Reporte de Laboratorios

Mario Bolaños[1]
2017-2

[1]Ingeniería de Sistemas y Computación
Pontificia Universidad Javeriana Cali

mandres1@javerianacali.edu.co

1. Node Red y Ubidots

"Node-RED es una herramienta de programación para conectar dispositivos de hardware, API y servicios en línea de manera nueva e interesante"[1]. Con Node Red logré adentrarme a un mundo nuevo de programación que nunca antes había explorado, ya que realmente es distinto a lo que estás acostumbrado, pero una vez te acostumbras, te parece intuitivo y agradable de usar.

1.1. Laboratorio 1

El principal objetivo del laboratorio 1 fue el de contextualizar en dispositivos de IoT. Para ello se usó una Raspberry Pi 3 con un módulo de pruebas con sensores de humedad y temperatura para realizar las actividades propuestas.

La idea era hacer un flujo en Node Red que encendiera un led de color amarillo cuando la humedad estuviera por encima del 70 %, además el flujo debía encender el led de color verde cuando las condiciones de humedad y temperatura fueran las normales (humedad menor del 70 % y temperatura mayor a a 20C) y prendía el buzzer cuando la temperatura aumentaba. A continuación se muestra el flujo.

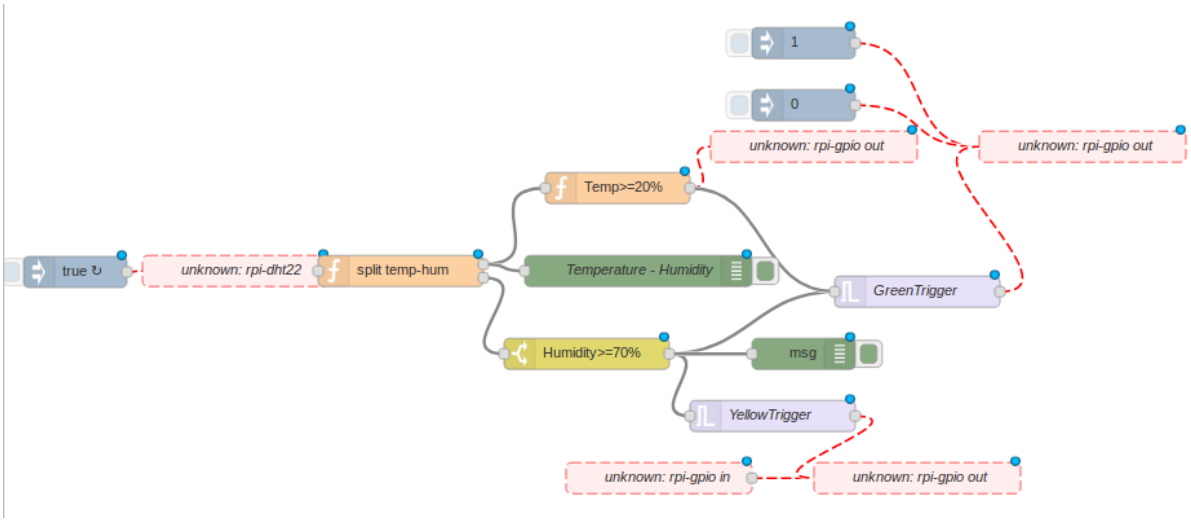

Figura 1. Flujo en Node Red

Con la ayuda de una función que dividía la temperatura y la humedad que arrojaba el flujo de la Pi en dos variables distintas, se pudo condicionar a éstas como se pedía en el enunciado del laboratorio.

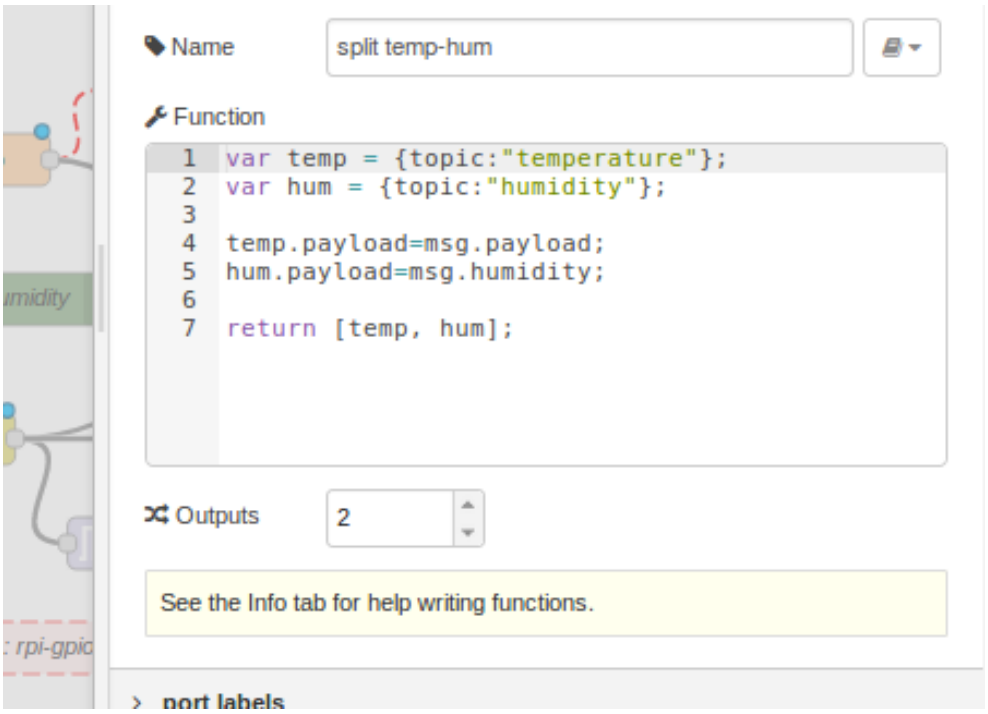

Figura 2. Flujo en Node Red

1.2. Laboratorio 2

En este laboratorio, se trabajó de igual manera con la Raspberry Pi 3 pero ahora impactando a Ubidots. La idea era hacer un flujo en Node Red para controlar desde Ubidots un límite para las dos variables (temperatura y humedad). Cuando el límite se sobrepasaba se prendía el buzzer y desde Ubidots era posible desactivarlo.

```
var temperatura  = context.get('temperatura') || 27;
var threshold = context.get('threshold') || 27;

var msgUbidots = {payload:
                    {temp:msg.payload,
                    humidity:msg.humidity}
                    };

var msgBuzzer = {topic:'cambioTemp', payload:0};

switch(msg.topic)
{
    case 'rpi-dht22':
        temperatura=msg.payload;
        context.set('temperatura',temperatura);
        break;
    case '/v1.6/devices/dev1/umbral/lv':
        threshold=msg.payload;
        context.set('threshold',threshold);
        break;
}

if(temperatura>threshold)
{
    msgBuzzer.payload=1;
}

return [msgUbidots, msgBuzzer];
```

Figura 3. JavaScript impactando a Ubidots

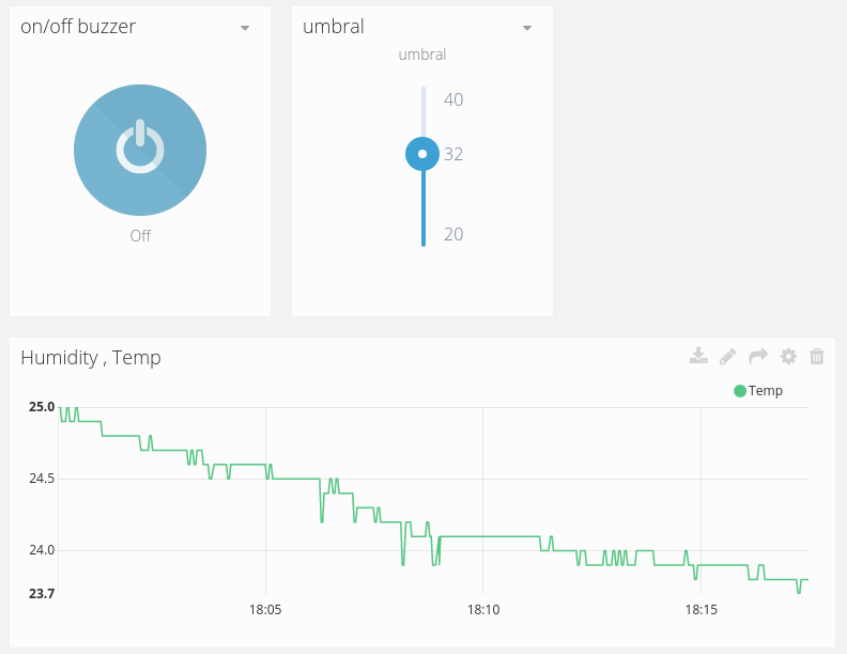

Figura 4. Variables en Ubidots

1.3. Laboratorio 3

El principal objetivo del laboratorio 3 fue el de entender cómo funciona la conectividad en IoT, para ello se usó MQTT (Mosquitto) y Wireshark.

Descripción: La idea es monitorear el tráfico de la red del protocolo de IoT MQTT con Wireshark observando los mensajes que son enviados con la sentencia *mosquitto_pub* al cliente que se suscribió previamente con *mosquitto_sub*.

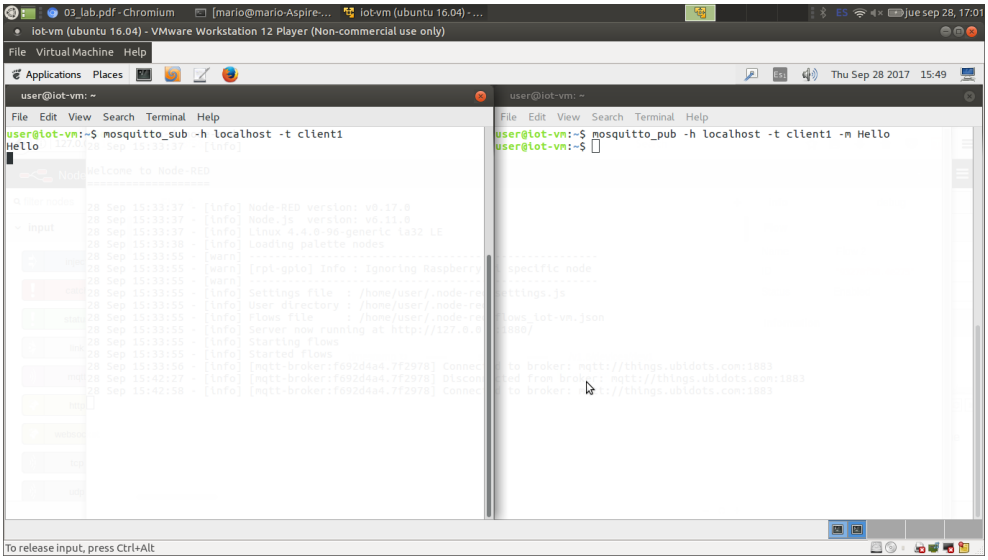

Figura 5. MQTT Comunicación

En Wireshark se puede observar el tráfico de la red a través del protocolo de MQTT.

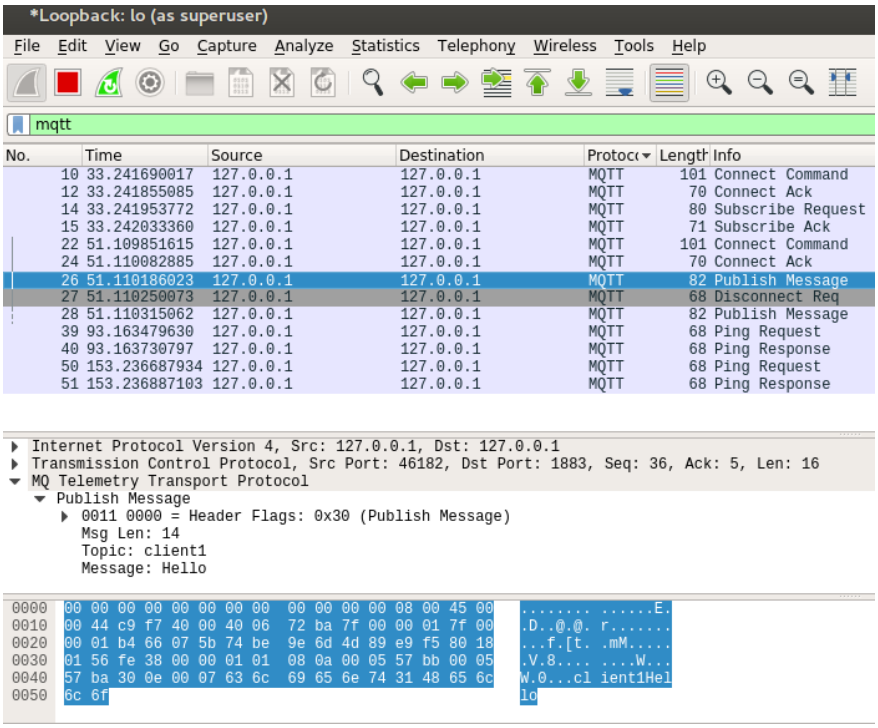

Figura 6. Tráfico MQTT en Wireshark

Luego de publicar un mensaje a Ubidots, es posible también monitorear este proceso en Wireshark, como se observa en la siguiente imagen.

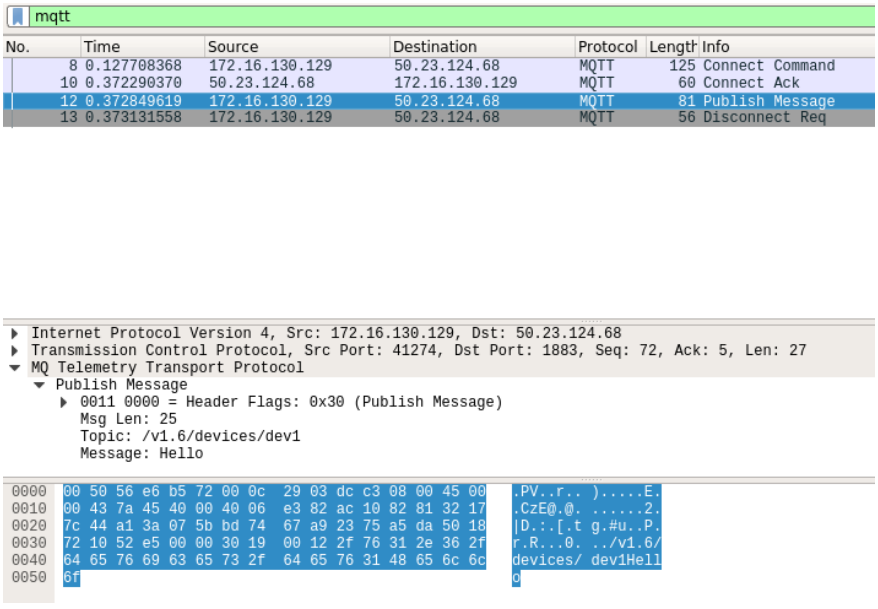

Figura 7. Tráfico MQTT Ubidots

En teoría cualquier persona podría acceder a los datos que se envían a través de la red ya que Node Red no es seguro, si obtienen la ip y el puerto en el que se está corriendo Node Red ya pueden monitorear el tráfico y ver lo que se está haciendo. En el settings.js de Node Red se pudo observar que hay una manera de volver más seguro esto con una contraseña.

2. MSC y SDL

El principal objetivo del laboratorio de MSC y SDL fue el de proponer una herramienta de modelado de sistemas de internet de las cosas. Así, con la ayuda de PragmaDev Studio se pudo modelar y simular el funcionamiento de una cafetera.

Antes de mostrar la solución de la cafetera en PragmaDev Studio, vale la pena dar un par de definiciones. “El lenguaje MSC (Message Sequence Chart) es un lenguaje gráfico y textual para la descripción y especificación de las interacciones entre los componentes de un sistema”[2]. Por su parte, SDL hace referencia a Specification and Descripting Language. El cual, valga la redundancia, “es un lenguaje dirigido a la especificación y descripción no ambigüa del comportamiento de sistemas reactivos y distribuidos”[3].

Siguiendo el ”Jumpstart Introduction to PragmaDev’s Real-Time Developer Studio and MSC Tracer”[4] del profesor Eugenio Tamura se implementó una cafetera básica que sigue el siguiente comportamiento (MSC):

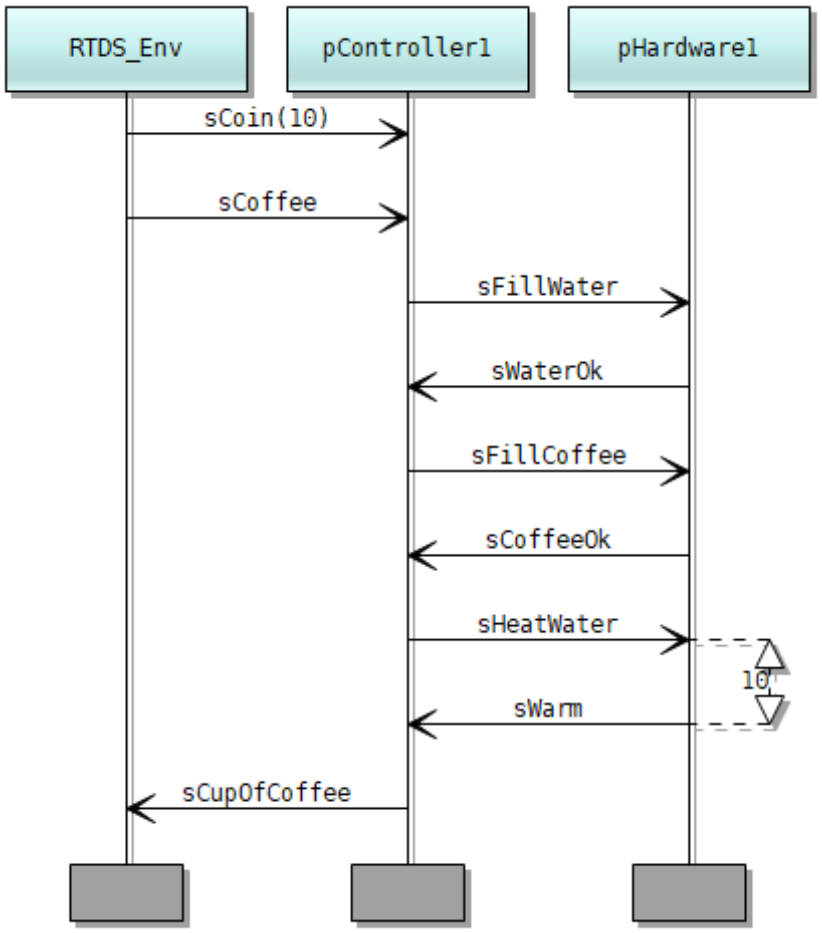

Figura 8. MSC Coffee Machine

Se crearon dos MSC más para definir el comportamiento de la cafetera junto con el té, como se muestra a continuación:

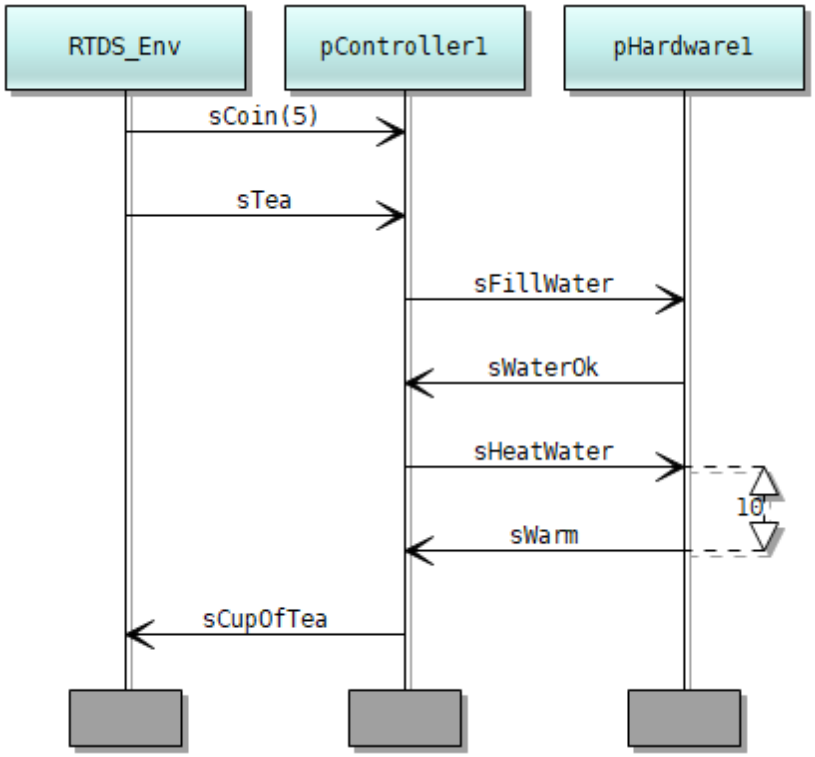

Figura 9. MSC Tea

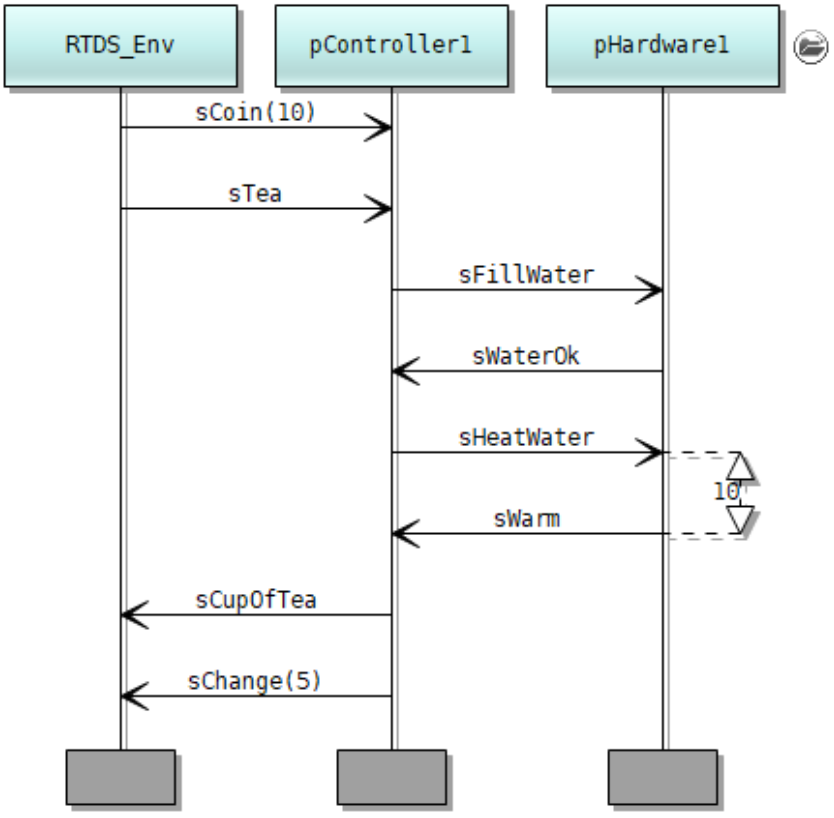

Figura 10. MSC Tea con cambio

La arquitectura del sistema de la cafetera luego de las modificaciones necesarias para que la cafetera también vendiera té estaba definida por las siguientes declaraciones de señales y tipos:

```
/* SYSTEM Coffee and Tea Machine */
```

```
/*A nickel is a coin with a face value of 5 cents*/
/*A dime is a coin with a face value of 10 cents*/
NEWTYPE CoinValues
  LITERALS NICKEL, DIME;
ENDNEWTYPE;

/*Product types for HotVendingMachine*/
NEWTYPE Products
  LITERALS COFFEE, TEA;
ENDNEWTYPE;
```

```
/* Signals from customer */
SIGNAL sCoffee, sTea, sCoin(CoinValues);

/* Signals to hardware */
SIGNAL sFillWater, sFillCoffee, sHeatWater;

/* Signals from hardware */
SIGNAL sWaterOk, sCoffeeOk, sWarm;

/* Signals to customer */
SIGNAL sCupOfCoffee, sCupOfTea, sChange(CoinValues), sMoreCoins;

/* Signals to cloud */
SIGNAL sSell(Natural), sSpent(Natural);
```

Figura 11. Declaraciones del sistema Coffee and Tea Machine

La arquitectura como tal del sistema de la cafetera con té se definía de la siguiente manera:

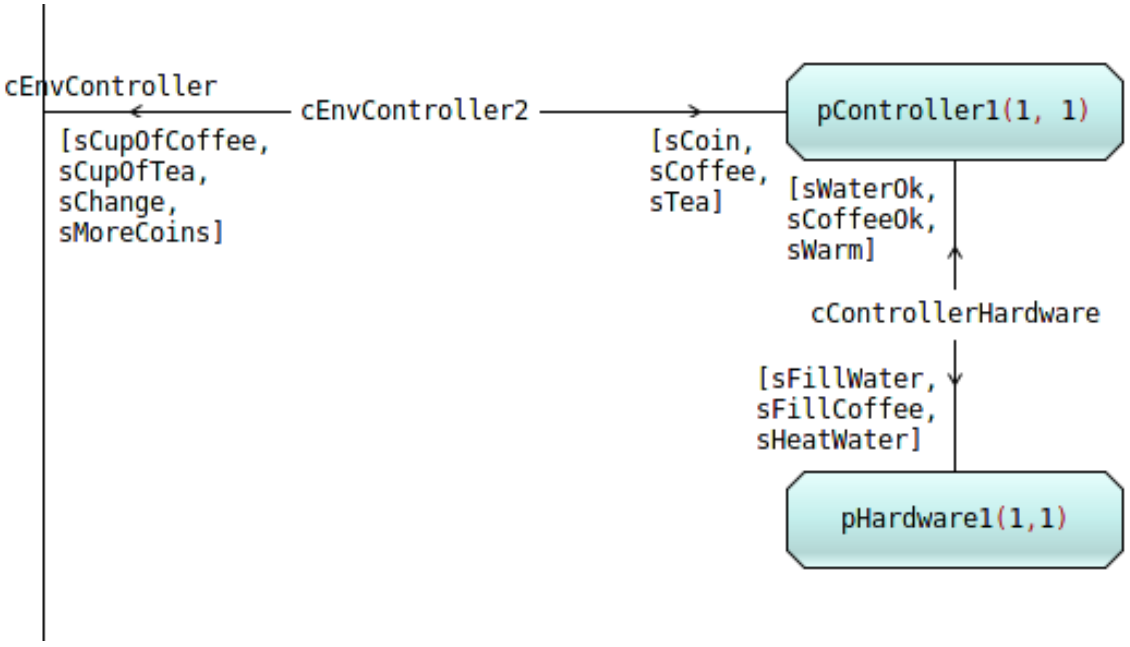

Figura 12. Sistema de Coffee Machine

El comportamiento del sistema de la cafetera con té se describe en pController1 y pHardware1. A continuación se muestra pController1:

```
/* PROCESS pController 1 */

DCL
  NbrOfCoffee Natural,          /* Number of coffee cups dispensed*/
  NbrOfTea    Natural,    /* Number of tea cups dispensed */
  Value       CoinValues, /* Coin face value */
  UserChoice  Products;  /*Choice of user (coffee or tea)*/
```

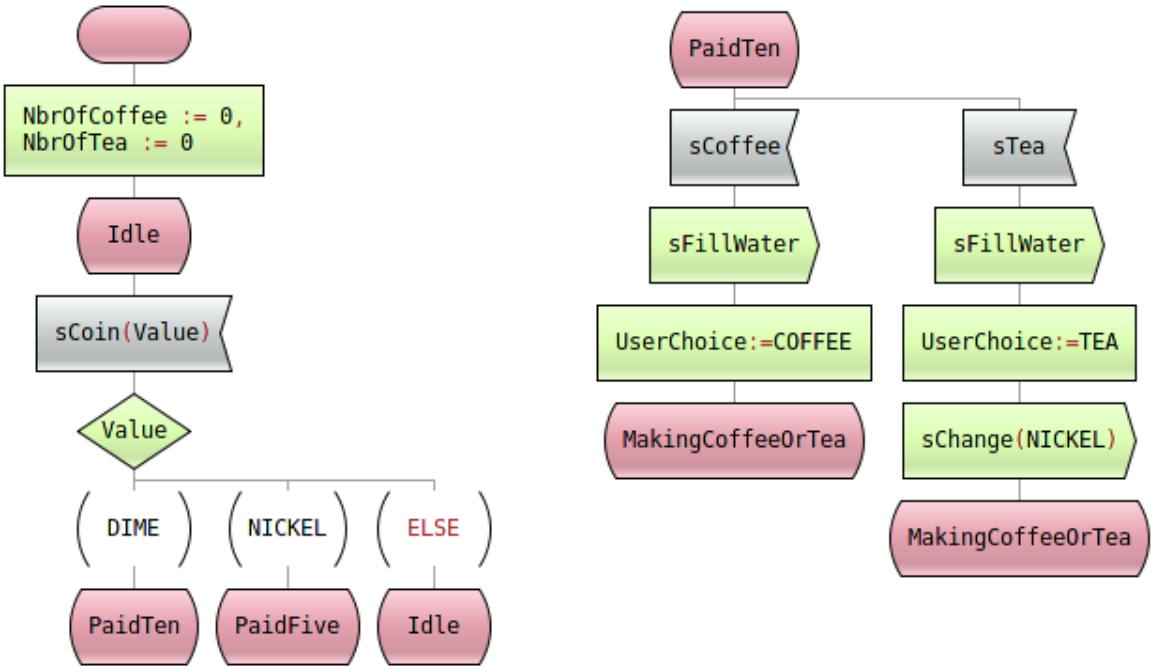

Figura 13. SDL pController1 Coffee and Tea Machine

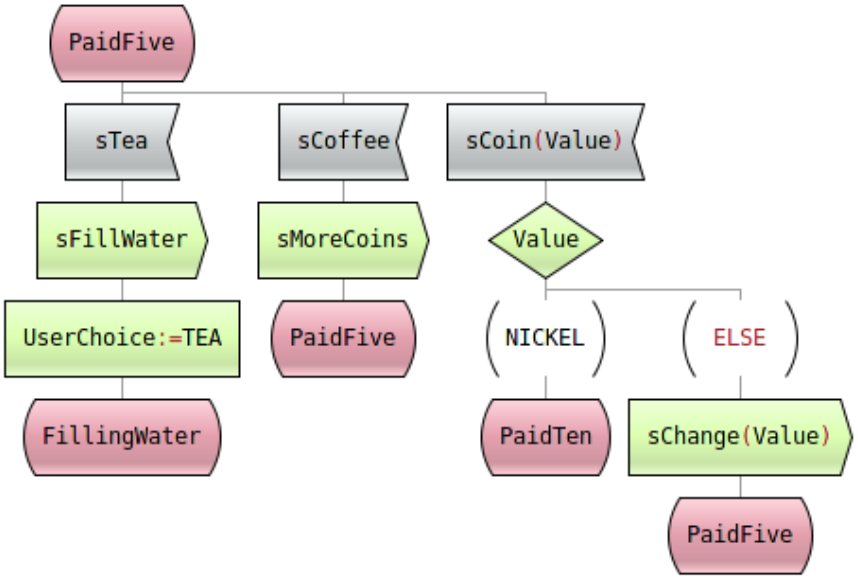

Figura 14. SDL pController1 Coffee and Tea Machine

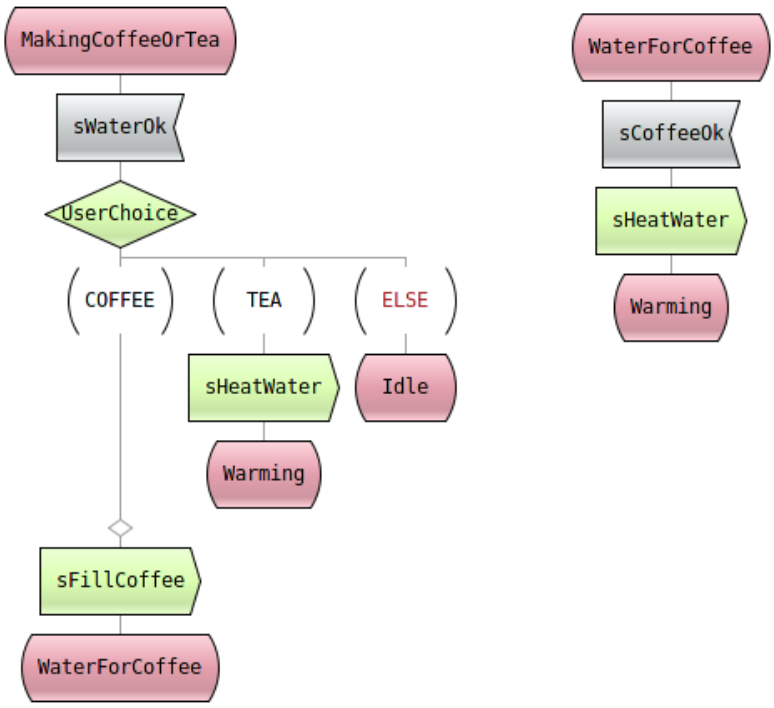

Figura 15. SDL pController1 Coffee and Tea Machine

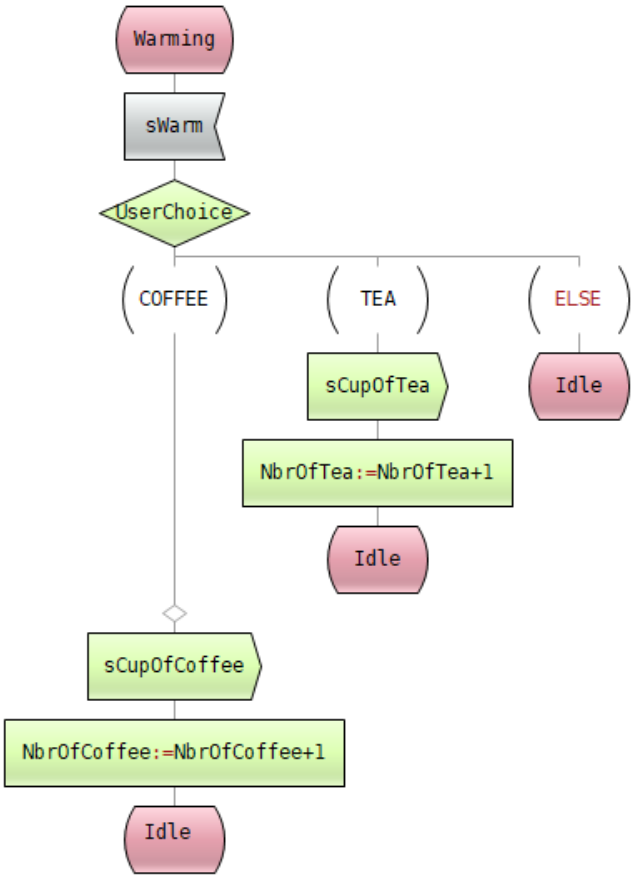

Figura 16. SDL pController1 Coffee and Tea Machine

El SDL de pHardware1 se definió de la siguiente manera:

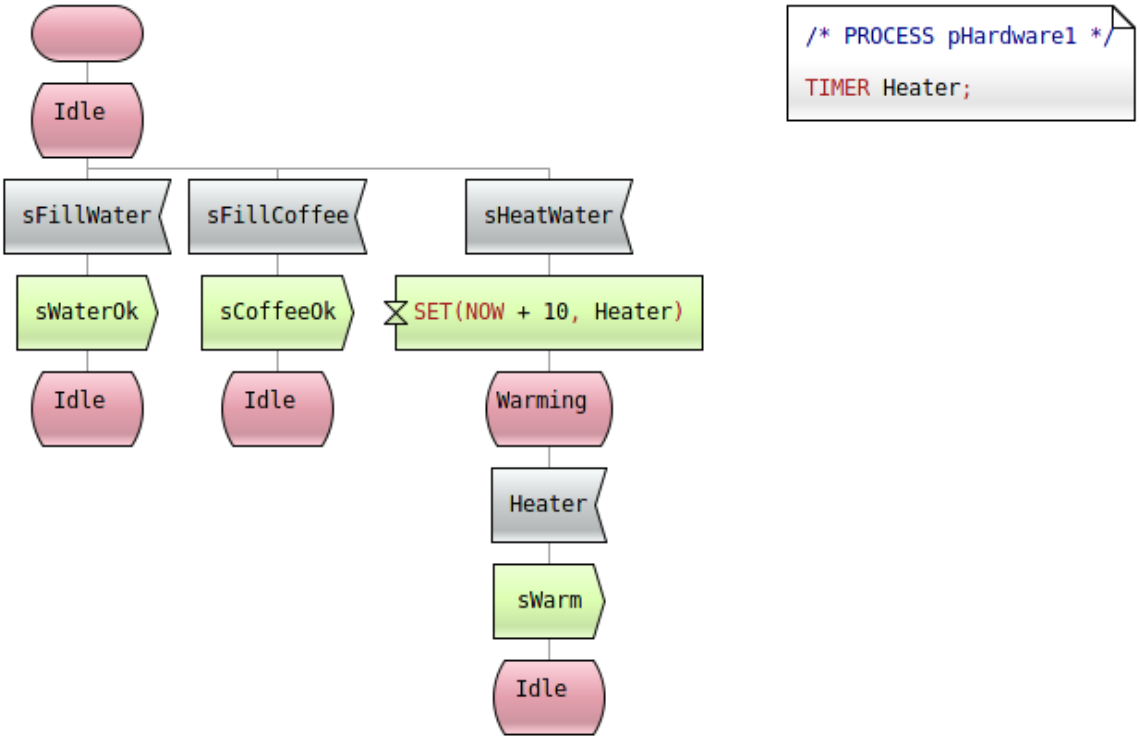

Figura 17. SDL pHardware1 Coffee and Tea Machine

Hasta aquí se tiene la cafetera con té. En el archivo adjunto a este documento de reporte, estará el comprimido con los diagramas de MSC y SDL correspondientes a lo anterior, de igual manera, se podrán observar las trazas correspondientes a la compra de café y té en la cafetera.

En el laboratorio de Coffee IoT, se requiere extender este modelo a la nube, partiendo de la siguiente arquitectura de sistema:

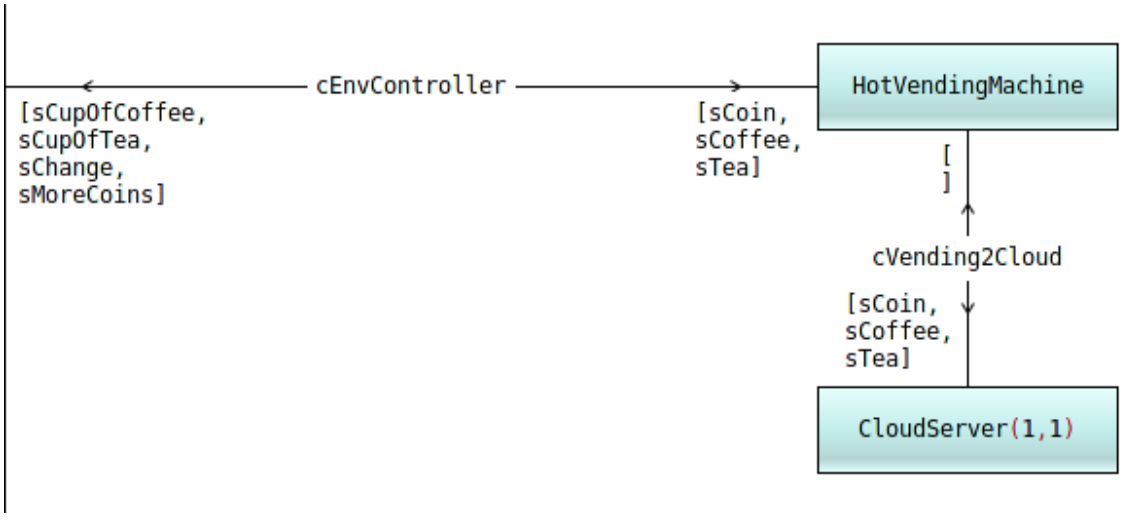

Figura 18. Arquitectura del sistema Coffee and Tea Machine

En donde el bloque de HotVendingMachine contiene lo que se tenía hasta el momento con la cafetera y el té y el bloque CloudServer contiene el comportamiento del componente de nube.

Se creo un MSC para reportar al CloudServer las ventas y gastos de varias cafeteras, que se muestra a continuación:

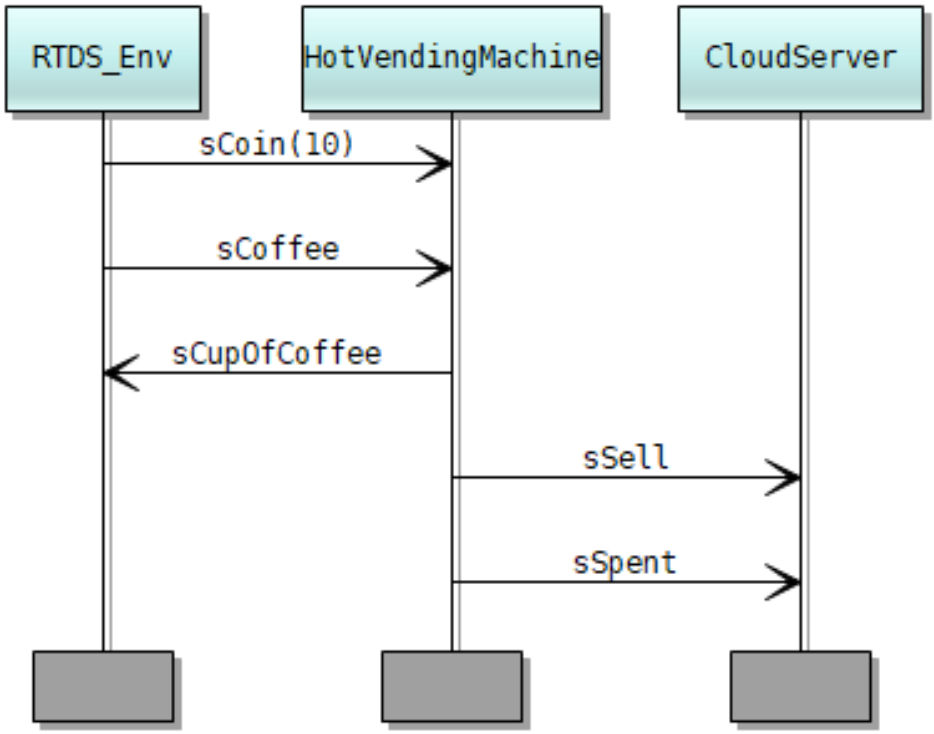

Figura 19. MSC Product Selling

Junto con el MSC, se tuvo que crear un modelo de datos para guardar la información de varias cafeteras que reportan al CloudServer sus ventas y gastos. Para ésto, se decidió crear un Struct que contuviera para cada cafetera precisamente lo gastado y lo vendido.

```
NEWTYPE CoffeeMachine
  STRUCT
     IDCM NATURAL;
     SELL  NATURAL;
     SPENT NATURAL;
ENDNEWTYPE;
```

Figura 20. CoffeeMachine Struct

Fue necesario crear un ID para cada cafetera que se mandó como parámetro de las señales sSpent y sSell para reconocer la cafetera a la que había que incrementarle el reporte de compras y ventas

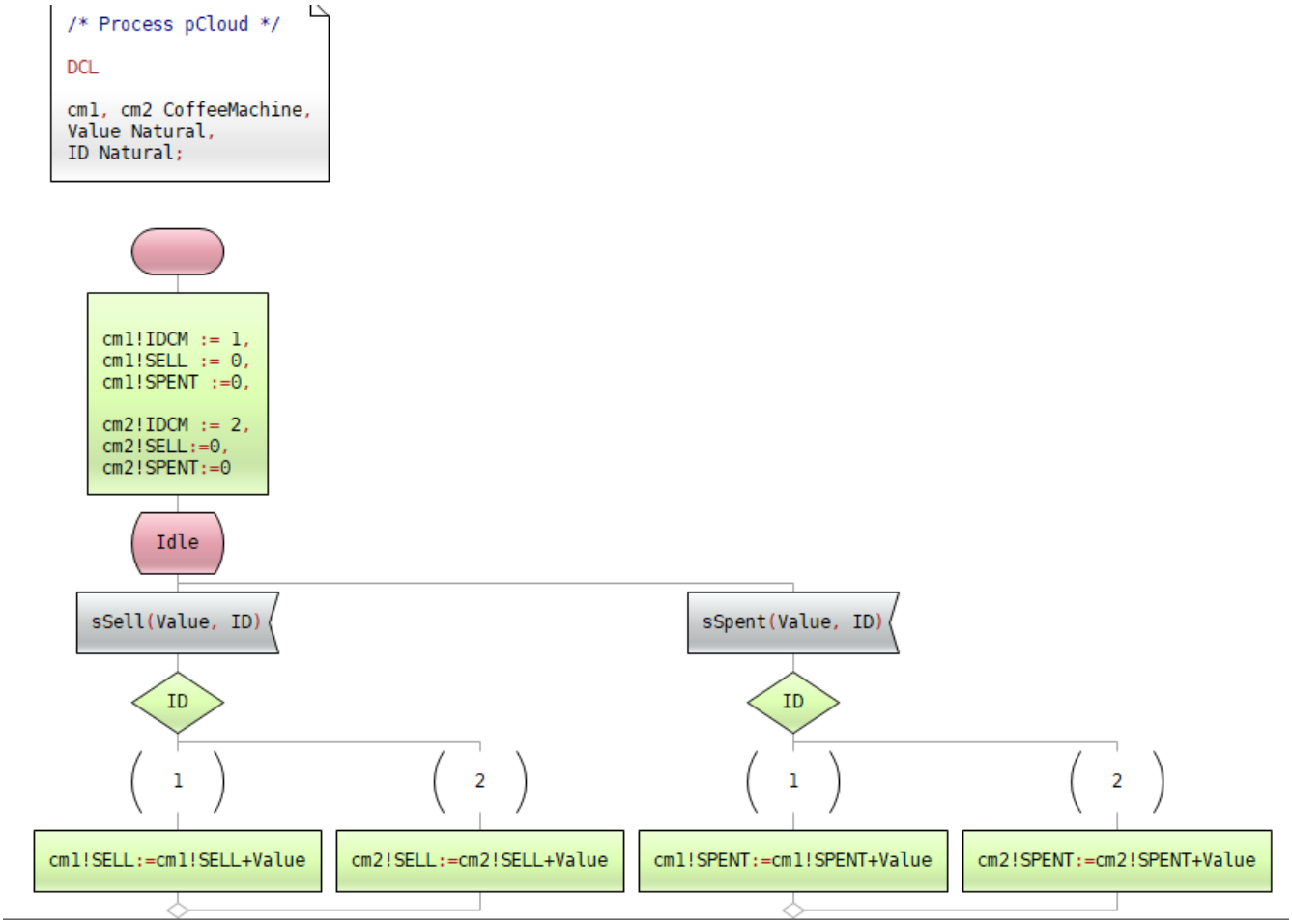

Figura 21. Proceso pCloud

Como conclusión acerca de MSC y SDL, puedo afirmar que constituyen una herramienta muy valiosa para modelar sistemas y específicamente sistemas de IoT. No conocía de software como PragmaDev y definitivamente mis impresiones a pesar de los problemas y pequeños ”bugs”que pueda tener la aplicación, son positivas.

3. Contiki

“Contiki es un sistema operativo de código abierto para Internet de las Cosas. Contiki conecta pequeños microcontroladores de bajo costo y bajo consumo a Internet. Contiki es una poderosa caja de herramientas para construir sistemas inalámbricos complejos”[5].

Dado el potencial que tiene este sistema operativo, se presentó en el curso como una alternativa a usar para desarrollar proyectos de IoT. El principal objetivo del laboratorio de Contiki fue el de adentrarnos en este OS para embebidos mediante la implementación del sistema modelado previamente en MSC-SDL de la cafetera.

Basándome en el código del profesor Andrés Rudqvist[6], implementé las funcionalidades adicionales del sistema de la cafetera que modelé en SDL para que vendiera a su vez té. A continuación presento unas imágenes correspondientes a lo que se editó del código proporcionado por el profesor:

En principio, fue necesario crear un archivo de recurso para el té, “res-tea.c”que sería activado desde el coffeeMachine.c

```c
/**
 * \file
 *      Coffee Machine example - tea
 * \author
 *      Mario Bolaños <mandres1@javerianacali.edu.co>
 * Based on
 *      Andres Rudqvist <asrudqvist@javerianacali.edu.co>
 */

#include "contiki.h"
#include <string.h>
#include "contiki.h"
#include "rest-engine.h"

#include "coffeeMachine.h"

extern process_event_t sTea;
extern struct process pController;

static void res_post_handler(void *request, void *response, uint8_t *buffe

/* Ask for tea */
RESOURCE(resTea,
         "title=\"Ask for tea\";rt=\"Signal\"",
         NULL,
         res_post_handler,
         NULL,
         NULL);

static void
res_post_handler(void *request, void *response, uint8_t *buffer, uint16_t p
{
    process_post (&pController, sTea, NULL);
    printf("post sTea\n");
}
```

Figura 22. Recurso res-tea.c

Se editaron los estados, con los correspondientes al SDL desarrollado, incluyendo un estado para pagar con cinco centavos y cambiando los nombres de algunos otros:

```
typedef enum
{
  Idle,
  PaidTen,
  PaidFive,
  MakingCoffeeOrTea,
  WaterForCoffee,
  Warming
} PCONTROLLER_STATES;
```

Figura 23. Estados

Asimismo, fue necesario crear las señales adicionales que se tenían en el modelo de SDL:

```
/*************** signals ***************************************************/
/* Signals from UI */
process_event_t sCoffee, sTea, sCoin;
/* Signals to Hardware */
process_event_t sFillWater, sFillCoffee, sHeatWater;
/* Signals from Hardware */
process_event_t sWaterOk, sCoffeeOk, sWarm;
/* to the customer */
process_event_t sCupOfCoffee, sCupOfTea, sChange, sMoreCoins;
```

Figura 24. Señales

Se puso una condición más en el proceso del pUserInterface para llamar el event handler cuando el ev fuera la señal del té:

```
while(1) {
  /* Wait forever */
  PROCESS_YIELD();

  if(ev == sCupOfCoffee) {
    /* Call the event_handler for this application-specific event. */
    printf("A cup of coffee ready to drink\n");
    cup_event.trigger();
  }
  else if(ev==sCupOfTea) {
    printf("A cup of tea ready to drink\n");
    cup_event.trigger();
  }
}
```

Figura 25. Proceso de pUserInterface

En el proceso del controlador se añadió la condición para para sCoin, de ser una moneda de 5 centavos:

```
case Idle:
  if (InSignal == sCoin)
  {
      //coin = (COINS)value;
      printf("pController: sCoin(%d)\n",value);
      if (value == 10){
        state_next = PaidTen;
        printf("pController: Paid Ten\n");
      }
      else if (value == 5){
        state_next = PaidFive;
        printf("pController: Paid Five\n");
      }
      else{
        state_next = Idle;
      }

  }
  break;
```

Figura 26. Estado Idle

Se modificó también el estado PaidTen y se creó el estado PaidFive siguiendo lo descrito en SDL:

```
case PaidTen:
  if ( InSignal == sCoffee )
  {
      printf("pController: sCoffee\n");
      process_post (&pHardware, sFillWater, NULL);
      userChoice = Coffee;
      state_next = MakingCoffeeOrTea;
      printf("pController: Making Coffee\n");
  }
  if ( InSignal == sTea )
  {
      printf("pController: sTea\n");
      process_post (&pHardware, sFillWater, NULL);
      userChoice = Tea;
      state_next = MakingCoffeeOrTea;
      process_post(&pHardware,sChange,NICKEL);
      printf("pController: Making Tea\n");
  }
case PaidFive:
  if ( InSignal == sTea )
  {
      printf("pController: sTea\n");
      process_post (&pHardware, sFillWater, NULL);
      userChoice = Tea;
      state_next = MakingCoffeeOrTea;
  }
  if (InSignal == sCoffee )
  {
      printf("pController: sCoffee\n");
      process_post (&pHardware, sMoreCoins, NULL);
  }
```

Figura 27. Estados PaidTend y PaidFive

El estado MakingCoffeeOrTea del proceso del controlador se expandió para decidir qué hacer dependiendo de la elección del usuario:

```
case MakingCoffeeOrTea:
  if ( InSignal == sWaterOk )
  {
      printf("pController: sWaterOk\n");
      if (userChoice == Coffee)
      {
          process_post (&pHardware, sFillCoffee, NULL);
          state_next = WaterForCoffee;
          printf("pController: Water For Coffee\n");
      }
      else if (userChoice == Tea)
      {
          process_post (&pHardware, sHeatWater, NULL);
          state_next = Warming;
      }
      else
      {
          state_next = Idle;
      }
  }
```

Figura 28. Estado MakingCoffeeOrTea

De igual manera, el estado Warming se editó para añadir la funcionalidad del té con la condición que depende de la elección del usuario. En ambos casos posibles para el usuario, se incrementa el contador correspondiente a café o té:

```
case Warming:
  if ( InSignal ==  sWarm)
  {
      printf("pController: sWarm\n");
      if (userChoice == Coffee)
      {
          process_post (&pUserInterface, sCupOfCoffee, NULL);
          state_next = Idle;
          nbrOfCoffee ++;
          printf("pController: coffee Ready (%d)\n",nbrOfCoffee);
      }
      else if (userChoice == Tea)
      {
          process_post (&pUserInterface, sCupOfTea, NULL);
          state_next = Idle;
          nbrOfTea ++;
          printf("pController: tea Ready (%d)\n",nbrOfTea);
      }
      else
      {
          state_next = Idle;
      }
  }
```

Figura 29. Estado Warming

En el proceso de hardware, se especificaron los dos estados del mismo, en donde se hace un post al proceso del timer que se mostrará en breve. Aquí se observan los cambios del proceso hardware:

```
switch ( state )
{
case Idle:
    if (InSignal == sFillWater)
    {
        printf("pHardware: post sWaterOk\n");
        process_post (&pController, sWaterOk, NULL);
        state_next = IdleH;

    }
    if (InSignal == sFillCoffee)
    {
        printf("pHardware: post sCoffee Ok\n");
        process_post (&pController, sCoffeeOk, NULL);
        state_next = IdleH;
    }
    if (InSignal == sHeatWater)
    {
        printf("pHardware: post sWater\n");
        OutSignal = (uint8_t) sSetTimer;
        value = 10;
        process_post (&pTimer, OutSignal, &value);
        state_next = Warming;
    }
    break;
case WarmingH:
    if (InSignal == Heater_et)
    {
        process_post (&pController, sWarm, NULL);
        state_next = IdleH;
    }
    break;
```

Figura 30. Proceso del Hardware

Se tuvo que crear un proceso para el timer que se manejó en SDL en el pHardware, éste proceso debe hacer post al ya mencionado proceso de hardware:

```
PROCESS_THREAD ( pTimer, InSignal, data ) {
    static PTIMER_STATES state,
           state_next;
    static struct etimer Heater_et;
    static uint8_t OutSignal,
                   value;
    PROCESS_BEGIN ();
    state_next = IdleT;
    for ( ; ; )
    {
        PROCESS_YIELD ();
        state = state_next;
        /* get value */
        value = *( (uint8_t *) data );
        switch ( state )
        {
          case IdleT:
            if ( InSignal == sSetTimer)
            {
                /* get value */
                printf("pTimer 1: &value = %p\n",data);
                printf("pTimer 1: value = %d\n",(*(uint8_t*)data));
                value = *((uint8_t *)data);

                printf("Timer set to %u secods\n",value);
                printf("pTimer 2: &valule = %p\n",&value);
                printf("pTimer 2: &valule = %d\n",value);

                etimer_set (&Heater_et, CLOCK_SECOND * value);
                state_next = WaitTimeout;
            }
            break;
          case WaitTimeout:
            if ( InSignal == PROCESS_EVENT_TIMER)
            {
                value = 0;
                printf("Timer timeout (send signal to pHardware)\n");
                process_post (&pHardware, sT, &value);
                printf("sT\n");
                state_next = IdleT;
            } else if ( InSignal == sResetTimer) {
                printf("Timer reset\n");
                etimer_stop (&Heater_et);
                state_next = IdleT;
            }
            break;
          default:
            printf("pTimer: unknown state");
```

Figura 31. Proceso del Timer

Referencias

[1] N. Red, “Flow-based programming for the internet of things.” https://nodered.org/.

[2] S. F. Society, “What is an msc?.” http://www.sdl-forum.org/MSC/.

[3] Wikipedia, “Specification and description language?.” https://en.wikipedia.org/wiki/Specification-and-Description-Language.

[4] E. Tamura, “Jumpstart introduction to pragmadev real-time developer studio and msc tracer.” ”http://moodledecc.javerianacali.edu.co/aulavirtual/pluginfile.php/2436/mod-resource/content/1/RTDStut1.pdf”.

[5] “Contiki: The open source os for the internet of things.” http://www.contiki-os.org/.

[6] A. Rudqvist, “Código base de la cafetera en contiki,”