Informe de Taller de Sistemas Ciber-Fisicos

Despliegue de red IoT

Prueba de concepto de despliegue de una red IoT utilizando el framework FIWARE y el dispositivo KitIoT

Martin Pacheco

Facultad de Ingeniería, UDELAR

2020

Índice

Propuesta inicial

La siguiente es la propuesta que fue planteada inicialmente para el proyecto. Esta fue posteriormente adaptada en función del hardware disponible y el enfoque abordado, y se detalla en la sección siguiente "Proyecto Realizado".

Despliegue de Internet de las Cosas (IoT)

Tanto en la industria como en la academia, se ha establecido una arquitectura de facto para el paradigma de IoT, que consiste en dispositivos finales que generan información en su rol sensor y reciben comandos en su rol actuador. Esta información se transmite hacia/desde plataformas de IoT “en la nube” (públicas o privadas) a través de un componente específico de comunicación asincrónica (denominado broker), que hace disponibles los datos a módulos de almacenamiento y análisis, que eventualmente toman acciones. Los mayores desafíos a nivel de transmisión de datos en este nuevo paradigma son las necesidades de largo alcance y bajo consumo energético debido a que los dispositivos pueden estar ubicados en zonas de difícil acceso y sin la infraestructura necesaria.

Es en este contexto se propone impulsar un proyecto de campus inteligente vinculado a las Facultades de la zona del Parque Rodó: FIC, FCEA, FADU, FING, que despliegue una plataforma que permita a desarrolladores creativos implementar servicios útiles para estudiantes y para el barrio, por ejemplo, conocer el estado de ocupación de salones y comedores, estado de ciclovías, datos de tránsito, entre otros. Como punto de partida, se dispone de material de redes inalámbricas (Access Points, enlaces y otros equipos donados por el Plan Ceibal), buena capacidad de cómputo virtualizado, y recientemente se han adquirido dispositivos con tecnología LoRa, WiFi y Blutooth para ser utilizados como sensores y actuadores. El proyecto, además de desplegar infraestructura, buscará generar una plataforma cognitiva (cómputo y conocimiento) que permita, mediante una interfaz definida, generar aplicaciones por parte de terceros.

Proyecto Realizado

El proyecto tiene un primer objetivo de experimentar y generar documentación respecto al funcionamiento, uso y configuración del dispositivo KitIoT[1], una iniciativa del laboratorio de IoT de Antel, orientado por un lado a el público académico, como herramienta que impulse la realización de proyectos en torno a la Internet de las Cosas; y por otro lado al público de las PYMES y startups, siendo de apoyo para testear ideas y pasarlas rápidamente a producción.

Se destaca que este kit está diseñado de manera modular, con un core de bajo consumo, que soporta la conexión de un cargador solar, baterias y conectores USB, y se le puede adicionar dos posibles módulos de comunicaciones, uno con NB-IoT y Lte M, usando la red de cobertura nacional de Antel, y otra con LoRa, ZigBee y BLE. Además, es extensible mediante headers ST Morpho y Arduno, permiten expandir rápidamente sus funcionalidades.

El segundo objetivo del proyecto es realizar un prototipo de despliegue completo de punto a punto de una red IoT, oficiando el KitIoT como dispositivo final. Para esto se eligió usar la plataforma abierta de FIWARE[2], uno de los principales impulsores del desarrollo en IoT. Además es ideal para este taller porque posee tutoriales que permiten realizar un rápido despliegue de una red IoT y de los diferentes servicios que la componen, haciendo uso de la tecnología de contenedores Docker.

El resultado del proyecto es la presente documentación, sintetizando una experiencia de aplicación de algunas tecnologias de IoT, la recopilación de soluciones a diferentes problemas que surgieron durante el despliegue, configuración y programación, y un listado de ideas interesantes a profundizar o mejorar sobre este trabajo en futuros proyectos.

[1] https://kitiot.antel.com.uy/introduccion
[2] https://www.fiware.org

FIWARE

Fiware es una plataforma abierta y estándar, impulsada por la Unión Europea, que proporciona un entorno basado en OpenStack y un conjunto de API (llamados **Generic Enablers** - GE) que facilitan la conexión a IoT, procesan y analizan grandes volúmenes de datos en tiempo real o incorporan funciones avanzadas para la interacción del usuario. FIWARE propone e introduce la idea de tener un estándar único para recopilar, gestionar, publicar e informar sobre los cambios en la información que está siendo recolectada por la plataforma. Hasta el momento no existe un estándar a nivel mundial para poder manipular toda esta información de contexto que se recolecta. La ausencia de estándares provoca, por ejemplo, que una solución IoT diseñada para una ciudad no pueda ser aplicada en otra sin tener que realizar esfuerzos de adaptación que seguramente incrementarán significativamente los costos del proyecto. Este estándar se llama NGSI.

Arquitectura

Los componentes principales que forman la arquitectura de FIWARE son:

- **IDAS**: Es una implementación del Backend Device Management GE. Está formado por los IoT Agents, que son los encargados de traducir los protocolos específicos de IoT de los dispositivos (Ultralight 2.0, MQTT, LWM2M/CoAP, etc) al protocolo de información de contexto NGSI. No se necesita este componente si los dispositivos o gateways admiten de forma nativa la API NGSI. Al usar los agentes, los dispositivos se representan en FIWARE como entidades NGSI en un Context Broker.

- **Orion Context Broker:** Es una implementación del Publish/Subscribe Context Broker GE, que provee las interfaces NGSI9 y NGSI10. Mantiene representaciones virtuales de los dispositivos físicos. Tiene la capacidad de mediar entre los sensores y los consumidores de los datos que éstos generan. El Context Broker almacena los últimos datos generados por los dispositivos en MongoDB. Si se quiere almacenar el histórico de los datos generados porlos dispositivos es necesario utilizar algún GE que lo permita y conectarlo con el Context Broker. Por ejemplo, es posible conectar el Context Broker con Cygnus que le permite conectarse con algún medio de almacenamiento específico como HDFS, CKAN o MySQL.

- **IDM & Auth:** Brindan los mecanismos para garantizar la confiabilidad, seguridad y privacidad en la entrega y uso de servicios.

*Esta introducción a FIWARE fue tomada de la tesis de grado titulada "**Relevamiento de arquitecturas paradesarrollo de aplicaciones de Internet delas Cosas**"[3], se recomienda referirse a la misma para más detalles.*

[3] https://www.colibri.udelar.edu.uy/jspui/bitstream/20.500.12008/19035/1/2810.pdf

Arquitectura

Para el despliegue de la red IoT se tomó como referencia la red que se presenta en el tutorial de FIWARE: **IoT over MQTT**[4] que es parte de una **serie de tutoriales**[5] que presentan diferentes tecnologias del framework.

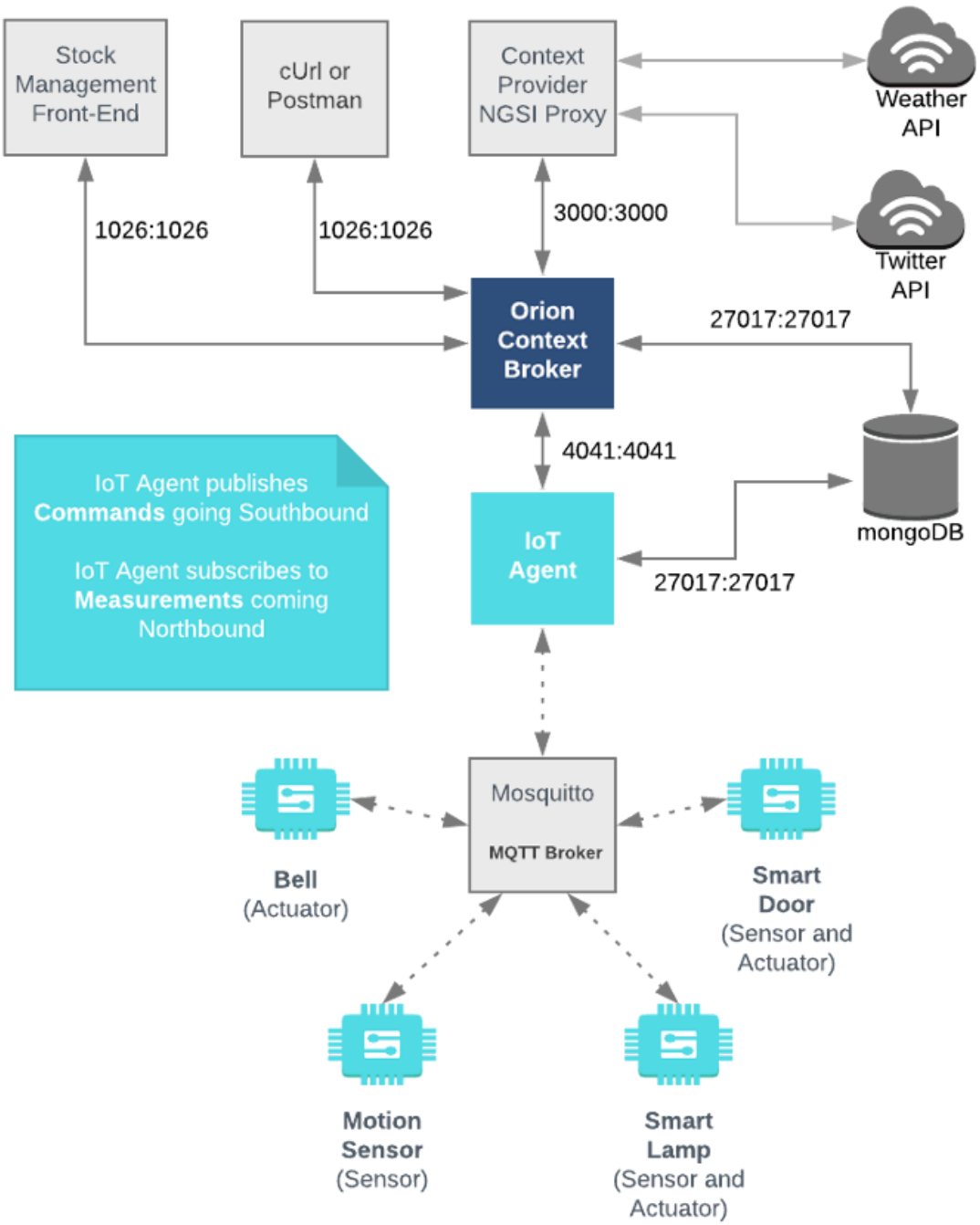
Este tutorial introduce el protocolo de comunicación **MQTT** basado en el modelo publicador/suscriptor que es muy usado para interactuar con dispositivos IoT.

Los elementos de la arquitectura son los siguientes:

- **Context broker → Orion:** recibe solicitudes mediante la API NGSI-v2
- **Broker MQTT → Mosquitto:** actua como un punto de comunicación central, pasando topicos MQTT entre el **Agente IoT** y los dispositivos.
- **Agente IoT** para UltraLight 2.0:
    - Recibe mensajes del southbound usando NGSI-v2 y los convierte a topicos MQTT de UltraLight 2.0 para el **broker MQTT**
    - Escucha en el **broker MQTT** por topicos a los que está registrados y envia medidas hacia northbound
- **Base de datos → MongoDB**:
    - Usada por Orion para almacenar información de contexto como entidades de datos, subscripciones y registros.
    - Usados por el **Agente IoT** para almacenar información de los dispositivos como URLs y claves
- **Servidor web:** actua como un conjunto de dispositivos dummy IoT usando el protocolo UltraLight 2.0 sobre MQTT, y presenta una interfaz para enviar comandos y ver mensajes e interacciones recibidas desde los dispositivos.

En la siguiente image se pueden ver todos los componentes y como interactuan entre ellos:

[4] https://fiware-tutorials.readthedocs.io/en/latest/iot-over-mqtt/index.html
[5] https://fiware-tutorials.readthedocs.io


Stock Management Front-End
cUrl or Postman
Context Provider NGSI Proxy
Weather API
Twitter API
1026:1026
1026:1026
3000:3000
Orion Context Broker
27017:27017
mongoDB
4041:4041
IoT Agent publishes **Commands** going Southbound
IoT Agent subscribes to **Measurements** coming Northbound
IoT Agent
27017:27017
Mosquitto
MQTT Broker
Bell (Actuator)
Smart Door (Sensor and Actuator)
Motion Sensor (Sensor)
Smart Lamp (Sensor and Actuator)

Hardware del KiT-IoT

Todos los detalles del hardware se pueden encontrar en la documentación oficial del kit[6].

- **Módulo base:** Está basado en el microcontrolador STM32 ARM Cortex M4[7] y se programa con el sistema Arm Mbed OS[8] que es un sistema embebido gratuito, open-source y diseñado especificamente para IoT.

- **Modem:** El shield NB-IoT del Kit_IoT se basa en el módulo BG96 de Quectel. Dicho módulo permite conectividad LTE Cat M1/Cat NB1/EGPRS, con tasas de transferencia de datos de hasta 375 kbps en subida y bajada. Se maneja mediante comandos AT a través de una interfaz UART. Un detalle importante es que requiere realizar una secuencia de encendido para utilizarlo, esto se explica en la documentación oficial y mas adelante en este documento.

- **Pines:** La placa cuenta con una variedad de pines para conectar otros dispositivos, sensores, etc. Se recomienda acceder a la documentación oficial para identificar cada uno de ellos. Un detalle importante es que la placa cuenta con reguladores de 3.3V y 5V controlados por el MCU que permiten mayor control en el consumo de energía de los sistemas que se conectan al Kit_IoT y estos deben ser encendidos.

[6] https://kitiot.antel.com.uy/disenohardware
[7] https://www.st.com/en/microcontrollers-microprocessors/stm32l433rc.html
[8] https://os.mbed.com/mbed-os/

Despliegue

Se realizó el despliegue de la infraestructura en una máquina virtual EC2 de AWS con el sistema operativo Amazon Linux AMI.

Siguiendo el tutorial mencionado anteriormente, para levantar la arquitectura es necesario tener instalado **git**, **docker**[9] y **docker-compose**[10]. Se puede seguir la siguiente guía para instalar **docker** en la AMI de AWS: Docker basics for Amazon ECS - Amazon Elastic Container Service

Y para instalar **docker-compose** se puden ejecutar los siguientes comandos:

```
sudo curl -L \
"https://github.com/docker/compose/releases/download/1.27.4/docker-co
mpose-$(uname -s)-$(uname -m)" \
-o /usr/local/bin/docker-compose
sudo chmod +x /usr/local/bin/docker-compose
```

Con los requerimientos instalados, la instalación de la infraestructura consiste en ejecutar los siguientes comandos:

```
  git clone https://github.com/FIWARE/tutorials.IoT-over-MQTT.git
  cd tutorials.IoT-over-MQTT
  git checkout NGSI-v2
  ./services create
```

Una vez completado el proceso, se levanta toda la infraestructura simplemente con el comando: `./services start` y para detenerla con el comando: `./services stop`

Esto levanta 5 contenedores docker con las imagenes:

- **fiware/orion** (puerto: 1026)
- **fiware/iotagent-ul** (puerto: 4041)
- **fiware/tutorials.context-provider** (puertos: 3000 y 3001)
- **eclipse-mosquitto** (puertos: 1883 y 9001)
- **mongo** (puerto: 27017)

[9] https://docs.docker.com/engine/install/
[10] https://docs.docker.com/compose/install/

Para mas detalles de la configuración de cada contenedor se puede consultar el archivo **docker-compose.yml** en la raiz del repositorio.

A partir de este momento ya es posible acceder a la interfaz web provista por la aplicación del contenedor **fiware/tutorials.context-provider** en el puerto 3000 desde donde se podrá ver el estado de los dispositivos IoT e interactuar con los mismos.

Para monitorear los logs del servidor mosquitto se puede ejecutar el comando:
`docker logs --tail 10 mosquitto`

Se aclara que en este informe se incluye solo una resumen de las secciones e indicaciones más relevantes del tutorial de fiware, pero se recomienda referirse al mismo para obtener mas detalles.

Programación del Kit IoT y e interacción con el modem

Preparación del entorno

Esta práctica fue realizada en el sistema operativo MacOS y las siguientes instrucciones son para este sistema operativo. Es necesario tener instalado previamente: el gestor de paquetes **brew**[11] (para Mac), **Python 3** y la utilidad **screen**.

De todas formas, pueden ser seguidas para Windows/Linux, cambiando la manera de instalar las dependencias y de identificar el puerto serial.

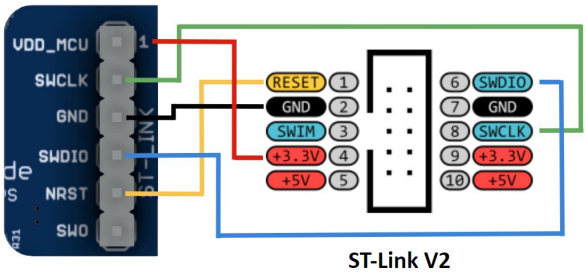
1. **Configurar entorno virtual** de Python:
   `python3 -m venv venv`
2. **Activar el entorno virtual**:
   `source venv/bin/activate`
3. **Instalar mbed-cli**[12]:
   `python3 -m pip install mbed-cli`
4. **Instalar compilador gcc arm:**
   `brew cask install gcc-arm-embedded`
5. **Configurar el GCC_ARM_PATH en mbed:**
   `mbed config -G GCC_ARM_PATH "/usr/local/bin"`
6. **Instalar stlink**:
   - Previamente se recomienda cambiar de versión con:
     `brew edit stlink` cambiando **1.6.1** por **1.6.0** (la versión**.1** dió problemas)
   - Luego si, se instala con:
     `brew install stlink`
7. **Instalar Mercurial:**
   `brew install mercurial`

[11] https://brew.sh/
[12] https://github.com/ARMmbed/mbed-cli/blob/1.8.3/README.md#installing-mbed-cli

Prueba de envio de mensajes mediante MQTT

Configuración

1. **Conectar Kit IoT a la computadora:**
   ***Nota**: Siempre que se quiera programar el dispositivo, ambos cables deben estar conectados.*
   - **Energía y Puerto Serial:** Conectar un cable USB al puerto micro-USB del Kit IoT.
   - **Programador:** Según las instrucciones de la documentación[13] se debe conectar el programador al Kit IoT según el siguiente diagrama, pero no es necesario conectar VDD_MCU (cable rojo) porque el cable USB ya provee la energía.



ST-Link V2

2. **Importación del proyecto nb-uart2terminal:**
   `mbed import https://gitlab.innova.antel.net.uy/kit-iot-publico/nb-uart2terminal`

3. **Acceder a la carpeta**:
   `cd nb-uart2terminal`

4. **Compilarlo con `mbed`:**
   `mbed compile -t GCC_ARM -m NUCLEO_L433RC_P && echo BUILD DONE`

   Nota: Durante el proceso puede que el sistema (MacOS) bloquee la ejecución de binarios por ser de un desarrollador desconocido, se debe aceptar (por única vez) la excepción para la ejecución para cada uno de los binarios.

5. **Flashear el código en el Kit IoT:**
   `st-flash write BUILD/NUCLEO_L433RC_P/GCC_ARM/nb-uart2terminal.bin 0x8000000`

6. **Conectarse a la interfaz serial de la placa con `screen`:**
   `screen /dev/tty.usbserial-14210 115200`

   Notas: El identificador del dispositivo es de la forma **tty.usbserial-*** y el número al final es variable (en MacOS). Al conectar el cable USB se debe identificar un dispositivo en **/dev/** con un nombre con este formato que puede cambiar de una conexión a otra.

[13] https://kitiot.antel.com.uy/disenohardware

7. **Enviar los siguientes comandos en la consola serial (en `screen`):**

   Nota: Escribir los comandos de a uno y esperar la respuesta.

   ```
   AT+CFUN=4
   AT+QCFG="nwscanmode",3,1
   AT+QCFG="iotopmode",1,1
   AT+CGDCONT= 1,"IP","testnbiot","0.0.0.0",0,0
   AT+CFUN=1
   AT+COPS=1,2,"74801",9
   AT+CEREG?
   AT+QNWINFO
   AT+CGACT=1,1
   AT+CGPADDR=1
   ```

   La ejecución del último comando debería de mostrar la IP que fue asignada. En caso de no hacerlo, puede demorar unos segundos en terminar de configurarse. Se puede repetir el último comando hasta que devuelva la IP.

Salida de la ejecución de comandos

Envio de mensajes (publicación)

Para el envio de mensajes de prueba usamos el servidor público MQTT de HiveMQ[14], alternativamente también se puede usar **test.mosquitto.org**, pero puede no estar operativo en ciertos momentos.

1. **En la consola serial ejecutar estos comandos para conectarse al servidor MQTT**
   - **Preparar red para conectarse**:
     `AT+QMTOPEN=0,"broker.hivemq.com",1883`
   - **Conectarse al servidor MQTT:**
     `AT+QMTCONN=0,"test1234"`
     "test1234" es el identificador del cliente, y este comando puede dar error si ya hay una conexión activa con ese identificador (cambiarlo por otro cualquiera en ese caso).
2. **Instalar mosquitto:**
   `brew install mosquitto`
3. **Conectarse a test.mosquito.org y subscribirse a kitiot/holaMundo:**
   `mosquitto_sub -h test.mosquitto.org -v -t "kitiot/holaMundo"`
4. **Enviar un mensaje desde la consola serial al topic kitiot/holaMundo:**
   Ingresar el comando `AT+QMTPUB=0,0,0,0,"kitiot/holaMundo"`, luego escribir el mensaje y enviarlo con **Ctrl + Z**
5. **Recepción del mensaje:**
   Se debería de recibir el mensaje en el comando `mosquitto_sub` ejecutado anteriormente.

Recepción de mensajes (subscripción)

1. **Conectarse al servidor MQTT**
   Luego de haber realizado el paso 1 de la explicación anterior para enviar mensajes, se debería tener una conexión activa, de lo contrario repetir los comandos para conectarse.
2. **Subscribirse a kitiot/holaMundo desde la consola serial:**
   `AT+QMTSUB=0,1,"kitiot/holaMundo",0`

[14] https://www.hivemq.com/public-mqtt-broker/

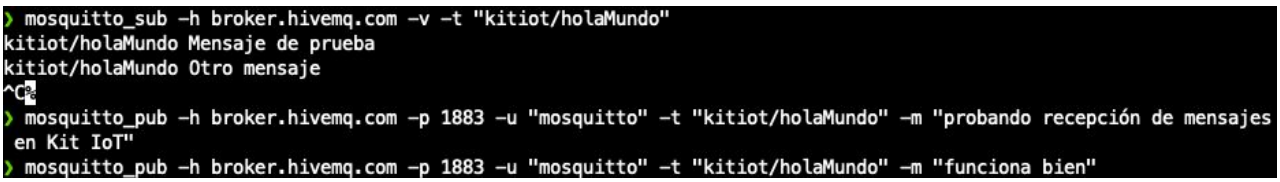
3. **Enviar mensaje con** `mosquitto_pub`:
   `mosquitto_pub -h broker.hivemq.com -p 1883 -u "mosquitto" -t "kitiot/holaMundo" -m "mensaje"`

4. Se deberían recibir los mensajes en la consola serial.

Ejemplo de envio y recepción

La siguiente imagen muestra el envio y recepción de mensajes en la consola serial:

Y los mismos mensajes siendo recibidos en mosquitto_sub y enviados por mosquitto_pub:



Implementación de un dispositivo IoT en el Kit IoT

Conexión automática del modem a la red NB-IoT

Una vez validada la conexión a la red Nb-IoT y el envio de mensajes MQTT mediante el ingreso manual de comandos AT por serial, se buscó la forma de realizar estas acciones mediante programación y el uso de APIs para abstraerse de la ejecución de comandos nativos del modem y lograr configurar la conexión de manera automática al encender el modem.

Alternativas

Para esto se buscaron implementaciones ya existentes que interactuen con el modem, envien mensajes MQTT y funcionen con el microcontrolador STM32. Se identificaron las siguientes:

1. Ejemplo de uso de API Celullar en Mbed OS: https://github.com/ARMmbed/mbed-os-example-cellular/
2. Instanciación del ejemplo (1) para usar el modem BG96: https://os.mbed.com/users/stkim92/code/mbed-os-example-cellular-BG96-TCP/
3. Repositorio de ejemplos usando BG96 en mbed de proovedor IoT Wiznet: https://github.com/Wiznet/wiznet-iot-shield-mbed-kr
4. Expansion Packages de ST (fabricante de STM32): https://www.st.com/en/embedded-software/stm32cube-expansion-packages.html
5. API para BG96 elaborada por Libelium para su producto propio Waspmote: https://github.com/Libelium/waspmoteapi
6. Librería desarrollada por ANTEL para Kit IoT: Existe una librería desarrollada por el laboratorio de IoT de Antel que por el momento se encuentra en versión alfa.

El repositorio (1) es un ejemplo oficial de mbed-os que hace uso de las API celullar y demuestra una conexión TCP o UDP con un servidor echo. El repositorio (2) es una aplicación de este ejemplo al modem BG96. El mismo sirvió de referencia para adaptarlo a el Kit IoT que usa el mismo modem, sin embargo se deben aplicar algunas modificaciones que se detallan en la sección **Conexión NB-IoT usando API Celullar de Mbed**.

En (3) hay varios ejemplos que usan BG96 y en particular hay uno de ellos en la carpeta **/samples**: WIZnet-IoTShield-BG96-MQTT[15] que implementa un wrapper alrededor de los comandos AT utilizados para comunicarse mediante MQTT usando el modem.

[15] https://github.com/Wiznet/wiznet-iot-shield-mbed-kr/tree/master/samples/WIoT-QC01_BG96/WIZnet-IoTShield-BG96-MQTT

Los Expansion Packages (4) son proyectos con ejemplos de código brindados por ST (el fabricante del chip STM32) con diferentes aplicaciones. Estos se analizaron con menor detenimiento que los anteriores, debido a que las implementaciones resultaron poco mas complejas de analizar y no estan en el formato de proyecto Mbed. De todas maneras se identificaron 3 paquetes que tienen implementada una API para interactuar con el modem y podrían ser de utilidad:

- **X-CUBE-CELLULAR:** https://www.st.com/en/embedded-software/x-cube-cellular.html
- **X-CUBE-CLD-GEN:** https://www.st.com/en/embedded-software/x-cube-cld-gen.html
- **X-CUBE-AZURE:** https://www.st.com/en/embedded-software/x-cube-azure.html

Se encontró otra implementación de una API para el modem (5), hecho por la Libelium para un producto propio llamado Waspmote. Lo destacable de la misma es que implementa llamadas para la mayoría de los comandos AT del modem (incluyendo los MQTT) y está implementada de una manera bastante simple, realizando gestión de errores, interpetando los mensajes de respuesta[16]. Esta implementación se podría portar para usar con Kit IoT o tomar partes del código de ejemplo.

Por último, sobre el final de esta búsqueda, se conoció que Antel está desarrollando una implementación propia de una API (6) para interactuar con el modem, esta aún no se encuentra pública pero se puede solicitar acceso a traves del canal de discord referenciado en la página del KitIoT.

[16] https://github.com/Libelium/waspmoteapi/tree/master/libraries/BG96

Conexión a NB-IoT usando API Celullar de Mbed

Se decidió tomar como base el código de ejemplo provisto en el repositorio[17] de GitHub de ARMBed para conectarse a un modem en mbed-os, en conjunto con algunas de las modificaciones incluidas en otro repositorio[18] que hace uso del modem BG96.

Para esto primero importamos el proyecto mbed con el comando:
`mbed import https://github.com/ARMmbed/mbed-os-example-cellular mbed-tscf-v1`

Luego le aplicamos las siguientes modificaciones:

- Configuración de parámetros de la conexión NB-IoT y pines de datos del modem
- Encendido del modem

El código de este proyecto con las modificaciones se encuentra en la branch **v1** del repositorio: https://gitlab.fing.edu.uy/kitiot/mbed-tscf

Configuración de parámetros de la conexión NB-IoT y pines de datos del modem

Cambios a aplicar a `mbed_app.json` en `"target_overrides"."*"`:

- Se cambian todos los parámetros `lwip.*` y `ppp.*` a el valor **`false`** y solo se deja en **`true`** el parámetro: `lwip.ipv4-enabled`
- Se quita el parámetro `cellular.use-apn-lookup`
- Se cambian los parámetros:
  - `nsapi.default-cellular-plmn` por el valor `"\"74801\""`
  - `nsapi.default-cellular-apn` por el valor `"\"testnbiot\""`
- Se agregan cuatro nuevos parámetros:
  - `cellular.radio-access-technology` con el valor `9`
  - `QUECTEL_BG96.tx` con el valor `"PA_9"`
  - `QUECTEL_BG96.rx` con el valor "`PA_10"`
  - `QUECTEL_BG96.provide-default` con el valor `true`

Por último, se quita `DISCO_L496AG` de `target_overrides`.

Encendido del modem

Según la documentación de Kit-IoT, el encendido del modem BG96 requiere una secuencia de activación especifica. Por fortuna, el repositorio en el que nos basamos que usa el modem BG96 ya incluye una función de encendido del modem de ejemplo que se ejecuta al comienzo del código.

[17] https://github.com/ARMmbed/mbed-os-example-cellular/
[18] https://os.mbed.com/users/stkim92/code/mbed-os-example-cellular-BG96-TCP/

Adaptamos esta función en base a las instrucciones de la documentación de ANTEL[19] y en los nombres de los pines correspondientes, resultando en el siguiente código:

```
void BG96_Modem_PowerON(void) {
   DigitalOut BG96_RESET(PB_6);
   DigitalOut BG96_PWRKEY(PB_15);
   BG96_RESET = 1;
   BG96_PWRKEY = 1;
   ThisThread::sleep_for(1000); //duermo al micro por 1 segundo.
   BG96_PWRKEY = 0;
}
```

A esta función la invocamos al comienzo del main en el código para asegurarse que el modem esté encendido antes de ser configurado y conectado.

Ahora se compila el proyecto y se flashea al kitiot con los comandos:
`mbed compile -t GCC_ARM -m NUCLEO_L433RC_P && echo BUILD DONE`
`st-flash write BUILD/NUCLEO_L433RC_P/GCC_ARM/mbed-tscf-v1.bin 0x8000000`

Despues de hacer el flash, conectados al serial (`screen /dev/tty.usbserial-14230 115200`) del KitIoT deberiamos ver la siguiente salida:

```
BG96 Power ON
Establishing connection
.

Connection Established.
TCP: connected with echo.mbedcloudtesting.com server
TCP: Sent 4 Bytes to echo.mbedcloudtesting.com
Received from echo server 4 Bytes


Success. Exiting
```

Lo que confirma que se logró encender el modem, configurarlo y conectarlo y realizar una prueba de conexión exitosa al servidor ECHO.

[19] https://kitiot.antel.com.uy/ejemplos_codigo#secuencia-de-encendido-del-modem

Programación del cliente MQTT

Una vez lograda la prueba de concepto anterior se analizaron las maneras en la que se podían conectar el dispositivo mediante MQTT y se identificaron 2 opciones:

- Mediante la invocación directa de comandos nativos MQTT del modem BG96
- Mediante una librería de mbed-os que implementa un cliente MQTT

Se optó por la segunda, ya que implicaba menos código y no depende del modem utilizado. De todas maneras se podría implementar en un trabajo futuro la otra opción y evaluar la performance en cada caso. Se nota que la librería que está desarrollando el laboratorio de IoT de ANTEL toma el primer enfoque.

Comunicación MQTT usando librería de Mbed

Modificamos el código de **mbed-tscf v1** para que en lugar de conectarse a un servidor ECHO, ahora se haga una conexión MQTT. Si bien para la librería de mbed no hay disponible en la documentación un ejemplo completo de uso, nos basamos en 2 implementaciones de referencia que encontramos:

- La primera es un repositorio con un proyecto de ejemplo **HelloMQTT**[20] para la versión anterior de la librería MQTT en mbed-os[21], que si bien actualmente está deprecada, un gran porcentaje del código es reutilizable sin tener que realizarle modificaciones.
- El otro lugar donde encontramos ejemplos de uso, es en el repositorio de la librería actual MQTT para mbed OS, en la carpeta de TESTS[22].

Procedemos entonces a modificar el código de **mbed-tscf-v1**, y a este nuevo proyecto lo llamamos **mbed-tscf-v2**, lo hacemos con el comando: `mbed import mbed-tscf-v1 mbed-tscf-v2`

El código final modificado se encuentra en la branch **v2** del repositorio de gitlab: https://gitlab.fing.edu.uy/kitiot/mbed-tscf

Luego vamos a agregar la librería MQTT de mbed con el comando siguiente:
`mbed add https://github.com/ARMmbed/mbed-mqtt.git`

Los cambios agregados al código son los siguientes:

[20] https://os.mbed.com/teams/mqtt/code/HelloMQTT/
[21] https://os.mbed.com/teams/mqtt/
[22] https://github.com/ARMmbed/mbed-mqtt/tree/master/TESTS/mqtt

- En el archivo de configuración mbed_app.json se definen el servidor MQTT que se usará con las variables (hivemq es un servidor MQTT público de prueba[23]):
    - `mqtt-server-hostname` = `broker.hivemq.com`
    - `mqtt-server-port` = `1883`
- Se mueven todas las funciones auxiliares para el archivo `funciones.cpp`
- Se reemplaza la función `test_send_recv` por la nueva función `test_recv_mqtt`
- Esta función crea un socket TCP de igual manera que la anterior, pero al servidor MQTT
- Se instancia el cliente MQTT (`MQTTClient`) y se asocia con el socket.
- Se realiza la conexión del cliente al servidor MQTT.
- Se subscribe a un topic y queda consultando el loop por mensajes nuevos, que si los hay los imprime en terminal.

Para probar cargamos el código en la placa y esperamos a que se conecte al servidor MQTT. Luego de esto podemos enviar un mensaje por MQTT con el siguiente comando (ejecutado desde cualquier lugar que tengamos instalado mosquitto):

```
mosquitto_pub -h broker.hivemq.com -p 1883 -u "mosquitto" -t "kitiot/holaMundo" -m "funciona bien"
```

Y en la consola serial deberiamos ver la siguiente salida:

```
mbed-os-example-cellular
Built: Nov 25 2020, 11:56:30
[MAIN], plmn: 74801
BG96 Power ON
Establishing connection
..
Connection Established.
Connecting to MQTT...Connected to MQTT. Awaiting for message...
Message arrived: qos 0, retained 0, dup 0, packetid 24327
Payload funciona bien
```

confirmando la recepción correcta de los mensajes enviados.

[23] https://www.hivemq.com/public-mqtt-broker/

Comunicación con la infraestructura desplegada

Luego de haber logrado una prueba de concepto de comunicación mediante MQTT en el código de mbed-tscf **v2** anterior, el paso siguiente es integrarse con la infraestructura anteriormente levantada de FIWARE.

Se plantea para esta etapa lograr enviar mensajes mediante MQTT hacia esta infraestructura, y para ello primero es necesario aprovisionar el dispositivo IoT en ORION y luego enviar los mensajes desde el dispositivo IoT siguiendo el formato de Ultralight 2.0.
El aprovisionamiento del dispositivo se detalla en la sección **Connecting IoT Devices**[24] del tutorial e implica ejecutar los siguientes comandos (en la VM de AWS):

1. Aprovisionar un service group para MQTT:

```
curl -iX POST \
  'http://localhost:4041/iot/services' \
  -H 'Content-Type: application/json' \
  -H 'fiware-service: openiot' \
  -H 'fiware-servicepath: /' \
  -d '{
 "services": [
  {
  "apikey":       "4jggokgpepnvsb2uv4s40d59ov",
  "cbroker":      "http://orion:1026",
  "entity_type": "Thing",
  "resource":     ""
  }
 ]
}'
```

2. Aprovisionar un sensor de movimiento:

```
curl -iX POST \
  'http://localhost:4041/iot/devices' \
  -H 'Content-Type: application/json' \
```

[24] https://fiware-tutorials.readthedocs.io/en/latest/iot-over-mqtt/index.html#connecting-iot-devices

```
    -H 'fiware-service: openiot' \
    -H 'fiware-servicepath: /' \
    -d '{
   "devices": [
     {
     "device_id":   "motion001",
     "entity_name": "urn:ngsi-ld:Motion:001",
     "entity_type": "Motion",
     "protocol":     "PDI-IoTA-UltraLight",
     "transport":   "MQTT",
     "timezone":     "Europe/Berlin",
     "attributes": [
          { "object_id": "c", "name": "count", "type":
     "Integer" }
     ],
     "static_attributes": [
          { "name":"refStore", "type": "Relationship",
     "value": "urn:ngsi-ld:Store:001"}
     ]
     }
   ]
  }
  '
```

A partir de este momento, si se envia el siguiente mensaje MQTT al servidor Mosquitto, en la interfaz web disponible en el puerto 3000 accediendo a la ruta /device/monitor se podrán ver los mensajes recibidos por el sensor:

```
docker run -it --rm --name mqtt-publisher --network \
 fiware_default efrecon/mqtt-client pub -h mosquitto -m "c|1" \
 -t "/4jggokgpepnvsb2uv4s40d59ov/motion001/attrs"
```

La definicion del formato del topic que se debe enviar y el formato de los mensajes siguen la especificación de Ultralight 2.0 y se puede leer el detalle en la página de documentación, que en particular tambien indica ciertas consideraciones para usarlo con MQTT[25].

[25] https://fiware-iotagent-ul.readthedocs.io/

La idea es entonces lograr que el dispositivo KitIoT envie este mismo mensaje de manera automática.

El código final se encuentra el a branch **v3** del repositorio **mbed-tscf** en el GitLab.

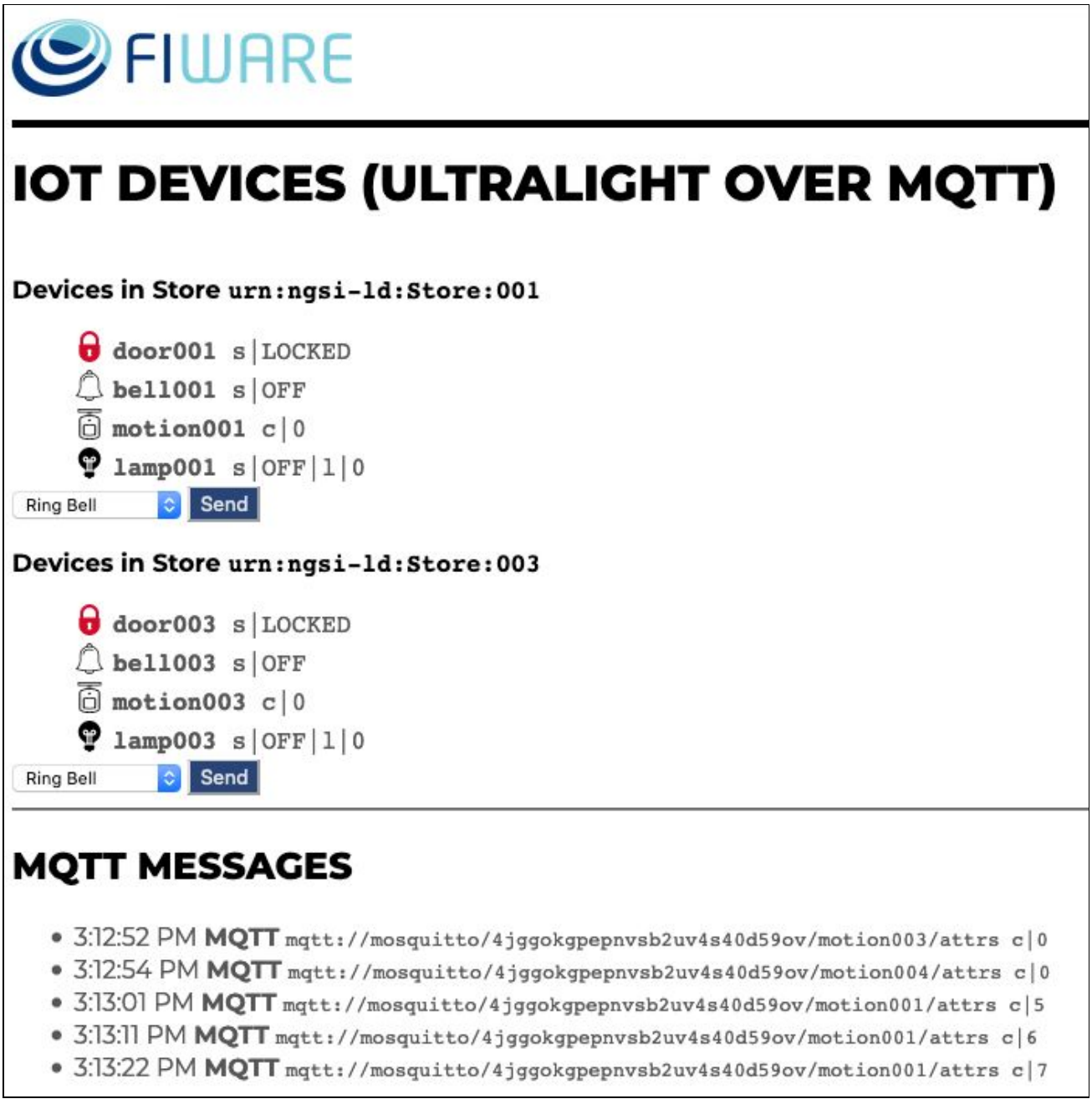
Se parte de la última versión del código (branch **v2**) y se realizan las siguientes modificaciones:

- En el archivo de configuración mbed_app.json se modifica el dominio del servidor MQTT por el de la infra de AWS (notar que es necesario configurar en el servidor para que el puerto 1883 sea accesible desde internet):
  - **mqtt-server-hostname** = ec2-18-221-73-186.us-east-2.compute.amazonaws.com
- Ahora en lugar de subscribirse a un tópico MQTT, se publica enviando un valor numérico incremental, de a intervalos de 10 segundos, esta modificación se puede ver en el cuerpo del **while(true)** de la función **test_recv_mqtt** (errata: su nombre ya no representa tu comportamiento, pero se olvidó de cambiar).

Posterior a cargar el código en el dispositivo, deberíamos ver los siguientes mensajes en la consola:

```
TCP: connected with 18.221.73.186 server
Connecting to MQTT...Connected to MQTT.
Sending the messages...
Sending the message: c|0
Sending the message: c|1
Sending the message: c|2
Sending the message: c|3
Sending the message: c|4
Sending the message: c|5
Sending the message: c|6
Sending the message: c|7
```

Y en la interfaz web ver los siguientes mensajes MQTT recibidos:


FIWARE
IOT DEVICES (ULTRALIGHT OVER MQTT)
Devices in Store urn:ngsi-ld:Store:001
door001 s|LOCKED
bell001 s|OFF
motion001 c|0
lamp001 s|OFF|1|0
Ring Bell
Send
Devices in Store urn:ngsi-ld:Store:003
door003 s|LOCKED
bell003 s|OFF
motion003 c|0
lamp003 s|OFF|1|0
Ring Bell
Send
MQTT MESSAGES
3:12:52 PM MQTT mqtt://mosquitto/4jggokgpepnvsb2uv4s40d59ov/motion003/attrs c|0
3:12:54 PM MQTT mqtt://mosquitto/4jggokgpepnvsb2uv4s40d59ov/motion004/attrs c|0
3:13:01 PM MQTT mqtt://mosquitto/4jggokgpepnvsb2uv4s40d59ov/motion001/attrs c|5
3:13:11 PM MQTT mqtt://mosquitto/4jggokgpepnvsb2uv4s40d59ov/motion001/attrs c|6
3:13:22 PM MQTT mqtt://mosquitto/4jggokgpepnvsb2uv4s40d59ov/motion001/attrs c|7

Envio y recepción de mensajes MQTT en paralelo

En este siguiente paso, la intención es lograr subscribrse de manera independiente a un tópico MQTT y publicar mensajes en otro. Esto nos permite implementar código de comunicación para usar por ejemplo un actuador y un sensor, y que puedan operar en paralelo, sin bloqueos entre ellos (cosa que pasaría si se usara un solo thread). En esta prueba se utilizó un sensor de movimiento como sensor y un buzzer como actuador.

Para comprender el uso de los threads en mbed-os se tomó de referencia un tutorial que hace uso de threads[26] y un código que hace uso del cliente MQTT de mbed-os, tambien con threads[27].

Se decidió definir el cliente MQTT (**MQTTClient**) usando una variable global para que puedan accederlo todos los threads que lo necesiten, y se usa un mutex para prevenir problemas de acceso concurrente, principalmente porque de hacerlo se puede provocar una excepción en la librería de red y crashear el programa.

Luego se definieron 2 funciones para los threads: **movementThread** y **bellThread** que son las responsables de enviar los datos del sensor de movimiento y recibir los comandos para activar el buzzer, respectivamente.

El código de esta parte se encuentra en la branch **v4** del repositorio y la explicación de como se programó el sensor de movimiento y actuador se explica en la siguiente sección.

En lo que respecta a FIWARE, en la parte anterior ya dimos de alta el sensor en ORION, pero ahora tambien necesitamos dar de alta el buzzer. Para esto ejecutamos el siguiente comando del tutorial indicados en la sección **"Provisioning an Actuator"**[28]:

```
curl -iX POST \
  'http://localhost:4041/iot/devices' \
  -H 'Content-Type: application/json' \
  -H 'fiware-service: openiot' \
  -H 'fiware-servicepath: /' \
  -d '{
  "devices": [
   {
   "device_id": "bell001",
```

[26] https://iotexpert.com/mouser-psoc-6-wifi-bt-mbed-l2-the-blinking-thread/
[27] https://github.com/ANRGUSC/mbed-esp8266-mqtt-example
[28] https://fiware-tutorials.readthedocs.io/en/latest/iot-over-mqtt/index.html#provisioning-an-actuator

```
    "entity_name": "urn:ngsi-ld:Bell:001",
    "entity_type": "Bell",
    "protocol": "PDI-IoTA-UltraLight",
    "transport": "MQTT",
    "commands": [
         { "name": "ring", "type": "command" }
         ],
         "static_attributes": [
         {"name":"refStore", "type":
    "Relationship","value": "urn:ngsi-ld:Store:001"}
    ]
    }
   ]
}
'
```

Luego cargamos el código **v4** en la placa, y ahora desde la interfaz web podemos enviar la acción "**Ring Bell**" para que en el dispositivo suene el buzzer:

El siguiente log en el dispositivo muestra que recibió el comando de ring 3 veces:

```
Sending the message: c|0
Sending the message: c|0
Message arrived: qos 0, retained 0, dup 0, packetid 36211
Payload bell001@ring|
Message arrived: qos 0, retained 0, dup 0, packetid 36211
Payload bell001@ring|
Sending the message: c|1
Message arrived: qos 0, retained 0, dup 0, packetid 36211
Payload bell001@ring|
Sending the message: c|1
```

Tambien en el log se ve el envio del de las medidas del sensor de movimiento (se envia cada 10 segundos) y en la interfaz web vemos los mensajes recibidos como ya lo haciamos antes, pero ahora con valores de un sensor real, significando 1 detección de movimiento:

```
MQTT mqtt://mosquitto/4jggokgpepnvsb2uv4s40d59ov/motion001/attrs c|0
MQTT mqtt://mosquitto/4jggokgpepnvsb2uv4s40d59ov/motion001/attrs c|0
MQTT mqtt://mosquitto/4jggokgpepnvsb2uv4s40d59ov/motion001/attrs c|1
MQTT mqtt://mosquitto/4jggokgpepnvsb2uv4s40d59ov/motion001/attrs c|1
MQTT mqtt://mosquitto/4jggokgpepnvsb2uv4s40d59ov/motion001/attrs c|1
```

Lectura de sensor de movimiento y actuación del buzzer

El sensor de movimiento es del tipo PIR y se pueden encontrar los detalles en su documentación[29]. Tiene 3 cables, siendo el de color rojo el de poder, que lo conectamos la fuente de 5V del KitIoT, el blanco es GND y el negro es el que nos provee las medidas del sensor, funcionando como un colector abierto.

El cable negro lo conectamos a un pin I/O del KitIoT y lo configuramos como pull up con el siguiente código:

```
DigitalIn movement(D5);
movement.mode(PullUp);
...
movement.read();
```

La lectura de las medidas se hace con movement.read() en donde si el valor es 1 no hay movimiento, y si es 0 hay movimiento.

Para hacer sonidos con el buzzer, se pueden utilizar librerias que mediante PWM logran reproducir ciertos tonos, sonidos, melodias, etc. En esta caso se encontró una librería simple llamada beep[30] que se puede agregar con el comando:

```
mbed add http://os.mbed.com/users/dreschpe/code/beep/
```

Al código de la librería se le debe hacer un único cambio para que funcione en versiones nuevas de **mbed-os** que es modificar el archivo **/beep/beep.cpp** cambiando la línea:
`toff.attach(this,&Beep::nobeep, time);`
por:
`toff.attach(callback(this,&Beep::nobeep), time);`

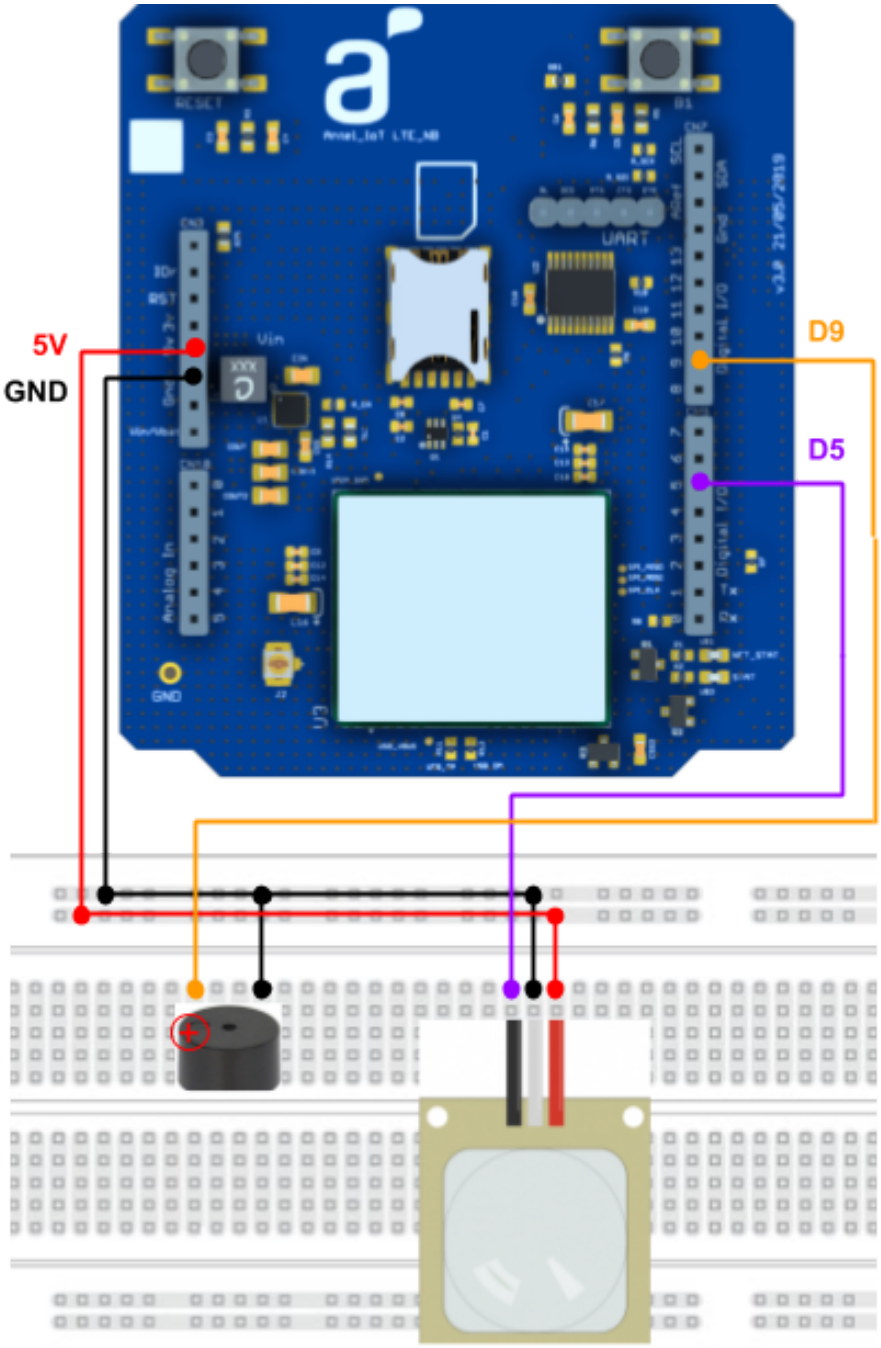
El buzzer se conecta un pin a GND y otro a un pin del header KitIoT que soporte PWM.
Luego simplemente definimos este pin donde está conectado de la siguiente manera:

```
Beep bell(D9);
...
bell.beep(500,0.1);
```

[29] https://www.sparkfun.com/products/13285
[30] https://os.mbed.com/users/dreschpe/code/beep/

Y con **bell.beep(500, 0.1)** se hace sonar el buzzer, siendo el primer parametro la frecuencia del tono en Hz y el segundo la duración en segundos.

En el siguiente diagrama se pueden ver como se conectan los componentes:



Problemas identificados

Socket con IP

En las pruebas que se hicieron no fue posible iniciar un socket de mbed indicando una IP en lugar de un dominio. El mensaje de respuesta al realizar la conexión que da es correcto, pero luego cualquier intento de usar el socket falla.

Flash del Kitlot

Al menos en el caso de MacOS, puede que al intentar hacer flash de la placa el comando de un error. 3 posibles soluciones son:

- Desconectar y conectar la placa
- Apretar y soltar el boton de reset al mismo tiempo que se da enter para ejecutar el comando de flash
- Desconectar la placa, dejar apretado el boton de reset y conectarla y luego sin soltar el boton de reset ejecutar el comando: st-info --probe luego de esto ya se puede flashear

Trabajo futuro

Se deja a continuación una serie de puntos identificados como trabajo futuro.

- Analizar si es posible quitar dependencias agregadas (que no se usan) para reducir el tiempo de compilación:
    - https://os.mbed.com/docs/mbed-os/v6.3/program-setup/build-rules.html
    - https://os.mbed.com/docs/mbed-os/v6.3/program-setup/adding-and-configuring-targets.html#extra-labels-extra-labels-add-and-extra-labels-remove
- Refactorizar partes del código quitando condicionales y otras cosas que no se utilizan
- Desarrollar una app propia (o tomar de base la del tutorial) para interactuar con los dispositivos, sin los dispositivos dummy que levanta el docker del tutorial
- Implementar otros tipos de sensores así como actuadores, luces inteligentes, etc