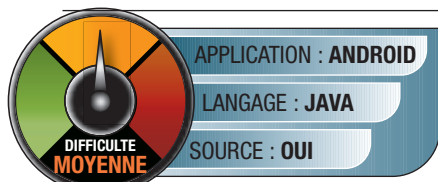


Détecteur d'événements sous Android : l'application BigBrother

Le SDK d'Android propose un modèle de composants et des API pour gérer différents dispositifs qui font la particularité des plates-formes mobiles : connectivité, capteurs, téléphonie, multimédia ... Dans cet article nous allons nous intéresser à la détection d'événements liés à la téléphonie et la géolocalisation.



L'activité est le premier composant essentiel permettant de gérer le cycle de vie d'une application et l'interactivité avec l'utilisateur ;

mais qu'en est-il lorsqu'on souhaite exécuter un traitement en tâche de fond, qui démarre automatiquement, et qui doit réagir à des événements externes comme un appel téléphonique ? Le framework propose pour cela des composants de type service et receiver qui utilisent des intentions (Intent) pour collaborer [Fig.1].

LES SERVICES : POUR DES TRAITEMENTS EN TÂCHE DE FOND

Le service peut être vu comme une activité à longue durée de vie (potentiellement infinie), en tâche de fond, et privée d'IHM ; il est implémenté par une classe qui doit étendre `android.app.Service`.

Démarrage du service : à la différence d'une activité, l'utilisateur ne dispose pas de raccourci dans son bureau, il faudra donc démarrer le service explicitement de manière programmatique (souvent depuis une activité). Les services d'une application doivent être déclarés dans son manifeste :

```
<application android:icon="@drawable/icon" android:label="@string/app_name">
  <activity .../>
  <service android:name=".Service" />
</application>
```

Un service peut, comme une activité, enregistrer des écouteurs spécialisés (listeners) pour obtenir des informations sur un capteur particulier, il est alors responsable du désenregistrement des listeners.

RECEIVER : UN DÉCLENCHEUR LÉGER

Le receiver est un composant susceptible de recevoir des intentions exprimées par le système Android ou d'autres applications. Les intentions symbolisent des requêtes (ou souhaits) et sont orchestrées par le framework ; cela offre un cadre simple et générique qui fait penser au style d'architecture REST (ou au Web), et confère un niveau d'abstraction intéressant pour faciliter l'intégration de composant. L'intérêt de ce composant réside en son exécution automatique dès qu'un événement correspondant survient. Il agit donc souvent comme un déclencheur : par exemple, pour lancer un service. Son implémentation nécessite d'étendre la classe `android.content.BroadcastReceiver` et de déclarer les filtres d'intentions dans le manifeste :

```
<receiver android:name=".EventReceiver">
  <intent-filter>
    <category android:name="android.intent.category.DEFAULT" />
    <action android:name="android.intent.action.PHONE_STATE" />
  </intent-filter>
</receiver>
```

Lors d'un broadcast d'événement du système, de nombreux récepteurs peuvent être prévenus. Pour éviter les embouteillages, le traitement de chaque récepteur doit être court, quitte à sous-traiter à un autre composant (le service par exemple).

DÉTECTION D'ÉVÉNEMENT : MISE EN ŒUVRE

Il existe deux approches pour détecter des événements sous Android : utiliser un broadcast receiver générique, ou enregistrer autant de listeners spécifiques que nécessaire.

Caractéristiques d'un broadcast receiver :

- pas besoin d'enregistrer le composant, il suffit de le déclarer
- réagit à tous les événements, pourvu que les bons filtres d'intentions soient déclarés
- simplicité de mise en oeuvre : une seule méthode à redéfinir (éviter tout de même l'inflation)
- pas besoin de démarrage : l'instanciation et l'invocation de la méthode sont automatiques
- les informations fournies sont souvent insuffisantes pour certains événements

Caractéristiques d'un listener :

- informations spécialisées et détaillées
- pas besoin de déclaration dans le manifeste
- nécessité d'enregistrer chaque listener
- peu de factorisation possible (particularités des listeners)

Pour ces raisons, il n'est pas rare d'utiliser les deux approches de manière complémentaire. Dans les deux cas, les permissions nécessaires devront bien évidemment être déclarées dans le manifeste. L'architecture générale de l'application est illustrée par les schémas : [Fig.2+3]

DÉTECTER UN APPEL TÉLÉPHONIQUE ENTRANT

Maintenant que les principes sont posés, voyons un exemple de mise en oeuvre avec une application « BigBrother » capable de détecter un appel téléphonique entrant. Nous avons besoin d'un objet métier pour encapsuler la notion d'événement :

```
package com.programmez.android.bigbrother.domain;
```

```
public class Event {
    public long eventId; // Identifiant unique de l'événement
    public String deviceId; // Identifiant unique du dispositif Android
    public String eventType; // Type de l'événement
    public long timestamp; // Timestamp de l'événement sur le dispositif
    public String params; // Eventuelles données supplémentaires
}
```

À noter : utilisation de propriétés publiques plutôt qu'un bean anémique.

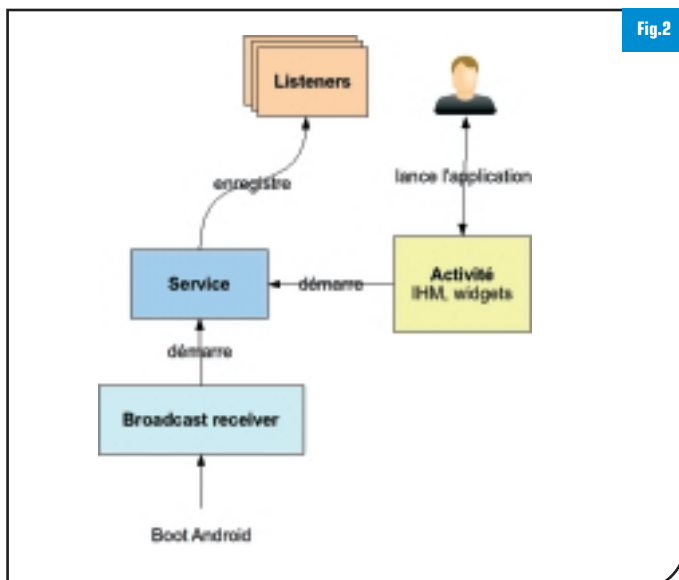
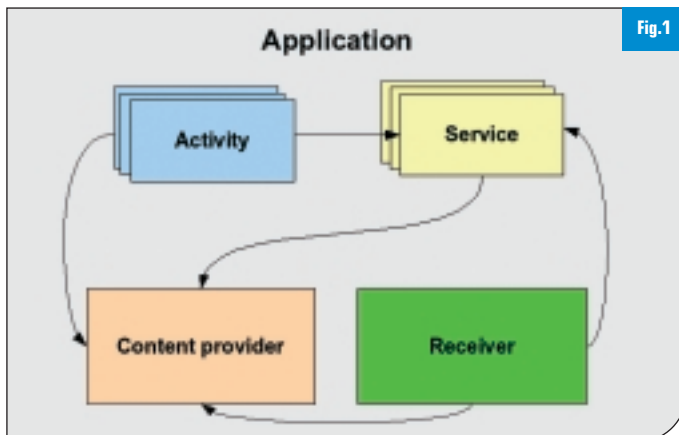
Réalisons le récepteur d'événements :

```
package com.programmez.android.bigbrother;

import static com.programmez.android.bigbrother.Constants.*;
import android.content.Context;
import android.content.Intent;
import android.telephony.TelephonyManager;
import android.util.Log;
import com.programmez.android.bigbrother.domain.Event;

public class EventReceiver extends android.content.BroadcastReceiver {
    private Event event;

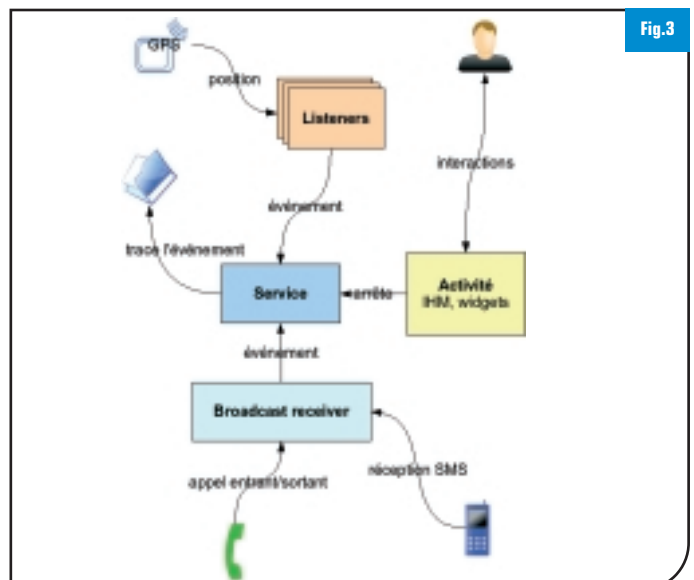
    private void onReceivePhoneStateChanged(final Context context,
```



```
final Intent intent) {
    final String phoneState = intent
        .getStringExtra(TelephonyManager.EXTRA_STATE);
    final String phoneNumber = intent
        .getStringExtra(TelephonyManager.EXTRA_INCOMING_NUMBER);
    event.eventType = EVENT_TYPE_PHONE_STATE_CHANGED;
    final StringBuilder sb = new StringBuilder();
    if (TelephonyManager.EXTRA_STATE_IDLE.equals(phoneState)) {
        sb.append("idle");
    } else if (TelephonyManager.EXTRA_STATE_OFFHOOK.equals(phoneState)) {
        sb.append("offhook");
    } else if (TelephonyManager.EXTRA_STATE_RINGING.equals(phoneState)) {
        sb.append("ringing");
    } else {
        sb.append(phoneState);
    }
    if (phoneNumber != null) {
        sb.append(',');
        sb.append("number:" + phoneNumber);
    }
    event.params = sb.toString();
    Log.v(LOG_TAG, "event : " + event);
}
```

@Override

```
public void onReceive(final Context context, final Intent intent) {
    event = null;
    final String action = intent.getAction();
    event = new Event();
    if (TelephonyManager.ACTION_PHONE_STATE_CHANGED.equals(action)) {
        onReceivePhoneStateChanged(context, intent);
    } else { // Default event code
        event.eventType = action;
        final String data = intent.getDataString();
        Log.v(LOG_TAG, "broadcast : action=" + action + ", data=" + data);
        Log.v(LOG_TAG, "event : " + event);
    }
}
```



Constantes de l'application :

```
package com.programmez.android.bigbrother;

public interface Constants {
    String LOG_TAG = BigBrother.class.getSimpleName();
    String EVENT_TYPE_PHONE_STATE_CHANGED = "phone state changed";
}
```

Notre récepteur est censé réagir à l'intention ACTION_PHONE_STATE_CHANGED ; il faut donc adapter le manifeste :

```
<receiver android:name=".EventReceiver">
    <intent-filter>
        <category android:name="android.intent.category.DEFAULT" />
        <action android:name="android.intent.action.PHONE_STATE" />
    </intent-filter>
</receiver>
```

Et ne pas oublier les permissions :

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

Fonctionnement du récepteur :

La seule méthode à redéfinir est *onReceive* ; elle fournit un contexte et l'intention qui a été exprimée. Dans notre cas, la méthode vérifie l'intention, délègue le traitement à une autre méthode s'il s'agit de l'intention qui nous intéresse, sinon applique un comportement par défaut. On se limite ici à tracer l'événement. L'intention fournit quelques informations supplémentaires (extra), indispensables pour bien qualifier l'événement : les états RINGING, OFFHOOK, IDLE correspondant respectivement à un appel qui arrive (sonnerie), la prise d'appel, et la fin d'appel. On dispose en plus du numéro appelant via la clé EXTRA_INCOMING_NUMBER ; on en profite pour l'ajouter à notre événement.

Pour cet exemple, le broadcast receiver répond bien à notre besoin en fournissant les informations pertinentes via le paramètre Intent. Ce n'est malheureusement pas toujours le cas, et pour certains événements les informations à disposition sont pauvres ; dans ce cas les listeners viennent en renfort pour fournir des informations spécialisées. En fait, le broadcast receiver correspond à un besoin de détecter un changement d'état d'un périphérique, pas beaucoup plus...

GÉOLOCALISATION : DÉTECTER UNE NOUVELLE POSITION

La géolocalisation est un bon exemple d'utilisation d'un listener. On souhaite obtenir la position géographique à chaque changement suivant des critères de distance et de période. L'acquisition des informations qui nous intéressent (latitude et longitude) nécessite la mise en œuvre d'un LocationListener.

Pour rendre le système industrialisable, une classe abstraite pourra figer le contrat des implémentations de listeners :

```
package com.programmez.android.bigbrother.listener;

import android.content.Context;

public abstract class Listener {
    private boolean registered;
```

```
public boolean isRegistered() {
    return registered;
}

public boolean register(final Context context) {
    registered = false;
    if (registerImpl(context)) {
        registered = true;
        return true;
    }
    return false;
}

protected abstract boolean registerImpl(final Context context);

public boolean unregister(final Context context) {
    if (unregisterImpl(context)) {
        registered = false;
        return true;
    }
    return false;
}

protected abstract boolean unregisterImpl(final Context context);
}
```

Implémentation du LocationListener :

```
package com.programmez.android.bigbrother.listener;

import static com.programmez.android.bigbrother.Constants.*;
import java.util.List;
import android.content.Context;
import android.location.Location;
import android.location.LocationManager;
import android.os.Bundle;
import android.util.Log;
import com.programmez.android.bigbrother.Service;
import com.programmez.android.bigbrother.domain.Event;

public class LocationListener extends Listener implements
    android.location.LocationListener {
    private static LocationManager manager;
    private static List<String> locationProviders;

    public void onLocationChanged(final Location location) {
        final Service service = Service.getInstance();
        final Event event = new Event();
        event.eventType = EVENT_TYPE_LOCATION;
        final StringBuilder sb = new StringBuilder();
        sb.append("latitude:");
        sb.append(location.getLatitude());
        sb.append(",longitude:");
        sb.append(location.getLongitude());
        event.params = sb.toString();
        service.acceptIncomingEvent(event);
    }
}
```

```

public void onProviderDisabled(final String provider) {
    // Do nothing
}

public void onProviderEnabled(final String provider) {
    // Do nothing
}

public void onStatusChanged(final String provider, final int status,
    final Bundle extras) {
    // Do nothing
}

@Override
protected boolean registerImpl(final Context context) {
    if (!Service.checkPermissions(context, new String[] {
        android.Manifest.permission.ACCESS_COARSE_LOCATION,
        android.Manifest.permission.ACCESS_FINE_LOCATION }))
        return false;
    manager = (LocationManager) context
        .getSystemService(Context.LOCATION_SERVICE);
    locationProviders = manager.getAllProviders();
    for (final String provider : locationProviders) {
        Log.v(LOG_TAG, "register location provider : " + provider);
        manager.requestLocationUpdates(provider, LOCATION_MIN_
UPDATE_TIME, LOCATION_MIN_UPDATE_DISTANCE, this);
    }
    return true;
}

@Override
protected boolean unregisterImpl(final Context context) {
    if (manager != null) {
        manager.removeUpdates(this);
        return true;
    }
    return false;
}
}

```

L'enregistrement du listener est déclenché par un composant de type service dont vous trouverez le code complet sur notre site www.programmez.com.

Lorsque le service démarre, il enregistre les différents listeners (ici seulement un LocationListener) ; et les désactive en s'arrêtant.

Le service fournit aussi la méthode *acceptIncomingEvent* pour traiter tous les événements arrivant de manière centralisée, raffiner l'événement avec l'identifiant du terminal (subscriberId) et le timestamp puis notifier l'événement (méthode notifyEvent qui se limite à tracer). Le service respectant le pattern singleton, un autre composant pourra facilement interagir avec lui grâce à la méthode *getInstance* et vérifier qu'il est bien en exécution (isRunning).

À noter l'utilisation des *Shared-Preferences* pour mémoriser l'état du service (démarré ou

pas). Le démarrage du service doit être déclenché par un autre composant, classiquement une activité.

Ajoutons donc un bouton au layout qui permettra à l'utilisateur de démarrer ou arrêter le service : [Fig.4]

```

<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/mainLayout" android:layout_height="fill_parent"
    android:layout_width="fill_parent" android:background="#FFFFFF"
    android:orientation="vertical">
    <TextView android:id="@+id/TextView01" android:layout_height="wrap_content"
        android:layout_width="fill_parent" android:gravity="center_
horizontal"
        android:paddingTop="20sp" android:paddingBottom="20sp"
        android:textStyle="bold" android:textColor="#000066" android:
text="@string/app_name"
        android:textSize="30sp"></TextView>
    <ToggleButton android:textOff="@string/start_service"
        android:textOn="@string/stop_service" android:layout_width="fill_parent"
        android:layout_height="wrap_content" android:id="@+id/
ToggleServiceButton" />
</LinearLayout>

```

Pour faire en sorte que le service démarre automatiquement, utilisons le broadcast receiver qui est capable de détecter un événement de fin de boot. Modifications de la classe EventReceiver :

```

public class EventReceiver extends android.content.Broadcast
Receiver {
    private Service service;
    private Event event;

    @Override
    public void onReceive(final Context context, final Intent intent) {
        service = null;
        event = null;
        final String action = intent.getAction();
        if (Intent.ACTION_BOOT_COMPLETED.equals(action)) {
            onReceiveBoot(context, intent);
            return;
        }
        service = Service.getInstance();
        if (service == null || !Service.isEnabled(context) || !Service.isRunning())
            return;
        event = new Event();
        if (TelephonyManager.ACTION_PHONE_STATE_CHANGED.equals(action))
            onReceivePhoneStateChanged(context, intent);
        else { // Default event code
            event.eventType = action;
            final String data = intent.getDataString();
            Log.v(LOG_TAG, "broadcast : action=" + action + ", data=" + data);
            service.acceptIncomingEvent(event);
        }
    }
}

```

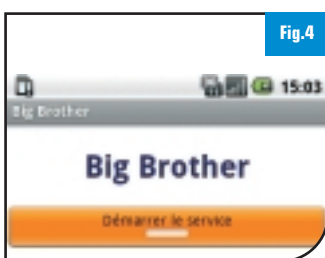


Fig.4

```
private void onReceiveBoot(final Context context, final Intent intent) {
    Log.v(LOG_TAG, "boot completed");
    if (Service.isEnabled(context) && !Service.isRunning())
        Service.start(context, false);
}
```

Remplacer la dernière ligne de la méthode `onReceivePhoneStateChanged` pour confier le traitement de l'événement au service :

```
private void onReceivePhoneStateChanged(,,, ) {
    ...
    service.acceptIncomingEvent(event);
}
```

Il reste à modifier le manifeste pour ajouter le nouveau filtre d'intention et les permissions :

```
<receiver android:name=".EventReceiver">
    <intent-filter>
        <category android:name="android.intent.category.DEFAULT" />
        <action android:name="android.intent.action.BOOT_COMPLETED" />
        <action android:name="android.intent.action.PHONE_STATE" />
    </intent-filter>
</receiver>
</application>
<uses-sdk android:minSdkVersion="3" />
<uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED" />
<uses-permission android:name="android.permission.READ_PHONE_STATE" />
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
```

Lorsque l'utilisateur clique sur le widget `ToggleButton`, le service démarre. L'utilisateur peut ensuite sortir de l'activité (touche back), cela n'aura pas d'effet sur le service qui continuera de tourner.

Si l'utilisateur éteint et rallume son téléphone Android, le service

sera lancé automatiquement juste après le boot par le broadcast receiver. L'état du `ToggleButton` est sauvegardé dans les préférences partagées et utilisé partout pour savoir si le service doit démarrer ou pas, et si les événements doivent être traités.

Ainsi, l'activité se résume à donner à l'utilisateur un moyen de contrôle sur le lancement du service.

POUR ALLER PLUS LOIN...

Cet embryon d'application peut facilement évoluer et gérer de nombreux événements en complétant le receiver, ou en implémentant d'autres listeners ; pour y intégrer par exemple la réception de SMS. Par ailleurs, le traitement de l'événement pourrait aussi évoluer pour publier les informations sur un serveur. Une application RESTful serait alors une solution de premier choix, d'autant que l'implémentation du client sous Android est très simple.

Cela implique, qu'au moment du traitement, le téléphone ait accès à Internet, sous peine de générer une erreur et potentiellement perdre l'événement. Là encore, une solution simple consiste à, tout d'abord, persister les informations localement dans une base SQLite ; le service se contentant alors d'alimenter la base, tandis qu'un nouveau composant (un thread poster) aurait en charge de consulter la base, publier les événements si une connexion 3G ou Wifi est disponible (pour ça aussi il y a un événement !), et purger la base.

Ce mode de fonctionnement est illustré dans les schémas : [Fig.5 et 6]

BigBrother is watching you !

Sur cette base, il est assez aisé d'imaginer une application «BigBrother» déployable, avec sur le serveur une interface Web qui permet de consulter les informations, faire des statistiques, et plein d'autres choses... Bien évidemment, outre les aspects techniques traités ici et la créativité que peut susciter ce type de développement, une telle application n'est pas sans poser un certain nombre

de questions d'ordre légal, moral et social : respect de la vie privée, conformité CNIL ...



■ Olivier Penhoat
Consultant & Formateur
Valtech Training

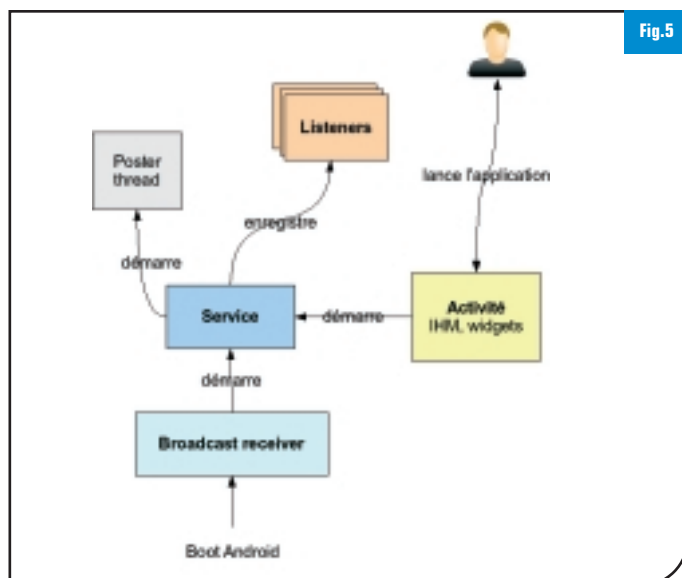


Fig.5

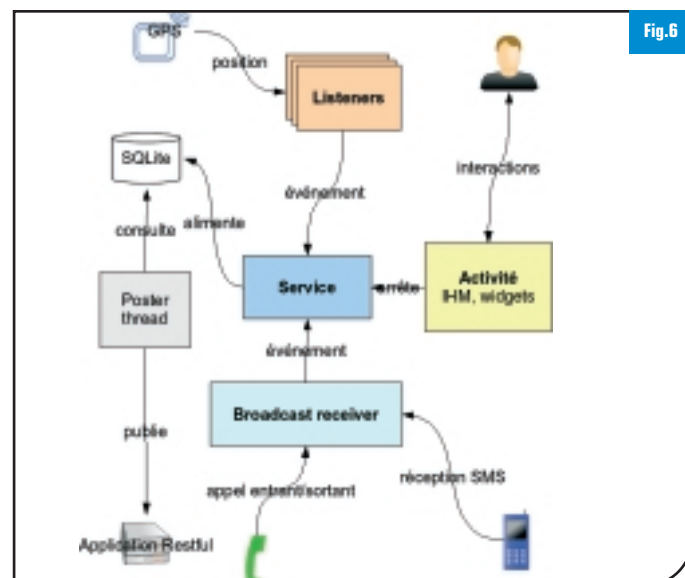


Fig.6