Final Project Report

Group 3(WLAN)

Table of Contents

Project Objectives:

1. Create a program capable of receiving information from Group 1 on the sensor device that can reliably send to our BaseStation.
2. Create a BaseStation that turns every parallel connection from the sensor to a stream and sends this information to Group 4's server.

Design:

The design was highly collaborative between the different groups in the class, and thus very difficult. There were many ideas as to the best solution for reliable delivery, and for the protocols used. Ultimately, the groups decided to use the simplest solution, where messages were separated in a stream by newlines and where the message contents were composed entirely of ASCII characters. Even after the initial protocol designs were decided upon, further changes and definitions were made. For example, changes were often made as to who would act as client and server in communications between applications.

Because Group 4 did not want an application layer protocol, we had to implement a push back mechanism to the sensor devices from the BaseStation. In the same way that Group 2 made a serial interface to their sensor devices, we implemented a stop-and-wait protocol between the sensors and the BaseStation. The sensor itself has an internal message queue of size 50, which was set at compile time and can be easily changed to the desired queue size. The queue is used in case the sensor doesn't receive acknowledgements from the BaseStation, this same principle also applies if the sensor is in a disconnected state. The BaseStation only sends these acknowledgements when we have an active connection to Group 4's server and the socket has successfully been sent the message. If the connection to Group 4 is down, each sensor handling thread will attempt to resend the pending message every five seconds until it determines that the source sensor connection has died or the connection to group four becomes active. If the sensor connection to the BaseStation goes down, then the sensor will attempt to reconnect to the BaseStation every five seconds. The sensor queue will keep up to 50 messages before dropping from the head to add new ones. When a sensor first connects to the BaseStation, it immediately sends its unique identifier before processing the message queue. Message size was set to 50 bytes, including the new line at the end, leaving 49 characters as the largest message size that can be added to the sensor queue. This value is also set at compile time and can be changed within the source code.

The communication design has one hole in terms of reliable delivery, which is that messages sent across the TCP socket to Group 4 cannot have guaranteed arrival because such ensurability would have to be done on the application layer. We do, however, have reliable delivery between the sensors and the BaseStation. We believe anyone using the connection interface to Group 4 would have a similar problem unless a message acknowledgement protocol is implemented. However, given that we were told in the beginning that the BaseStation and server could be co-located; this problem may be trivial in implementation.

Given the initial description of the sensors, the BaseStation can be best described as a multiple serial stream compactor that produces a single TCP stream. We also used TCP between the sensors and the BaseStation which covers any possible data link layer message anomalies. The BaseStation associates each inbound socket request from the sensors with a unique identifier, which is then stored along with the socket so that reverse mapping can be done. If the unique identifier is already in use, then the older socket connection is clobbered. This behavior was decided given it took so long for a previous socket connection to unbind from the OS. If the BaseStation ever receives a directed message to a sensor identifier that has not been recognized, we respond with the error message "E|<UID>". If the sensor connection is not available and we cannot forward this message, we store it to be sent on the next time it reconnects.

Implementation Approach/Status:

After an initial, fully functional designed Perl BaseStation, we decided to try and do everything in C++ for speed and efficiency. We used two threads on the sensor with one thread per sensor on the BaseStation and semaphores to guarantee that only one thread can send to Group 4 at a time and to secure the data structure storing unique IDs and socket IDs. The first thing we created was the sensor. On the BaseStation, the semaphores are queuing-semaphores, which makes sure each sensor has a fair chance to send its payload in a cyclic fashion to Group 4 if it has data to send. Though the initial design seemed straight forward, we encountered issues when it was first completed. Firstly, we had issues with passing information about the sockets between the functions that used them in the sensor, which also tied into problems with keeping the interfaces serial and the sockets unique. Our progress was further slowed by the fact that the code that works on the sensor does not work with most g++ compilers, and we had to wait and connect to a server the TA provided in order to compile and test.

After the sensor was created, we attempted to make the server with very similar functions and network structure. The issues caused by changing from a single connection to the server handling multiple connections became apparent. A large part of this was that we needed to use multiple threads on the BaseStation. There was a lot of confusion and many errors as we were passing socket data through a thread. We ended up with segmentation faults on the data structures handling message queues for the multiple interfaces and as different calls accessed address space when they should have been accessing the data itself. After a few days straight of the entire team attempting to fix errors, we decided it would be better to recreate the entire BaseStation code. One part of the team worked on another C++ implementation while Joe wrote a Java version. The C++ team got a complete socket and unique id interface done in about the same time that Joe got the majority of the Java version done, so the team switched to the Java version.

We then worked on the problems that were in the Java version and met up with the other teams to implement it. At this point, we also needed to make a change in the way things connect, because we discovered that Group 4 was expecting us to connect to them, unlike what was originally planned and openly documented. At the time of the demo, the program only had three problems. The sensor was adding an extra new line in the first message that registered the sensor with our BaseStation, which was fixed by the time of the demo but was not the version of our module that Group 1 had compiled with. The second problem was obvious given that we had such difficulty getting the sensor to recover properly. It turns out that the

embedded OS environment fails to handle the program properly and reliably if the application startup priority does not list the program last compared to all the other support modules it needs to run. Since the presentation, we have identified this failure and modified the startup script such that this issue is now resolved. The last problem, also now fixed, was that messages sometimes repeated themselves when being sent from a full queue. The way we interpreted the lower bound of the queue was invalid, and is now fixed.

Testing:

General Testing Environment:

The location picked to run these tests was done to simulate a high traffic location with many network and access points running to create a real-world simulation of a deployment environment. All ranges were at least 10 feet and ranged up to 50 feet. Wireshark was run to create charts and graphs while collecting all network activity. iStumbler was used to view signal strength and noise. One gumstix was used for the following test scenarios.

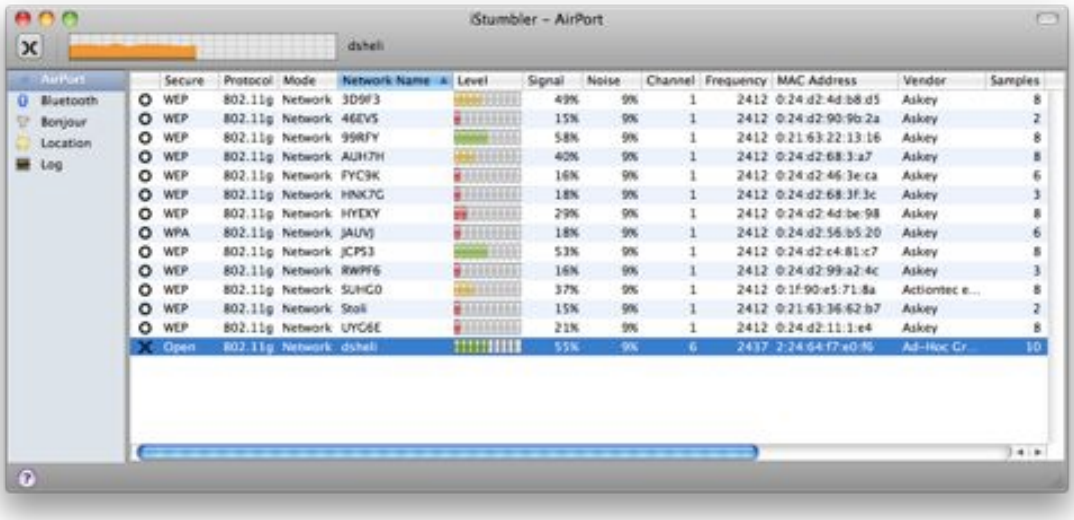

Figure 0: iStumber with Network Activity in Testing Area

**For All Graphs: Red – Traffic From Sensor, Black – Traffic From BaseStation, Green – Overall Traffic*

Test Case 1: Connection from sensor on startup to running BaseStation

Method:

Run BaseStation code and begin test harness on sensor to send messages.

Expected Outcome:

Sensor should auto-connect to BaseStation and begin transmitting messages when a connection is established.

Actual Outcome:

The sensor auto-connected to the BaseStation and began transmitting messages. At 5 seconds you can see a spike in traffic as the sensor is connecting to the BaseStation then the rate of messages levels out. You can barely see the traffic from the sensor because it is nearly equal to the traffic from the BaseStation and covered in the graph.

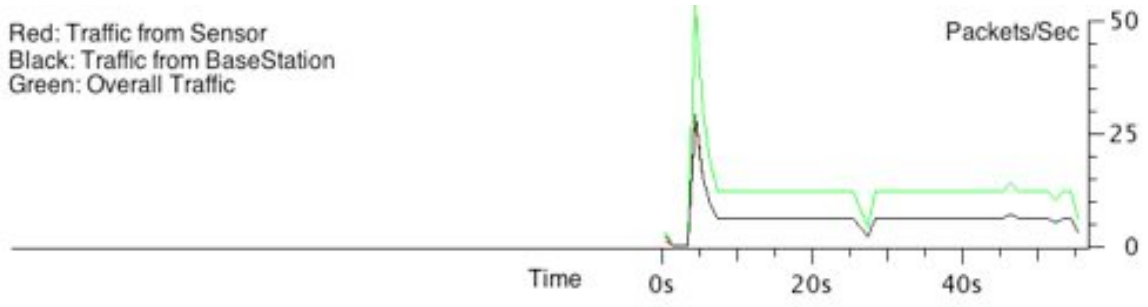

Figure 1: Results of Test Case 1

Conclusion:

This function is working correctly and was duplicated several times.

Test Case 2: Connection of active sensor to starting BaseStation

Method:

Have test harness running on sensor that is actively trying to connect to down BaseStation. Run BaseStation code.

Expected Outcome:

Sensor should connect to BaseStation and begin transmitting its queue until empty.

Actual Outcome:

The sensor auto-connected to the BaseStation, transmitted its queue and no messages were lost.

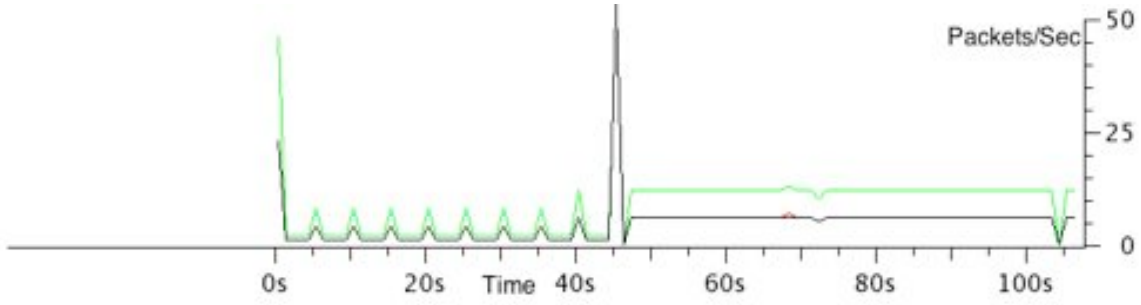

Figure 2: Results of Test Case 2

Here you can see the sensor actively trying to connect to the BaseStation at 5 second intervals. At approximately 50 seconds it connects and pushes its queue to the BaseStation - traffic levels out after this point.

Conclusion:

This function is working correctly and was duplicated several times.

Test Case 3: Simulation of power loss or crash at BaseStation while sensor is running.

Method:

Have test harness running on sensor while BaseStation is running, kill BaseStation process and restart after 20 seconds.

Expected Outcome:

The sensor will reconnect to the BaseStation and continue to transmit messages from when it left off when the BaseStation went down. If the BaseStation is down long enough for the queue to become full, messages may be lost.

Actual Outcome:

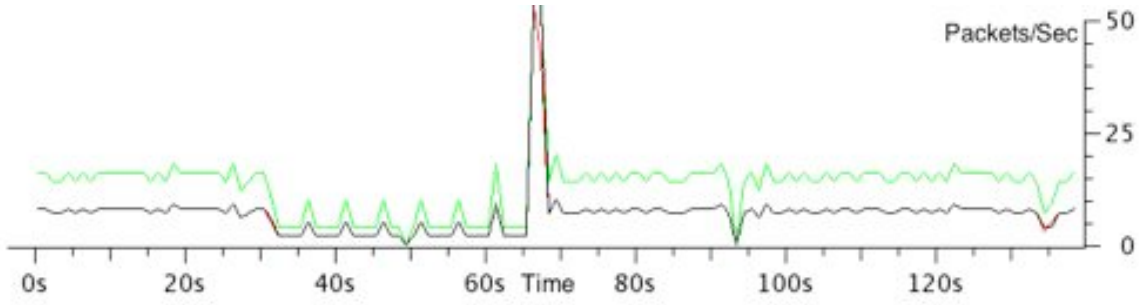

Figure 3: Results of Test Case 3

The sensor was sending messages normally from time 0 seconds to time 30 seconds, the BaseStation disconnected at 30 seconds, and came back at time 65 seconds. Meanwhile the sensor tries to reconnect every five seconds, and finally reconnects to the BaseStation and dumps its queue at time 65 seconds. After that, the sensor is operating normally.

Conclusion:

This function is working correctly and was duplicated several times.

Test Case 4: Simulation of power loss at gumstix and reconnection to running BaseStation

Method:

Have BaseStation running and sensor actively sending messages. Unplug sensor, reconnect and wait for auto-startup script to run.

Expected Outcome:

Sensor should auto-connect to BaseStation and begin sending messages.

Actual Outcome:

Sensor would not start up at current startup priority. Had to change to priority to 70 from 80 and reproduce test. After this change there were no problems.

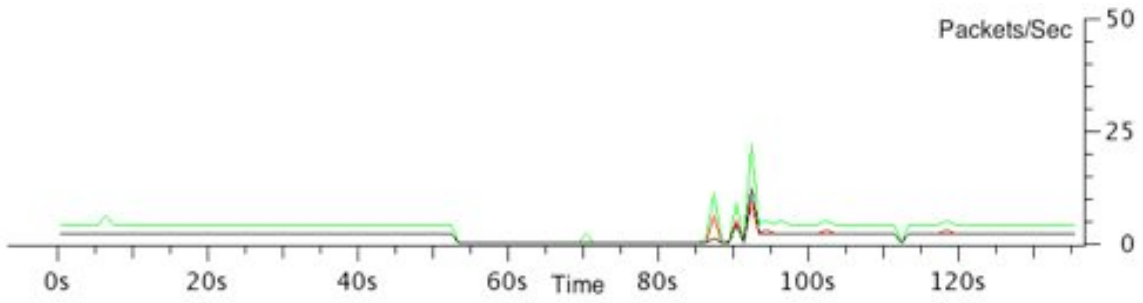

Figure 4: Results of Test Case 4

Power loss occured at 55 seconds for 15 seconds, and recovery at 90 seconds

Conclusion:

This function is working correctly and was duplicated several times.

Test Case 5: Simulation of heavy network utilization and arrivals at sensor

Method:
Connect another device to the sensor to send a high rate of traffic. We used a message rate of 100 messages per second of 1024-byte messages to the sensor. The message frequency from the sensor was 1 message/second.

Expected Outcome:

Messages should still be arriving to the subscriber.

Actual Outcome:

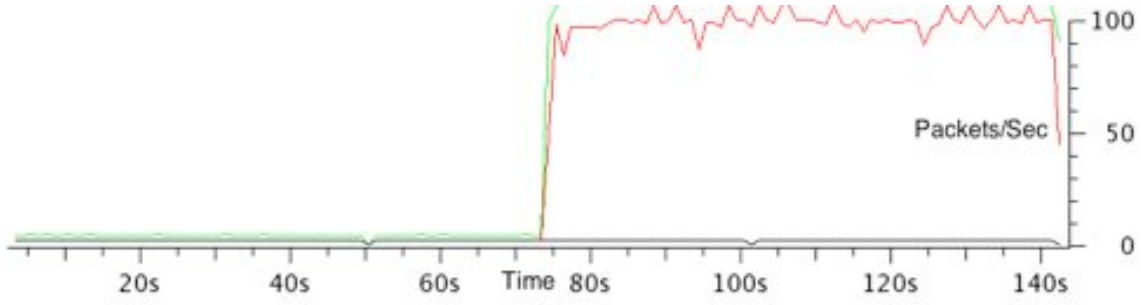

Figure 5: Results of Test Case 5

Messages continue to arrive to the subscriber at the same rate as before. If there was a difference in rates, it was negligible.

Conclusion:

This function is working correctly and was duplicated several times. As the congestion increases, then a higher delay between the sensor and subscriber might be seen, which could cause the sensor's queue to fill up and messages to be dropped.

Test Case 6: Simulation of total network outage.

Method: When the program is running, disconnect network interface on BaseStation, wait appropriate amount of time and reconnect network interface.

Expected Outcome:

All connections should be reestablished and messages should continue to be delivered.

Actual Outcome: Connections were reestablished and messages continued to be delivered. The queue functioned properly and was transmitted once the network came back online.

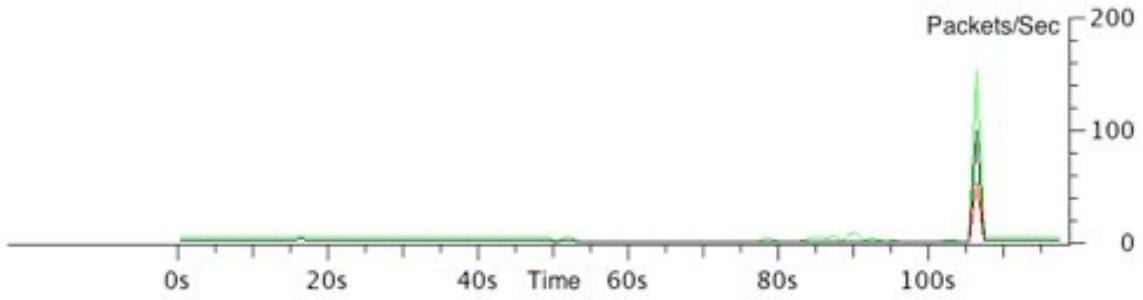

Figure 6: Results of Test Case 6

In the graph you can see messages were being transmitted and at 55 seconds the network interface went down, at approximately 105 seconds the connections were reestablished and all message from the buffer were sent. No messages were lost.

Conclusion:

This function is working correctly and was duplicated several times.

Test Case 7: Maximum messages from sensor

Method:

The BaseStation should be up and the sensor should be sending messages to it. Message will be enqueued at a rate of 1 message/second and increased in an order of magnitude until messages are lost or a higher rate cannot be achieved.

Expected Outcome:

The BaseStation and sensor should continue to be exchanging messages.

Actual Outcome:

We experienced a hardware bottleneck in the speed in which messages could be queued, we were not able to create a situation in which the queue was filled up solely from the device or messages were dropped in transmission. This appeared to be approximately 200 messages/second.

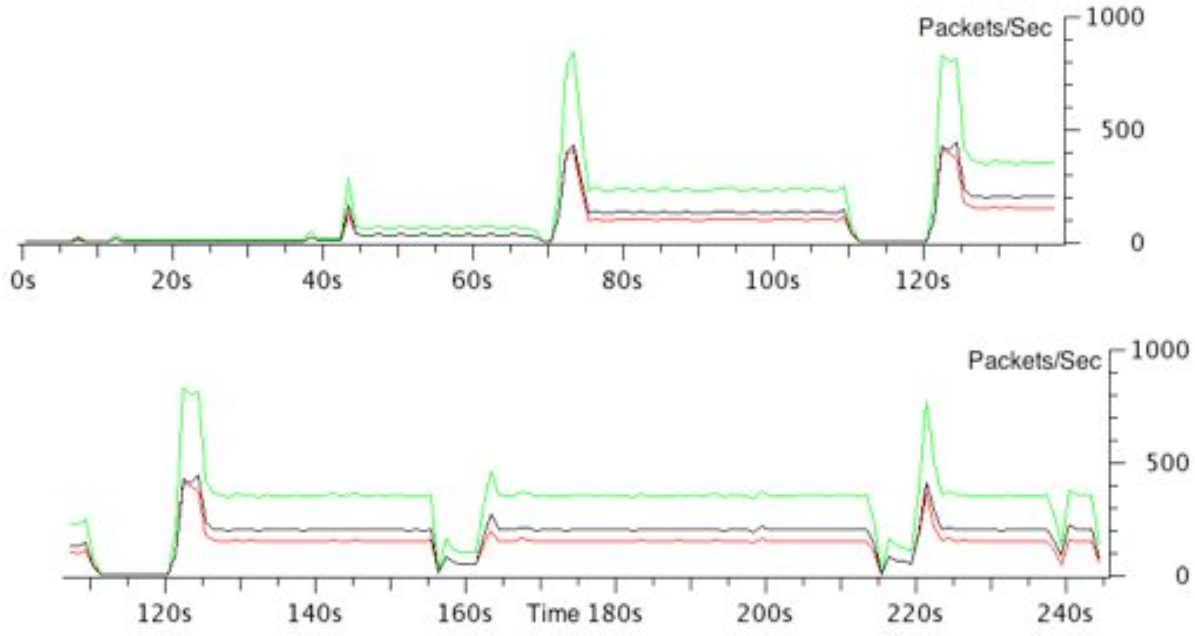

Figure 7: Results of Test Case 7

Conclusion:

This function is working correctly and was duplicated several times.

Test Case 8: Filling up the queue

Method:

We start the sensor and let it fill up its queue. Then we start the BaseStation so the sensor dumps its queue to the BaseStation.

Expected Outcome:

The BaseStation will receive the queued messages with no duplicates. The oldest messages the sensor generates will be lost because of the queue being full and overwriting them.

Actual Outcome:

We were able to fill up the queue and properly drop extra messages. After reconnection all messages from the queue are sent to the BaseStation.

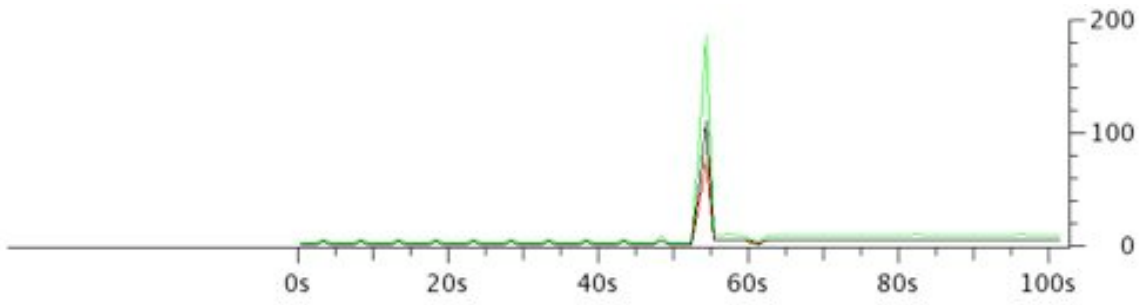

Figure 8: Results of Test Case 8

Conclusion:

This function is working correctly and was duplicated several times.

Test Case 9: Multiple sensors

Method:

Run multiple copies of the program on the gumstix with different MAC addresses

Expected Outcome:

The BaseStation will receive the messages from the gumstix as coming from multiple sensors

Actual Outcome:

The BaseStation received messages from two different MAC addresses when we ran two copies of the program on the gumstix

Conclusion:

The BaseStation works as expected when multiple sensors are connected. This was duplicated several times.

Test Case 10: Sending messages to sensors

Method:

Run multiple copies of the program on the gumstix with different MAC addresses. Send valid messages to each sensor to change their modes.

Expected Outcome:

Each sensor will receive messages from the BaseStation as expected.

Actual Outcome:

The mode of the sensor 00:00:00:00:00 was set by the server to aggregate mode with 10 second value and the mode of the sensor 00:00:00:00:01 was set by the server to immediate mode. Both sensors received these messages appropriately and displayed appropriate modes.

Conclusion:

Sending messages to sensors from the server works as expected.

Test Case 10: Microwave Interference / Signal Strength

Method:

Attempt to create enough signal interference to get connection to drop. The interval of messages was 1 message/second.

Expected Outcome:

If signal drops, when signal reconnects messages in queue will be transmitted. If signal is disconnected for long period of time or is poor enough to slow the network speed, message will drop from queue.

Actual Outcome:

We created various scenarios to attempt create signal interference.

Scenario 1: Near Microwave

We hung the sensor and antenna from the microwave handle and heated a bowl of water with the microwave. There was no loss in messages or speed. Signal strength on the utility iStumbler was unchanged.

Figure 10a: Sensor Hanging From Microwave Door with Microwave On

Scenario 2: In Microwave

Sensor was placed inside the microwave in an attempt to use its shielding to block our signal. Signal strength on the utility iStumbler was slightly reduced, however there was no loss in messages or speed.

Figure 10b: Sensor inside of Microwave with Microwave Off

Scenario 3: In the Refrigerator

Sensor was placed inside a refrigerator in an attempt to use its shielding to block our signal. There was a reduction in signal strength, approximately 30% down from 60% original. However, no message loss or speed reduction was recorded.

Figure 10c: Power cable running into Refrigerator

Scenario 4: Cell Phone Interference

Four 3G-cell phones with active calls were places directly on top of the sensor's antenna in an attempt to create interference. There was no change in signal strength, no message loss or speed reduction.

Figure 10d: Four Cell Phone Dogpile on Sensor

Scenario 5: Cell Phone Interference + Microwave Shielding

Four 3G-cell phones with active calls were placed directly on top of the sensor's antenna and placed inside of a microwave. The microwave was not turned on. There was no change in signal strength, no packet loss or speed reduction.

Figure 10e: Four Cell Phone Dogpile inside Microwave with Microwave Off

Conclusion:

It is very difficult to create enough interference to cause our system to run slow enough to buffer messages. Neither cell phones, microwaves, nor placing the sensor inside a refrigerator caused a noticeable decrease in network speed from the sensor to the BaseStation.

It appears that on modern microwaves, the shielding is sufficient for our wireless frequency to not cause interference, but we were not able to run tests using older microwaves that may have worse shielding. Further testing is suggested in that case.

Overall Conclusions:

Our test cases cover all of the functionality of our program and all of the ways our program is able to recover from failure. Since it passed all of them, we conclude that it is working as expected.

Our testing environment simulates a real world environment where the gumstix would be deployed because many wireless access points are present. Additionally, on test case 10, we

made the testing environment worse than would be encountered in the real world and our program was still able to function correctly.

Conclusion:

Though our program now meets specifications, the presentation was weak and disjointed. The initial set-up in the room was made on computers other than those we originally integrated on, and we had very little time to start our own part of the system. After the other groups did their parts of the presentation, we were finally allowed to use the computers and we quickly set-up our part. We successfully demonstrated all asked parts of the program, and everything runs very well and looks very clean. In terms of lessons we learned, we learned that it would be better in the future to establish a leader more definitely and early in the process. We kept constant communication, which worked very well due to the regular updates we received from one another via Google Groups™. When we realized that our project was in jeopardy, we made it our top priority and spent all available time to rectify the situation.

Group Members:

-Joseph Copenhaver, jpc062000

-Ruben Gabriel Cavazos, rgc061000

-Aaron Leonard, aml066000

-Samuel Young, say051000

-Brent Frost, bmf061000