Capítulo 13
Conexión por red

Resumen: En este capítulo aprenderemos cómo podemos indagar sobre el estado de conectividad de Android, y cómo conseguir que el sistema nos informe ante algún cambio. También realizaremos un cliente y un servidor sencillos utilizando el API básico de conexión con sockets, y terminaremos haciendo una aplicación que hace uso de un servicio web.

Práctica 13.1: Estado de la red

La conectividad a la red en Android se puede conseguir a través de diferentes adaptadores, dependiendo del dispositivo y del momento. Aunque en principio el sistema operativo da soporte para Ethernet e incluso WiMAX, la mayor parte de los dispositivos existentes basan su conectividad en wi-fi y redes móviles (como 3g o 4g).

Para obtener información sobre el estado de la conectividad del dispositivo se hace uso del servicio del gestor de conectividad, ConnectivityManager. A él le podemos preguntar sobre el estado de conexión de cada uno de los diferentes tipos de adaptadores soportados, así como el predefinido, que será el que se utilice para encaminar tráfico. Para eso, la aplicación requiere el permiso ACCESS_NETWORK_STATE

En esta práctica vamos a colocar varias etiquetas en la actividad principal, en las que mostraremos el estado de conectividad del dispositivo.

1. Crea un nuevo proyecto y llámalo DM2E.estadodelared.
2. Dado que querremos obtener el estado de la red, modifica el fichero de manifiesto para solicitar el permiso.

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

1. En el layout, queremos mostrar tres líneas. La primera hará referencia al estado de la conexión por wi-fi, la segunda a la conexión móvil (3g/4g), y la tercera a cuál es la predefinida. Pondremos tres etiquetas en un LinearLayout, y las construiremos aprovechando cadenas con formato.

```
<LinearLayout ...
android:orientation="vertical">
<TextView ...
android:id="@+id/wifi"
android:text="@string/wifi"/>
<TextView ...
android:id="@+id/redMovil"
android:text="@string/redMovil"/>
<TextView ...
android:id="@+id/estadoRed"
android:layout_marginTop="6dp"/>
</LinearLayout>
```

2. Crea las cadenas asociadas a las etiquetas, como por ejemplo:

wifi: Wi-fi: %1$s
redMovil: Red móvil: %1$s
hayRed: Hay conectividad de red (%1$s)
noHayRed: No hay conectividad de red

3. Por comodidad, crea un método para establecer el texto de una etiqueta.

```
protected void setLabel(int id, String str) {
TextView tv = (TextView) findViewById(id);
tv.setText(str);
}
```

4. Crea un método para actualizar las etiquetas en función del estado de red devuelto por el ConnectivityManager.

```
protected void updateNetworkType(NetworkInfo ni,
NetworkInfo predefinida,
int labelId, int textId) {
String template = getString(textId);
String str = String.format(template, ni.getState());
if ((predefinida != null) &&
ni.getType() == predefinida.getType())
str += " (*)";
setLabel(labelId, str);
}
protected void updateNetworkStatus() {
ConnectivityManager cm = (ConnectivityManager)
getSystemService(Context.CONNECTIVITY_SERVICE);
NetworkInfo predefinida = cm.getActiveNetworkInfo();
NetworkInfo ni;
ni = cm.getNetworkInfo(ConnectivityManager.TYPE_WIFI);
updateNetworkType(ni, predefinida, R.id.wifi, R.string.wifi);
ni = cm.getNetworkInfo(ConnectivityManager.TYPE_MOBILE);
updateNetworkType(ni, predefinida, R.id.redMovil,
R.string.redMovil);
String str;
if ((predefinida != null) && predefinida.isConnected()) {
// Hay conexión.
str = getString(R.string.hayRed);
str = String.format(str, predefinida.getTypeName());
}
else
str = getString(R.string.noHayRed);
setLabel(R.id.estadoRed, str);
}
```

5. En el onCreate() llama a updateNetworkStatus().
6. Prueba la práctica en un móvil, con diferentes configuraciones de la red.

Práctica 13.2: Notificaciones del cambio de estado

El gestor de conectividad envía notificaciones de difusión al sistema cuando cambia el estado de la red. Las aplicaciones pueden registrar de manera estática (en su fichero de manifiesto) un receptor, de forma que cuando cambie el estado el receptor será lanzado

independientemente de que lo estuviera algún otro componente de la aplicación. Por ejemplo, un cliente de correo podría querer registrar un receptor de mensajes para ser lanzado cuando se detecte un cambio. Si éste ha sido para conseguir conexión, podría consultar si hay mensajes nuevos, y lanzar el proceso de consulta periódica.

No obstante, salvo en casos específicos, no se recomienda hacerlo así. Lo normal será que la aplicación quiera saber si cambia el estado de conectividad mientras esté activa, sin preocuparle si cambia cuando el usuario no la haya lanzado: a un navegador no le importa saber que vuelve a haber red si el usuario no lo está usando. Para evitar que el receptor se lance continuamente, es preferible registrarlo y desregistrarlo de manera dinámica de manera similar a como hicimos en la práctica 5.2.

1. Haz una copia de la práctica anterior.
2. Define un atributo networkMonitor que sea un objeto de una subclase de BroadcastReceiver. En el método onReceive(), que será llamado por el sistema cuando cambie la configuración de red del dispositivo, llama al método updateNetworkState() que hemos programado antes. Puedes hacerlo todo a la vez utilizando una clase anónima.

```
BroadcastReceiver networkMonitor = new BroadcastReceiver() {
@Override
public void onReceive(Context context, Intent intent) {
updateNetworkStatus();
}
};
```

3. En el método onCreate() registra el objeto como receptor de mensajes, interesándose por los cambios de configuración de red.

```
protected void onCreate(Bundle savedInstanceState) {
...
IntentFilter filter = new IntentFilter(
ConnectivityManager.CONNECTIVITY_ACTION);
registerReceiver(networkMonitor, filter);
}
```

4. En el onDestroy(), desregistralo para no tener fugas de contextos.

```
@Override
protected void onDestroy() {
super.onDestroy();
unregisterReceiver(networkMonitor);
}
```

5. Ejecuta la práctica, modificando, con ella lanzada, la configuración de red. Comprueba que se actualiza el el estado correctamente.

Práctica 13.3: Cliente de sumas

En esta práctica vamos a crear un cliente de red sencillo en Android. Parapoder hacerlo, necesitaremos antes un servidor al que conectarnos. Vamos a empezar creando el servidor, directamente en Java. El servidor se quedará escuchando en un puerto TCP (en el

9898) y esperará recibir "clientes de sumas". El cliente le enviará dos números nada más conectarse, el servidor contestará con el resultado de sumar ambos y cerrará la conexión.

Para eso, el servidor crea un server socket, y entra en un bucle infinito a la espera de clientes. Cada vez que llegue uno leerá una línea con los dos números y devolverá la salida.

1. Crea un programa en Java que implemente el servidor.

```
import java.net.*;
import java.io.*;
import java.util.Scanner;
public class ServidorSumas {
public static final int SERVER_PORT = 9898;
public static void main(String[] args) {
try {
ServerSocket ss = new ServerSocket(SERVER_PORT);
while(true) {
System.out.println("Esperando un cliente...");
Socket cliente = ss.accept();
Scanner sc = new Scanner(cliente.getInputStream());
PrintStream ps = new PrintStream(
cliente.getOutputStream());
int a, b;
a = sc.nextInt();
b = sc.nextInt();
System.out.println(a + " + " + b + " = " + (a+b));
ps.println(a+b);
sc.close();
ps.close();
} // while
} catch(IOException e) {
System.out.println(e);
}
} // main
}
```

2. Compila el servidor y ejecútalo.
3. Pruébalo realizando una conexión manual con cualquier cliente de red sencillo. Escribe dos números, y deberías ver el resultado.
4. Deja lanzado el servidor.

Queremos que el cliente en Android se conecte al servidor cada segundo, le envíe dos números aleatorios y muestre al usuario el resultado. Además, queremos que sea cuidadoso con el estado de la red para que no se conecte si no hay conexión, y que sea educado y cuando la actividad pierda el foco también deje de hacer las conexiones.

5. Haz una copia de la práctica anterior.
6. Renombra el paquete principal y llamalo DM2E.sumador.
7. Modifica el layout para que tenga una etiqueta en la parte superior de la ventana en la que escribiremos información sobre las sumas, y una en la parte inferior donde mostraremos el estado de la conexión. Utiliza un RelativeLayout.

```
<RelativeLayout ...>
<TextView
```

```
android:id="@+id/resultado"
android:text="@string/hebraDetenida"
android:layout_width="wrap_content"
android:layout_height="wrap_content" />
<TextView
android:id="@+id/estadoRed"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:layout_alignParentBottom="true"/>
</RelativeLayout>
```

8. Modifica las cadenas disponibles. Cambia el título de la aplicación por Sumas, manten las cadenas hayRed y noHayRed, y borra el resto. Añade una hebraDetenida con el texto Hebra cliente detenida.
9. Crea dos constantes en la clase con la IP y puertos del servidor. Ten en cuenta que en una aplicación real permitiríamos al usuario configurar estos valores desde la propia aplicación, y los guardaríamos en las preferencias.

```
protected static final String SERVER_IP = "10.0.2.2";
protected static final int SERVER_PORT = 9898;
```

10. Crea una nueva clase ClienteSumas que se encargue del cliente de red. No vamos a hacerla que herede de AsyncTask porque impediríamos la ejecución de cualquier otra tarea en la aplicación, dado que pretendemos que el cliente dure por siempre. Cosas importantes:

- Aunque sea un "cliente infinito", la hebra debe ser interrumpible. Si desde fuera se le solicita que termine, debe hacerlo limpiamente.
- Queremos que muestre la cadena Conectando... cuando intente el inicio de conexión al servidor.
- Cuando se conecte, queremos que elija dos números aleatorios, los mande al servidor y espere el resultado.
- Cuando lo recoja, queremos ver la suma solicitada y el resultado en la etiqueta superior.
- Si se produce cualquier error, queremos verlo.
- Queremos que espere un segundo entre solicitud y solicitud.
- Necesitamos un atributo de la clase para guardar la hebra actual y poderla manipular.

```
class ClienteSumas extends Thread {
TextView tv;
@Override
public void run() {
tv = (TextView) findViewById(R.id.resultado);
while(!isInterrupted()) {
try {
setText("Conectando...");
Socket socket = new Socket(SERVER_IP, SERVER_PORT);
int a, b;
String result;
a = (int) (Math.random() * 1000); // 0..999
b = (int) (Math.random() * 1000);
PrintStream ps = new PrintStream(
```

```
socket.getOutputStream());
Scanner sc = new java.util.Scanner(
socket.getInputStream());
ps.println("" + a + " " + b);
result = sc.next();
setText("" + a + " + " + b + " = " + result);
socket.close();
}
catch (Exception e) {
if (!isInterrupted())
setText("ERROR: " + e);
}
try {
Thread.sleep(1000);
}
catch(InterruptedException ie) {
interrupt();
}
} // while no haya que parar
setText(R.string.hebraDetenida);
android.util.Log.i("ClienteSumador", "Hebra detenida");
} // run
protected void setText(final String s) {
runOnUiThread(new Runnable() {
@Override
public void run() {
tv.setText(s);
}
});
}
protected void setText(final int stringId) {
runOnUiThread(new Runnable() {
@Override
public void run() {
tv.setText(stringId);
}
});
}
} // ClienteSumas
ClienteSumas _cliente;
```

11. El atributo _cliente será null cuando no tengamos la hebra lanzada (porque estamos suspendidos o porque no hay conexión a la red disponible). Cuando tengamos la hebra lanzada y queramos pararla, tendremos que solicitar su detención y borrar el atributo. Haz un método detenerHebraCliente() que lo haga.

```
protected void detenerHebraCliente() {
if (_cliente != null) {
_cliente.interrupt();
_cliente = null;
}
}
```

12. Para ser mejores "ciudadanos", el broadcast receiver vamos a registrarlo en la pareja de métodos onStart() y onStop(). Así cuando la actividad no sea visible no nos preocuparemos de los cambios de red. Crea ambos métodos, llama en cada caso al equivalente en la clase padre, y mueve el registro y desregistro del broadcast receiver. Además, cuando nos suspendamos tendremos que detener la hebra cliente.

```
@Override
protected void onStart() {
super.onStart();
IntentFilter filter = new IntentFilter(
ConnectivityManager.CONNECTIVITY_ACTION);
registerReceiver(networkMonitor, filter);
}
@Override
protected void onStop() {
super.onStop();
unregisterReceiver(networkMonitor);
detenerHebraCliente();
}
```

13. Modifica el método updateNetworkStatus() para que no se preocupe de las etiquetas de los detalles sobre qué tipo de red es la disponible. Lo único que nos interesa es la parte final, mirando si tenemos o no conectividad. Si la tenemos, lanzamos la hebra, si no lo está ya, y si no, la detenemos.

```
protected void updateNetworkStatus() {
ConnectivityManager cm = (ConnectivityManager)
getSystemService(Context.CONNECTIVITY_SERVICE);
NetworkInfo predefinida = cm.getActiveNetworkInfo();
String str;
if ((predefinida != null) && predefinida.isConnected()) {
// Hay conexión.
str = getString(R.string.hayRed);
str = String.format(str, predefinida.getTypeName());
if (_cliente == null) {
_cliente = new ClienteSumas();
_cliente.start();
}
}
else {
str = getString(R.string.noHayRed);
detenerHebraCliente();
}
setLabel(R.id.estadoRed, str);
}
```

14. Añadir permiso de internet.

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

15. Prueba la aplicación, y comprueba que funciona. Prueba a detener el servidor, anular la conectividad de red u ocultar la aplicación y comprueba que funciona bien.

Práctica 13.4: Servidor de sumas

1. Crea un nuevo proyecto.
2. En el activity_main crea dos botones.

```
<Button
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="@string/LanzarServicio"
    android:onClick="onLanzarServicio"
    />

<Button
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="@string/pararServicio"
    android:onClick="onPararServicio"
    />
```

3. Añade, en MainActivity, los dos métodos de los botones.

```
public void onLanzarServicio(View v) {

    startService(new Intent(this, ServidorSumas.class));
}

public void onPararServicio(View v) {
    stopService(new Intent(this, ServidorSumas.class));
}
```

4. Crea una nueva clase ServidorSumas que herede de Service

```
public class ServidorSumas extends Service {
```

5. Crea un constructor vacío.
6. Declara la constante del puerto.

```
private static final int SERVER_PORT = 9898;
```

7. Crea una nueva clase interna HebraServidora que herede de Thread. Para el método run() copia el código del servidor de la práctica anterior que ejecutamos directamente en Java. Sustituye los System.out.println por android.util.Log.i(TAG, ...).
8. ¿Consideras razonable que la clase anterior sea interna a ServidorSuma?
9. Modifica la clase para que sea interrumpible.
10. Crea un atributo de la clase anterior, y llámalo _hebraServidora.

```
class HebraServidora extends Thread {

    @Override
    public void run() {
        try {
            ServerSocket ss = new ServerSocket(SERVER_PORT);
```

```
            while(!isInterrupted()) {
                android.util.Log.i(TAG, "Esperando un cliente...");
                Socket cliente = ss.accept();
                Scanner sc = new Scanner(cliente.getInputStream());
                PrintStream ps = new
PrintStream(cliente.getOutputStream());
                int a, b;
                a = sc.nextInt();
                b = sc.nextInt();
                android.util.Log.i(TAG, a + " + " + b + " = " + (a +
b));
                ps.println(a+b);
                sc.close();
                ps.close();
            }
        } catch(IOException e) {
            android.util.Log.i(TAG, e.toString());
        }
        _hebraServidora = null;
    }

} // HebraServidora

HebraServidora _hebraServidora;

protected final String TAG = getClass().getSimpleName();
```

11. En el método onCreate(), clase principal, crea la hebra y lánzala.

```
@Override
public void onCreate()
{
    android.util.Log.i(TAG, "onCreate()");
    if (_hebraServidora == null) {
        _hebraServidora = new HebraServidora();
        _hebraServidora.start();
    }
}
```

12. En el método onDestroy() detenla.

```
public void onDestroy()
{
    android.util.Log.i(TAG, "onDestroy()");
    if (_hebraServidora != null)
        _hebraServidora.interrupt();
    _hebraServidora = null; // Superfluo...
}
```

13. Añade el método OnBind.

```
@Override
public IBinder onBind(Intent intent) {
    android.util.Log.i(TAG, "onBind(" + intent + ")");
    throw new UnsupportedOperationException("Unsupported");
}
```

14. Si el servicio estaba lanzado y Android se ve obligado a destruirlo, querríamos que se volviera a lanzar. En onStartCommand(), devuelve START_STICKY.

```
@Override
public int onStartCommand(Intent intent, int flags, int startId) {
    android.util.Log.i(TAG, "onStartCommand(" + intent + "," + flags +
", " + startId + ")");
    return START_STICKY;
}
```

15. Lanza la aplicación en un AVD, y pulsa el botón de "Lanzar servicio".
16. Usando telnet conéctate al puerto de control del AVD y redirige el puerto 9898 del AVD al puerto 9000 del anfitrión. .

- telnet localhost 5554
- auth numero-fichero. '/Users/XXXXX/.emulator_console_auth_token'
- redir add tcp:9000:9898

17. Usando telnet conéctate al puerto 9000 del anfitrión y hazle una consulta. Comprueba que responde correctamente.

telnet localhost 9000

18. Vuelve a la conexión de configuración del AVD y elimina la redirección. Si no lo has hecho ya, para el servidor en Java (que estará escuchando en el puerto 9898 del anfitrión), y añade una redirección nueva para que el puerto 9898 del AVD se mapee al puerto 9898 del servidor.

redir del tcp:9000

19. Lanza un segundo AVD y ejecuta en él la práctica anterior. Comprueba que funciona.

En su estado actual, el servidor tiene muchos puntos de mejora. Algunas cuestiones:

- Conéctate manualmente al servidor (con telnet, como hicimos para probarlo antes en el punto 13) y escribe algo que no encaje con el protocolo. ¿Qué ocurre? ¿Cómo lo resolverías?
- Si la conectividad de red se pierde, ¿qué ocurre? ¿Qué mejorarías y cómo?
- Si quisiéramos que el servidor se lanzara automáticamente cuando hubiera conexión de red, ¿qué harías?
- Si el servidor no fuera tan rápido y quisiéramos tener múltiples clientes, ¿cómo lo harías?

Práctica 13.5: Viabicing

En esta práctica vamos a conectarnos a un servicio web del Ayuntamiento de Barcelona, que pone a disposición de las aplicaciones la localización y estado de sus "estaciones" de aparcamiento de bicicletas públicas. El servicio web está accesible en http://wservice.viabicing.cat/v1/getstations.php?v=1.
https://api.bicing.cat/v1/stations.xml

Las anteriores no funcionan cogemos las de Madrid.

https://datos.madrid.es/datosabiertos/CIUAB/MINT/APARCABICIS/2025/01/aparcabicis202501.xml

1. Crea un nuevo proyecto y llámalo Viabicing.
2. Añádele los permisos de monitorizar la red y acceder a Internet.

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

3. Crea una constante con la URL del servicio web que vamos a utilizar.

```
protected static final String WEB_SERVICE_URL =
"http://wservice.viabicing.cat/v1/getstations.php?v=1";
```

4. Como en prácticas anteriores, vamos a querer detectar cuándo se produce un cambio en el estado de la red. Crea el broadcast receiver habitual. En lugar de utilizar el método updateNetworkStatus() que hemos estado utilizando, mete directamente en la clase anónima que se mire si se ha pasado a tener o a perder la conexión, y que llame a dos métodos nuevos, onNetworkOn() y onNetworkOff() según el caso. Registra y desregistra el receptor de mensajes en el onStart() y onStop(). Además, cuando la aplicación pierda el foco no queremos que utilice la red, por lo que simularemos que se ha desconectado.

```
@Override
protected void onStart() {
super.onStart();
IntentFilter filter = new IntentFilter(
ConnectivityManager.CONNECTIVITY_ACTION);
registerReceiver(networkMonitor, filter);
}
@Override
protected void onStop() {
super.onStop();
unregisterReceiver(networkMonitor);
onNetworkOff();
}
protected void onNetworkOn() {
}
protected void onNetworkOff() {
}
BroadcastReceiver networkMonitor = new BroadcastReceiver() {
@Override
public void onReceive(Context context, Intent intent) {
ConnectivityManager cm = (ConnectivityManager)
getSystemService(Context.CONNECTIVITY_SERVICE);
NetworkInfo predefinida = cm.getActiveNetworkInfo();
if ((predefinida != null) && predefinida.isConnected())
// Hay conexión.
onNetworkOn();
else
onNetworkOff();
```

```
}
}; // networkMonitor
```

5. Cuando detectemos que la red está activa, vamos a lanzar una tarea a través de una AsyncTask, que se descargará del servicio web la información. Vamos a querer pasarle desde la hebra principal la URL donde está el servicio web, por lo que el tipo para el primer argumento de la clase genérica será String. Las otras de momento las dejaremos en Void. Crea la clase vacía por el momento, y un atributo con ese tipo.

```
class ObtenerBicis extends AsyncTask<String, Void, Void> {
@Override
protected Void doInBackground(String... params) {
return null;
}
@Override
protected void onPostExecute(Void aVoid) {
}
} // ObtenerBicis
ObtenerBicis _asyncTask;
```

6. Cuando se detecte la disponibilidad de la red, crea, si no lo está ya, la AsyncTask y lánzala. Cuando se detecte la desconexión, cancela la. Además, para mantener correctamente el atributo de la clase, en el método onPostExecute() de la tarea asíncrona ponlo a null para marcar que ha terminado.

```
protected void onNetworkOn() {
if (_asyncTask == null) {
_asyncTask = new ObtenerBicis();
_asyncTask.execute(WEB_SERVICE_URL);
}
}
protected void onNetworkOff() {
if (_asyncTask != null) {
_asyncTask.cancel(true);
_asyncTask = null;
}
}
class ObtenerBicis extends AsyncTask<String, Void, Void> {
...
protected void onPostExecute(Void aVoid) {
_asyncTask = null;
}
...
}
```

7. En la tarea asíncrona vamos a conectarnos al servicio web y obtener el XML. Para eso, utilizamos las clases URL y HttpURLConnection, de la librería estándar de Java. En las primeras versiones de Android la implementación tenía muchos errores y lo habitual era utilizar la librería de Apache para conexiones HTTP, pero desde Android 2.3 Google recomienda hacer uso de HttpURLConnection. Por comodidad, vamos a hacer un método nuevo en la ObtenerBicis que realice la conexión y devuelve un InputStream del que se puede obtener la respuesta.

```
private InputStream downloadUrl(String urlString)
throws IOException {
URL url = new URL(urlString);
HttpURLConnection httpCon;
httpCon = (HttpURLConnection) url.openConnection();
httpCon.setRequestMethod("GET");
httpCon.setDoInput(true);
httpCon.connect();
return httpCon.getInputStream();
}
```

8. Para probar que la lectura se realiza correctamente, vamos a llamarlo desde doInBackground() y vamos a escribir todo el contenido de la cadena en el log.

```
try {
InputStream is = downloadUrl(params[0]);
java.util.Scanner s;
s = new java.util.Scanner(is);
while (s.hasNext())
android.util.Log.i(TAG, s.nextLine());
s.close();
} catch (IOException e) {
e.printStackTrace();
}
```

9. Declara en la clase de la actividad el atributo TAG convenientemente.
10. Comprueba que todo funciona correctamente.

La cadena devuelta por el servidor es un XML que tenemos que analizar y guardarlo de manera cómoda para obtener por código la información. Si analizas el valor devuelto es fácil intuir la estructura:

```
<?xml version="1.0" encoding="UTF-8"?>
<bicing_stations>
<updatetime> tiempo unix </updatetime>
<station>
<id>1</id>
<type>BIKE</type>
<lat>41.397952</lat>
<long>2.180042</long>
<street>...</street>
<height>...</height>
<streetNumber>...</streetNumber>
<nearbyStationList>...</nearbyStationList>
<status>...</status>
<slots>...</slots>
<bikes>...</bikes>
</station>
<station> .... </station>
</bicing_stations>
```

Si usamos el de Madrid:

```
<records>
<record>
<ID>65239</ID>
```

<DESC_CLASIFICACION>Aparcabicicletas</DESC_CLASIFICACION>
<COD_BARRIO>215</COD_BARRIO>
<BARRIO>CORRALEJOS</BARRIO>
<COD_DISTRITO>21</COD_DISTRITO>
<DISTRITO>BARAJAS</DISTRITO>
<ESTADO>OPERATIVO</ESTADO>
<COORD_GIS_X>449732.02</COORD_GIS_X>
<COORD_GIS_Y>4479526.49</COORD_GIS_Y>
<SISTEMA_COORD>ETRS89</SISTEMA_COORD>
<LATITUD>40.46489143</LATITUD>
<LONGITUD>-3.59293482</LONGITUD>
<TIPO_VIA>CALLE</TIPO_VIA>
<NOM_VIA>BAHIA DE POLLENSA</NOM_VIA>
<NUM_VIA>5</NUM_VIA>
<COD_POSTAL>28042</COD_POSTAL>
<DIRECCION_AUX>PORTAL</DIRECCION_AUX>
<NDP>31011498</NDP>
<FECHA_INSTALACION>2021-09-14T00:00:00</FECHA_INSTALACION>
<CODIGO_INTERNO>1034222</CODIGO_INTERNO>
<CONTRATO_COD>MU21</CONTRATO_COD>
<MODELO>MU-51</MODELO>
</record>
<record>
<ID>71328</ID>
<DESC_CLASIFICACION>Aparcabicicletas</DESC_CLASIFICACION>
<COD_BARRIO>207</COD_BARRIO>
<BARRIO>CANILLEJAS</BARRIO>
<COD_DISTRITO>20</COD_DISTRITO>
<DISTRITO>SAN BLAS - CANILLEJAS</DISTRITO>
<ESTADO>OPERATIVO</ESTADO>
<COORD_GIS_X>447951.56</COORD_GIS_X>
<COORD_GIS_Y>4477290.42</COORD_GIS_Y>
<SISTEMA_COORD>ETRS89</SISTEMA_COORD>
<LATITUD>40.44463800</LATITUD>
<LONGITUD>-3.61375175</LONGITUD>
<TIPO_VIA>CALLE</TIPO_VIA>
<NOM_VIA>BOLTAÑA</NOM_VIA>
<NUM_VIA>11</NUM_VIA>
<COD_POSTAL>28022</COD_POSTAL>
<DIRECCION_AUX>FRENTE FACHADA</DIRECCION_AUX>
<NDP>20151791</NDP>
<FECHA_INSTALACION>2021-09-16T00:00:00</FECHA_INSTALACION>
<CODIGO_INTERNO>1080237</CODIGO_INTERNO>
<CONTRATO_COD>MU21</CONTRATO_COD>
<MODELO>MU-51</MODELO>
</record>

No queremos conservar toda la información. Nos interesa el tiempo unix (última actualización de los datos), y para cada lugar, el identificador, la calle y número, y el número de huecos libres y de bicis. Crearemos dos clases, una para la información de cada localización, y otra para la lista de localizaciones y el instante de la última actualización.

1. Crea una nueva clase (en un fichero separado) que guarde la información de cada localización.

```
class Station {
public int id;
public String street;
public String streetNumber; // Cadena; a veces no hay ("")
public int slots;
public int bikes;
public String toString() {
StringBuilder sb = new StringBuilder();
sb.append(id).append(": ");
sb.append(street).append(", no ").append(streetNumber);
sb.append("; ").append(slots).append(" slots; ");
sb.append(bikes).append(" bikes");
return sb.toString();
}
}
```

2. Crea una nueva clase que guare una lista de localizaciones y el tiempo Unix de la última actualización.

```
class ViabicingInfo {
public void addStation(Station s) {
stations.add(s);
}
List<Station> stations = new ArrayList<Station>();
int timestamp = -1;
}
```

Para hacer el análisis del XML vamos a utilizar un XML Pull parser integrado con Android. Esta familia de analizadores facilita la programación a través de analizadores descendentes. La librería recorre, bajo demanda, cada elemento del XML. En cada paso le preguntamos qué viene a continuación, y actuamos en consecuencia. Así, por ejemplo, si nos encontramos un nuevo elemento de tipo station invocaremos a un método auxiliar encargado de analizar los elementos <station> y que devolverá un objeto Station.

Crea una nueva clase ViabicingXMLParser que contenga los métodos encargados del análisis, todos ellos estáticos.

```
public class ViabicingXMLParser {
private static final String ns = null;
public static ViabicingInfo parse(InputStream is)
throws XmlPullParserException, IOException {
try {
XmlPullParser parser = Xml.newPullParser();
parser.setFeature(
XmlPullParser.FEATURE_PROCESS_NAMESPACES, false);
parser.setInput(is, null); // Codificación predefinida.
parser.nextTag(); // Nos saltamos el "inicio"
return readStations(parser);
} finally {
is.close();
}
} // parse
//----------------------------------------------------------
private static ViabicingInfo readStations(XmlPullParser parser)
```

```
throws XmlPullParserException, IOException {
ViabicingInfo result = new ViabicingInfo();
parser.require(XmlPullParser.START_TAG, ns, "bicing_stations"); //cambiar por records
while (parser.next() != XmlPullParser.END_TAG) {
if (parser.getEventType() != XmlPullParser.START_TAG)
continue;
String name = parser.getName();
switch(name) {
case "station": //cambiar por record
result.addStation(readStation(parser));
break;
case "updatetime":  //no existe
if (result.timestamp != -1) {
// ¡¡Ya nos habíamos encontrado uno!!
// El XML está roto.
throw new XmlPullParserException("Two timestamps");
}
result.timestamp =
Integer.parseInt(readString(parser));
break;
default:
// ¿¿??
skip(parser);
}
}
return result;
} // readStations
//-------------------------------------------------------------
private static Station readStation(XmlPullParser parser)
throws XmlPullParserException, IOException {
parser.require(XmlPullParser.START_TAG, ns, "station");
Station result = new Station();
while (parser.next() != XmlPullParser.END_TAG) {
if (parser.getEventType() != XmlPullParser.START_TAG) {
continue;
}
String name = parser.getName();
switch(name) {
case "id": //case “ID”
result.id = Integer.parseInt(readString(parser));
break;
case "street": //NOM_VIA
result.street = readString(parser);
break;
case "streetNumber": //NUM_VIA
result.streetNumber = readString(parser);
break;
case "slots": //UN VALOR NUMERICO
result.slots = Integer.parseInt(readString(parser));
break;
case "bikes": //UN VALOR NUMERICO
result.bikes = Integer.parseInt(readString(parser));
break;
default:
```

```
skip(parser);
} // switch
} // while
return result;
} // readStation
//-----------------------------------------------------
private static String readString(XmlPullParser parser)
throws XmlPullParserException, IOException {
String result;
if (parser.next() == XmlPullParser.TEXT) {
result = parser.getText();
parser.nextTag();
return result;
}
else
return "";
} // readString
//-----------------------------------------------------
private static void skip(XmlPullParser parser)
throws XmlPullParserException, IOException {
if (parser.getEventType() != XmlPullParser.START_TAG)
throw new IllegalStateException();
int depth = 1;
while (depth != 0) {
switch (parser.next()) {
case XmlPullParser.END_TAG:
depth--;
break;
case XmlPullParser.START_TAG:
depth++;
break;
}
}
} // skip
} // ViabicingXMLParser
```

3. Modifica el método doInBackground() para que utilice la clase anterior para analizar el XML descargado y obtenga la información completa. Devuelve un objeto ViabicingInfo como resultado. Tendrás que ajustar convenientemente los tipos genéricos de la AsyncTask de la que estamos heredando, así como el prototipo de onPostExecute(). Para probar, escribe en el log la información recopilada.

```
protected ViabicingInfo doInBackground(String... params) {
ViabicingInfo viabicingInfo;
try {
InputStream is = downloadUrl(params[0]);
viabicingInfo = ViabicingXMLParser.parse(is);
android.util.Log.i(TAG, "Ultima actualizacion: " +
viabicingInfo.timestamp);
for (Station st: viabicingInfo.stations) {
android.util.Log.i(TAG, st.toString());
}
} catch (IOException e) {
android.util.Log.i(TAG, e.toString());
```

```
e.printStackTrace();
} catch (XmlPullParserException e) {
android.util.Log.i(TAG, e.toString());
e.printStackTrace();
}
return viabicingInfo;
}
```

Ya sólo nos falta enseñárselo al usuario.

1. Añade las cadenas habituales de Hay conectividad de red o No hay conectividad de red.

```
<string name="hayRed">Hay conectividad de red</string>
<string name="noHayRed">No hay conectividad de red</string>
```

2. Pon una etiqueta en la parte inferior de la ventana donde mostraremos el estado de la red.
3. Pon una etiqueta en la parte superior donde mostraremos la última actualización de los datos.
4. Pon en medio un ListView.

```
<RelativeLayout ...>
<TextView
android:id="@+id/ultimaActualizacion"
android:layout_width="match_parent"
android:layout_height="wrap_content"
android:layout_alignParentTop="true"/>
<ListView
android:id="@+id/listaLocalizaciones"
android:layout_width="match_parent"
android:layout_height="wrap_content"
android:layout_below="@+id/ultimaActualizacion"
android:layout_above="@+id/estadoRed"/>
<TextView android:id="@+id/estadoRed"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:layout_alignParentBottom="true"/>
</RelativeLayout>
```

5. Modifica los métodos onNetworkOn() y onNetworkOff() para que actualicen la etiqueta del estado de la red.

   ```
   ((TextView)findViewById(R.id.estadoRed)).setText(R.string.hayRed);

   ((TextView)findViewById(R.id.estadoRed)).setText(R.string.noHayRed);
   ```

6. Añade en el onPostExecute() todo el código para enviar a la lista los datos descargados.

```
protected void onPostExecute(ViabicingInfo viabicingInfo) {
TextView tvTimestamp = (TextView)
```

```
findViewById(R.id.ultimaActualizacion);
ListView lv = (ListView)
findViewById(R.id.listaLocalizaciones);
if (viabicingInfo == null) {
// Hubo error :(
[ Mensaje en la etiqueta ]
}
else {
ArrayAdapter<Station> aa;
aa = new ArrayAdapter(MainActivity.this,
android.R.layout.simple_list_item_1,
viabicingInfo.stations);
lv.setAdapter(aa);
Date time = new Date((long)viabicingInfo.timestamp*1000);
tvTimestamp.setText(time.toString());
}
_asyncTask = null;
}
```

1. Prueba la aplicación, y comprueba que aparece la lista de lugares.

Hay diferentes puntos de mejora:

- Vista vacía para la lista. Podríamos poner directamente una imagen animada de espera.
- Procesar la cancelación.
- Gestionar los errores y notificárselos al usuario.
- Actualizar correctamente el interfaz cuando se pierde la conectividad de red.
- Limpiar las etiquetas para que no salgan entidades HTML.

Notas bibliográficas

Curso: IFC02CM15 - Programación avanzada de dispositivos móviles - Julio 2015 Pedro Pablo Gómez Martín
http://developer.android.com/training/basics/network-ops/connecting.html
http://developer.android.com/training/basics/network-ops/managing.html
http://developer.android.com/training/basics/network-ops/xml.html
http://developer.android.com/training/connect-devices-wirelessly/index.html
http://developer.android.com/tools/help/emulator.html
http://developer.android.com/tools/devices/emulator.html