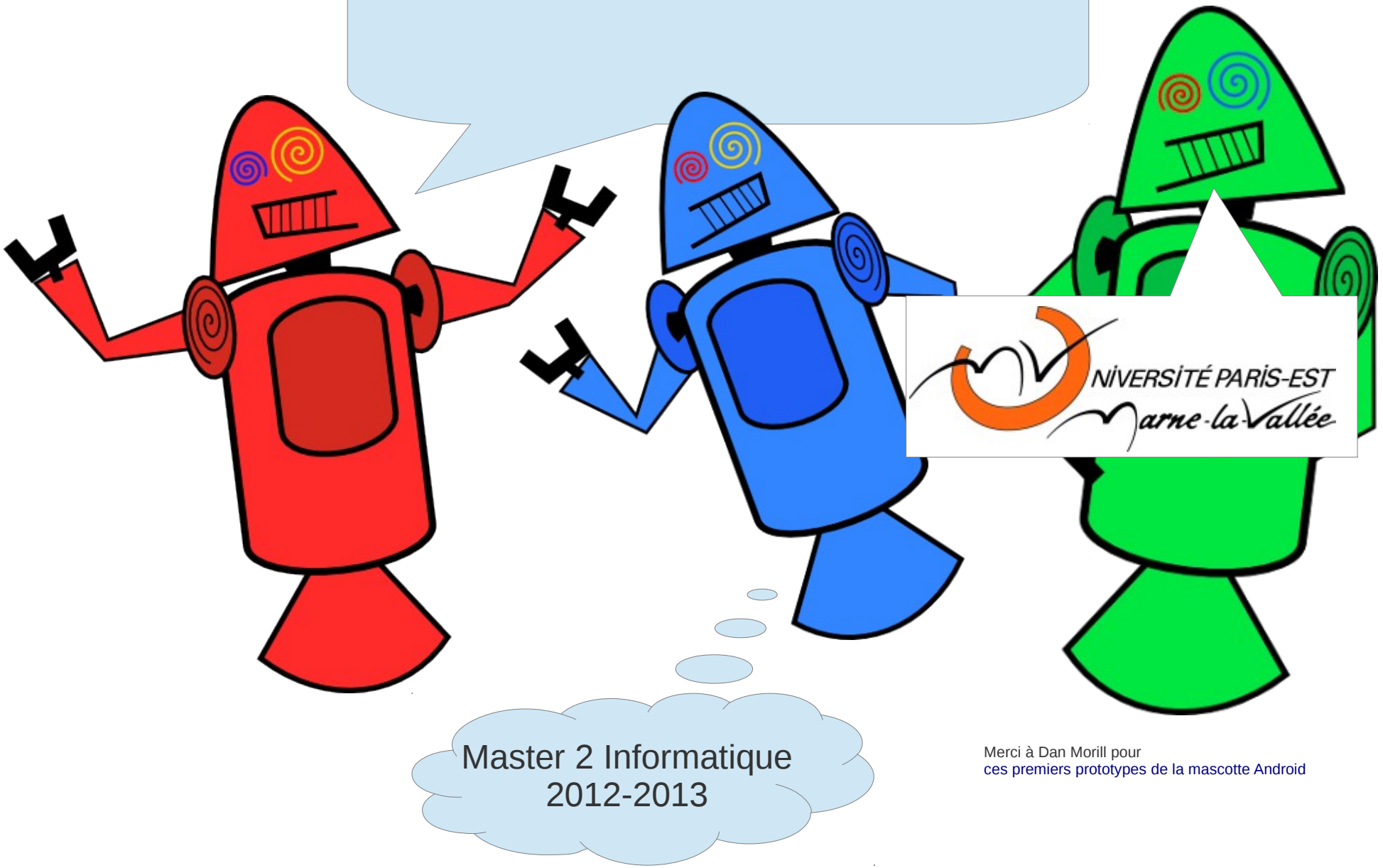
IPC sous Android



Master 2 Informatique
2012-2013

Merci à Dan Morill pour
ces premiers prototypes de la mascotte Android

Problématique de la communication inter-processus

- Antagonisme à résoudre
  - Nécessité d'assurer une sécurité en cloisonnant les processus (mémoire centrale et mémoire de stockage)
  - Nécessité de communiquer entre les différents processus
- Approche proposée
  - Communication par messages synchrones d'un composant (implantant un client) vers un autre (implantant un serveur)
  - Implantation : Binder (dérivé du projet OpenBinder pour BeOS)

Binder

- Implantations à 3 niveaux
  - Driver kernel pour partage de mémoire et gestion de la sécurité
  - Bibliothèque C++ en espace utilisateur avec support de sérialisation, paradigme client/serveur (threads de traitement pour serveur)
  - API Java de communication avec appels JNI à la bibliothèque C++
- Registre des binders assuré par un binder spécial (context manager)

Service

- Service = composante d'application exécutant du code sans exposition graphique
  - Opérations en arrière plan au long cours
  - Exposition d'une API d'une application à d'autres applications
- Implantation par dérivation de classe abtraite Service avec redéfinition possible de :
  - IBinder onBind() ; peut être redéfini en retournant null si l'on ne souhaite pas offrir d'interface
  - void onCreate() : événement de création du service
  - int onStartCommand(Intent intent, int flags, int startID)
  - void onDestroy()
- Par défaut, un service partage le processus et la thread principale de son application hôte
- L'exécution des méthodes onX() doit être brève, les travaux de calcul doivent être réalisés par de nouvelles threads
- Un processus contenant un service et aucune activité en avant-plan peut être tué
- Deux modes d'utilisation de service :
  - Soumission de travaux en asynchrone par Intent
  - Évocation de méthodes distantes après connexion au service

Soumission de travaux à un service

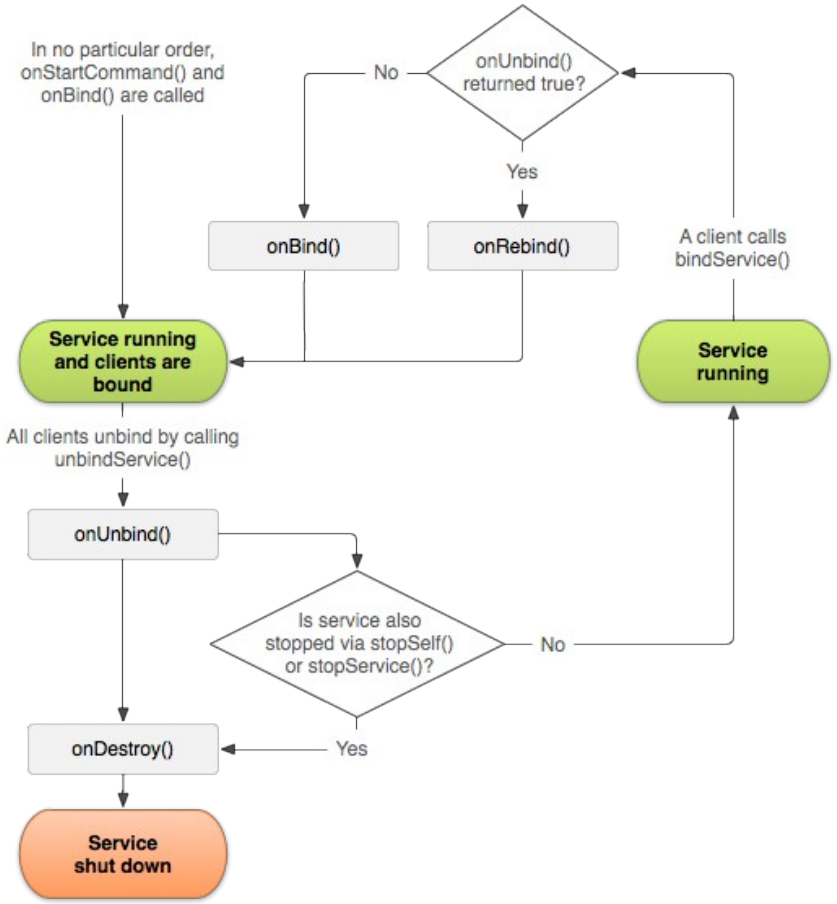
- On envoie un *Intent* avec *Context.startService(Intent i)* ; le service est créé si inexistant
- L'Intent est récupérable par le service avec *onStartCommand(Intent intent, int flags, int startID)* :
  - *flags* indique 0, *START_FLAG_REDELIVERY* ou *START_FLAG_RETRY*
  - onStartCommand peut retourner 0, *START_STICKY*, *START_NOT_STICKY* ou *START_REDELIVER_INTENT* (comportement pour le redémarrage si le service est stoppé par le système)
- Pas de retour d'*Intent* (communication unidirectionnelle)
- Le service peut être stoppé :
  - par lui-même avec *stopSelf()*
  - par le composant l'ayant lancé avec *stopService(Intent i)*
- La classe dérivée *IntentService* est utile pour implanter un service avec une thread de travail traitant les Intent
  - Il faut redéfinir *onHandleIntent(Intent i)*

Connexion à un service

- Un composant appelle *Context.bindService(Intent service, ServiceConnection conn, int flags)* pour se lier à un service
  - *ServiceConnection* est un listener avec les méthodes :
    - *onServiceConnected(ComponentName name, IBinder service)* : appelé lors de la connexion au service
    - *onServiceDisconnected(ComponentName name)* : appelé lorsque la connexion est perdue
  - *flags* : *BIND_AUTO_CREATE* (démarre le service automatiquement), *BIND_NOT_FOREGROUND* (n'accorde pas une priorité de 1er plan au service), *BIND_ABOVE_CLIENT* (priorité service > priorité client), *BIND_WAIVE_PRIORITY* (pas d'impact du contexte sur la priorité)
- La méthode *onBind()* du service est appelée : elle doit être redéfinie pour retourner un *IBinder* exposant les méthodes publiques.
- On utilise la méthode *Context.unbindService(ServiceConnection conn)* avec le *ServiceConnection* précédemment utilisé pour se déconnecter du service ; la déconnexion a également lieu lors de l'arrêt du composant appelant
- Lors de la déconnexion du service, sa méthode *onUnbind()* est appelée ; si cette méthode retourne true, une prochaine connexion appelera *onRebind()* et non *onBind()*.

Cycle de vie d'un service

http://developer.android.com/guide/components/bound-services.html



- Le service peut être tué par le système pour pénurie de ressources.
- Sa susceptibilité d'être tué dépend de sa priorité.
- Par défaut un service hérite de la plus forte priorité des contextes qui sont connectés
  - Il est possible de changer le comportement par défaut (drapeaux lors de *bindService()*, passage d'une activité en 1er plan avec notification
- Les méthodes du service doivent avoir une exécution rapide ; pour les longs travaux, on utilise des threads séparées
  - Il faut interrompre les threads de travail dans *onDestroy()*

Utilisation de AIDL

- *Android Interface Definition Language* (AIDL) = langage de définition d'interface (IDL) pour évocation distante de méthodes (RPC)
- Procédé
  - Définition des méthodes d'un service à exposer dans un fichier AIDL
  - Génération d'un proxy (client) pour l'appel des méthodes et d'un stub (serveur) pour l'implantation d'un service avec méthodes appelables
  - A l'exécution, le proxy envoie un message de son binder vers celui du service ; le service lui répond ; les messages sont constitués de types primitifs sérialisés dans des paquets (*Parcel*).

Langage AIDL

- Une interface AIDL ressemble à une interface Java à quelques exceptions près :
  - seulement des méthodes, pas de champ statique, pas de modificateurs
  - les types primitifs (ainsi que *String*, *CharSequence*, *List<T>* implanté comme une *ArrayList<T>*, *Map* non générique comme une *HashMap*) sont utilisables sans précaution particulière
  - pas de support des exceptions
  - pas de support d'héritage d'interface
  - les classes Java utilisées comme type de retour ou arguments doivent implanter l'interface *Parcelable* ; les types doivent être préfixés par in, out ou inout selon la directionnalité de la communication des objets
- Une interface AIDL est compilable en classe Java fournissant une souche (Stub), i.e. une classe abstraite Java offrant un serveur RPC avec méthodes à définir ; il faut redéfinir cette classe et la retourner par la méthode *onBind()* du service.

Parcelable

- *Parcelable* : pour la sérialisation/désérialisation de classes Java par le mécanisme d'IPC
- Une classe *T* implantant *Parcelable* doit :
  - Implanter une méthode *void writeToParcel(Parcel out)* pour sérialiser ses données dans un *Parcel*
  - Implanter une méthode *int describeContents()* retournant 0
  - Avoir un champ *static final CREATOR* de type *Parcelable.Creator<T>* avec les méthodes implantées :
    - *T createFromParcel(Parcel source)* pour désérialiser une instance depuis un *Parcel*
    - *T[] newArray(int size)* : pour créer un nouveau tableau de références nulles de taille size de type T[]
- Un en-tête dans un fichier .aidl doit être présent pour chaque classe complexe référencée dans une interface AIDL

Une bibliothèque matricielle

```
/* A parcelable matrix of integers */
public class Matrix implements Parcelable
{
 private final int rows, cols;
 private final int[][] content;

 public Matrix(int rows, int cols)
 {
  this.rows = rows; this.cols = cols;
  this.content = new int[rows][];
  // Initialize the matrix
  for (int i = 0; i < rows; i++) this.content[i] = new int[cols];
 }

 public static Matrix createIdentity(int n)
 {
  Matrix m = new Matrix(n, n);
  for (int i = 0; i < n; i++) m.set(i, i, 1);
  return m;
 }

 public int getRows() { return rows; }
 public int getCols() { return cols; }
 public int get(int i, int j) { return content[i][j]; }
 public void set(int i, int j, int value) { content[i][j] = value; }

 @Override public int describeContents()  { return 0; }

 @Override public void writeToParcel(Parcel dest, int flags)
 {
  dest.writeInt(rows); dest.writeInt(cols);
  // Write the matrix row by row
  for (int i = 0; i < rows; i++)
   for (int j = 0; j < cols; j++)
    dest.writeInt(content[i][j]);
 }

 public static final Parcelable.Creator<Matrix> CREATOR =
  new Parcelable.Creator<Matrix>()
  {
   @Override
   public Matrix createFromParcel(Parcel source)
   {
    int rows = source.readInt(), cols = source.readInt();
    Matrix m = new Matrix(rows, cols);
    for (int i = 0; i < rows; i++)
     for (int j = 0; j < cols; j++)
         m.set(i, j, source.readInt());
    return m;
   }

   @Override
   public Matrix[] newArray(int size) { return new Matrix[size]; }
  };

 @Override
 public String toString() { return Arrays.deepToString(content); }
}
```

```
package fr.upemlv.matrix;

import fr.upemlv.matrix.Matrix;

interface IMatrixMultiplier
{
        Matrix multiply(in Matrix m1, in Matrix m2);
        Matrix pow(in Matrix m, int power);
}
```

3. Définition AIDL IMatrixMultiplier pour un service de multiplication de matrices

1. Classe Matrix sérialisable dans un Parcel

2. En-tête AIDL pour la classe Matrix

```
package fr.upemlv.matrix;

parcelable Matrix;
```

Un service de multiplication de matrices

```
/** A service providing multiplications on matrices */
public class MatrixMultiplierService extends Service
{
 private final IBinder multiplier = new fr.upemlv.matrix.IMatrixMultiplier.Stub(
 {
  @Override
  public Matrix pow(Matrix m, int power) throws RemoteException
  {
   if (power == 0) return Matrix.createIdentity(m.getRows());
   if (power % 2 == 0)
   {
    Matrix m2 = pow(m, power / 2);
    return multiply(m2, m2);
   } else
   {
    // power % 2 == 1
    return multiply(m, pow(m, power-1));
   }
  }

  @Override
  public Matrix multiply(Matrix m1, Matrix m2)
   throws RemoteException
  {
   if (m1.getRows() != m2.getCols())
    throw new IndexOutOfBoundsException(
     "Dimension of matrices not compatible");
   Matrix r = new Matrix(m1.getRows(), m2.getCols());
   for (int i = 0; i < r.getRows(); i++)
    for (int j = 0; j < r.getCols(); j++)
    {
     int v = 0;
     for (int k = 0; k < m1.getCols(); k++)
      v += m1.get(i, k) * m2.get(k, j);
     r.set(i, j, v);
    }
   return r;
  }
 };

 @Override
 public IBinder onBind(Intent arg0) { return multiplier; }
}
```

On implante la souche de IMatrixMultiplier
La souche a été auto-générée depuis l'interface AIDL

On peut se connecter au service et appeler les méthodes distantes

```
ServiceConnection conn = new ServiceConnection()
  {

   @Override public void onServiceDisconnected(ComponentName name)
   {
    Toast.makeText(MatrixMultiplier.this, "Disconnected", Toast.LENGTH_LONG).show();
   }

   @Override public void onServiceConnected(ComponentName name, IBinder service)
   {
    Toast.makeText(MyActivity.this, "Connected", Toast.LENGTH_LONG).show();
    IMatrixMultiplier matmul = IMatrixMultiplier.Stub.asInterface(service) ;
    Matrix m = new Matrix(2,2);
    m.set(0,0, 1); m.set(0,1, 2);
    m.set(1,0, 3); m.set(1,1, 4);
    Matrix r = null;
    try {
     r = matmul.multiply(m, m);
    } catch (RemoteException e) {
     e.printStackTrace();
    }
    if (r != null)
     t.setText(r.toString());
   }
  };
boolean connected =
 bindService(new Intent().setClassName("fr.upemlv.matriservice",
  "fr.upemlv.matriservice.MatrixMultiplierService"),
  conn, Context.BIND_AUTO_CREATE);
```

Service local

```
public class LocalService implements Service
{
    public class LocalBinder extends Binder
    {
        LocalService getService()
        {
            return LocalService.this ;
        }
    }

    private final IBinder binder = new LocalBinder() ;

    @Override public IBinder onBind(Intent intent)
    {
        return binder;
    }

    ...
}
```

Inutile de mettre en œuvre une interface AIDL si le service n'est utilisé que localement dans l'application

Retour graphique d'un service

- Normalement un service ne peut afficher une vue graphique directement
- Solutions néanmoins possible pour retour graphique :
  - Usage d'un Toast (Toast.makeToast("texte", duration).show())
  - Utilisation d'une Notification (message apparaissant dans la tiroir escamotable de notification)
  - Récupération de données par une activité et affichage

Notifications

- Zone centralisée d'affichage d'informations (tiroir de notifications) pour l'utilisateur sous trois formats :
  - petite icône dans la barre de notification
  - vue normale dans le tiroir ouvert
  - vue étendue dans le tiroir ouvert
- Ajouter ou mettre à jour une notification :
  - Créer un objet *Notification* avec *NotificationCompat.Builder.build()*
  - Soumission de la notification au *NotificationManager* avec *NotificationManager.notify(String tag, int id, Notification notification)* : (tag, id) est un identificateur unique de notification dans l'application
    - On obtient une instance de *NotificationManager* avec *Context.getSystemService(Context.NOTIFICATION_SERVICE)*
- Supprimer une notification :
  - *NotificationManager.cancel(String tag, int id)*

Services en avant-plan

- Possibilité de mettre en avant-plan un service :
  - apparition dans la zone de notification
  - susceptibilité faible d'être tué
- Service.startForeground(int id, Notification notification)
- Exemples d'utilisation : enregistreur de traces GPS, lecteur de musique, ...

Chronomètre notificateur

```
public class ChronoService extends Service
{
	public static final String ACTION_START = ... ; public static final String ACTION_STOP = ... ; public static final String ACTION_RESET = ...

	// We don't use the RPC capability
	@Override public IBinder onBind(Intent intent) { return null; }

	private long cumulatedTime = 0; private long startTime = -1;

	private Thread updateThread = null;

	private Notification createNotification(String text)
	{
		NotificationCompat.Builder mBuilder = new NotificationCompat.Builder(this)
			.setSmallIcon(android.R.drawable.ic_media_play).setContentTitle("Chronometer").setContentText(text);
		Intent resultIntent = new Intent(this, NotifiedChronometer.class).putExtra("running", startTime >= 0);
		mBuilder.setContentIntent(PendingIntent.getActivity(this, 0, resultIntent, 0));
		return mBuilder.build();
	}

	@Override
	public int onStartCommand(Intent intent, int flags, int startId)
	{
		final NotificationManager nm = (NotificationManager)getSystemService(NOTIFICATION_SERVICE);
		if (intent == null) return 0;
		if (intent.getAction().equals(ACTION_START) && startTime == -1)
		{
			startTime = System.nanoTime();
			// Put in the foreground
			startForeground(0, createNotification("Running"));
			updateThread = new Thread(new Runnable()
			{
				@Override public void run()
				{
					while (! Thread.interrupted())
					{
						long time = (cumulatedTime + System.nanoTime() - startTime) / 1000000000;
						nm.notify(0, createNotification("Running: " + time + " s"));
						try { Thread.sleep(1000); }
						catch (InterruptedException e) { return ; }
					}
				}
			});
			updateThread.start();
		}
		else if (intent.getAction().equals(ACTION_STOP) && startTime >= 0)
		{
			cumulatedTime += System.nanoTime() - startTime;
			stopForeground(true); nm.cancel(0);
			updateThread.interrupt();
			startTime = -1;
			// stopSelf();
		}
		return 0;
	}

	@Override public void onDestroy()
	{
		if (updateThread != null) updateThread.interrupt();
	}

}
```

DreamService (depuis API17)

- Économiseur d'écran interactif s'affichant lors de la charge de l'appareil (affichage d'informations, météo...)
- Cycle de vie spécifique :
  - *onAttachedToWindow()* : le rêve est initialisé, on définit la vue affichée avec *setContentView(View)*
  - *onDreamingStarted()* : le rêve démarre avec ses éventuelles animations
  - *onDreamingStopped()* : le rêve est arrêté
  - *onDetachFromWindow()* : détachement du rêve avec libération des ressources utilisées
- Méthodes utiles :
  - *setInteractive(boolean)* pour indiquer si l'utilisateur peut interagir avec le rêve (sinon un clic le quitte)
  - *setFullScreen(boolean)* pour éventuellement cacher la barre de statut
  - *finish()* pour sortir du rêve

Exemple de déclaration dans le manifeste

```
<service android:name=".MyDream" android:exported="true"
    android:icon="@drawable/dream_icon" android:label="@string/dream_label" >
    <intent-filter>
        <action android:name="android.service.dreams.DreamService" />
        <category android:name="android.intent.category.DEFAULT" />
    </intent-filter>
</service>
```

Services systèmes

- Généralement services assurant un accès au matériel ou à des fonctionnalités globales
- Récupération d'une API publique par l'intermédiaire d'un Manager : *Context.getSystemManager(String)*
  - Les noms de services X_SERVICE sont disponibles comme constantes de Context : *WINDOW*, *LAYOUT*, *ACTIVITY*, *POWER*, *ALARM*, *NOTIFICATION*, *KEYGUARD*, *LOCATION*, *SEARCH*, *VIBRATOR*, *CONNECTIVITY*, *WIFI*, *INPUT_METHOD*, *UI_MODE*, *DOWNLOAD*...

Intent

- Message de communication de haut niveau définie dans l'API Java pour communiquer une action d'un composant à un autre
- Types de transmissions d'Intent :
  - Unicast : vers un composant explicitement nommé (classe Java)
  - Anycast : vers un composant assurant une certaine action (visualisation, édition...) sur un certain type de données
  - Multicast : diffusion vers des récepteurs de broadcast inscrits pour recevoir un type d'Intent
- Permet une communication unidirectionnelle ou bidirectionnelle (Intent réponse à un Intent demande)

Structure d'un Intent

- Elements constitutifs :
  - Action à réaliser
    constantes *ACTION_** dans Intent pour les actions systèmes
  - Donnée sur laquelle réaliser l'action sous forme d'URI
  - Paramètres additionnels (*EXTRA*)
- L'Intent est *Parcable* pour être envoyé entre binders
- Création d'un Intent :
  - *new Intent(String action, Uri uri)* pour appel implicite
  - *new Intent(Context c, Class<?> cls)* pour appel explicite ; pour un appel inter-application, on peut utiliser *setClassName(String package, String className)*
  - Ajout de catégories avec *addCategory(String category)* pou appel implicite : un composant doit valider toutes les catégories pour recevoir l'Intent
  - Méthodes *putExtra(String key, X value)* pour ajouter des arguments additionels
  - Ajout de drapeaux (constantes entières combinables avec |) avec *setFlags(int flags)*

Actions d'Intent pour activités

- ACTION_MAIN : action principale
- ACTION_VIEW : visualisation de données
- ACTION_ATTACH_DATA : attachement de données sur une URI
- ACTION_EDIT : éditer des données
- ACTION_PICK : choisir sur un répertoire de données
- ACTION_CHOOSER : force l'apparition d'un menu pour choisir l'application à utiliser ; EXTRA_INTENT contient l'Intent originel, EXTRA_TITLE le titre du menu
- ACTION_GET_CONTENT : obtenir un contenu selon son type MIME
- ACTION_SEND : envoyer un contenu spécifié par EXTRA_TEXT ou EXTRA_STREAM à un destinataire non spécifié
- ACTION_SENDTO : envoyer un contenu à un destinataire spécifié dans l'URI
- ACTION_INSERT : insère un élément vierge dans le répertoire spécifié par l'URI
- ACTION_DELETE : supprime l'élément désigné par l'URI
- ACTION_PICK_ACTIVITY : propose un menu de sélection d'activité selon l'EXTRA_INTENT fourni ; retourne la classe choisie mais ne la lance pas
- ACTION_SEARCH, ACTION_WEBSEARCH : réalise une recherche (chaîne à rechercher avec clé SearchManager.QUERY)
- ...

Quelques drapeaux d'Intent

- Autorisation temporaire sur données :
  - FLAG_GRANT_READ_URI_PERMISSION
  - FLAG_GRANT_WRITE_URI_PERMISSION
- Drapeaux pour activités :
  - FLAG_ACTIVITY_CLEAR_TASK : activité seule dans une tâche
  - FLAG_ACTIVITY_CLEAR_TOP : si l'activité tourne dans la tâche courante, elle est mise en haut de pile en supprimant les activités au-dessus
  - FLAG_ACTIVITY_CLEAR_WHEN_TASK_RESET : marque un point de retour dans la pile d'activités
  - FLAG_ACTIVITY_NEW_TASK : créé une nouvelle tâche pour l'activité
  - FLAG_ACTIVITY_REORDER_TO_FRONT : ramène l'activité en haut de pile de tâche si elle est déjà lancée
  - FLAG_ACTIVITY_SINGLE_TOP : pas de lancement si l'activité est déjà en haut de pile
  - ...
- Drapeau pour récepteur de broadcast :
  - FLAG_RECEIVER_REGISTERED_ONLY : cible uniquement les récepteurs enregistrées (pas de lancement de récepteu r déclaré dans le manifeste)

Démarrage d'activité

- Démarrage d'une activité sans retour :
  - *startActivity(Intent intent)*
- Démarrage d'une activité avec retour :
  - *startActivityForResult(Intent intent, int requestCode)*
    - Il faut redéfinir *onActivityResult(int requestCode, int resultCode, Intent data)* pour récupérer le résultat
- Si l'activité n'est pas trouvée (Intent implicite) : *ActivityNotFoundException*
- Au niveau de l'activité enfant démarrée :
  - Récupération de l'Intent de démarrage avec *getIntent()*
  - Spécification du résultat avec *setResult(int resultCode, Intent data)*
    - typiquement resultCode est RESULT_CANCELED ou RESULT_OK
    - il est possible d'incorporer des paramètres extra sur le data.

BroadcastReceiver

- Composant d'application pour la réception d'Intent diffusé
- Méthode *void onReceive(Context context, Intent intent)* pour traiter l'Intent diffusé
- Création et destruction de BroadcastReceiver :
  - Automatique à la diffusion d'un Intent intéressant si déclaration par balise <receiver> dans le manifeste
    - Certains événements de survenue fréquente ne peuvent pas être ainsi gérés
    - Ne permet pas de se connecter à un service (uniquement de lancer un service)
  - Manuelle en l'initialisant et l'enregistrant (et le désenregistrant) dans la thread principale de l'application (depuis un Context comme une activité) avec :
    - *Intent registerReceiver(BroadcastReceiver br, IntentFilter filter)*
      - L'Intent éventuellement retourné est un sticky broadcast vérifiant le filtre
      - Version longue : *registerReceiver(BroadcastReceiver br, IntentFilter filter, String permission, Handler scheduler)* pour indiquer une permission à vérifier pour l'émetteur ainsi qu'un Handler personnalisé
    - *void unregisterReceiver(BroadcastReceiver br)*

Types d'événements broadcast

- Broadcast normal/ordonné :
  - Normal : broadcast diffusé simultanément à tous les récepteurs, l'ordre de réception est indéterminé
  - Ordonné : broadcast propagé à une chaîne de récepteurs organisée par priorité décroissante ; le récepteur de plus forte priorité est le premier de la chaîne et peut décider de le propager (avec possible modification de l'Intent) ou non au prochain récepteur de la chaîne, et ainsi de suite
- Broadcast non collant/collant :
  - Non collant : le récepteur doit être enregistré avant l'émission du broadcast
  - Collant : l'enregistrement du récepteur permet d'obtenir un broadcast collant envoyé antérieurement

Envoi d'un événement broadcast

- Utilisation de méthodes sur *Context* (Activity, Service)
- Émission de broadcast non-collant :
  - *sendBroadcast(Intent intent, String permission)*
  - *sendOrderedBroadcast(Intent intent, String permission)*
- Émission de broadcast collant (nécessite la permission BROADCAST_STICKY) :
  - *sendStickyBroadcast(Intent intent)*
  - *sendStickyOrderedBroadcast(Intent intent, BroadcastReceiver finalReceiver, Handler scheduler, int initialCode, String initialData, Bundle initialExtras)*
- Suppression d'un broadcast collant :
  - *removeStickyBroadcast(Intent intent)*

Broadcast local

- *LocalBroadcastManager* permet de gérer des événements broadcast localement dans un processus (évite le coût de l'IPC)
- Méthodes utiles :
  - Méthode statique d'obtention : *LocalBroadcastManager.getInstance(Context context)*
  - Méthodes classiques :
    - *registerReceiver(BroadcastReceiver receiver, IntentFilter filter)*
    - *unregisterReceiver(BroadcastReceiver receiver)*
    - *sendBroadcast(Intent intent)*
  - Envoi synchrone d'un broadcast :
    - *sendBroadcastSync(Intent intent)*

Événements broadcast système

- Listés comme constantes dans Intent :
  - ACTION_TIME_TICK : émis toutes les minutes
  - ACTION_TIME_CHANGED, ACTION_TIMEZONE_CHANGED : remise à l'heure, changement de fuseau horaire
  - ACTION_BOOT_COMPLETED : fin de la séquence de démarrage, utile pour démarrer des services démons
  - ACTION_PACKAGE_{ADDED, CHANGED, REMOVED, RESTARTED, DATA_CLEARED} : événements concernant des modifications sur les applications (EXTRA_UID fournit le user ID)
  - ACTION_UID_REMOVED : suppression d'un user ID (EXTRA_UID fourni)
  - ACTION_BATTERY_CHANGED : broadcast sticky informant sur l'état de la batterie, ne peut être reçu par déclaration dans le manifeste
  - ACTION_POWER_{CONNECTED, DISCONNECTED} : état de chargement secteur
  - ACTION_SHUTDOWN : extinction du système
  - ...

Une jauge à batterie recevant ACTION_BATTERY_CHANGED

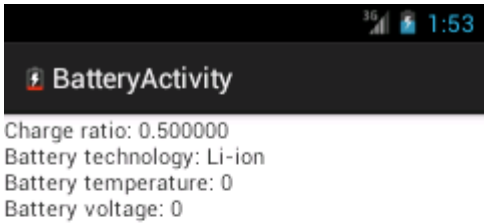
```
public class BatteryActivity extends Activity
{
	private BroadcastReceiver batteryReceiver = new BroadcastReceiver()
	{
		@Override
		public void onReceive(Context context, Intent intent)
		{
			// Fetch battery state data in the intent
			updateStatus(String.format(
				"Charge ratio: %f\nBattery technology: %s\nBattery temperature: %s\nBattery voltage: %d\n",
				(float)intent.getIntExtra(BatteryManager.EXTRA_LEVEL, -1) /
intent.getIntExtra(BatteryManager.EXTRA_SCALE, -1), // charge ratio
				intent.getStringExtra(BatteryManager.EXTRA_TECHNOLOGY),
				intent.getIntExtra(BatteryManager.EXTRA_TEMPERATURE, -1),
				intent.getIntExtra(BatteryManager.EXTRA_VOLTAGE, -1)),
				intent.getIntExtra(BatteryManager.EXTRA_ICON_SMALL, -1));
		}
	};

	@Override protected void onCreate(Bundle savedInstanceState)
	{
		super.onCreate(savedInstanceState);
		setContentView(R.layout.activity_battery);
	}

	private void updateStatus(String text, int logo)
	{
		TextView view = (TextView)findViewById(R.id.mainBatteryTextView);
		view.setText(text);
		getActionBar().setLogo(logo); // Set an icon for the battery state
	}

	@Override protected void onResume()
	{
		super.onResume();
		registerReceiver(batteryReceiver, new IntentFilter(Intent.ACTION_BATTERY_CHANGED));
	}

	@Override protected void onPause()
	{
		super.onPause();
		unregisterReceiver(batteryReceiver);
	}
}
```



Déclaration d'un BroadcastReceiver dans le manifeste

- Déclaration permettant de recevoir un Intent :
  - Validant un filtre spécifié
  - Ou adressé directement depuis une autre application au *BroadcastReceiver*

```
<receiver android:enabled=["true" | "false"]
          android:exported=["true" | "false"]
          android:icon="drawable resource"
          android:label="string resource"
          android:name="string"
          android:permission="string"
          android:process="string" >
    . . .
</receiver>
```

Instantiation automatique par le système

Accessible depuis d'autres applications

Classe implantant le receiver

Permission nécessaire pour envoyer un broadcast

Filtre d'Intent

- Définissable dynamiquement avec IntentFilter :
  - *IntentFilter(String action[, String dataType])*
- Spécifiable avec balises <intent-filter> dans <activity>, <service> ou <receiver>
  - <action android:name="nom.de.l.action" />
  - <category android:name="category" />
    - Quelques catégories : android.intent.category.{DEFAULT, BROWSABLE, TAB, ALTERNATIVE, LAUNCHER, HOME, PREFERENCE, TEST, ...}
  - <data android:mimeType="typeMIME" ... />

```
<intent-filter>
    <action android:name="android.intent.action.MAIN" />
    <category android:name="android.intent.category.LAUNCHER" />
</intent-filter>
```

```
 <intent-filter android:label="@string/jpeg_editor">
     <action android:name="android.intent.action.VIEW" />
     <action android:name="android.intent.action.EDIT" />
     <data android:mimeType="image/jpeg" />
 </intent-filter>
```

Quelques exemples

Intents implicites

- Intent implicite défini par :
  - une action
  - des catégories (*addCategory(String)*)
  - et une donnée (*setData(Uri)* pour l'URI, *setType(String)* pour le type MIME)
- Maintien d'un catalogue de composants associés à leur *IntentFilter* :
  - Le PackageManager cherche les composants traitant l'Intent implicite fourni (avec respect des permissions) ; on peut demander manuellement *Package.query{IntentActivities, IntentServices, BroadcastReceivers, ContentProviders}(Intent intent, int flags)*
  - Si plusieurs composants compatibles :
    - pour un broadcast normal, l'Intent est envoyé à tous les récepteurs
    - pour un broadcast ordonné, l'Intent est envoyé au récepteur de plus forte priorité
    - pour un Intent de démarrage d'activité ou de service, l'Intent est délivré au composant de plus forte priorité
    - pour démarrer une activité, il est possible d'afficher un popup de choix avec *Intent.createChooser(Intent target, String title)*