CE-4020-004

Team Omicron

Milestone 3 Cycle Report

Submitted April 14th, 2019

# Executive Summary

LogiSteps is a full stack application that is designed to collect, process, and display user fitness data in a seamless, self-powered construct. LogiSteps allows a user to pair their Bluetooth enabled smart sole with their mobile device and stream data to the cloud in a manner that is unobtrusive and relies very little on the user. Prior to beginning spring quarter, four milestones were set, and since then, four of four have been successfully completed. The completion of this third milestone, and hence, third development cycle, represents the full-stack operability of the Logisteps platform, with data originating at the smartsole, and ending up in the Google Cloud web application. This added functionality marks a major milestone in the project, with the ability for users generate step data and view it in the cloud. This milestone is a culmination of all previous milestone and would not be possible in the absence of any individual milestone.

# Introduction

The Logisteps project has completed its sixth week of development in the final quarter of development. The major goals of this quarter are completion of system integration, which includes physical construct to sensors, sensors to microcontroller (both data and power), microcontroller to Android application, Android application to web server, and web server to user. Logisteps initially began the quarter on track, but fell slightly behind in week 3, and has since then adjusted and recovered lost time that occurred in week 3. The main focus for the project during this development cycle (week 4 – week 6) was the completion of full stack data transmission, and world-wide online interaction. Online accessibility of the web application is documented in a separate milestone (4) and will not be documented in depth in this milestone report.

The remainder of this document will go into further detail regarding risks to the project, as well as possible contingency plans that are ready to be implemented if needed, overall performance of the team regarding project plan, the deliverables created for this third development cycle, updates to development and planning, followed by a final conclusion. The completion of this third milestone marks a significant point in Logisteps progress, as it completes the core functionality of the product.

# Discussion

The completion of this development cycle caps off a major deliverable required for usability of the Logisteps system (for end users) – the ability to transmit data from one end of the Logisteps stack to the other. Prior to this milestone, integration of adjacent components was completed, but full integration of the system had not yet been achieved. This milestone marks the completion of full system integration. Further discussion will describe the process of configuring the Logisteps web technology for deployment on Google virtual machines.

In a previous milestone, Bluetooth communication was achieved, and fully documented. This milestone report will focus on the efforts that were taken to process received Bluetooth data, transform the data into Step instances, and POST the step data to the online web server.

## Full Stack Data Communication

As previously stated, Bluetooth communication was achieved in a previous milestone, so this milestone will not focus on the operability of the Bluetooth-related code. Rather, this document will highlight the logic implemented to process step data and communicate the step data to the cloud.

### Data Format from Microcontroller

Before processing any received Bluetooth data in the Android application, the format of the raw data must be known. When the data is received in the Android application, it is received as a Data object[[1]](#footnote-1), with has no primitive types, or metadata related to the format of the received data. It is up to the implementor to implement a way to decode the data into appropriate Android primitive data types.

To make data parsing as simple as possible in the Android application, the Logisteps development team decided that data should be packed together in the embedded source code prior to being transmitted over Bluetooth. What this means is that ADC sensor readings are placed into a single 8-bit array of length 15. When the embedded application senses that a user is placing pressure on the sensor, it increases its sampling rate, and fills the buffer with 15 individual ADC sensor readings. When little (or no) pressure is being placed on the piezo-electric sensors, only 5 ADC readings are done, and the rest of the buffer is filled with copies of the 5 individual sensor readings. This ensures that at all times, exactly 15 8-bit data pieces are in the buffer. This is done for each sensor, and these readings are sent on a fixed interval using separate Bluetooth data characteristics. The format of the data can be seen in the following figure.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
| sr0 | sr0 | sr0 | sr1 | sr1 | sr1 | sr2 | sr2 | sr2 | sr3 | sr3 | sr3 | sr4 | sr4 | sr4 |

Table - Data format of raw sensor readings when no pressure is detected.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
| sr0 | sr1 | sr2 | sr3 | sr4 | sr5 | sr6 | sr7 | sr8 | sr9 | sr10 | sr11 | sr12 | sr13 | sr14 |

Table - Data format of raw sensor readings when pressure is detected.

Using this data format, regardless of the user’s movement or the data characteristic, the Android application does not need to account for different length arrays of sensor readings.

### Reception of Sensor Data

Once data is transmitted over Bluetooth, it is handled by the Android OS, which eventually calls the LogistepsSensorDataCallback. This class is instantiated twice and applies a transformation of the raw data before it is received in the LogistepManager’s Bluetooth callback functions. The job of the LogistepsSensorDataCallback is to transform the raw data into Android primitive values and/or objects which can be more appropriately handled in the LoistepManager’s Bluetooth callback functions.

In the LogistepSensorDataCallback, the size of the data payload is verified, and if incorrect/corrupt, it calls a function which handles data transmission failure. If the data size is valid, each sensor reading is parsed into a SINT8, and added to an ArrayList. Once all data is parsed and placed into the ArrayList, the LogistepManager’s onSensorDataReceived method is called, passing the list of sensor readings.

@Override  
**public void** onDataReceived(@NonNull **final** BluetoothDevice device, @NonNull **final** Data data) {  
 **if** (data.size() != **SENSOR\_READINGS**) {  
 onInvalidDataReceived(device, data);  
 **return**;  
 }  
  
 ArrayList<Integer> sensorReadings = **new** ArrayList<>();  
  
 **for** (**int** i = 0; i < **SENSOR\_READINGS**; i++) {  
 sensorReadings.add(data.getIntValue(Data.***FORMAT\_SINT8***, i));  
 }  
 onSensorDataRecieved(device, sensorReadings);  
}

In the LogistepManger’s Bluetooth callback functions, the sensor data readings are used to instantiate SensorReading instances, which are then added to a new ArrayList. Essentially, all this method does is further transform the data into objects so that the data can be more easily managed. Step instances are not created in the Bluetooth callback functions, because the creation of a step depends on sensor data from both the top sensor and the bottom sensor. Separate callbacks are used for the top sensor and the bottom sensor, and they do not share data. In order to determine if a step has been taken, a StepManager class was created. This class implements the filtering algorithm for determining if a user’s sensor readings indicate an actual step has been taken. Shown below is the logic that has been implemented for transforming sensor reading data into SensorReading instances.

**private final** LogistepsSensorDataCallback **mTopSensorCallback** = **new** LogistepsSensorDataCallback() {  
 @Override  
 **public void** onSensorDataRecieved(@NonNull BluetoothDevice device, List<Integer> sensorReadings) {  
 ArrayList<SensorReading> sensorReadingsList = **new** ArrayList<>();  
 **for** (**int** i = 0; i < sensorReadings.size(); i++ ){  
 sensorReadingsList.add(**new** SensorReading(**"T"**, sensorReadings.get(i)));  
 }  
  
 **mStepManager**.addTopSensorReadings(sensorReadingsList);  
 }  
  
 @Override  
 **public void** onInvalidDataReceived(@NonNull **final** BluetoothDevice device,  
 @NonNull **final** Data data) {  
 log(Log.***WARN***, **"Invalid data received: "** + data);  
 }  
 };

### StepManager

The StepManager was implemented to provide logic for determining if sensor data received from the top sensor and bottom sensor callback functions represent an actual step. In addition, when the class is instantiated, it requires an event listener that gets called when a step has been detected. In other words, the StepManager listens for sensor data, and once it has detected a step, it alerts a listening class that a step has been created and passes the step data to that class. This implements the subscriber architecture pattern, which eliminates unnecessary polling between classes.

In the current design, the StepManager is instantiated by the ShoeViewModel for the left and right shoe. The ShoeViewModel also instantiates an instance of the LogistepsManager and passes the instance of the StepManager to the LogistepsManager. This allows the LogistepsManager to pass the SensorReading instances to the StepManager for analysis. When the ShoeViewModel instantiates the StepManager, it passes a callback function, which includes logic for handling new Step instances. The instantiation of these classes is shown in the following code snippet.

@Inject  
 **public** ShoeViewModel(@NonNull **final** Application application) {  
 **super**(application);  
  
 **sharedPreferences** = application.getSharedPreferences(**"userCredentials"**, Context.***MODE\_PRIVATE***);  
  
 **steps** = **new** ArrayList<>();  
  
 *//Create step manager. Collects sensor readings and creates steps* **stepManager** = **new** StepManager();  
 **stepManager**.setOnStepCreatedEventListener(step -> {  
 step.setShoe(**shoe**.getFoot());  
 **steps**.add(step);  
 **if**(**steps**.size() == 10) {  
 postSteps();  
 **steps**.clear();  
 }  
 });  
  
 *// Initialize the manager* **mLogistepsManager** = **new** LogistepsManager(getApplication(), **stepManager**);  
 **mLogistepsManager**.setGattCallbacks(**this**);  
}

#### Determining a Valid Step

To determine if a valid step has been taken, an algorithm was developed which analyzes the time characteristics of the sensor readings, as well as the values of the sensor readings. The algorithm which will be described is the first version of the algorithm and will be improved with further testing.

When sensor readings are received, the StepManager first checks to see if the data was received within an expected time window. If too much time has occurred, data has been corrupted, and the current sensor readings should be discarded. If the time since the last received sensor reading is short enough, the sensor reading is added to a list of recently received sensor readings. After a sensor reading has been received, a function is called, which performs checks to see if a full transmission has been received.

**public void** addBottomSensorReadings(List<SensorReading> sensorReadings) {  
 *//****TODO try to create step if possible. If possible, call the event listener* long** now = System.*currentTimeMillis*();  
  
 *//Check to see if the data is a new data burst, and there is already sensor data  
 //If there is, then we need to reset* **if**(**bottomSensorReadings**.size() > 0) {  
  
 **if** (now - **lastBottomSensorUpdate** > **TIME\_THRESHOLD**) {  
 *//Need to reset the data. A potential error occurred* **bottomSensorReadings**.clear();  
 }  
 }  
  
 **bottomSensorReadings**.addAll(sensorReadings);  
 **lastBottomSensorUpdate** = now;  
  
 onSensorDataReceived();  
}

Once onSensorDataReceived has been called, it checks if the number of sensor readings saved are greater than or equal to 15, which is the expected payload size of the Bluetooth transmission. If this is true, then the StepManager calculates the average pressure for both the top sensor and bottom sensor. For a step to have occurred, both the top and bottom sensors must have experienced an average pressure above a set threshold. If this has occurred, then a Step instance is created, and the event listener is called. If the average pressure for both sensors is not greater than the threshold, then the sensor readings are discarded, and the StepManager waits for additional data.

**private void** onSensorDataReceived() {  
 **if** (**topSensorReadings**.size() >= **STEP\_EVENT\_SIZE** && **bottomSensorReadings**.size() >= **STEP\_EVENT\_SIZE**) {  
  
 *//If both sensors have average pressure larger than the threshold, create a step* **if** (getAvgPressure(**topSensorReadings**) >= **PRESSURE\_THRESHOLD**

&& getAvgPressure(**bottomSensorReadings**) >= **PRESSURE\_THRESHOLD**) {  
 createStep();  
 }  
  
 **topSensorReadings**.clear();  
 **bottomSensorReadings**.clear();  
 }  
}

### On Step Creation

Once a step has been created, the data is received in the ShoeViewModel. This logic was already shown in the code snippet showing the constructor of the ShoeViewModel. Once ten steps have been cached, the step data is sent to the web server via a POST request. To do this, a StepRepository was implemented which contains the logic for making the network call to the web server. This separates the webservice details from the ShoeViewModel, making the code more maintainable and testable.

The repository makes use of the Retrofit library, which implements the webservice interface, which makes the TCP/IP network call to the upstream web server. To make the POST request to the web server, the StepRepository simply needed to do what is shown in the following code snippet.

**public void** postSteps(List<Step> steps, User user) {  
 **executor**.execute(() -> {  
 **try** {  
 Response<List<Step>> response = **webservice** .postSteps(Credentials.*basic*(  
 user.getBaseUser().getUsername(),  
 user.getBaseUser().getPassword()),  
 steps)  
 .execute();  
 } **catch** (IOException e) {  
 Log.*e*(***TAG***, e.toString());  
 }  
 });  
}

Upon successful execution of the postSteps function, step data is sent to the web server, completing the full-stack transmission of step data.

### Final Remarks

This is an initial implementation of the step filtering logic needed by the application. In the final weeks of the project, this algorithm will be tweaked to better represent the actual representation of a physical step. As a result, this software will likely increase in complexity, so this documentation should not serve as a final representation of the step filtering algorithm.

The following figure is a high-level overview of the logical flow of data when creating a Step instance.

A screenshot of a cell phone

Description automatically generated

Figure - High-level overview of Step Filtering design.

# Risks to the project

With the recent developments in the project, little risks remain for the project. Most of the remaining work involves fine tuning and improvement of existing software. In addition to this, enough parts were purchased to recreate the insole technology several times, in the case that hardware malfunctions. The only remaining risks are related to version control for the software. Step filtering algorithms will be fine tuned in the future, meaning that the software will be changed. It’s important to maintain proper version control, using git, so that any breaking changes can be rolled back.

# Deliverables

The following list summarizes the deliverables completed for milestone three. This is a summarized list of all deliverables and not an exhaustive list of every single task completed.

* Full stack communication
* Advanced step filtering

# Development Plan / Plan Update

This milestone was completed on time and marks the final completion of major development milestones. Future development will be focused making changes to the software to improve the quality of it. This will be conducted via system integration testing.

# Conclusion

The completion of the third milestone, and third development cycle, brings about a major reduction in risk to the project. By making it possible to send data from a self-powered insole, all the way to the online web server, the project is able to complete its major function. Additional work will be done for improving the quality of the product. The coming weeks of the project will focus on fine-tuning of existing software, and completion of poster design.

1. <https://github.com/NordicSemiconductor/Android-BLE-Library/blob/master/ble/src/main/java/no/nordicsemi/android/ble/data/Data.java> [↑](#footnote-ref-1)