Hydrogen Car Mobile Display

Andrew Schulze

Course Instructor:

Dr. Guilherme DeSouza, PhD

ECE 4220 Project Report

Department of Electrical and Computer Engineering

University of Missouri – Columbia

December 2010

Abstract

This project is utilizes an Android application to satisfy another approach for remote monitoring of operational hydrogen car metrics. To accomplish this all relevant information from the hydrogen car is relayed to a web server where it is stored in a HTML file. This operates independently of the mobile application as it retrieves these metrics on its own and displays the changes on-screen for the user.

Introduction

This project is composed of two parts: an additional module to a Base Station application that receives performance metrics from the Mizzou Hydrogen Car and an Android mobile application that displays the metrics. The performance metrics of interest for this project were the car's speed, fuel efficiency, and error codes produced by the fuel cell. When data is received by the Base Station application, the additional module created for this project saves it to a web server. It is then the job of the Android application to retrieve these recent performance metrics and display it for the user. The two parts mentioned above do not require interaction to function correctly. The metrics may be constantly updated to a web server while the Android application is not attempting to retrieve any data. Likewise, the Android application may attempt retrievals from the web server even if the Base Station has not received any new metrics for a significant period of time.

Problem Description

This project aimed to achieve another method for the Mizzou Hydrogen Car team to remotely monitor car performance. The Base Station is the current method for remote monitoring and

effectively facilitates this project to give more monitoring ability to the team. The team may find having the car's performance metrics available helpful on a mobile phone if they chose to place members around the race track to assist the team with possible issues or fine-tuning runs around the track. To achieve this additional method, it therein created new problems regarding the implementation. Firstly, it was necessary to find a way to relay data from the Base Station to a mobile phone application and integrate it into the current Base Station implementation. Secondly, to continuously show performance metric updates on a mobile phone, the creation of a mobile application was necessary with the ability to stop retrieving if desired by the user.

Proposed Approach

A visual model displaying the approach taken for the project is seen below in Figure 1 Block Diagram. The approach is comprised of multiple components indicated by the Base Station Application and Mobile Phone, including interfacing with a web server. Each component's proposed approach will be described in detail in the following sections of this report.

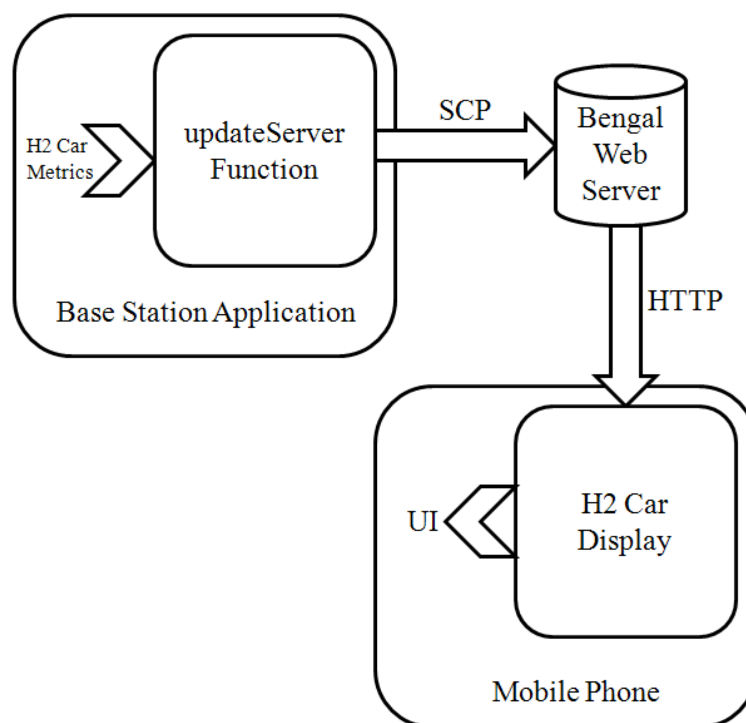

Figure 1 Block Diagram

Base Station

To accomplish a way to relay information received from the hydrogen car at the Base Station application, a new function called "updateServer" (and an accompanying function called "LaunchCommandLineApp" that serves as a worker for updateServer) was developed to work in this application that writes performance metrics to an HTML file and copies it to a web server. As mentioned previously, performance metrics of interest for this project were the car's speed, fuel efficiency, and error codes produced by the fuel cell. Secure Copy (SCP) protocol was used as a transfer method for storing the file on a server for its security based on the Secure Shell (SSH) protocol, and its relative ease of set up and usage. Security may be of concern to the team to prevent other hydrogen car teams from intercepting performance metrics of the car and to remain competitive. Because a SCP call to this specific server (bengal.missouri.edu) requires a

user to enter their password when trying to copy a file, a public/private key pair was set up to prevent a password prompt every time. There were numerous SCP protocol ports for the C# language, however having luck with the PSCP software (a command-line SCP from PuTTY) steered my resulting implementation towards forking off a new process to call PSCP.

updateServer Function

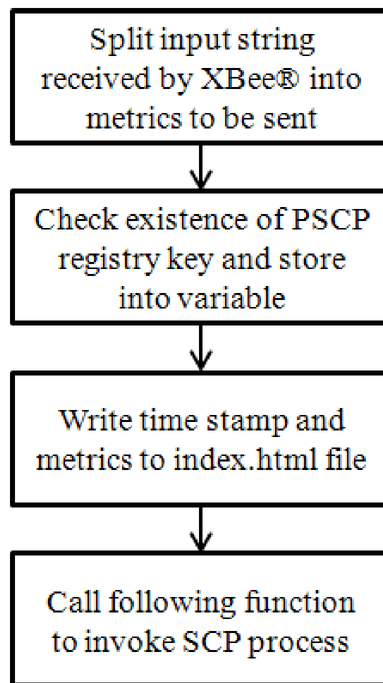

Figure 2 updateServer Function Flow Chart

The program flow for the updateServer function is seen above in Figure 2. As this function is part of the Base Station application which receives performance metrics directly from the hydrogen car, it must parse the incoming data to obtain the speed, fuel efficiency, and error codes. Following this, it must check if the PSCP registry key is stored on the user's operating system. PSCP will prompt the user if they would like to store this key on their computer if this software had not been run before. This posed an obstacle as it would freeze the transfer waiting

for input from the user. The desired implementation would show no command window to the user when these transfers occur. Therefore, after checking if the key is stored on the user's computer prior to a transfer, it will toggle a variable used later in LaunchCommandLineApp when handling this input request. The most important part of this function is generating the file to be stored on the server. Creating an HTML file named "index.html" in the "www" directory would allow the file to be viewable from a web browser at the URL address on the server. This was desired as the mobile application would use this URL to read the text off the webpage via HTTP, as explained later in the Mobile Application section. A timestamp, speed, fuel efficiency, and error code were stored as a comma-delimited string in the HTML file to allow easy parsing for the mobile application. The timestamp is generated within this function to signify to the mobile application when the data was received. Finally, the LaunchCommandLineApp function is invoked to do the actual SCP transfer.

LaunchCommandLineApp Function

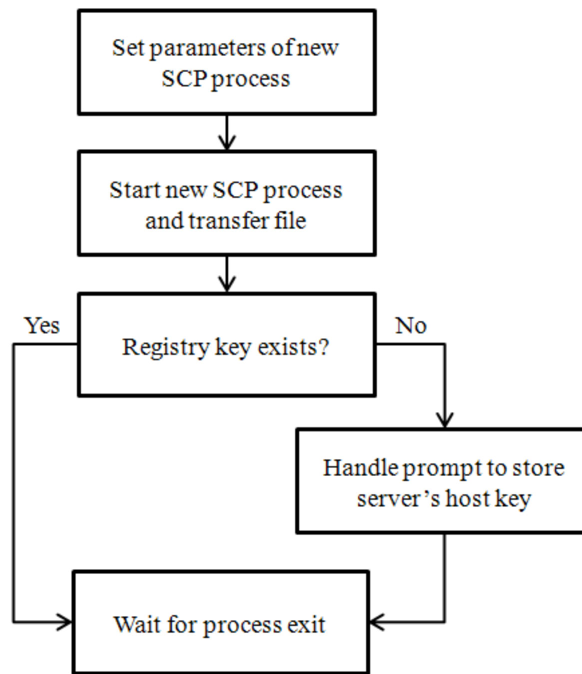

Figure 3 LaunchCommandLineApp Function Flow Chart

The program flow for the LaunchCommandLineApp function is seen above in Figure 3. This function effectively sets up how this SCP process will run and execute it. The parameters it sets are part of a ProcessStartInfo object within the C# language that allow one to specify important aspects such as the .exe to run, its command-line arguments, whether or not to show a window when creating the process, and redirecting Standard I/O. In this realization, the .exe was pointed to the PSCP executable, and the command-line arguments contained the private key, index.html file, and destination on the web server (" -i key,ppk index.html pawprint@bengal.missouri.edu:www/”). As mentioned before, to produce a seamless transfer to the user running the Base Station application, it was desired to show no command-line window. Standard Input was redirected so that the registry key prompt could be handled. The process was

then started with these parameters. If the registry key variable stored earlier was not true, this indicated that the key did not exist on the user's computer and the prompt would occur. In this case, it was necessary to write a "y" to the console's Standard Input telling it to save the registry key, preventing any further prompts. This function then waited for the transfer to finish before continuing its next operations. The registry key variable needed to be updated so that nothing more would be written to Standard Input during following transfers so the existence of the registry key was checked again as a precaution. If storage of the registry key was successful, then the registry key variable was toggled so that following transfers would only just wait for the transfer to finish. If the registry had not been stored, nothing changed and any following transfers would attempt to store the registry key again as there would be more following prompts to store it. Provided that the registry key storage was successful on the user's computer, following SCP transfers would occur and wait for the transfer to finish.

Mobile Application

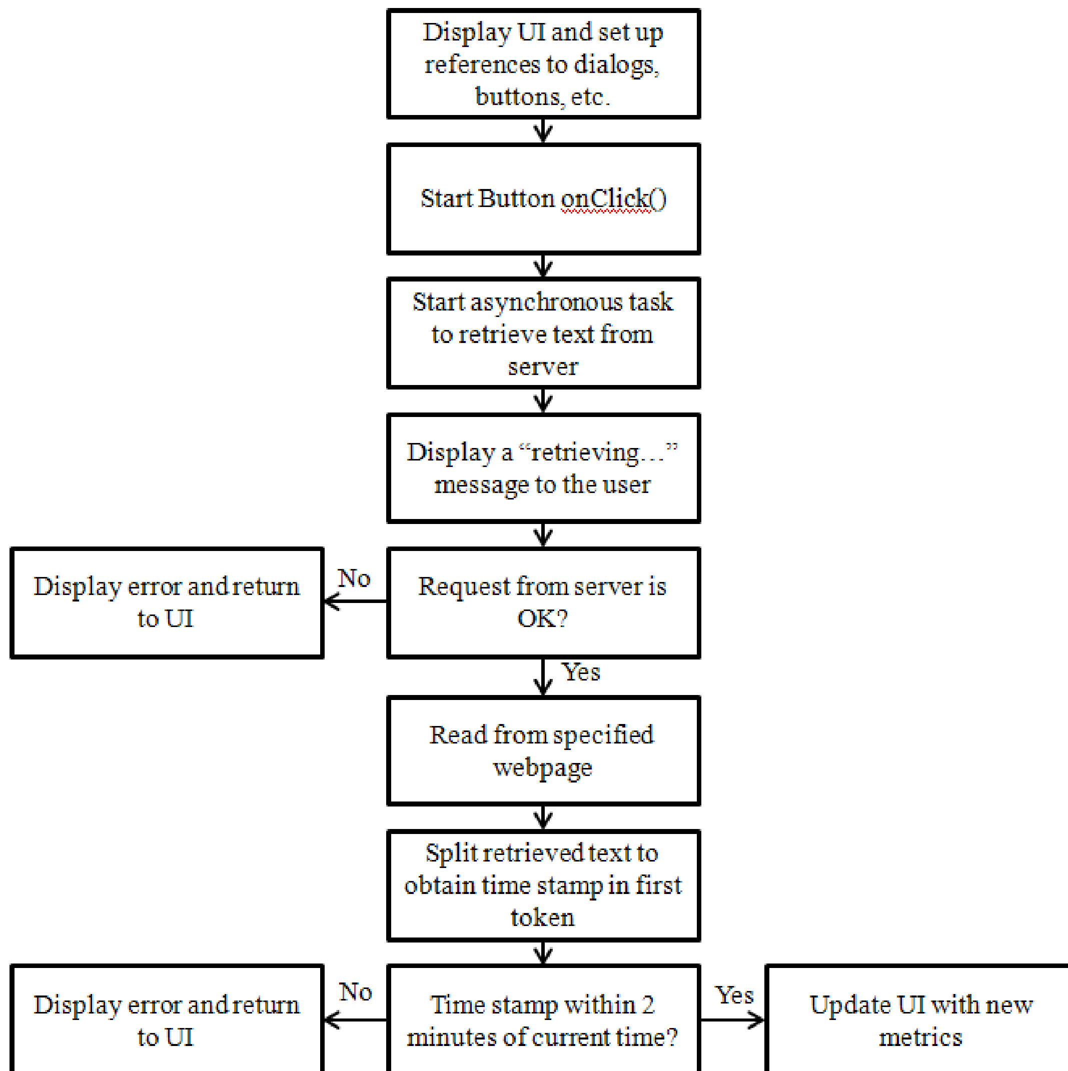

Figure 4 Mobile Application Flow Chart

The program flow for the Android mobile application is seen above in Figure 4. This component of the project was developed using the Android Software Development Kit (SDK). It provides an emulator allowing one to test any Android applications in an Android OS environment (used Android OS version 2.2). The main components of an Android application are the .java file that does any computations and the file named "main.xml" that one can use to design the user

interface (UI). The elements coded into main.xml are displayed below in Figure 5. It was set up with a start button, and four “TextViews” that are used to display the performance metrics. As noted by each file’s file extensions, their respective languages for programming are as such. The following sections will describe how this application is initialized and the following retrieval operations executed within an asynchronous task.

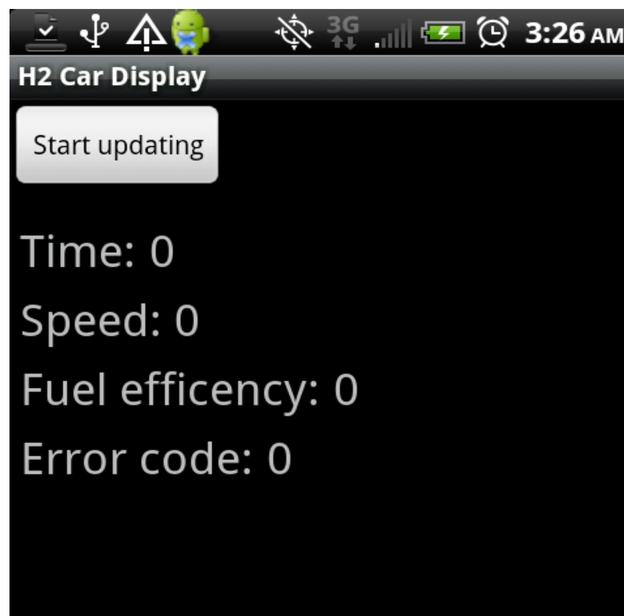

Figure 5 Application Start View

Initialization

The function common to all Android applications is called “onCreate”. As noted by its name, the commands within its body happen once the activity (essentially a task) is created (effectively the “main” function of a program). The main actions taken following this are setting the UI and creating references to the start button and dialog boxes. The function “setContentView” sets the look of the UI of the user by providing it with the parameter “R.layout.main”. “Main” in this parameter is actually the main.xml mentioned previously. The R (short for res/resources) and

layout elements are directories within this project's folder structure (inside Eclipse workspace directory: H2CarDisplay/res/layout/main.xml). A string is also created that contains the URL used later when accessing the web server.

User Interface Initalization

A UI does not necessarily need to be defined within the main.xml file. It is possible to create buttons and dialogs within the .java file by creating button and dialog objects and modifying their properties. However, most Android developers favor using the main.xml file as it provides more flexibility for more professional-looking layouts. This is worth mentioning as the former method was how an alert dialog was created for this program. This dialog will be used later when relaying error messages to the user. An AlertDialog object is created and modified to use an "onClickListener". This function allows the program to take action once the button on the dialog has been clicked. For the implementation of this program, the only action taken was to dismiss the dialog from view as its only purpose was to alert the user of a problem.

Start Button

Similar to the alert dialog, a button object is created in the same way, but provides reference (R.id.start) for modification to the start button defined in main.xml. The start button again uses an onClickListener to take action when it is clicked. The main purpose of this button click is to start an asynchronous task that will perform retrieval from the web server and display the metrics on the UI. For this reason, it was necessary to disable the button to prevent any further clicks, effectively preventing creation of more tasks doing the same process. An asynchronous task is executed simply using the command "new RetrieveHTML().execute(URL)" where RetrieveHTML is the name of the task and URL is the string defined earlier.

AsyncTask Initalization

Once called by the start button click, execution goes to the class declaration for the task, RetrieveHTML. During declaration, it is necessary to make sure it extends the AsyncTask class (Android's implementation of an asynchronous task). It is also possible to set the input parameters of the defined functions within an AsyncTask class by setting parameters as one does normally ("RetrieveHTML extends AsyncTask<String, String, String[]>"). Android has defined an AsyncTask such that the only functions that may be used within it are "onPreExecute", "doInBackground", "onProgressUpdate", and "onPostExecute". The input parameters mentioned correspond with doInBackground, onProgressUpdate, and onPostExecute respectively. When the execute(URL) command is called earlier, it passes the string to doInBackground so it is necessary that the input parameter for this function be defined as a string to prevent compilation errors. The reason for using an AsyncTask was such that an "AsyncTask enables proper and easy use of the UI thread. This class allows to perform background operations and publish results on the UI thread without having to manipulate threads and/or handlers." [1]

As is common with any classes within a program, there was necessary initialization within RetrieveHTML as well. A ProgressDialog object was created and used often to indicate to the user that the application is performing some sort of operation. Similar to the button and dialog objects created before, any text in the UI is called a "TextView" object. By providing a reference to the ID of a TextView (R.id.speed) defined in main.xml, one can modify the text at any point thereon. Four TextView objects were created with reference to the timestamp, speed, fuel efficiency, and error code texts.

The function onPreExecute is as the name describes. It performs any tasks on the UI thread before the background computations are done in doInBackground. The only action taken within onPreExecute was the ProgressDialog was shown with a message "Retrieving HTTP data.." to the user indicating forthcoming retrieval from the web server.

Background Thread

Once onPreExecute finishes, doInBackground function starts. Because its input parameter was defined as a string, it is able to access the URL passed to it when the AsyncTask was executed. The main purpose of this function is to read the text from the URL on the server, parse it into the performance metrics, and pass these metrics to the UI thread. It is important to note that this function cannot update the UI thread directly as doInBackground is actually within another thread separate of the UI thread. There is a way to pass messages between threads called the "publishProgress" command. When this command is called, it passes whatever information is provided to it (defined by the parameters of onProgressUpdate – in this case, a string array) to the UI thread.

From here on, doInBackground performs access to the URL and reads the text from the webpage. Throughout this process, if any errors are encountered such as if the status code from HTTP response is not "OK" or the string read from the webpage is null, the publishProgress command is used to signify to onProgressUpdate that there was an error (the type of update is handled within that function, explained in more detail below). Because the metrics were stored in a comma-delimited string on the webpage, they must be split to obtain each string separately. The first part of that string is the timestamp, and is important for this implementation as it determines how recent the Base Station had been updated with metrics. Within this proposed

approach, if the current time minus the received timestamp was less than two minutes, then the data is valid and it is updated on the UI. If the difference is greater or equal to two minutes, the data is not recent enough to be deemed effective for the team and instead the UI will be updated via publishProgress with an error signifying this occurrence.

During/Post Background Execution

The onProgressUpdate function is what is running in the UI thread while the background thread is performing a HTTP retrieval and parsing the text. As mentioned previously, the string array it takes in is used to update the UI differently based on the type of update sent from doInBackground. The first index of the string array contains an “integer” (written as a string) and the second is a string. Based on the value of this integer, this function’s purpose is to decide how the UI will be updated. A switch/case statement is used to accomplish this. If the value is of the integer was a 0, then there was an error in the background thread and the error message in the second index is written to the alert dialog. All other integer values do not indicate error and are only used to update the TextViews (timestamp, speed, fuel efficiency, and error code). This is accomplished by using the “setText(string[1])” command upon a TextView object, i.e. “speed.setText(string[1)” would cause the speed TextView to change to a new speed value.

Following doInBackground finishing its execution, focus is again returned to the UI thread via the onPostExecute function. Because the start button had previously been disabled to prevent the AsyncTask from being called multiple times, it is now re-enabled to allow further button presses and HTTP retrievals.

Results

Base Station

The results from the new module within the Base Station application are that the HTML file is copied to the web server with the recent performance metrics from the hydrogen car. This is the original intention of this module. Due to the way the HTML was written to the server, i.e. the index.html file in the www directory, it allows one to verify these results by simply navigating a web browser to the destination URL and viewing the metrics. The Base Station is updated at least every 300 ms (due to the implementation of the software modules running on the car) so the HTML file will be rewritten and uploaded at roughly this frequency as well, given some extra leeway time due to computations within BaseStation. If the web server was more instantaneous with updates to the server, this frequency of change would be noticeable in the web browser if one refreshes the web page. However, due to any extra time taken on the server-end to populate the new web page, a slower frequency may be observed. Nonetheless, the timestamp stored in the HTML file is not affected and the accuracy of the metrics data is still intact.

Mobile Application

The results of the Android application show that it can retrieve the performance metrics from the server and display it on the UI based on user input (start button click). Prior to clicking the start button, the look of the UI is seen again in Figure 5. Once clicking the button, the UI is updated with the progress dialog executed in preExecute seen in Figure 6.

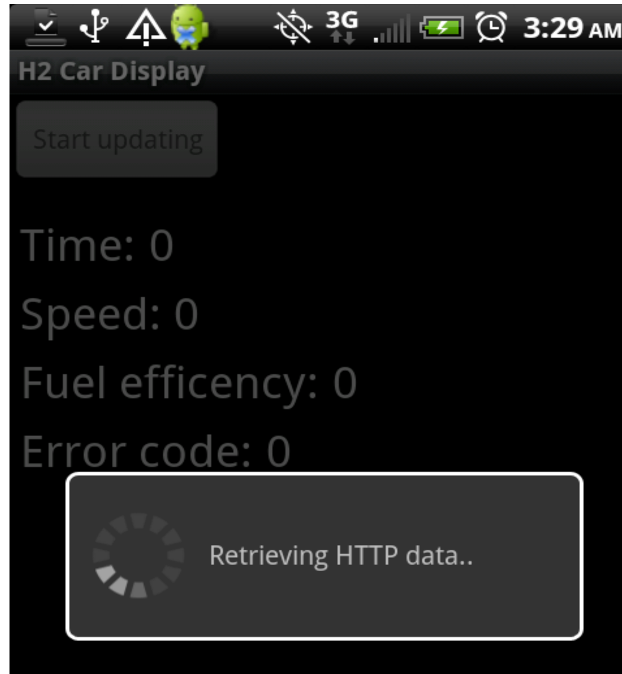

Figure 6 Application Retrieving View

If there is an error retrieving data from the server, an alert dialog will be displayed to the user indicating the error. The UI is displayed in Figure 7.

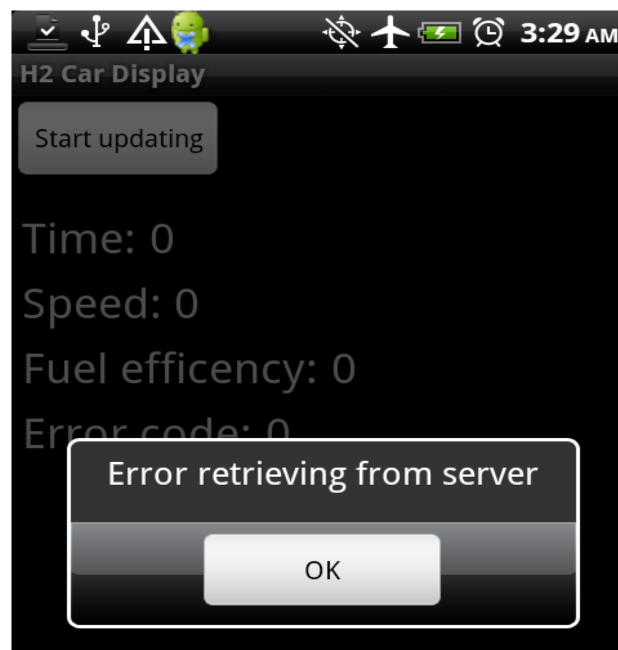

Figure 7 Application Error View

If the timestamp difference is greater than two minutes, the results are displayed in Figure 8.

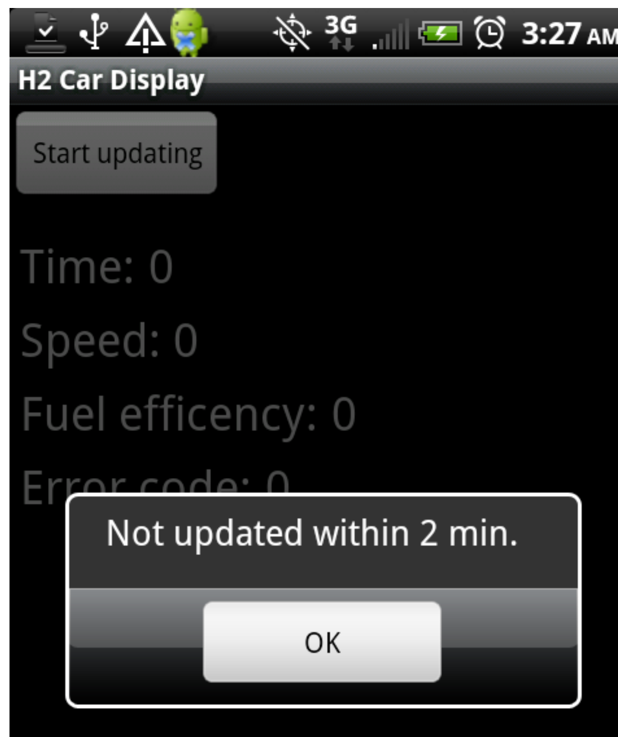

Figure 8 Application Dated Timestamp View

Lastly, if the application is able to retrieve data from the server and the timestamp difference is less than two minutes, the retrieved performance metrics of the hydrogen car are updated in the TextViews. This result is seen in Figure 9.

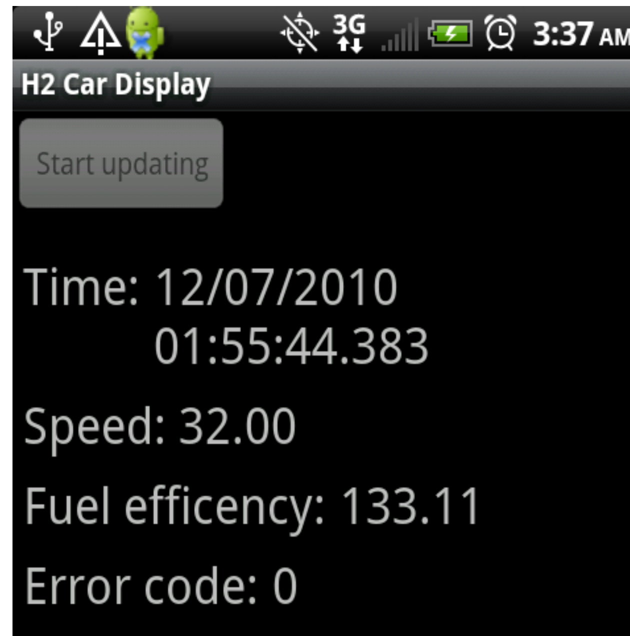

Figure 9 Application Post-Update View

Additionally, the results of the Android application may be viewed in the supplementary videos.

One problem mentioned in the Problem Description above that was not successful was to continuously refresh updates in the mobile application with the ability to stop refreshes via a stop button. However, because developing on the Android OS was a learning process, there was knowledge gained in regards to best practices for the usage of an asynchronous task. During the development process, a stop button had been implemented but its functionality was broken. Due to the scope of the onClickListener created for the start button, another onClickListener for another button may not be created within its parent function. If this was possible, child onClickListener would have access to the "cancel" function used to kill the AsyncTask. Creating another onClickListener outside of the start button creation obviously will not provide the function access to the cancel method private to another class (essentially a thread private to that class). If the stop button were functional, the code is already set up to allow continuously retrieve

from the server (with a five second sleep time allowing the server to repopulate if necessary). In its current implementation without the stop button, the application can retrieve continuously but it will never stop until the actual program is force quit by the operating system. The resulting affirmation based on development using an asynchronous task within Android is that an AsyncTask is more suited to making sure the UI thread stays responsive for the user, rather than performing successive computations. This was evident in the four available functions within the AsyncTask class that split up executions between the UI thread and a background thread. The reasons for this are obvious as all the user interacts with is the UI thread and any unresponsiveness is seen as the application performing poorly. In the end, the implementation was left as a button press returning metrics to the UI.

Conclusions and Further Work

To conclude the work done in this project, it should be noted that the new Base Station modules assist updates to the server via SCP when data is received. The Android application retrieves information from the web server via HTTP and displays the hydrogen car's performance metrics on-screen. In order to further the functionality and usability, there are some additions/changes I think would benefit this project. Rather than using text labels to display the metrics, analog gauges would be used instead allowing the user to reduce interpretation time of the metric values. An apparent change to be made to implementation of the mobile application is to change the AsyncTask threading approach to an approach that will allow one to stop refreshing metric data. A proposed approach would be to use the Handler/Looper threading model. Reasons for using a handler are such that the "main uses for a Handler [are]: (1) to schedule messages and runnables to be executed as some point in the future; and (2) to enqueue an action to be

performed on a different thread than your own." [2] The handler would interact with the UI thread, and the "different thread" would be performing HTTP retrievals. The scheduled messages would be passed to the looper used as a message passing method between threads. In this implementation, if the stop button onClickListener registers a click, a message could be passed to the looper to quit the thread performing the retrievals.

References

[1] Android. (2010) Android Developers. [Online].
http://developer.android.com/reference/android/os/AsyncTask.html

[2] Android. (2010) Android Developers. [Online].
http://developer.android.com/reference/android/os/Looper.html