Análisis e implementación de clientes MQTT sobre servidores locales y en la nube.

Programación Distribuida y Tiempo Real.

Alumnos: Butera, Blas; Dal Bianco, Pedro Alejandro.

Facultad de Informática, 2020.

Índice

1. Introducción

1.1 Desarrollo del proyecto

Este proyecto busca explorar el protocolo MQTT y su utilización para propósitos de IoT tanto a través de un servidor local como de uno remoto, utilizando concretamente el provisto por la plataforma de IoT de IBM[1] y con clientes distribuidos que se simularán implementándose en el lenguaje C.

Su desarrollo consta de dos partes: una primera donde se realizan pruebas locales del protocolo, con un servidor local que se comunica con clientes ejecutándose en su misma computadora y una segunda parte donde se migra el servidor local a la mencionada plataforma cloud de IBM, desarrollando las diferencias y explorando las diferentes funcionalidades provistas.

1.2 MQTT

MQTT[2] (Transporte de telemetría de colas de mensajes) es un protocolo de red liviano y simple del tipo publicación-subscripción pensado para el envío de mensajes entre máquinas (M2M). Fue diseñado para ser utilizable por dispositivos con recursos limitados y que no dispongan de un gran ancho de banda. Este protocolo se monta típicamente sobre TCP/IP y por las características mencionadas es uno de los más utilizados en campos como *Internet of Things*.

MQTT define dos entidades de red: un servidor (más específicamente llamado ***broker***) y un número de **clientes** conectados a dicho *broker*.

Los mensajes se organizan por **tópico** y funcionan de la siguiente manera: cada vez que un cliente quiera publicar un mensaje debe especificar el tópico al cual será publicado, y el *broker* se encargará de distribuir el mensaje enviado entre todos los clientes que se hayan suscrito a dicho tópico. Para un tópico se pueden definir subtópicos utilizando '/' de la forma "*nivel/subnivel*", y es posible además, definir las llamadas *wildcards*, que son símbolos que funcionan como comodines. El símbolo '#' refiere a cualquier elemento que esté en ese nivel o subnivel, mientras que el '+' refiere a cualquier elemento en un único nivel.

Por último, es posible definir para una conexión MQTT la llamada **calidad del servicio**[3] (*quality of service*, o *QoS*). Este valor determina el nivel de confirmación (*acknowledge*) que espera recibir o debe enviar un cliente respecto de la publicación de un mensaje, y puede ser 0: no espera confirmación; 1: reenvía el mensaje hasta obtener confirmación; o 2: se

---

[1] https://cloud.ibm.com/catalog/services/internet-of-things-platform?cm_sp=ibmdev-_-developer-articles-_-cloudreg

[2] http://mqtt.org/

[3] https://en.wikipedia.org/wiki/MQTT#Quality_of_service_(QoS)

establece un *handshake* entre el transmisor y el receptor para garantizar que el mensaje se envía una sola vez.

2. Pruebas locales

El objetivo de esta primera etapa fue el probar distintos aspectos del protocolo en un ambiente local centralizado, como un primer acercamiento a este. Para esto se realizó una instalación local de un servidor MQTT, y se instaló también una librería con métodos que facilitan la implementación de un cliente en lenguaje C.

Utilizando las herramientas descritas, se programaron clientes MQTT que a través de este protocolo buscan simular una ejecución en un entorno distribuido.

2.1 Servidor: Mosquitto

Mosquitto[4] es un *broker* MQTT de código abierto. Según describe la página oficial, es un *broker* liviano y apto para usarse tanto en computadoras de bajos recursos como en grandes servidores.

Para la primera etapa de este proyecto, todas las pruebas se realizaron sobre dos servidores distintos:

- Una instalación local de *Mosquitto*.
- Un servidor de prueba provisto por sus desarrolladores, al que se puede acceder a través de https://mqtt.eclipse.org/.

La instalación local de *Mosquitto* para este trabajo se realizó sobre un sistema operativo Ubuntu 20.04, siguiendo las instrucciones de descarga de su página oficial[5]. Luego, a través del comando "`mosquitto`" es posible iniciar el servidor, que escuchará por defecto en el puerto 1883 para conexiones no encriptadas y en el 8883 para conexiones encriptadas.

Para este proyecto se utilizó la configuración por defecto de *Mosquitto*, que no contempla ningún método de autenticación o control sobre qué clientes se conectan. Esto es posible a través del archivo de configuración de *Mosquitto*, sin embargo queda por fuera del alcance de esta sección del trabajo.

A continuación se ilustra la puesta en marcha del *broker* con Mosquitto.

```
user@desktop:~$ sudo mosquitto
mosquitto version 1.6.9 starting
Config loaded from /mosquitto/config/mosquitto.conf.
Opening ipv4 listen socket on port 1883.
Opening ipv6 listen socket on port 1883.
```

[4] https://mosquitto.org/
[5] https://mosquitto.org/download/

2.2 Cliente: Eclipse Paho

Eclipse Paho[6] es una implementación de código abierto de los métodos necesarios para utilizar MQTT desde un cliente en distintos lenguajes.

2.2.1 Instalación

En este proyecto se realizaron implementaciones de programas cliente en el lenguaje C, para lo cual se utilizó el cliente de *Eclipse Paho* en dicho lenguaje. Para su instalación, en primer lugar se debió instalar el paquete de desarrollo *OpenSSL* (en Ubuntu se instaló a través del manejador de paquetes por defecto: "`sudo apt-get install libssl-dev`") y luego se siguieron los pasos descritos en su página oficial de descargas[7] para compilarlo e instalarlo a partir de su código fuente.

```
git clone https://github.com/eclipse/paho.mqtt.c.git
cd org.eclipse.paho.mqtt.c.git
make
sudo make install
```

2.2.2 MQTT a través de la consola

Eclipse Paho incluye también *scripts* para utilizar MQTT a través de la consola de comandos: "`paho_c_sub`" para suscripción y "`paho_c_pub`" para publicación.

Su uso es sencillo, basta con indicarles el tópico a través del parámetro `-t` y en el caso de "`paho_c_pub`" quedará a la espera del ingreso de mensajes, mientras que en el de "`paho_c_sub`" mostrará todos los mensajes que reciba que se correspondan con el tópico ingresado.

Por defecto intentan conectarse con el servidor local, por lo que en la figura 1.a se ilustra un ejemplo del uso de estos comandos para enviar un mensaje "*Hola mundo*" con el tópico "*ejemplo*" a través de una instancia local del servidor *Mosquitto*.

```
user@desktop:~$ paho_c_sub -t ejemplo
Hola mundo
```

```
user@desktop:~$ paho_c_pub -t ejemplo
Hola mundo
```

Fig 1.a. El mensaje enviado a través del comando *paho_c_pub* es recibido a través del comando *paho_c_sub*.

[6] https://www.eclipse.org/paho/
[7] https://www.eclipse.org/paho/clients/c/

En caso de querer utilizar un servidor distinto al local, se puede utilizar el parámetro `-h` para indicar la dirección del servidor, como por ejemplo en el caso de querer utilizar el servidor de prueba mencionado en la sección “Servidor”, lo que se muestra entonces en la figura 1.b:

```
user@desktop:~$ paho_c_sub -t ejemplo -h mqtt.eclipse.org
Hola mundo
```

```
user@desktop:~$ paho_c_pub -t ejemplo -h mqtt.eclipse.org
Hola mundo
```

Fig. 1.b. Se envía el mismo mensaje que en el ejemplo anterior, pero esta vez a través del servidor alojado en mqtt.eclipse.org

2.2.3 MQTT sobre C

Eclipse Paho nos brinda la librería “*MQTTClient.h*” que nos provee de métodos y estructuras de datos que implementan las distintas funcionalidades del protocolo MQTT.

Algunas de las principales funciones y estructuras que brinda son:

- `MQTTClient`: una estructura para representar una instancia de un cliente MQTT.
- ```
  int MQTTClient_create (
        MQTTClient* handle,
        const char* serverURL,
        const char* clientId,
        int persistence_type,
        void* persistence_context):
  ```

  Esta función nos permite obtener una instancia válida de un MQTTClient pasado como parámetro, especificando la dirección del *broker*, un identificador del cliente y opciones para la persistencia de este. Devuelve la constante entera `MQTTCLIENT_SUCCESS` en caso de éxito, y caso contrario devuelve el código de error correspondiente.

- ```
  int MQTTClient_connect (
        MQTTClient handle,
        MQTTClient_connectOptions* options):
  ```

  Esta función intenta conectar una instancia válida de un cliente con el *broker* correspondiente, definiendo un conjunto de opciones.

- ```
  int MQTTClient_setCallbacks (
        MQTTClient handle,
        void* context,
        MQTTClient_connectionLost* cl,
        MQTTClient_messageArrived* ma,
        MQTTClient_deliveryComplete* dc):
  ```
  ```

Esta función nos permite definir funciones *callbacks* que se ejecutarán de forma asincrónica. En caso de no utilizar una de estas funciones, se puede indicar con el valor *null*.

- ```
  int MQTTClient_disconnect (
        MQTTClient handle,
        int timeout):
  ```

  Esta función intenta desconectar el cliente del *broker* correspondiente, definiendo un tiempo máximo de espera por la confirmación de recepción de mensajes.

- `void MQTTClient_destroy (MQTTClient* handle)`: Esta función libera la memoria ocupada por un cliente que ya no va a ser utilizado.

En caso de necesitar que nuestro cliente **publique** mensajes, resultan de interés las funciones:

- ```
  int MQTTClient_publishMessage (
        MQTTClient handle,
        const char* topicName,
        MQTTClient_message* msg,
        MQTTClient_deliveryToken* dt):
  ```

  Esta función intenta publicar el mensaje *msg* a través del cliente *handle*, bajo el tópico *topicName*. Cuando la función se ejecuta correctamente esta devuelve un token de envío en *dt*.

- ```
  int MQTTClient_waitForCompletion (
        MQTTClient handle,
        MQTTClient_deliveryToken dt,
        unsigned long timeout):
  ```

  Esta función nos permite esperar de forma sincrónica la confirmación de la publicación del mensaje que se corresponde con el token *dt* por a lo sumo *timeout* segundos.

- ```
  typedef void MQTTClient_deliveryComplete(
        void* context,
        MQTTClient_deliveryToken dt):
  ```

  En caso de querer realizar la confirmación de la publicación de los mensajes de forma asincrónica, se debe implementar esta función. Dicha implementación debe pasarse como parámetro a la función *setCallbacks*.

En caso de que nuestro cliente deba **suscribirse** a un tópico en particular, nos serán de utilidad las siguientes funciones:

- ```
  int MQTTClient_subscribe (
        MQTTClient handle,
        const char* topic,
        int qos):
  ```

  Esta función suscribe al cliente *handle* al tópico *topic*. Permite definir además la llamada calidad del servicio (*quality of service* o *QoS*) a través del parámetro *qos*, que funciona tal como se la describió en la introducción de este trabajo.

- ```
  typedef int MQTTClient_messageArrived(
        void* context,
        char* topicName,
        int topicLen,
        MQTTClient_message* message):
  ```

  Esta es una función *callback* donde podemos definir el método a ejecutar ante la llegada de un mensaje *message*.

Una descripción más exhaustiva de estas funciones y el resto de las funciones y estructuras que brinda esta librería se pueden encontrar en su documentación oficial[8].

2.2.3.1 Simulación de sensores MQTT

Utilizando los métodos anteriormente descritos se realizó una implementación en C de programas que simulan sensores que consistió de los siguientes programas:

- *temperature.c* representa un sensor de temperatura, que se encarga de publicar al tópico "*temperature*" un valor representando la temperatura actual (para esta implementación se representó con un número al azar) cada un segundo.

- *switch.c* representa el control de un interruptor de luz a través de una variable booleana que indica su estado. Está suscrito al tópico "*switch/request*" donde recibe mensajes que contienen un comando. En caso de recibir un mensaje "*toggle*", cambiará el estado de la luz, y en caso de recibir un mensaje "*status*", publicará un mensaje al tópico "*switch/status*" indicando el estado de la luz (0 o 1).

- *control_panel.c* representa un panel de control para interactuar con los sensores anteriores. Nos permite consultar la temperatura actual, la que actualiza cada vez que recibe un mensaje en el tópico "*temperature*"; cambiar el estado de la luz y consultar por este estado.

La figura 2 ilustra cómo interactúan los programas a través de los distintos tópicos MQTT, donde una flecha del programa *x* al tópico *y* indica que *x* publica al tópico *y*, mientras que una flecha del tópico *y* al programa *x* indica que *x* está suscrito a *y*.

[8] https://www.eclipse.org/paho/files/mqttdoc/MQTTClient/html/index.html

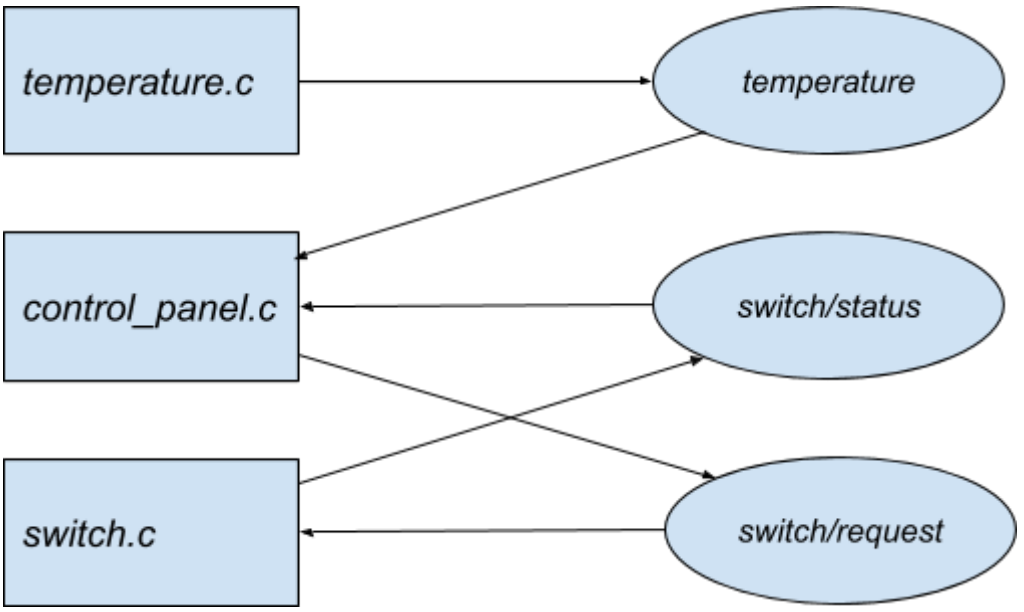

Fig. 2. Esquema de programas (izquierda) y tópicos (derecha) y las respectivas publicaciones/suscripciones.

Por último la figura 3.a muestra la consola durante un ejemplo de ejecución de *control_panel.c* donde se probaron las funcionalidades anteriormente descritas, y la figura 3.b muestra la consola correspondiente a *switch.c* durante la misma ejecución.

```
user@desktop:~$ ./cp.out
Elija una opción entre:
 1) Consultar la temperatura
 2) Prender/apagar la luz
 3) Consultar estado de la luz
1
La temperatura es 1.715e+09
3
La luz se encuentra apagada
2
Se modificó el estado de la luz
3
La luz se encuentra encendida
```

Fig 3.a. Ejecución de *control_panel.c*.

```
user@desktop:~$ ./switch.out
El sensor se encuentra en línea
Reportando estado 0
Se activó el interruptor, el estado es ahora 1
Reportando estado 1
```

Fig 3.b. Ejecución de *switch.c*.

Todos los programas listados permiten un primer parámetro opcional para indicar la dirección del servidor al que intentarán conectarse, siendo este por defecto el servidor local. En caso de querer conectarse a otro servidor, utilizar este parámetro con la forma *protocolo://dirección:puerto*. Por ejemplo, para conectarse al servidor de prueba provisto por eclipse, sobre el cual también se probaron estos programas, se deben ejecutar de la siguiente manera: "`./cp.out tcp://mqtt.eclipse.org:1883`".

2.2.3.2 Desarrollo de *switch.c*

Se procede a mostrar el uso concreto de algunos de los métodos y estructuras mencionados anteriormente para la implementación del programa *switch.c* ya descrito.

En primer lugar, se definen, al igual que en los otros programas, las constantes CLIENTID, que contiene un identificador para dicho cliente, y QOS, indicando la calidad del servicio. Respecto a estos valores, en este caso el identificador del cliente se utilizará al momento de conectarse, pero no será de importancia ya que no se estableció ningún método de autenticación desde el *broker*, y se definió una *QoS* de 0 ya que se consideró que la confirmación de envió del mensaje no resulta relevante para estos casos que los mensajes envían pequeña cantidad de información y que en caso de falla no se pierde ninguna información crucial que no pudiese ser solicitada de nuevo.

Además, la variable *address* indica la dirección del *broker* al que se intentará conectar (por defecto el servidor local), la variable *client* contendrá los datos del cliente y *switch_status* representa el estado del *switch* (prendido o apagado).

```
#define CLIENTID    "switch"
#define QOS         0

char* address = "tcp://localhost:1883";
MQTTClient client;

int switch_status = 0;
```

La función *report_status* se encarga de publicar un mensaje al tópico "*switch/status*" con el estado del *switch*. Para esto se define un mensaje MQTT *pubmsg* del tipo *MQTTClient_message*, en cuyo contenido (*payload*) se indica el correspondiente estado. Luego se cargan otros campos requeridos como la longitud del mensaje y la *QoS* y se envía el mensaje con el método *MQTTClient_publishMessage*, a través del cliente ya definido, al tópico "*switch/status*" y enviando un valor nulo para el token de verificación ya que se definió una *QoS* de 0, por lo que no habrá verificación.

```
void report_status() {

    MQTTClient_message pubmsg = MQTTClient_message_initializer;

    char* payload = switch_status ? "1" : "0";
    pubmsg.payload = payload;
    pubmsg.payloadlen = (int)strlen(payload);
    pubmsg.qos = QOS;
    MQTTClient_publishMessage(client, "switch/status", &pubmsg, NULL);
}
```

La función *msgarrvd* es una función *callback* que se ejecutará ante la llegada de un mensaje. Como se describió en la sección anterior, el contenido de los mensajes que le llegarán al *switch* a través de "*switch/request*" contienen un comando, por lo que esta función verifica cual es el comando recibido y, en caso de un "*toggle*", cambia el estado del *switch* (a través de la función *toggle*) y lo imprime en consola, y en caso de un "*status*" informa de su recepción por consola y llama a *report_status*. Por último, a través de las funciones *MQTTClient_freeMessage* y *MQTTClient_free* libera la memoria ocupada por el mensaje recibido.

```
int msgarrvd(void *context, char *topicName, int topicLen,
MQTTClient_message *message) {

    char* content = (char*)message->payload;

    if (strcmp(content, "toggle") == 0) {
        printf("Se activó el interruptor, el estado es ahora %d\n",
toggle());
    }
    else if (strcmp(content, "status") == 0) {
        printf("Reportando estado %d\n", switch_status);
        report_status();
    }
    else {
        printf("Se recibió un comando inválido\n");
    }

    MQTTClient_freeMessage(&message);
    MQTTClient_free(topicName);
    return 1;
}
```

Por último, en el método *main*, el programa inicializa en *conn_opts* las opciones por defecto para una conexión MQTT, chequea si recibió un parámetro de servidor al cual conectarse, y

de ser así actualiza el valor de *address*, inicializa el cliente con la función *MQTTClient_create* y configura sus respectivas *callbacks* con el método *MQTTClient_setCallbacks* (*msgarrvd* y *connlost* para informar si se perdió la conexión). Luego intenta conectarse utilizando el método *MQTTClient_connect* y en caso de éxito (si este devuelve el valor *MQTTCLIENT_SUCCESS*) lo informa a través de la consola, caso contrario finaliza su ejecución. Por último se suscribe al tópico "*switch/request*" y entra en un *loop* a la espera de mensajes.

```
int main(int argc, char* argv[]) {

    MQTTClient_connectOptions conn_opts =
MQTTClient_connectOptions_initializer;
    int rc;
    int ch;

    if (argc >= 2) {
        address = argv[1];
    }

    MQTTClient_create(&client, address, CLIENTID,
MQTTCLIENT_PERSISTENCE_NONE, NULL);
    MQTTClient_setCallbacks(client, NULL, connlost, msgarrvd, NULL);

    if ((rc = MQTTClient_connect(client, &conn_opts)) !=
MQTTCLIENT_SUCCESS) {
        printf("Fallo en la conexión con codigo de error %d\n", rc);
        exit(EXIT_FAILURE);
    } else {
        printf("El sensor se encuentra en linea\n");
    }


    MQTTClient_subscribe(client, "switch/request", QOS);

    while (1) {}

    MQTTClient_unsubscribe(client, "switch/request");
    MQTTClient_disconnect(client, 10000);
    MQTTClient_destroy(&client);
    return rc;
}
```

Se puede encontrar este archivo completo junto al resto de los archivos fuente descritos en la sección de archivos adjuntos, y para compilarlos se debe usar el comando "`gcc archivo_fuente -l paho-mqtt3c -o archivo_objeto`".

3. Pruebas sobre servidor remoto

El objetivo de esta sección fue migrar el servidor MQTT a un servidor remoto utilizando la plataforma IoT de IBM[9]. Para este proyecto se utilizó una cuenta gratuita y se busca ilustrar las principales diferencias encontradas con respecto al uso de un servidor local, replicando lo desarrollado en la sección anterior.

3.1 Configuración del servidor

Para poder levantar y configurar correctamente el servidor, en primer lugar fue necesario registrarse en IBM Cloud[10], en este caso con el plan gratuito. Luego desde la página de la plataforma de IoT de IBM[9] se creó el servicio requerido, seleccionando como región Dallas (la opción por defecto) y con el nombre de PDyTR - MQTT broker. Una vez hecho se puede iniciar el servicio accediendo al menú lateral > “Lista de recursos” > “Servicios” > “PDyTR - MQTT broker”, y luego a través del botón Lanzar. Esto iniciará el servidor y nos llevará a su correspondiente pantalla de gestión (Fig 4).

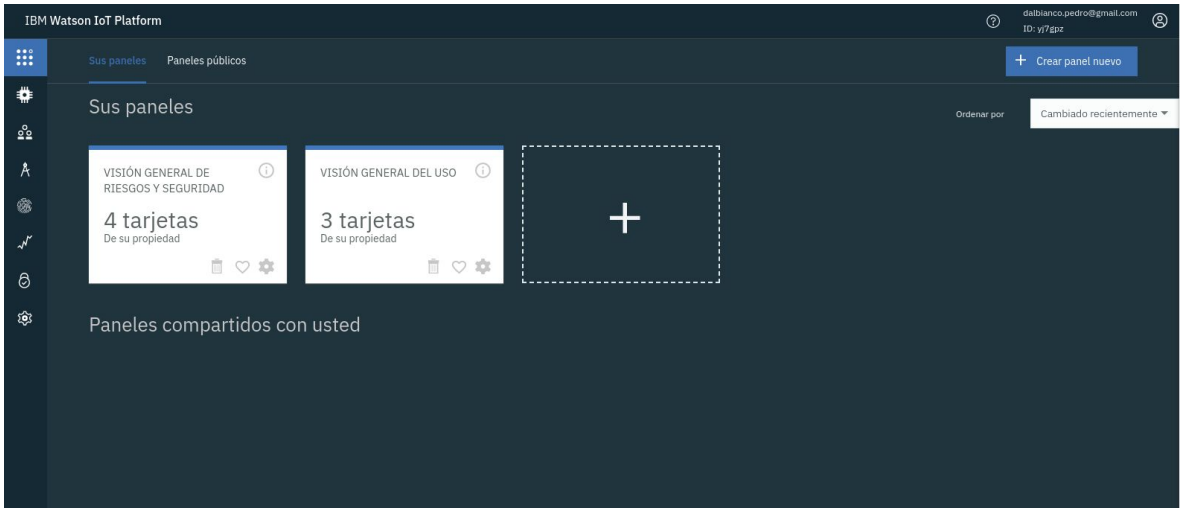

Fig 4. Pantalla principal de gestión del servidor.

Notar también que una vez creado disponemos de un ID único para el servidor que se puede observar en la esquina superior derecha (en este caso *yj7gpz*).

Para este proyecto la única configuración que fue necesaria previa a proceder a conectar los dispositivos y aplicaciones fue la de habilitar el puerto 1883 para permitir conexiones no encriptadas, ya que la encriptación de los mensajes no estaba prevista inicialmente en los

9 https://cloud.ibm.com/catalog/services/internet-of-things-platform?cm_sp=ibmdev-_-developer-articles-_-cloudreg

10 https://cloud.ibm.com/

clientes desarrollados. Para esto se debió navegar dentro de la pestaña de “Configuración” > “Seguridad” > “Seguridad de conexión”, y seleccionar la opción “TLS opcional”.

3.2 Elementos nuevos de la plataforma IoT

Una de las novedades incluidas en la plataforma utilizada es que esta nos brinda elementos que permiten clasificar por un lado los distintos clientes, y por otro los mensajes MQTT que estos envían.

Los clientes se clasifican en **dispositivos**, **aplicaciones** y ***gateways***. Los dispositivos serán los encargados de captar y enviar información, o reaccionar a los comandos enviados por las aplicaciones, que además de comandar a los dispositivos pueden también suscribirse a los distintos tópicos y serán en general quienes procesen la información enviada por los dispositivos. Las gateways permiten centralizar varios dispositivos en uno único, que reciba los comandos y junte la información enviada por estos, pero no serán utilizados para el ejemplo práctico desarrollado en este proyecto. Para recrear la arquitectura descrita en la sección anterior, tanto el cliente *switch* como el cliente *temperature* serán representados por dispositivos, mientras que al panel de control se lo representará como una aplicación.

Los mensajes MQTT se pueden clasificar ahora en **eventos**, **comandos**, y mensajes de **estado**. Los primeros son los mensajes que envían los dispositivos para informar de, efectivamente, un evento sucedido. Este es el caso, en la arquitectura presentada, de los mensajes que informaban la temperatura o un cambio en el estado del *switch*. Los comandos son los mensajes enviados por aplicaciones para generar una reacción por parte de un dispositivo específico, este es el caso, por ejemplo, de los mensajes enviados por el panel de control para indicar al *switch* que cambie su estado. Por último, los mensajes de estado sirven para informar cuestiones técnicas de un dispositivo, como por ejemplo, el estado de su conexión.

3.3 Conexión de un dispositivo

En primer lugar debemos mencionar que en caso de querer enfocarte en desarrollar únicamente el programa que se encargue de recibir y procesar los datos de los sensores, el servicio de IBM provee la posibilidad de simular sensores directamente desde su interfaz. Dado que el propósito de este trabajo consistía también en la implementación del código cliente MQTT que implementara la transmisión de los datos de los sensores, dicha opción no se tuvo en cuenta y se procedió directamente a transferir la arquitectura ya desarrollada a este servidor, reutilizando los clientes ya implementados con las modificaciones pertinentes.

Para la conexión de los clientes se siguió la guía provista por IBM[11], donde se describe paso a paso cómo llevarla a cabo, y el artículo “*Communicating with devices (MQTT)*”[12], que muestra más detalladamente el correcto formato que deben tener los mensajes MQTT emitidos por un cliente para conectarse a su servidor.

[11] https://cloud.ibm.com/docs/IoT/index.html
[12] https://www.ibm.com/support/knowledgecenter/SSQP8H/iot/platform/devices/mqtt.html

Se procede entonces a ilustrar cómo se llevó a cabo dicha conexión en el caso del cliente *switch*, y que modificaciones debieron ser realizadas respecto al código exhibido en la sección 2.2.3.2.

3.3.1 Conexión de *switch.c*

Conectar un cliente como **dispositivo** al servidor IBM consta de dos partes: una primera parte donde se debe registrar el dispositivo en el servidor, definiendo entre otras cosas un identificador y una clave que le resultarán necesarias para poder establecer la conexión, y una segunda parte en la que se debe configurar el cliente para que efectivamente use los datos definidos en el momento del registro para conectarse.

3.3.1.1 Registro del dispositivo

El **registro** de un dispositivo se realiza también a través de la aplicación web, accediendo desde el menú lateral de la página de gestión a la opción “Dispositivos” y clickeando en el botón “Agregar dispositivo”, lo que te redirige al formulario de registro de dispositivos.

En primer lugar la aplicación solicita un tipo de dispositivo y un identificador. El tipo de dispositivo será una cadena que sirva para agrupar un conjunto de dispositivos, y el identificador deberá ser único para cada uno. Para el caso del *switch* se definió el tipo “Switch” y el identificador “LIGHTSWITCH1”. Luego es posible añadir metadatos del dispositivo físico, los cuales al no disponer de dicho dispositivo al momento del registro, no fueron definidos. El paso siguiente consiste en definir una Señal de autenticación, que le servirá al dispositivo a modo de token para poder conectarse. Esta señal puede ser definida manualmente o generada de forma automática, y en este caso se decidió por definirla de forma manual como “PDyTR2020”. Por último, se muestra un resumen de la información ingresada y permite finalizar el registro (Fig. 5).

Fig 5. Pantalla de confirmación de datos de registro.

3.3.1.2 Configuración del cliente *switch.c*

Respecto a la configuración del cliente, se debieron modificar algunos datos de la conexión para que el servidor lo identifique como el dispositivo registrado y le permita conectarse. En primer lugar la dirección a la que deberá conectarse nuestro dispositivo es "*orgid*.messaging.internetofthings.ibmcloud.com", donde "*orgid*" es el identificador del servidor que se nos asignó (mencionado en la sección 3.1), y ya que lo hemos habilitado previamente, se conectará con el puerto 1883. También se debió modificar el id del cliente para que coincida con el formato "*d:orgId:deviceType:deviceId*", donde la letra "*d*" indica que es un dispositivo (*device*) y "*deviceType*" y "*deviceId*" son el tipo y el identificador previamente definidos, respectivamente. Estas dos definiciones se ilustran en la figura 6.a.

```
#define CLIENTID    "d:yj7gpz:Switch:LIGHTSWITCH1"
#define ADDRESS     "yj7gpz.messaging.internetofthings.ibmcloud.com:1883"
```

Fig 6.a. Identificador y dirección utilizados en el cliente *switch*.

Luego, el nombre de usuario del cliente a la hora de conectarse debe ser "use-token-auth", indicando que se conectará a través de una señal de autenticación, y dicha señal debe ser enviada en el campo correspondiente a la contraseña. En la Fig.6.b se muestra como se redefinieron estos dos campos sobre la configuración inicial brindada por la librería *Eclipse Paho*.

```
MQTTClient_connectOptions conn_opts = MQTTClient_connectOptions_initializer;
conn_opts.username = "use-token-auth";
conn_opts.password = "PDyTR2020";
```

Fig 6.b. Definición de nombre de usuario y contraseña.

También se debieron modificar los formatos de los tópicos a los que suscribe y publica, ahora con la forma "iot-2/evt/*event_id*/fmt/*format_string*", donde *evt* indica publicación de un evento (en caso de suscripción utilizar *cmd*, que indica suscripción a un comando), *event_id* refiere al tópico al que se quiere publicar (o suscribir) y *format_string* refiere al formato de los datos enviados, por defecto *json*.

Además se agregó un método que permite interceptar las interrupciones, *handle_sigint*, para indicar al proceso que debe finalizar pero que lo haga finalizando la conexión correctamente. Esto se realizó debido a que el no finalizar correctamente una conexión puede llevar a problemas al intentar conectarse de nuevo con el mismo identificador.

Estas dos modificaciones mencionadas se ilustran en la figura 6.c.

```
signal(SIGTSTP, handle_sigint);

MQTTClient_subscribe(client, "iot-2/cmd/switch_request/fmt/json", QOS);
report_status();
while (!finished) {}

printf("\nDesconectando\n");
MQTTClient_unsubscribe(client, "iot-2/cmd/switch_request/fmt/json");
MQTTClient_disconnect(client, 10000);
MQTTClient_destroy(&client);
return rc;
```

Fig 6.c. Configuración del método *handle_sigint* para interceptar interrupciones y tópicos modificados.

Por último, al definirse que los mensajes se enviarán en formato *json*, se debió modificar el método encargado de la publicación de mensajes para que los formatee de la forma correspondiente. El método modificado se muestra en la figura 6.d.

```
void report_status() {

    MQTTClient_message pubmsg = MQTTClient_message_initializer;

    char* payload = switch_status ? "{\"status\":true}" :
        "{\"status\":false}";
    pubmsg.payload = payload;
    pubmsg.payloadlen = (int)strlen(payload);
    pubmsg.qos = QOS;
    MQTTClient_publishMessage(client, "iot-2/evt/switch_status/fmt/json",
        &pubmsg, NULL);
}
```

Fig 6.d. Método encargado de la publicación de mensajes luego de las modificaciones mencionadas.

Se puede encontrar este archivo completo junto al resto de los clientes en la sección de archivos adjuntos.

3.3.2 Acceso a los datos del dispositivo

Una vez conectados correctamente los dispositivos clientes, la plataforma de IBM nos provee de distintas herramientas para acceder a los datos publicados por estos. En primer lugar, inmediatamente después de conectar un dispositivo nuevo, podremos visualizar los datos publicados por este mediante la misma interfaz web a través de su pantalla de dispositivos, junto con otra información acerca de este, como se muestra en la figura 7.

Además, la plataforma también brinda una API completa para realizar consultas acerca de los distintos dispositivos y la información que publican. Luego en este informe se desarrolla cómo utilizar esta API a través de una aplicación para acceder a los datos de un dispositivo, más concretamente a los publicados por el cliente *switch*.

Fig 7. Desde la vista de un dispositivo (DIspositivos > ID del dispositivo) se puede ver los últimos mensajes publicados por este, y al estar en formato *json* es posible visualizar cada propiedad enviada (sección Estado).

3.4 Conexión de una aplicación

Los pasos a seguir para conectar un cliente que funcione como aplicación en la plataforma son distintos a los descritos para conectar dispositivos, ya que son otras las opciones que se deben tener en cuenta para configurarla. En este caso ya no se debe registrar un dispositivo nuevo, si no que debemos crear una clave de aplicación con su respectivo rol, el cual sirve para definir sus permisos, y los datos definidos para conectarse responderán a otro formato. A continuación, se procede a ilustrar cómo se realiza esta conexión y estas particularidades mencionadas a partir del cliente *control_panel* descrito en la sección 2.2.3.1.

3.4.1 Conexión de *control_panel.c*

Como se mencionó anteriormente, el cliente *control_panel* se implementará en forma de **aplicación**, que publicará mensajes al tópico *switch_request* en forma de **comandos** y estará suscrito a los **eventos** *temperature* del sensor de temperatura y *switch_status* del *switch*. Implementará la misma funcionalidad que el cliente desarrollado para conectarse al *broker* local.

3.4.1.1 Generación de una clave de aplicación

Entonces, en primer lugar debemos generar una clave de aplicación. Esta clave consta de un nombre de usuario y un *token* de acceso que nos permitirán autenticarnos a la hora de conectarnos de esta forma al *broker*. Para esto debemos ir a la sección “Aplicaciones”, del menú lateral, y en la esquina superior derecha encontraremos la opción “Generar clave de

API". Ingresando a esta opción nos abrirá un formulario para generar dicha clave, permitiendo agregar una descripción y una fecha de caducidad. Luego, en la pantalla siguiente nos permitirá definir un "rol" para la aplicación que utilice esa clave, el cual limitará los tipos de consultas que es capaz de realizar. Para este caso, se eligió el rol "Aplicación estándar", que permite tanto la publicación de comandos como la suscripción a eventos de dispositivos, lo necesario para la implementación a realizar. Más información acerca de los roles disponibles se puede encontrar en su respectiva documentación[13]. Una vez completado este formulario la aplicación muestra la clave generada con su respectivo *token* de autenticación y un resumen de sus características como se muestra en la figura 8. La clave generada para este experimento (que no se corresponde con la figura 8 que es una captura con fines ilustrativos) es "a-yj7gpz-kjetrawpdj", y el *token* de autenticación "x0s?gW0-k4g4_eryaj".

Fig 8. Resultado de generación de clave de API.

3.4.1.2 Configuración del cliente *control_panel.c*

Respecto a la configuración del cliente, al igual que en el caso del *switch*, se debieron modificar los datos de identificación del cliente de forma similar, aunque con ciertas modificaciones que corresponden a la identificación de una aplicación. La dirección a la que debe conectarse es la misma que en el caso de los clientes ("yj7gpz.messaging.internetofthings.ibmcloud.com:1883"), pero el identificador del cliente debe tener el formato "*a:orgId:applicationKey*", donde la letra "*a*" indica que se trata de una aplicación, "*orgId*" se corresponde con el identificador de la organización y "*applicationKey*" es la clave de aplicación generada (solo la última parte, que es exclusiva de cada aplicación). Esta definición se muestra en la figura 9.a.

13 https://www.ibm.com/support/knowledgecenter/SSQP8H/iot/platform/reference/rbac/index.html#user-application-and-gateway-roles

```
#define CLIENTID    "a:yj7gpz:kjetrawpdj"
#define ADDRESS     "yj7gpz.messaging.internetofthings.ibmcloud.com:1883"
```

Fig 9.a. Identificador y dirección utilizados en el cliente *control_panel*.

Para autenticarse correctamente como aplicación, al conectarse se debe definir el nombre de usuario y la contraseña como la clave de aplicación y el *token* de autenticación generados respectivamente, como se muestra en la figura 9.b.

```
MQTTClient_connectOptions conn_opts = MQTTClient_connectOptions_initializer;
conn_opts.username = "a-yj7gpz-kjetrawpdj";
conn_opts.password = "x0s?gW0-k4g4_eryaj";
```

Fig 9.b. Definición de nombre de usuario y contraseña.

Otro punto que resulta importante mencionar, es que tanto para la publicación de comandos como para la suscripción a eventos se debe explicitar en el tópico el identificador del dispositivo y del tipo de dispositivo que se quiera que reciba los comandos o que publique los eventos, y en caso de querer generalizar es posible utilizar *wildcards*. Entonces, una suscripción a un evento *event_id*, emitido por el dispositivo *device_id* del tipo *device_type* tendrá la forma "iot-2/type/*device_type*/id/*device_id*/evt/*event_id*/fmt/*format_string*", y la publicación de un comando *command_id* para que lo reciba el mismo dispositivo, tendrá el siguiente formato: iot-2/type/*device_type*/id/*device_id*/cmd/*command_id*/fmt/*format_string*. La figura 9.c. muestra las líneas en las que el cliente *control_panel* se suscribe a los tópicos "*temperature*" y "*switch_status*".

```
MQTTClient_subscribe(client,
"iot-2/type/Sensor/id/TEMPERATURE1/evt/temperature/fmt/json", QOS);
MQTTClient_subscribe(client,
"iot-2/type/Switch/id/LIGHTSWITCH1/evt/switch_status/fmt/json", QOS);
```

Fig 9.c. Suscripción a los tópicos "*temperature*" y "*switch_status*".

Por último, siendo que los mensajes son enviados en formato *json*, su contenido debe ser parseado al momento de leerlo. Dado que en esta implementación se envían pocos datos simples, el parseo resultó sencillo y se realizó de forma manual, pero en caso de tener mayor cantidad de datos o en caso de que estos fueran más complejos podría resultar más cómoda la utilización de una librería para manejar este formato.

Una vez realizadas estas modificaciones, se logró que esta implementación funcione de la misma forma que la descrita en la sección 2.2.3.1, pero esta vez sobre el *broker* provisto por IBM. Se puede encontrar este archivo completo junto al resto de los clientes en la sección de archivos adjuntos.

3.5 Acceso a dispositivos través de la API (conexión con *switch.c*)

Como se describió anteriormente, la plataforma de IBM brinda una API que nos permite acceder a los datos de los distintos dispositivos, y a su vez publicar comandos. Para poder utilizar dicha API necesario en primer lugar **generar una clave de aplicación** (de la misma manera que se hizo para el cliente *control_panel* en la sección 3.4.1.1) y luego **crear una (o más) interfaz física y una (o más) interfaz lógica** que se corresponda con la física para definir que datos y cómo serán publicados. Los datos obtenidos al llevar a cabo este proceso servirán para poder realizar una **consulta a la API** por estos dispositivos. Se procede ahora a describir cómo se realizan y cuál es la utilidad de cada uno de los pasos mencionados, exceptuando la generación de una clave de aplicación ya descrita anteriormente, ilustrando el caso concreto de acceder a los datos del cliente *switch* a través de una conexión de estas características.

3.5.1 Creación de interfaces de dispositivo

Para poder acceder a los datos de un dispositivo a través de la API, en primer lugar habrá que definir una interfaz física y lógica para el tipo correspondiente a ese dispositivo. La interfaz física sirve para modelar la interfaz entre el dispositivo y el servidor, mientras que la interfaz lógica permite definir la vista del dispositivo para quienes traten de accederlo a través de la API.

Para crear estas interfaces, se debe acceder desde el menú lateral a “Dispositivos” y luego a la pestaña “Tipos de dispositivo”. Una vez allí, seleccionar el tipo de dispositivo deseado y la pestaña “Interfaz”. La plataforma brinda un asistente simple de creación de interfaces y uno avanzado. El asistente simple permite seleccionar propiedades de los dispositivos y crear automáticamente las interfaces físicas y lógicas y sus respectivos mapeos para mostrar estos a través de la API, mientras que el asistente avanzado permite configuraciones más profundas, y la creación de varias interfaces lógicas que se correspondan de distinta manera con las físicas. En el caso del cliente *switch* nos basta con el asistente simple para publicar su estado, por lo que se seleccionará esta opción y luego “Crear interfaz”. Luego, tendremos la posibilidad de agregar las propiedades que queramos que sean accesibles a través de la API, para las que podemos elegir en propiedades que se encuentren en la caché de la plataforma porque el dispositivo ya las haya enviado, que es el caso de *status*, o definir propiedades nuevas con el formato *json*. Una vez creadas las interfaces de esta manera, se nos mostrará un botón “Activar” que permite poner en funcionamiento las interfaces definidas.

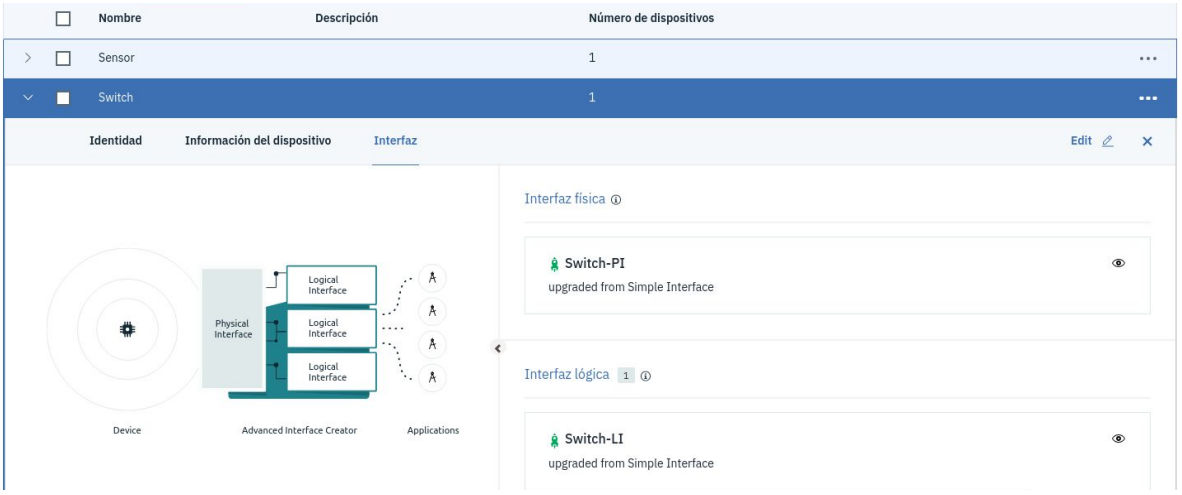

Fig 8.b. Resultado de creación de interfaces para el cliente *switch*.

3.5.2 Acceso a la API desde la interfaz web

Para cada servidor creado en la plataforma, esta brinda una API a la que se puede acceder con una gran cantidad de métodos distintos. Además, es posible probar estos métodos a través de una interfaz web particular del servidor creado[14]. Estos se dividen en categorías, y en este momento nos interesa más específicamente la sección de estado[15], que nos permite hacer consultas acerca de las interfaces físicas, lógicas, tipos de dispositivos y dispositivos particulares.

Desde esta interfaz web misma podemos consultar por las interfaces lógicas activas, entre las que deberíamos encontrar la definida en la sección anterior. Para esto debemos desplegar la sección “Logical interfaces” y el método “GET /logicalinterfaces”, que devuelve todas las interfaces lógicas definidas. La opción “Try it out” nos permite ejecutar una llamada a este método y observar la respuesta, dentro de la cual encontraremos la interfaz buscada, como se muestra en la figura 8.c.

[14] https://yj7gpz.internetofthings.ibmcloud.com/docs/index.html

[15] https://yj7gpz.internetofthings.ibmcloud.com/docs/v0002/state-mgmt.html#/Logical%20Interfaces/get_logicalinterfaces

```
{
      "version": "active",
      "created": "2020-07-11T17:54:49Z",
      "createdBy": "dalbianco.pedro@gmail.com",
      "updated": "2020-07-11T17:57:54Z",
      "updatedBy": "dalbianco.pedro@gmail.com",
      "name": "Switch-LI",
      "description": "upgraded from Simple Interface",
      "id": "5f0a356dac00880008af6dfb",
      "schemaId": "5f09fce9ac00880008af6d5a",
      "refs": {
         "schema": "/api/v0002/schemas/5f09fce9ac00880008af6d5a",
         "rules":
"/api/v0002/logicalinterfaces/5f09fce9ac00880008af6d5b/rules"
      },
      "alias": "switch"
}
```


Fig 8.c. Recorte de la respuesta del *endpoint* "/logicalinterfaces" con los datos de la interfaz de *switch*.

Además de utilizar esta página para probar los distintos métodos que nos ofrece la API, resulta útil para obtener información importante, como en este caso el id de la interfaz, el que utilizaremos luego para los *scripts* que quieran consultar por estos datos.

3.5.3 Acceso a la API a través de un *script*

El último paso consiste entonces en poder acceder a esta API a través de un *script* externo a la plataforma utilizando HTTP. La principal diferencia entre conectarse de esta forma respecto a usar la interfaz mencionada en el inciso anterior, es que al querer conectarse desde un *script* externo deberemos autenticarnos correctamente, y para esto se utilizará la clave y el *token* de autenticación generados en la sección 3.3.1.1. La forma de utilizarlas se describe en la documentación oficial del servicio[16].

Para probar esto, se desarrolló un script en *javascript* que realiza una consulta al *endpoint* "/device/types/{typeId}/devices/{deviceId}/state/{logicalInterfaceId}" definido en la documentación, donde *typeId* es el identificador del tipo de dispositivo, *deviceId* es el identificador de dispositivo (ambos definidos al momento de su creación) y *logicalInterfaceId* es el identificador de la interfaz lógica correspondiente, que se obtuvo a través de la consulta ilustrada en la sección anterior.

Para llevar a cabo esta parte del desarrollo se utilizó *axios*[17], una librería *javascript* que facilita la implementación de consultas HTTP, y esto y el resultado de su ejecución se muestran en las figuras 8.d y 8.e.

[16] https://www.ibm.com/support/knowledgecenter/en/SSQP8H/iot/platform/reference/api.html
[17] https://github.com/axios/axios

```
const getSwitchStatus = () =>
axios.get("device/types/Switch/devices/LIGHTSWITCH1/state/5f0a356dac0088
0008af6dfb", {
    headers: {
        accept: "application/json"
    },
    auth: {
        username: "a-yj7gpz-huxbxokmhr",
        password: "e0y()oCB1k)Mes8UzY"
    }
}).then(
    res => console.log(res.data)
).catch(logError)
```

Fig 8.d. Función javascript encargada de realizar la consulta a la API.

```
user@desktop:~$ node api_call.js
{
  timestamp: '2020-07-12T00:51:51Z',
  updated: '2020-07-11T22:00:41Z',
  state: { status: 'false' }
}
```

Fig 8.e. Resultado de la ejecución del *script*.

Además, se desarrolló otro *script* que, de la misma forma que el mostrado anteriormente, consulta por la temperatura publicada por el cliente *temperature*, y un último *script* que cambia el estado del *switch* publicando un comando a través del *endpoint* "/application/types/{*typeId*}/devices/{*deviceId*}/commands/{*commandId*}", donde *typeId* y *deviceId* representan los mismos datos que en el *endpoint* ya comentado, y *commandId* es el identificador del comando a enviar.

Estos *scripts* se pueden encontrar en la sección de archivos adjuntos, en el archivo *api_calls.js*.

4. Archivos adjuntos

En esta sección se listan los archivos adjuntos mencionados a lo largo del trabajo. También es posible encontrarlos todos en una carpeta de *Google Drive* accediendo al siguiente enlace: https://drive.google.com/drive/folders/1i4vxqy1bckduzGcp4mqLtvZI6ZoSU5PW

Pruebas locales:

1. *control_panel.c*: https://drive.google.com/file/d/1f--hxAtdZC7RK_aMM3TL7D7XjuQfYLXw/
2. *switch.c*: https://drive.google.com/file/d/18BuWSTSIprgmKiuiD0ODs5biLch-GqVU/
3. *temperature.c*: https://drive.google.com/file/d/1C43nCvbrP89i2I1F1ErU3-faMAwx-YID/

Pruebas en servidor remoto IBM:

4. *control_panel.c*: https://drive.google.com/file/d/1OksJrr9GjqGhvKJHaRkUwqgV5XpbIZ6L/
5. *switch.c*: https://drive.google.com/file/d/1U2xCpJOKxSQGjgrAb3XWncAx1sW4CLDu/
6. *temperature.c*: https://drive.google.com/file/d/1f2RAVXu7NGWyoGzQyeCwhc0vH4U9YJkm/

Consultas a API:

7. *api_calls.js*: https://drive.google.com/file/d/1vR-xsQnyoI-pfGo6B06Ma0VPiXD8Lz1b/