wireless sensor networks such as the GNATs. These sensor networks usually do not have a centralized server or reliable communication channels. It is because of these limitations that a reference broadcast scheme must be employed.

Reference broadcast schemes work by using a third party to aid two nodes in synchronizing with each other. This method lends itself naturally to sensor networks where most communication is done through broadcasts. The process is as follows: (please refer to Figure 1). First, node A broadcasts a short "ping" message to its neighbors. This message contains only minimal information such as node A's ID number and a message ID number. It does not matter when this ping is sent and there is no timestamp on the message.

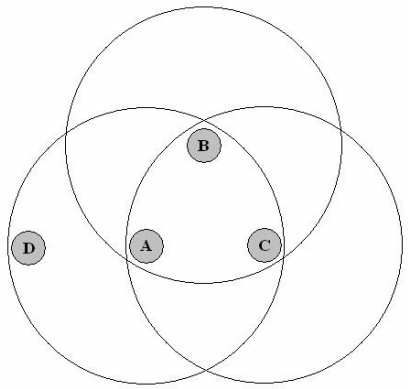

Figure 1 – Reference Broadcast

Nodes B, C, and D will all receive this ping at nearly the same time. This is another advantage of this algorithm with the GNATs. The GNATs use infrared light to communicate over short distances (less than a meter). Therefore any discrepancy in the arrival time is solely due to the infrared receiver saturating and the microcontroller interrupt handler. The infrared receiver saturation time can be neglected, and the interrupt service routine will be triggered three or four μs after arrival. Nodes B, C, and D must now store A's ID, the ping message ID, and their local clock when they received the ping.

Then, these nodes will use this information to synchronize with each other. For example, node B will broadcast a synchronization request to it neighbors, and of these, C will be able to synch itself with B. Elson et al. recorded the clock offsets then used linear regression to estimate the drift between each pair of nodes. Because of the limited resources of the GNATs, a simpler scheme had to be developed. Basically, when a node receives a synchronization request, it averages the times each node heard the ping. Then it uses the difference to update its current time and skew. This has the effect of averaging all the clocks in the network.

Time on the GNATs

Before the in depth analysis of the algorithm, it will help to have an overview of time on the GNATs. The GNATs have a 4 MHz RC oscillator which is subject to quite a bit of drift. In a small experiment with only 6 GNATs, after only 18 seconds of running, there was over half a second difference between the fastest and slowest GNATs. This is

about ±1.5 % of drift. This discrepancy arises from variable battery voltage and manufacturing differences in the timer circuitry. There are however, two easy ways to tune the clock for precise timing. These methods allow the compensation for the clock skew to be stored in memory for later use. Because of the limited memory of the GNATs, usually only one application can be run at a time. So the skews can be adjusted once, then when another application is swapped in, it can take advantage of the pre-tuned clocks.

The first method of clock adjustment is a hardware tuning register. This 5 bit register allows for up to ±12.5 % of tuning. This is equivalent to 0.78% of tuning per bit, which seems too coarse of a resolution for this purpose. However, this method is attractive because the tuning is done entirely in hardware.

The second and preferred method of clock tuning sets the period at which the timer overflows. The 4 MHz oscillator gets divided by 4 because there are 4 oscillator periods per instruction on the GNAT's PIC16F87 microcontroller. This 1 μs instruction clock is then divided by 16 in a pre-scaler. Then the timer overflows every 256 increments. Finally, this goes through the post-scaler which enforces 16 overflows per interrupt. This interrupt increments the "official time" of the GNATs. With the settings mentioned here, this interrupt gets called about 15 times per second. This 15th of a second is the native unit of time on the GNATs and is called a "tick". The entire calculation can be seen in Figure 2.

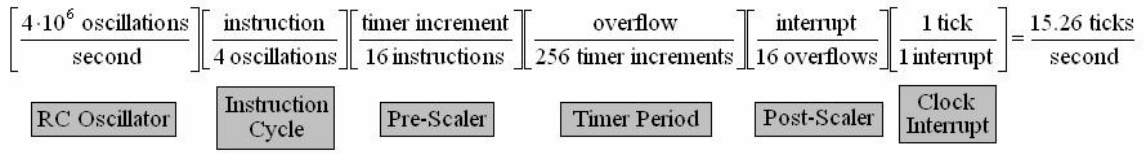

Figure 2 – GNATs Timer Calculation

Because of other restrictions in the GNAT OS, there was only an 8 bit timer available for general timekeeping. This timer normally overflows at 0xFF, but its period can be set. This is the second method for tuning the clock. By changing the period of the timer, we have a much finer grained control of the skew of this clock.

In preparation for this algorithm to be run on the GNATs, many changes to the API were made. Despite these changes, an unforeseen hardware limitation prevented any real testing on the GNATs. The infrared communication protocol the GNATs use requires precise timing of the message bits. This means that when a GNAT is sending or receiving a message, all other interrupts must be disabled. So if a message is long enough, it will make the GNAT skip a tick. The messages we need to send last 55 ms and the interrupt occurs every 66 ms, so there is a significant probability of the GNAT skipping a tick on every message. This problem can be remedied by checking the timer before and after the message interrupts then adding in the skipped tick if necessary. The trouble arises because the value of the post-scaler (16 overflows per interrupt) is not available to the user program. This is essential to be able to determine if there is a lost

tick. It was determined that this synchronization scheme cannot handle this high tick loss rate, and testing on the GNATs was abandoned.

The Simulator

The reason this algorithm was not implemented in simulation first is due to the difficulty in getting a discrete and sequential simulator to accurately model the diverging clocks of a sensor network. Most simulators run the code in strict lock-step. Either each agent's code gets run sequentially then the changes applied in parallel (as in TeamBots [1]), or they are run purely sequentially (as in MASON [7]). The MASON simulator was chosen for this project due to its features and the author's familiarity with the software.

In MASON, the code for each agent gets run sequentially. The mechanism which allows us to simulate diverging clocks is the modulo operator. Each agent randomly chooses a skew at instantiation. Then that agent's code is only run when the iteration mod that skew is zero. The skews were chosen uniformly random between 50 and 60. It was found that this range seemed to model the speed of clock divergence of the GNATs. Another feature of MASON randomizes the order in which the agents are run at every iteration. This further helps to simulate a real network.

Additionally, code does not transfer very well between simulators and the real devices. Code for the GNATs is event based. Meaning that you send messages to other GNATS, and you have a specific function that gets called whenever you receive a message. In MASON, there is no concept of sending a message. There is only data structures. To "send a message" involves you collecting a list of pointers your neighbor objects, then dereferencing the pointers and putting your data in your neighbors. Although it gets the job done, it is a slightly backwards way of thinking about (and coding) the problem.

Full Description of Algorithm

This algorithm involves only three simple equations to average the clock offset and skew between two neighbors. Figure 3 shows an overview of the message passing involved. This diagram is from the viewpoint of node C. The only times node C needs to know about are when it heard the ping, when its neighbor heard the ping, and the current time.

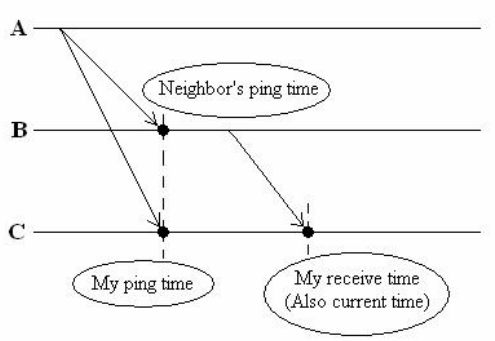

Figure 3 – Important Message Times

The first step in the algorithm is to average the times when both you and your neighbor heard the ping from an external source. This is simply:

$$\text{averaged ping time} = \frac{\text{neighbor's ping time} + \text{my ping time}}{2}. \qquad (1)$$

Keep in mind that the neighbor's ping time will be in the synchronization message sent to you, and your ping time will be in your own list of pings. The next step is to scale the current time appropriately due to the new ping time. This is most easily thought of as a "similar triangles" argument and is illustrated in Figure 4.

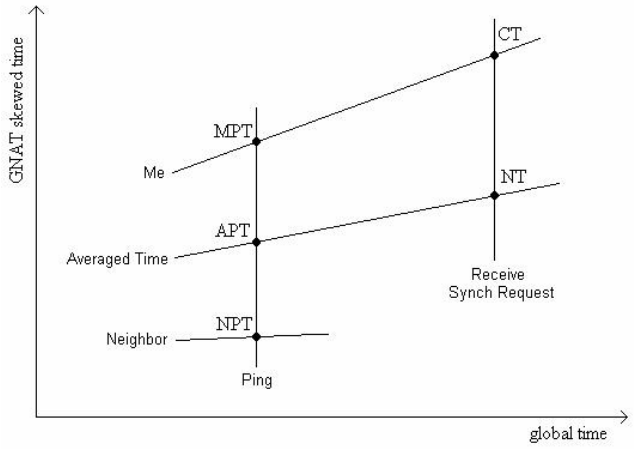

Figure 4 – Current Time Update Argument

In this figure, MPT = my ping time, APT = average ping time, NPT = neighbor ping time, and CT is the current time which are all known. The current time becomes the old time, and the new time (NT) is simply:

$$\text{new time} = \frac{\text{old time} \cdot \text{averaged ping time}}{\text{my ping time}}. \qquad (2)$$

Please note that "old time" was the most up to date clock reading until this equation. It will have to be stored for use in the next equation. The final step is to modify the skew of

the clock. This is an important step for clock synchronization. The skew update equation is just:

$$\text{skew} = \frac{\text{skew} \cdot (\text{old time} - \text{my ping time})}{\text{new time} - \text{averaged ping time}}. \quad (3)$$

It is easy to see the intuition behind this equation from examination of Figure 5. The skew merely needs to be adjusted by the ratio of A/B as seen in the figure. It may seem counter-intuitive that we are turning back the clock, yet increasing the skew value (from the example in the diagram). This is because the larger skew value will make the clock slow down, which is what we want.

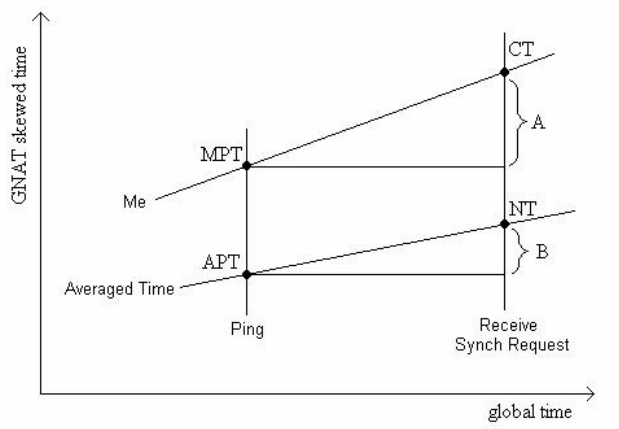

Figure 5 – Skew Update Argument

It was found to be necessary to remove the ping from memory after you have completed this synchronization. This prevents the node from re-synching to the same data more than once. Here is some simple pseudo code to further elaborate the algorithm. This pseudo code is written from an event-based paradigm similar to the one found on the GNATs. It is assumes that there is external code to call `On_Receive_Ping` and `On_Receive_Synch` appropriately as soon as a message is received. Also not shown are appropriate bounds on the `NewTime` and `Skew` equations, which will be device dependant.

```
On_Receive_Ping( ping_msg )
{
    if (ping_msg.pingerID is already in ping_list)
        Overwrite old ping by same neighbor
    else
        Add ping_msg to ping_list
}

On_Receive_Synch(synch_msg)
{
    Let 'line' be the line of the ping_list where synch_msg.pingerID and
        synch_msg.pingMsgID match an entry
    if (no match)
        return
    AveragedPingTime = (ping_list[line].my_ping_time + synch_msg.ping_time) / 2
    OldTime = getTimer();
    NewTime = OldTime * AveragedPingTime / ping_list[line].my_ping_time
    setTimer(NewTime);
```

```
    Skew = Skew * (OldTime - ping_list[line].my_ping_time) / (NewTime - AveragedPingTime)
    Remove line from ping_list
}

main()
{
    while(1)
    {
        With small probability
            Send a ping message
        With small probability
            Send a synch message
    }
}
```

Results and Discussion

Several experiments were run in simulation to evaluate the effectiveness of this algorithm. The same parameters of the algorithm were used for all experiments. Skews are randomly chosen between 50 and 60. Nodes emit a visual flash every 3 "ticks" for human visual verification. Keep in mind a tick occurs every 50 – 60 iterations depending on the skew of the node. Every time the ticks were incremented, each node sent a ping message with 0.2 probability and a synchronization message with 0.2 probability. The simulator ran at about 250 iterations per second. Therefore, each node sent a ping and synchronization message about once per second.

The first and simplest experiment is a group of 19 nodes which are fully connected as illustrated in Figure 6.

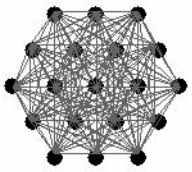

Figure 6 – 19 Fully Connected Nodes

The experiment was run as follows: The nodes were allowed to run normally without any synchronization code for 2000 iterations. This allowed the clocks to diverge a fair amount. At this time, it was observed that the nodes stopped blinking in unison, and there was continuous blinking. Then the synchronization code was switched on for 2000 iterations. After this time, the nodes were seen to be blinking in unison again. Next, the synchronization code was turned off for the final 2000 iterations, and the nodes continued to blink in unison. This demonstrated that the algorithm fully synchronized the clocks and that the clocks stayed synched even after the algorithm was not active. The clock values for each node were recorded every 100 iterations. The divergence of the clock values can be seen plotted in Figure 7. The divergence is simply the max time minus the min time value at that iteration.

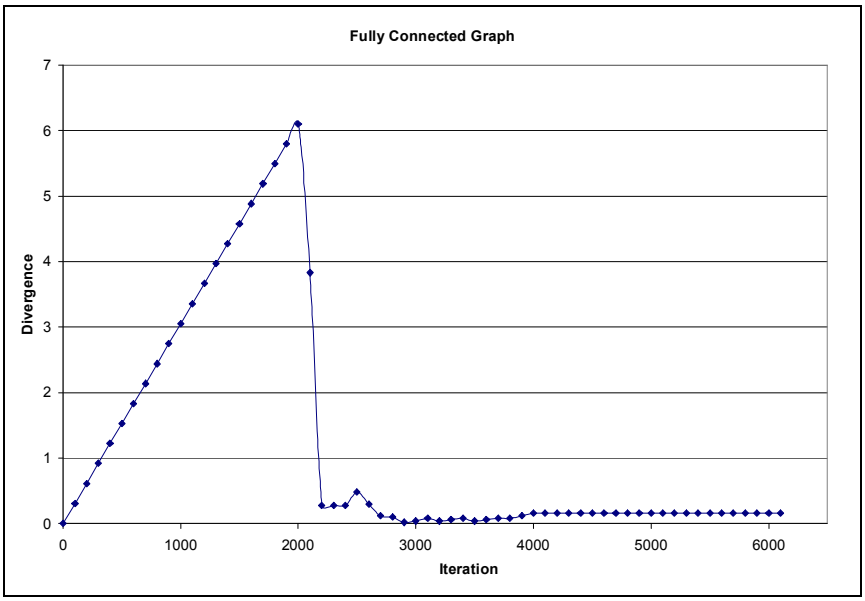

Figure 7 – Fully Connected Graph Results

It is easy to see the three portions of the graph where the synchronization code was on or off. It is remarkable how fast the network converges. After only about 200 iterations, the divergence was down to a reasonable level.

The next experiment was on a hexagonal grid of 19 nodes which is not fully connected as seen in Figure 8. This configuration is much more typical for sensor networks.

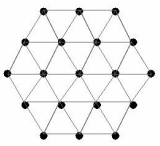

Figure 8 – Hexagonal Grid

Again, the synchronization was turned on at iteration 2000, however this time it was necessary to leave it on until iteration 15000. This configuration converged much more slowly as seen in Figure 9.

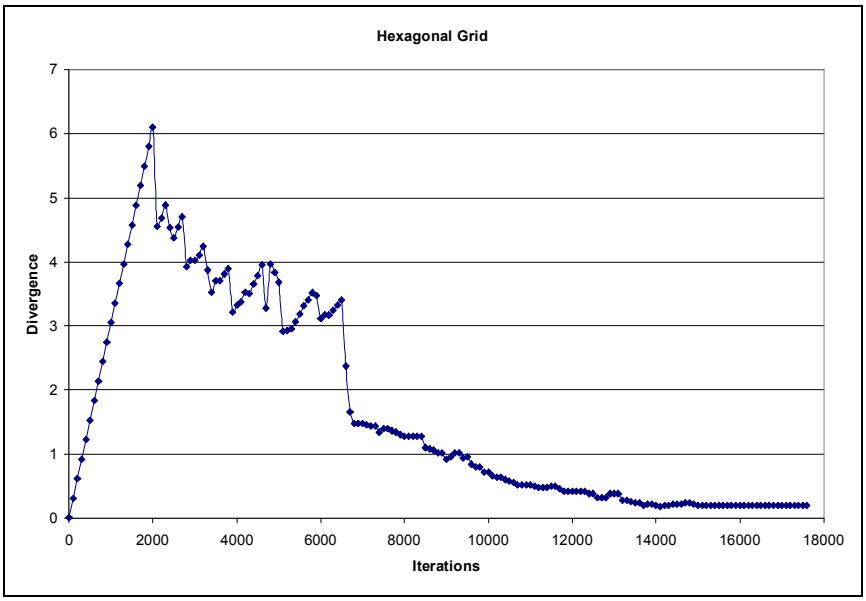

Figure 9 – Hexagonal Grid Results

From these results we can conclude that this algorithm is much more suited to fully connected situations. This is because each node only averages with one neighbor at a time. When you average to one neighbor, it can bring you farther away from a neighbor on the other side. It seems that this algorithm requires nodes to have many mutual neighbors. A hexagonal grid is the minimal requirement for this. A fully connected network is ideal. And a square network (as seen in Figure 10) will fail outright.

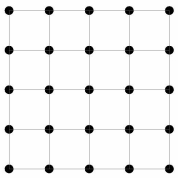

Figure 10 – Square Network with No Mutual Neighbors

To continue testing this algorithm we try starting the synchronization code immediately. This was run on the hexagonal grid above. The results are strikingly similar to the previous run as can be seen in Figure 11. Despite starting with zero clock divergence, the algorithm had trouble converging the clocks. This indicates that the skew does not converge unless there is a large offset in the clocks. This is verified by examining Equation 3. If there is a small difference in the times, the skew will either be multiplied or divided by a number very close to zero. When this occurs, bounds checking will not allow the skew to be changed.

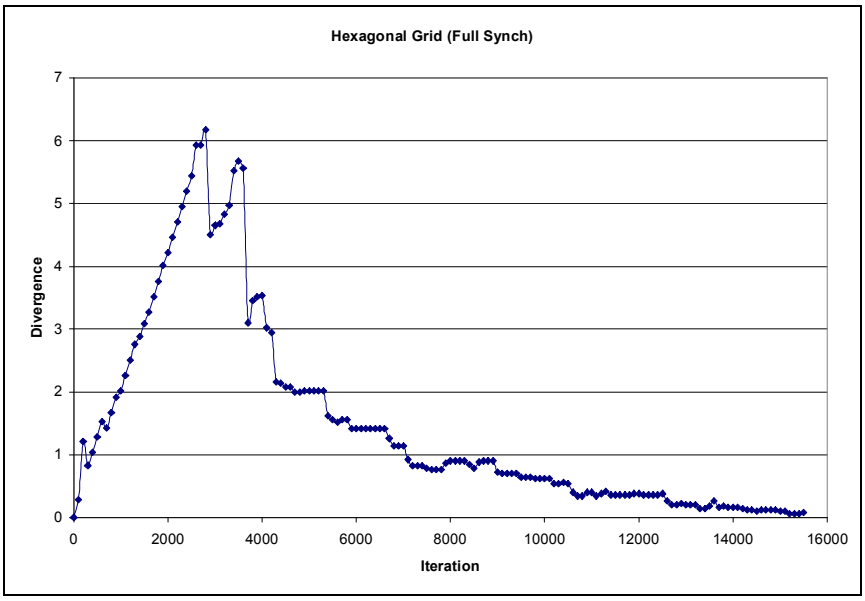

Figure 11 – Hexagonal Grid Results (Full Synch)

Another interesting arrangement that we can use to test this algorithm is the H-Bridge as seen in Figure 12. This network has two fully connected islands connected through a single link.

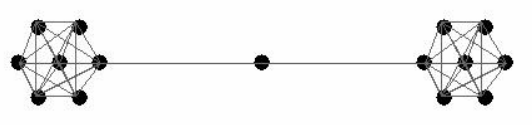

Figure 12 – H-Bride Network

The results show that the two islands converge to be internally consistent very quickly. However, because of the single link between them, the entire system cannot converge. So, with slightly different skews, the two islands diverge. This can be seen in the results in Figure 13. In this experiment the synchronization code was switched on at 2000 iterations, then left on.

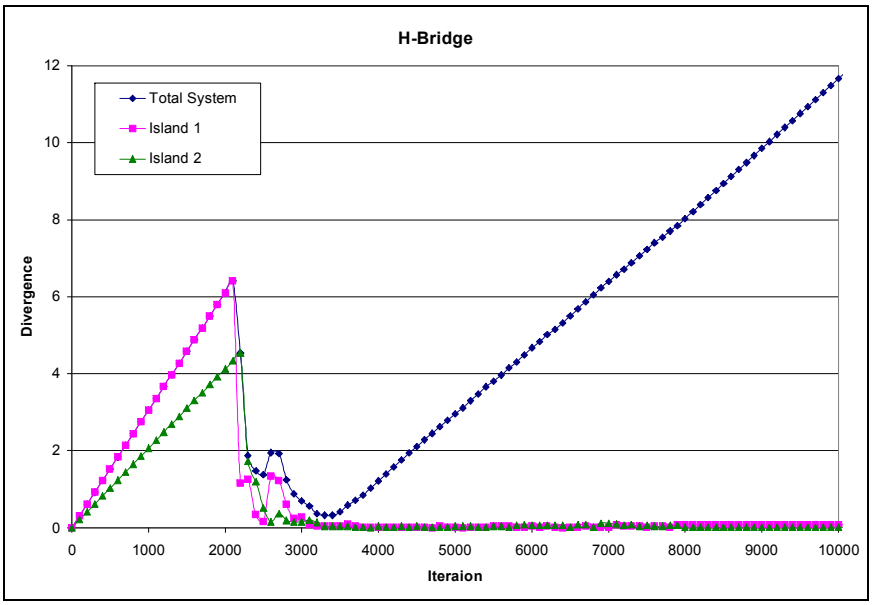

Figure 13 – H-Bridge Results

Extensions

This algorithm seeks to rectify the problem of clock synchronization by averaging all the clocks in the system. While this does not guarantee global "correct" time as one would get with a centralized approach, it is acceptable because the clocks can then be thought of as logical clocks. This is sufficient for many applications. See for example Lamport's logical clocks [6].

It was found that this algorithm only works well in fully connected networks. This may be acceptable for some applications, but most sensor networks do not have this luxury. As mentioned earlier, this algorithm requires nodes to have many mutual neighbors. This limitation is perhaps indicative of all reference broadcast synchronization schemes. Therefore, it should be examined how this algorithm can be improved and extended to work in regular grid type networks.

Perhaps a hybrid scheme is necessary to remedy this problem. In addition to using reference broadcasts, the nodes should also measure round-trip delay and adjust clocks similar to NTP. To improve accuracy, the nodes should only employ this method with the neighbors it suspects are not mutual. For example, only the bridge nodes will employ this scheme in the H-Bridge network.

Another small implementation issue is the fact that this algorithm assumes the nodes have a common start time. While it was shown that some amount of divergence can be handled, the amount this algorithm can handle is unclear. If, for example, there are several minutes between the start up times of the nodes, the convergence time may be unreasonable. Again, it may be possible to hybridize this algorithm with another approach to remedy this. However, more investigation is required.

It would be nice to eventually implement some sort of time synchronization on the GNATs. Whether this can be done with the current hardware and software still remains to be seen. It may be possible that a re-organization of resources will allow finer resolution on the timer and hence the ability to account for tick losses. However, this undertaking is beyond the scope of this project.

This algorithm makes a strong assumption that all nodes in the network are benign. If one of the nodes was faulty either by a crash or malevolent code, it could throw off the entire network. There are many methods for dealing with byzantine faults such as [2] and [10].

With synchronized clocks, many applications open up to the GNATs. They might even be able to be used as a more formal distributed system. If so, they will require fault tolerant operation and discovery, replicated data [5], data fusion, and possibly global state estimation [3].

Conclusion

This paper presented a new simplified version of reference broadcast time synchronization called reference broadcast time averaging. This method is simple enough to be run on the smallest of sensor networks such as the GNATs. It was found that this method works very well for fully connected networks; however the performance degrades as the number of mutual neighbors reduces. This characteristic is probably indicative of all reference broadcast schemes. The algorithm was tested both in simulation and on the actual GNATs. While hardware issues prevented any real testing on the GNATs, the algorithm did run on them, and appeared to work as it should.

Acknowledgements

A special thanks to Keith O'Hara for his help with the significant modifications to the GNAT API necessary for this project.

References

[1] Tucker Balch, *TeamBots Simulator*. http://www.teambots.org

[2] M. Castro, B. Liskov, "*Practical Byzantine Fault Tolerance,*" OSDI, Feb. 1999.

[3] Chandy, M. and Lamport, L., "*Distributed Snapshots: Determining Global States of Distributed Systems*," ACM Trans. on Computer Systems, February 1985.

[4] Jeremy Elson, Lewis Girod, and Deborah Estrin, "*Fine-Grained Network Time Synchronization Using Reference Broadcasts*," OSDI 2002.

[5] Gifford, D., "*Weighted Voting for Replicated Data*," *ACM Symp. on Operating Systems Principles*, December 1979.

[6] L. Lamport. "*Time, clocks, and the ordering of events in a distributed system*." Communications of the ACM, 21(7):558–65, 1978.

[7] Sean Luke, Claudio Cioffi-Revilla, Liviu Panait, and Keith Sullivan. "*MASON: A New Multi-Agent Simulation Toolkit.*" 2004. Proceedings of the 2004 SwarmFest Workshop.

[8] D. Mills, "*Network Time Protocol (Version 3) specification, implementation and analysis*," IETF RFC-1305, March 1992.

[9] Keith J. O'Hara, Daniel B. Walker, and Tucker R. Balch. "*The Gnats – Low-Cost Embedded Networks For Supporting Mobile Robots*", Third International Multi-Robot Systems Workshop. March 2005.

[10] F. B. Schneider. "*A Paradigm for Reliable Clock Synchronization,*" Proc. Advanced Seminar of Local Area Networks, Bandol, France, April 1986, p. 85—104