Capitulo 5.

Comunicaciones en Android Things

Por SALVADOR SANTONJA

El término “Internet de las Cosas” es sin duda una de las palabras de moda dentro y fuera de los círculos tecnológicos. Como suele pasar en estos casos, se ha extendido tanto y en tan poco tiempo que ha dado pie a multitud de interpretaciones y definiciones sobre cuáles son exactamente sus capacidades, o sobre donde empieza y donde termina exactamente un proyecto de Internet de las Cosas. Sin embargo, su propio nombre es tan auto-descriptivo que no deja lugar a dudas de cuáles son sus pilares fundamentales: **las Cosas**, esa interacción con el mundo físico, una digitalización mediante soluciones electrónicas y de sistemas embebidos que permite transformarlas al mundo digital; e **Internet**, esa conectividad que nos permite integrar el mundo físico en una red de comunicaciones, y convertirlo en servicios para poder interactuar con él desde cualquier lugar.

En la unidad anterior hemos aprendido cómo funciona el pilar de “**las Cosas**”, cómo podemos interactuar mediante entradas y salidas genéricas, así como mediante buses de comunicación, con elementos del mundo real. En esta segunda unidad nos enfrentaremos al pilar de “**Internet**” en su sentido más amplio: revisaremos las opciones de conectividad que nos aporta Android Things en general y la Raspberry Pi 3 en particular, y aprenderemos a trabajar con las opciones de comunicaciones más relevantes que nos propone el universo del Internet de las Cosas.

Objetivos:

- Conocer y comprender las características de las comunicaciones en el Internet de las Cosas.
- Enumerar las principales alternativas para comunicación offline.
- Descubrir y configurar nodos IoT usando Nearby Connections.
- Comparar los principales modelos de comunicaciones online: request/response y publish/subscribe.
- Desarrollar un servidor web embebido en la Raspberry Pi.
- Utilizar servicios web REST en Android Things.
- Aprender a utilizar MQTT para aplicaciones de IoT.

Índice

1.1. Opciones de comunicación en Android Things

La plataforma Android Things integra de forma nativa las comunicaciones a través de redes IP, sobre las que establecemos sockets para intercambiar datos. Esto nos permite utilizar de forma transparente las interfaces más típicas como Ethernet o WiFi. Por su origen en el mundo de la tecnología móvil, también integra conectividad Bluetooth (tecnología punto a punto para conectar un equipo principal o maestro con elementos periféricos). Bluetooth es un estándar que ha evolucionado mucho desde su creación, agregando cada vez más opciones y servicios, permitiendo en sus últimas versiones crear incluso redes de dispositivos para intercambio de datos. Además de esto, Android Things también integra librerías de comunicación LoWPAN, diseñadas para interfaces de comunicación de bajo consumo y baja tasa de transferencia de datos, muy utilizadas en redes inalámbricas de sensores.

Es posible clasificar las tecnologías de comunicación de muchas formas, no solo como cableadas o inalámbricas. Por ejemplo, podemos hablar de comunicaciones offline u online, según si éstas ofrecen una conexión a través de Internet o no. Un ejemplo de comunicación offline sería la conexión de un altavoz inalámbrico con el móvil, el intercambio de una foto entre dos móviles por Bluetooth, o incluso el intercambio de ficheros entre dos PCs por WiFi a través de un punto de acceso doméstico. En una comunicación online, nuestro equipo estaría conectado a Internet para ofrecer o consumir servicios de él: escritorio remoto con un equipo en otra ubicación, navegar por páginas web, consultar el correo electrónico, utilizar aplicaciones web o un servicio de almacenamiento cloud.

También podemos clasificar las tecnologías de comunicación como públicas (cuando la infraestructura la provee un operador a aquellos usuarios dispuestos a pagar su cuota, como las redes GSM o 3G) o privadas (cuando la infraestructura la gestiona una organización o particular y no está abierta a todo el público, como una red de área local). Otra opción es clasificarlas por su rango de alcance, como por ejemplo las tecnologías de área extensa (WAN - wide área networks), de área local (LAN – local área networks) o incluso de área personal (PAN – personal área networks).

Al final, la tecnología de comunicaciones es la responsable de proveernos de un canal físico para el intercambio de datos entre nuestro sistema y un sistema remoto, o dicho de otra forma, nos ofrece “conectividad”. En el mundo del Internet de las Cosas, hay tantas aplicaciones posibles que las tecnologías que aparecen en todas estas clasificaciones pueden ser susceptibles de ser utilizadas. Según las necesidades de conectividad, un paso vital en el diseño de una solución IoT será la selección del hardware adecuado para soportar estas comunicaciones.

En nuestro caso, la Raspberry Pi 3 dispone de las siguientes tecnologías de comunicación integradas en placa:

- 10/100 Ethernet
- WLAN a 2.4GHz 802.11b,g,n
- Bluetooth 4.1
- Bluetooth Low Energy

Además, es posible agregar otras interfaces de comunicación mediante USB, o mediante los buses serie de la placa, como por ejemplo GSM/3G, o un módulo 802.15.4 para comunicaciones LoWPAN.

1.2. Comunicaciones offline

En la era de Internet podría parecer extraño dedicar esfuerzos a comunicaciones offline, pero lejos de lo que podría parecer, este tipo de tecnologías tienen multitud de ventajas que las convierten en soluciones clave para muchas aplicaciones. Las principales características de una conexión offline son:

- **Baja latencia**: al comunicar directamente dos dispositivos sin someterlos a la carga temporal que sufren los mensajes a través de Internet, los tiempos de envío y recepción son los más bajos que podemos encontrar. Esto hace que las comunicaciones offline sean el sistema ideal para controles en tiempo real, como el control remoto de un robot, así como juegos multi-usuario.
- **Altas tasas de transferencia**: al evitar intermediarios, dos equipos conectados directamente mediante una interfaz de comunicación común, tendrán la posibilidad de exprimir esa interfaz al máximo. Una conexión Ethernet de 1Gbps tiene un potencial enorme en una red local, pero pierde su sentido al utilizar una conexión a Internet de 50 Mbps. Las aplicaciones de transferencia de datos se ven muy beneficiadas con este tipo de conexiones.
- **Conexiones estables**: una conexión a Internet está sujeta a caídas de servicio, tanto por el propio acceso (cobertura de redes móviles, proveedores con cortes de conexión, zonas remotas con conexiones intermitentes), como por los nodos intermedios de los que depende una conexión extremo a extremo. En una red offline, es más fácil conseguir los requisitos de estabilidad de tu aplicación, y mantenerlos en unos límites acotados.
- **Coste**: en la mayoría de casos, un acceso a Internet incurre en costes con el proveedor de servicios: máquinas de bebidas que controlan su stock, contenedores de basura inteligentes que avisan cuando están llenos, vehículos conectados, etc. Sin embargo, las conexiones offline pueden reducir la cantidad de puntos a conectar a Internet. Por ejemplo, los sensores en una granja pueden conectarse entre ellos con WiFi o LoWPAN, y salir por un único punto hacia Internet a través de un concentrador.
- **Seguridad y privacidad**: es importante tener en cuenta los riesgos que conlleva una conexión a Internet. Por un lado, nuestros datos circulan libremente, y según su criticidad, puede ser necesario utilizar mecanismos para asegurar la privacidad de la información, y hacer seguras las comunicaciones. Por otro lado, un sistema IoT conectado a Internet está sujeto a posibles ataques de seguridad, que puedan provocar desde denegaciones de servicio hasta tomar su control completo.
- **Consumo energético**: el consumo en comunicaciones de nuestro sistema depende profundamente de la tecnología y protocolos de comunicaciones uti-

lizados. Las comunicaciones móviles, como 3G/4G, utilizan sistemas de señalización con las antenas base y elevadas potencias de transmisión, que agotan la batería de los móviles en poco tiempo. WLAN es una tecnología diseñada para imitar una LAN sin cables, por lo que los dispositivos finales deben mantener una conectividad y una señalítica constante con el punto de acceso. Sin embargo, los equipos IoT son más propensos a presentar requisitos de bajo consumo, puesto que son equipos sobre los que no se espera realizar un mantenimiento diario, así que incluir un módem 4G en un sistema con baterías resulta prohibitivo. El uso de tecnologías de bajo consumo, como Zigbee o Lora, permite comunicar muchos dipositivos con poco consumo, y alimentar un único dispositivo central, que ofrezca en enlace a internet.

Podemos desarrollar aplicaciones con comunicación offline a través de cualquier interfaz, incluso WiFi y Ethernet. Pero también hay tecnologías que han sido desarrolladas de forma nativa para comunicaciones offline, como Bluetooth o Zigbee. Las 3 principales opciones de comunicación offline que nos ofrece el SDK de Android Things son Bluetooth, LoWPAN y Nearby Communications.

1.2.1. Bluetooth

Bluetooth es un estándar de comunicación inalámbrica para el intercambio de voz y datos entre dispositivos. Se encuentra especificado en la norma IEEE802.15.1, y está basado en una comunicación maestro – esclavo. Puede formar picoredes de hasta 7 esclavos, donde todos mantienen un enlace punto a punto con el maestro. El objetivo inicial de Bluetooth es el de eliminar cableado y conectores para facilitar la comunicación entre equipos fijos y móviles (ratones, auriculares, altavoces inalámbricos) y para la sincronización de equipos (intercambio de datos entre el móvil y el PC).

La norma Bluetooth cubre la pila de comunicaciones completa y establece requisitos de hardware, software e interoperabilidad entre equipos. Sus especificaciones están divididas en dos grandes bloques:

- Core specifications: define los componentes básicos de la tecnología que se utilizan para el funcionamiento de Bluetooth y su interoperabilidad con otros dispositivos. Esto incluye la descripción de las capas del protocolo, desde la interfaz radio hasta el enlace de datos, el descubrimiento de servicios, canales básicos de comunicación, controladores de host y emuladores de buses serie, y cumplimiento de interfaces y modos de test. En general, las core specifications definen cómo funciona la tecnología.
- Profile specifications: están orientadas a cómo se usa la tecnología Bluetooth para dar soporte a las distintas aplicaciones que se puede encontrar. Cada perfil indica cómo utilizar la tecnología del core specification para implementar un modelo de uso concreto. Existen más de 100 perfiles que incluyen teléfono inalámbrico, auriculares inalámbricos, intercambio de ficheros, sincronización, pulsera deportiva, etc.

La última versión de la norma es la 5.1, y es una agrupación de diversos modos de funcionamiento Bluetooth como el clásico (hasta 3 Mbps), el de alta velocidad (que utiliza wifi para el intercambio de datos, hasta 32 Mbps) y el Low Energy (que reduce alcance y tasa de datos a cambio de un consumo muy bajo de energía, hasta 2 Mbps). El alcance depende de la clase de dispositivo: clase 1 (1 m), clase 2 (10 m) y clase 3 (100 m).

1.2.2. LoWPAN

Low Rate Wireless Personal Area Networks (LR-WPAN o LoWPAN) es una tecnología de red inalámbrica de muy bajo consumo, baja tasa de transferencia y soporte a enlaces intermitentes. Está diseñado para dispositivos agrupados en áreas de pequeño tamaño, como redes de sensores inalámbricas, juguetes inteligentes, controles remotos o domótica. Su especificación base es la norma IEEE802.15.4, que describe la interfaz radio y la capa de enlace de datos. En ella se define una tasa de transferencia de 250 kbps, 40 kbps o 20 kbps; y la utilización de 16 canales en la banda de 2.4 GHz, 10 canales en la de 915 MHz o 1 canal en la de 868 MHz.

Sobre estas especificaciones se construyen las diversas tecnologías que podemos encontrar en la actualidad como Zigbee, MiWi, ISA100.11a, WirelessHART, SNAP, Thread o 6LoWPAN. La tendencia actual es la de incluir soporte IP en este tipo de tecnologías, para que los dispositivos sean direccionables desde Internet y su integración en el Internet de las Cosas sea transparente. Sin embargo, estos equipos presentan una fuerte restricción de hardware, energía, y tamaño de transmisiones, por lo que se utiliza una versión modificada de IPv6 conocida como 6LoWPAN. La pasarela de la red LoWPAN, encargada de realizar en cambio de un dominio inalámbrico LoWPAN a una conexión a Internet, es también la encargada de adaptar 6LoWPAN a IPv6.

Preguntas de repaso: Comunicaciones Offline

1.2.3. Nearby Connections

Nearby Connections es la propuesta de Google para abstraernos de las complejidades inherentes al uso de Bluetooth y Wifi a la hora de realizar conexiones directas entre equipos. Es un API de proximidad, con comunicaciones punto a punto completamente offline, que nos permite anunciar nuestro equipo a su entorno, descubrir equipos cercanos, establecer conexiones e intercambiar datos. Las conexiones establecidas están completamente seguras (siempre están encriptadas, y opcionalmente podemos autenticar los equipos), y presentan un elevado ancho de banda y una baja latencia. Esto hace que su uso sea idóneo para aplicaciones como juegos multipantalla (un teléfono o Tablet como control, y una AndroidTV como visualización), partidas multijugador (un jugador establece la partida e invita al resto a unirse), pizarras colaborativas, mensajería de proximidad o transferencias de ficheros sin conexión a Internet.

Por debajo, Nearby Connections utiliza una combinación de Bluetooth, Bluetooth Low Energy y WiFi, según las fases de la conexión y el tipo de transferencia a realizar. Las interfaces Bluetooth y Wifi se gestionan automáticamente por el API, de forma que no es necesario solicitar al usuario que las active; y al cerrar la app vuelven a su estado anterior. Esto permite que la experiencia del usuario sea mucho menos invasiva.

El funcionamiento de Nearby Connections se resume de la siguiente forma:

- **Fase de pre-conexión**. Existen dos roles: los **anunciantes**, y los **descubridores**. Los anunciantes lanzan balizas para que los descubridores cercanos sepan que están ahí, y que tienen algo que ofrecer. Muestran un nombre amigable que permite que los descubridores diferencien distintos anunciantes, y puedan decidir a quién quieren conectarse.
- **Fase de establecimiento de conexión**. Cuando un descubridor desea conectarse a un anunciante, lanza una solicitud que inicia un proceso de autenticación simétrica. Ambos lados pueden decidir si aceptan o no esta conexión. Si se acepta en ambos, se establece una conexión encriptada.
- **Fase de comunicación**. Con la conexión establecida, tenemos un enlace punto a punto entre dos dispositivos, y los roles de anunciante y descubridor desaparecen. A partir de ahora, los roles serán de emisor y receptor, pudiendo utilizar cada nodo uno de estos roles o ambos, lo que definirá si el canal es de un sentido o full-dúplex. Nearby Communications dispone de 3 tipos de intercambio de datos:
    - **BYTES**: transmisión de un byte array de hasta 32k, ideal para intercambiar mensajes y metadatos.
    - **FILE**: intercambio eficiente de ficheros de cualquier tamaño.
    - **STREAM**: intercambio de datos al vuelo, sin un tamaño conocido previamente, ideal para aplicaciones multimedia.
- **Fase de desconexión**. Uno de los nodos (el anunciante o el descubridor) toma la decisión de finalizar la conexión, y la cierra de forma unilateral. El otro nodo detecta la desconexión y libera sus recursos.

La conexión de los dispositivos puede seguir distintas **estrategias**, según la topología de red que deseemos para nuestra aplicación:

- **P2P_STAR**: para topologías en estrella, que permite la conexión de 1 a N. Es la estrategia idónea cuando se desea tener un dispositivo anunciante, al que conectar varios dispositivos de forma simultánea.
- **P2P_CLUSTER**: establece grupos con conexiones de M a N, es decir, donde cada dispositivo puede iniciar conexiones hacia M dispositivos y recibir conexiones de N dispositivos. Es la estrategia más flexible, puesto que forma una topología de tipo malla. Sin embargo presenta un menor ancho de banda y una mayor latencia que el resto de soluciones. Esta estrategia es perfecta cuando se intercambian pequeñas cargas que no necesitan pasar por un punto central.

- **P2P_POINT_TO_POINT**: la estrategia más sencilla, pero recién incorporada en la última versión del SDK. Permite la conexión punto a punto entre dos dispositivos dentro de alcance. Consigue el máximo ancho de banda, pero no permite múltiples conexiones.

Para que los dispositivos anunciantes y descubridores puedan detectarse, deben tener dos parámetros en común:

- La **estrategia**: por motivos de operación, la estrategia para formar y gestionar la red debe ser la misma. P2P_CLUSTER es la estrategia por defecto.
- El **serviceID**: un identificador que permite segmentar fácilmente nuestras aplicaciones. Por ejemplo: tenemos un cartel turístico inteligente con una RP3, que lanza mensajes a los turistas cercanos. Nos interesará que la aplicación del móvil sólo busque balizas de este servicio, y no otras balizas relacionadas con otros asuntos. Una buena práctica es utilizar el nombre de paquete de nuestra app (por ejemplo, `com.google.example.mipaquete`).

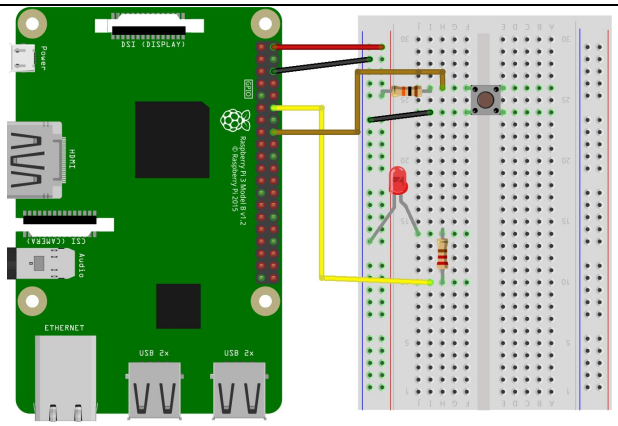
Nota: *montaje hardware para los ejercicios de esta unidad*

El objetivo de esta unidad es conocer y experimentar con las principales tecnologías de comunicaciones utilizadas en el Internet de las Cosas, por lo que la interacción con el hardware no es tan relevante como en la unidad anterior. Para agilizar el desarrollo de los ejercicios, se utilizará un único montaje hardware con un LED como salida y un pulsador como entrada.

Material necesario:

- un LED
- una resistencia de 220 Ω (color de resistencia rojo, rojo, marrón)
- un pulsador de 4 pines
- una resistencia de 10 kΩ (color de resistencia marrón, negro, naranja)
- tablero de prototipos y cables

Monta los componentes sobre la placa de prototipos y conectalos a la Raspberry Pi. El siguiente esquema te muestra cómo hacerlo:



+ Para el LED:

- Conecta el pin de salida GPIO elegido (BCM18) a un extremo de una resistencia de 220 Ω en serie.

- Conecta el otro extremo de la resistencia al ánodo del LED (pata más larga).

- Conecta el cátodo del LED (pata más corta) a tierra.

+ Para el pulsador:

- Conecta uno de los pines con una resistencia de pull-up de 10 kΩ a Vcc, y con un cable al pin de entrada GPIO elegido (BCM23).

- Conecta el otro pin a tierra.

Ejercicio: *Conectividad básica con Nearby Connections*

En este primer ejercicio vamos a construir la estructura software necesaria para trabajar con Nearby Connections. Puesto que necesitamos 2 equipos Android para establecer esta comunicación, utilizaremos el terminal móvil y la Raspberry Pi 3. El móvil mostrará una app con un botón para encender y apagar el LED de la RP3, y el API de Nearby Connections nos ofrecerá un enlace directo con ella. En este ejemplo, lo más lógico es que sea la RP3 la que ofrezca el LED a su entorno como un servicio disponible, así que será ella la que estará en modo anunciante. Cuando pulsemos el botón del terminal móvil se realizará su descubrimiento, el proceso de conexión con la RP3, y el envío del comando para actuar sobre el LED.

1. Crea un nuevo proyecto, indicando que vas a desarrollar tanto para teléfono como para Android Things. Esto generará un proyecto con dos módulos java, uno para la app del terminal móvil, y uno para la app de la RP3. Utiliza los siguientes datos:

```
Application name: Nearby Connections

☑ Phone and Tablet

      API 16: Android 4.1 (Jelly Bean)
```

```
☑ Android Things
        API 27: Android 8.1 (Oreo)
Add an activity to Mobile: Empty Activity
Add an activity to Things: Android Things Empty Activity
        ☐ Generate a UI layout File
```

2. Abre el desplegable de los scripts Gradle, verás que aparece un script de proyecto y uno específico para cada módulo. Nearby Connections forma parte del SDK de Google Play Services, así que para utilizarlo será necesario añadir la siguiente dependencia en el *build.gradle* de los dos módulos *(Module: mobile* y *Module: things)*:

```
dependencies {
    …
    implementation 'com.google.android.gms:play-services-nearby:11.8.0'
   …
}
```

3. Ahora añade los permisos necesarios en el manifiesto (Android*Manifest.xml*) de ambas aplicaciones: la de **mobile** y la de **things**. Los necesarios para el uso de Nearby Communications (Bluetooth y Wifi) deberán estar en ambos manifiestos, y el de **USE_PERIPHERAL_IO** sólo en el de **things**, para poder utilizar las E/S de nuestra RP3:

```
<manifest …>
<!—Necesario para Nearby Connections -->
<uses-permission android:name="android.permission.BLUETOOTH" />
<uses-permission android:name="android.permission.BLUETOOTH_ADMIN" />
<uses-permission android:name="android.permission.ACCESS_WIFI_STATE" />
<uses-permission android:name="android.permission.CHANGE_WIFI_STATE" />
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION"
/>
<!—Necesario para usar las GPIO de la RP3 -->
<uses-permission android:name="com.google.android.things.permis-
sion.USE_PERIPHERAL_IO" />

    <application …>
```

El permiso `ACCESS_COARSE_LOCATION` se considera peligroso, pero es necesario para el uso de Nearby Connections. En el caso de Android Things no hay problema, puesto que los permisos se conceden en el arranque. Pero en el caso del terminal móvil, será necesario solicitar al usuario este permiso en tiempo de ejecución. Volveremos a esto más adelante.

4. Opcionalmente, puedes aprovechar para incluir en el manifiesto de la aplicación móvil los temas y estilos que más te gusten, como por ejemplo:

```
<application
    …
    android:theme="@style/Theme.AppCompat.Light.NoActionBar">
    <activity
          …
```

5. Comenzamos ahora con el código para la app de nuestra RP3. Abre la clase `MainActivity` del módulo **things**, y añade el siguiente contenido:

```
public class MainActivity extends Activity {
  // Consejo: utiliza como SERVICE_ID el nombre de tu paquete
  private static final String SERVICE_ID = "com.example.mipaquete";
  private static final String TAG = "Things:";
  private final String PIN_LED = "BCM18";
  public Gpio mLedGpio;
  private Boolean ledStatus;

  @Override protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    // Configuración del LED
    ledStatus = false;
    PeripheralManager service = PeripheralManager.getInstance();
    try {
      mLedGpio = service.openGpio(PIN_LED);
      mLedGpio.setDirection(Gpio.DIRECTION_OUT_INITIALLY_LOW);
    } catch (IOException e) {
      Log.e(TAG, "Error en el API PeripheralIO", e);
    }
    // Arrancamos modo anunciante
    startAdvertising();
  }

  private void startAdvertising() {
    Nearby.getConnectionsClient(this).startAdvertising(
      "Nearby LED", SERVICE_ID, mConnectionLifecycleCallback,
      new AdvertisingOptions(Strategy.P2P_STAR))
      .addOnSuccessListener(new OnSuccessListener<Void>() {
        @Override public void onSuccess(Void unusedResult) {
          Log.i(TAG, "Estamos en modo anunciante!");
        }
      })
      .addOnFailureListener(new OnFailureListener() {
        @Override public void onFailure(@NonNull Exception e) {
          Log.e(TAG, "Error al comenzar el modo anunciante", e);
        }
      });
  }

  private void stopAdvertising() {
    Nearby.getConnectionsClient(this).stopAdvertising();
    Log.i(TAG, "Detenido el modo anunciante!");
  }

  private final ConnectionLifecycleCallback mConnectionLifecycleCallback =
    new ConnectionLifecycleCallback() {
      @Override public void onConnectionInitiated(
        String endpointId, ConnectionInfo connectionInfo) {
        // Aceptamos la conexión automáticamente en ambos lados.
        Nearby.getConnectionsClient(getApplicationContext())
          .acceptConnection(endpointId, mPayloadCallback);
```

```
        Log.i(TAG, "Aceptando conexión entrante sin autenticación");
      }

      @Override public void onConnectionResult(String endpointId,
                               ConnectionResolution result) {
        switch (result.getStatus().getStatusCode()) {
          case ConnectionsStatusCodes.STATUS_OK:
            Log.i(TAG, "Estamos conectados!");
            stopAdvertising();
            break;
          case ConnectionsStatusCodes.STATUS_CONNECTION_REJECTED:
            Log.i(TAG, "Conexión rechazada por uno o ambos lados");
            break;
          case ConnectionsStatusCodes.STATUS_ERROR:
            Log.i(TAG, "Conexión perdida antes de ser aceptada");
            break;
        }
      }

      @Override
      public void onDisconnected(String endpointId) {
        Log.i(TAG, "Desconexión del endpoint, no se pueden " +
          "intercambiar más datos.");
        startAdvertising();
      }
    };

  private final PayloadCallback mPayloadCallback = new PayloadCallback() {
    @Override public void onPayloadReceived(String endpointId,
                                Payload payload) {
      String message = new String(payload.asBytes());
      Log.i(TAG, "Se ha recibido una transferencia desde (" +
        endpointId + ") con el siguiente contenido: " + message);
      disconnect(endpointId);
      switch (message) {
        case "SWITCH":
          switchLED();
          break;
        default:
          Log.w(TAG, "No existe una acción asociada a este " +
            "mensaje.");
          break;
      }
    }

    @Override public void onPayloadTransferUpdate(String endpointId,
                                PayloadTransferUpdate update) {
      // Actualizaciones sobre el proceso de transferencia
    }
  };

  public void switchLED() {
    try {
```

```
      if (ledStatus) {
        mLedGpio.setValue(false);
        ledStatus = false;
        Log.i(TAG, "LED OFF");
      } else {
        mLedGpio.setValue(true);
        ledStatus = true;
        Log.i(TAG, "LED ON");
      }
    } catch (IOException e) {
      Log.e(TAG, "Error en el API PeripheralIO", e);
    }
  }

  protected void disconnect(String endpointId) {
    Nearby.getConnectionsClient(this)
      .disconnectFromEndpoint(endpointId);
    Log.i(TAG, "Desconectado del endpoint (" + endpointId + ").");
    startAdvertising();
  }

  @Override
  protected void onDestroy() {
    super.onDestroy();
    stopAdvertising();
    if (mLedGpio != null) {
      try {
        mLedGpio.close();
      } catch (IOException e) {
        Log.e(TAG, "Error en el API PeripheralIO", e);
      } finally {
        mLedGpio = null;
      }
    }
  }
}
```

Vamos a revisar paso a paso qué hace este código:

1. Al arrancar (método `OnCreate()`) configuramos el pin del LED como salida, y lanzamos el método `startAdvertising()` para que la RP3 comience a anunciarse.
2. En el método `startAdvertising()`, llamamos a la librería Nearby Connections para comenzar el modo anunciante, pasando como parámetros: el nombre de nuestra baliza de anuncio ("**Nearby LED**"), el identificador de servicio, un objeto callback `mConnectionLifecycleCallback` que gestionará todo el proceso de las peticiones de conexión que recibamos, y la estrategia a utilizar. Como queremos que dispositivos externos se conecten a nuestra RP3 para actuar sobre su LED, utilizaremos una estrategia P2P_STAR. El nombre de la baliza y el identificador de servicio serán los mismos que utilizaremos en la app del móvil. Añadimos

además dos listeners para detectar si hemos conseguido iniciar el modo anunciante.

3. El callback `mConnectionLifecycleCallback` es invocado durante el proceso de conexión de un descubridor con nosotros. Dentro sobrescribimos los siguientes métodos:
    a. `onConnectionInitiated()`: el anunciante es avisado a través de este método de que un descubridor solicita conectar. En nuestro código aceptamos automáticamente esta conexión, pasando como parámetro el objeto callback `mPayloadCallback`, que será donde se gestione las transferencias de datos. Éste también sería el lugar idóneo para realizar un proceso de autenticación: ambos extremos reciben un mismo token de conexión en este método, por lo que pueden mostrarlo al usuario para confirmar que es el mismo en ambos dispositivos, por ejemplo a través de la pantalla. La autenticación es opcional, pero el canal que se cree siempre estará encriptado.
    b. `onConnectionResult()`: se invoca este método cuando el proceso de conexión ha finalizado, indicando si se ha creado la conexión correctamente o si ha habido un error. Por sencillez, cuando la conexión ha sido correcta, dejamos de anunciarnos para que no conecten nuevos clientes.
    c. `onDisconnect()`: se invoca este método cuando la comunicación entre ambos dispositivos se pierde antes de poder finalizar el proceso completo de conexión. La utilizaremos para volver a activar el modo anunciante.
4. El callback `mPayloadCallback` es invocado cuando se reciben datos por una conexión establecida con otro dispositivo. El método `onPayloadTransferUpdate()` nos indica el estado del proceso de transferencia, lo que nos permite mostrar el porcentaje transferido al usuario, o detectar posibles errores. Sin embargo, esto sólo es útil para las transferencias de tipo FILE, y nosotros utilizaremos el tipo BYTE. Por otro lado, el método `onPayloadReceived()` recibe la carga tipo BYTE que hemos enviado desde el móvil. Aquí comprobamos el mensaje recibido para encender o apagar el LED de la RP3 mediante `doRemoteAction()`. Cuando recibimos el mensaje, cerramos la conexión con `disconnect()`.
5. El método `disconnect()` lo utilizamos para cerrar nuestro lado de la conexión, lo que será notificado al descubridor, que cerrará automáticamente su lado. Realizamos la desconexión en el receptor del mensaje para asegurarnos que hemos recibido la transferencia correctamente antes de liberar la conexión.

6. Pasamos ahora a trabajar con el módulo para el terminal móvil. Comenzaremos con el layout de la aplicación, dirígete a *mobile* > *res* > *layout* > *activity_main.xml*, y en modo texto, utiliza el siguiente código:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
```

```
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">
    <TextView
        android:id="@+id/textView1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Pulse el botón para comenzar"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintVertical_bias="0.3" />
    <Button
        android:id="@+id/buttonLED"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Acción!"
        style="@style/Widget.AppCompat.Button.Colored"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintVertical_bias="0.6" />
</android.support.constraint.ConstraintLayout>
```

Este layout estará formado por una etiqueta de texto (*textView1*) que nos dará feedback sobre el estado actual, y un botón (*buttonLED*) que lanzará las acciones.

7. A continuación, abre la clase `MainActivity` del módulo **mobile**, y añade el siguiente contenido:

```
public class MainActivity extends AppCompatActivity {
 private static final int MY_PERMISSIONS_REQUEST_ACCESS_COARSE_LOCATION=1;
  // Consejo: utiliza como SERVICE_ID el nombre de tu paquete
  private static final String SERVICE_ID = "com.example.mipaquete";
  private static final String TAG = "Mobile:";
  Button botonLED;
  TextView textview;

  @Override protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    textview = (TextView) findViewById(R.id.textView1);
    botonLED = (Button) findViewById(R.id.buttonLED);

    botonLED.setOnClickListener(new View.OnClickListener() {
      public void onClick(View v) {
        Log.i(TAG, "Boton presionado");
        startDiscovery();
```

```
        textview.setText("Buscando...");
      }
    });

    // Comprobación de permisos peligrosos
    if (ContextCompat.checkSelfPermission(this,
      Manifest.permission.ACCESS_COARSE_LOCATION)
      != PackageManager.PERMISSION_GRANTED) {
      ActivityCompat.requestPermissions(this,
        new String[]{Manifest.permission.ACCESS_COARSE_LOCATION},
        MY_PERMISSIONS_REQUEST_ACCESS_COARSE_LOCATION);
    }
  }

  // Gestión de permisos
  @Override  public void onRequestPermissionsResult(int requestCode,
                    String permissions[], int[] grantResults) {
    switch (requestCode) {
      case MY_PERMISSIONS_REQUEST_ACCESS_COARSE_LOCATION: {
        if (grantResults.length > 0
          && grantResults[0] == PackageManager.PERMISSION_GRANTED) {
          Log.i(TAG, "Permisos concedidos");
        } else {
          Log.i(TAG, "Permisos denegados");
          textview.setText("Debe aceptar los permisos para comenzar");
          botonLED.setEnabled(false);
        }
        return;
      }
    }
  }

  private void startDiscovery() {
    Nearby.getConnectionsClient(this).startDiscovery(SERVICE_ID,
      mEndpointDiscoveryCallback, new DiscoveryOptions(Strategy.P2P_STAR))
      .addOnSuccessListener(new OnSuccessListener<Void>() {
        @Override public void onSuccess(Void unusedResult) {
          Log.i(TAG, "Estamos en modo descubrimiento!");
        }
      })
      .addOnFailureListener(new OnFailureListener() {
        @Override public void onFailure(@NonNull Exception e) {
          Log.e(TAG, "Modo descubrimiento no iniciado.", e);
        }
      });
  }

  private void stopDiscovery() {
    Nearby.getConnectionsClient(this).stopDiscovery();
    Log.i(TAG, "Se ha detenido el modo descubrimiento.");
  }

  private final EndpointDiscoveryCallback mEndpointDiscoveryCallback =
```

```
    new EndpointDiscoveryCallback() {

      @Override public void onEndpointFound(String endpointId,
                          DiscoveredEndpointInfo discoveredEndpointInfo) {
        Log.i(TAG, "Descubierto dispositivo con Id: " + endpointId);
        textview.setText("Descubierto: " + discoveredEndpointInfo
               .getEndpointName());
        stopDiscovery();
        // Iniciamos la conexión con al anunciante "Nearby LED"
        Log.i(TAG, "Conectando...");
        Nearby.getConnectionsClient(getApplicationContext())
               .requestConnection("Nearby LED",endpointId,
          mConnectionLifecycleCallback)
          .addOnSuccessListener(new OnSuccessListener<Void>() {
            @Override public void onSuccess(Void unusedResult) {
              Log.i(TAG, "Solicitud lanzada, falta que ambos " +
                "lados acepten");
            }
          })
          .addOnFailureListener(new OnFailureListener() {
            @Override public void onFailure(@NonNull Exception e) {
              Log.e(TAG, "Error en solicitud de conexión", e);
              textview.setText("Desconectado");
            }
          });
      }

      @Override public void onEndpointLost(String endpointId) {}
    };

  private final ConnectionLifecycleCallback mConnectionLifecycleCallback =
    new ConnectionLifecycleCallback() {

      @Override public void onConnectionInitiated(
        String endpointId, ConnectionInfo connectionInfo) {
        // Aceptamos la conexión automáticamente en ambos lados.
        Log.i(TAG, "Aceptando conexión entrante sin autenticación");
        Nearby.getConnectionsClient(getApplicationContext())
               .acceptConnection(endpointId, mPayloadCallback);
      }

      @Override public void onConnectionResult(String endpointId,
                                               ConnectionResolution result) {
        switch (result.getStatus().getStatusCode()) {
          case ConnectionsStatusCodes.STATUS_OK:
            Log.i(TAG, "Estamos conectados!");
            textview.setText("Conectado");
            sendData(endpointId, "SWITCH");
            break;
          case ConnectionsStatusCodes.STATUS_CONNECTION_REJECTED:
            Log.i(TAG, "Conexión rechazada por uno o ambos lados");
            textview.setText("Desconectado");
            break;
```

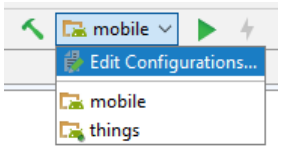
```
          case ConnectionsStatusCodes.STATUS_ERROR:
            Log.i(TAG, "Conexión perdida antes de poder ser " +
                    "aceptada");
            textview.setText("Desconectado");
            break;
        }
      }

      @Override public void onDisconnected(String endpointId) {
        Log.i(TAG, "Desconexión del endpoint, no se pueden " +
              "intercambiar más datos.");
        textview.setText("Desconectado");
      }
    };

  private final PayloadCallback mPayloadCallback = new PayloadCallback() {
    // En este ejemplo, el móvil no recibirá transmisiones de la RP3
    @Override public void onPayloadReceived(String endpointId,
                                Payload payload) {
      // Payload recibido
    }

    @Override public void onPayloadTransferUpdate(String endpointId,
                               PayloadTransferUpdate update) {
      // Actualizaciones sobre el proceso de transferencia
    }
  };

  private void sendData(String endpointId, String mensaje) {
    textview.setText("Transfiriendo...");
    Payload data = null;
    try {
      data = Payload.fromBytes(mensaje.getBytes("UTF-8"));
    } catch (UnsupportedEncodingException e) {
      Log.e(TAG, "Error en la codificación del mensaje.", e);
    }
    Nearby.getConnectionsClient(this).sendPayload(endpointId, data);
    Log.i(TAG, "Mensaje enviado.");
  }
}
```

Puesto que necesitamos el permiso `ACCESS_COARSE_LOCATION`, y éste se considera un permiso peligroso, deberemos realizar la solicitud al usuario en tiempo de ejecución. Para ello, utilizamos `checkSelfPermission()` y `requestPermissions()` en el método `onCreate()`. Por otro lado, utilizamos `extends` para heredar de la clase `AppCompatActivity`, y sobreescribimos el método `onRequestPermissionsResult()`, de forma que podamos realizar distintas acciones si se dispone del permiso o no. En este caso y por sencillez, si no se dispone del permiso se desactiva el botón, y se muestra un texto informativo al usuario.

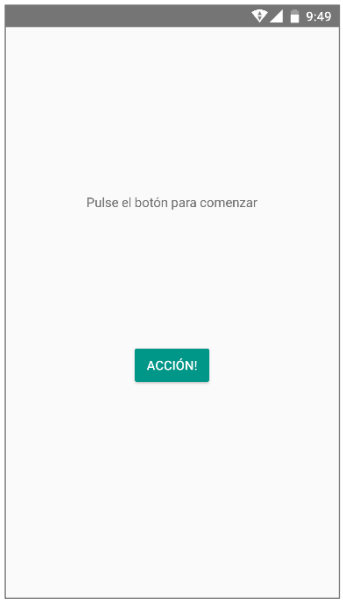
Una vez resuelta la gestión de permisos peligrosos, el resto de aplicación funciona de la siguiente forma:

1. En el método `onCreate()` añadimos un listener para detectar el click del botón. Cuando se realiza, se lanza el método `startDiscovery()`.
2. En el método `startDiscovery()`, llamamos a la librería Nearby Connections para comenzar el modo descubridor, pasando como parámetros: el identificador de servicio (el mismo que el utilizado en el módulo de **things**), el objeto callback `mEndpointDiscoveryCallback` que gestionará todo el proceso de descubrimiento de anunciantes, y la estrategia a utilizar (que será la misma que hemos utilizado en el módulo de **things**, es decir, **P2P_STAR**).
3. El callback `mEndpointDiscoveryCallback` es invocado cada vez que se detecta un anunciante (lanzando el método `onEndpointFound()`) o que desaparece (`onEndpointLost()`). Estos anunciantes se registran con un endpointID, que los identifica de forma unívoca. En nuestro caso y por sencillez, cuando detectemos un anunciante con nuestro identificador de servicio podremos asumir que es el de nuestra RP3, así que conectaremos directamente con él mediante `requestConnection()`, pasando el nombre de la baliza de anuncio (el mismo que el utilizado en el módulo de **things**, "**Nearby LED**"), el endpointID que tiene asociado, y el callback `mConnectionLifecycleCallback` para gestionar el proceso de conexión. Añadimos además dos listeners para detectar si hemos conseguido enviar correctamente esta solicitud de conexión.
4. El callback `mConnectionLifecycleCallback` es invocado durante el proceso de conexión con el anunciante. Dentro sobreescribimos los siguientes métodos:
    a. `onConnectionInitiated()`: aunque nosotros lancemos la solicitud de conexión, ambos lados deben aceptarla. Recibiremos un token de seguridad para decidir si queremos conectar o no con el anunciante. En nuestro caso, aceptaremos la conexión automáticamente sin realizar autenticación, pasando como parámetro el objeto callback `mPayloadCallback`, que será el que gestione las transferencias de datos.
    b. `onConnectionResult()`: se invoca este método cuando el proceso de conexión ha finalizado, indicando si se ha creado la conexión correctamente o si ha habido un error. Si la conexión es correcta, utilizamos el método `sendData()` para enviar el mensaje "ACCION" a la RP3.
    c. `onDisconnect()`: se invoca este método cuando la comunicación entre ambos dispositivos se pierde antes de poder finalizar el proceso completo de conexión.
5. El callback `mPayloadCallback` es invocado cuando se reciben datos del anunciante. En este ejercicio no recibiremos nada, pero lo dejaremos como referencia.
6. El método `sendData()` invoca al método `sendPayload()` de la librería Nearby Connections para enviar el mensaje "ACCION" al anunciante.

8. Carga y lanza cada módulo en su dispositivo destino. Deberás seleccionar primero uno de los dos módulos, presionar en “Run”, y cuando acabe hacer lo mismo con el otro módulo.



9. Comprueba que funciona correctamente la aplicación, y que el LED de la RP3 se enciende y apaga cuando se pulsa el botón del terminal móvil.



Aunque este es un ejercicio muy sencillo, habrás detectado varias posibilidades de mejora para que el sistema sea más versátil y eficiente:

- Por un lado, estamos conectando nuestro descubridor a una baliza Nearby predefinida, de nombre “Nearby LED”. Sin embargo, el sistema sería más versátil si al entrar en el modo descubridor, mostrara por pantalla una lista de las balizas encontradas, y que fuera el usuario el que seleccionara a qué baliza conectar.
- Por otro lado, cada vez que pulsamos el botón “Conmutar LED” realizamos un proceso que incluye la búsqueda de nuestra baliza “Nearby LED”, la conexión, la transferencia y la desconexión. Si el accionamiento del botón es ocasional, no supone un problema, pero si queremos realizar acciones más seguidas, esto supone una pérdida de eficiencia. Si pruebas a encender y apagar varias veces el LED, verás que hay momentos en

que la comunicación se alarga incluso hasta decenas de segundos (activación y desactivación de wifi/bluetooth, búsqueda y conexión de dispositivos…). Si quisiéramos utilizar Nearby Connections como control remoto de un robot hecho con Android Things, esto sería excesivamente limitante.

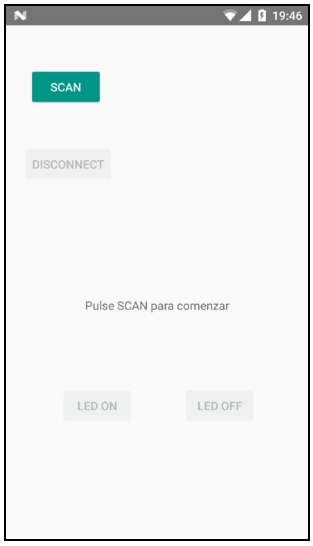
Práctica: *Conectividad avanzada con Nearby Connections*

Tras realizar el ejercicio anterior, modifica el proyecto para eliminar los 2 problemas que acabamos de comentar:

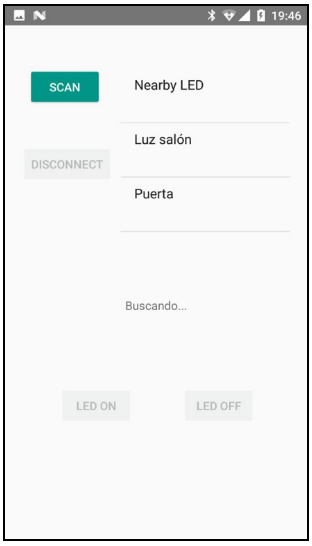
a) Primero, modifica el módulo **mobile** para que, en lugar de realizar todas las acciones al pulsar el botón "*Conmutar LED*", tengamos los siguientes botones: "*Scan*" (para buscar anunciantes), "Connect" (para conectar con "Nearby LED"), "*ON*" (para encender el LED), "*OFF*" (para apagar el LED) y "*Disconnect*" (para cerrar la conexión con el anunciante). Ten en cuenta que esto significa que también deberás modificar el código del módulo **things** (ahora ya no hay un mensaje ACCION, sino un mensaje ON y un mensaje OFF), y que la función disconnect que utilizaba el anunciante, ahora la utilizará el descubridor.

b) A continuación, modifica el módulo **mobile** para que la baliza a la que nos conectamos no sea "Nearby LED", sino el nombre de la primera baliza que recibamos en modo descubrimiento. Opcional: muestra en la app móvil un *listview* de las balizas detectadas, y cuando pulses sobre una de ellas, inicia el proceso de conexión con ella (ya no será necesario el botón de "Connect").

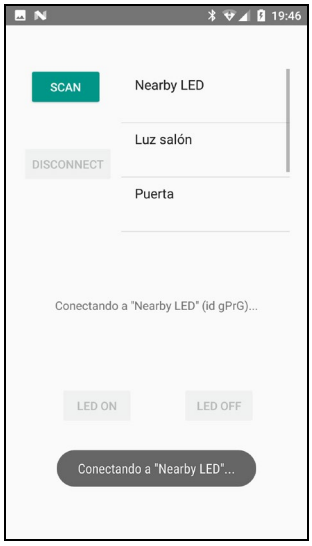
Lanza el nuevo proyecto. Verás como la activación y desactivación del LED ahora es instantánea, lo que nos abre la posibilidad a multitud de proyectos de control en tiempo real. A continuación se muestra un posible ejemplo de cómo quedaría esta aplicación:



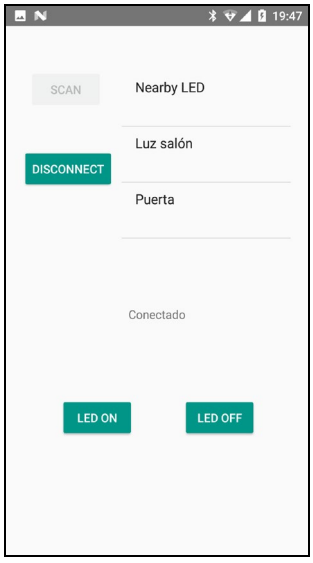
Pantalla inicial



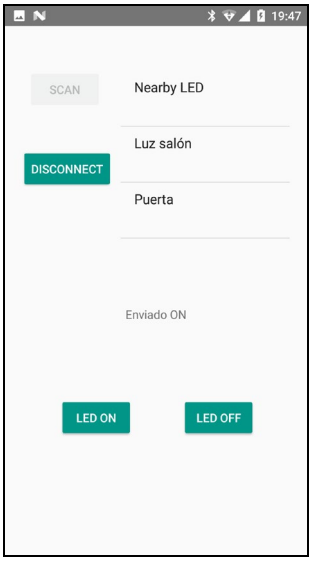
Al pulsar SCAN



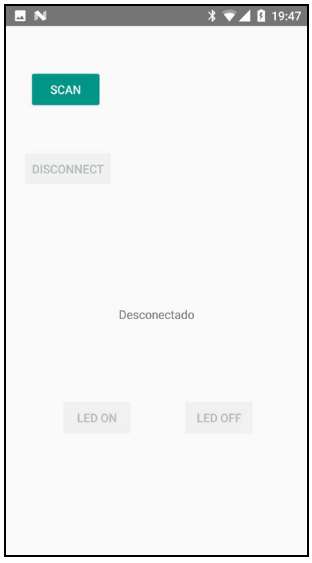
Click sobre “Nearby LED”



Conexión correcta



Al pulsar LED ON



Click sobre DISCONNECT

Como hemos visto, *Nearby Connections* es una tecnología muy interesante para realizar todo tipo de interacciones offline, ya que resulta imbatible a la hora de reducir la latencia y aumentar el ancho de banda en las comunicaciones entre dispositivos cercanos. Pero presenta además una ventaja fundamental para el universo IoT. Y es que el primer paso a realizar cuando tenemos un nuevo dispositivo inteligente para nuestro hogar es conectarlo a la red inalámbrica, pero la falta de pantallas y teclados es un problema para esta primera configuración. En estas situaciones, realizar una app móvil que permite al usuario emparejarse con el dispositivo IoT para

indicarle a qué wifi conectarse y con qué contraseña, simplifica enormemente el proceso.

Ejercicio: *Configuración remota del wifi del dispositivo IoT*

En este ejercicio experimentarás con el uso de Nearby Connections para configurar dispositivos IoT cercanos. En este caso, tendremos una app móvil que buscará balizas Nearby, se conectará a ella, y añadirá la configuración wifi de tu punto de acceso doméstico a la lista de redes de la RP3. Para ello necesitarás tener a mano el nombre de tu punto de acceso y la contraseña, o crear un nuevo AP con un terminal móvil (distinto al que vayas a utilizar en este ejercicio).

Al añadir una red wifi, ésta queda persistida en la lista de redes, y la RP3 se conectará siempre a ella, estemos en la app que estemos. Por lo tanto, esta es una configuración que afecta al S.O. de nuestra RP3, no sólo a nuestra app: es una configuración global a nivel de sistema.

> ***NOTA:*** *según la gestión actual de permisos de Android, sólo la app que ha creado una nueva entrada en la lista de redes wifi, puede posteriormente eliminarla.*

1. Comienza a partir del ejercicio anterior, donde utilizábamos Nearby Connections para actuar sobre el LED de la RP3.
2. En el módulo **things**, añade una nueva clase (botón derecho en MainActivity, y luego click en *New* > *Java Class*). Utiliza como nombre **WifiUtils**.
3. Utiliza el siguiente código para esta nueva clase:

```
public class WifiUtils {
  WifiManager wifiManager;
  WifiConfiguration wifiConfig;
  private static final String TAG = "WifiUtils";
  Context context;

  public WifiUtils(Context context) {
    this.context = context;
    wifiManager = (WifiManager) context.getSystemService(Context
         .WIFI_SERVICE);
    wifiConfig = new WifiConfiguration();
  }

  public void listNetworks() {
    List<WifiConfiguration> redes = wifiManager.getConfiguredNetworks();
    Log.i(TAG, "Lista de redes configuradas:\n " + redes.toString());
  }

  public String getConnectionInfo() {
    Log.i(TAG, "Red actual: " + wifiManager.getConnectionInfo()
      .toString());
    return new String(wifiManager.getConnectionInfo().getSSID() + ", " +
      wifiManager.getConnectionInfo().getLinkSpeed() + " Mbps, (RSSI: " +
      wifiManager.getConnectionInfo().getRssi() + ")");
  }
```

```
public void removeAllAPs() {
  // Solo se pueden eliminar las redes que haya creado esta app!!
  // Si el resultado es false, no se ha eliminado
  List<WifiConfiguration> redes = wifiManager.getConfiguredNetworks();
  wifiManager.disconnect();
  for (WifiConfiguration red : redes) {
    Log.i(TAG, "Intento de eliminar red  " + red.SSID + " con " +
          "resultado " + wifiManager.removeNetwork(red.networkId));
  }
  wifiManager.reconnect();
}

public int connectToAP(String networkSSID, String networkPasskey) {
  WifiManager wifiManager = (WifiManager) context.getSystemService
       (Context.WIFI_SERVICE);
  for (ScanResult result : wifiManager.getScanResults()) {
    if (result.SSID.equals(networkSSID)) {
      String securityMode = getScanResultSecurity(result);
      WifiConfiguration wifiConfiguration = createAPConfiguration
             (networkSSID, networkPasskey, securityMode);
      int res = wifiManager.addNetwork(wifiConfiguration);
      Log.i(TAG, "Intento de añadir red: " + res);
      boolean b = wifiManager.enableNetwork(res, true);
      Log.i(TAG, "Intento de activar red: " + b);
      wifiManager.setWifiEnabled(true);
      boolean changeHappen = wifiManager.saveConfiguration();
      if (res != -1 && changeHappen) {
        Log.i(TAG, "Cambio de red correcto: " + networkSSID);
      } else {
        Log.i(TAG, "Cambio de red erróneo.");
      }
      return res;
    }
  }
  return -1;
}

private String getScanResultSecurity(ScanResult scanResult) {
  final String cap = scanResult.capabilities;
  final String[] securityModes = {"WEP", "PSK", "EAP"};
  for (int i = securityModes.length - 1; i >= 0; i--) {
    if (cap.contains(securityModes[i])) {
      return securityModes[i];
    }
  }
  return "OPEN";
}

private WifiConfiguration createAPConfiguration(String networkSSID,
                    String networkPasskey, String securityMode) {
  WifiConfiguration wifiConfiguration = new WifiConfiguration();
  wifiConfiguration.SSID = "\"" + networkSSID + "\"";
```

```
    if (securityMode.equalsIgnoreCase("OPEN")) {
      wifiConfiguration.allowedKeyManagement.set(WifiConfiguration
            .KeyMgmt.NONE);
    } else if (securityMode.equalsIgnoreCase("WEP")) {
      wifiConfiguration.wepKeys[0] = "\"" + networkPasskey + "\"";
      wifiConfiguration.wepTxKeyIndex = 0;
      wifiConfiguration.allowedKeyManagement.set(WifiConfiguration
            .KeyMgmt.NONE);
      wifiConfiguration.allowedGroupCiphers.set(WifiConfiguration
            .GroupCipher.WEP40);
    } else if (securityMode.equalsIgnoreCase("PSK")) {
      wifiConfiguration.preSharedKey = "\"" + networkPasskey + "\"";
      wifiConfiguration.hiddenSSID = true;
      wifiConfiguration.status = WifiConfiguration.Status.ENABLED;
      wifiConfiguration.allowedGroupCiphers.set(WifiConfiguration
            .GroupCipher.TKIP);
      wifiConfiguration.allowedGroupCiphers.set(WifiConfiguration
            .GroupCipher.CCMP);
      wifiConfiguration.allowedKeyManagement.set(WifiConfiguration
            .KeyMgmt.WPA_PSK);
      wifiConfiguration.allowedPairwiseCiphers.set(WifiConfiguration
            .PairwiseCipher.TKIP);
      wifiConfiguration.allowedPairwiseCiphers.set(WifiConfiguration
            .PairwiseCipher.CCMP);
      wifiConfiguration.allowedProtocols.set(WifiConfiguration.Protocol
            .RSN);
      wifiConfiguration.allowedProtocols.set(WifiConfiguration.Protocol
            .WPA);
    } else {
      Log.i(TAG, "Modo de seguridad no soportado: " + securityMode);
      return null;
    }
    return wifiConfiguration;
  }
}
```

En esta clase tenemos las herramientas básicas para manipular las redes wifi de nuestra RP3:

1. El método `listNetworks()` nos muestra por el *Logcat* el listado de redes que tenemos configuradas en la RP3. La utilizaremos para hacer pruebas durante el desarrollo.
2. El método `getConnectionInfo()` nos muestra la información de la conexión wifi actual. En el *LogCat* nos muestra la información completa, pero además este método nos devuelve un String con información simplificada, ideal para mostrar al usuario en la app móvil.
3. El método `removeAllAPs()` elimina todas las redes inalámbricas que hayan sido configuradas desde esta app.
4. El método `connectToAP()` utiliza como parámetros de entrada el nombre de la red wifi (o SSID) y su contraseña. Este método se encarga de comprobar el método de seguridad que utiliza, y si es compatible, añade la red a la lista de redes. Tras ello, trata de conectar con esta nueva red wifi.

5. Podemos encontrar además un par de métodos privados (no visibles desde fuera de esta clase), y que se utilizan internamente para gestionar la seguridad del punto de acceso a la hora de añadirlo a la lista de redes.

4. Abre ahora la clase `MainActivity` del módulo **things**, y añade las siguientes líneas para crear un objeto `WifiUtil`:

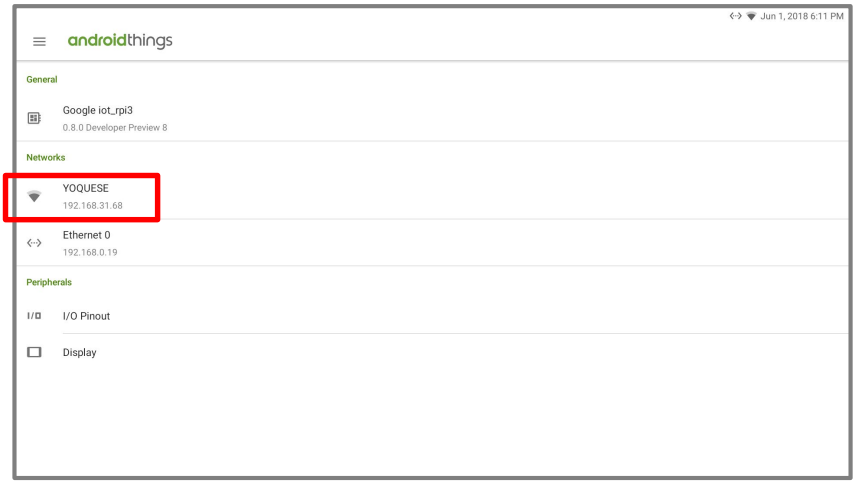
```
public class MainActivity extends Activity {
    …
    private WifiUtils wifiutils;

    @Override
    protected void onCreate(Bundle savedInstanceState) {

        super.onCreate(savedInstanceState);
        wifiutils = new WifiUtils(this);
        …
```

5. Dirígete ahora al método `doRemoteAction()`, y modifica su contenido por el siguiente:

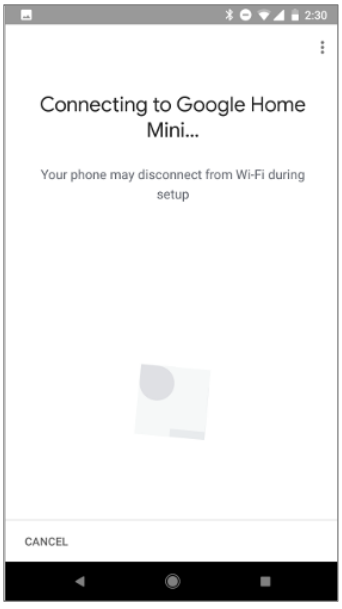
```
public void doRemoteAction() {
    wifiutils.connectToAP("SSID", "pass");
    wifiutils.listNetworks();
    wifiutils.getConnectionInfo();
}
```

Modifica el texto "*SSID*" y "*pass*" por el nombre de tu red wifi y la contraseña, respectivamente. Este sistema debería de funcionar sin problemas en redes WEP, WPA y WPA2-PSK, así como en redes sin contraseña. Tras añadir esta red y conectar con ella, se muestra la lista de redes configuradas por *LogCat*, e información de la conexión actual.

6. Carga el proyecto y comprueba que, al pulsar en el botón "*ACCIÓN!*" del móvil, el LogCat de la RP3 muestra la red añadida. Si además se ha conectado correctamente, aparecerá la información de esta conexión. Otra forma de comprobar que ha conectado correctamente, y que este cambio es a nivel de sistema, es parar la ejecución de la app actual (con el botón de stop), y con una pantalla conectada al HDMI de la RP3, revisar las conexiones actuales:



Hemos realizado un ejercicio muy sencillo. En un proyecto más completo tendríamos un *wizard* que, al abrir, buscaría dispositivos IoT cercanos para configurar. El usuario seleccionaría el dispositivo deseado de esta lista, estableciéndose la conexión Nearby entre ambos. Una vez conectados, la app mostraría un listado de las redes wifi al alcance. El usuario seleccionaría una de estas redes, introduciría su contraseña, y estos parámetros se enviarían al dispositivo IoT. Si la configuración wifi es correcta y la conexión ha tenido éxito, se informaría a la app móvil de que el proceso ha finalizado, mostrando datos de esta conexión (velocidad o nivel de señal).



Práctica: *Configurador wifi avanzado*

Tras realizar el ejercicio anterior, modifica el proyecto para darle más flexibilidad y potencia, eligiendo al menos 3 de las siguientes opciones:

a) Al igual que en la práctica anterior, permite que sea el usuario el que controle la conexión Nearby (escanear, elegir a qué baliza conectar, y desconectar cuando desee).

b) Haz que la información de la red a configurar (SSID y pass) la envíe el terminal móvil, en lugar de estar fija en el código de la RP3.

c) Añade un botón para eliminar todas las redes wifi que se haya configuradas en la RP3.

d) Haz que la app móvil escanee las redes wifi de su alrededor, y que las muestre al usuario. El usuario seleccionará una de ellas, escribirá su contraseña, y esta información (SSID y pass) se enviará a la RP3.

e) Añade un botón para mostrar en la app móvil información sobre la red wifi a la que el usuario está conectado.

f) Añade un botón para mostrar en la app móvil la lista de redes (SSID) que hay configuradas en la RP3.

Preguntas de repaso: API Nearby Communications

1.3. Comunicaciones online

Tal y como su propio nombre indica, los dispositivos de Internet de las Cosas alcanzan su mayor potencial cuando se conectan a la red por excelencia: **Internet**. Esta conectividad permite acceder al equipo remotamente desde cualquier ubicación, en cualquier momento y a través de cualquier tipo de red, siempre que ambos extremos estén conectados a Internet. El dispositivo IoT puede utilizar esta conexión tanto para ofrecer servicios a la red como para consumirlos, obteniendo a su vez información de otros nodos IoT, aplicaciones móviles, bases de datos privadas, servicios de información pública (meteorología, tráfico, calidad del aire…) etc. Esta conexión a Internet abre una nueva dimensión de servicios para nuestras “cosas”.

En este capítulo vamos a explorar las opciones más relevantes para conectar nuestra Raspberry Pi 3 a Internet, e intercambiar servicios con fuentes externas que nos permitan interactuar con ella.

Preguntas de repaso: Comunicaciones online

1.3.1. Servidor web en Android Things

Las páginas web son documentos de hipertexto o hipermedios (texto, imágenes, vídeos…) que nos permiten interaccionar con sistemas remotos para leer noticias, ver anuncios y contenido multimedia, realizar compras online, etc. Su gran versatilidad y universalidad han llevado también al éxito de las aplicaciones web, soluciones software que se ejecutan desde un navegador web, de forma que ya no es necesario tener la aplicación instalada, sino sólo un navegador con acceso a Internet.

Como interfaz con un sistema físico, la web nos aporta este mismo beneficio: no es necesario instalar una aplicación, desde cualquier sistema informático con acceso a Internet y un navegador, podremos acceder y manipular remotamente nuestro sistema.

La Raspberry Pi 3 tiene la suficiente potencia para albergar un sencillo servidor web HTTP, en el que ofrecer una interfaz web con la que controlar el funcionamiento de los sistemas físicos que conectemos. Para ello, necesitamos seguir los siguientes pasos:

- Montar un servidor web HTTP que gestione las peticiones entrantes.
- Crear una página HTML que contenga la interfaz de usuario a mostrar (controles, visualizadores, etc)
- Enlazar el funcionamiento del servidor HTTP con las funciones de nuestro código.

Ejercicio: *Implementación de servidor HTTP en la RP3*

En este ejercicio aprenderás a implementar un servidor web HTTP sencillo en la Raspberry Pi 3, para utilizarlo como interfaz de usuario remota desde cualquier sistema con acceso a Internet. Para ello utilizaremos el projecto NanoHttpd (https://github.com/NanoHttpd/nanohttpd), un servidor HTTP ligero que nos permite realizar proyectos web con un mínimo consumo de recursos.

1. Crea un nuevo proyecto de Android Things.
2. Añade la siguiente dependencia en *build.gradle (Module: app)*:

```
dependencies {
    …
    implementation 'org.nanohttpd:nanohttpd:2.3.1'
}
```

3. Añade los permisos necesarios en el manifiesto (Android*Manifest.xml*). Utilizaremos **INTERNET** para poder abrir el socket donde escuchará el servidor web, y **USE_PERIPHERAL_IO** para utilizar las E/S de nuestro sistema:

```
<manifest …>
    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="com.google.android.things.permission
                                .USE_PERIPHERAL_IO" />
    <application …>
```

4. Crea una nueva clase llamada `WebServer` para recoger los detalles de implementación del servidor y web y manejar las solicitudes entrantes. Esta clase deberá heredar de `NanoHTTPD` y tener el siguiente contenido:

```
public class WebServer extends NanoHTTPD {
  Context ctx;

  public interface WebserverListener {
    //Boolean ledStatus = false;
    Boolean getLedStatus();
    void switchLEDon();
    void switchLEDoff();
  }

  private WebserverListener listener;

  public WebServer(int port, Context ctx, WebserverListener listener) {
    super(port);
    this.ctx = ctx;
    this.listener = listener;
    try {
      start();
      Log.i(TAG, "Webserver iniciado");
    } catch (IOException ioe) {
      Log.e(TAG, "No ha sido posible iniciar el webserver", ioe);
    }
  }

  private StringBuffer readFile() {
    BufferedReader reader = null;
    StringBuffer buffer = new StringBuffer();
    try {
      reader = new BufferedReader(new InputStreamReader(
        ctx.getAssets().open("home.html"),"UTF-8"));
      String mLine;
      while ((mLine = reader.readLine()) != null) {
        buffer.append(mLine);
        buffer.append("\n");
      }
    } catch (IOException ioe) {
      Log.e(TAG, "Error leyendo la página home", ioe);
    } finally {
      if (reader != null) {
        try {
          reader.close();
```

```
        } catch (IOException e) {
          Log.e(TAG, "Error cerrando el reader", e);
        } finally {
          reader = null;
        }
      }
    }
    return buffer;
  }

  @Override
  public Response serve(IHTTPSession session) {
    Map<String, List<String>> parms = session.getParameters();
    // Analizamos los parámetros que ha modificado el usuario
    // Según estos parámetros, ejecutamos acciones en la RP3
    if (parms.get("on") != null) {
      listener.switchLEDon();
    } else if (parms.get("off") != null) {
      listener.switchLEDoff();
    }
    // Obtenemos la web original
    String preweb = readFile().toString();
    // Si queremos mostrar algún valor de salida, la modificamos
    // En este caso, sustituimos palabras clave por strings
    String postweb;
    if (listener.getLedStatus()) {
      postweb = preweb.replaceAll("#keytext", "ENCENDIDO");
      postweb = postweb.replaceAll("#keycolor", "MediumSeaGreen");
      postweb = postweb.replaceAll("#colorA", "#F2994A");
      postweb = postweb.replaceAll("#colorB", "#F2C94C");
    } else {
      postweb = preweb.replaceAll("#keytext", "APAGADO");
      postweb = postweb.replaceAll("#keycolor", "Tomato");
      postweb = postweb.replaceAll("#colorA", "#3e5151");
      postweb = postweb.replaceAll("#colorB", "#decba4");
    }
    return newFixedLengthResponse(postweb);
  }
}
```

En este código vemos cómo es posible disponer de un servidor web en muy pocos pasos:

1. Creamos una interfaz `WebserverListener` con las funciones que ejecutaremos mediante la interfaz web. En este caso `SwitchLedON` y `SwitchLedOFF`.
2. Creamos un constructor que lanza el servidor mediante el puerto suministrado. El método `start()` inicia el servidor web.
3. Con *@Override* sobreescribimos el método `serve`, de forma que podamos interceder en las peticiones del usuario e implementar nuestra lógica de control. Su contenido es muy sencillo, si la petición contiene alguno de los parámetros asociados a los controles de la web, leemos su contenido y

actuamos en consecuencia. Si queremos lanzar alguna acción, llamamos al método correspondiente del objeto `WebserverListener`. Por el contrario, si queremos mostrar algún nuevo valor en la web (como el de una entrada de la Raspberry Pi 3), editaremos el html de salida antes de enviarlo. En este caso, el html contendrá etiquetas clave (como "#keytext"), que al ser procesadas por este método, se modificarán al texto correpondiente ("ENCENDIDO" o "APAGADO").

4. Con el objeto `readfile` obtenemos el contenido de la web a mostrar (**home.html**), almacenándolo en un *StringBuffer* listo para ser servido como respuesta a una petición del navegador.

5. Vamos a almacenar la web html en una carpeta de recursos "Assets" de Android. Para crearla pulsa con el botón derecho sobre la carpeta app, y luego ve a *New* > *Folder* > *Assets Folder*.
6. Crea un fichero html con el siguiente contenido, y añádelo a la carpeta de assets:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html>
  <head>
    <title>Webserver para control remoto del LED</title>
    <style type="text/css">
        * {
         font-family: arial;
         color: white;
        }
        body {
          background: #colorA;
          background: -webkit-linear-gradient(to right, #colorA, #colorB);
          background: linear-gradient(to right, #colorA, #colorB);
        }
        .btn {
          border: none;
          color: white;
          padding: 14px 28px;
          cursor: pointer;
        }
        .on {background-color: #2196F3;}
        .on:hover {background: #0b7dda;}
        .off {background-color: #e7e7e7; color: black;}
        .off:hover {background: #ddd;}
    </style>
    <script type="text/javascript">
          function setON() { window.location = '?on=true'; }
          function setOFF() { window.location = '?off=true'; }
    </script>
  </head>
  <body align="center">
    <h1><br><br>Control remoto de LED<br></h1>
    Estado actual:
    <strong><small><span style="color:#keycolor;">#keytext</span></small>
    </strong><br><br>Lanzar acción<br><br>
```

```
    <button class="btn on" onclick="setON();">ON</button>
    <button class="btn off" onclick="setOFF();">OFF</button>
  </body>
</html>
```

Vemos que es posible generar una página con estilos CSS, puesto que el contenido del html es simplemente texto plano que interpretará el navegador. En este código se generan 2 botones, que al pulsarse llaman a las funciones javascript "`setON()`" y "`setOFF()`". Estas funciones simplemente añaden parámetros a la dirección web de consulta, de forma que podamos conocer qué ha pulsado el usuario. También podemos ver algunas de las palabras clave que se modificarán cuando se genere el html para el cliente, como **"#keycolor**" o "**#keytext**".

7. Ya tenemos la implementación del servidor web, ahora hay que integrarla en nuestra app de Android Things. Desde aquí podremos iniciar y parar el servidor web, e implementar la lógica del *listener* para tomar acciones cuando el usuario nos envía datos a través de la página HTML. Abre la clase `MainActivity` y añade el siguiente contenido:

```
public class MainActivity extends Activity implements WebServer
    .WebserverListener {
  private WebServer server;
  private final String PIN_LED = "BCM18";
  public Gpio mLedGpio;

  @Override protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    server = new WebServer(8180, this, this);
    PeripheralManager service = PeripheralManager.getInstance();
    try {
      mLedGpio = service.openGpio(PIN_LED);
      mLedGpio.setDirection(Gpio.DIRECTION_OUT_INITIALLY_LOW);
    } catch (IOException e) {
      Log.e(TAG, "Error en el API PeripheralIO", e);
    }
  }

  @Override protected void onDestroy() {
    super.onDestroy();
    server.stop();
    if (mLedGpio != null) {
      try {
        mLedGpio.close();
      } catch (IOException e) {
        Log.e(TAG, "Error en el API PeripheralIO", e);
      } finally {
        mLedGpio = null;
      }
    }
  }

  @Override public void switchLEDon() {
    try {
```

```
      mLedGpio.setValue(true);
      Log.i(TAG, "LED switched ON");
    } catch (IOException e) {
      Log.e(TAG, "Error on PeripheralIO API", e);
    }
  }

  @Override public void switchLEDoff() {
    try {
      mLedGpio.setValue(false);
      Log.i(TAG, "LED switched OFF");
    } catch (IOException e) {
      Log.e(TAG, "Error on PeripheralIO API", e);
    }
  }

  @Override public Boolean getLedStatus() {
    try {
      return mLedGpio.getValue();
    } catch (IOException e) {
      Log.e(TAG, "Error on PeripheralIO API", e);
      return false;
    }
  }
}
```

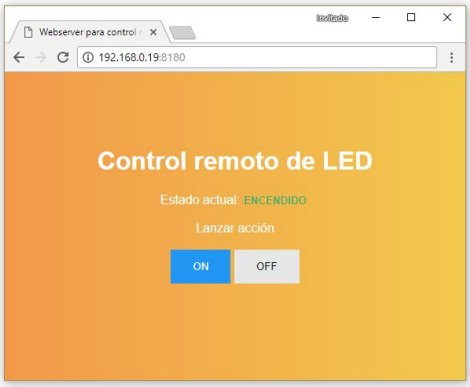
Hay diversos detalles en este código que conviene explicar:

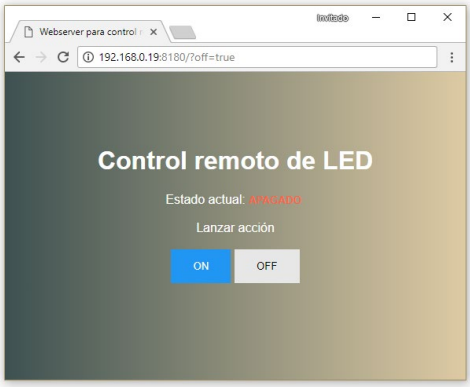
1. Nuestra clase `MainActivity` se declara especificando que implementa el `WebserverListener`, que es la interfaz de callback.
2. Al arrancar la app (`onCreate()`) se crea e inicia el servidor web en el puerto 8180, y pasamos nuestro listener como parámetro.
3. Vemos la implementación de tres métodos del callback: `switchLEDon`, `switchLEDoff` y `getLedStatus`.

8. Carga y lanza el proyecto. Para acceder a la web, utiliza el navegador web del PC con la siguiente URL (donde `android_things_ip` es la IP de tu Raspberry):

```
http://android_things_ip:8180/home.html
```

Comprueba que puedes encender y apagar el LED desde la web, y cómo cambia su contenido en función del estado actual de LED.





Práctica: *Lectura remota del fotorresistor*

Tras realizar el ejercicio anterior, modifica el proyecto para mostrar en la web el valor del fotorresistor.

Preguntas de repaso: Servidor web embebido en la Raspberry Pi 3

1.3.2. Protocolos de comunicaciones

Ya hemos visto cómo es posible montar un servidor web con Android Things, de forma que podemos servir una web html para que un usuario interactúe con nuestra RP3. Y es que HTML se creó precisamente como un lenguaje para que las máquinas pudieran interactuar con las personas, una visualización mejorada de texto, controles y multimedia para enriquecer la experiencia de usuario. Sin embargo, esta solución no nos sirve cuando queremos realizar comunicaciones más automatizadas, como comunicaciones entre máquinas (o machine-to-machine, M2M). Es el caso de querer mostrar los datos de nuestros sensores en otro tipo de sistemas que no sean una web (como otra Raspberry, aplicaciones de escritorio…), hacer públicos los datos de sensores para que cualquiera pueda desarrollar aplicaciones que los muestren, o simplemente para aportar más flexibilidad a la interacción con la Raspberry.

Con el objetivo de que dos elementos independientes puedan intercambiar datos y entenderse mutuamente, surgen los protocolos estándares de comunicaciones. Al implementar un mismo protocolo en dos sistemas aislados, tendremos la seguridad de que podrán intercambiar información y comprender su contenido. Los protocolos de comunicaciones son la columna vertebral de los sistemas IoT, habilitan su conectividad y la interacción de las aplicaciones.

Existen protocolos de comunicaciones a distintos niveles: capa física y de acceso al medio (Wifi, Ethernet…), de red (IP), de transporte (TCP/UDP)… pero lo que estudiaremos aquí son los estándares de capa de aplicación. Estos estándares se encapsulan dentro del resto de capas, por lo que pueden circular a través de cualquier tecnología de red, y permiten que la aplicación de dos sistemas remotos pueda dialogar. En un símil con el lenguaje humano, la capa de aplicación sería el idioma, y la tecnología de acceso sería una carta, la voz humana o un email. Para la aplicación, lo importante es que el idioma sea el mismo. No importa si un extremo envía un email y el otro extremo lo escucha reproducido. Esto nos aporta una gran versatilidad, puesto que cada dispostiivo accederá a Internet de la forma más cómoda para su situación, pero todos intercambiarán datos en un formato común que entienden.

Los protocolos de aplicación están fuertemente ligados al modelo de comunicación utilizado. Existen muchos modelos, entre los que cabe destacar:

- **Request / response**: basado en un escenario de cliente-servidor, un dispositivo cliente lanza un request al servidor, que procesa la petición y devuelve un response. Es el funcionamiento de la web, mediante el protocolo de aplicación HTTP, o su versión reducida para M2M, llamada CoAP.
- **Polling**: un elemento central maneja completamente el acceso al medio de los dispositivos remotos, que no pueden comunicarse a menos que se les de permiso. De forma secuencial, el elemento central consulta uno a uno a todos estos dispositivos, y les pregunta si tienen algo que enviar. Si tienen demasiados datos que enviar, es habitual que se le solicite parar para no alterar los tiempos del resto de dispositivos, y se le vuelve a dar una oportunidad de continuar en la siguiente vuelta.
- **Publish / subscribe**: escenario “muchos a muchos”, donde los dispositivos publican su información, y pueden suscribirse a información de su interés de otros dispositivos. Este tipo de escenarios puede apoyarse en un bróker central que gestione las suscripciones y el reenvío de los datos (como AMQP, MQTT, XMPP), o puede ser completamente descentralizado (como DSS).
- **Push / Pull**: también conocido como productor / consumidor, es un escenario basado en colas, donde los productores inyectan datos en una cola, y los consumidores recogen datos de estas colas. Las colas permiten desacoplar los productores y los consumidores, y actúan como un buffer intermedio.
- **Emparejado exclusivo**: consiste en una conexión persistente y full-duplex entre dos dispositivos (cliente-servidor), donde se pueden realizar intercambio de mensajes o datos entre ambas aplicaciones. Es una conexión con estado, y el servidor es consciente de todas las conexiones que tiene activas. Es el caso de los websockets (RFC 6455), que permite un stream de datos a través de una conexión TCP entre navegadores, apps móviles, equipos IoT, etc.

1.3.2.1. Modelo request/response: servicios web RESTful

Los servicios web son un tipo de comunicación para intercambio de datos entre aplicaciones que se apoya sobre la pila de protocolos web, haciendo uso de HTTP. Al utilizar protocolos nativos para esta red, los mensajes circulan muy

fácilmente a través de Internet, y se ven mínimamente afectados por cortafuegos intermedios. Por el contrario, a nivel de comunicaciones se considera poco efectivo, puesto que está basado en intercambio de texto. Los servicios web son el pilar fundamental sobre el que se construyen las aplicaciones con arquitecturas orientadas a servicios. Por ejemplo, el desarrollo de aplicaciones web suele separarse en un front (la interfaz de usuario) y un back (el servidor) conectado mediante servicios web. El servidor en esta tecnología es un servidor web.

REST (*Representational State Transfer*) es un conjunto de principios para diseño de servicios web, y está enfocado a los recursos de un sistema, a cómo se gestionan e intercambian sus estados entre un cliente y un servidor. Es una arquitectura sin estado, de forma que cada petición request/response debe ser independiente del resto, y deben contener toda la información necesaria para resolver esa petición.

Se conoce como servicio web RESTful a un API web implementado con HTTP y utilizando los principios REST.

1.3.2.1.1 Servidor REST

Ejercicio: *API RESTful para control remoto del LED*

En este ejercicio aprenderás a implementar un servidor web sencillo en la Raspberry Pi 3 para gestionar peticiones REST. Para ello utilizaremos el projecto Restlet (https://restlet.com/open-source/), un framework multi-plataforma muy utilizado para el desarrollo de este tipo de APIs.

1. Crea un nuevo proyecto para Android Things.
2. Añade la siguiente dependencia en *build.gradle (Module: app)*:

```
dependencies {
    …
    implementation 'org.restlet.android:org.restlet:2.3.7'
    implementation 'org.restlet.android:org.restlet.ext.json:2.3.7'
    implementation 'org.restlet.android:org.restlet.ext.nio:2.3.7'
}
```

3. Añade los permisos necesarios en el manifiesto (*AndroidManifest.xml*). Utilizaremos **INTERNET** para poder abrir el socket donde escuchará el servidor web, y **USE_PERIPHERAL_IO** para utilizar las E/S de nuestro sistema:

```
<manifest …>
    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="com.google.android.things.permission
                                .USE_PERIPHERAL_IO" />
    <application …>
```

4. Añade ahora una nueva clase `LEDModel`, y utiliza este código:

```
public class LEDModel {
    private static LEDModel instance = null;
```

```
    PeripheralManager service;
    private Gpio mLedGpio;
    private final String PIN_LED = "BCM18";

    public static LEDModel getInstance() {
        if (instance == null) {
            instance = new LEDModel();
        }
        return instance;
    }

    private LEDModel() {
        service = PeripheralManager.getInstance();
        try {
            mLedGpio = service.openGpio(PIN_LED);
            mLedGpio.setDirection(Gpio.DIRECTION_OUT_INITIALLY_LOW);
        } catch (Exception e) {
            Log.e(TAG, "Error en el API PeripheralIO", e);
        }
    }

    static Boolean setState(boolean state) {
        try {
            getInstance().mLedGpio.setValue(state);
            return true;
        } catch (IOException e) {
            Log.e(TAG, "Error en el API PeripheralIO", e);
            return false;
        }
    }

    public static boolean getState() {
        boolean value = false;
        try {
            value = getInstance().mLedGpio.getValue();
        } catch (IOException e) {
            Log.e(TAG, "Error en el API PeripheralIO", e);
        }
        return value;
    }
}
```

En esta clase modelamos nuestro LED como un objeto. El LED interactuará con el mundo externo mediante dos métodos: `getState()`, que devolverá un valor de true o false según si está encendido o apagado; y `setState()`, al que le indicaremos el valor al que queremos que cambie. El siguiente paso, es convertir estos dos métodos en dos servicios web.

5. Añade una nueva clase de nombre `LEDResource`, que herede de `org.restlet.resource.ServerResource`, y utiliza el siguiente código:

```
public class LEDResource extends ServerResource {

  @Get("json")
```

```
  public Representation getState() {
    JSONObject result = new JSONObject();
    try {
      result.put("estado", LEDModel.getState());
    } catch (Exception e) {
      Log.e(TAG, "Error en JSONObject: ", e);
    }
    return new StringRepresentation(result.toString(),
      MediaType.APPLICATION_ALL_JSON);
  }

  @Post("json")
  public Representation postState(Representation entity) {
    JSONObject query = new JSONObject();
    JSONObject fullresult = new JSONObject();
    String result;
    try {
      JsonRepresentation json = new JsonRepresentation(entity);
      query = json.getJsonObject();
      boolean state = (boolean) query.get("estado");
      Log.d(this.getClass().getSimpleName(), "Nuevo estado del LED: " +
            state);
      if (LEDModel.setState(state)) result = "ok";
      else result = "error";
    } catch (Exception e) {
      Log.e(TAG, "Error: ", e);
      result = "error";
    }
    try {
      fullresult.put("resultado", result);
    } catch (JSONException e) {
      Log.e(TAG, "Error en JSONObject: ", e);
    }
    return new StringRepresentation(fullresult.toString(),
      MediaType.APPLICATION_ALL_JSON);
  }
}
```

El objetivo de esta clase es realizar una representación RESTful del LED, es decir, conectar los servicios web del LED con los métodos que acabamos de crear para el objeto LED. Vemos que hay dos anotaciones: `@Get("json")` y `@Post("json")`. Esto significa que las peticiones GET que nos lleguen sobre el LED, y con contenido tipo JSON, utilizarán el método `getState()`, que hace una llamada al método `LEDModel.getState()`. Por otro lado, las peticiones POST utilizarán el método `postState()`, y harán uso del método `LEDModel.setState()` para modificar el LED.

6. En general, un API REST será una funcionalidad que estará siempre activa en nuestra Raspberry, mientras que en nuestro código principal podemos dedicarnos a realizar otras tareas. La mejor forma de generar un servicio de fondo es mediante `IntentService`, puesto que nos permite manejar peticiones de forma asíncrona y bajo demanda. Para ello, añade una nueva clase `RESTfulService` que herede de `IntentService`:

```
public class RESTfulService extends IntentService {
  private static final String ACTION_START="com.example.mypackage.START";
  private static final String ACTION_STOP="com.example.mypackage.STOP";
  private Component mComponent;

  public RESTfulService() {
    super("RESTfulService");
    Engine.getInstance().getRegisteredServers().clear();
    Engine.getInstance().getRegisteredServers()
      .add(new HttpServerHelper(null));
    mComponent = new Component();
    Router router = new Router(mComponent.getContext()
         .createChildContext());
    // Configuración del webserver
    mComponent.getServers().add(Protocol.HTTP, 8080);
    mComponent.getDefaultHost().attach("/rest", router);
    router.attach("/led", LEDResource.class);  }

  public static void startServer(Context context) {
    Intent intent = new Intent(context, RESTfulService.class);
    intent.setAction(ACTION_START);
    context.startService(intent);
  }

  public static void stopServer(Context context) {
    Intent intent = new Intent(context, RESTfulService.class);
    intent.setAction(ACTION_STOP);
    context.startService(intent);
  }

  @Override protected void onHandleIntent(Intent intent) {
    if (intent != null) {
      final String action = intent.getAction();
      if (ACTION_START.equals(action)) {
        handleStart();
      } else if (ACTION_STOP.equals(action)) {
        handleStop();
      }
    }
  }

  private void handleStart() {
    try {
      mComponent.start();
    } catch (Exception e) {
      Log.e(getClass().getSimpleName(), e.toString());
    }
  }

  private void handleStop() {
    try {
      mComponent.stop();
    } catch (Exception e) {
```

```
      Log.e(getClass().getSimpleName(), e.toString());
    }
  }
}
```

La mayoría de contenido de esta clase está ligada al funcionamiento de `IntentService`. Como una buena práctica, sustituye "`com.example.mypackage`" por el nombre de tu paquete.

La parte más importante para el webserver del API REST la encontramos en el constructor `public RESTfulService()`, donde se realizan las siguientes configuraciones:

- El servidor se lanza a la escucha en el puerto 8080
- Se enrutarán los mensajes que lleguen por la ruta `/rest`. Esta es una forma de utilizar el servidor web no sólo como API REST, sino también para alojar webs en otras rutas, así como en la raíz.
- Se añade un primer recurso para las consultas que lleguen por `/rest/led`, y se redirigen a la clase `LEDResource`. Como habíamos visto, esta clase tiene distintas acciones según le lleguen peticiones GET o POST.

7. Ahora sí, ya podemos pasar a dar contenido a nuestra `MainActivity`. Al utilizar `IntentService`, el contenido será mínimo, simplemente para arrancar y detener el servicio:

```
public class MainActivity extends Activity {
  @Override protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    RESTfulService.startServer(this); // Arrancar el servidor
  }
  @Override protected void onDestroy() {
    super.onDestroy();
    RESTfulService.stopServer(this); // Detener el servidor
  }
}
```

8. Finalmente, ya solo nos queda añadir el servicio `RESTfulService` en el manifiesto. Abre *AndroidManifest.xml* y añádelo entre las etiquetas `<application>` y `<activity>` de la siguiente forma:

```
<application …>
    <service android:name=".RESTfulService"></service>
    <activity …>
```

9. Carga y lanza el proyecto. Realizar la consulta del estado del LED es muy sencillo, al ser una llamada de tipo GET se puede hacer desde la misma barra de direcciones del navegador web del PC. Utiliza la siguiente URL (donde `android_things_ip` es la IP de tu Raspberry):

```
http://android_things_ip:8080/rest/led
```

Sin embargo, para los otros métodos debemos utilizar otros sistemas, como *curl* (Linux), *Postman* (Extensión de Chrome) o *Hurl.io* (servicio online). En este ejercicio, explicaremos como se realizaría mediante *Postman*. Si no lo conoces,

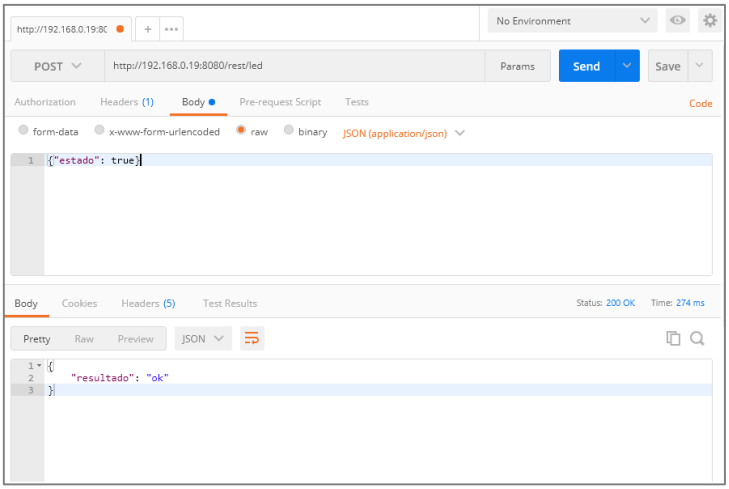
es la oportunidad idónea para que descubras una de las herramientas más relevantes para testear APIs web.

10. Busca Postman en las extensiones de Chrome, instálala y ábrela. Prueba a realizar la llamada anterior con GET, verás que es muy sencillo.

    Para el POST, selecciona en el desplegable a la izquierda de la barra de direcciones, el método POST. A continuación pulsa en *Body* (el de la mitad superior, ya que el de la mitad inferior es para ver las respuestas del servidor), luego marca "*raw*", y luego selecciona en el desplegable "*JSON (Application/json)*". Escribe el contenido del envío POST:

```
{"estado": true}
```

Si funciona correctamente, el LED debería encenderse y el servidor responder con un **OK**.



Ejercicio: *Interacción con API REST desde una web*

En este ejercicio aprenderás a desarrollar una web que lance consultas al API REST que acabamos de desarrollar en la Raspberry. Para ello vamos a desarrollar una web estática, que podremos tener en nuestro disco duro como un fichero .html y abrirlo en cualquier momento. Esta web mostrará un par de botones para encender y apagar el led. Es muy importante remarcar que, a diferencia del ejercicio del servidor web, esta web no está alojada en la Raspberry, sino en nuestro PC, y que la interacción se realiza totalmente mediante servicios web.

1. Abre el Notepad++ o cualquier otro editor de texto, y genera un nuevo fichero "**interruptor.html**".
2. Crea la estructura básica de un archivo html:

```
<html>
  <head>
    <title>Web de control remoto del LED</title>
  </head>
  <body>
```

```
    …
  </body>
</html>
```

3. Vamos a añadir 3 botones en la web. Uno para encender el LED, otro para apagarlo, y otro para consultar el valor actual. Añade el siguiente contenido entre las etiquetas body:

```
<body align="center">
  <h1>Control remoto de LED</h1>
  <input type="button" onClick="switchOn()" value="ON"/>
  <input type="button" onClick="switchOff()" value="OFF"/><br><br>
  <input type="button" onClick="getReading()" value="Valor actual"/>
</body>
```

Abre la web para ver su aspecto. Los botones aún no funcionan, pero podemos ver en el código que al hacer click sobre ellos, se invoca a una serie de funciones: `switchOn()`, `switchOff()` y `getReading()`.

4. Vamos a generar ahora el contenido de estas funciones, encargadas de realizar la comunicación con la RP3. Utilizaremos lenguaje javascript y la librería Ajax jquery. Añade el siguiente contenido entre las etiquetas de head:

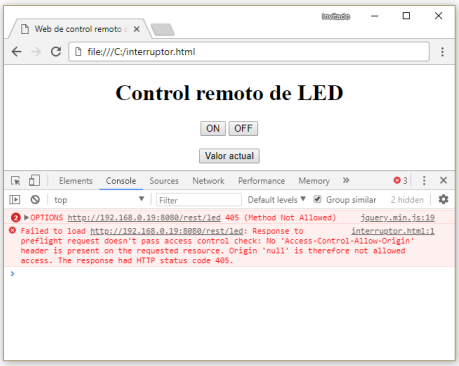
```
<head>
  <title>Web de control remoto del LED</title>
  <script
    src="http://ajax.googleapis.com/ajax/libs/jquery/1.3.2/jquery.min.js"
    type="text/javascript" charset="utf-8"></script>
  <script>
    var android_things_ip = "192.168.0.19";
    var android_things_port = "8080";
    var url = "http://" + android_things_ip + ":" + android_things_port +
      "/rest/led";
    var content_type = "application/json; charset=utf-8";

    function switchOn(){
      $.ajax({type: "POST", url: url, data: "{\"estado\": true}",
      contentType: content_type, dataType: "json"});
    }
    function switchOff(){
      $.ajax({type: "POST", url: url, data: "{\"estado\": false}",
      contentType: content_type, dataType: "json"});
    }
    function getReading(){
      $.ajax({type: "GET", url: url, success: callback});
    }
    function callback(data, status){
      alert(data);
    }
  </script>
</head>
```

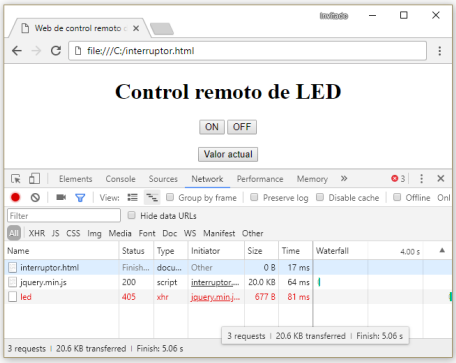
El contenido de este script es muy sencillo: disponemos un una variable con la IP de nuestra RP3, y otra con el puerto. Modifícalas en función de tu situación. A continuación, se genera una variable url a partir de ellas, que apunta a la dirección del API REST.

Por otro lado, tenemos las tres funciones de los botones que aparecen en la web:

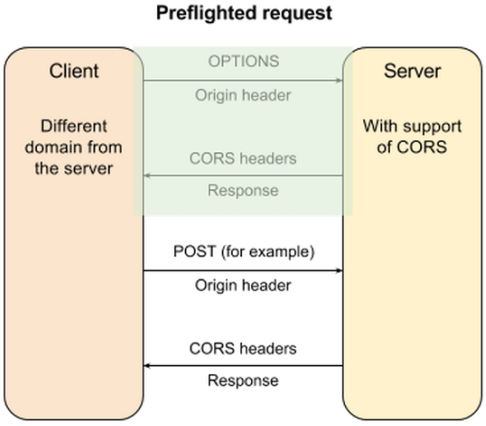
- `switchOn()`: realiza un POST a nuestra API con el contenido {“estado”:true}
- `switchOff()`: realiza un POST a nuestra API con el contenido {“estado”:false}
- `getReading()`: lanza un GET a nuestra API, e indica que la respuesta será gestionada por la función callback.
- `Callback()`: muestra un mensaje con el dato recibido.

5. Abre ahora tu fichero HTML desde Chrome. Comprobarás que al pulsar los botones, no tiene ningún efecto sobre la Raspberry. Abre el modo desarrollador (F12) en vista de consola o de red:





Al utilizar los servicios RESTful desde una aplicación web, nos encontramos sujetos a la política de CORS, que gestiona el acceso de servicios web entre distintos dominios. Nuestro navegador está enviando una consulta “pre-flight”, es decir, antes de lanzar el GET a nuestro servidor, lanza un OPTIONS para comprobar permisos de acceso por dominio, autenticación, métodos HTTP disponibles, etc.

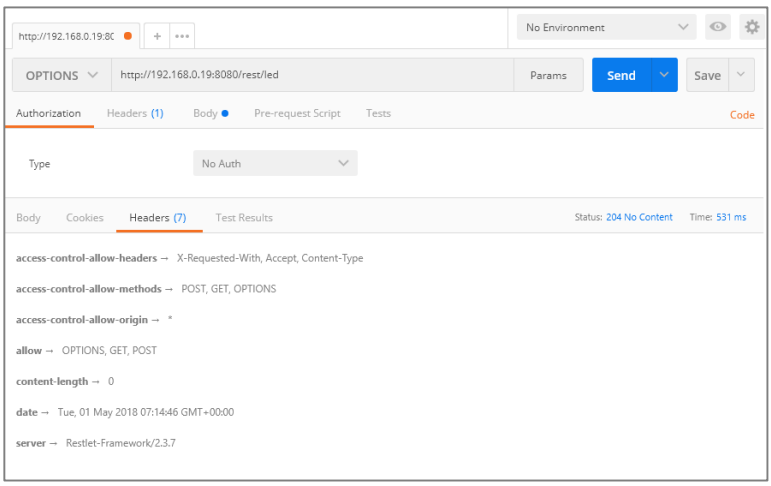


6. Abre la clase `LEDResource`, donde teníamos la gestión de los métodos GET y POST, y añade un nuevo apartado para el método OPTIONS:

```
public class LEDResource extends ServerResource {

  @Options
  public void getCorsSupport() {
    Set<String> head = new HashSet<>();
    head.add("X-Requested-With");
    head.add("Content-Type");
    head.add("Accept");
    getResponse().setAccessControlAllowHeaders(head);

    Set<Method> methods = new HashSet<>();
    methods.add(Method.GET);
    methods.add(Method.POST);
    methods.add(Method.OPTIONS);

    getResponse().setAccessControlAllowMethods(methods);
    getResponse().setAccessControlAllowOrigin("*");
  }
    …
```

En este código estamos editanto la respuesta de nuestro servidor con `getResponse()`, y le estamos añadiendo diversas cabeceras necesarias para soportar CORS. Por ejemplo, indicamos que los métodos que permite la llamada a /REST/LED son GET, POST y OPTIONS, y que se permite el acceso desde cualquier dominio (*Access-Control-Allow-Origin : **).

7. Haz una consulta de tipo OPTIONS con Postman, y comprueba cómo queda la estructura de cabeceras de la respuesta. Averigua qué indica cada una de estas opciones.



8. Vuelve a cargar la web "**interruptor.html**" y comprueba su funcionamiento. Comprueba que los 3 botones ya tienen funcionalidad, pero si activas el modo

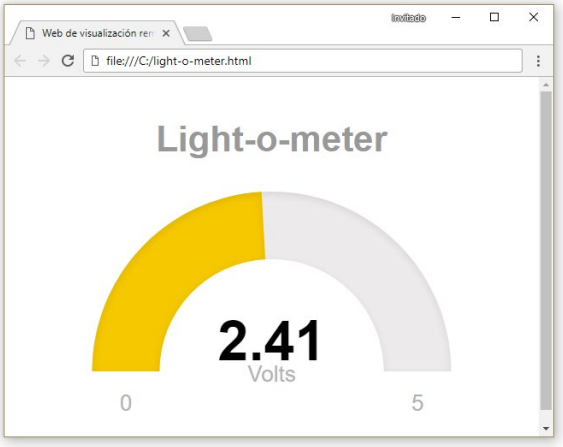
desarrollador (F12) y verificas la consola, sigue apareciendo un error. ¿Qué nos indica este error? (*ver respuesta*[1]).

9. Abre la clase `LEDResource`, y añade en los métodos GET y POST la siguiente línea, justo antes de la llamada al `return`:

```
    getResponse().setAccessControlAllowOrigin("*");
    return …
}
```

10. Vuelve a comprobar el funcionamiento de la web "**interruptor.html**", verás que ya no aparecen avisos en la consola.

Práctica: *API REST para lectura de iluminación*



Añade la electrónica para lectura del fotorresistor. Después, modifica el ejercicio anterior y añade un nuevo recurso REST para él. Deberás crear una clase `LDRmodel` y una clase `LDRResource`, y añadir una ruta **/rest/ldr** al router del webserver. Este recurso sólo servirá peticiones OPTIONS y GET, devolviendo un valor float entre 0 y 5, correspondiente a la tensión detectada (que por el funcionamiento del fotorresistir, será proporcional a la iluminación). La respuesta tendrá el siguiente formato:

```
    {"result": 3.3}
```

Para la visualización, crea una web html con el siguiente código. Deberás editarlo para que apunte a la dirección y puerto de tu servidor:

```
<html>
<head>
<title>Web de visualización remota de iluminación</title>
```

[1] Este error indica que las respuestas recibidas a las consultas GET y POST no contienen la cabecera de "*Access-control-allow-origin*".

```
<script
 src="http://ajax.googleapis.com/ajax/libs/jquery/1.7.2/jquery.min.js"
 type="text/javascript" charset="utf-8"></script>
<script
 src="https://cdnjs.cloudflare.com/ajax/libs/raphael/2.1.0/raphael-min.js"
 type="text/javascript" charset="utf-8"></script>
<script src="http://cdn.jsdelivr.net/justgage/1.0.1/justgage.min.js"
 type="text/javascript" charset="utf-8"></script>
<script>
 var android_things_ip = "192.168.0.19";
 var android_things_port = "8080";
 var url = "http://" + android_things_ip + ":" + android_things_port +
    "/rest/ldr";
 function callback(data, status){
    if (status == "success") {
      volts = parseFloat(data.result);
      volts = volts.toFixed(2);
      g.refresh(volts);
      setTimeout(getReading, 1000);
    } else {
      alert("Hubo un problema");
    }
 }
 function getReading(){
    $.get(url, {}, callback);
 }
 </script>
</head>
<body>
  <div id="gauge" class="200x160px"></div>
  <script>
    var g = new JustGage({
      id: "gauge",
      value: 0,
      min: 0,
      max: 5,
      label: "Volts",
      title: "Light-o-meter"
    });
    getReading();
  </script>
</body>
</html>
```

Preguntas de repaso: Servidor API REST

1.3.2.1.2 Cliente REST

Disponer de un servidor web en nuestra RP3 resulta muy cómodo: no se necesita recurrir a un servidor adicional en Internet. Tanto en estos ejercicios como en la página web alojada en la RP3, no hemos necesitado nada más que la Raspberry y nuestro navegador. Sin embargo, en la práctica es problemático disponer de un servidor web en la RP3, por 3 razones principales:

- Consumo de recursos: un servidor web, incluso ligero, consume muchos recursos. Si se espera que muchos clientes externos puedan acceder a este equipo, la RP3 se encontrará con dificultades para servir todas las peticiones.
- Un dispositivo conectado a Internet con un servicio abierto es un problema de seguridad. Los rastreadores de Internet estarán continuamente accediendo a él para ve lo que ofrece, y cualquier vulnerabilidad puede ser utilizada por un atacante para dejar sin servicio el sistema, o incluso tomar su control.
- Los equipos IoT son equipos externos a Internet, alojados tras routers con NAT. Ofrecer un servicio al exterior requiere tener acceso a ese NAT y gestionar adecuadamente la apertura y redirección de puertos, lo que no siempre es posible o práctico.

Al final, la solución habitual en los proyectos IoT es utilizar un servidor en Internet. Para grandes proyectos con miles de nodos, se recurre a backends cloud, capaces de ingerir grandes cantidades de datos de dispositivos a lo largo de todo el mundo. En estos casos, el servidor del API web estará en este backend, y nuestro dispositivo IoT realizará la función contraria, la de cliente del API.

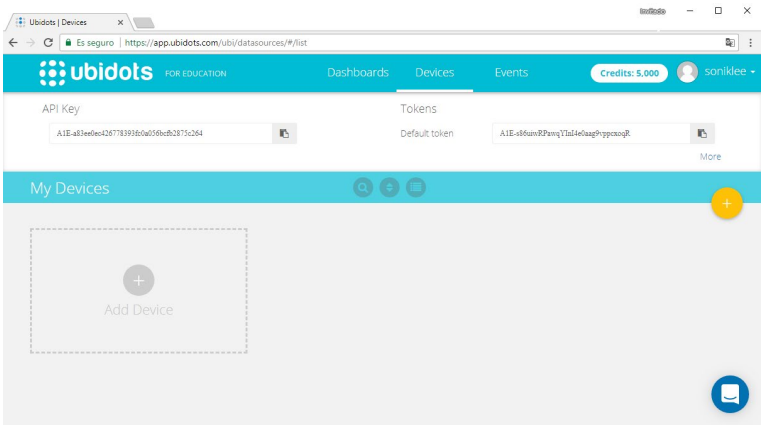
Ejercicio: *Interacción con Ubidots mediante API REST*

Ubidots es una plataforma muy potente e interesante para proyectos de Internet de las Cosas, que nos permite integrar datos de sensores, almacenarlos, mostrarlos en un dashboard, y generar distintos tipos de eventos. En este ejercicio aprenderemos a utilizarlo para nuestros proyectos IoT, y a implementar un cliente en Android Things para interactuar con su API REST. Fíjate que en este caso ya no es necesario disponer de un servidor web para ofrecer servicios REST, sino que estos servicios estarán alojados en el backend IoT de Ubidots, y por tanto deberás implementar un cliente HTTP para comunicarte con él.

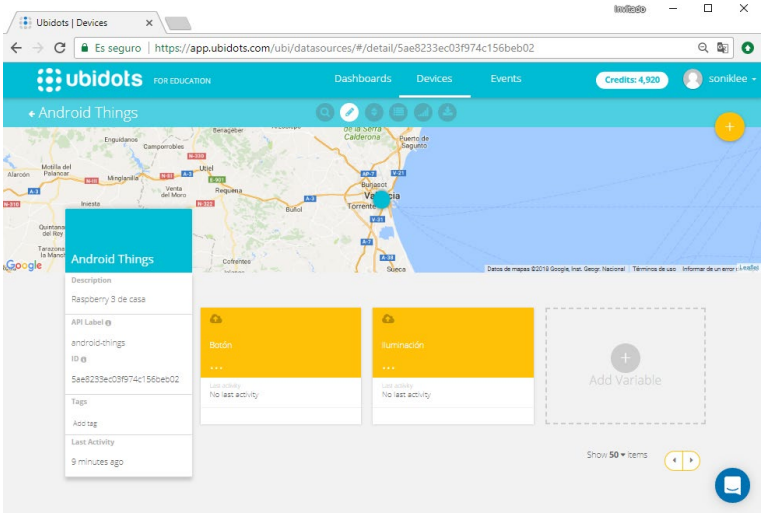
1. Entra en **Ubidots** y crea una cuenta de estudiante:

```
https://app.ubidots.com
```

2. Cuando estés dentro de tu cuenta, haz click en tu nombre de usuario (arriba derecha) y luego en *API credentials*. Verás que aparece tu *API key* (no la necesitas, se utiliza para generar tokens válidos) y un ***default token***. Este token no caduca, y permite autenticar a un cliente cuando utiliza el API de Ubidots para enviar datos a tu cuenta, así que será el que utilicemos. También es posible solicitar tokens al vuelo, mediante una consulta POST al API, que tienen una caducidad de 6h.



3. Dirígete a "*Devices*" y añade un nuevo dispositivo, de nombre "**Android Things**". Haz click en él para ver su contenido. Edita el campo "*Description*", y añade la ubicación del dispositivo.
4. Pulsa en *Add variable* > *Default* para añadir la sensorización que vamos a enviar desde nuestro dispositivo. Utilizaremos una variable "**Iluminación**" para el valor del fotorresistor, y una variable "**Botón**" para conocer si el botón está pulsado.



5. Accede a la variable "*Iluminación*" y edita sus propiedades. Utilizaremos como unidades "volts", y como rango permitido "0-5". Si tratamos de enviar valores fuera de este rango, el API nos devolverá un error, lo que nos aporta un método de controlar la validez de las lecturas desde la propia API. Haz lo mismo con la variable "*Botón*", sin unidades y con un rango "0-1". Fíjate en el campo **ID** de ambas variables, lo utilizaremos más adelante para el envío de los datos a través del API.
6. Crea un nuevo proyecto para Android Things.
7. Comenzamos con las dependencias. Utilizaremos **retrofit** para desarrollar el cliente HTTP (http://square.github.io/retrofit/), y **gson** para la serialización /

deserialización de los JSON (https://github.com/google/gson). Añade las siguientes dependencias en *build.gradle (Module: app)*:

```
dependencies {
    …
    implementation 'com.squareup.retrofit2:retrofit:2.3.0'
    implementation 'com.google.code.gson:gson:2.8.2'
    implementation 'com.squareup.retrofit2:converter-gson:2.3.0'
}
```

8. Añade los permisos necesarios en el manifiesto (*AndroidManifest.xml*). Utilizaremos el permiso **INTERNET** y **USE_PERIPHERAL_IO**:

```
<manifest …>
    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="com.google.android.things.permission
                                .USE_PERIPHERAL_IO" />
    <application …>
```

9. Comenzamos creando una clase `Data`, donde crearemos nuestro modelo de datos. Según la guía de referencia para el API de Ubidots, cuando queremos enviar los datos de varios sensores, debemos de utilizar el siguiente formato:

```
[{"variable": "{VAR_ID_1}", "value":41.2}, {"variable": "{VAR_ID_1}", "va-
lue":88.3}]
```

Vemos que tenemos un JSON con un array de parejas "variable" – "valor". Nuestra clase `Data` expondrá estos dos campos, el primero de tipo *String* y el segundo de tipo *Double*.

Crea la clase `Data` y añade el siguiente contenido:

```
public class Data {

  @SerializedName("variable")
  @Expose private String variable;

  @SerializedName("value")
  @Expose private Double value;

  public String getVariable() { return variable; }

  public void setVariable(String variable) { this.variable = variable; }

  public Double getValue() { return value; }

  public void setValue(Double value) { this.value = value; }
}
```

10. A continuación crea una interfaz de nombre `UbiAPI`, con el siguiente contenido:

```
import okhttp3.ResponseBody;
import retrofit2.Call;
import retrofit2.http.Body;
import retrofit2.http.POST;
import retrofit2.http.Query;

public interface UbiAPI {
  @POST("/api/v1.6/collections/values")
```

```
  public Call<ResponseBody> sendValue(
    @Body ArrayList<Data> dataList, @Query("token") String token);
}
```

Esta interfaz representa el API al que invocamos. La anotación `@POST` indica el método que se utilizará al llamar a la interfaz. Este método acepta como entrada un array de objetos `Data`, tal y como los hemos definido en el punto anterior; y un token de seguridad para conectar con el API. También se indica la dirección a la que apuntará el método POST, según lo especificado en la guía de referencia de Ubidots[2].

11. Ahora crearemos el cliente que manejará la comunicación con Ubidots. Crea una clase `UbiClient` y utiliza el siguiente contenido:

```
import retrofit2.Call;
import retrofit2.Callback;
import retrofit2.Response;
import retrofit2.Retrofit;
import retrofit2.converter.gson.GsonConverterFactory;

public class UbiClient {
  private static final String TAG = UbiClient.class.getSimpleName();
  private static final String UBI_BASE_URL = "http://things.ubidots.com/";
  private static UbiClient client;
  private UbiAPI api;
  private Retrofit retroClient;

  private UbiClient() {
    retroClient = new Retrofit.Builder()
      .baseUrl(UBI_BASE_URL)
      .addConverterFactory(GsonConverterFactory.create())
      .build();
  }

  public static final UbiClient getClient() {
    if (client != null) return client;
    client = new UbiClient();
    return client;
  }

  private UbiAPI getUbiClient() {
    return retroClient.create(UbiAPI.class);
  }

  public void sendData(ArrayList<Data> dList, String token) {
    api = client.getUbiClient();
    Call c = api.sendValue(dList, token);
    c.enqueue(new Callback() {
      @Override public void onResponse(Call call, Response response) {
        Log.d(TAG, "onResponse");
        Log.d(TAG, "Result: " + response.isSuccessful() + " - " +
```

[2] https://ubidots.com/docs/api/#send-values-to-many-variables

```
                response.message());
      }
      @Override public void onFailure(Call call, Throwable t) {
        t.printStackTrace();
      }
    });
  }
}
```

Aquí encontramos la URL base para la comunicación con Ubidots, la creación del cliente, y el método para el envío de datos: `sendData()`. Este método dispone de un *callback* que nos indica el resultado de la operación, de forma que podríamos realizar operaciones adicionales según la respuesta del API.

12. Ahora ya podemos desarrollar el código principal de nuestra `MainActivity`. Utiliza el siguiente contenido:

```
public class MainActivity extends Activity {
  // IDs Ubidots
  private final String token = "A1E-s86uiwRPawqYInI4e0aag9vppcxoqR";
  private final String idIluminacion = "5aeec55ec03f97502473e6de";
  private final String idBoton = "5aeec565c03f97516e8480e0";

  private final String PIN_BUTTON = "BCM23";
  private Gpio mButtonGpio;
  private Double buttonstatus = 0.0;
  private Handler handler = new Handler();
  private Runnable runnable = new UpdateRunner();

  @Override protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    PeripheralManager service = PeripheralManager.getInstance();
    try {
      mButtonGpio = service.openGpio(PIN_BUTTON);
      mButtonGpio.setDirection(Gpio.DIRECTION_IN);
      mButtonGpio.setActiveType(Gpio.ACTIVE_LOW);
      mButtonGpio.setEdgeTriggerType(Gpio.EDGE_FALLING);
      mButtonGpio.registerGpioCallback(mCallback);
    } catch (IOException e) {
      Log.e(TAG, "Error en PeripheralIO API", e);
    }
    handler.post(runnable);
  }

  @Override protected void onDestroy() {
    super.onDestroy();
    handler = null;
    runnable = null;
    if (mButtonGpio != null) {
      mButtonGpio.unregisterGpioCallback(mCallback);
      try {
        mButtonGpio.close();
      } catch (IOException e) {
        Log.e(TAG, "Error en PeripheralIO API", e);
      }
```

```
    }
  }

  // Callback para envío asíncrono de pulsación de botón
  private GpioCallback mCallback = new GpioCallback() {
    @Override public boolean onGpioEdge(Gpio gpio) {
      Log.i(TAG, "Botón pulsado!");
      if (buttonstatus == 0.0) buttonstatus = 1.0;
      else buttonstatus = 0.0;
      final Data boton = new Data();
      boton.setVariable(idBoton);
      boton.setValue(buttonstatus);
      ArrayList<Data> message = new ArrayList<Data>() {{add(boton);}};
      UbiClient.getClient().sendData(message, token);
      return true;  // Mantenemos el callback activo
    }
  };

  // Envío síncrono (5 segundos) del valor del fotorresistor
  private class UpdateRunner implements Runnable {
    @Override public void run() {
      readLDR();
      Log.i(TAG, "Ejecución de acción periódica");
      handler.postDelayed(this, 5000);
    }
  }
  private void readLDR() {
    Data iluminacion = new Data();
    ArrayList<Data> message = new ArrayList<Data>();
    Random rand = new Random();
    float valor = rand.nextFloat() * 5.0f;
    iluminacion.setVariable(idIluminacion);
    iluminacion.setValue((double) valor);
    message.add(iluminacion);
    UbiClient.getClient().sendData(message, token);
  }
}
```

Demos un vistazo a lo que contiene este código. Comenzamos definiendo algunos identificadores claves para la comunicación con el API de Ubidots. Utiliza como `token` tu *default token* (ver paso 2), y como `idIluminación` e `idBoton`, las IDs de las variables que has creado en tu cuenta de Ubidots.

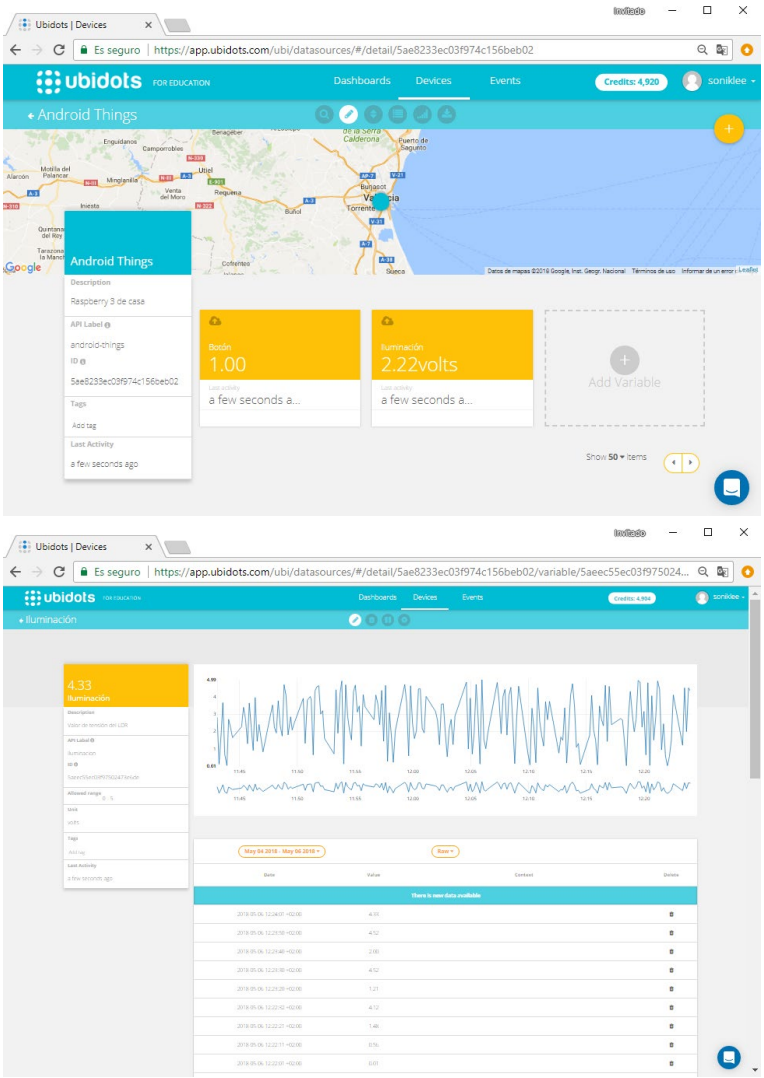
Para cada variable que queremos enviar, creamos un objeto `Data`, al que le añadimos el ID de variable y su valor actual. Como el API de Ubidots acepta un array de objetos `Data` como entrada, lo añadimos a un *ArrayList*, y luego lo enviamos con `sendData()`.

En envío de datos a Ubidots se realiza de dos formas:

- Pulsador: de forma asíncrona. Cuando presionamos el pulsador, se invoca al callback mCallback, donde construimos el objeto `Data` y llamamos a `sendData()`.

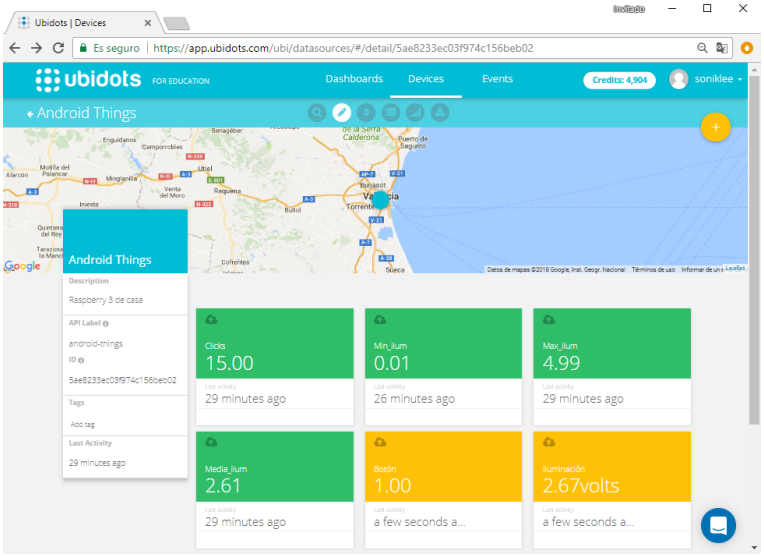
- Fotorresistor: de forma síncrona. Mediante el uso de las clases `Handler` y `Runnable`, generamos un evento cada 5 segundos, donde se lee el valor de fotorresistor, se genera el objeto `Data`, y se envía con `sendData()`.

13. Carga y prueba el proyecto. La vista de variables en Ubidots debería comenzar a recibir valores. Presiona el pulsador de la placa de prototipado para ver cómo se actualiza su lectura. Si accedes a las variables, puedes ver el histórico en formato tabla y gráfica.

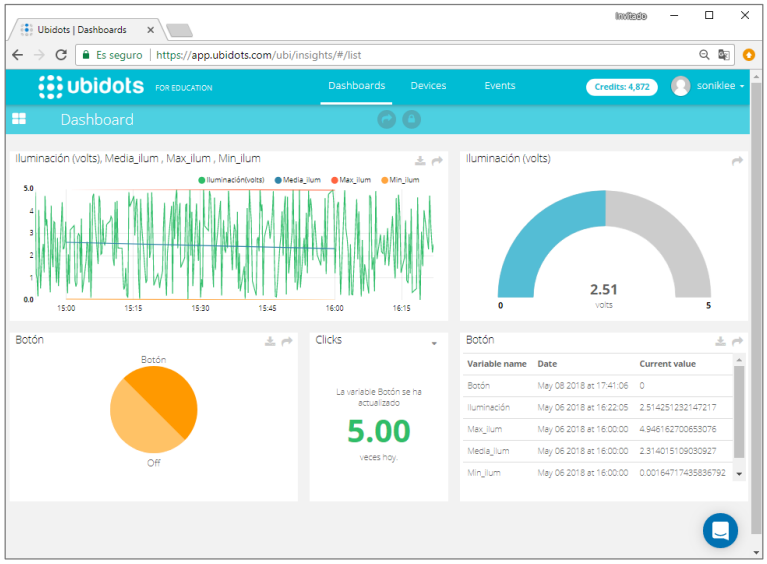


14. Una característica muy interesante de las plataformas IoT es la generación de variables virtuales, formadas a partir de operaciones sobre las variables físicas. Accede de nuevo al dispositivo "**Android Things**" en Ubidots y crea 4 nuevas variables de tipo "*Rolling window*". Crea una con la media horaria del valor de iluminación, otra con el máximo horario, y otra con el mínimo horario. Después,

crea una variable con la cuenta diaria de clicks del botón. Estas variables se muestran en color verde para diferenciarlas de las variables físicas (en amarillo).



15. Actualmente estamos utilizando Ubidots como un servicio de almacenamiento, donde se ingieren las medidas de nuestro dispositivo y se almacenan en forma de histórico. Ubidots nos ofrece además dos formas de explotar estos datos: ***dashboards*** (para visualización) y ***events*** (para acciones). Vamos a revisar la función de *dashboards*, intenta generar el siguiente panel:



NOTA: *Los dashboards son elementos muy importantes en aplicaciones de supervisión, mando y control; puesto que permiten conocer de un vistazo el estado de nuestra flota de dispositivos, o de los procesos que monitorizan. Es importante realizar un diseño inteligente de este tipo de paneles, ya que no siempre más información es mejor. En general se*

prefiere menor información pero de mayor calidad: por ejemplo en industria este tipo de indicadores suelen referirse como "key performance indicators" o KPIs.

Preguntas de repaso: Plataformas IoT en la nube

1.3.2.2. Modelo publish/subscribe: MQTT

MQTT (*Message Queuing Telemetry Transport*) es un protocolo de aplicación que utiliza un modelo de comunicación publish / subscribe, estandarizado como ISO/IEC PRF 20922. Existen 3 roles: publicador, suscriptor y bróker. Los publicadores son fuentes de información, que envían al bróker etiquetada con un "topic". Los suscriptores son consumidores de información. Indican al bróker su interés en uno o varios "topics". El bróker es el mediador entre ambas entidades: tiene un registro de los suscriptores y aquellos topics que les interesa, y al recibir el mensaje de un publicador, lo reenvía a todos los suscriptores que están interesados en ese topic. Un mismo elemento final puede actuar tanto como publicador de los datos de sus sensores, y como suscriptor a los datos de otros elementos.

MQTT es un protocolo agnóstico al contenido, lo que significa que provee al desarrollador de un canal de comunicaciones entre un publicador y varios suscriptores, y en ese canal puede enviar todo tipo de información (datos binarios, texto, XML, JSON… etc). No ofrece un mecanismo propio de seguridad, esto debe garantizarse en otros niveles de la comunicación. Por otro lado, es un sistema sin memoria, por lo que un elemento que se acabe de conectar no podrá recibir mensajes pasados, sólo el último mensaje recibido por el bróker, y los futuros mensajes.

MQTT presenta las siguientes operaciones:

- CONNECT: establecimiento de conexión con el bróker, indicando su IP y puerto. Esto requiere el establecimiento de un socket TCP, y su mantenimiento mediante *keepalives*.
- PUBLISH: mensaje enviado al bróker, que contiene un topic y una carga útil. El topic sirve para clasificar los mensajes, filtrarlos, facilitar la suscripción y en general poder diferenciarlos por intereses. La carga útil puede ser de cualquier tipo, son las aplicaciones de ambos extremos (publicador y suscriptor) las responsables de entender su contenido.
- SUBSCRIBE: mensaje enviado al bróker, indicando a qué topic deseamos suscribirnos para recibir sus mensajes.
- DESUBSCRIBE: mensaje enviado al bróker para eliminar nuestra suscripción de un topic, y dejar de recibir sus mensajes.
- DISCONNECT: cierre de la conexión con el bróker y liberación de los recursos de la sesión TCP/IP.

También ofrece un control de entrega de mensajes, al que se refiere como "nivel de QoS". Con esto, podemos indicar si queremos que la conexión bróker-cliente

asegure la entrega de mensajes o no. Esto permite realizar un diseño adaptado a las necesidades: mayor robustez para luchar contra nodos en redes con mayor tasa de pérdidas, menor robustez para mensajes menos prioritarios, redes más estables o con mayor cantidad de nodos, etc.

MQTT se considera el protocolo estrella de Internet de las Cosas por diversas razones:

- Es un estándar abierto que todo el mundo puede implementar, y da libertad a utilizar el contenido (payload) que se requiera para cada aplicación.
- Sus requisitos de recursos computacionales son muy bajos, por lo que la huella necesaria en los dispositivos finales es mínima.
- Su implementación y funcionamiento es muy sencilla
- El bróker no modifica los mensajes, solamente los reencamina, por lo que es capaz de manejar tráfico de grandes cantidades de dispositivos con un impacto bajo
- Presenta una latencia muy baja: tal y como se genera un mensaje, lo reciben los suscriptores
- Los niveles de QoS permiten a adaptarse a redes poco robustas y proponesas a la pérdida de datos, como las inalámbricas.

Ejercicio: *Conexión y publicación con MQTT*

En este ejercicio aprenderás cómo realizar la configuración y conexión con un bróker MQTT. Para ello utilizaremos un código muy sencillo que conectará y publicará un “Hello World!” en un topic, y que visualizaremos desde un cliente MQTT de escritorio.

Para este ejercicio no es necesario realizar ningún montaje hardware.

1. Crea un nuevo proyecto para Android Things.
2. Comenzamos añadiendo las dependencias. Para implementar MQTT podemos encontrar multitud de librerías Java, pero de entre todas ellas utilizaremos **Eclipse Paho**, que nos ofrece las siguientes ventajas: es un proyecto Eclipse estable, bien documentado y mantenido por una extensa comunidad, y dispone de soporte Android nativo (https://eclipse.org/paho/clients/android/).

   En primer lugar, debemos de añadir la ruta al repositorio en el *build.gradle (Project)*:

```
repositories {
    google()
    jcenter()
    mavenCentral()
}
```

A continuación, añade las siguientes dependencias en el *build.gradle (Module:app):*

```
dependencies {
    …
    implementation 'org.eclipse.paho:org.eclipse.paho.client.mqttv3:1.1.1'
}
```

3. Añade los permisos necesarios en el manifiesto (*AndroidManifest.xml*). Utilizaremos el permiso **INTERNET**:

```
<manifest …>
    <uses-permission android:name="android.permission.INTERNET" />
    <application …>
```

4. Este primer ejercicio utilizará una estructura muy sencilla, directamente en la `MainActivity`, para mostrar los pasos básicos de una publicación en MQTT. Utiliza el siguiente código:

```
public class MainActivity extends Activity {
  private static final String TAG = "Things";
  private static final String topic = "<user>/test";
  private static final String hello = "Hello world!";
  private static final int qos = 1;
  private static final String broker = "tcp://iot.eclipse.org:1883";
  private static final String clientId = "Test134568789";

  @Override protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    try {
      MqttClient client = new MqttClient(broker, clientId,
        new MemoryPersistence());
      Log.i(TAG, "Conectando al broker " + broker);
      client.connect();
      Log.i(TAG, "Conectado");
      Log.i(TAG, "Publicando mensaje: " + hello);
      MqttMessage message = new MqttMessage(hello.getBytes());
      message.setQos(qos);
      client.publish(topic, message);
      Log.i(TAG, "Mensaje publicado");
      client.disconnect();
      Log.i(TAG, "Desconectado");
    } catch (MqttException e) {
      Log.e(TAG, "Error en MQTT.", e);
    }
  }
}
```

En este código comenzamos declarando diversas variables para el funcionamiento de MQTT:

- `topic`: cadena completa del topic donde publicaremos el mensaje. Pon tu nombre de usuario UPV en el campo ***<user>***, para que tus mensajes no se crucen con los de tus compañeros.
- `hello`: mensaje de "*Hello world!*" que enviamos al conectar con el bróker.

- `qos`: nivel de calidad de servicio que queremos para nuestras comunicaciones MQTT con el bróker. El nivel 1 es un compromiso entre garantía de entrega y bajo intercambio de mensajes de control.
- `broker`: dirección del servidor y puerto de escucha donde se encuentra el bróker MQTT al que conectamos. Utilizamos un bróker público de Eclipse, disponible a los usuarios para el testeo de aplicaciones, y que no requiere de autenticación. En caso de que no estuviera disponible, el bróker Mosquitto[3] también dispone de un bróker abierto de pruebas. En esta dirección[4] se mantiene una lista actualizada de brókers abiertos.
- `clientId`: identificador único de cliente en el bróker al que conectamos. Modifícalo aleatoriamente para que no coincida con ningún otro usuario, por ejemplo cambiando los números del final.

El resto de código se encuentra en el método `onCreate()`. En primer lugar se crea un cliente MQTT de tipo `MqttClient`, indicando la dirección del bróker, el clientId, y un buffer de memoria para tratar los mensajes. Con la llamada al método `connect()` solicitamos una conexión con este bróker.

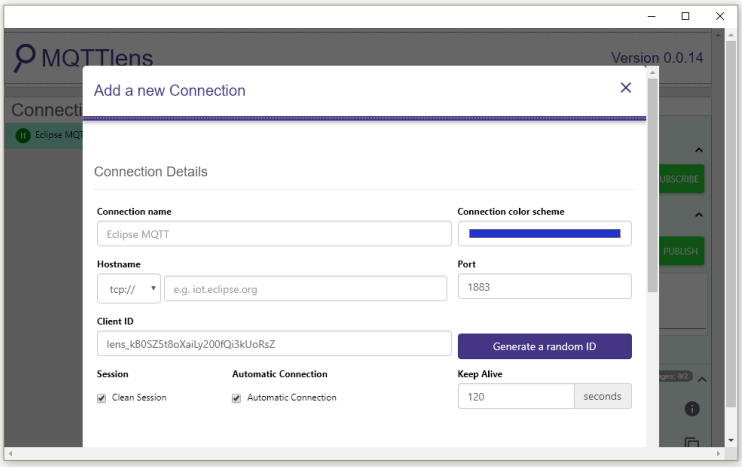
A continuación creamos un mensaje MQTT con un objeto `MqttMessage`. Este objeto, además del mensaje, contiene otros campos modificables como la QoS del mensaje.

Para enviar el mensaje, utilizamos el método `publish()`. Si hubiera algún problema, como por ejemplo que no hayamos podido conectar con el bróker, nos saltaría una excepción que recogería el `catch()`. Si todo es correcto, cerramos nuestra conexión con el bróker, liberando los recursos de la sesión.

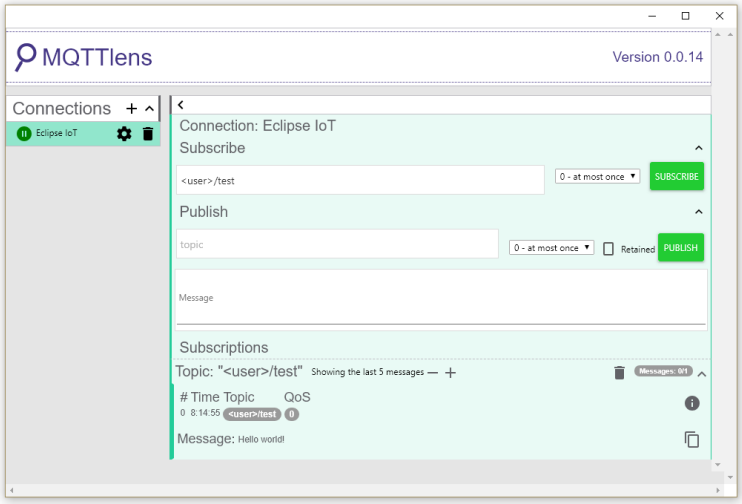
5. Para comprobar el mensaje que la Raspberry envía al bróker MQTT, necesitamos conectarnos a este bróker con un cliente y suscribirnos al tópic donde va a publicar. Podemos encontrar multitud de clientes para móvil (MyMQTT para Android y MQTTInspector para IOS), escritorio (MQTTLens como extensión de Chrome, MQTT.fx para Win/MacOSX/Linux, y MQTT-spy en Java multiplataforma) o incluso online (HiveMQ y Mitsuruog). A la hora de configurar estos clientes, hay que tener en cuenta si la conexión al bróker debe hacerse de forma directa (MQTT a puerto 1883) o con websockets (generalmente los clientes online).
   En este ejercicio utilizaremos como ejemplo MQTTLens, instálalo y lánzalo.
6. Añade una nueva conexión con los siguientes datos (utiliza como Client ID cualquier valor aleatorio):

[3] https://test.mosquitto.org

[4] https://github.com/mqtt/mqtt.github.io/wiki/public_brokers



7. Una vez añadido, comprueba que estás correctamente conectado al bróker, lo que se indica con un icono verde al lado de la conexión.
8. Suscríbete al tópic *<user>/test*, utilizando como **<user>** tu usuario de UPV.
9. Carga y lanza el proyecto de Android Things. Comprueba que recibes el mensaje de "*Hello World!*" en *MQTTLens*. Si tienes algún problema, comprueba que los tópics en *MQTTLens* y en el proyecto de Android Things coinciden.



Fíjate en la información que nos indica *MQTTLens* del mensaje recibido. Tenemos el tópic, importante cuando nos suscribimos con un comodín y podemos recibir mensajes de distintos tópics; la marca temporal de recepción de mensaje, la QoS y el contenido del mensaje. Si pinchamos en el icono de información de la derecha, nos mostrará también si existen mensajes duplicados.

Práctica: *Publicación bajo demanda*

Modifica el ejercicio anterior para realizar una aplicación que publique un mensaje MQTT cada vez que se presione un pulsador conectado a la Raspberry Pi. En resumen, esta nueva aplicación debe:

- Publicar los mensajes en el topic *"<user>/boton*", donde *<user>* es tu usuario UPV.
- Al conectarse al bróker, publicar el siguiente mensaje de bienvenida: "*Hello world! Android Things conectada.*"
- Hacer el montaje hardware de un pulsador. Al presionarlo, publicar el siguiente mensaje: "*click!*"
- Realizar la desconexión del bróker en el método `onDestroy()`.

Ejercicio: *Suscripción para control remoto del LED*

Ahora que ya hemos comprobado lo sencillo que es publicar mensajes en MQTT, vamos a centrarnos en el proceso de suscripción. Para ello vamos a desarrollar un ejercicio en el que, publicando desde MQTTLens un mensaje de "ON" o "OFF", actuaremos sobre el LED de la placa. Además, ampliaremos un poco nuestro código base para que el programa de conexión a MQTT sea más completo.

1. Crea un nuevo proyecto para Android Things.
   Añade las dependencias para Eclipse Paho. Primero la ruta al repositorio en el *build.gradle (Project)*:

```
repositories {
    google()
    jcenter()
    mavenCentral()
}
```

   A continuación, añade las siguientes dependencias en el *build.gradle (Module:app):*

```
dependencies {
    …
    implementation 'org.eclipse.paho:org.eclipse.paho.client.mqttv3:1.1.1'
}
```

2. Añade los permisos necesarios en el manifiesto (*AndroidManifest.xml*). Utilizaremos los permisos **INTERNET** y **USE_PERIPHERAL_IO**:

```
<manifest …>
    <uses-permission android:name="android.permission.INTERNET" />
```

```
    <uses-permission android:name="com.google.android.things.permission
                                .USE_PERIPHERAL_IO" />
    <application …>
```

3. En la `MainActivity`, utiliza el siguiente código:

```
public class MainActivity extends Activity implements MqttCallback {
  private static final String TAG = "Things";
  private final String PIN_LED = "BCM18";
  public Gpio mLedGpio;
  private static final String topic_gestion = "<user>/gestion";
  private static final String topic_led = "<user>/led";
  static final String hello = "Hello world! Android Things conectada.";
  private static final int qos = 1;
  private static final String broker = "tcp://iot.eclipse.org:1883";
  MqttClient client;

  @Override protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    PeripheralManager service = PeripheralManager.getInstance();
    try {
      mLedGpio = service.openGpio(PIN_LED);
      mLedGpio.setDirection(Gpio.DIRECTION_OUT_INITIALLY_LOW);
    } catch (IOException e) {
      Log.e(TAG, "Error en el API PeripheralIO", e);
    }
    try {
      String clientId = MqttClient.generateClientId();
      client = new MqttClient(broker, clientId, new MemoryPersistence());
      client.setCallback(this);
      MqttConnectOptions connOpts = new MqttConnectOptions();
      connOpts.setCleanSession(true);
      connOpts.setKeepAliveInterval(60);
      connOpts.setWill(topic_gestion, "Android Things desconectada!"
            .getBytes(), qos, false);
      Log.i(TAG, "Conectando al broker " + broker);
      client.connect(connOpts);
      Log.i(TAG, "Conectado");
      Log.i(TAG, "Publicando mensaje: " + hello);
      MqttMessage message = new MqttMessage(hello.getBytes());
      message.setQos(qos);
      client.publish(topic_gestion, message);
      Log.i(TAG, "Mensaje publicado");
      client.subscribe(topic_led, qos);
      Log.i(TAG, "Suscrito a " + topic_led);
    } catch (MqttException e) {
      Log.e(TAG, "Error en MQTT.", e);
    }
  }

  @Override protected void onDestroy() {
    super.onDestroy();
    try {
      if (client != null && client.isConnected()) {
```

```
        client.disconnect();
      }
    } catch (MqttException e) {
      Log.e(TAG, "Error en MQTT.", e);
    }
    if (mLedGpio != null) {
      try {
        mLedGpio.close();
      } catch (IOException e) {
        Log.e(TAG, "Error en el API PeripheralIO", e);
      } finally {
        mLedGpio = null;
      }
    }
  }

  @Override public void connectionLost(Throwable cause) {
    Log.d(TAG, "Conexión perdida...");
  }

  @Override public void messageArrived(String topic, MqttMessage message)
       throws Exception {
    String payload = new String(message.getPayload());
    Log.d(TAG, payload);
    switch (payload) {
      case "ON":
        mLedGpio.setValue(true);
        Log.d(TAG, "LED ON!");
        break;
      case "OFF":
        mLedGpio.setValue(false);
        Log.d(TAG, "LED OFF!");
        break;
      default:
        Log.d(TAG, "Comando no soportado");
        break;
    }
  }

  @Override public void deliveryComplete(IMqttDeliveryToken token) {
    Log.d(TAG, "Entrega completa!");
  }
}
```

En resumen, en este código nos encontramos con lo siguiente:

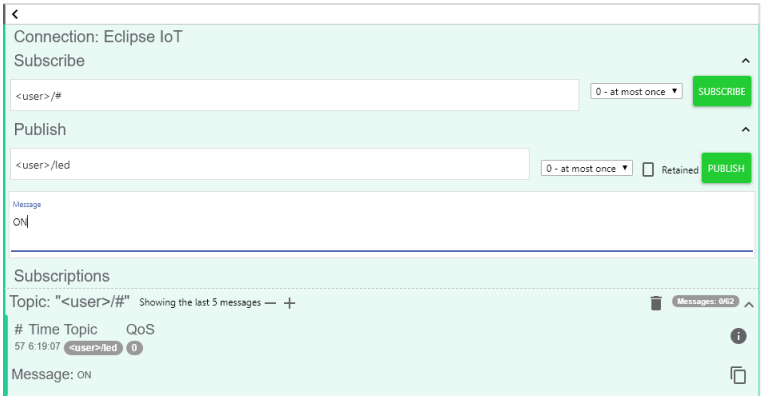
- Esta vez se indican 2 topics, uno para mensajes de gestión, y otro para los comandos del LED.
- Ya no especificamos un *clientID*. Para asegurarnos que sea aleatorio y distinto cada vez, utilizamos el método `generateClientId()`.

- Utilizamos por primera vez el objeto `MqttConnectOptions` para establecer opciones de nuestra conexión MQTT con el bróker. Indicaremos que queremos iniciar siempre una sesión limpia, que se envíe un keepalive cada minuto, y establecemos un mensaje de testamento. Estas opciones se pasan al realizar la llamada a `connect`.
- Nos suscribimos al topic de control de LED con el método `client.subscribe()`, indicando además el nivel de QoS que queremos mantener con el bróker para los mensajes recibidos.
- Se indica que nuestra clase implementa `MqttCallback`, para poder gestionar adecuadamente la recepción de mensajes MQTT a los topics que estemos suscritos.
- Con `client.setCallback(this)` indicamos que queremos establecer esta clase para la recepción de los callbacks.
- Al implementar la clase `MqttCallback`, añadimos tres métodos: `connectionLost()`, `messageArrived()` y `deliveryComplete()`. El primero se invoca cuando se pierde la conexión con el bróker, y podemos utilizarlo para lanzar alguna operación local como encender un LED de aviso, y para tratar de reconectar. Como hemos indicado que la sesión se inicia de forma limpia, deberemos volver a indicar a qué queremos suscribirnos. El segundo se invoca cuando la publicación de un mensaje se completa con éxito, lo que está ligado al nivel de QoS solicitado. Si hay QoS, este método no se invocará hasta que no se reciban los mensajes del bróker de confirmación de recepción. El tercer método es el que gestionará la recepción de un mensaje MQTT al que estamos suscritos.

4. Abre **MQTTLens**, conéctate al mismo bróker y suscríbete al siguiente tópic (donde ***<user>*** es tu nombre de usuario UPV):

```
<user>/#
```

5. Carga y prueba el proyecto. Cuando la Raspberry haya conectado con el bróker, deberás visualizar el mensaje de bienvenida. Fíjate que, al haberte suscrito al topic *<user>/#,* te llegarán todos los mensajes de los topics que cuelguen de esta raíz, lo que incluye los dos topics utilizados en el proyecto: *<user>/gestion* y *<user>/led*.
6. Prueba a activar y desactivar el LED. Para eso envía un mensaje con el texto ON ó OFF al topic *<user>/led*.



7. Cuando termines las pruebas, pulsa STOP en Android Studio para detener la ejecución del programa en la Raspberry. Comprueba que se recibe el mensaje de testamento.

Ejercicio: *Cliente web de MQTT mediante websockets*

El protocolo MQTT soporta tanto la conexión por un socket a un puerto estándar (1883), como la conexión mediante websockets. Los websockets son una herramienta fundamental para las aplicaciones web, pues permiten establecer sockets mediante los puertos 80 y 8080, de forma que pueden intercambiar datos a través de Internet sin sufrir los problemas asociados a cortafuegos y configuraciones NAT. Los brokers más habituales soportan la conexión de clientes tanto mediante puerto estándar como mediante websockets, tanto en sus versiones seguras como no seguras.

En este ejercicio aprenderás a generar un cliente MQTT web mediante websockets, en el que realizaremos la conexión, suscripción, visualización de mensajes entrantes, publicación de mensajes, y desconexión de bróker. Para ello utilizaremos la librería Paho de Eclipse para Javascript[5]. Esto te permitirá diseñar todo tipo de aplicaciones para interactuar con nodos MQTT, tanto para visualización como de control.

1. Abre el Notepad++ o cualquier otro editor de texto, y genera un nuevo fichero "**mqtt_webclient.html**".
2. Crea la siguiente estructura básica:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
```

[5] https://www.eclipse.org/paho/clients/js/

```
  <title>Cliente Websocket de MQTT</title>

</head>
<body align="center">


</body>
</html>
```

3. Comenzamos con el contenido de la cabecera. Inserta los siguientes scripts entre las etiquetas <head> … </head>:

```
<head>
 <script
  src="http://ajax.googleapis.com/ajax/libs/jquery/1.10.2/jquery.min.js"
  type="text/javascript"></script>
 <script
  src="https://cdnjs.cloudflare.com/ajax/libs/paho-mqtt/1.0.1/mqttws31.js"
  type="text/javascript"></script>
 <script type="text/javascript">
   var client = new Paho.MQTT.Client("iot.eclipse.org", Number(80), "/ws",
        "myclientid_" + parseInt(Math.random() * 100, 10));

    //Se invoca si la conexión MQTT se pierde
    client.onConnectionLost = function (responseObject) {
      alert("connection lost: " + responseObject.errorMessage);
    };

    //Se invoca cuando se recibe un mensaje de nuestras suscripciones
    client.onMessageArrived = function (message) {
      $('#messages').append('<span><small><b>Topic:</b> ' +
      message.destinationName + '  <b>| Mensaje:</b> ' +
      message.payloadString + '<small></span><br/>');
    };

    //Opciones de conexión
    var options = {
      timeout: 3,
      //Se invoca si la conexión ha sido satisfactoria
      onSuccess: function () {
        alert("Connected");
      },
      //Se invoca si no se ha podido establecer la conexión
      onFailure: function (message) {
        alert("Connection failed: " + message.errorMessage);
      }
    };

    //Publicación de mensajes
    var publish = function (payload, topic, qos) {
      var message = new Paho.MQTT.Message(payload);
      message.destinationName = topic;
      message.qos = qos;
      client.send(message);
    }
```

```
  </script>
</head>
```

En este código hay dos partes principales: la carga de las librerías de jquery y paho-mqtt, y el código del script en sí mismo.

En el código del script, comenzamos generando un cliente MQTT. Según la documentación del bróker de Eclipse, la conexión por websockets se realiza a la dirección:

```
iot.eclipse.org:80/ws
```

Esta dirección está formada por 3 partes: el host, el puerto, y el path. Para generar el cliente, el método utiliza la siguiente estructura, donde introduciremos un clientId aleatorio:

```
new Client(host, port, path, clientId)
```

A continuación tenemos un par de callbacks: `onConnectionLost` para realizar acciones si la conexión MQTT se pierde, y `onMessageArrived` para gestionar los mensajes recibidos sobre los topics a los que estamos suscritos. En ese caso, cuando recibimos un mensaje, lo mostramos por pantalla.

También se incluye un bloque para la gestión de la conexión, y otro para el método de publicación.

4. Vamos ahora con el contenido del cuerpo. Agrega el siguiente código entre las etiquetas <body> … </body>:

```
<body align="center">
 <h1>Cliente Websocket de MQTT<br></h1>
 <button onclick="client.connect(options);">1. Conectar</button>
 <button
   onclick="client.subscribe('<user>/#', {qos: 1}); alert('Subscribed');">
   2.   Suscribir</button>
 <button onclick="publish('Hello world! Soy el cliente websocket',
   '<user>/gestion',1);"> 3. Publicar</button>
 <button onclick="client.disconnect();">4. Desconectar</button><br><br>
 <div id="messages"></div>
</body>
```

Este código genera 4 botones en la página web: conectarse al bróker, suscribirse al topic “<user>/#” (modifica la etiqueta <user> por tu usuario UPV), publicar un mensaje de Hello en el topic “<user>/gestion” (modifica también esta etiqueta), y desconectar del bróker.

5. Guarda el fichero y ábrelo desde un navegador web.
6. Desde MQTTLens, suscríbete a <user>/# y publica mensajes en <user>/test. Deberías visualizarlos tanto en MQTTLens como en el cliente web.
7. Pulsa en “Publicar”. Deberías visualizar los mensajes tanto en el cliente web como en MQTTLens. Esto se debe a que ambos clientes están suscritos a los mismos topics, así que estén donde estén, recibirán estos mensajes.

Práctica: *Control de Raspberry mediante cliente web MQTT*

Haz el montaje de la Raspberry Pi con un LED externo y un pulsador. A partir de los dos últimos ejercicios, desarrolla un proyecto en el que podamos controlar desde una web, mediante el cliente MQTT websockets, el encendido y apagado del LED de la Raspberry. Utiliza también la interfaz web para indicar la acción sobre el pulsador.

Ejercicio: *Shake it! MQTT en el móvil*

En este ejercicio aprenderás a utilizar MQTT también en el móvil. Tras ello, ya podremos desarrollar clientes MQTT para móvil, Android Things y web, de forma que podremos comunicar aplicaciones entre distintas plataformas con un mínimo coste computacional y una elevada escalabilidad, ideal en proyectos de IoT.

Para este ejercicio vamos a desarrollar una aplicación móvil con botones de encendido y apagado del LED, y además, al sacudir el teléfono, el LED parpadeará 4 veces.

1. Crea un módulo para Android Things y otro para Móvil/Tablet.
   Añade las dependencias para Eclipse Paho. Primero la ruta al repositorio en el *build.gradle (Project)*:

```
repositories {
    google()
    jcenter()
    mavenCentral()
}
```

   A continuación, añade las siguientes dependencias en el *build.gradle* de ambos módulos *(Module:app) y (Module:mobile):*

```
dependencies {
    …
    implementation 'org.eclipse.paho:org.eclipse.paho.client.mqttv3:1.1.1'
}
```

2. Añade los permisos necesarios en el manifiesto de módulo para Android Things (*AndroidManifest.xml*). Utilizaremos los permisos **INTERNET** y **USE_PERIPHERAL_IO**:

```
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="com.google.android.things.permission
                                                    .USE_PERIPHERAL_IO" />
```

3. Ahora añade los permisos para el módulo del móvil:

```
<uses-permission android:name="android.permission.INTERNET" />
```

4. Comenzamos con la aplicación móvil. Utiliza el siguiente layout en *activity_main.xml*:

```
<android.support.constraint.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">
    <TextView
        android:id="@+id/textview"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent" />
    <Button
        android:id="@+id/buttonConnect"
        style="@style/Widget.AppCompat.Button.Colored"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Conectar"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintHorizontal_bias="0.198"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintVertical_bias="0.177" />
    <Button
        android:id="@+id/buttonDisconnect"
        style="@style/Widget.AppCompat.Button.Colored"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Desconectar"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintHorizontal_bias="0.851"
        app:layout_constraintLeft_toLeftOf="parent"
```

```
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintVertical_bias="0.177" />
    <Button
        android:id="@+id/buttonON"
        style="@style/Widget.AppCompat.Button.Colored"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="ON"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintHorizontal_bias="0.222"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintVertical_bias="0.777" />
    <Button
        android:id="@+id/buttonOFF"
        style="@style/Widget.AppCompat.Button.Colored"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="OFF"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintHorizontal_bias="0.831"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintVertical_bias="0.777" />
</android.support.constraint.ConstraintLayout>
```

5. Ahora crea una nueva clase `ShakeListener`, que implemente `SensorListener`, Esta clase será la responsable de detectar la sacudida del teléfono móvil e invocar al listener onShake:

```
public class ShakeListener implements SensorListener {
  private static final int FORCE_THRESHOLD = 350;
  private static final int TIME_THRESHOLD = 100;
  private static final int SHAKE_TIMEOUT = 500;
  private static final int SHAKE_DURATION = 1000;
  private static final int SHAKE_COUNT = 3;
  private SensorManager mSensorMgr;
  private float mLastX = -1.0f, mLastY = -1.0f, mLastZ = -1.0f;
  private long mLastTime;
  private OnShakeListener mShakeListener;
  private Context mContext;
  private int mShakeCount = 0;
  private long mLastShake;
  private long mLastForce;

  public interface OnShakeListener {
    public void onShake();
  }

  public ShakeListener(Context context) {
    mContext = context;
```

```
    resume();
  }

  public void setOnShakeListener(OnShakeListener listener) {
    mShakeListener = listener;
  }

  public void resume() {
    mSensorMgr = (SensorManager) mContext.getSystemService(Context
        .SENSOR_SERVICE);
    if (mSensorMgr == null) {
      throw new UnsupportedOperationException("Sensores no soportados");
    }
    boolean supported = mSensorMgr.registerListener(this,
      SensorManager.SENSOR_ACCELEROMETER,
      SensorManager.SENSOR_DELAY_GAME);
    if (!supported) {
      mSensorMgr.unregisterListener(this, SensorManager
          .SENSOR_ACCELEROMETER);
      throw new UnsupportedOperationException("Aceler. no soportado");
    }
  }

  public void pause() {
    if (mSensorMgr != null) {
      mSensorMgr.unregisterListener(this, SensorManager
          .SENSOR_ACCELEROMETER);
      mSensorMgr = null;
    }
  }

  public void onAccuracyChanged(int sensor, int accuracy) { }

  public void onSensorChanged(int sensor, float[] values) {
    if (sensor != SensorManager.SENSOR_ACCELEROMETER) return;
    long now = System.currentTimeMillis();
    if ((now - mLastForce) > SHAKE_TIMEOUT) {
      mShakeCount = 0;
    }
    if ((now - mLastTime) > TIME_THRESHOLD) {
      long diff = now - mLastTime;
      float speed = Math.abs(values[SensorManager.DATA_X] +
          values[SensorManager.DATA_Y] + values[SensorManager.DATA_Z] -
          mLastX - mLastY - mLastZ) / diff * 10000;
      if (speed > FORCE_THRESHOLD) {
        if ((++mShakeCount >= SHAKE_COUNT) && (now - mLastShake >
            SHAKE_DURATION)) {
          mLastShake = now;
          mShakeCount = 0;
          if (mShakeListener != null) {
            mShakeListener.onShake();
          }
        }
```

```
        mLastForce = now;
      }
      mLastTime = now;
      mLastX = values[SensorManager.DATA_X];
      mLastY = values[SensorManager.DATA_Y];
      mLastZ = values[SensorManager.DATA_Z];
    }
  }
}
```

6. En la clase `MainActivity`, utiliza el siguiente código, modificando los campos <user> por tu usuario UPV:

```
public class MainActivity extends AppCompatActivity implements
     MqttCallback, ShakeListener.OnShakeListener {
  private static final String TAG = "Mobile";
  private static final String topic_gestion = "<user>/gestion";
  private static final String topic_led = "<user>/led";
  static final String hello = "Hello world! Android Mobile conectado.";
  private static final int qos = 1;
  private static final String broker = "tcp://iot.eclipse.org:1883";
  MqttClient client;
  MqttConnectOptions connOpts;
  Button botonConnect, botonDisconnect, botonON, botonOFF;
  TextView textview;

  @Override protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    ShakeListener test = new ShakeListener(this);
    test.setOnShakeListener(this);
    textview = (TextView) findViewById(R.id.textview);
    botonConnect = (Button) findViewById(R.id.buttonConnect);
    botonDisconnect = (Button) findViewById(R.id.buttonDisconnect);
    botonON = (Button) findViewById(R.id.buttonON);
    botonOFF = (Button) findViewById(R.id.buttonOFF);

    try {
      String clientId = MqttClient.generateClientId();
      client = new MqttClient(broker, clientId, new MemoryPersistence());
      client.setCallback(this);
      connOpts = new MqttConnectOptions();
      connOpts.setCleanSession(true);
      connOpts.setKeepAliveInterval(60);
      connOpts.setWill(topic_gestion, ("Android Mobile " +
        "desconectado!").getBytes(), qos, false);
    } catch (MqttException e) {
      Log.e(TAG, "Error en MQTT.", e);
    }

    botonConnect.setOnClickListener(new View.OnClickListener() {
      public void onClick(View v) {
        Log.i(TAG, "Boton presionado");
        textview.setText("Conectando...");
        try {
```

```
            Log.i(TAG, "Conectando al broker " + broker);
            client.connect(connOpts);
            Log.i(TAG, "Conectado");
            Log.i(TAG, "Publicando mensaje: " + hello);
            MqttMessage message = new MqttMessage(hello.getBytes());
            message.setQos(qos);
            client.publish(topic_gestion, message);
            Log.i(TAG, "Mensaje publicado");
            textview.setText("Conectado");
          } catch (MqttException e) {
            Log.e(TAG, "Error en MQTT.", e);
            textview.setText("Error al conectar");
          }
        }
      });
      botonDisconnect.setOnClickListener(new View.OnClickListener() {
        public void onClick(View v) {
          Log.i(TAG, "Boton presionado");
          textview.setText("Desconectando...");
          try {
            if (client != null && client.isConnected()) {
              client.disconnect();
            }
            textview.setText("Desconectado");
          } catch (MqttException e) {
            Log.e(TAG, "Error en MQTT.", e);
            textview.setText("Error al desconectar");
          }

        }
      });
      botonON.setOnClickListener(new View.OnClickListener() {
        public void onClick(View v) {
          Log.i(TAG, "Boton presionado");

          try {
            String mensaje = "ON";
            Log.i(TAG, "Publicando mensaje: " + mensaje);
            MqttMessage message = new MqttMessage(mensaje.getBytes());
            message.setQos(qos);
            client.publish(topic_led, message);
            Log.i(TAG, "Mensaje publicado");
            textview.setText("Publicado ON");
          } catch (MqttException e) {
            Log.e(TAG, "Error en MQTT.", e);
            textview.setText("Error al publicar");
          }
        }
      });
      botonOFF.setOnClickListener(new View.OnClickListener() {
        public void onClick(View v) {
          Log.i(TAG, "Boton presionado");
          try {
```

```
            String mensaje = "OFF";
            Log.i(TAG, "Publicando mensaje: " + mensaje);
            MqttMessage message = new MqttMessage(mensaje.getBytes());
            message.setQos(qos);
            client.publish(topic_led, message);
            Log.i(TAG, "Mensaje publicado");
            textview.setText("Publicado OFF");
          } catch (MqttException e) {
            Log.e(TAG, "Error en MQTT.", e);
            textview.setText("Error al publicar");
          }
        }
      });
    }

    @Override protected void onDestroy() {
      super.onDestroy();
      try {
        if (client != null && client.isConnected()) {
          client.disconnect();
        }
      } catch (MqttException e) {
        Log.e(TAG, "Error en MQTT.", e);
      }
    }

    @Override public void connectionLost(Throwable cause) {
      Log.d(TAG, "Conexión perdida...");
    }

    @Override public void messageArrived(String topic, MqttMessage message)
      throws Exception { }

    @Override public void deliveryComplete(IMqttDeliveryToken token) {
      Log.d(TAG, "Entrega completa!");
    }

    @Override public void onShake() {
      Log.i(TAG, "Shake!");
      try {
        String mensaje = "Shake!";
        Log.i(TAG, "Publicando mensaje: " + mensaje);
        MqttMessage message = new MqttMessage(mensaje.getBytes());
        message.setQos(qos);
        client.publish(topic_led, message);
        Log.i(TAG, "Mensaje publicado");
        textview.setText("Publicado Shake!");
      } catch (MqttException e) {
        Log.e(TAG, "Error en MQTT.", e);
        textview.setText("Error al publicar");
      }
    }
}
```

En el método `onCreate()` creamos el listener para el movimiento del teléfono, con `new ShakeListener(this)` y `setOnShakeListener(this)`. Cuando se sacuda el teléfono, se invocará el método `onShake()`, donde publicamos el mensaje "Shake!". El resto de código es muy similar a lo que ya hemos visto hasta ahora: tenemos 4 botones para conectar y desconectar del bróker y para encender y apagar el LED, y ninguna suscripción porque no tenemos que realizar ninguna acción con la recepción de mensajes.

7. Vamos ahora con la aplicación para Android Things. Reutiliza el código del **Ejercicio:** *Suscripción para control remoto del LED*. Añade en el método `messageArrived()` la acción para el mensaje "*Shake!*":

```
case "Shake!":
  Log.d(TAG, "Parpadeo!");
  for (int i = 0; i < 4; i++) {
    mLedGpio.setValue(true);
    Thread.sleep(500);
    mLedGpio.setValue(false);
    Thread.sleep(500);
  }
  break;
```

8. Carga el código en el móvil y en la Raspberry y comprueba que funciona.

Preguntas de repaso: MQTT