Développement Android : Activité n°2 (Communication réseau)

Thierry Vaira <tvaira@free.fr> http://tvaira.free.fr/

25/08/2016 (rev. 1)

Table des matières

Activité n°2 : Communication réseau

Objectif

L'objectif est d'écrire une application Android qui communique avec un serveur TCP/UDP en réseau pour commnder des E/S à distance.

En bonus : les notifications.

Pré-requis

Vous devez avoir réalisé :

1. l'activité n°1

Puis :

- Copier et renommer le dossier de la première application en `MyApplication2`
- Renommer l'attribut *package* dans `$HOME/AndroidStudioProjects/MyApplication2/app/src/main/AndroidManifest.xml`
- Renommer la valeur d'*applicationId* dans `/home/tv/AndroidStudioProjects/MyApplication2/app/build.gradle`
- Démarrer `Android Studio` et ouvrir le projet `MyApplication2`
- Faire 'Synchronize'
- Refactoriser le nom du *package* puis renommer la valeur d'*app_name* dans `strings.xml` si nécessaire

Connaissances de base

La communication par socket en Java

L'interface Java des *sockets* (package java.net) offre un accès simple aux sockets.

On l'utilise en plaçant au début du fichier : `import java.net.*;`

Les différentes classes

Plusieurs classes interviennent lors de la réalisation d'une communication par sockets :

- La classe `java.net.InetAddress` permet de manipuler des adresses IP.
- La classe `java.net.SocketServer` permet de programmer l'interface côté serveur en mode connecté.
- La classe `java.net.Socket` permet de programmer l'interface côté client et la communication par flot via les *sockets*.
- Les classes `java.net.DatagramSocket` et `java.net.DatagramPacket` permettent de programmer la communication en mode datagramme UDP en mode non connecté.

Remarque : si le compilateur émet des messages du genre "must catch exception", il faudra donc utiliser un bloc `try/catch` ainsi :

```
try
{
   ... // code
}
catch (Exception e)
{
   e.printStackTrace();
}
```

La classe InetAddress

Cette classe représente les adresses IP et un ensemble de méthodes pour les manipuler. Elle encapsule aussi la résolution des noms.

```
public static InetAddress getLocalHost() throws UnknownHostException
public static InetAddress getByName(String host) throws UnknownHostException
public static InetAddress[] getAllByName(String host) throws UnknownHostException

// obtient le nom complet correspondant à l'adresse IP
public String getHostName()

// obtient l'adresse IP sous forme %d.%d.%d.%d
public String getHostAddress()

// obtient l'adresse IP sous forme d'un tableau d'octets
public byte[] getAddress()
```

La classe Socket

Côté client, on utilise les constructeurs suivants :

```
public Socket(String host, int port) throws UnknownHostException, IOException
public Socket(InetAddress address, int port) throws IOException
public Socket(String host, int port, InetAddress localAddr, int localPort) throws UnknownHostException,
    IOException
public Socket(InetAddress addr, int port, InetAddress localAddr, int localPort) throws IOException
```

Les deux premiers constructeurs construisent une socket connectée à la machine et au port spécifiés. Par défaut, la connexion est de type **TCP** fiable. Les deux autres interfaces permettent en outre de fixer l'adresse IP et le numéro de port utilisés côté client (plutôt que d'utiliser un port disponible quelconque).

Ces constructeurs correspondent à l'utilisation des primitives `socket`, `bind` (éventuellement) et `connect`.

Remarque : Côté serveur, la méthode `accept()` de la classe `ServerSocket` renvoie une Socket de service connecté au client.

La classe ServerSocket

Ces constructeurs créent un objet serveur à l'écoute du port spécifié. La taille de la file d'attente des demandes de connexion peut être explicitement spécifiée via le paramètre *backlog*. Si la machine possède plusieurs adresses, on peut aussi restreindre l'adresse sur laquelle on accepte les connexions .

```
public ServerSocket(int port) throws IOException
public ServerSocket(int port, int backlog) throws IOException
public ServerSocket(int port, int backlog, InetAddress bindAddr) throws IOException
```

Ces constructeurs correspondend à l'utilisation des primitives `socket`, `bind` et `listen`.

La méthode essentielle est l'acceptation d'une connexion d'un client : `public Socket accept() throws IOException`

Cette méthode est bloquante, mais l'attente peut être limitée dans le temps par l'appel préalable de la méthode `setSoTimeout()`. Cette méthode prend en paramètre le délai de garde exprimé en millisecondes. La valeur par défaut 0 équivaut à l'infini. À l'expiration du délai de garde, l'exception `java.io.InterruptedIOException` est levée.

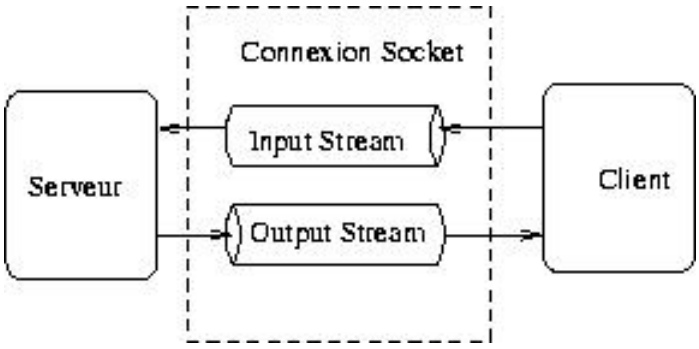
La fermeture du socket d'écoute s'exécute par l'appel de la méthode `close()`.

Enfin, les méthodes suivantes retrouvent l'adresse IP ou le port d'un socket d'écoute :

```
public InetAddress getInetAddress()
public int getLocalPort()
```

Émission-Réception TCP

La communication sur une connexion par socket utilise la notion de flots de données (`java.io.OutputStream` et `java.io.InputStream`).



Les deux méthodes suivantes sont utilisées pour obtenir les flots en entrée et en sortie :

```
public InputStream  getInputStream()  throws IOException
public OutputStream getOutputStream() throws IOException
```

Les flots obtenus servent de base à la construction d'objets de classes plus abstraites telles que `java.io.DataOutputStream` et `java.io.DataInputStream` (pour le JDK1), ou `java.io.PrintWriter` et `java.io.BufferedReader` (JDK2).

Une opération de lecture sur ces flots est bloquante tant que des données ne sont pas disponibles. Cependant, il est possible de fixer une délai de garde à l'attente de données : `public void setSoTimeout(int timeout) throws SocketException`.

Un ensemble de méthodes permet d'obtenir les éléments constitutifs de la liaison établie :

```
public InetAddress getInetAddress() : fournit l'adresse IP distante
public InetAddress getLocalAddress() : fournit l'adresse IP locale
public int getPort() : fournit le port distant
public int getLocalPort() : fournit le port local
```

L'opération `close()` ferme la connexion et libère les ressources du système associées à la socket.

Les classes DatagramSocket et DatagramPacket

La classe `DatagramSocket` permet d'envoyer et de recevoir des paquets (**datagrammes UDP**). Il s'agit donc de messages non fiables (possibilités de pertes et de duplication), non ordonnés (les messages peuvent être reçus dans un ordre différent de celui d'émission) et dont la taille (assez faible, souvent 4Ko) dépend du réseau.

```
public DatagramSocket() throws SocketException
public DatagramSocket(int port) throws SocketException
```

Émission-Réception UDP

La communication utilise les méthodes :

```
public void send(DatagramPacket p) throws IOException
public void receive(DatagramPacket p) throws IOException
```

Ces opérations permettent d'envoyer et de recevoir un paquet. Un paquet est un objet de la classe `DatagramPacket` qui possède une zone de données et (éventuellement) une adresse IP et un numéro de port (destinataire dans le cas `send`, émetteur dans le cas `receive`).

Les principales méthodes sont :

```
public DatagramPacket(byte[] buf, int length)
public DatagramPacket(byte[] buf, int length, InetAddress address, int port)

public InetAddress getAddress()
public int getPort()
public byte[] getData()
public int getLength()

public void setAddress(InetAddress iaddr)
public void setPort(int iport)
public void setData(byte[] buf)
public void setLength(int length)
```

Divers

Diverses méthodes renvoient le numéro de port local et l'adresse de la machine locale (`getLocalPort()` et `getLocalAddress()`), et dans le cas d'une socket connectée, le numéro de port distant et l'adresse distante (`getPort()` et `getInetAddress()`). Comme précédemment, on peut spécifier un délai de garde pour l'opération receive avec `setSoTimeout()`. On peut aussi obtenir ou réduire la taille maximale d'un paquet avec `getSendBufferSize()`, `getReceiveBufferSize()`, `setSendBufferSize()` etsetReceiveBufferSize()‘.

Enfin, la méthode `close()` libère les ressources du système associées à la socket.

Attention, la commincation réseau par socket sous Android nécessitera d'utiliser des threads.

Multitâche

Toute application Android est composée d'une multitude de threads.

L'**UI thread** est le fil d'exécution d'une activité. Les méthodes `onCreate()`, `onStart()`, `OnPause()`, `onResume()`, `onStop()`, `onDestroy()` de l'activité sont toutes exécutées dans ce thread.

L'UI thread est responsable de l'affichage et des interactions avec l'utilisateur et surtout c'est **le seul thread qui doit modifier l'affichage**.

Remarque : on ne peut pas effectuer des traitements consommateurs en temps dans l'UI thread car celui-ci se “figerait” et ne répondrait plus aux actions de l'utilisateur. D'autre part si une activité réagit en plus de cinq secondes, elle sera tuée par l'`ActivityManager` qui la considérera comme morte.

En Java, il y a plusieurs façons de créer et exécuter un thread. Le principe de base revient à dériver (`extends`) une classe `Thread` ou à implémenter (`implements`) l'interface `Runnable` et à écrire le code du thread dans la méthode `run()`. Ensuite, on appelera la méthode `start()` pour démarrer le thread et la méthode `stop()` pour l'arrêter.

Example :


```
// Une classe "Thread"
public class TClientTCP implements Runnable
{
    // Le code du thread
    public void run()
    {
    }
}

// Instanciation
TClientTCP threadClientTCP = new TClientTCP();
Thread threadClient = new Thread(threadClientTCP);

// Démarrage
threadClient.start(); // -> appel run()
```

Attention : par contre les threads de l'applications ne pourront pas accéder directement à l'IHM car seul l'UI thread peut modifier l'affichage. Pour cela Android porpose plusieurs solutions et nous utiliserons la classe `Handler`.

Handler et les messages

L'`Handler` est associé à l'activité qui le déclare et travaille au sein du *thread* d'IHM (l'UI thread). Ce qui signifie que tout traitement effectué par le *handler* "gèle" l'IHM le temps qu'il soit effectué. Il faut donc considérer le *handler* comme celui qui met à jour l'IHM, le *thread* qui appelle le *handler* a en charge le traitement. Le *handler* ne doit se charger que de mettre à jour l'IHM, tout autre comportement est une erreur de conception.

```
public class MyActivity extends AppCompatActivity implements View.OnClickListener
{
    // Gère les communications avec le thread réseau
    final private Handler handler;

    ...
}
```

Un *thread* communique avec cet *handler* au moyen de **messages** (ou des objets `Runnable`). Pour cela :

- le *thread* récupère un objet « Message » du *pool* du *handler* par la méthode `Message.obtain()`. Il peut ensuite ajouter un code (*what*) et d'autres paramètres (arg1, arg2 ou obj) ;
- le *thread* envoie le message au *handler* en utilisant la méthode `sendMessage()` qui envoie le message et le place à la fin de la queue,
- le *handler* doit surcharger sa méthode `handleMessage()` pour répondre aux messages qui lui sont envoyés.

Côté *thread* :

```
public class ClientTCP
{
    private Socket socket = null;
    private String adresseIPDuServeur;
    private int numeroDePortDuServeur;
    public final static int CODE_CONNEXION = 0;
    public final static int CODE_RECEPTION = 1;
    public final static int CODE_DECONNEXION = 2;
    private TClientTCP threadClientTCP;
    private Thread threadClient;

    ...

    public ClientTCP(String adresseServeur, int portServeur, Handler handlerUI)
    {
        adresseIPDuServeur = adresseServeur;
        numeroDePortDuServeur = portServeur;
        // Récupère l'handler de l'IHM
        handler = handlerUI;
        ...
    }
```



```
    ...

    // Une méthode pour diffuser un message vers l'IHM
    public void diffuser(int code, String message)
    {
        Message msg = Message.obtain();
        msg.what = code;
        if(message != null)
            msg.obj = message;
        handler.sendMessage(msg);
    }
}
```

Exemple d'utilisation :

```
diffuser(CODE_CONNEXION, null);
// ou :
diffuser(CODE_RECEPTION, "Hello world!");
```

Côté activité (IHM) :

```
public class MyActivity extends AppCompatActivity implements View.OnClickListener
{
    ...

    // Gère les communications avec le thread réseau
    final private Handler handler = new Handler()
    {
        public void handleMessage(Message msg)
        {
            super.handleMessage(msg);
            if(msg.what == ClientTCP.CODE_CONNEXION)
            {
                // Modifie l'IHM
                edtAdresseIP.setEnabled(false);
                edtPort.setEnabled(false);
                btnConnecter.setEnabled(false);
                btnDeconnecter.setEnabled(true);
                swLed1.setEnabled(true);
                TextView textViewReception = (TextView) findViewById(R.id.textViewReception);
                textViewReception.setText("");
            }

            ...

        }
    };

    ...
}
```

Activité n°2

AndroidManifest.xml

L'application a besoin des droits d'accès (`uses-permission`) au réseau pour fonctionner.

Pour cela, il faut modifier le fichier `AndroidManifest.xml`de l'application et reconstruire le projet :

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
          package="com.example.tv.myapplication2">

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



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

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity
            android:name=".MainActivity"
            android:label="@string/app_name"
            android:theme="@style/AppTheme.NoActionBar">
            <intent-filter>
                <action android:name="android.intent.action.MAIN"/>
                <category android:name="android.intent.category.LAUNCHER"/>
            </intent-filter>
        </activity>
        <activity android:name=".MyActivity" android:theme="@style/AppTheme.NoTitleBar">
        </activity>
    </application>
</manifest>
```

1. Éditer le fichier `AndroidManifest.xml` de l'application et reconstruire le projet.

Service réseau

L'application nécessite la présence d'un réseau pour fonctionner. On va tout d'abord vérifier son fonctionnement avant de valider les boutons qui en dépendent.

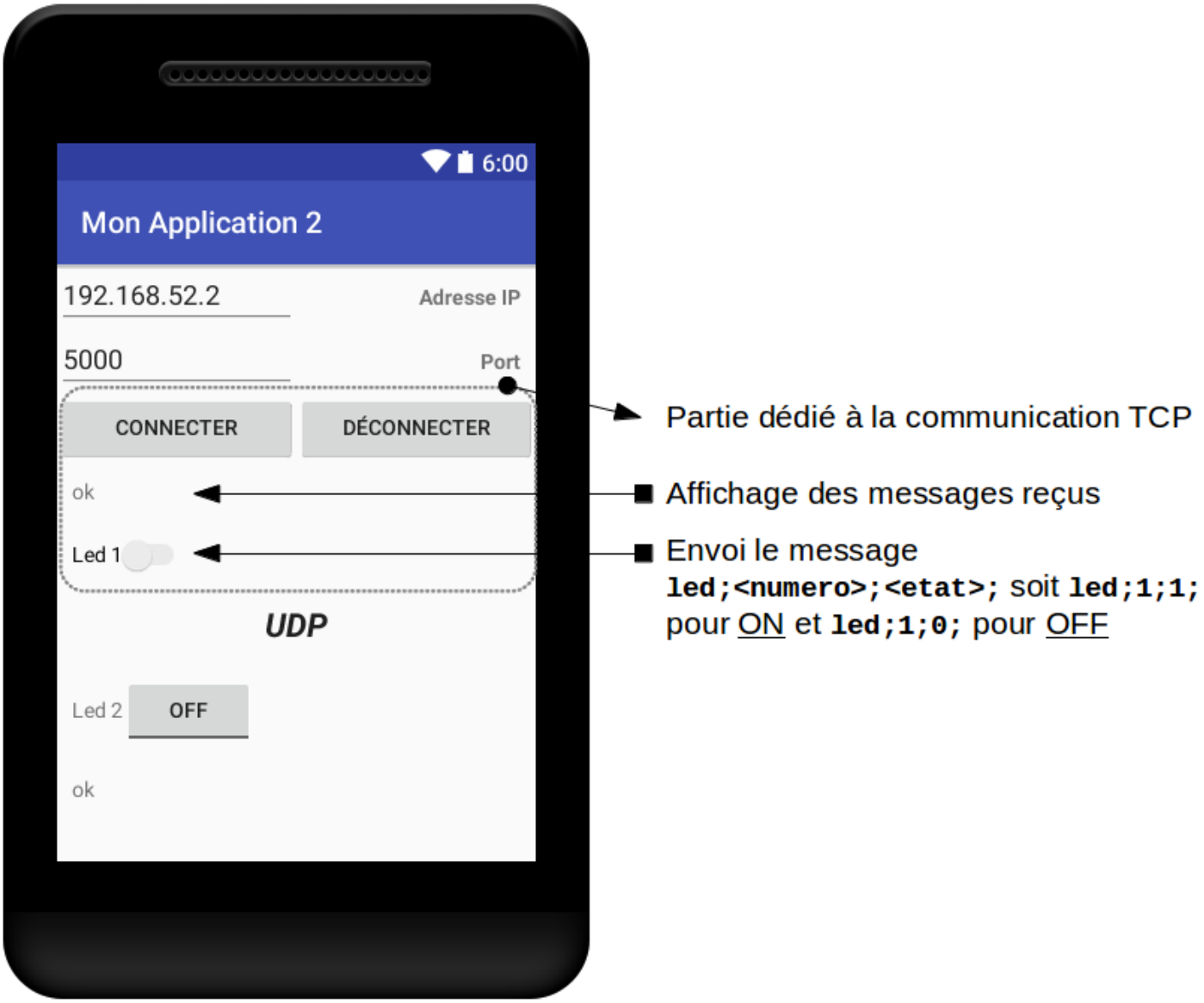
Pour cela, on ajoute le code suivant dans la méthode `onCreate()` de l'activité :

```
ConnectivityManager cm = (ConnectivityManager)getApplicationContext().getSystemService(Context.
    CONNECTIVITY_SERVICE);
if(cm != null && (cm.getActiveNetworkInfo() == null || !cm.getActiveNetworkInfo().isConnected()))
{
    btnConnecter.setEnabled(false);
    btnDeconnecter.setEnabled(false);
    swLed1.setEnabled(false);
    tbLed2.setEnabled(false);
    Toast toast = Toast.makeText(getApplicationContext(), "Aucun réseau disponible !", Toast.LENGTH_SHORT);
    toast.show();
}
else
{
    btnConnecter.setEnabled(true);
    btnDeconnecter.setEnabled(false);
    swLed1.setEnabled(false);
    tbLed2.setEnabled(true);
}
```

2. Tester en activant puis en désactivant le wifi du *smartphone*.

Communication TCP

Le principe de la communication TCP à mettre en oeuvre est le suivant :



L'application joue le rôle de **client TCP**.

Pour les besoins des tests, on pourra utiliser `netcat` comme serveur TCP sur un PC du réseau :

```
$ nc.traditional -l -p 5000
```

On va créer deux classes : `ClientTCP` qui contiendra les méthodes de communication et `TClientTCP` qui aura a la charge d'exécuter le code du *thread*.

3. Compléter (cf. **TODO**) le code de la classe `ClientTCP` :

```
package com.example.tv.myapplication2;

import android.os.Handler;
import android.os.Message;
import java.io.*;
import java.net.*;
import java.util.concurrent.LinkedBlockingQueue;

public class ClientTCP
{
    private Socket socket = null;
    private String adresseIPDuServeur;
```



```
    private int numeroDePortDuServeur;
    private Handler handler;
    public final static int CODE_CONNEXION = 0;
    public final static int CODE_RECEPTION = 1;
    public final static int CODE_DECONNEXION = 2;
    private LinkedBlockingQueue<String> qReception;
    private LinkedBlockingQueue<String> qEmission;
    private TClientTCP threadClientTCP;
    private Thread threadClient;

    public ClientTCP(String adresseServeur, int portServeur, Handler handlerUI)
    {
        adresseIPDuServeur = adresseServeur;
        numeroDePortDuServeur = portServeur;
        handler = handlerUI;
        qReception = new LinkedBlockingQueue<String>();
        qEmission = new LinkedBlockingQueue<String>();
    }

    public void demarrer()
    {
        if(threadClient == null)
        {
            threadClientTCP = new TClientTCP(this);
            threadClient = new Thread(threadClientTCP);
            threadClient.start();
        }
    }

    public void arreter() throws IOException
    {
        if(threadClient != null)
        {
            threadClientTCP.stop();
            Thread t = threadClient;
            threadClient = null;
            t.interrupt();
        }
    }

    public void connecter()
    {
        // TODO

        // crée une adresse InetAddress à partir de l'adresse IP du serveur spécifié

        // crée et connecte la socket au serveur

        // diffuse un message de connexion à l'IHM

    }

    public void deconnecter()
    {
        // TODO

        // socket connecté ?

        // alors fermer la socket

        // diffuser un message de déconnexion à l'IHM

    }

    public void envoyer(String message)
    {
```

```
        qEmission.add(message);
    }

    public void setMessageLu(String message)
    {
        byte[] src = message.getBytes();
        byte[] dst = new byte[message.length()];
        System.arraycopy(src, 0, dst, 0, dst.length);

        String messageLu = new String(dst);
        diffuser(CODE_RECEPTION, messageLu);

        qReception.add(messageLu);
    }

    public String getMessageLu()
    {
        return qReception.poll();
    }

    public String getMessage()
    {
        return qEmission.poll();
    }

    public Socket getSocket()
    {
        return socket;
    }

    public void diffuser(int code, String message)
    {
            Message msg = Message.obtain();
            msg.what = code;
            if(message != null)
                msg.obj = message;
            handler.sendMessage(msg);
    }

    public void fermer() throws IOException
    {
        arreter();
        deconnecter();
    }
}
```

Remarque : les messages à envoyer au serveur sont stockés dans une queue. Le *thread* les récupérera et les enverra sur la *socket*.

4. Compléter le code (cf. **TODO**) de la classe `TClientTCP` :

```
package com.example.tv.myapplication2;

import java.io.*;
import java.net.*;

public class TClientTCP implements Runnable
{
    private ClientTCP clientTCP = null;
    private Socket socket = null;
    private PrintWriter out = null;
    private BufferedReader in = null;
    private volatile boolean fini = false;
    private InputStream stream = null;

    public TClientTCP(ClientTCP clientTCP)
```

```
{
    this.clientTCP = clientTCP;
    fini = false;
}

public void run()
{
    // TODO : connecter le client


    socket = clientTCP.getSocket();
    if(socket == null)
    {
        return;
    }

    initialiser();

    // boucle d'émission et de réception de messages réseaux
    String message = null;
    String messageLu = null;

    while(!fini)
    {
        // un message à envoyer ?
        message = clientTCP.getMessage();
        if(message != null)
        {
            out.println(message);
            out.flush();
            System.out.println("Envoi : " + message);
        }

        // un message à receptionner ?
        try
        {
            if(stream.available() > 1)
            {
                messageLu = in.readLine();
            }
        }
        catch (IOException e)
        {
            e.printStackTrace();
        }
        if(messageLu != null)
        {
            clientTCP.setMessageLu(messageLu);
            System.out.println("Lu : " + messageLu);
            messageLu = null;
        }
    }
    // fin de la boucle

    // TODO : déconnecter le client

}

private void initialiser()
{
    // crée et associe le flux de sortie à la socket
    try
    {
        if(socket != null)
        {
            out = new PrintWriter(new BufferedWriter(new OutputStreamWriter(socket.getOutputStream())), true
```

```
    );
            }
        }
        catch (UnknownHostException e)
        {
            e.printStackTrace();
        }
        catch (IOException e)
        {
            e.printStackTrace();
        }

        // crée et associe le flux d'entrée à la socket
        try
        {
            if(socket != null)
            {
                stream = socket.getInputStream();
                in = new BufferedReader(new InputStreamReader(stream));
            }
        }
        catch (UnknownHostException e)
        {
            e.printStackTrace();
        }
        catch (IOException e)
        {
            e.printStackTrace();
        }
    }

    public void stop() throws IOException
    {
        if(fini == false)
        {
            fini = true;
            if(in != null && out != null)
            {
                in.close();
                out.close();
            }
        }
    }
}
```

Maintenant, il faut compléter le code dans l'activité :

– pour démarrer un client TCP :

```
if(clientTCP == null)
{
    clientTCP = new ClientTCP(edtAdresseIP.getText().toString(), Integer.parseInt(edtPort.getText().toString()),
     handler);
    clientTCP.demarrer();
}
```

– pour arrêter le client TCP :

```
if(clientTCP != null)
{
    try
    {
        clientTCP.arreter();
    }
    catch (IOException e)
    {
        e.printStackTrace();
```

```
    }
    clientTCP = null;
}
```

– pour envoyer un message au serveur :

```
if(clientTCP != null)
{
    clientTCP.envoyer("led;1;1;");
}
```

5. Modifier la classe `MyActivity` afin de répondre aux besoins.
6. Tester la communication réseau.

Pour les besoins des tests, on pourra utiliser `netcat` comme serveur TCP sur un PC du réseau :

```
$ nc.traditional -l -p 5000
```

Communication UDP

Le principe de la communication UDP à mettre en oeuvre est le suivant :

L'application joue le rôle de **client UDP**.

Pour les besoins des tests, on pourra utiliser `netcat` comme serveur UDP sur un PC du réseau :

```
$ nc.traditional -u -l -p 5000
```

On va créer deux classes : `ClientUDP` qui contiendra les méthodes de communication et `TClientUDP` qui aura a la charge d'exécuter le code du *thread*.

7. Éditer le code de la classe `ClientUDP` :

```
package com.example.tv.myapplication2;

import android.os.Handler;
import android.os.Message;

import java.io.*;
import java.net.*;
import java.util.concurrent.LinkedBlockingQueue;

public class ClientUDP
{
    private InetAddress serverAddr = null;
    private DatagramSocket socket = null;
    private TClientUDP tClientUDP;
    private Thread threadClientUDP;
    private LinkedBlockingQueue<DatagramPacket> qEmission;

    public ClientUDP()
    {
        try
        {
            socket = new DatagramSocket();
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }

        qEmission = new LinkedBlockingQueue<DatagramPacket>();
    }

    private void demarrer()
    {
        if(threadClientUDP == null)
        {
            tClientUDP = new TClientUDP(this, socket);
            threadClientUDP = new Thread(tClientUDP);
            threadClientUDP.start();
        }
    }

    private void arreter()
    {
        if(threadClientUDP != null)
        {
            tClientUDP.stop();
            Thread t = threadClientUDP;
            threadClientUDP = null;
            t.interrupt();
        }
    }

    public void envoyer(String leMessage, String adresseServeur, int portServeur)
    {
        byte[] message= new byte[leMessage.length()];
        message = leMessage.getBytes();
```



```
        if(threadClientUDP == null)
        {
            demarrer();
        }

        try
        {
            serverAddr = InetAddress.getByName(adresseServeur);
        }
        catch (UnknownHostException e)
        {
            e.printStackTrace();
        }

        try
        {
            DatagramPacket packet = new DatagramPacket(message, leMessage.length(), serverAddr, portServeur);
            setPaquet(packet);
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
    }

    public DatagramPacket getPaquet()
    {
        return qEmission.poll();
    }

    public void setPaquet(DatagramPacket packet)
    {
        qEmission.add(packet);
    }

    public void fermer()
    {
        arreter();
        socket.close();
    }
}
```

Remarque : les messages à envoyer au serveur sont stockés dans une queue. Le *thread* les récupérera et les enverra sur la *socket*.

8. Compléter le code (cf. **TODO**) de la classe `TClientUDP` :

```
package com.example.tv.myapplication2;

import java.io.IOException;
import java.net.*;

public class TClientUDP implements Runnable
{
    private ClientUDP clientUDP = null;
    private DatagramSocket socket = null;
    private volatile boolean fini = false;

    public TClientUDP(ClientUDP clientUDP, DatagramSocket socket)
    {
        this.clientUDP = clientUDP;
        this.socket = socket;
        fini = false;
    }
```



```
    public void run()
    {
        while(!fini)
        {
            DatagramPacket packet = clientUDP.getPaquet();
            while(packet != null)
            {
                try
                {
                    // TODO : envoyer le paquet en utilisant la méthode send() de l'objet socket

                }
                catch (IOException e)
                {
                    e.printStackTrace();
                }
                packet = clientUDP.getPaquet();
            }
        }
    }

    public void stop()
    {
        if(fini == false)
        {
            fini = true;
        }
    }
}
```

Remarque : vous pouvez ajouter une réception de paquets UDP.

Maintenant, il faut compléter le code dans l'activité :

– pour créer un client UDP :

```
ClientUDP clientUDP = new ClientUDP();
```

– pour envoyer un message au serveur :

```
if(clientUDP != null)
{
    clientUDP.envoyer("led;2;1;", edtAdresseIP.getText().toString(), Integer.parseInt(edtPort.getText().toString
    ()));
}
```

9. Modifier la classe `MyActivity` afin de répondre aux besoins.
10. Tester la communication réseau.

Pour les besoins des tests, on pourra utiliser `netcat` comme serveur UDP sur un PC du réseau :

```
$ nc.traditional -u -l -p 5000
```

Bonus : les notifications

On va ajouter un mécanisme de notification (avec vibration) lorsque des données sont recues sur la socket TCP.

L'application a besoin des droits d'accès (`uses-permission`) pour faire vibre le *smartphone*. Pour cela, on modifie le fichier `AndroidManifest.xml`de l'application :

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
          package="com.example.tv.myapplication2">

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



```
    ...

</manifest>
```

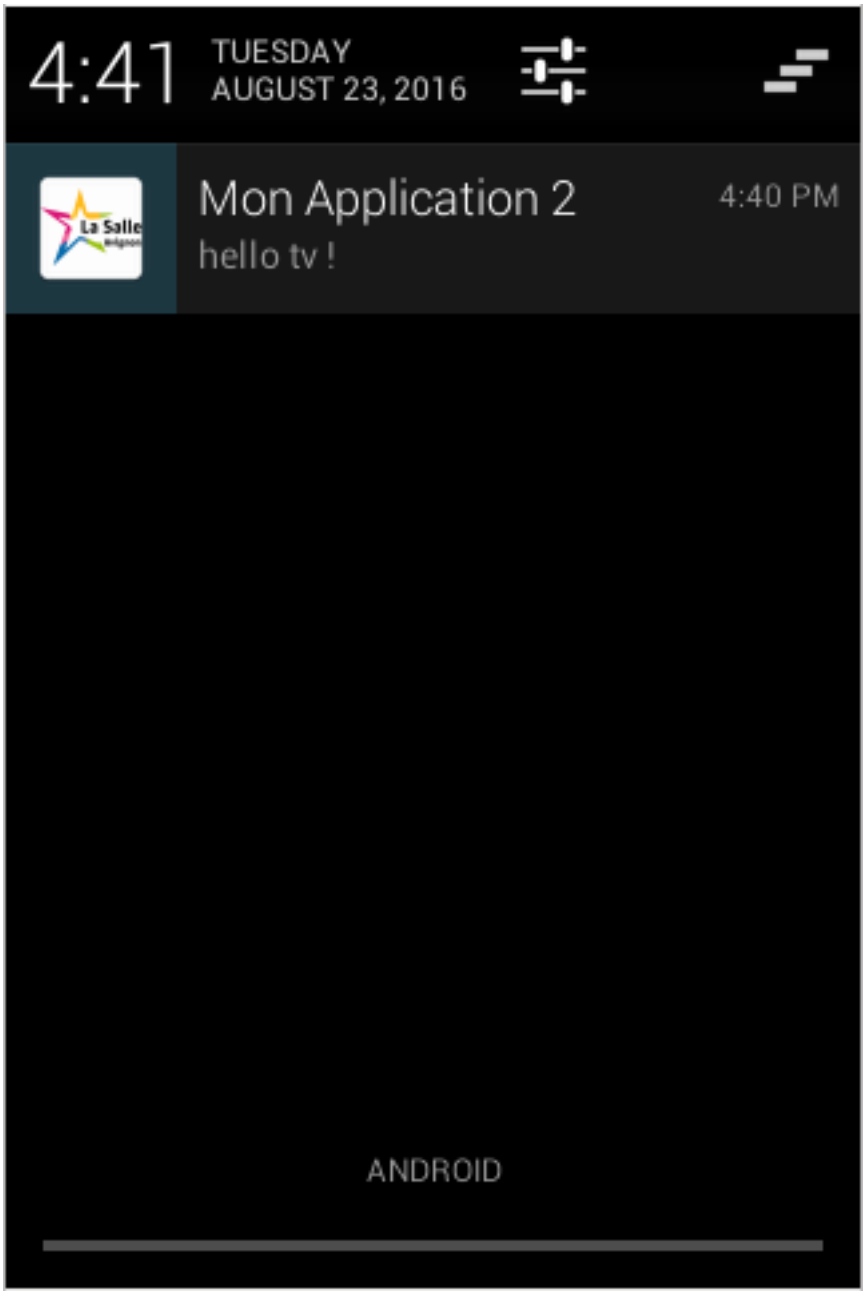
On crée ensuite une méthode `creerNotification()` que l'on ajoute à l'activité :

```
private void creerNotification(String message)
{
    // On crée un "gestionnaire de notification"
    NotificationManager notificationManager = (NotificationManager)getSystemService(Context.NOTIFICATION_SERVICE
    );

    // On définit le titre de la notification
    String titreNotification = getApplicationName(getApplicationContext());
    // On définit le texte qui caractérise la notification
    String texteNotification = message;

    // On crée la notification
    NotificationCompat.Builder notification = new NotificationCompat.Builder(this)
                    .setSmallIcon(R.mipmap.ic_launcher)
                    .setContentTitle(titreNotification)
                    .setContentText(texteNotification);

    // On pourrait ici crée une autre activité
    PendingIntent pendingIntent = PendingIntent.getActivity(this, 0, new Intent(), PendingIntent.
    FLAG_UPDATE_CURRENT);

    // On associe notre notification à l'Intent
    notification.setContentIntent(pendingIntent);

    notification.setAutoCancel(true);

    // On ajoute un style de vibration à notre notification
    // L'utilisateur est donc également averti par les vibrations de son téléphone
    // Ici les chiffres correspondent à 0sec de pause, 0.2sec de vibration, 0.1sec de pause, 0.2sec de vibration
    , 0.1sec de pause, 0.2sec de vibration
    notification.setVibrate(new long[] {0,200,100,200,100,200});

    // Enfin on ajoute notre notification et son ID à notre gestionnaire de notification
    notificationManager.notify(ID_NOTIFICATION, notification.build());
}

public static String getApplicationName(Context context)
{
    int stringId = context.getApplicationInfo().labelRes;
    return context.getString(stringId);
}
```

On ajoute maintenant l'envoi de la notificiation à partir de la méthode `handleMessage()` :

```
...
creerNotification((String)msg.obj); // obj contient les données reçues
```

On obtient :



Documentation

- Guide de référence d'Android SDK
- Initiation au développement sur cibles Android
- Cours et tutoriels pour Android
- FAQ Android

Retour

