UNIVERSIDAD POLITECNICA DE VALENCIA

ESCUELA POLITECNICA SUPERIOR DE GANDIA

Grado en Ing. Sist. de Telecom., Sonido e Imagen

"USO DE UNA APLICACIÓN ANDROID PARA LA MONITORIZACIÓN OTT DE LA CALIDAD DE SERVICIO DEL USUARIO MÓVIL"

TRABAJO FINAL DE GRADO

Autor/a:

Fermín Basso Della Vedova

Tutor/a:

Jose Francisco Monserrat del Río

GANDIA, Junio 2016

Resumen

El siguiente Trabajo Final de Grado tiene como objetivo desarrollar una APP Android para la monitorización OTT de la calidad de servicio (QoE) móvil del usuario y enviar dicha información a un servidor, que simulará al operador de red, para poder proporcionar a este, datos más concretos acerca del servicio que ofrece así como una mayor optimización de la red.

This Bachelor's degree thesis aims at developing an Android APP for OTT monitoring of the Quality of Service of mobile users and also sending this information to a server that will simulate the network operator, in order to provide specific data about the given service and also getting a better network optimization.

Contenido

1. Introducción

1.1 Motivación

Hoy en día disponemos de muchas herramientas que nos hacen conocer la calidad de servicio de red (QoS – *Quality of Service*) a la que estamos conectados. Por ejemplo, es fácil saber la velocidad a la que se descarga una película o a la que se está compartiendo un archivo, ya sea desde un ordenador, como desde un teléfono móvil o una tableta.

Sin embargo, desde el punto de vista del operador de la red móvil, como Movistar o Vodafone, la información de que disponen está muy sesgada ya que sólo conoce indicadores de red agregados de las estaciones base, por lo que, aunque puede disponer de valores medios de tasa de usuario, esos valores distan mucho de aportar la información pertinente sobre las prestaciones realmente experimentadas por el usuario.

Por otro lado, el sistema Android que muchos dispositivos usan se está desarrollando cada vez más, y esto es, en parte, gracias a que las herramientas para crear contenido son de fácil acceso para todos los desarrolladores.

Es por esto que este proyecto se plantea el objetivo de crear una aplicación Android que sea capaz de monitorizar la calidad del servicio de la conexión celular a la que estamos conectados experimentada en nuestro terminal. La información disponible en el terminal se enviará a un servidor remoto, de forma que emulemos el caso de uso en que un operador acumule la información de sus clientes para así optimizar la red o adaptarla a las necesidades específicas de cada uno. Además, de manera colateral, el propio usuario celular sabrá si dispone de una buena conexión y, por consiguiente, de buena cobertura.

1.2 Objetivos

En este TFG se pretende identificar una alternativa de recolección y envío de información de calidad por parte de los usuarios. Se estudiará el tipo de información disponible, capacidad de configuración de medidas por parte de la red y sobrecarga de señalización. En concreto se intuye el interés de una fuente de datos que pudiera recopilarse desde aplicaciones software instaladas en los propios terminales (APPs) que servirían para cualquier estándar radio subyacente. En el trabajo se completará una prueba de concepto, evaluando complejidad, gestión de la base de datos, consumo de potencia y reducción de prestaciones, así como su interés de cara a la optimización de la red.

Planteado este contexto, los objetivos del TFG son los siguientes:

- Identificar una alternativa de recolección y envío de información de calidad por parte de los usuarios.
- Estudiar el tipo de información disponible en caso de utilizar una aplicación de terminal, capacidad de configuración de medidas por parte de la red y sobrecarga de señalización.
- Implementar una APP bajo las limitaciones encontradas.
- Diseñar e implementar la BBDD subyacente.
- Completar una evaluación-prueba de concepto que permita ver las prestaciones en términos de consumo de batería y validez de la información almacenada.

1.3 Metodología

Para alcanzar los objetivos marcados, Google ofrece una serie de herramientas que permiten dicho desarrollo como, por ejemplo, *plugins* para entornos de desarrollo, librerías específicas, programas completos, etc.

En nuestro caso, realizaremos el desarrollo en Android que permite programar de manera sencilla sobre terminales móviles.

Android es una plataforma de desarrollo libre y de código abierto desarrollada por Google que dispone de las siguientes singularidades:

- El núcleo del sistema está basado en un Linux al que han hecho ciertas modificaciones para que pueda ejecutarse en teléfonos y terminales móviles.
- Dispone de una gran cantidad de servicios disponibles, como GPS, lectores de código de barras, etc.
- Aplicación hecha de componentes: Cada aplicación es realmente un puzle de varias piezas que luego se coordinarán para que cumpla las funciones del programa. Y estas piezas podrán utilizarse en otros programas o, incluso, ser reemplazadas por otras.
- Multitud de información: Con cada versión del sistema operativo, Google pone a disposición de los programadores la información necesaria para el uso de las nuevas funcionalidades.
- Multimedia: Mejora de capacidad visual mejorada en cada versión, ya sea en gráficos y en sonido como añadiendo mayor soporte a formatos de vídeo y audio, y mejorando la reproducción incluso en dispositivos poco potentes.
- Seguridad: Se provee al desarrollador una serie de herramientas para evitar que se desconfíe de su aplicación, así como proteger datos personales que se puedan introducir. Además, mientras las aplicaciones se encuentran en ejecución, varias capas de seguridad certifican el aislamiento de los datos entre ellas, de modo que una aplicación no tiene acceso a la otra y en caso que se quede colgada solo afectará a esa aplicación.
- Gestión de ciclo de vida automático: Todo está pensado para dispositivos con poca capacidad de proceso, poca memoria y poca batería. En Android existen unos ciclos de vida para las aplicaciones, y la gestión de este ciclo es llevada a cabo desde el mismo sistema operativo. Esto quiere decir que ya se encargará el propio sistema en cerrar algunas aplicaciones que no se estén usando en ese momento facilitar recursos a aquellas otras que se acaben de abrir.
- Múltiple hardware: El rango de dispositivos sobre los que se ejecutan aplicaciones Android va desde los conocidos teléfonos hasta microondas o lavavajillas. Desde el principio se ha pensado para múltiples plataformas.
- Adaptación para nuevas utilidades: Siguiendo el punto anterior, Android ha saltado de los teléfonos móviles y en una carrera de innovación, se está aplicando a cualquier elemento que pueda tener electrónica, como relojes, pulseras para controlar pulsaciones, gafas e incluso automóviles.

Pues bien, podemos ver que Android es un sistema muy versátil que dispone de una gran variedad de características para desarrollar diferentes tipos de aplicaciones. Para el desarrollo de este TFG hemos hecho uso de las siguientes herramientas:

- Eclipse como entorno de desarrollo
- *Plugin* de Android para eclipse.
- Material Design para el aspecto gráfico.

Una vez implementado el software, se realizarán pruebas experimentales para evaluar las prestaciones reales del software desarrollado y comprobar el consumo de batería que esta APP puede acarrear desde el punto de vista del usuario. El objetivo es que sea una función operando en segundo plano y que, por tanto, no le cause ninguna molestia al cliente.

1.4 Estructura del TFG

Este TFG está estructurado de la siguiente manera: en el capítulo 2 podremos encontrar el desarrollo en Android de la aplicación; en el capítulo 3 explicaremos el diseño e implementación del servidor, la base de datos y el cliente; en el cuarto capítulo veremos el desarrollo gráfico; en el capítulo 5 realizaremos las pruebas de funcionamiento y, por último, en el capítulo 6 encontraremos las conclusiones y las posibles líneas futuras que plantea este proyecto.

2 Diseño de la aplicación

2.1 Desarrollo de la aplicación

Esta sección se centra en describir el diseño y la estructura interna de la aplicación desarrollada en este TFG.

La idea de la aplicación es que se vaya comprobando "*in background*" los parámetros de calidad de servicio de manera periódica. Es decir, cada periodo preestablecido, la aplicación calculará la latencia en la conexión y las tasas de subida y bajada. Dichos cálculos los llamaremos "capturas" y se irán almacenando en una base de datos en un servidor externo que simulará al operador telefónico.

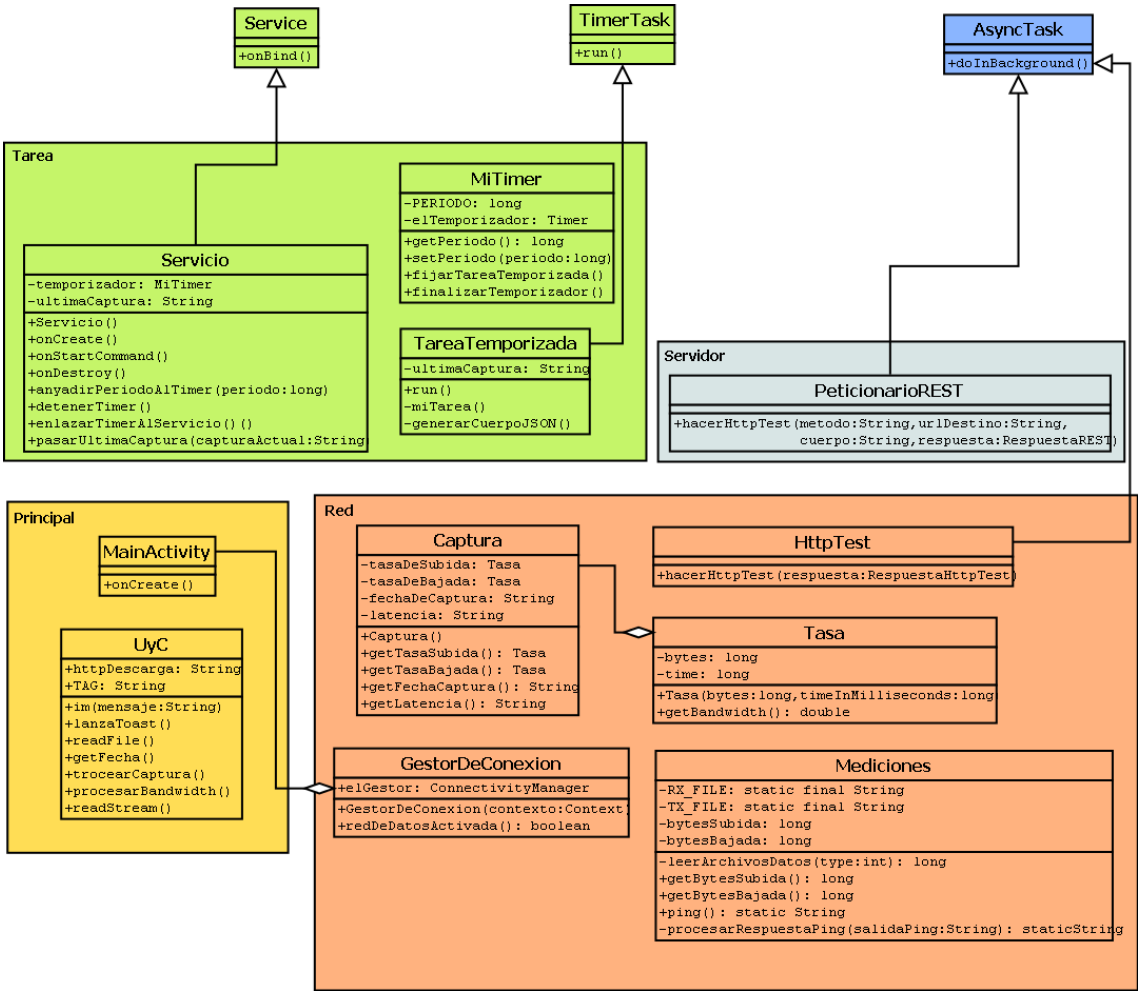

Figura 1: Esquema interno de la aplicación

En la *Figura 1* podemos ver que, a primera vista, la aplicación está formada de varios bloques, pero, en realidad, se basa en la relación que establece la Actividad principal, MainActivity (En el cuadro amarillo), con la clase Servicio (cuadro verde), que representa la acción que realizaremos en segundo plano.

En primer lugar, nos interesa averiguar cómo realizar servicios en hilos de ejecución diferentes al hilo principal donde corre la aplicación para no bloquearla y para que pueda seguir funcionando en segundo plano aunque se utilice el terminal para otros aspectos.

Al igual que ocurre con las actividades, los servicios también disponen de ciclos de vida, ya que estos pasan por diferentes estados y, en el caso de que se requiera de la liberación de recursos, también pueden ser destruidos por el propio sistema. A la hora de ejecutar un servicio diferenciamos entre dos escenarios:

- El servicio es lanzado y posteriormente detenido desde la actividad a través de los métodos `startService()` y `stopService()`.
- La actividad u otro componente se conecta con el servicio. Para ello se usará el método `bindService()`. Este caso se aproxima bastante al concepto de productores de servicios y consumidores de los mismos.

Si nos centramos en nuestro caso, nos interesa el primer escenario, ya que nosotros seremos quienes queramos tanto iniciarlo como detenerlo desde la actividad principal.

Construir un servicio implica extender la clase abstracta `android.app.Service` cuyos métodos más importantes que nos interesan son los siguientes:

- `onCreate()`: método que será invocado cuando se cree el servicio por primera vez.
- `onStartCommand()`: es el método invocado por el sistema cuando desde una actividad se utiliza el método `startService()`. Una vez que, a través de este método, el servicio sea arrancado, podrá seguir en funcionamiento indefinidamente si no se invoca el método `stopService()` (desde la actividad) o al `stopSelf()` (desde el propio servicio). Este método retornará un entero y, dependiendo de su valor, el sistema actuará de diferentes formas en caso de que el componente tenga que ser detenido por falta de recursos en el dispositivo. Los valores de retorno están recogidos en las siguientes constantes:
  - `START_STICKY`: el sistema tratará de volver a crear el servicio cuando disponga de memoria suficiente.
  - `START_NOT_STICKY`: el servicio se creará de nuevo cuando le llegue una nueva solicitud.
- `onDestroy()`: será invocado por el sistema cuando el servicio vaya a ser destruido.

2.2 Servicio y tareas periódicas

Al crear un servicio extendiendo la clase `Service`, existe la opción de sobrescribir sus diferentes métodos, aunque únicamente nos obliga a definir el método `onBind()`. Sin embargo, con solo extender la clase `Service` no podemos decir que tenemos el servicio. Al igual que en el caso de las actividades, los servicios también deben ser registrados en el fichero *AndroidManifest.xml* a través de la etiqueta `<service/>`.

```
<service android:name="com.example.tarea.Servicio"/>
```

Código 1: Etiqueta HTML que declara el Servicio

Una vez creado el servicio, podemos comenzar a utilizarlo desde una actividad, invocando a los métodos `startService` y `stopService`. Ya desde la actividad, si queremos invocarlo, debemos

```
Intent intencion = new Intent (this,Servicio.class);

startService(intencion);
```

Código 2: Declaración que lanza el Servicio.

usar la clase Intent, que se utiliza para ejecutar actividades o servicios.

Otro aspecto a tener en cuenta es el paso de parámetros al `Intent` a través del método `intención.putExtra("clave","valor")`, donde insertamos pares de valores que podrán ser leídos por el servicio. En nuestro caso, este método servirá para transmitirle parámetros como el periodo de medición de capturas que quiera seleccionar el usuario.

2.2.1 Tareas periódicas y la clase MiTimer.java

Como hemos dicho al principio, la aplicación debe realizar la tarea de medir las capturas de manera periódica. Es por esto que, existe una manera de configurar nuestro servicio para que se consiga dicha finalidad. Esta acción es atribuible a servicios basados en la clase `Timer` cuyo objetivo es crear procesos en hilos secundarios que se ejecutarán de forma periódica o programada.

Para empezar, debemos crear un objeto `Timer` y una variable tipo `long` que defina el periodo dentro de nuestro Servicio. Sin embargo, crearemos nuestra propia clase llamada `MiTimer` donde personalizaremos las características de la clase `Timer` a nuestro gusto.

Así pues, la clase `MiTimer` tendrá el siguiente aspecto:

```
public class MiTimer {

    private long PERIODO;
    private Timer elTemporizador;

    public MiTimer (){
        this.PERIODO = 0;
        this.elTemporizador = new Timer();
    }

    public long getPeriodo (){
        return this.PERIODO;
    }

    public void setPeriodo (long periodo){
        this.PERIODO = UyC.periodoEnMiliseg(periodo);
    }

    public void fijarTareaTemporizada (){
        UyC.im("Llegamos hasta fijarTareaTemporizada()");
         this.elTemporizador.scheduleAtFixedRate(new TareaTemporizada(), 0, this.PERIODO);
    }

    public void finalizarTemporizador (){
        elTemporizador.cancel();
    }
}
```

Código 3: Clase MiTimer

Como podemos ver, definimos dos objetos privados que serán un objeto del tipo `Timer`, que vendrá a ser nuestro temporizador, y una variable tipo `long` que definirá el periodo de la tarea. En el constructor inicializamos el periodo y el temporizador para no obtener errores de punteros que no apuntan a nada en posteriores simulaciones.

Añadimos métodos del tipo `set` y `get` para añadir y extraer el periodo, y el método `finalizarTemporizador()`, que nos servirá para detener el `Timer` y, por tanto, la tarea.

Finalmente, es muy importante explicar el método `fijarTareaTemporizada()` porque es el que se encargará de programar la tarea en cuestión de acuerdo al periodo que hayamos establecido. Esto se consigue a través del método `scheduleAtFixedRate()` de la clase `Timer`, cuyos argumentos son: la tarea que queramos realizar, la fecha a la que debe empezar y el periodo.

Luego analizaremos el objeto que representará a nuestra tarea, que en *Código 3* se ve como `new TareaTemporizada()`. Antes de eso, explicaremos brevemente los métodos que funcionarán de intermediarios entre nuestro servicio y nuestro temporizador.

2.2.2 Clase Servicio.java

Para que el servicio funcione de manera periódica se debe declarar un objeto `MiTimer`:

```
public class Servicio extends Service{

    @Override
    public IBinder onBind(Intent arg0) {
        // TODO Auto-generated method stub
        return null;
    }

    private MiTimer temporizador;
    private static String ultimaCaptura;

    public Servicio(){
    }

    @Override
    public void onCreate(){
        super.onCreate();
        //Creamos un nuevo timer para cada hilo y así poder añadirle el periodo
        temporizador = new MiTimer();
    }

    @Override
    public int onStartCommand (Intent intencion, int flags, int startId){
        UyC.im("onStartCommand()");
        long periodo = intencion.getLongExtra("periodo", 1);
          UyC.im("El Servicio ha recibido el periodo " +   Long.toString(periodo));
        anyadirPeriodoAlTimer (periodo); //Añadimos el periodo al temporizador
        enlazarTimerAlServicio(); //Enlazamos la tarea que queremos hacer
        return START_STICKY;
    }

    @Override
    public void onDestroy (){
        detenerTimer();
    }
    // Métodos vinculados al timer
    public void anyadirPeriodoAlTimer (long periodo){
        temporizador.setPeriodo(periodo);
    }

    public void detenerTimer (){
        temporizador.finalizarTemporizador();
    }

    private void enlazarTimerAlServicio (){
        temporizador.fijarTareaTemporizada();
    }

    //Método para que se guarde la última captura
    public void pasarUltimaCaptura (String capturaActual){
        ultimaCaptura = capturaActual;
        Intent enviarCaptura = new Intent ("captura-hecha");
        enviarCaptura.putExtra("captura", ultimaCaptura);
        LocalBroadcastManager.getInstance(this).sendBroadcast(enviarCaptura);
    }
}
```

Código 4: Clase Servicio

En la mitad inferior podemos ver los métodos vinculados al `Timer` excepto el último (`pasarUltimaCaptura()`), que servirá para recibir las capturas que irá midiendo la tarea. Pues bien, los métodos servirán para recibir los datos de la intención procedente de `MainActivity` y se los pasarán a nuestro objeto privado temporizador.

El método `onStartCommand()` recibe la intención de la clase `MainActivity` de la que extraeremos todos los parámetros que le hayamos querido pasar. En este caso, recogemos el periodo de la intención y se la añadimos al `Timer` con `anyadirPeriodoAlTimer()`. Finalmente, ejecutamos el método `enlazarTimerAlServicio()` que se encargará de ejecutar el método

`fijarTareaTemporizada()` de la clase `MiTimer` y que, por lo tanto, hará que arranque la tarea.

Más adelante explicaremos el método `pasarUltimaCaptura()`.

2.2.3 Clase TareaTemporizada.java

Continuamos ahora con la explicación de la clase `TareaTemporizada` que será en la que definamos nuestra tarea a realizar de manera periódica.

```
public void fijarTareaTemporizada (){
        UyC.im("Llegamos hasta fijarTareaTemporizada()");
        this.elTemporizador.scheduleAtFixedRate(new TareaTemporizada(), 0, this.PERIODO);
}
```

Código 5: Método fijarTareaTemporizada, clase MiTimer

Al método `scheduleAtFixedRate` de la clase `Timer`, se le debe pasar un objeto del tipo `TimerTask`, que se trata de una interfaz que implementa el método `run()` de la clase `Runnable`. Todo lo que se incluya en el método `run()` tendrá un hilo de ejecución propio (Al heredar de la clase `Runnable`) y será lo que se ejecute de manera periódica, de manera que, conviene crear nuestra propia clase que extienda de `TimerTask` e implementar en ésta todo lo que necesitemos para un desarrollo más sencillo.

Así pues la clase `TareaTemporizada` se plantearía de la siguiente manera:

```
public class TareaTemporizada extends TimerTask{

    @Override
    public void run() {
        // TODO Auto-generated method stub
        miTarea();
    }

    private void miTarea(){

        //Aquí escribiremos el algoritmo para medir capturas...

    }

}
```

Código 6: Clase TareaTemporizada

Como leemos en la imagen, en el método privado `miTarea()` escribiremos el algoritmo que medirá las capturas.

2.3 Cálculo de tasas y latencia

Vamos a pasar pues a explicar todo el procedimiento que realizaremos para medir las capturas.

Recordemos que los parámetros que tenemos que medir son:

- Latencia de la conexión.
- Tasa de subida.
- Tasa de bajada.

En primer lugar, explicaremos cómo medir la Latencia.

2.3.1 Latencia

Para medir la latencia realizaremos un *ping* convencional y extraeremos de este la latencia. Los ping en Android se pueden hacer mediante la inserción de un comando en un objeto `Runtime`, que vendría a ser la consola interna del sistema Android.

El método que realizará el *ping* lo escribiremos en una clase de métodos estáticos que contendrá funciones específicas de red.

La clase se llamará `Mediciones.java`:

```
public class Mediciones {

    public static String ping ()throws IOException, InterruptedException{

        Log.d("TFG", "Entramos en el m鷗do ping()");

        String comando = "/system/bin/ping -c 1 8.8.8.8"; //comando a insertar

        Process procesoPing = Runtime.getRuntime().exec(comando); //Ejecuto el comando

        int result = procesoPing.waitFor();
        Log.d("TFG", "result waitFor(): "+String.valueOf(result));

        InputStream in = procesoPing.getInputStream(); //Lanza excepción IOException
        Log.d("TFG", "obtenemos InputStream");

        String respuesta = UyC.readStream(in);//Leemos la respuesta, lanza IOException.
        Log.d("TFG", "Leemos la respuesta");

        procesoPing.destroy();//Terminamos el proceso
        Log.d("TFG", "Proceso terminado");

        String latencia = procesarRespuestaPing(respuesta);

        return latencia;

    }
}
```

Código 7: Método ping de la clase Mediciones

Como podemos ver, este método ejecutará el comando *ping*:

La IP a la que realizamos el *ping* es *8.8.8.8* que corresponde a un servidor de Google especial para comprobar nuestra conexión a Internet y, que por lo tanto, nos proporcionará datos reales de nuestra conexión.

El argumento "`-c 1`" hace que envíe un solo paquete. De no incluir este argumento, la aplicación se quedaría en bucle realizando *pings* infinitos y, por tanto, bloquearía el hilo principal de la aplicación.

Creamos el proceso en el que se ejecutará el comando y recogemos su respuesta con el método `getInputStream` que guardamos en la variable "`in`" del tipo `InputStream`. Esta respuesta hay que convertirla a texto para poder extraer la latencia posteriormente. Para convertir el `InputStream` en texto lo hacemos mediante un método llamado `readStream()` que explicaremos posteriormente.

A continuación, destruimos el proceso para no bloquear ningún hilo de ejecución y extraemos la latencia mediante un método privado llamado `procesarRespuestaPing()` para, finalmente, devolver la latencia.

Expliquemos el método `procesarRespuestaPing()` ya que es el que nos devuelve en sí la latencia:

```
String comando = "/system/bin/ping -c 1 8.8.8.8"; //comando a insertar

Process procesoPing = Runtime.getRuntime().exec(comando); //Ejecuto el comando
```

Código 8: Comando ping

Si analizamos la respuesta en sí del comando *ping* y, tras convertirla a texto con el comando `readStream()`, podemos ver lo siguiente:

```
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=52 time=52.7 ms
--- 8.8.8.8 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 52.740/52.740/52.740/0.000 ms
```

Figura 2: Respuesta PING en Android

Como podemos ver, nos interesa sacar de ese texto el valor de "`time= X ms`". Aquí tenemos el método `procesarRespuestaPing`, que irá escrito también en la clase `Mediciones`.

```
private static String procesarRespuestaPing (String salidaPing){

    String [] lineas = salidaPing.split("\n"); //Separar cada línea de la respuesta

    String tiempo = "";

    for(String linea : lineas){
        if(!linea.contains("time=")) continue;
            //Encontrar el índice de time=
        int indice = linea.indexOf("time=");

        tiempo = linea.substring(indice);
    }
    return tiempo;
}
```

Código 9: Método procesarRespuestaPing, clase Mediciones

Lo que hace este método es "recortar" la parte que nos dice el tiempo de latencia del texto que devuelve el comando ping. Primero separa el texto por líneas con el método `split` de la clase `String`, y después, en el bucle `for`, buscamos el texto "`time=`" y extraemos todo lo que tenga a continuación en esa línea.

Pasemos ahora a explicar la medición de las tasas de subida y bajada.

2.3.2 Tasas

En Android se puede acceder a la cantidad de bytes que ha enviado y recibido el terminal en todo momento. Esto se puede realizar de dos maneras diferentes:

- Mediante la clase `TrafficStats` que devuelve los bytes enviados y recibidos por el terminal, ya sean los de la red móvil, como los totales (contando WiFi y red de datos).
- Accediendo al archivo *rmnet0* de los archivos internos de android en el que se van almacenando los bytes enviados y recibidos de la red móvil desde el arranque del teléfono.

La principal diferencia entre las dos maneras es que la clase `TrafficStats` devuelve "0" si la red WiFi está conectada mientras que leyendo el archivo *rmnet0* obtendremos la cantidad de bytes que se hayan enviado hasta el momento con la red de datos. Si la red WiFi está activada, este número no variará.

Nosotros haremos la segunda manera, ya que nuestra aplicación está pensada para que funcione de fondo mientras se le da un uso normal al terminal.

Para leer el archivo *rmnet0* implementaremos un método dentro de la clase `Mediciones.java`

```
private static final String RX_FILE = "/sys/class/net/rmnet0/statistics/rx_bytes";
    // Archivo del sistema donde se van acumulando los bytes usados con datos en RECEPCIÓN

    private static final String TX_FILE = "/sys/class/net/rmnet0/statistics/tx_bytes";
    // Archivo del sistema donde se van acumulando los bytes usados con datos en TRANSMISIÓN

    private static long bytesSubida;  //bytes de transmisión
    private static long bytesBajada;  //bytes de recepción

    private static long leerArchivosDatos (int type){ //0, tx --- 1, rx.
        if (type == 0){
            bytesSubida = UyC.readFile(TX_FILE); //Bytes acumulados de subida
            return bytesSubida;
        } else if(type == 1 ){
            bytesBajada = UyC.readFile(RX_FILE); //Bytes acumulados de bajada
            return bytesBajada;
        }
        return 0;
    }

    public static long getBytesSubida (){//Leo los datos,
         return leerArchivosDatos(0);
    }

    public static long getBytesBajada (){
        return leerArchivosDatos(1);
    }
```

Código 10: Variables y métodos de la clase Mediciones

En *Código 10* podemos ver las variables y métodos relacionados con el cálculo del *throughput*. En primer lugar declaramos dos variables tipo `String` que contendrán las rutas de los dos archivos a leer. Como hemos dicho, éstos se encuentran en la carpeta *rmnet0*, que contiene los diferentes archivos y subcarpetas con datos relacionados a la red móvil.

Vemos, también, que hemos declarado dos variables tipo `long` que almacenarán tanto los bytes de subida como los de bajada leídos de los archivos anteriores.

Para leer los archivos implementamos el método `leerArchivoDatos()` al que le introduciremos un `int` para especificar cuál de los dos archivos queremos que lea. Si es 0, leerá los bytes de subida, y si es 1, los de bajada. El método `readFile` lo explicaremos más adelante cuando expliquemos la clase `UyC.java`, pero lo que hace principalmente es acceder al directorio intro-

ducido y extraer su contenido. Una vez leídos los archivos, almacenamos los resultados en las variables `bytesSubida` y `bytesBajada`.

Utilizaremos los métodos `getBytesSubida()` y `getBytesBajada()` para extraer los datos de las variables tipo `long` anteriormente descritas.

Una vez que tenemos una herramienta para leer los bytes en el instante que deseemos, necesitamos realizar cualquier acción que produzca alguna variación del número de bytes almacenados en estos archivos para poder medir la diferencia. Es decir: medir la cantidad de bytes al principio, realizar alguna acción que consuma datos, medir la cantidad de bytes cuando se haya finalizado la acción y, finalmente, calcular la diferencia de bytes para saber cuántos se han empleado en esa acción.

Hay un detalle en este proceso que cabe tener en cuenta y es que, en ese intervalo de tiempo de simulación en el que estemos haciendo el test de conexión, el terminal puede que consuma datos por necesidad de otra aplicación, y eso se verá reflejado en el número que calculemos. No solo aparecerá en ese número los que hayamos consumido nosotros sino también los que hayan usado otras aplicaciones en ese mismo instante.

Esto se debe a que el método no distingue entre aplicación y aplicación ya que almacena los datos TOTALES consumidos por la red de datos. No obstante, esto no influye en el resultado final.

Ahora hay que elegir la acción que consuma bytes de datos móviles. Es importante que se pueda separar bien tanto la conexión de subida como la de bajada para una medición más exacta.

En nuestro caso hemos elegido una conexión HTTP a un sitio web ya que es fácil controlar por programa la parte que corresponde a solicitud (subida) y la parte que corresponde a respuesta (bajada) de manera independiente.

La idea, por tanto, es la siguiente: cada vez que nuestro terminal deba medir una captura del estado de servicio, consultará la cantidad de bytes iniciales, se conectará a la página web, calculará la latencia de la conexión, consultará la cantidad de bytes finales y calculará la diferencia para, finalmente, dividirlos por el tiempo empleado en este proceso y así obtener las tasas de subida y bajada.

Es un proceso bastante simple por lo que no consumirá muchos recursos del sistema y esto resulta ideal para aplicaciones pensadas para funcionar en segundo plano.

Sin embargo, se nos plantea el siguiente problema. Todas las conexiones a un servidor web que se necesiten hacer desde una aplicación Android se deben realizar desde un hilo de ejecución diferente del que se esté trabajando en ese momento. Esto se debe a que Android está constantemente destruyendo procesos si tiene problemas de recursos, concretamente, si los hilos de ejecución se quedan bloqueados más de 5 segundos, son automáticamente destruidos. Por lo que, si la conexión falla y está mucho tiempo intentando realizar la conexión web, cabe la posibilidad de que nuestra aplicación se destruya.

Para solucionar este problema, haremos uso de la clase `AsyncTask<>` que nos permite realizar una tarea asíncrona, en segundo plano y en un hilo de ejecución secundario y diferente del hilo de ejecución principal. Además, no hay que preocuparse de finalizar dicho hilo secundario porque será automáticamente cerrado una vez la tarea que corría por él haya acabado.

`AsyncTask<Params,Progess,Result>` funciona de la siguiente manera: Se debe extender de la clase `AsyncTask` donde nos encontraremos con cuatro métodos principales:

- `onPreExecute()`: método utilizado para los trabajos de configuración previos a la tarea.
- `doInBackground(Params…)`: se ejecutará después del método `onPreExecute()`. Será usado para procesar tareas asíncronas que generalmente tardan en ejecutarse. Puede utilizar el método `publishProgress(Progress…)` para publicar el progreso de la tarea.
- `onProgressUpdate(Progress…)`: método que es invocado después de la llamada a `publishProgress(Progress…)`. Recibe información del progreso de la tarea.
- `onPostExecute(Result)`: método que es invocado al terminar el proceso en segundo plano, es decir, cuando el método `doInBackground` ha finailizado.

Extender la clase `AsyncTask` también implica crear tres parámetros genéricos:

- `Params`: parámetros que recibirá la ejecución de la tarea.
- `Progress`: tipo de dato utilizado para mostrar el progreso de la tarea.
- `Result`: dato que retornará una vez terminada la tarea.

Así pues, así se presenta la clase que ejecutará el test HTTP:

```
public class HttpTest extends AsyncTask <Void, Void, Boolean> {
```

Código 11: Encabezado clase HttpTest

`Void`, `Void`, `Boolean` son los parámetros genéricos que necesita la clase `AsyncTask <Params, Progress, Result>`. De esta manera, si seguimos el orden podemos ver que los parámetros de entrada serán del tipo `Void`, o sea, no habrá parámetros de entrada. Los parámetros de progreso también serán de del tipo `Void` y, por último, los resultados serán tipo `Boolean`, para saber si ha funcionado el test o no.

Aquí tenemos la clase `HttpTest`:

```
public class HttpTest extends AsyncTask <Void, Void, Boolean> {

    public interface RespuestaHttpTest{

        public void callback (long bytesSubida, long bytesBajada,
        long tiempoSubida, long tiempoBajada, String latencia);
    }

    private long bytesSubidaInicio;
    private long tiempoSubidaInicio;
    private long bytesSubidaFinal;
    private long tiempoSubidaFinal;
    private long bytesBajadaInicio;
    private long tiempoBajadaInicio;
    private long bytesBajadaFinal;
    private long tiempoBajadaFinal;

    private RespuestaHttpTest laRespuesta;
```

Código 12: Esqueleto clase HttpTest

Como vemos en el *Código 12*, se han incluido dos métodos necesarios cuando se extiende de `AsyncTask` y, además, hemos añadido el método `hacerHttpTest(RespuestaHttpTest)`, que será el que usaremos desde "fuera" para iniciar la tarea, y un constructor.

También vemos que hemos implementado una interfaz llamada `RespuestaHttpTest` con un método llamado `callback`. Esto nos servirá para poder procesar las respuestas del test desde la clase de la que hayamos invocado el método `hacerHttpTest()`.

Es decir, cuando se llame a hacer `hacerHttpTest(RespuestaHttpTest)` se hará de la siguiente manera:

```
new HttpTest().hacerHttpRest(new HttpTest.RespuestaHttpTest(){

    @Override
    public void callback (long bytesSubida, long bytesBajada, long tiempoSubida,
    long tiempoBajada, String latencia){

        //Hacer lo que queramos con los resultados del test en este método

    }

});
```

Código 13: Ejemplo de llamada del método hacerHttpTest

A continuación veremos que el método `onPostExecute` llamará al método `callback` de la interfaz `RespuestaHttpTest`, por lo tanto, lo que estamos haciendo en esta llamada es escribir qué hará el método `callback` una vez sea llamado.

Analicemos ahora el procedimiento del test HTTP y las mediciones de bytes y tiempo realizadas.

```
private long bytesSubidaInicio;
private long tiempoSubidaInicio;
private long bytesSubidaFinal;
private long tiempoSubidaFinal;
private long bytesBajadaInicio;
private long tiempoBajadaInicio;
private long bytesBajadaFinal;
private long tiempoBajadaFinal;
private RespuestaHttpTest laRespuesta;
private long bytesSubida;
private long bytesBajada;
private long tiempoSubida;
private long tiempoBajada;
private String latencia;
```

Código 14: Variables privadas de la clase HttpTest

Como vemos en el *Código 14*, tenemos declaradas un gran número de variables privadas que nos servirán para realizar las mediciones correspondientes. El propio nombre de las variables indica lo que van a ir almacenando durante el proceso.

Vamos a explicar el proceso `doInBackground` que será en el que haremos los cálculos.

```
@Override
    protected Boolean doInBackground(Void... arg0) {
        try{
            this.tiempoSubidaInicio = System.currentTimeMillis();
            this.bytesSubidaInicio = Mediciones.getBytesSubida();
            this.tiempoBajadaInicio = System.currentTimeMillis();
            this.bytesBajadaInicio = Mediciones.getBytesBajada();

            URL url = new URL (UyC.httpDescarga);
            //Creamos una simple petición HTTP
            HttpURLConnection connection = (HttpURLConnection) url.openConnection();
            this.latencia = Mediciones.ping();//saco la latencia
            connection.setRequestMethod("GET");

            connection.setDoInput(true);

            int rc = connection.getResponseCode();

            String rm = connection.getResponseMessage();
            String respuesta = "Codigo: " + rc + " Respuesta: " + rm;
            UyC.im(respuesta);

            connection.disconnect();

            this.bytesSubidaFinal = Mediciones.getBytesSubida();
            this.tiempoSubidaFinal = System.currentTimeMillis();

            this.bytesBajadaFinal = Mediciones.getBytesBajada();
            this.tiempoBajadaFinal = System.currentTimeMillis();

            this.bytesSubida = bytesSubidaFinal - bytesSubidaInicio;
            //Bytes empleados en realizar la solicitud HTTP
            this.tiempoSubida = tiempoSubidaFinal - tiempoSubidaInicio;
            //Tiempo tardado para realizar la solicitud HTTP

            this.bytesBajada = this.bytesBajadaFinal - this.bytesBajadaInicio;
            this.tiempoBajada = this.tiempoBajadaFinal - this.tiempoBajadaInicio;

            UyC.im("Bytes usados en la subida: " + Long.toString(bytesSubida));
            UyC.im("Tiempo empleado en la subida " + Long.toString(tiempoSubida));
            UyC.im("Bytes usados en la bajada: " + Long.toString(bytesBajada));
            UyC.im("Tiempo empleado en la bajada " + Long.toString(tiempoBajada));

            return true;

        }catch(MalformedURLException ex){
            UyC.im("Ocurrió un error con la URL: " + ex.getMessage() );
        }catch(IOException es){
            UyC.im("Error al establecer la conexión" + es.getMessage());
        }catch(InterruptedException ie){
            UyC.im("Acción interrumpida: " + ie.getMessage());
        }
        return false;
    }
```

Código 15: Método doInBackground de la clase HttpTest

Inicialmente, capturamos el número de bytes de subida y bajada almacenados en ese momento. Con el método `currentTimeMillis` de la clase `System`, capturamos la hora actual expresada en milisegundos.

Realizamos la conexión HTTP mediante las clases `URL` y `HttpURLConnection`, y calculamos la latencia con el método `ping` de la clase `Mediciones`. Después de esto, volvemos a medir los bytes y los tiempos, y calculamos la diferencia *Final* menos *Inical* de todos los aspectos. Finalmente, llamamos al método `callback` de la interfaz `RespuestaHttpTest` en el `onPostExecute` y le pasamos las diferencias. Recordemos que se implementará el contenido del método `callback` en el momento en que se llama a `hacerHttpTest`. Esto lo hacemos para procesar los resultados del test de manera separa al proceso del cálculo.

```
protected void onPostExecute (Boolean comoFue){
        UyC.im("onPostExecute() comoFue = " + comoFue);
        this.laRespuesta.callback(this.bytesSubida,this.bytesBajada,
        this.tiempoSubida,this.tiempoBajada,this.latencia);
    }
```

Código 16: Método onPostExecute de la clase HttpTest

Por último, falta explicar cómo arrancamos la tarea. Hemos dicho que se hará a través del método `hacerHttpTest`, que será el que sea llamado desde "fuera" para iniciar el cálculo. No obstante, se debe relacionar esto con cómo arrancar las tareas creadas a través de la clase `AsyncTask<>`. Pues bien, basta con llamar al método `execute` de dicha clase:

```
public void hacerHttpTest (RespuestaHttpTest respuesta){
    this.laRespuesta = respuesta;
    this.execute();

    try{
       this.get();
       //Método para esperar a que termine el hacerHttpTest()
       UyC.im("Hacemos el get() de AsyncTask");

    }catch(Exception ex){
        UyC.im("Ha habido algn problema con el get() de AsyncTask");
    }

}
```

Código 17: Método hacerHttpTest

El método `hacerHttpTest()` hace un poco la función de constructor y, a su vez, hace que empiece el cálculo. En primer lugar, apuntamos la respuesta introducida en el argumento a la variable privada `laRespuesta`. En segundo lugar, arrancamos la tarea con el método `execute()`. Finalmente, ejecutamos el método `get()`. El método `get()` hace que no se ejecute todo lo que venga después hasta que no haya terminado toda la tarea. Es decir, cuando se ejecuta el `execute()` se empieza a realizar la tarea. Pues bien, hasta que ésta no se haya terminado (llamado a `onPostExecute()`), no se ejecutará lo que venga a continuación del `get()`. Esto se hace por si necesitamos que se termine la tarea al completo antes de continuar el programa. Recordemos que `AsyncTask` ejecuta una tarea en segundo plano de manera asíncrona al hilo principal, por lo que si funciones del hilo principal dependen de resultados de la tarea asíncrona y ésta no ha terminado en el momento que se ejecutan estas funciones, puede producirse un error.

Pasemos a explicar las clases `Captura` y la clase `Tasa`.

2.3.3 Clases Captura.java y Tasa.java

Hemos visto el algoritmo de cálculo del *throughput* y cómo hacer la llamada al método que nos lo calculará. Ahora viene el dónde haremos la llamada. Para ello, presentamos las dos siguientes clases: `Captura.java` y `Tasa.java`.

La clase `Tasa` está constituida de la siguiente manera:

```
public class Tasa {

    private long bytes; //Cantidad de bytes
    private long time; //Tiempo en milisegundos!!! Lo paso a segundos en el metodo
getBandwidth()

    public Tasa (long bytes, long timeInMilliseconds){
        this.bytes = bytes;
        time = timeInMilliseconds;
    }

    public Tasa (){
        bytes = 0;
        time = 0;
    }

    public double getBandwidth (){

        Long b = Long.valueOf(bytes); //Creamos un objeto Long a partir de long
        Long t = Long.valueOf(time);

        double a = b.doubleValue(); //Convertimos el long en double
        double c = t.doubleValue();

        double bandwidthEnBps = (a*8)/(c/1000);  // a*8 es para pasar los bytes a
               //bits y c/1000 es para pasar el tiempo a segundos

        return bandwidthEnBps;
    }

}
```

Código 18: Clase Tasa

En primer lugar, declaramos dos variables privadas. Una será la cantidad de bytes empleados en el enlace y la otra el tiempo utilizado.

Escribimos dos constructores, uno sin argumentos por lo que inicializamos las variables a 0 para no tener errores de puntero, y otro en el que incluimos los argumentos `bytes` y `timeInMilliseconds` para apuntar a las variables privadas. En todo momento se está trabajando con la cantidad de bytes en bytes, y con el tiempo en milisegundos. Ya en el método `getBandwidth` de esta misma clase, lo convertiremos a bits por segundo (*bps*).

Después de los constructores tenemos el método `getBandwidth()` que calcula la tasa en sí. Divide los bytes empleados entre el tiempo que ha tardado en emplearlos y lo convierte a *bps*. Devuelve una variable tipo `double` porque es probable que el resultado contenga decimales que se deban considerar.

La clase `Captura`, por su parte, se constituye así:

```
public class Captura {

    private Tasa tasaDeSubida;
    private Tasa tasaDeBajada;
    private String fechaDeCaptura;
    private String latenciaCapturada;
```

```
    public Captura (){

        UyC.im("Llegamos hasta el constructor de Caputra");

        new HttpTest().hacerHttpTest(new HttpTest.RespuestaHttpTest() {

            @Override
            public void callback(long bytesSubida, long bytesBajada,
            long tiempoSubida, long tiempoBajada,String latencia) {

                tasaDeSubida = new Tasa (bytesSubida,tiempoSubida);
                tasaDeBajada = new Tasa (bytesBajada,tiempoBajada);
                fechaDeCaptura = UyC.getFecha();
                latenciaCapturada = latencia;

            }
        });
        UyC.im("Termina el constructor de Captura");
    }

    public Tasa getTasaSubida (){
        return tasaDeSubida;
    }

    public Tasa getTasaBajada (){
        return tasaDeBajada;
    }

    public String getFechaCaptura (){
        return fechaDeCaptura;
    }

    public String getLatencia(){
        return latenciaCapturada;
    }
}
```

Código 19: Clase Captura

La clase `Captura` es una representación de las capturas de calidad de servicio. Es por esto que encontramos dos variables que se corresponden con las tasas de subida y bajada, otra variable que corresponde con la latencia de la conexión y otra que corresponde a la fecha en la que se ha medido la captura.

Pasemos a explicar el constructor que es la parte más importante. Como hemos adelantado antes, estábamos explicando dónde se debe declarar el método `hacerHttpTest` para realizar las mediciones. Pues bien, el constructor de la case `Captura` es donde realizaremos el checkeo ya que la idea es que cada vez que se haga `new Captura()` se haga una nueva medición.

El constructor de esta clase, pues, es el que realiza la medición HTTP. En el `callback` declaramos objetos con nuestras variables privadas a partir de lo que devuelve el test HTTP. Es decir, creamos las tasas de subida y bajada a partir de los bytes y tiempos medidos, guardamos la fecha de la captura y, por último, la latencia.

Es muy importante que se comience con "`new HttpTest().hacerHttpTest(…)`", por 2 razones importantes: la primera es que cada "`new`" declara un objeto nuevo y, por lo tanto, una captura nueva y separada de la anterior. La segunda es que con cada "`new`" crearemos un hilo de ejecución diferente.

Finalmente, escribimos un método "`get`" para cada variable privada para poder extraerlas por separado.

Hemos visto como hemos diseñado e implementado el algoritmo y las clases pertinentes para calcular las capturas de calidad de servicio. A continuación vamos a explicar dónde iremos guardando las capturas.

3 Diseño del servidor y la base de datos

Vamos a pasar a explicar cómo hemos implementado el servidor y la base de datos. Como hemos adelantado, este servidor servirá para emular las capturas de información por parte del operador de red.

3.1 Base de datos

En primer lugar explicaremos la base de datos:

La base de datos se ha realizado con el lenguaje de programación *SQLite*, a través de la aplicación *sqlite3*. *Sqlite3* es una herramienta que se ejecuta por la consola del sistema Windows (en nuestro caso) y que permite crear, modificar, consultar y eliminar bases de datos además de otras funciones. Se trata de una pequeña biblioteca y basta con ejecutar un *.exe* para empezar a utilizarla.

Lo primero que hay que hacer es crear un archivo *.sql* que contendrá las tablas que constituirán nuestra base de datos:

```
-- sqlite3 Capturas Operador.db
-- .read crearBiblioteca.sql
-- .tables
-- .schema
-- .quit

-- borro tabla capturas
--drop table capturas;

-- creo una tabla nueva
create table capturas (
    fecha varchar(20) not null,
    tx int(4) not null,
    rx int(4) not null,
    primary key (fecha)
);

--inserto algunos valores predeterminados

insert into capturas values ('fecha ejemplo 1',1,1);
insert into capturas values ('fecha ejemplo 2',2,2);
insert into capturas values ('fecha ejemplo 3',3,3);
```

Código 20: Archivo que contiene las tablas que almacenarán la base de datos

Como vemos, hemos creado una tabla llamada "`capturas`" que, a su vez, incluye los 4 campos que forman una captura. En verde aparece como texto comentado, los pasos a realizar una vez creado el archivo *.sql*.

Los campos que forman la tabla "`capturas`" deben ser variables del mismo tipo que generemos en el terminal. Es decir, "`fecha`" deberá ser un `String` (`varchar` en *sql*), las tasas serán de tipo `int` y la latencia otro `String`. El número entre paréntesis indica la longitud de las variables.

Tras crear el archivo, e instalar la biblioteca *sqlite3*, creamos una base de datos mediante el comando en la línea de comandos:

```
sqlite3 nombre_de_la_base_de_datos.db
```

Código 21: Comando para crear la base de datos

Una vez hecho esto, entramos en el entorno de trabajo de *sqlite3*:

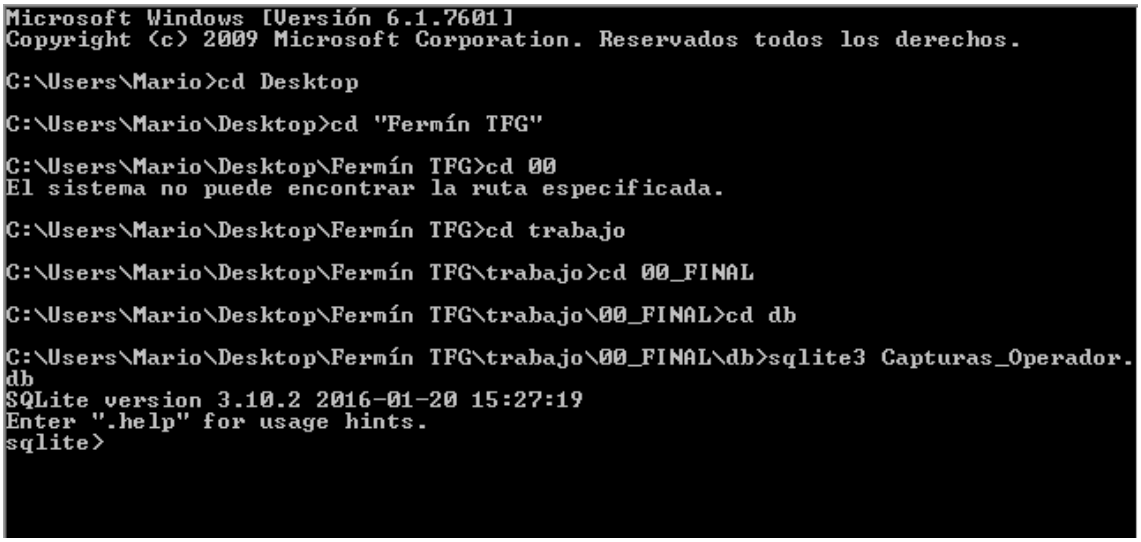

Figura 3: Entrada al entorno de desarrollo de Sqlite3

Una vez dentro del entorno, debemos hacer que la base de datos creada lea el archivo que incluye las tablas que hemos escrito previamente:

```
.read crear_capturasOrdenador.sql
```

Código 22: Comando sql para abrir el archivo que contiene las tablas

Es imprescindible que todo lo que se refiere a la base de datos esté en el mismo directorio.

Pues bien, a partir de aquí nuestra base de datos ya estaría creada.

3.2 Servidor

Ahora vamos a explicar cómo hemos hecho el servidor que contendrá a la base de datos Previamente, hemos comentado que el servidor estará escrito en *javascript* y se conectará con el terminal a través del lenguaje *REST* de HTTP.

Pues bien, en primer lugar debemos escribir un archivo en *javascript* que sirva de "nexo" entre la base de datos y el servidor. Este archivo se llamará "*logica.js*" y la conexión será a través de funciones que utilizará el servidor para modificar la tabla, consultarla o eliminarla. Así se plantea el archivo logica.js:

```
// ------------------------------------------------------------------------
// logica.js
// ------------------------------------------------------------------------

// importar la bilbioteca para acceder a bases de datos SQLite
var sqlite3 = require('sqlite3');

//conectar con la base de datos
var miBaseDeDatos = new sqlite3.Database("../db/Capturas_Operador.db");

// ------------------------------------------------------------------------
// ------------------------------------------------------------------------
// funcion "privada" de este módulo (fichero)
function hazLog (msg){
    console.log("mensaje: > " + msg + " <");
}//()

// ------------------------------------------------------------------------
// ------------------------------------------------------------------------
// funcion que exportamos (puede ser utilizada desde fuera)

exports.consultaDePrueba = function (){
}//()

// ------------------------------------------------------------------------
// ------------------------------------------------------------------------
//

exports.modificacionDePrueba = function(){
}//()
// ------------------------------------------------------------------------
// ------------------------------------------------------------------------
exports.buscarTXPorFecha = function(fecha, callback){
}//()
// ------------------------------------------------------------------------
// ------------------------------------------------------------------------
exports.buscarRXPorFecha = function(fecha, callback){
}//()
// ------------------------------------------------------------------------
// ------------------------------------------------------------------------
exports.buscarTXyRXPorFecha = function(fecha, callback){
}//()
// ------------------------------------------------------------------------
// ------------------------------------------------------------------------
exports.insertarTasa = function(datosTasa, callback){
}//()

// ------------------------------------------------------------------------
// ------------------------------------------------------------------------

exports.borrarTasa = function(fecha, callback) {
}//()
```

Código 23: Estructura de Logica.js

En este archivo encontramos dos variables (`sqlite3` y `miBaseDeDatos`) que sirven para importar los archivos con los que hemos trabajado antes: la biblioteica *sqlite3* y la base de datos. Además, encontramos las funciones que podemos implementar desde el servidor, pero será la función `insertarTasa` en la que nos centraremos a explicar porque será la que ejecutaremos principalmente.

```
exports.insertarTasa = function(datosTasa, callback){

    console.log(datosTasa.fecha);
    console.log(datosTasa.tx);
    console.log(datosTasa.rx);

    var sql = "insert into capturas values ('" + datosTasa.fecha + "',"+
                                            datosTasa.tx + "," + datosTasa.rx + ");";
    console.log("sql = " + sql);
    miBaseDeDatos.run(sql,callback);
}//()
```

Código 24: Método insertarTasa

No vamos a entrar en detalle de *javascript*, así que explicaremos el algoritmo de esta función. Lo que hace es formar un comando *sql*, lo ejecuta en la base de datos y recoge la respuesta de ella, mediante la función `callback`. Es un algoritmo muy parecido al utilizado por nosotros en `HttpTest` y la interfaz `RespuestaHttpTest`, ya que aquí "`callback`" será una función que escribiremos desde el archivo del servidor para procesar la respuesta. Finalmente, el método `console.log("mensaje")` sirve para ir mostrando por consola tanto las variables como los comandos insertados en la base de datos.

Vamos a explicar ahora el archivo que ejecutará las funciones del archivo `logica.js` y que convertirá estas ejecuciones en lenguaje *REST* para poder conectarse con el cliente Android.

Aquí tenemos pues, el archivo `servidor.js`

```
// ------------------------------------------------------
// servidor.js (seguro)
// ver: http://expressjs.com/4x/api.html
// ------------------------------------------------------

// .....................................................
// requires

var fs = require('fs');
var express = require('express');

var laLogica = require('../logica/logica.js');

var servidorExpress = express();

// ------------------------------------------------------
// ------------------------------------------------------
// servidorExpress.use (bodyParser());
// Esto es para copiar lo que haya en el cuerpo
// de la peticion HTTP al campo "body"  de "req"
// y que se pueda consultar luego.
//

servidorExpress.use (function(req, res, next) {
});

// ------------------------------------------------------
// reglas de peticiones REST
// ------------------------------------------------------
// .....................................................
// buscarTXyRXPorFecha()
// ej. GET /fecha/'fecha ejemplo 1'
// devuelve en el cuerpo las tasas para esa fecha
// (una tasa por linea)
// .....................................................
servidorExpress.get('/fecha/:laFecha', function(req, res){

});
```

```
// ....................................................
// insertarTasa()
// ej. POST /tasa
// en el cuerpo un JSON: {"fecha": "9998", "tx": 5, "rx": 10}
//
// ....................................................
servidorExpress.post('/tasa', function (req,res){
});

// ....................................................
// "main()"
// ....................................................
servidorExpress.listen (8080);
console.log ("todo preparado, espero en 8080 (http)");
```

Código 25: Servidor

Aquí podemos ver varias cosas. Primeramente, importamos mediante la declaración `require` todas las bibliotecas necesarias para constituir el servidor, le decimos al servidor que escuche en el puerto 8080 y escribimos la función que arranque el servidor. A continuación, se escriben las funciones del servidor. Estas funciones servirán de traductor entre el lenguaje *REST* que establecerán el terminal móvil y el servidor, y el *SQL* que comunica la lógica con la base de datos. La función `servidorExpress.get()` sirve para consultar la base de datos mientras que `servidorExpress.post` servirá para añadir una nueva captura a la base de datos.

Expliquemos ahora, brevemente, cómo funciona el lenguaje *REST* de HTTP. El lenguaje *REST* se basa en peticiones encabezadas por una acción que indica la operación a realizar. Por ejemplo, podemos encontrar:

- GET: Para consultar, acceder.
- POST: Para modificar. Suele ir acompañada de un cuerpo que incluye la modificación.
- DELETE: Para eliminar.
- PUT: Para actualizar, reemplazar.

Pues bien, tras haber explicado esto, podemos ver que el nombre de las funciones de nuestro servidor explica la acción que realiza: `servidorExpress.get()` recibirá las peticiones de consulta por parte del terminal, por lo que ejecutará las funciones del archivo `lógica.js` correspondientes, mientras que `servidorExpress.post()` recibirá las capturas realizadas por el terminal, por lo que ejecutará el método `insertarTasa()` del archivo `lógica.js`.

Aquí tenemos los dos métodos:

```
servidorExpress.get('/fecha/:laFecha', function(req, res){

    // se ejecuta cuanto llega una petición GET de cliente http
    console.log (" * GET /fecha/:laFecha      " + req.url );

    var fecha = req.params.laFecha;

    laLogica.buscarTXyRXPorFecha(fecha, function (err, resultados) {

        if (err) {
            res.writeHead(409, {'Content-Type': 'text/plain'});
            res.end();
            return;
        }

        if (resultados == undefined) {
            res.writeHead(404, {'Content-Type': 'text/plain'});
            res.end();
            return;
        }

        res.writeHead(200, {'Content-Type': 'text/plain'});
        res.write ( JSON.stringify (resultados) + "\n");
        res.end();

    });
});
```

Código 26: Métodos Servidor.js

```
// ....................................................
// insertarTasa()
// ej. POST /tasa
// en el cuerpo un JSON: {"fecha": "9998", "tx": 5, "rx": 10}
//
// ....................................................
servidorExpress.post('/tasa', function (req,res){
    console.log(" * POST /tasa     " + req.url);

    // console.log ("            " + JSON.stringify(req.body));
    console.log ("           " + req.body);

     var datosTasa = JSON.parse (req.body);

    laLogica.insertarTasa(datosTasa, function (err) {
        if (err) {
            res.writeHead(409, {'Content-Type': 'text/plain'});
            res.end();
            return;
        }

        console.log ("                  insercion correcta ");

        res.writeHead(200, {'Content-Type': 'text/plain'});
        res.end();
    });
});
```

Código 27: Métodos Servicio.js

Centrémonos en el método `servidorExpress.post()`, ya que es a la que llamará nuestro terminal.

Como hemos dicho, la petición *POST* va acompañada de un cuerpo que incluye la modificación. Pues bien, ese cuerpo tiene que venir construido en formato *JSON*, un formato de texto ligero de intercambio de datos que tiene la siguiente estructura:

```
{"fecha": "9998", "tx": 5, "rx": 10,"latencia": "time=15 ms"}
```

Código 28: Ejemplo cuerpo JSON

En la función correspondiente a POST podemos ver que la petición HTTP tiene que comenzar por `/tasa` para que el servidor entienda que se trata de una petición POST y, así, operar en consecuencia. También vemos que se procesa el cuerpo a través de la función `JSON.parse()`, y a continuación, se llama a la función de la Lógica. Como vemos se escribe una función en el argumento de `insertarTasa()`; esto se corresponde, como dijimos, a la función (`callback`) que tiene que realizar con las respuestas del método `insertarTasa()`. En este caso, muestra por consola si ha habido un error o, si por el contrario, la inserción ha sido correcta. Para cada caso, devuelve un código (409 y 200) que nos servirán desde el terminal para saber si la inserción ha sido correcta o no.

Con esto, hemos visto cómo realizar una base de datos en un servidor externo.

Ahora introduciremos el cliente que hemos implementado en el terminal Android.

3.3 Cliente Android

El cliente Android que hemos implementado está basado en el mismo concepto que el testeo HTTP explicado previamente: Una tarea asíncrona en segundo plano mediante `AsyncTask` y la introducción de una interfaz con un método `callback` que se ejecutará a partir de los resultados de la tarea.

Así pues, el cliente que incluirá nuestra aplicación tendrá la siguiente estructura:

```
public class PeticionarioREST extends AsyncTask<Void, Void, Boolean> {

    //-------------------------------------------------------
    //-------------------------------------------------------

    public interface RespuestaREST {
        public void callback (int codigo, String cuerpo);
    }

    private String elMetodo;
    private String urlDestino;
    private String elCuerpo = null; //Arquitectura JSON
    private RespuestaREST laRespuesta;

    private int codigoRespuesta;
    private String cuerpoRespuesta = "";

    //-------------------------------------------------------
    //-------------------------------------------------------

    public PeticionarioREST(){

    }

    public void hacerPeticionarioREST (String metodo, String urlDestino,
    String cuerpo, RespuestaREST respuesta){
    }

    //-------------------------------------------------------

    @Override
    protected Boolean doInBackground(Void... arg0) {
    }
    //-------------------------------------------------------
    //-------------------------------------------------------

    protected void onPostExecute (Boolean comoFue){
    }
}
```

Código 29: Estructura cliente Android

Como vemos, primero encontramos la interfaz `RespuestaREST` con el método `callback` cuyos argumentos son el código y el cuerpo de la respuesta que vendrán del servidor. El código nos ayudará a saber cómo han sido las operaciones en el servidor:

- 200: OK
- 405: Error
- 404: Not found

Como variables privadas podemos encontrar lo siguiente: las 4 primeras hacen referencia al tipo de solicitud que queramos hacerle al servidor (*GET*, *POST*, etc.), al URL del servidor, al cuerpo que debamos incluir en la petición y, por último, la respuesta que vayamos a recibir. Por otra parte, las 2 variables siguientes se corresponden a las componentes que incluirá la respuesta del servidor.

Pasemos ahora a explicar los métodos:

En primer lugar encontramos un constructor y el método `hacerPeticionarioREST()`. Estos dos métodos sirven para arrancar la petición desde nuestra tarea temporizada.

```
public PeticionarioREST(){
}

public void hacerPeticionarioREST (String metodo, String urlDestino,
    String cuerpo, RespuestaREST respuesta){

    this.elMetodo = metodo;
    this.urlDestino = urlDestino;
    this.elCuerpo = cuerpo;
    this.laRespuesta = respuesta;

    this.execute(); //thread que ejecutará doInBackground
}
```

Código 30: Constructor y método hacerPeticionarioREST

En el argumento de `hacerPeticionarioREST` incluimos lo siguiente:

`String método`: Texto que indicará el tipo de solicitud. Puede ser *GET*, *POST*, *DELETE* … Se corresponde al lenguaje *REST*.

`String urlDestino`: Texto con la dirección del servidor. La dirección IP será siempre la misma, no obstante, la URL cambiará en función del tipo de solicitud que estemos haciendo. Por ejemplo:

- Petición *GET*: http://84.76.156.110:8080/fecha/03-02-2014
- Petición *POST*: http://84.76.156.110:8080/tasa

`String cuerpo`: Texto que queramos que incluya el cuerpo de la URL con la modificación. En nuestro caso, la nueva captura que se vaya midiendo cada periodo

`RespuestaREST`: *Callback* con la función que queramos que se ejecute tras conocerse la respuesta del servidor.

Pues bien, en el método asignamos los argumentos a nuestras variables privadas y ejecutamos la tarea con `execute()`. A diferencia de `hacerHttpTest`, aquí no es necesario incluir la función `get()` puesto que no necesitamos las respuestas del servidor para calcular la siguiente captura.

Tras esto encontramos el `doInBackground()`, el método que realizará la acción en segundo plano.

```
protected Boolean doInBackground(Void... arg0) {
        UyC.im("doInBackground PeticionarioREST");
        try{
            //envio de peticion
            //ordenador casa http://84.76.156.110:8080

            UyC.im("Me conecto a >"+ this.urlDestino + "<");

            URL url = new URL (this.urlDestino);
            //Creo una conexion HTTP y le digo el tipo de solicitud
            HttpURLConnection connection = (HttpURLConnection) url.openConnection();
            UyC.im("Creo la conexi□TTP y le digo el tipo de solicitud");
            connection.setRequestMethod(this.elMetodo);//Introduzco el tipo de solicitud
            UyC.im(this.elMetodo);
            connection.setDoInput(true);
```

```
            //veo si la peticion no es GET y si el cuerpo no est□ac
            if(!this.elMetodo.equals("GET") && this.elCuerpo != null){
                //Quiere decir que el m鋖do es POST
                UyC.im("doInBackGround(): no es get, pongo cuerpo");
                connection.setDoOutput(true);
                //a□ el cuerpo de la peticion. ARQUITECUTRA JSON
                DataOutputStream dos = new DataOutputStream(connection.getOutputStream());
                dos.writeBytes(this.elCuerpo);
                dos.flush();
                dos.close();
            }

            //peticion enviada
            UyC.im("doInBackGround(): peticion enviada");
```

Código 31: doInBackground 1/2

En primer lugar explicaremos la parte en la que elaboramos la petición a partir de los datos introducidos a través de `hacerPeticionaroREST`:

Al igual que en la clase `HttpTest`, creamos un objeto `URL` con la dirección de destino y, a partir de este objeto, arrancamos una nueva conexión HTTP con el método `openConnection()`. Tras haber arrancado la conexión, le paso el tipo de solicitud que vamos a realizar con el método `setRequestMethod()`.

```
//obtencion de respuesta del servidor

    int rc = connection.getResponseCode(); //codigo de la respuesta
    String rm = connection.getResponseMessage(); //mensaje de la respuesta
    //String respuesta = "" + rc + " : " + rm;
    UyC.im("recibo respuesta: codigo>"+Integer.toString(rc)+"< - respuesta>"+rm+"<");
    this.codigoRespuesta = rc;

    try{
    //Buffers para leer el cuerpo de la respuesta del servidor
       InputStream is = connection.getInputStream();
    //Le paso la respuesta del servidor
       BufferedReader br = new BufferedReader (new InputStreamReader(is));

       UyC.im("leyendo cuerpo");

       StringBuilder acumulador = new StringBuilder();
       String linea;
       while ((linea = br.readLine())!=null){
            UyC.im(linea);
            acumulador.append(linea); //a□ las lineas al StringBuilder
       }
       UyC.im("FIN leyendo cuerpo");



       this.cuerpoRespuesta = acumulador.toString();
            UyC.im("cuerpo recibido: " + this.cuerpoRespuesta);

            connection.disconnect();
       }catch(IOException e){
    //Dispara excepcion cuando la respuesta REST no tiene cuerpo
          UyC.im("Parece que no hay cuerpo en la respuesta");
       }

       return true;
  }catch(Exception ex){
       UyC.im("ocurri□guna otra excepcon: " + ex.getLocalizedMessage());
  }

  return false;
}
```

Código 32: doInBackground parte 2/2

Tras esto, compruebo el tipo de solicitud que hemos introducido para saber si hace falta añadirle el cuerpo. Para ello, nos fijamos si el método no es *GET* y si el cuerpo no está vacío, de esta manera sabremos que la solicitud es tipo *POST*. Veamos ahora la recepción de datos del servidor:

Extraemos de la conexión tanto el código de la respuesta como el mensaje que pueda contener. Asigno el código que nos hayan enviado a nuestra variable privada (`this.codigoRespuesta`) y leemos el cuerpo de la respuesta a través de los objetos `InputStream` y `BufferedReader`. Mediante el objeto `StringBuilder` convierto el cuerpo de la respuesta a `String` para poder trabajarlo como texto simple y así poder asignárselo a la variable privada correspondiente (`this.cuerpoRespuesta`). Por último cierro la conexión a través del método `disconnect()`. Si todo ha ido correctamente, la tarea devolverá "`true`".

Por último, expliquemos el método `onPostExecute()`.

```
protected void onPostExecute (Boolean comoFue){
        UyC.im("onPostExecute() comoFue = " + comoFue);
        this.laRespuesta.callback(this.codigoRespuesta, this.cuerpoRespuesta);
}
```

Código 33: onPostExecute, clase PeticionarioREST

Como vemos, recoge el resultado de `doInBackground`, "`true`" o "`false`", y ejecuta el método `callback` de la interfaz `RespuestaPeticionarioREST` en función de las 2 variables privadas relacionadas con las respuestas del servidor.

Hemos visto como se implementa un cliente en Android. Ahora, para llamarlo, basta hacerlo de la misma manera que `HttpTest`.

```
new PeticionarioREST().hacerPeticionarioREST("POST",
"http://84.76.156.110:8080/tasa", cuerpoJSON, new PeticionarioREST.RespuestaREST() {
                @Override
                public void callback(int codigo, String cuerpo) {
                    // TODO Auto-generated method stub

                }
});
```

Código 34: Llamada a hacerPeticionarioREST

Ahora, pasaremos a explicar la clase `TareaTemporizada` de nuestra aplicación que será la que realice los envíos al servidor así como la medición de las capturas.

3.3.1 Cliente en TareaTemporizada.java

Como habíamos planteado anteriormente, la clase TareaTemporizada tendrá la siguiente forma:

```
public class TareaTemporizada extends TimerTask{

    @Override
    public void run() {
        // TODO Auto-generated method stub
        miTarea();
    }

    private void miTarea(){

        //Aquí escribiremos el algoritmo para medir capturas...

    }

}
```

Código 35: Clase TareaTemporizada

Será en el método `miTarea()` donde escribiremos las acciones que se tendrán que repetir periódicamente.

Pues bien, en primer lugar debemos medir una nueva captura. Para ello, basta con declarar un nuevo objeto `Captura`. Recordemos que al hacer `new Captura` estaremos haciendo directamente el cálculo de las tasas y la latencia.

A continuación, debemos extraer los componentes del objeto `Captura` por separado:

```
private void miTarea (){

        Captura laCaptura = new Captura();
        //Al hacer new Captura(), se generar□n nuevo chequeo HTTP
        Tasa up = laCaptura.getTasaSubida();
        Tasa down = laCaptura.getTasaBajada();
        String fecha = laCaptura.getFechaCaptura();
        String latencia = laCaptura.getLatencia();
```

Código 36: Extracción por separado de las partes de la captura

A partir de ahora, pueden pasar 2 cosas: que la captura falle o que sea correcta. Que la captura falle quiere decir que puede haberse producido algún error interno y que, por lo tanto, alguna de las componentes sea `null` (que no apunta a ningún objeto). En ese caso, la captura se considera fallida y habrá que esperar a la medición siguiente. Si, por el contrario, es correcta, calculamos los anchos de banda, devolvemos a nuestro `Servicio.java` los resultados y los mandamos a la base de datos.

```
if (up == null || down == null || fecha==null || latencia==null){
            UyC.im("Captura fallida");

        } else {
            UyC.im("Captura correcta");

            double upstreamBandwidth = up.getBandwidth();
            double downstreamBandwidth = down.getBandwidth();

            String up_txt = UyC.procesarBandwidth(upstreamBandwidth);
            //Ancho de banda de subida en texto
            String down_txt = UyC.procesarBandwidth(downstreamBandwidth);
            //Ancho de banda de bajada en texto

            String captura = "Captura medida: subida>" + up_txt + "<\n bajada >"
            + down_txt + "<\n fecha >" + fecha + "<";

            String capturaParaTrocear = up_txt + ";" + down_txt + ";" + fecha + ";" + latencia;

            ultimaCaptura = capturaParaTrocear;
```

Código 37: Comprobación de si la captura es correcta

Como vemos en el *Código 36*, extraemos los anchos de banda en *upstream* y *downstream* con el método `getBandwidht` de la clase `Tasa`, y, después, los pasamos a texto con el método `procesarBandwidth` de la clase `UyC`. La variable `ultimaCaptura` sirve para que, entre medición y medición, permanezca guardada la última captura que se haya medido.

Ahora creamos una variable de tipo `String` llamada `capturaParaTrocear` que incluye los 4 resultados de la `Captura` separados por ";". Esto nos sirve para mandar de una tacada los resultados al Servicio y que se encargue este de mandarlos al `MainActivity` para que sean procesados para mostrarlos al usuario.

Ahora viene cuando le enviamos al Servicio la última captura medida.

Para ello hace falta, previamente, escribir un método en la clase `Servicio` que se encargue de recoger las capturas de `TareaTemporizada`. Al igual que hicimos con la clase `MiTimer`, deben existir métodos "nexo" entre el Servicio y la clase `TareaTemporizada`.

```
//Envío la captura a MainActivity:
   new Servicio().pasarUltimaCaptura(ultimaCaptura);
   //Le devolvemos al servicio la última captura

   UyC.im(captura);

   String cuerpoJSON = this.generarCuerpoJSON(fecha, upstreamBandwidth, downstreamBandwidth);

   UyC.im(cuerpoJSON);

   //Guardar captura en el servidor
   new PeticionarioREST().hacerPeticionarioREST("POST", "http://84.76.156.110:8080/tasa",
   cuerpoJSON, new PeticionarioREST.RespuestaREST() {
         @Override
         public void callback(int codigo, String cuerpo) {
             // TODO Auto-generated method stub
             if (codigo == 200){
                 UyC.im("Inserci□n la db correcta");
             }
         }
   });
```

Código 38: Envío de resultados al Servicio y al Servidor

Dicho método se llamará `pasarUltimaCaptura` que explicaremos posteriormente.

Tras esto, generamos el cuerpo de nuestra solicitud HTTP con nuestra última captura medida. Se ha escrito un pequeño método para generar un cuerpo *JSON*:

```
private String generarCuerpoJSON (String fecha, double up, double down){
        String cuerpoJSON = "{\"fecha\":\"" + fecha + "\",\"tx\":" +
        Double.toString(up)+ ",\"rx\":"+ Double.toString(down)+ "}";
        return cuerpoJSON;
}
```

Código 39: Método generarCuerpoJSON

3.3.2 Mostrando los resultados

Hemos introducido en el apartado anterior el método `pasarUltimaCaptura` de la clase `Servicio.java`, que servirá para ir recibiendo en el servicio las capturas que va midiendo la tarea temporizada.

```
public void pasarUltimaCaptura (String capturaActual){
        ultimaCaptura = capturaActual;
        Intent enviarCaptura = new Intent ("captura-hecha");
        enviarCaptura.putExtra("captura", ultimaCaptura);
        LocalBroadcastManager.getInstance(this).sendBroadcast(enviarCaptura);
}
```

Código 40: Método pasarUltimaCaptura

Como vemos, recoge la captura actual y la guarda en una variable privada (`ultimaCaptura`). Creamos un `Intent` para pasarle la captura a nuestra actividad principal e insertamos la captura mediante el método `putExtra()` que hemos explicado previamente (véase *Creando y arrancando el Servicio*).

A continuación se utiliza la clase `LocalBroadcastManager` que clase se usa para mandar `Intent`s en *broadcast* a las clases que hayan implementado un objeto `BroadcastReceiver`. Esta es la manera que tiene Android de ir mandando y recibiendo eventos que vayan ocurriendo de manera continua, como puede ser el cálculo de Capturas de calidad de servicio.

Finalmente, y para terminar la explicación de lo que sería la parte interna de la aplicación, vamos a explicar la clase `UyC.java`

3.4 UyC.java

Posiblemente, el lector haya notado que en varias partes del código aparecen varias llamadas a la clase `UyC`. Como por ejemplo:

```
UyC.lanzaToast(this, "Debe estar la red de datos como red principal.");
UyC.im("Recibo el periodo: " + periodoInsertado);
```

Código 41:Ejemplo de uso de la clase UyC

Además, anteriormente hemos pospuesto la explicación de alguno de sus métodos.

Pues bien, esta clase se llama `UyC` porque es la abreviatura de “Utilidades y Constantes”. Lo que va a contener esta clase será una serie de métodos de apoyo tanto para el desarrollador como para el usuario, como por ejemplo, un método que lanza mensajes por pantalla al usuario u otro método para sacar por consola mensajes internos para facilitar el desarrollo al programador.

He aquí la clase `UyC.java`.

```
public class UyC { //Utilidades Y Constantes

    public static final String httpDescarga =
"https://www.dropbox.com/s/w96agv7wbyzseqy/file.txt?dl=0";

    public static final String TAG = "com.example.through_check_1"; // TAG para el Log.d

    public static void im (String mensaje){//mensaje interno
    }
```

```
    public static void lanzaToast (Context contexto, String mensaje){//Mostrar un Toast
    }

    public static long readFile(String fileName){ //Lector de archivos
    }

    public static String getFecha (){ //Obtener fecha
    }

    public static long periodoEnMiliseg (long periodoEnSegundos){ //Periodo en milisegundos
    }

    public static String [] trocearCaptura (String ultimaCaptura){//Troceador
    }

    public static String procesarBandwidth (double anchoBanda){ //Redondear ancho de banda
    }

    public static String readStream(InputStream is) throws IOException {//Lector de stream
    }
}
```

Código 42: Estructura clase UyC

Explicaremos brevemente lo que hace cada variable privada y cada método

- `String httpDescarga`: URL a la que nos conectamos para realizar el chequeo HTTP
- `String TAG`: Texto para el método `Log.d`.
- `im (String mensaje)`: Saca un mensaje por consola mediante el método `Log.d`.
- `lanzaToast()`: Método que lanza un `Toast` al usuario. Un `Toast` es un mensaje que aparece durante un periodo de tiempo finito. Es muy utilizado en Android.

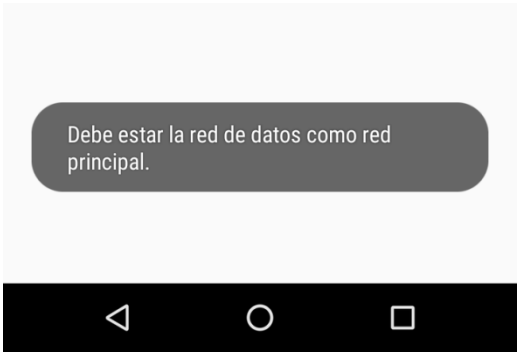

Figura 4: Toast

- `readFile()`: Método para leer archivos y extraer su contenido y decirnos los bytes que ocupa.
- `getFecha()`: Método que nos dice la fecha exacta en el instante que sea llamada. Lo usaremos para saber el momento en el que hemos hecho la captura.
- `periodoEnMiliseg()`: Método para pasar el periodo en minutos introducido por el usuario en la interfaz gráfica a milisegundos.
- `trocearCaptura()`: Método para separar cada componente de la captura.
- `procesarBandwidth()`: Método para redondear a un decimal el ancho de banda.
- `readStream()`: Lector de *streams*. Sirve para trabajar con las respuestas que devuelve, por ejemplo, un comando insertado en el *Runtime* de Android.

Como vemos, la clase contiene una cantidad de métodos que no tienen ningún tipo de relación entre sí, ya que esta clase cumple un poco la función de "caja de herramientas" de la propia aplicación.

4 Actividad principal y aspecto gráfico

Una vez terminada la parte que se refiere al diseño "interno" de la aplicación, vamos a pasar a explicar la parte de la aplicación que interactúa con el usuario y, que a su vez, implementa todo el aspecto gráfico de la aplicación.

4.1 Actividad principal

`MainActivity.java`, o la Actividad Principal, es la clase más importante de nuestra aplicación por varios motivos. El primero es que es la única Actividad de nuestra aplicación, por lo tanto, si no estuviera, no tendríamos aplicación. En segundo lugar, es la parte que sirve de nexo entre nuestra aplicación y el usuario. Por último, ejecutará la parte gráfica de nuestra aplicación.

```
public class MainActivity extends Activity {

    private GestorDeConexion elGestor;

    private TextView miTextView; //TextView para indicar si el servicio ha arrancado
    private EditText editTextPeriodo;

    private TextView mostradorCaptura;

    private String ultimaCaptura; //Ultima captura realizada

    private String ultimaTasaSubida;
    private String ultimaTasaBajada;
    private String fechaUltimaCaptura;
    private String latencia;


    @Override
    protected void onCreate(Bundle savedInstanceState) {
    }

    @Override
    protected void onSaveInstanceState(Bundle estado){ //Guardo el estado de la clase;
    }

    @Override
    public void onDestroy(){
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
    }

    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
    }

    public void onClick (View quien){
    }

    private String formatoCaptura (){
    }

    private BroadcastReceiver observadorDelServicio = new BroadcastReceiver(){
    };
}
```

Código 43: Clase MainActivity

En primer lugar, vamos a explicar la estructura de la actividad, ya que consta de varias partes:

1) Métodos propios de la actividad: Esta clase incluye una serie de métodos que se deben añadir para que la clase se comporte como una actividad. Dichos métodos vienen relacionados directamente con el ciclo de vida de la aplicación.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
}

@Override
protected void onSaveInstanceState(Bundle estado){ //Guardo el estado de la clase;
}

@Override
public void onDestroy(){
}

@Override
public boolean onCreateOptionsMenu(Menu menu) {
}

@Override
public boolean onOptionsItemSelected(MenuItem item) {
}
```

Código 44: Métodos propios de la actividad

2) Métodos que interactúan con el usuario: Son métodos que ejecuta el usuario directamente, por ejemplo: El método que se ejecuta tras apretar un botón o el método que hace que se lance una notificación al usuario.

```
public void onClick (View quien){
}

private String formatoCaptura (){
}

private void recopilarCaptura (String [] capturaTroceada){
}
```

Código 45: Métodos que interactúan con el usuario

3) Métodos exigidos por el propio programa: Métodos que, a priori, no vienen establecidos en el diseño pero que deben ser implementados por exigencias de otras clases o herramientas.

```
private BroadcastReceiver observadorDelServicio = new BroadcastReceiver(){
};
```

Código 46: Métodos exigidos por el propio programa

4.1.1 Métodos propios de la actividad

Vamos a empezar explicando los métodos propios de la actividad.

En primer lugar nos encontramos con el método `onCreate()`. Este método es el primero en toda la aplicación en ser ejecutado, y en el encontramos lo siguiente:

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    elGestor = new GestorDeConexion (this);
    miTextView = (TextView) findViewById (R.id.textView1);
    editTextPeriodo = (EditText) findViewById (R.id.editText1);
    mostradorCaptura = (TextView) findViewById (R.id.mostradorCaptura);
    UyC.im("Arranca la aplicacion");

    //Registro para recibir mensajes del servicio
    //Tenemos un observador (observadorDelServicio) para recibir intents
    //con el nombre de accion: "captura-hecha"

    LocalBroadcastManager.getInstance(this).registerReceiver(observadorDelServicio,
    new IntentFilter ("captura-hecha"));

    if (savedInstanceState != null){
        ultimaCaptura = savedInstanceState.getString("ultima_captura");
        int texto_inicio = savedInstanceState.getInt("texto_arranque");
        mostradorCaptura.setText(ultimaCaptura);
        miTextView.setText(texto_inicio);
    }
}
```

Código 47: Método onCreate de la clase MainActivity

Las dos primeras líneas de código de este método vienen escritas por defecto cuando se crea el proyecto en Eclipse, la primera es una llamada al método de la clase predecesora y la segunda es el método que enlaza la actividad con el archivo XML que contiene el aspecto gráfico de la actividad.

A partir de aquí es todo introducido por el desarrollador:

En primer lugar tenemos un puntero que apunta a un objeto de `GestorDeConexion`, para saber a qué red estamos conectados. Después tenemos 3 enlaces a elementos del archivo referente a la parte gráfica. Esto se debe para manipular cada elemento por separado:

`TextView`: Lo que vendría a ser un cuadro de texto.

`EditText`: Un campo que recoge texto.

A continuación, encontramos una llamada a través de la clase `LocalBroadcastManager` que sirve para recibir los eventos del Servicio. Y, por último, encontramos una condición en función del objeto que se le pasa al método `onCreate()`. Este método suele almacenar todo aquello que queramos guardar de la aplicación para que se recupere tras ser destruida.

Tras el método `onCreate()` encontramos el resto de métodos relacionados con la clase Activity. Explicaremos brevemente su función:

- `onSaveInstanceState()`: Método que se llama antes de destruir una aplicación (por falta de recursos en el sistema, por ejemplo) para poder guardar estado de la clase, cosa que, al recuperar la actividad, recupere el estado en el que estaba cuando se destruyó.
- `onDestroy()`: Método que se llama cuando se va a destruir la aplicación.
- `onCreateOptionsMenu()`: Método creado por defecto por si queremos crear un menú con opciones en nuestra aplicación.

- `onOptionsItemSelected()`: Método creado por defecto para usar cuando se selecciona algo en el menú de opciones.

A continuación aparece el método más importante de la clase. Se trata del método `onClick()`, que es el que ejecutará el servicio.

4.1.2 Métodos que interactúan con el usuario

Aquí tenemos el contenido del método `onClick`:

```
public void onClick (View quien){

    //Lanzar capturas
    if (elGestor.redDeDatosActivada()){

        if (quien.getId() == R.id.boton_iniciar){

            String periodoInsertado = editTextPeriodo.getText().toString();

            if (!periodoInsertado.matches("")){

                this.miTextView.setText(R.string.servicio_iniciado);
           //Texto que se muestra por pantalla para indicar que se ha iniciado el checking

                UyC.im("Recibo el periodo: " + periodoInsertado);
                long periodo = Long.parseLong(periodoInsertado);


                Intent intencion = new Intent (this,Servicio.class);
                intencion.setData(Uri.parse("Anyadir periodo"));

                //Le paso el periodo a eleccion del usuario
                intencion.putExtra("periodo", periodo); //("clave","valor")

                startService(intencion);

            }else{
                UyC.lanzaToast(this, "Debes insertar algo en el campo");
            }

        }else if (quien.getId() == R.id.boton_detener){

            stopService (new Intent (this, Servicio.class));
            //Si pulsamos el bot□Detener", detenemos el servicio
            miTextView.setText(R.string.servicio_detenido);
            //A□mos el texto de detener servicio

        }
    } else {
        UyC.lanzaToast(this, "Debe estar la red de datos como red principal.");
    }
}
```

Código 48: Método onClick de la clase MainActivity

Este método se va a ejecutar cuando el usuario pulse tanto el botón de “Iniciar servicio” como el de “Detener servicio”. Expliquemos su algoritmo:

En primer lugar, el método comprobará si el terminal tiene configurada la red de datos como red principal. Si no, lanzará un `Toast` con el mensaje: “Debe estar la red de datos como red principal”

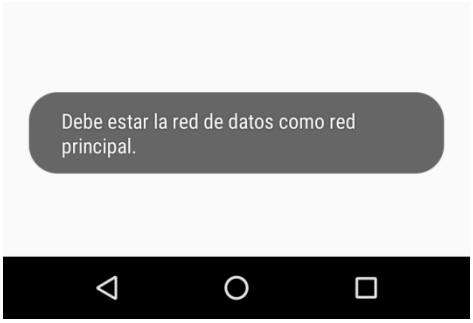

Figura 5: Toast

Una vez que la red de datos se encuentra como red principal (por principal entendemos que no funcionamos a través de WiFi), el método comprueba qué botón ha sido pulsado, si el de iniciar o el de detener. Si el botón ha sido el de iniciar, recogemos el periodo, creamos una intención, le añadimos el periodo y lanzamos la intención hacia la clase `Servicio`

```
if (!periodoInsertado.matches("")){

    this.miTextView.setText(R.string.servicio_iniciado);
    //Texto que se muestra por pantalla para indicar que se ha iniciado el checking

    UyC.im("Recibo el periodo: " + periodoInsertado);
    long periodo = Long.parseLong(periodoInsertado);


    Intent intencion = new Intent (this,Servicio.class);
    intencion.setData(Uri.parse("Anyadir periodo"));

    //Le paso el periodo a eleccion del usuario
    intencion.putExtra("periodo", periodo); //("clave","valor")

    startService(intencion);

}else{
    UyC.lanzaToast(this, "Debes insertar algo en el campo");
}

}else if (quien.getId() == R.id.boton_detener){

    stopService (new Intent (this, Servicio.class));
    //Si pulsamos el botón "Detener", detenemos el servicio
    miTextView.setText(R.string.servicio_detenido);
    //Añadimos el texto de detener servicio

}
```

Código 49: Método onClick, clase MainActivity

Si el botón que se ha pulsado es el de detener, detenemos el Servicio mediante `stopService`, cerramos el *Broadcast* mediante el cual recibimos las capturas del Servicio y lanzamos un mensaje al usuario avisándole que el servicio ha sido terminado.

El otro método que interactúa con el usuario es `formatoCaptura`. Este método manipula las capturas recibida como si de un texto se tratara para mostrarlas por pantalla:

```
private String formatoCaptura (){
        String formatoSalida = "Ultima captura: \n" +
                               "Up: " + this.ultimaTasaSubida + " bps\n" +
                               "Down: " + this.ultimaTasaBajada + " bps\n"+
                               "latencia: " + this.latencia + "\n"+
                               "fecha: " + this.fechaUltimaCaptura;
        return formatoSalida;
}
```

Código 50: Método formatoCaptura, clase MainActivity

A continuación tenemos el método `recopliarCaptura` que, simplemente, le pasamos el array de String con la captura troceada y nos almacena su información en las variables privadas de la clase:

```
private void recopilarCaptura (String [] capturaTroceada){
        this.ultimaTasaSubida = capturaTroceada[0];
        this.ultimaTasaBajada = capturaTroceada[1];
        this.fechaUltimaCaptura = capturaTroceada[2];
        this.latencia = capturaTroceada[3];
}
```

Código 51: Método recopilarCaptura

Veremos que este es el texto que se mostrará cuando realicemos las pruebas de funcionamiento.

4.1.3 Métodos exigidos por el programa

Para terminar de explicar la clase `MainActivity`, vamos a explicar el método `observadorDelServicio`. Este método será el encargado de recibir las capturas del Servicio. Decimos que es

```
private BroadcastReceiver observadorDelServicio = new BroadcastReceiver(){

        @Override
        public void onReceive(Context context, Intent intent) {
            // TODO Auto-generated method stub

            ultimaCaptura = intent.getStringExtra("captura");

            String [] capturaTroceada =  UyC.trocearCaptura(ultimaCaptura);

            recopilarCaptura(capturaTroceada);

            mostradorCaptura.setText(formatoCaptura());//Muestro captura
        }

};
```

Código 52: Recibidor de broadcast

exigido por el programa porque es lo que se debe escribir para ejecutar un escuchador de servicio. Viene impuesto por la herramienta que estamos utilizando.

En realidad, `observadorDelServicio` no se trata de un método, sino de un objeto de la clase `BroadcastReceiver` que implementa un `callback`, `onReceive`, para manipular la respuesta del `Service`.

En el *Código 51* podemos ver que extraemos la captura del `intent` proveniente del Servicio, separamos su contenido y guardamos las variables por separado.

4.2 Aspecto gráfico

Vamos a explicar ahora el aspecto gráfico de la aplicación. Como hemos adelantado al inicio de este proyecto, la parte gráfica en el desarrollo de aplicaciones Android se realiza mediante código *XML*.

Pues bien, Eclipse genera una serie de archivos y subcarpetas, cada vez que se crea un nuevo proyecto, para implementar todo esto.

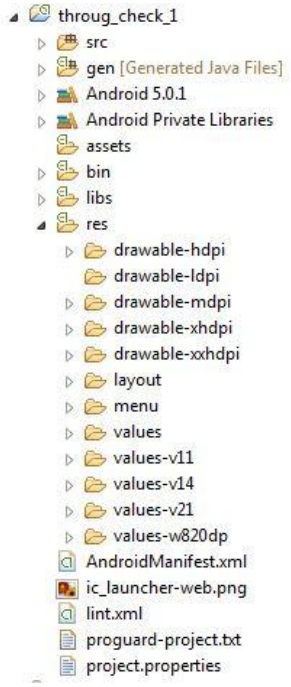

Figura 6: Subcarpetas del proyecto

En la introducción hemos adelantado que realizaremos la aplicación gráfica a través de Material Design.

Material Design es una librería de métodos y etiquetas introducida por Google a partir de la salida del sistema Android 5.0 *Lollipop*.

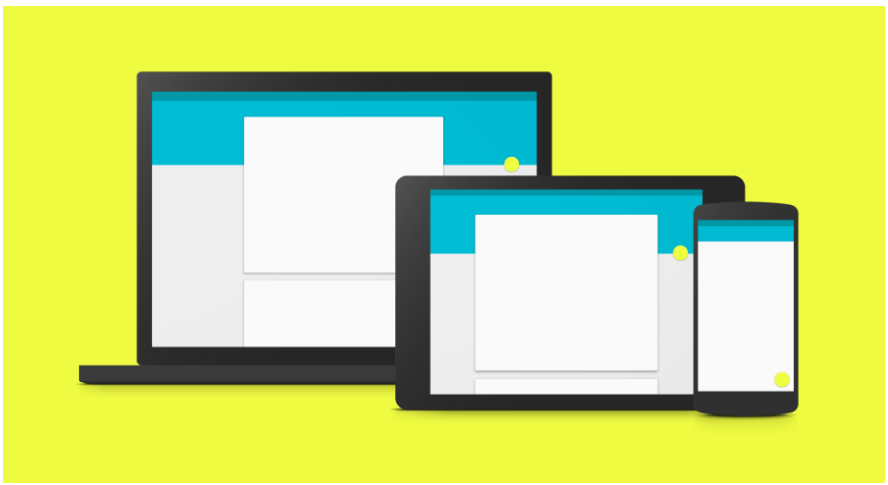

Figura 7: Material Design

Si bien no vamos a realizar un despliegue gráfico intenso, puesto que no es el objetivo del proyecto, vamos a implementar algunas etiquetas básicas como definir estilos, colocar botones, campos de texto, etc.

4.2.1 Estructura y desarrollo

En primer lugar, veamos la estructura de la actividad principal:

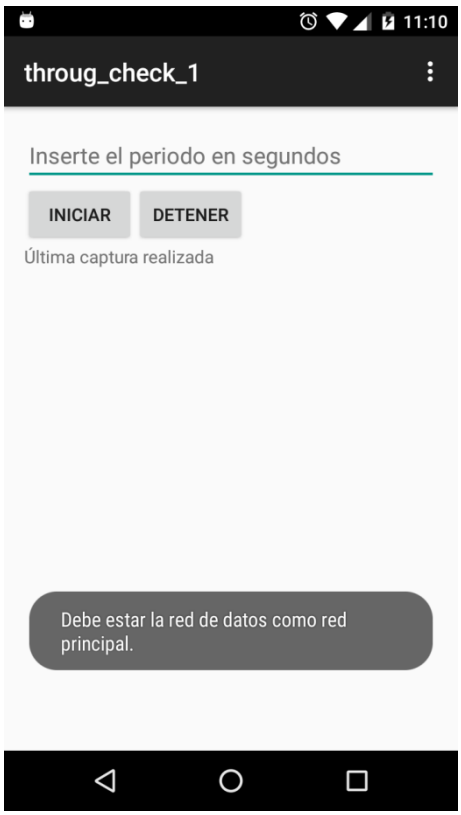

Figura 8: Aplicación

Como vemos, se trata de algo bastante simple: 2 botones, uno de iniciar y otro de detener, un `EditText` donde insertaremos el periodo de chequeo y un `TextView` donde mostraremos las capturas.

Este es el archivo *XML* donde se implementa:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/LinearLayout1"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_gravity="center_vertical"
    android:orientation="vertical"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    tools:context="com.example.throug_check_1.MainActivity" >

    <EditText
        android:id="@+id/editText1"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:ems="10"
        android:hint="@string/insertar_periodo_minutos"
        android:inputType="number"
        android:gravity="center"/>

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:gravity="center"
        >
```

```
        <Button
            android:id="@+id/boton_iniciar"
            android:text="@string/iniciar"
            style="@style/Botones"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:onClick="onClick" />

        <Button
            style="@style/Botones"
            android:id="@+id/boton_detener"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/detener"
            android:onClick="onClick" />

        <TextView
            android:id="@+id/textView1"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content" />

    </LinearLayout>

    <TextView
        android:id="@+id/titulocaptura"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/ultima_captura" />

    <TextView
        android:id="@+id/mostradorCaptura"
        android:labelFor="@id/titulocaptura"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />

</LinearLayout>
```

Código 53: Archivo activity_main.xml

Al tratarse de lenguaje *XML*, encontramos etiquetas que definen objetos. Estos objetos se introducen en *Layouts*, que vienen a ser patrones de distribución de los diferentes elementos. Por ejemplo, los `LinearLayout` sirven para estructurar los elementos de manera horizontal o vertical, como si fueran elementos en fila o en columna.

En nuestro caso, hemos colocado un `LinearLayout` cuyos elementos se distribuirán verticalmente en el siguiente orden:

1. Un `EditText` para recibir el periodo del usuario
2. Un `LinearLayout` de distribución horizontal donde colocaremos 2 botones, el de arrancar y el de detener, y un `TextView` para mostrar un mensaje en el momento que se inicia el chequeo.
3. Un `TextView` para mostrar por pantalla las tasas y la latencia de la conexión.

Además, podemos observar que cada objeto dispone de una serie de características y parámetros, en el *Código 52* en color rojo y lila, que nos sirven tanto para alinear y colocar nuestros objetos como queramos, como para configurar qué métodos se ejecuten en `MainActivity` cuando estos sean utilizados.

Describamos las características más importantes de los elementos:

- `android:id`, se usa para darle un nombre al elemento de manera que se lo pueda llamar desde el `MainActivity`

```
android:id="@+id/textView1"

miTextView = (TextView) findViewById (R.id.textView1); //LLamada desde MainActivity
```

- `android:text`, argumento que usamos para añadir texto al elemento.

```
<Button
   android:id="@+id/boton_iniciar"
   android:text="@string/iniciar"
```

Código 54: Etiqueta Button en XML

Un apunte, para introducir textos en objetos, se puede hacer de dos maneras, o bien escribiendo el texto directamente (`android:text="Iniciar"`) o bien definiendo una entrada de texto en el archivo *string.xml* y acceder a ella desde cualquier objeto usando el nombre que se le ha dado a la entrada.

```
<resources>

    <string name="app_name">throug_check_1</string>
    <string name="action_settings">Settings</string>
    <string name="iniciar">Iniciar</string>
    <string name="detener">Detener</string>
    <string name="servicio_iniciado">Ha empezado el checking</string>
    <string name="servicio_detenido">Detenemos el servicio</string>
    <string name="insertar_periodo_minutos">Inserte el periodo en segundos</string>
    <string name="ultima_captura">Última captura realizada</string>

</resources>
```

Código 55: Archivo strings.xml

- `android:onClick`, solo usado en el elemento `Button`, sirve para indicar qué método del `MainActivity` implementar en el momento que se pulse el botón

```
android:onClick="onClick"
```

Código 56: Parámetro onClick

Las demás características sirven para centrar los elementos y establecer el tamaño de estos.

Por último vamos a ver como definimos el estilo de la aplicación

4.2.2 Estilo

Los estilos en Android sirven para definir el aspecto visual de nuestra aplicación, ya que en ellos vamos a incluir los colores de fondo, de los botones, el estilo de letra, etc.

Para ello hay que crear un archivo *XML* llamado *styles.xml* y guardarlo en la carpeta correspondiente a la API de nuestra aplicación. API 21 en nuestro caso.

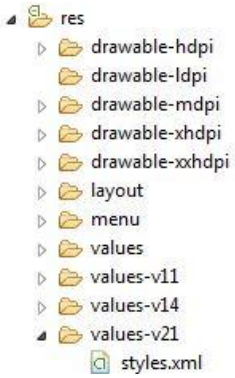

Figura 9: Subcarpetas del aspecto gráfico

Este es el archivo *styles.xml*

Para definir un estilo nuevo, se debe hacer mediante la etiqueta `<style>`. Dentro de ella, basta

```
<resources xmlns:android="http://schemas.android.com/apk/res/android">

    <style name="AppBaseTheme" parent="@android:style/Theme.Material.Light.DarkActionBar">
        <!-- API 21 theme customizations can go here. -->
        <item name="android:backgroundTint">#F7CB2D</item>
        <item name="android:textColor">#000000</item>

    </style>

    <style name="Botones">
        <item name="android:backgroundTint">#000000</item>
        <item name="android:textColor">#FFFFFF</item>
    </style>

</resources>
```

Código 57: Styles.xml

con definir etiquetas `<item>` e incluir en estas los parámetros que queramos.

En nuestro caso, hemos definidos dos estilos, uno que será el tema base de la aplicación y otro que servirá para los botones.

El resultado es el siguiente: Un color de fondo, y un cambio de color en la letra y de fondo en los botones.

Figura 10: Aplicación tras aplicarle el estilo

Por último, para aplicar el estilo debemos irnos al archivo *AndroidManifest.xml*

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.throug_check_1"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk
        android:minSdkVersion="21"
        android:targetSdkVersion="23" />

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

    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name=".MainActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>

        <!-- Declaracion del servicio -->
        <service android:name="com.example.tarea.Servicio"/>

    </application>

</manifest>
```

Código 58: AndroidManifest.xml

En la etiqueta `<application>` escribimos el parámetro `android:theme` y, mediante `@style` para llamar a los archivos *style*, escribimos el nombre de nuestro tema, en este caso `AppTheme`.

5 Pruebas de funcionamiento de la aplicación

Una vez terminada la parte de desarrollo de la aplicación, procedemos a hacer pruebas de simulación para ver cómo funciona.

En dicho estudio se deben comprobar cosas como el consumo de batería de la aplicación y su ciclo de vida:

- Ciclo de vida: Que la aplicación permanece trabajando aunque se esté realizando en el terminal otras funciones y que cuando volvemos a acceder a ella se encuentra en el estado correcto.
- Consumo de batería.

5.1 Ciclo de vida

Recordemos que las actividades en Android disponen de un ciclo de vida, una serie de cambios de estado por los que pasa desde que se crea hasta que se destruye.

Aquí tenemos los estados por los que pasará la aplicación:

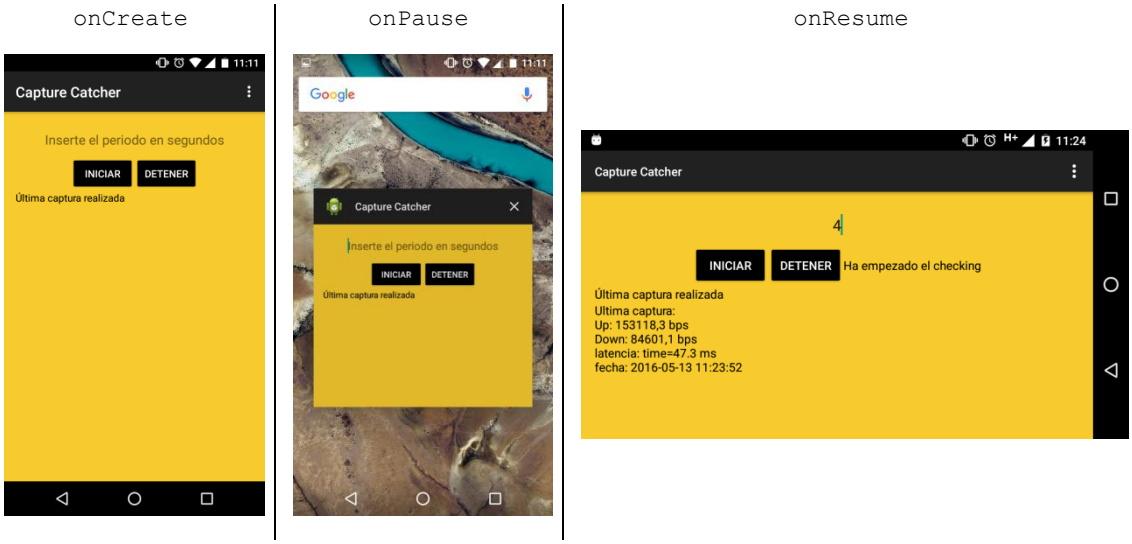

Figura 11: Estados de la actividad

Pues bien, es importante comprobar que cuando iniciamos la aplicación, empezamos un chequeo de capturas, salimos de la aplicación y volvemos a entrar en ella, se haya guardado la información y se muestra la captura más reciente. En otras palabras, si el cambio de estados que sufre una actividad no ha afectado.

La manera más rápida de comprobar si se guarda la información y luego se muestra correctamente, es cambiar la orientación del teléfono con la aplicación abierta. Si la información no está bien salvada o recuperada, el aspecto de la aplicación será igual al que tiene recién abierta.

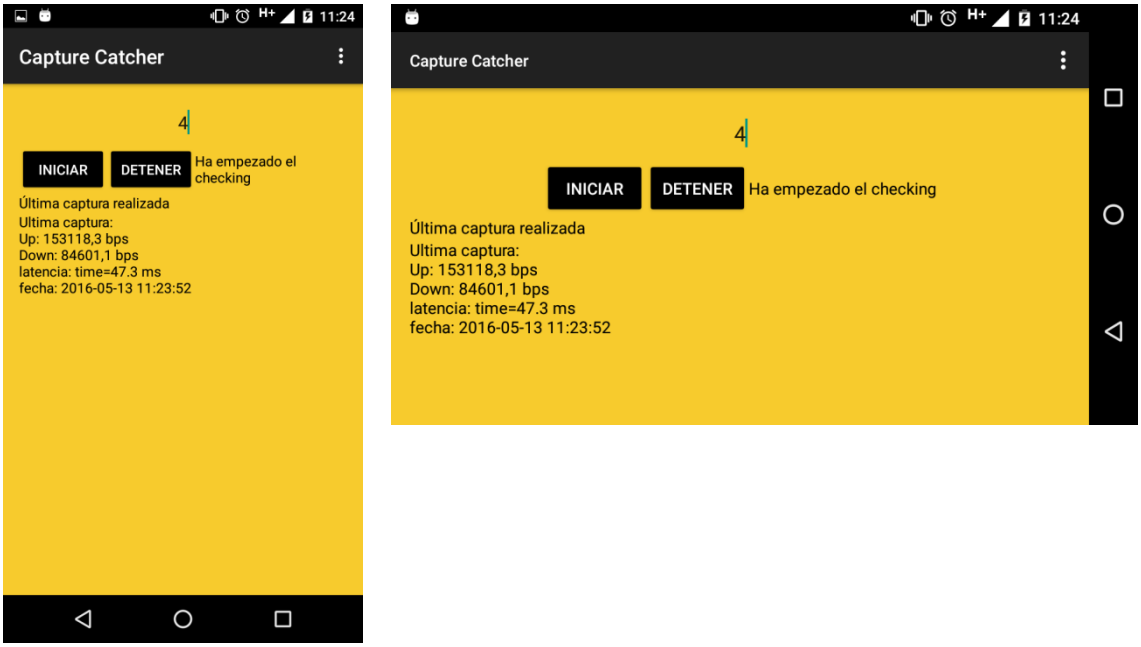

Figura 12: Aplicación vertical y apaisada

Como vemos, la información se guarda perfectamente. Este mismo ejemplo justifica que cuando volvamos a abrir la aplicación tras haberla iniciado, mostrará, también, la información correcta.

5.2 Estudio de potencia

Una vez terminado el estudio de funcionamiento, realizaremos un estudio de potencia de la aplicación. Al tratarse de una aplicación pensada para funcionar de fondo, mientras se le da un uso cotidiano al móvil, es conveniente estudiar el consumo de batería que realiza. Principalmente, el consumo vendrá relacionado con el periodo de actividad de la aplicación; cuantas más veces tenga que comprobar el estado de la red en un mismo intervalo de tiempo, más rápido decrecerá el nivel de batería. Es importante comentar que durante el proceso de simulación el terminal efectúa otros procesos (relacionados con otras aplicaciones) que pueden influir en el estado de batería, así que no se puede conseguir una medida totalmente exacta. Por otro lado, no habrá ningún tipo de uso del terminal por parte del usuario mientras se realiza el estudio. El estudio de potencia va consistir de la siguiente manera:

- Se fijará un periodo de simulación
- Se fijará un estado inicial y un estado final de batería
- Se medirá el tiempo que tarda en descargase ese porcentaje para ese periodo de simulación

El porcentaje de batería que utilizaremos para el estudio será desde el 100% hasta el 95%, es decir un 5%. Pues bien, veremos ahora una tabla con las mediciones realizadas:

Periodo simulación (s)	Hora inicial	Hora final	Tiempo (en min)	Núm. capturas	Tasa de captura
5	19:34	20:45	71	852	0,20
4	19:08	20:44	96	1440	0,25
3	17:58	19:19	81	1620	0,33
2	18:33	19:34	61	1830	0,50

En primer lugar se hizo la prueba con 10 segundos como periodo, pero el tiempo de descarga fue alrededor de 2,5 horas, así que se redujo a 4 periodos cortos para facilitar la simulación. Además, esto ya adelantaba que el gasto de batería no iba a ser muy alto.

Como vemos en la tabla superior, todas las simulaciones han tardado más de una hora en descargarse desde el 100% hasta el 95%. Con esto podemos ver que se trata de un consumo muy pequeño que, por otro lado, es lo adecuado para aplicaciones pensadas para funcionar de fondo mientras se usa el terminal para otras funciones.

6 Conclusiones y líneas de trabajo futuras

Este Trabajo Final de Grado planteaba el problema de desarrollar una aplicación Android que capturara el estado de red del usuario y lo mandara a una base de datos en un servidor que simulara al operador de red para facilitarle a este la calidad de servicio en sus clientes.

Para conseguir esto se ha explicado cómo generar un servicio que realice tareas de fondo de manera asíncrona en Android. Además, se ha implementado un servidor y una base de datos que simulaban al operador de red y se ha diseñado un pequeño aspecto gráfico implementando Material Design. Por último, se ha realizado un estudio de funcionamiento en el que se ha comprobado tanto el ciclo de vida de la aplicación como el consumo de batería que genera.

Mediante todo esto, hemos podido comprobar que Android es una herramienta apta para poder resolver el problema que se nos planteaba. Si bien existen otras maneras de desarrollar un algoritmo que realice la misma función, hemos conseguido que con unas pocas utilidades de Android, como AsyncTask, podamos realizar un primer acercamiento al desarrollo de este tipo de aplicaciones.

Una buena continuación de este trabajo sería añadir más campos en las capturas que al operador móvil le pudieran interesar o elaborar estadísticas y gráficos a partir de los resultados obtenidos. Además, durante el desarrollo de la base de datos y del servidor, se ha visto que se planteaban métodos que serían usados por los usuarios en el caso en que estos quisieran acceder a la propia base de datos.

Otra buena continuación que podría mejorar la aplicación sería vincularla con Google Maps. En este caso, si se combina el envío de capturas periódicas con un recorrido, el operador podría evaluar la calidad de servicio no solo de un usuario sino de localidades enteras.

7 Bibliografía

[1]. J. Monserrat, et al, "Rethinking the mobile and wireless network architecture: The METIS research into 5G", in European Conference on Networks and Communications (EuCNC), pp. 1-5, June 2014.

[2]. 5G-PPP, "The 5G Infrastructure Public Private Partnership: the next generation of communication networks and services", March 2015, available at http://5g-ppp.eu/wp-content/uploads/2015/02/5G-Vision-Brochure-v1.pdf .

[3]. Jose F. Monserrat and Mikael Fallgren (Editors)," Report on simulation results and evaluations" ICT-317669 METIS Deliverable 6.5, March 2015.

[4]. Z. Yingxiao, Z. Ying Jun, "User-centric virtual cell design for Cloud Radio Access Networks" IEEE Signal Processing Advances in Wireless Communications (SPAWC), pp.249-253, June 2014.

[5]. J. F. Monserrat, G. Mange, V. Braun, H. Tullberg, G. Zimmermann4 and Ö. Bulakci, "METIS research advances towards the 5G mobile and wireless system definition", EURASIP Journal on Wireless Communications and Networking 2015, 2015:53.

[6]. F. Boccardi, R.W. Heath, A. Lozano, T.L. Marzetta, P. Popovski, "Five disruptive technology directions for 5G", IEEE Communications Magazine, vol.52, no.2, pp.74,80, February 2014.

[7]. P. Agyapong, M. Iwamura, D. Staehle, W. Kiess, A. Benjebbour, "Design considerations for a 5G network architecture", IEEE Communications Magazine, vol.52, no.11, pp.65,75, Nov. 2014.

[8]. Nokia Siemens Networks, Acquisition and Retention White Paper, 2013. http://networks.nokia.com/sites/default/files/document/acquisition_retention_white_paper.pdf .

[9]. D.Z. Yazti, S. Krishnaswamy, "Mobile Big Data Analytics: Research, Practice, and Opportunities", IEEE 15th International Conference on Mobile Data Management (MDM), July 2014.

[10]. R. Kreher, UMTS Performance Measurement: A Practical Guide to KPIs for the UTRAN Environment, Wiley, July 2006.

[11]. S. Mehrotra, "On the implementation of a primal-dual interior point method", SIAM J Optim, 2, 575, 601, 1992.

[12]. V. Osa, J. Matamales, J. Monserrat, and J. Lopez, "Localization in wireless networks: The potential of triangulation techniques", Wireless Personal Communications, vol. 68, no. 4, pp. 1525, 1538, 2013.

[13]. Joan Ribas Lequerica, "Desarrollo de aplicaciones para Android. Edición 2016. Incluye *wearables* y Material Design", ANAYA Multimedia, 2015

[14]. W. Frank Ableson, Robi Sen, Chris King, "Android, Guía para desarrolladores", ANAYA Multimedia, 2012

[15]. Roberto Montero Miguel, "Desarrollo de aplicaciones para Android", Ra-Ma, 2012.

[16]. Biblioteca oficial para desarrolladores Android:

i. BufferdReader, https://developer.android.com/reference/java/io/BufferedReader.html?hl=es

ii. Calendar, https://developer.android.com/reference/java/util/Calendar.html?hl=es

iii. Common Intents, http://developer.android.com/intl/es/guide/components/intents-common.html

iv. ConnectivityManager, http://developer.android.com/intl/es/reference/android/net/ConnectivityManager.html

v. ConnectivityManager.NetworkCallback, http://developer.android.com/intl/es/reference/android/net/ConnectivityManager.NetworkCallback.html

vi. Date, https://developer.android.com/reference/java/util/Date.html?hl=es
vii. FileOutputStream, http://developer.android.com/intl/es/reference/java/io/FileOutputStream.html
viii. Intent, http://developer.android.com/intl/es/reference/android/content/Intent.html
ix. IntentService, http://developer.android.com/intl/es/reference/android/app/IntentService.html
x. LinkProperties, http://developer.android.com/intl/es/reference/android/net/LinkProperties.html
xi. Log, https://developer.android.com/reference/android/util/Log.html?hl=es
xii. Network, http://developer.android.com/intl/es/reference/android/net/Network.html
xiii. NetworkCapabilities, http://developer.android.com/intl/es/reference/android/net/NetworkCapabilities.html
xiv. NetworkInfo, http://developer.android.com/intl/es/reference/android/net/NetworkInfo.html
xv. Notification, http://developer.android.com/intl/es/reference/android/app/Notification.html
xvi. Notification.Builder, http://developer.android.com/intl/es/reference/android/app/Notification.Builder.html
xvii. NotificationCompat.Builder, http://developer.android.com/intl/es/reference/android/support/v4/app/NotificationCompat.Builder.html
xviii. NotificationManager, http://developer.android.com/intl/es/reference/android/app/NotificationManager.html
xix. PendingIntent, http://developer.android.com/intl/es/reference/android/app/PendingIntent.html
xx. Process, https://developer.android.com/reference/java/lang/Process.html?hl=es
xxi. ProcessBuilder, http://developer.android.com/intl/es/reference/java/lang/ProcessBuilder.html
xxii. Runtime, http://developer.android.com/intl/es/reference/java/lang/Runtime.html
xxiii. Spinners, https://developer.android.com/guide/topics/ui/controls/spinner.html?hl=es
xxiv. TelephonyManager, https://developer.android.com/reference/android/telephony/TelephonyManager.html?hl=es
xxv. Time, https://developer.android.com/reference/android/text/format/Time.html?hl=es
xxvi. Timer, https://developer.android.com/reference/java/util/Timer.html?hl=es
xxvii. TimerTask, https://developer.android.com/reference/java/util/TimerTask.html?hl=es
xxviii. TrafficStats, https://developer.android.com/reference/android/net/TrafficStats.html?hl=es

[17]. Jesús Tomás, “Android Programación Aplicaciones”, lista de reproducción en YouTube de la Universitat Politècnica de València, con enlace: https://www.youtube.com/playlist?list=PL6kQim6ljTJvgZ8RiRSA5zUfpOjY6NYDv