SAND2015-0642R

Student: Brett Decker
Adviser: Shriram Krishnamurthi
Brown University
Master's Project Report
January 21, 2015

TIERLESS PROGRAMMING FOR THE INTERNET OF THINGS

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

The Internet of Things (IoT) is about Internet-addressability and connectivity for everyday devices. The goal of this project was to create a framework to allow developers to more easily control IoT devices and turn their interactions into meaningful applications. We leveraged a tierless approach for Software Defined Networking (SDN) to build this framework. We expanded Flowlog, a tierless programming language for SDN controllers, to support IoT devices developed by Spark IO to build this framework.

BACKGROUND

In traditional networking routers handle the networking functions. In Software-Defined Networking (SDN) a controller determines the networking functions. In the simplest case the switches forward all traffic to the controller which then determines the traffic flow. Ideally, the controller gives the switches the traffic policy and only packets that do not match any policy are sent to the controller. The controller can also update the policy on the switches which allows for dynamic network configurations.

Figure 1 shows the SDN architecture:

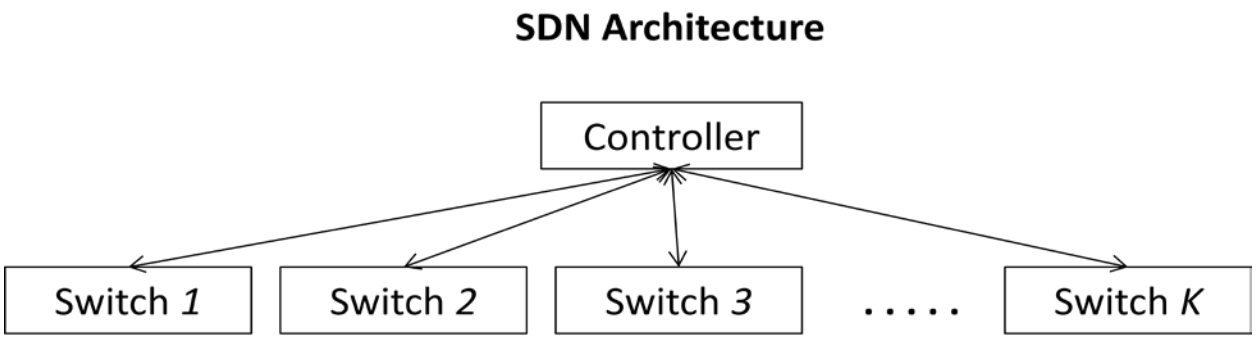

FIGURE 1: SDN ARCHITECTURE

In SDN, the interactions between the controller and the switches require control information, data, and state. A tierless programming language provides a unified abstraction for programming three tiers: the control plane, data-plane, and controller state [1].

Flowlog is a tierless programming language. A Flowlog application specifies behavior for both a controller (a program with state that takes actions on its own) and switches with rules installed (switches also take actions on their own). Flowlog generates the needed SQL code and OpenFlow rules which allows developers to avoid explicit management of all the tiers. Instead the developer can focus on the networking application as a whole. Flowlog uses a database for the state and allows developers to define and create tables. Flowlog is stateful which allows the controller to change and update policies based on the state. Flowlog also performs proactive compilation – it doesn't wait for packets to arrive before installing the rules down to the switches. Thus, the functionality of the controller and switches can be expressed in a single tierless Flowlog program. The remainder of this report will assume a basic familiarity with Flowlog (see [1] for a reference).

APPLYING SDN TO THE INTERNET OF THINGS

Applications in the domain of the Internet of Things (IoT) have a similar structure to network routing. Multiple devices must communicate to perform a specific function (just as routers perform networking functions). Applying the SDN architecture to the IoT, we can extract the controller function for the IoT devices. Then the connected devices need only pass messages and perform actions based on the message content, just as switches transmit packets based on packet content (but do not perform the controller functions). This SDN-like IoT architecture is shown in Figure 2:

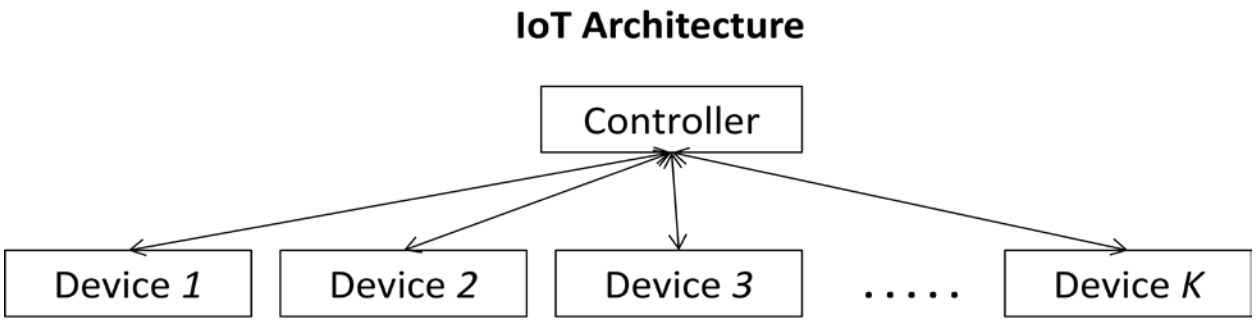

FIGURE 2: IOT ARCHITECTURE

This IoT architecture allows end devices to not require global knowledge of the application that more appropriately belongs at the controller. The IoT architecture provides dynamic IoT applications as the SDN architecture provides dynamic networking policies.

The key difference between these architectures lies in the capabilities of the end point nodes. All switches perform the same functions (broadly speaking) and thus their functional domain is uniform. By contrast, each IoT device may interface with and control a unique combination of sensors and actuators.

We decided to expand the Flowlog language and run-time to enable Flowlog to support this IoT architecture. Before proceeding, we had to find a hardware framework for IoT devices that could coexist with Flowlog and our IoT architecture.

IOT HARDWARE FRAMEWORK

We found Spark IO, a company dedicated to developing products for the IoT [2]. The most attractive feature of the Spark IO framework is that it is open source from top to bottom, from software to hardware.

Spark IO provides connected computing devices. The device we selected to use is called the Spark Core [2]. The Spark Core has a processor, memory, and most importantly, a Wi-Fi module that allows it to connect to the Internet. Sensors and actuators can be connected to the many exposed pins on the Spark Core.

Figure 3: Spark Core with LED shows a Spark Core (with breadboard) with a light-emitting diode (LED) attached.

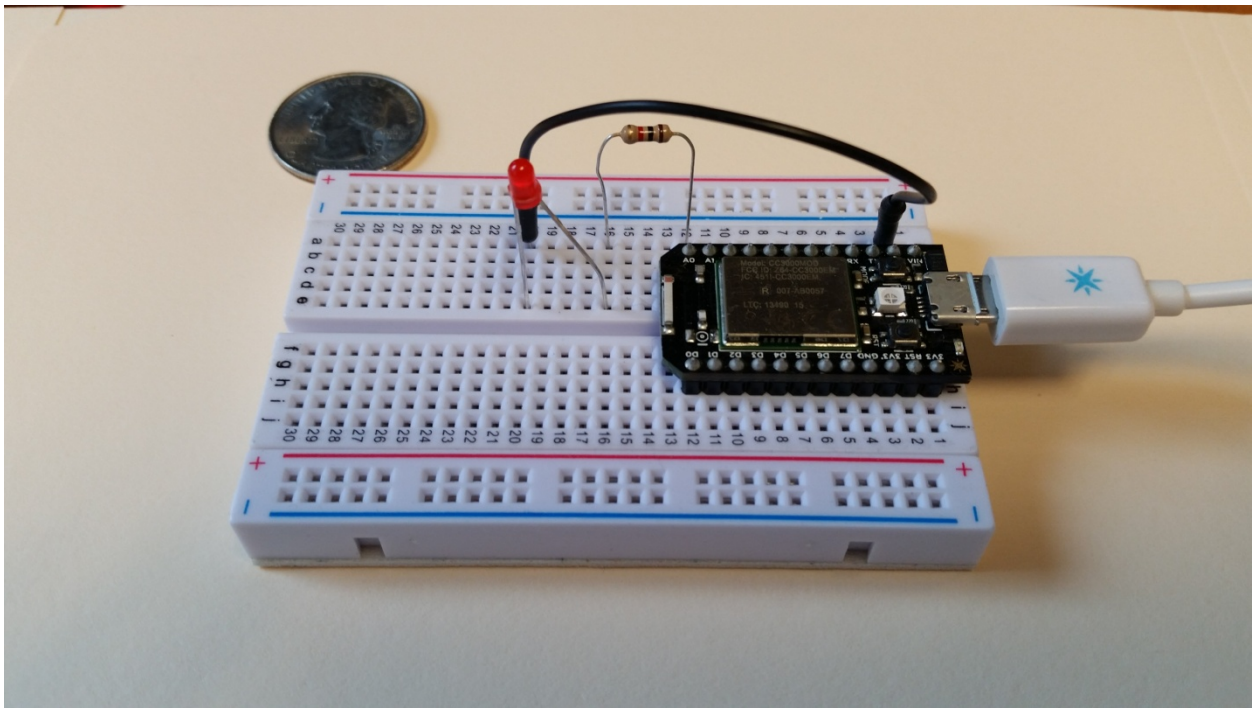

FIGURE 3: SPARK CORE WITH LED

Note: the quarter in-frame is for size comparison. The USB plug is for power or optional serial debugging, but *not* for Internet connection.

Another attractive feature of the Spark Core is its communication protocol. Spark IO has cloud services based on Representational State Transfer (REST) architecture [3]. This allows for communication between devices via Hypertext Transfer Protocol (HTTP). Spark IO provides a RESTful Application Program Interface (API) to control and access the Spark Core devices. In this way, the Spark Core can send and receive events. Events are sent as simple HTTP push requests. Events are received using Server-Sent Events (SSEs). Essentially, SSEs allow for a client to receive data from a server via an HTTP connection (see [4] for details).

Spark IO also provides a compiler that takes Arduino style C code (referred to as Arduino C, hereafter) and creates a binary file that the Spark Core can execute. Flashing the Spark Core is a simple process because the Spark Core supports over-the-air flashing. Spark IO provides tools for both the compile and flash steps.

Many different sensors and actuators can be used with the Spark Core. We focused on supporting the following sensors and actuators:

1. LED – light emitting diode
2. FSR – force sensitive resistor
3. PIR – passive infrared sensor (motion detector)
4. Servo Motor – simple motor
5. Digital pins (for input and output)

After selecting a hardware framework, we next worked on augmenting and developing both the Flowlog language and run-time to allow Flowlog to control the Spark Core devices.

FLOWLOG AUGMENTATION

The Flowlog run-time is implemented in the OCaml programming language. OCaml supports functional, imperative, and object oriented programming [5]. OCaml has an advanced type system which provides expressiveness and safety [5]. In order to augment the Flowlog run-time, I had to learn OCaml.

Flowlog required two areas of development to support the IoT architecture with Spark Core devices:

- Run-time communication with the Spark Cloud
- Means to handle unique Spark Core capabilities in language and run-time

COMMUNICATION WITH SPARK CLOUD

I had to develop a proxy to interface between the Flowlog run-time and the Spark Cloud infrastructure to allow communication between the Spark Cores and the Flowlog controller. Spark IO provides a JavaScript Library, SparkJS, to interface with the Spark Cloud REST API. I developed one side of the proxy in Node.JS – to register with the Spark Cloud for Server-Sent Events (SSEs) and then send the events to the Flowlog run-time. We chose to use Node.JS to exercise Flowlog's RPC interface. The other side of the proxy I developed in OCaml – to send events to the Spark Cloud which could then be received by Spark Cores (the Spark Cores must first register with the Spark Cloud for SSEs). With this communication proxy in place, we leverage the Spark Cloud framework for our controller to end node device communication.

HANDLING UNIQUE SPARK CORE CAPABILITIES

As noted above, the most significant difference from the SDN architecture to the IoT architecture is the difference in capabilities of the end node devices. In Flowlog, the switches can all be controlled by OpenFlow rules. But the IoT devices perform many different and possibly unique functions. Thus the IoT controller must have knowledge of the capabilities of each end node device in order to adequately control it. Thus, we had to devise a way for our Flowlog program (which runs the controller) to know of the capabilities of the Spark Core devices with which it interacts.

I developed the syntax and semantics for a simple domain-specific language (Spark Specification Language) to express the capabilities of all Spark Cores used in a Flowlog application. To show how this domain-specific language works, consider the following configuration:

Two Spark Cores – one with an LED and passive infrared motion sensor (PIR) attached; the other with just an LED attached.

Figure 4 provides a picture with the described setup:

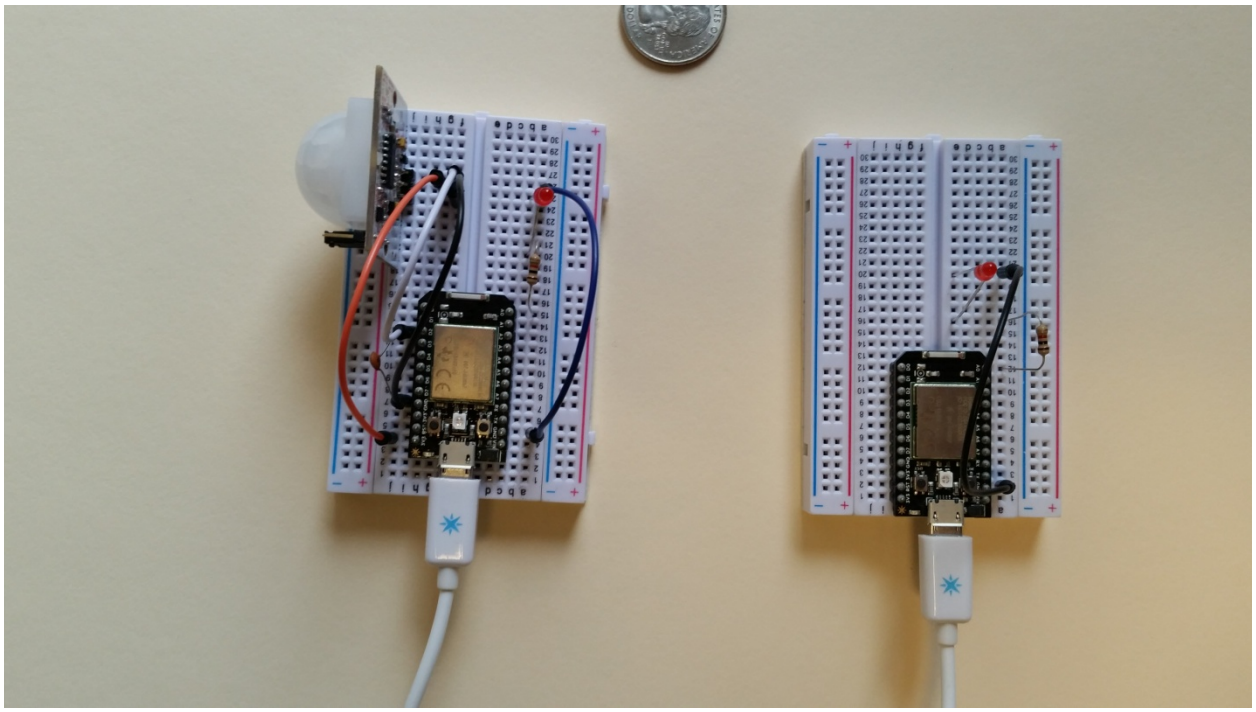

FIGURE 4: SPARK CORES

Figure 5 shows how to specify the Spark Cores from Figure 4:

```
// This Spark has a motion sensor
SPARKCORE AT 70:ff:76:00:f0:b0 WITH ID 54ff6f066667515157391567 HAS
    spark_led   ("A0", 0, 90, 0) AND   // LED
    spark_pir   ("D2");                // motion sensor

// This Spark controls LED lighting
SPARKCORE AT 70:ff:76:00:ef:f4 WITH ID 53ff74066667574829281667 HAS
    spark_led   ("A0", 0, 90, 0);      // LED
```

FIGURE 5: EXAMPLE SPARK SPECIFICATION

The above code states that there is a Spark Core at MAC address `70:ff:76:00:f0:b0`, with an ID `54ff6f066667515157391567`, that has a PIR attached to pin "`D2`." There is also a Spark Core at MAC address `70:ff:76:00:ef:f4`, with an ID `53ff74066667574829281667`, that has an LED attached to pin "`A0`." The other three parameters for the LED are the minimum, maximum, and initial values for the LED. There are no additional parameters with the PIR, because it is a simple sensor that only gives one digital output, whether motion is detected or not. Each sensor and actuator that is supported has a set number of parameters, but that number varies per sensor or actuator. The MAC address and ID for a Spark Core are determined by Spark IO. The pin associated with each sensor or actuator is determined by the developer (and must reflect the actual hardware configuration). Any number of the same sensors or actuators can be used with a Spark Core (with the sole limitation being the actual number of appropriate pins on the hardware).

I built a parser, lexical analyzer, and compiler for the Spark Specification Language and integrated these tools into the Flowlog run-time. The compiler builds an Abstract Syntax Tree (AST) from the Spark Specification Language that Flowlog uses to create Spark specific tables and events. This allows developers to create Flowlog applications that have knowledge of the capabilities of the Spark Cores that they interact with. Assuming Figure 5 as the example Spark specification, Figure 6: Flowlog-Spark Built-in Tables shows the contents of the Flowlog built-in tables after startup:

```
// LED built-in tables
spark_leds      (70:ff:76:00:ef:f4, "A0";
                 70:ff:76:00:f0:b0, "A0")
spark_leds_min  (70:ff:76:00:ef:f4, "A0", 0;
                 70:ff:76:00:f0:b0, "A0", 0)
spark_leds_max  (70:ff:76:00:ef:f4, "A0", 90;
                 70:ff:76:00:f0:b0, "A0", 90)
spark_leds_init (70:ff:76:00:ef:f4, "A0", 0;
                 70:ff:76:00:f0:b0, "A0", 0)
// PIR built-in table
spark_pirs      (70:ff:76:00:f0:b0, "D2")
// No other built-in tables for PIRs
```

FIGURE 6: FLOWLOG-SPARK BUILT-IN TABLES

These tables can be accessed in a Flowlog application just like any other developer defined table. It should be noted that we intentionally chose to have multiple tables with smaller arity for each sensor or actuator instead of one table with a larger arity because of our use of the tool Alloy (discussed in the section Formal Methods). One additional hurdle to overcome remained for Flowlog to support the Spark hardware.

The Spark Cores must be coded in Arduino C code, but the controller is written completely in Flowlog. To rectify this, we augmented Flowlog to generate the needed Arduino C code for each Spark Core. This is essential to achieving tierless programming in Flowlog for the IoT architecture.

I developed a template approach for the Arduino C code generation. I developed the needed Arduino C code for the Flowlog run-time to be able to receive data from and command each supported sensor and actuator. I developed code for the Flowlog run-time to parse the AST created by the Spark specification file and determine what Arduino C code each Spark Core needed (based on the template code for each sensor and actuator).

Building from the same example as above (see Figure 5, Figure 6: Flowlog-Spark Built-in Tables), the following figures show some of the Arduino C code generated for the two Spark Cores.

Global constants are generated from the Spark specification parameters to each sensor and actuator. Here are the generated global constants for the Spark Core with the single LED attached:

```
const char MY_TARGET[18] = "70:ff:76:00:ef:f4";
const int  LED_SET_SSCANF_ARGS = 3;
const int  LED_A0_MIN = 0;
const int  LED_A0_MAX = 90;
const char LED_A0_PIN_ID[] = "a0";
const int  LED_A0_PIN = A0;
```

FIGURE 7: CONSTANTS – SPARK CORE WITH THE LED

Note the corresponding parameters for the LED: pin, minimum, and maximum value to the constants "LED_A0_PIN", "LED_A0_PIN_ID", "LED_A0_MIN", and "LED_A0_MAX". The constant "MY_TARGET" comes from the device's MAC address. The initial value parameter is used later in the Spark setup code:

```
Spark.subscribe("_spark_led_set", handleLEDSet);

pinMode(LED_A0_PIN, OUTPUT);
analogWrite(LED_A0_PIN, 0);
```

FIGURE 8: SETUP – SPARK CORE WITH THE LED

The first line shows how a Spark Core "registers" with the Spark Cloud for specific events. The value handleLEDSet identifies the call-back function for the event (Figure 9 shows the code for handleLEDSet). The second line sets up the specified pin to be used as an output pin. The third line sets the pin to the initial value (determined by the initial value parameter in the Spark Specification).

Note: the event name is pre-pended with an underscore. This is needed because Spark IO reserves all events that start with "spark." The Flowlog run-time adds an underscore to the event name when sending and strips off the underscore when receiving events.

Here is the code for the call-back function that handles the command ('_spark_led_set') from Flowlog:

```
void handleLEDSet(const char *event, const char *data)
{
        int sscanf_ret = 0;
        unsigned int vs = 0;

        char target[18];
        char pin[3];
        target[17] = '\0';
        pin[3] = '\0';

        sscanf_ret = sscanf(data, "{\"target\":\"%17s\",\"pin\":\"%2s\",\"vs\":\"%u\"}",
                            target, pin, &vs);

        if (sscanf_ret == LED_SET_SSCANF_ARGS) {

                if (strcmp(target, MY_TARGET) == 0) {

                        if (strcmp(pin, LED_A0_PIN_ID) == 0) &&
                           (vs <= LED_A0_MAX && vs >= LED_A0_MIN) {
                               analogWrite(LED_A0_PIN, vs);
                        }
                } // else - data is not meant for this device

        } // else - data format unknown
}
```

FIGURE 9: EVENT – SPARK CORE WITH THE LED

The function "handleLEDSet" first scans the data string with the expected Flowlog command format. If the format is correct, and the "target" field is the same as the device's MAC address, then the command is meant for that device. Note that all Spark Cores that register for the same event will see

all traffic whether or not it is meant for the individual device; this is why Flowlog sends the data with the "`target`" and "`pin`" fields – to uniquely identify every sensor or actuator. For every LED attached to a device, the "`pin`" field in the event is compared to each LED's pin (in the above example there is only one LED attached, so there is only one comparison needed). If the two pins match, and the new value is within bounds, the new value is written to the pin. This is how Flowlog can control the value of every LED individually.

By using Flowlog, the developer doesn't have to write any code for the Spark Cores. The code needed will be generated automatically by Flowlog. This is one of the advantages to using a tierless programming language.

With the Spark Specification Language support, the built-in tables for Spark Core capabilities, and code generation for Spark Cores, the Flowlog run-time can handle the unique capabilities of the Spark Cores.

FLOWLOG IOT APPLICATIONS USING SPARK CORES

This section describes an example Flowlog application that uses the Flowlog IoT architecture. Excerpts from working Flowlog code are shown and explained.

Application: Auto Night-Lights

Architecture: The Flowlog run-time is executing the Flowlog application (referred to as the controller hereafter). The controller communicates with Spark Cores with LEDs and Spark Cores with PIRs (referred to as motion sensors hereafter).

Description: This application controls automatic night-lights. Motion sensors are "enabled" at a specified time (night-time) and "disabled" at a specified time (day-time). The notion of "enabled" and "disabled" is purely in the controller software – the motion sensors are always powered on and working, but the controller only responds to motion detected events of "enabled" motion detectors. While the motion sensors are "enabled", if they trigger, the associated LED(s) will brighten up and stay on for a specified amount of time (on-time) and then fade down. Note that the code running on the Spark Cores has no knowledge of the overall application. The Spark Cores with LED(s) only respond to events that request the LED(s) be set to a specific value. The Spark Cores with motion sensor(s) only send an event when motion is detected.

Excerpts: The following Flowlog database tables are used to store which motion sensors are enabled and the relationship that defines which motion sensor(s) trigger which LED(s).

```
TABLE enabled_motion_sensors (macaddr, string);
TABLE motion_sensors_to_leds (macaddr, string, macaddr, string);
```

Note: in the tables above, "`macaddr`" and "`string`" are the types of the column for the table relation. Each LED and motion sensor is identified by a MAC address of the Spark Core and the pin of the Spark Core attached to the sensor or actuator. The pin is stored as a string. Thus the "`enabled_motion_sensors`" table identifies which motion sensors are enabled (by MAC address and pin) and the "`motion_sensors_to_leds`" table identifies the motion sensor (a MAC address and pin pair) to LED (again a MAC address and pin pair) relation.

The following Flowlog events are used to receive data from motion sensors and set LEDs (these are built-in events).

```
EVENT spark_pir_motion_detected {target: macaddr, pin: string};
EVENT spark_led_set             {target: macaddr, pin: string, vs: int};
```

Note: "`target`" is the name of the first field of the event (again, "`macaddr`" is the type); "`pin`" is the name of the second field of the event (again, "`string`" is the type) and so on.

The following excerpt shows how the tables and events can be used to implement the application described. The Flowlog code states that on a "`spark_pir_motion_detected`" event where the sensor is enabled, set the LED value of the corresponding LED(s) associated with the sensor to a new value (for the complete code example see Appendix A – Flowlog Application).

```
1  ON spark_pir_motion_detected(motion) WHERE
2      enabled_motion_sensors(motion.target, motion.pin) AND
3      … // any other needed conditions
4      motion_sensors_to_leds(motion.target, motion.pin, ledTarget, ledPin):
5
6      DO spark_led_set_out(set) WHERE
7          set.target = ledTarget AND
8          set.pin = ledPin       AND
9          … // define new_brightness
10         new_brightness = set.vs;
```

Line 1 is the start of a new rule or policy. The rule is in place when a "`spark_pir_motion_detected`" event occurs. This process starts at a motion sensor. When the motion sensor triggers it sends a "`spark_pir_motion_detected`" event to the Spark Cloud, which pushes the event down to the Flowlog proxy. The proxy then sends the event to Flowlog.

At the end of Line 1, there is a "`WHERE`" clause. This allows us to define any conditions that must be met (when the event occurs) for the rule to execute. Line 2 is the first condition: the motion sensor that triggered the event (identified by target, the MAC address, and pin) must be in the table "`enabled_motion_sensors`". Line 4 is the last condition but serves a different purpose: it binds "`ledTarget`" and "`ledPin`" to any LED target and pin combination associated with the motion sensor that triggered. Line 6 is the start of a "`DO`" statement. Flowlog will send out a "`spark_led_set_out`" event. The value "`set`" is the name bound to the event. In Lines 7 & 8 the fields of "`set`" are bound to "`ledTarget`" and "`ledPin`." The last field of "`set`" is bound to "`new_brightness`" (Appendix A – Flowlog Application shows how "`new_brightness`" is defined). Finally, Flowlog sends the "`spark_led_set_out`" event defined by "`set`" to the proxy, which then sends the event to the Spark Cloud. The Spark Cloud pushes the event down to the Spark Cores (registered to receive the event), where the event is handled (in this case the Spark Core set the LED value to the new brightness).

Though this is a simple application (and the needed Arduino C code for the Spark Cores could be argued to be less than the Flowlog code), the tierless paradigm of Flowlog allows a developer to build applications at a high level. The low-level code needed to control the Spark Cores is generated for the developer. Thus the application could be modified without the need to change the Spark Core code.

Another benefit to using Flowlog is that it can also provide SDN. Flowlog could be integrated with a home network to provide custom network configurations (e.g. firewall). Flowlog also provides tierless analysis with formal methods.

FORMAL METHODS

Alloy is a formal methods language and tool for performing verification and validation [6]. Alloy generates sets of instances that satisfy all constraints. The Flowlog run-time has the capability to generate Alloy specifications. Developers can then add properties to these specifications and perform verification. Formal reasoning of all tiers can be accomplished since Flowlog applications are tierless.

Another signification area of analysis is change impact – how does a specific change impact the application. Change impact can be achieved by comparing and analyzing the Alloy sets of instances to determine the how a change influences the model. Flowlog has built-in change impact allowing developers to verify modifications. We chose to develop our IoT framework on top of Flowlog in part because it already provided formal methods.

FUTURE WORK

One signification area of future work is compilation. Currently, Flowlog uses a template to generate code for the Spark Cores from a Flowlog application. It would be interesting for Flowlog to have the capability to compile applicable parts of the Flowlog application down to the Spark Cores. Flowlog could reprogram the Spark Cores throughout the application if needed or wanted since the Spare Cores can be flashed via over-the-air communication.

CONCLUSION

We developed an architecture for the IoT and expanded the language and run-time of Flowlog to provide a tierless programming language for our IoT architecture. This architecture uses WiFi-enabled connected devices from Spark IO. An application can be developed in the Flowlog programming language, and the Flowlog run-time handles code generation, compilation, and flashing of the Spark Cores to provide a tierless approach to developing IoT applications.

ACKNOWLEDGEMENTS

Thanks to Shriram Krishnamurthi and Tim Nelson for their mentorship, guidance, technical advice, and support for this project. Thanks to Tim Nelson for his continued development of the Flowlog language to allow this project to succeed.

SKILLS LEARNED

In order to complete this project I had to learn the following programming languages and tools:

1. Flowlog programming language – to create application in Flowlog
2. OCaml – to augment the Flowlog run-time
3. JavaScript and Node.JS – to build the Flowlog-Spark proxy
4. ocamllex & ocamlyacc – to develop the Spark Specification Language parser, lexical analyzer, and compiler (to an AST in OCaml)

I already had experience with Alloy and Arduino C. I had to increase my hardware skills to create simple circuits with sensors and actuators on a breadboard with the Spark Core attached. Specifically I had to learn how to design circuits to use LEDs (light-emitting diodes), FSRs (force-sensitive resistors), PIRs (passive infrared sensors), and Servo motors.

REFERENCES

[1] TIM NELSON, ANDREW D. FERGUSON, MICHAEL J.G. SCHEER, AND SHRIRAM KRISHNAMURTHI. Tierless Programming and Reasoning for Software-Defined Networks. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI '14).*

[2] Spark IO. *http://spark.io/*.

[3] Spark Docs, Cloud API. *http://docs.spark.io/api/*.

[4] Server-Sent Events. W3C. *http://www.w3.org/TR/eventsource/*.

[5] OCaml Programming Language. *https://ocaml.org/*.

[6] Alloy. *http://alloy.mit.edu/alloy/*.

APPENDIX A – FLOWLOG APPLICATION

This appendix contains the complete Flowlog application described in section Flowlog IoT Applications using Spark Cores.

```
//=====================================================================
//                  App Description
//=====================================================================
//
// This application controls automatic night lights. Motion sensors are
// enabled at Night Time (default is 10:00 PM) and disabled at Day Time
// (default is 6:30 AM). While the motion sensors are enabled, if they
// trigger, the associated LED(s) will fade up and stay on for a specified
// amount of time (default is 15 seconds) and then fade down.
//
// The Night Time, Day Time, and LED on time values can be changed by the
// following events:
// - update_night_time
// - update_day_time
// - update_led_on_time_msecs
//
//=====================================================================

TABLE enabled_motion_sensors (macaddr, string);
TABLE motion_sensors_to_leds (macaddr, string, macaddr, string);

TABLE led_on_time_msecs      (macaddr, string, int);
TABLE led_brightness         (macaddr, string, int);

TABLE night_time             (int, int, int);
TABLE day_time               (int, int, int);

EVENT update_led_on_time_msecs  {target: macaddr, pin: string, ms: int};
EVENT update_night_time         {hr24: int, min: int, sec: int};
EVENT update_day_time           {hr24: int, min: int, sec: int};

EVENT start_spark_timer         {ms: int, id: string, mac: macaddr, pin: string};
EVENT spark_timer_expired       {id: string, mac: macaddr, pin: string};

EVENT set_alarm                 {hr24: int, min: int, sec: int, id: string};
EVENT alarm_expired             {hr24: int, min: int, sec: int, id: string};

// Use the Flowlog Timer utility at Paort 9091
OUTGOING start_spark_timer_out(start_spark_timer) THEN
    SEND TO 127.0.0.1:9091;

OUTGOING set_alarm_out(set_alarm) THEN
    SEND TO 127.0.0.1:9091;

//=====================================================================
ON startup(e):
    INSERT (mac, pin, 2000) INTO led_on_time_msecs WHERE
        spark_leds(mac, pin);
    INSERT (mac, pin, brightness) INTO led_brightness WHERE
        spark_leds_init(mac, pin, brightness);
    INSERT (mac, pin) INTO enabled_motion_sensors WHERE
        spark_pirs(mac, pin);
    // Simple mapping – all motion sensors trigger all LEDs.
    // This relation could obviously be more complicated.
    INSERT (mac1, pin1, mac2, pin2) INTO motion_sensors_to_leds WHERE
```

```
        spark_pirs(mac1, pin1) AND
        spark_leds(mac2, pin2);
    INSERT (22, 0, 0) INTO night_time;  // 22 hours - 10:00 PM
    INSERT (6, 30, 0) INTO day_time;    // 6 hours, 30 mins - 6:30 AM
    // Set Night Time Alarm
    DO set_alarm_out(alarm) WHERE
        night_time(alarm.hr24, alarm.min, alarm.sec) AND
        alarm.id = "night-time";
    // Set Day Time Alarm
    DO set_alarm_out(alarm) WHERE
        day_time(alarm.hr24, alarm.min, alarm.sec) AND
        alarm.id = "day-time";

// Enable all motion sensors at Night Time
ON alarm_expired(alarm) WHERE alarm.id = "night-time":
    INSERT (mac, pin) INTO enabled_motion_sensors WHERE
        spark_pirs(mac, pin);
    // Reset alarm
    DO set_alarm_out(new_alarm) WHERE
        night_time(new_alarm.hr24, new_alarm.min, new_alarm.sec) AND
        new_alarm.id = alarm.id;

// Disable all motion sensors at Day Time
ON alarm_expired(alarm) WHERE alarm.id = "day-time":
    DELETE (ANY, ANY) FROM enabled_motion_sensors;
    // Reset alarm
    DO set_alarm_out(new_alarm) WHERE
        day_time(new_alarm.hr24, new_alarm.min, new_alarm.sec) AND
        new_alarm.id = alarm.id;

// Update LED on time
ON update_led_on_time_msecs(led):
    DELETE (led.target, led.pin, ANY) FROM led_on_time_msecs;
    INSERT (led.target, led.pin, led.ms) INTO led_on_time_msecs WHERE
        spark_leds(led.target, led.pin);

// Update Night Time
ON update_night_time(time):
    DELETE (ANY, ANY, ANY) FROM night_time;
    INSERT (time.hr24, time.min, time.sec) INTO night_time;

// Update Day Time
ON update_day_time(time):
    DELETE (ANY, ANY, ANY) FROM day_time;
    INSERT (time.hr24, time.min, time.sec) INTO day_time;

// Motion detected
ON spark_pir_motion_detected(motion) WHERE
    enabled_motion_sensors(motion.target, motion.pin)                    AND
    motion_sensors_to_leds(motion.target, motion.pin, ledTarget, ledPin) AND
    led_brightness(ledTarget, ledPin, min)                               AND
    spark_leds_min(ledTarget, ledPin, min):
    DELETE (ledTarget, ledPin, ANY) FROM led_brightness;
    INSERT (ledTarget, ledPin, new_brightness) INTO led_brightness WHERE
        led_brightness(ledTarget, ledPin, brightness) AND
        add(brightness, 30, new_brightness);
    DO spark_led_set_out(set) WHERE
        set.target = ledTarget AND
        set.pin = ledPin       AND
        led_brightness(ledTarget, ledPin, brightness) AND
        add(brightness, 30, new_brightness)           AND
```

```
        new_brightness = set.vs;
    DO start_spark_timer_out(timer) WHERE
        timer.ms  = 1100           AND
        timer.id  = "led_increase" AND
        timer.mac = ledTarget      AND
        timer.pin = ledPin;

// LED fade up
ON spark_timer_expired(timer) WHERE timer.id = "led_increase":
    DELETE (timer.mac, timer.pin, ANY) FROM led_brightness;
    INSERT (timer.mac, timer.pin, new_brightness) INTO led_brightness WHERE
        led_brightness(timer.mac, timer.pin, brightness) AND
        add(brightness, 30, new_brightness);
    DO spark_led_set_out(set) WHERE
        set.target = timer.mac AND
        set.pin = timer.pin    AND
        led_brightness(set.target, set.pin, set.vs);
    DO start_spark_timer_out(new_timer) WHERE
        led_brightness(timer.mac, ANY, brightness) AND
        spark_leds_max(timer.mac, ANY, max)        AND
        lessthan(brightness, max)  AND
        new_timer.ms  = 1100       AND
        new_timer.id  = timer.id   AND
        new_timer.mac = timer.mac  AND
        new_timer.pin = timer.pin;
    DO start_spark_timer_out(new_timer) WHERE
        led_brightness(timer.mac, timer.pin, brightness)      AND
        spark_leds_max(timer.mac, timer.pin, max)             AND
        brightness = max                                      AND
        led_on_time_msecs(timer.mac, timer.pin, new_timer.ms) AND
        new_timer.id  = "led_decrease"  AND
        new_timer.mac = timer.mac       AND
        new_timer.pin = timer.pin;

// LED fade down
ON spark_timer_expired(timer) WHERE timer.id = "led_decrease":
    DELETE (timer.mac, timer.pin, ANY) FROM led_brightness;
    INSERT (timer.mac, timer.pin, new_brightness) INTO led_brightness WHERE
        led_brightness(timer.mac, timer.pin, brightness) AND
        add(brightness, -30, new_brightness);
    DO spark_led_set_out(set) WHERE
        set.target = timer.mac AND
        set.pin = timer.pin    AND
        led_brightness(set.target, set.pin, set.vs);
    DO start_spark_timer_out(new_timer) WHERE
        led_brightness(timer.mac, ANY, brightness) AND
        spark_leds_min(timer.mac, ANY, min)        AND
        lessthan(min, brightness)  AND
        new_timer.ms  = 1100       AND
        new_timer.id  = timer.id   AND
        new_timer.mac = timer.mac  AND
        new_timer.pin = timer.pin;
```

APPENDIX B – SPARK SPECIFICATION

This appendix contains a complete Spark Specification file for the Flowlog application in Appendix A – Flowlog Application.

```
// This Spark has a motion sensor
SPARKCORE AT 70:ff:76:00:f0:b0 WITH ID 54ff6f066667515157391567 HAS
    spark_pir   ("D2");  // motion sensor

// This Spark controls an LED
SPARKCORE AT 70:ff:76:00:f0:29 WITH ID 54ff6f066667515128381567 HAS
    spark_led   ("A1", 0, 90, 0);

// This Spark controls an LED
SPARKCORE AT 70:ff:76:00:f0:ac WITH ID 54ff70066667515127451467 HAS
    spark_led   ("A2", 0, 90, 0);

// This Spark controls an LED
SPARKCORE AT 70:ff:76:00:ef:f4 WITH ID 53ff74066667574829281667 HAS
    spark_led   ("A0", 0, 90, 0);
```

APPENDIX C – FLOWLOG-SPARK CONSTRUCTS

This appendix contains the type of sensors and actuators that Flowlog currently supports and the built-in tables and events used to interact and control the sensors and actuators.

Supported Sensors:

1. PIR – (passive infrared motion sensor)

   Built-in Table: `spark_pirs (MAC address, pin)`

   Built-in Event: `spark_pir_motion_detected (MAC address, pin)`

2. FSR – (force-sensitive resister)

   Built-in Tables:
   ```
   spark_fsrs (MAC address, pin)
   spark_fsrs_threshold (MAC address, pin, minimum value)
   ```

   Built-in Events:
   ```
   spark_fsr_under_threshold (MAC address, pin)
   spark_fsr_over_threshold (MAC address, pin)
   ```

3. DPinIn – (digital pin, input)

   Built-in Tables:
   ```
   spark_dpins_in (MAC address, pin)
   spark_dpins_in_init (MAC address, pin, initial value)
   ```

   Built-in Events:
   ```
   spark_dpin_in_on (MAC address, pin)
   spark_dpin_in_off (MAC address, pin)
   ```

Supported Actuators:

4. LED – (light-emitting diode)

   Built-in Tables:
   ```
   spark_leds (MAC address, pin)
   spark_leds_min (MAC address, pin, minimum value)
   spark_leds_max (MAC address, pin, maximum value)
   spark_leds_init (MAC address, pin, initial value)
   ```

   Built-in Event: `spark_led_set (MAC address, pin, value)`

5. Servo – (servo motor)

   Built-in Tables:
   ```
   spark_servos (MAC address, pin)
   spark_servos_min (MAC address, pin, minimum value)
   spark_servos_max (MAC address, pin, maximum value)
   spark_servos_init (MAC address, pin, initial value)
   ```

   Built-in Event: `spark_servo_write (MAC address, pin, value)`

6. DPinOut – (digital pin, output)

   Built-in Tables:
   ```
   spark_dpins_out (MAC address, pin)
   spark_dpins_out_init (MAC address, pin, initial value)
   ```

   Built-in Event: `spark_dpin_out_set (MAC address, pin, value)`