Servicios

Índice

En Android un `Service` es un componente que se ejecuta en segundo plano, sin interactuar con el usuario. Cualquier desarrollador puede crear nuevos Service en su aplicación. Cuentan con soporte multitarea real en Android, ya que pueden ejecutar en su propio proceso, a diferencia de los hilos de las `Activity`, que hasta cierto punto están conectados con el ciclo de vida de las actividades de Android.

Para comunicarse con los servicios se utiliza el mecanismo de comunicación entre procesos de Android, Inter-Process Communication (IPC). La interfaz pública de esta comunicación se describe en el lenguaje AIDL.

1. Servicios propios

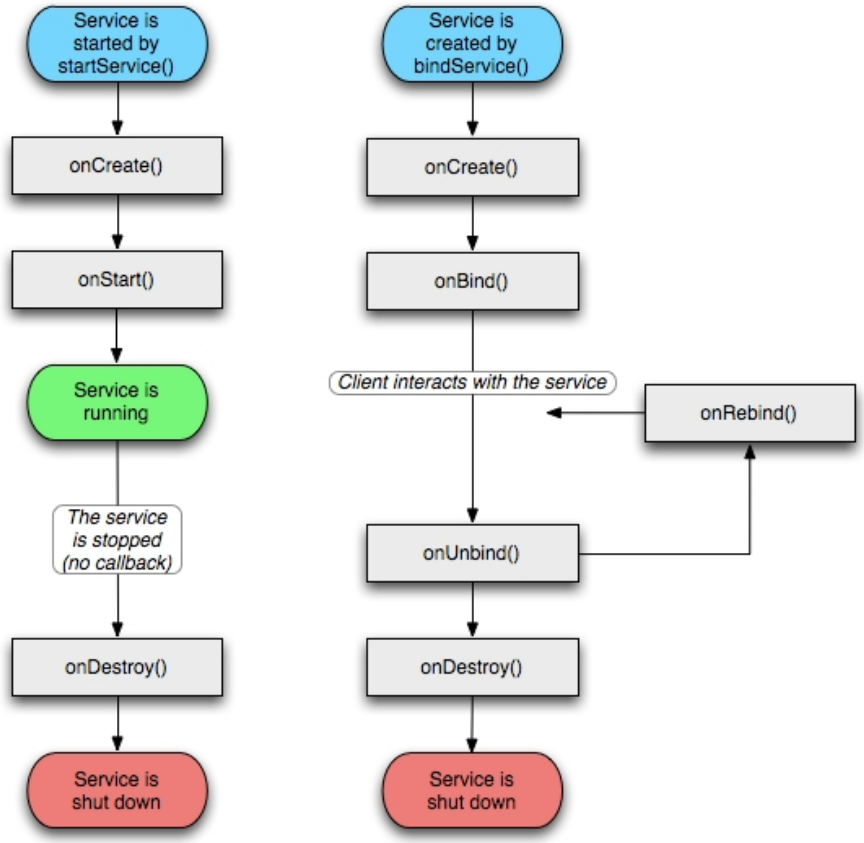
Podemos crear un servicio propio para realizar tareas largas que no requieran la interacción con el usuario, o bien para proveer de determinado tipo de funcionalidad a otras aplicaciones. Los servicios propios se deben declarar en el `AndroidManifest.xml` de la aplicación. En el código java se debe crear una clase que herede de la clase `Service` de forma directa o a través de alguna clase hija.

```
<service
        android:name="MiServicio"
        android:process=":mi_proceso"
        android:icon="@drawable/icon"
        android:label="@string/service_name"
        >
</service>
```

Para pedir permiso a otras aplicaciones para que utilicen el servicio se utiliza el atributo `uses-permission`.

Se puede especificar que un servicio debe ejecutarse en un proceso aparte a través del atributo `android:process`. De esta manera cuenta con su propio hilo y memoria y cualquier operación costosa que pueda tener no afectará al hilo principal donde estaría la interfaz gráfica. Si no lo declaramos con el atributo `android:process`, debemos utilizar hilos o `AsyncTask` para las tareas con procesamiento intensivo.

Una actividad puede iniciar un servicio a través del método `startService()` y detenerlo a través de `stopService()`. Si la actividad necesita interactuar con el servicio se utiliza el método `bindService()` del servicio. Esto requiere un objeto `ServiceConnection` que permite conectar con el servicio y que devuelve un objeto `IBinder`. Este objeto puede ser utilizado por la actividad para comunicarse con el servicio.



El ciclo de vida de un servicio de Android.

Una vez iniciado el servicio, se invoca su método `Service.onCreate()`. A continuación se invoca el método `Service.onStartCommand(Intent, int, int)` con la información del `Intent` proporcionado por la actividad. Para los servicios que se lanzan desde la aplicación principal, el método `onStartCommand()` se ejecuta en ese mismo hilo, el de la interfaz gráfica. Es bastante común crear un hilo nuevo y lanzarlo desde el `onStartCommand()` para realizar el procesamiento en segundo plano. Si se lanza ese hilo, la ejecución se devuelve rápidamente al método `onStartCommand()` dejándolo terminar en muy poco tiempo. A través del valor retorno de `onStartCommand()` podemos controlar el comportamiento de reinicio.

- `Service.START_STICKY` es el comportamiento estándar, en este caso el método `onStartCommand()` será invocado cada vez que el servicio sea reiniciado tras ser terminado por la máquina virtual. Nótese que en este caso al reiniciarlo el Intent que se le pasa por parámetro será `null`. `Service.START_STICKY` se utiliza típicamente en servicios que controlan sus propios estados y que se inician y terminan de manera explícita con `startService` y `stopService`. Por ejemplo, servicios que reproduzcan música u otras tareas de fondo.
- `Service.START_NOT_STICKY` se usa para servicios que se inician para procesar

acciones específicas o comandos. Normalmente utilizarán `stopSelf()` para terminarse una vez completada la tarea a realizar. Cuando la máquina virtual termine este tipo de servicios, éstos serán reiniciados sólo si hay llamadas de `starService` pendientes, de lo contrario el servicio terminará sin pasar por `onStartCommand`. Este modo es adecuado para servicios que manejen peticiones específicas, tales como actualizaciones de red o polling de red.

- `Service.START_REDELIVER_INTENT` es una combinación de los dos anteriores de manera que si el servicio es terminado por la máquina virtual, se reiniciará sólo si hay llamadas pendientes a `starService` o bien el proceso fue matado antes de hacer la llamada a `stopSelf()`. En este último caso se llamará a `onStartCommand()` pasándole el valor inicial del `Intent` cuyo procesamiento no fue completado. Con `Service.START_REDELIVER_INTENT` nos aseguramos de que el comando cuya ejecución se ha solicitado al servicio, sea completada hasta el final.

(En versiones anteriores a la 2.0 del Android SDK (nivel 5 de API) había que implementar el método `onStart` y era equivalente a sobrecargar `onStartCommand` y devolver `START_STICKY`).

El segundo parámetro del método `onStarCommand()` es un entero que contiene flags. Éstos se utilizan para saber cómo ha sido iniciado el servicio:

- `Service.START_FLAG_REDELIVERY` indica que el `Intent` pasado por parámetro es un reenvío porque la máquina virtual ha matado el servicio antes de ocurrir la llamada a `stopSelf`.
- `Service.START_FLAG_RETRY` indica que el servicio ha sido reiniciado tras una terminación anormal. Sólo ocurre si el servicio había sido puesto en el modo `Service.START_STICKY`.

Por ejemplo comprobamos si el servicio había sido reiniciado:

```
public class MiServicio extends Service {
        @Override
        public void onCreate() {

        }

        @Override
        public void onDestroy() {

        }

        @Override
        public int onStartCommand(Intent intent,
                                  int flags, int startId) {
                if((flags & START_FLAG_RETRY) == 0){
                        // El servicio se ha reiniciado
                } else {
                        // Iniciar el proceso de fondo
                }
                return Service.START_STICKY;
        }

        @Override
```

```
        public IBinder onBind(Intent arg0) {
                return null;
        }
}
```

El método `onBind()` debe ser sobrecargado obligatoriamente y se utiliza para la comunicación entre procesos.

1.1. Iniciar un servicio

Para iniciar un servicio de forma explícita se utiliza el método `startService()` con un `Intent`. Para detenerlo se utiliza `stopService()`:

```
        ComponentName servicio = startService(
                                 new Intent(getApplicationContext(),
MiServicio.class));
        // ...
        stopService(new Intent(getApplicationContext(),
                                                 servicio.getClass()));
```

1.1.1. Iniciar al arrancar

Hay aplicaciones que necesitan registrar un servicio para que se inicie al arrancar el sistema operativo. Para ello hay que registrar un `BroadcastReceiver` al evento del sistema `android.intent.action.BOOT_COMPLETED`. En el AndroidManifest tendríamos:

```
<uses-permission
 android:name="android.permission.RECEIVE_BOOT_COMPLETED" />
  <application android:icon="@drawable/cw"
               android:label="@string/app_name">
    <receiver android:name=".OnBootReceiver">
      <intent-filter>
        <action
         android:name="android.intent.action.BOOT_COMPLETED" />
      </intent-filter>
    </receiver>
  </application>
</manifest>
```

La actividad sobrecargaría el método `onReceive()`:

```
        public class MiReceiver extends BroadcastReceiver {
                @Override
                public void onReceive(Context context, Intent intent) {
                        Intent servicio = new Intent(context,
MiServicio.class);
                        context.startService(servicio);
                }
        }
```

Si la aplicación está instalada en la tarjeta SD, entonces no estará disponible en cuanto arranque el sistema. En este caso hay que registrarse para el evento

`android.intent.action.ACTION_EXTERNAL_APPLICATIONS_AVAILABLE`. A partir de Android 3.0 el usuario debe haber ejecutado la aplicación al menos una vez antes de que ésta pueda recibir el evento `BOOT_COMPLETED`.

1.1.2. Servicios prioritarios

Es posible arrancar servicios con la misma prioridad que una actividad que esté en el foreground para evitar que Android pueda matarlos por necesidad de recursos. Esto es peligroso porque si hay muchos servicios de foreground se degrada el rendimiento del sistema. Por esta razón al iniciar un servicio en el foreground se debe notificar al usuario.

Los servicios de foreground se inician realizando desde dentro del servicio una llamada al método `startForeground()`:

```
int NOTIFICATION_ID = 1;
Intent        intent        =
        new Intent(this, MiActividad.class);
PendingIntent pendingIntent =
        PendingIntent.getActivity(this, 1, intent, 0));
Notification  notification  =
        new Notification(R.drawable.icon,
                "Servicio prioritario iniciado",
                System.currentTimeMillis());
notification.setLatestEventInfo(this,
        "Servicio", "Servicio iniciado", pendingIntent);
notification.flags = notification.flags |
        Notification.FLAG_ONGOING_EVENT; //Mientras dure

startForeground(NOTIFICATION_ID, notification);
```

Esta notificación debe durar mientras el servicio esté en ejecución. Se puede volver del foregroudn con el método `stopForeground()`. En general los servicios no deben ser iniciados en el foreground.

1.2. Servicios y AsyncTask

Las operaciones lentas de un servicio deben realizarse en un hilo aparte. La manera más cómoda suele ser a través de una `AsyncTask`. En el siguiente ejemplo un servicio utiliza una `AsyncTask` para realizar una cuenta desde 1 hasta 100.

```
public class MiCuentaServicio extends Service {
        MiTarea miTarea;

        @Override
        public int onStartCommand(Intent intent,
                        int flags, int startId) {
                Log.i("SRV", "onStartCommand");
                miTarea.execute();
                return Service.START_STICKY;
        }

        @Override
        public void onCreate() {
```

```
		super.onCreate();
		Toast.makeText(this,"Servicio creado ...",
				Toast.LENGTH_LONG).show();
		Log.i("SRV","onCreate");
		miTarea = new MiTarea();
	}


	@Override
	public void onDestroy() {
		super.onDestroy();
		Toast.makeText(this,"Servicio destruido ...",
				Toast.LENGTH_LONG).show();
		Log.i("SRV","Servicio destruido");
		miTarea.cancel(true);
	}

	@Override
	public IBinder onBind(Intent arg0) {

		return null;
	}


	private class MiTarea
			extends AsyncTask<String, String, String>{
		private int i;
		boolean cancelado;

		@Override
		protected void onPreExecute() {
			super.onPreExecute();
			i = 1;
			cancelado = false;
		}

		@Override
		protected String doInBackground(String... params) {
			for(; i<100; i++){
				Log.i("SRV",
						"AsyncTask: "+"Cuento hasta "+i);
				publishProgress(""+i);
				try {
					Thread.sleep(5000);
				} catch (InterruptedException e) {
					e.printStackTrace();
				}
				if(cancelado)
					break;
			}
			return null;
		}

		@Override
		protected void onProgressUpdate(String... values) {
			Toast.makeText(getApplicationContext(),
					"Cuento hasta "+values[0],
					Toast.LENGTH_SHORT).show();
		}

		@Override
		protected void onCancelled() {
			super.onCancelled();
			cancelado = true;
		}
```

```
        }
}
```

Para completar el ejemplo, el código de una actividad con dos botones que inicien y detenga este servicio tendría el siguiente aspecto:

```
public class Main extends Activity {
        Main main;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        main = this;

        ((Button)findViewById(R.id.Button01)).setOnClickListener(
        new OnClickListener() {
                        @Override
                        public void onClick(View v) {
                                startService(new Intent(main,
                                        MiCuentaServicio.class));
                        }
                });
        ((Button)findViewById(R.id.Button02)).setOnClickListener(
        new OnClickListener() {
                        @Override
                        public void onClick(View v) {
                                stopService(new Intent(main,
                                        MiCuentaServicio.class));
                        }
                });
    }
}
```

2. Broadcast receiver

2.1. Declaración y registro en el Manifest

Un receptor de broadcast es una clase que recibe `Intents` generados a través del método `Context.sendBroadcast()`. La clase debe heredar de `BroadcastReceiver` y debe implementar el método `onReceive()`. Sólo durante la ejecución de este método el objeto estará activo, por tanto no se puede utilizar para hacer ninguna operación asíncrona. Concretamente, no se podría mostrar un diálogo ni realizar un `bind` a un servicio. Las alternativas serían usar el NotificationManager en el primer caso y `Context.startService()` en el segudo.

La clase que herede de `BroadcastReceiver` estar declarada en el `AndroidManifest.xml`. También se declaran los `Intent` que la clase recibirá:

```
<application>
        <!-- [...]  -->
```

```
        <receiver
                android:name=".paquete.MiBroadcastReceiver"
                android:enabled="false">
           <intent-filter>
              <action
android:name="android.intent.ACTION_TIMEZONE_CHANGED" />
                <action android:name="android.intent.ACTION_TIME" />
           </intent-filter>
        </receiver>
</application>
```

La clase declarada en el ejemplo siguiente recibiría un intent al cambiar la hora (debido a un ajuste del reloj) o al cambiar la zona horaria:

```
public class MiBroadcastReceiver extends BroadcastReceiver {

    @Override
    public void onReceive(Context context,
                                            Intent intent) {
        String action = intent.getAction();
        if (action.equals(Intent.ACTION_TIMEZONE_CHANGED)
                || action.equals(Intent.ACTION_TIME_CHANGED)) {

            //Ejemplo: Actualizar nuestro
            // Widget dependiente de la hora

        }
    }
}
```

Otro ejemplo típico es el de recibir el intent de una llamada de teléfono entrante. Habría que declarar el intent `android.intent.action.PHONE_STATE` en el `AndroidManifest.xml` y declarar el `BroadcastReceiver` con el siguiente código:

```
public class LlamadaReceiver extends BroadcastReceiver {

        @Override
        public void onReceive(Context context, Intent intent) {
                Bundle extras = intent.getExtras();
                if (extras != null) {
                        String state = extras.getString(
                                TelephonyManager.EXTRA_STATE);
                        if (state.equals(
TelephonyManager.EXTRA_STATE_RINGING)) {
                                String phoneNumber = extras.getString(
TelephonyManager.EXTRA_INCOMING_NUMBER);
                                Log.i("DEBUG", phoneNumber);
                        }
                }
        }
}
```

Hay intents para toda clase de eventos, como por ejemplo pulsar el botón de la cámara, batería baja, o incluso cuando se instala una nueva aplicación. Los componentes propios también pueden enviar `Broadcast`.

2.2. Registro dinámico

Un `BroadcastReceiver` se puede registrar a un `IntentFilter` dinámicamente en lugar de hacerlo a través del `AndroidManifest.xml`.

```
MiBroadcastReceiver intentReceiver = new MiBroadcastReceiver();

IntentFilter intentFilter =
        new IntentFilter(Intent.ACTION_CAMERA_BUTTON);
intentFilter.addAction(Intent.ACTION_PACKAGE_ADDED);

registerReceiver(intentReceiver, intentFilter);

unregisterReceiver(intentReceiver);
```

Es recomendable registrar el receiver en el método `onResume()` y desregistrarlo en el método `onPause()` de la actividad.

3. PendingIntents y servicios del sistema

Al pasar un `PendingIntent` a otra aplicación, por ejemplo, el Notification Manager, el Alarm Manager o aplicaciones de terceros, se le está dando permiso para ejecutar determinado código que nosotros definimos en nuestra propia aplicación.

El código a ejecutar lo colocaremos en un `BroadcastReceiver`:

```
public class MiBroadcastReceiver extends BroadcastReceiver {
        @Override
        public void onReceive(Context context, Intent intent) {
                Toast.makeText(context,
                        "Otra aplicación es la causa de esta tostada.",
                                Toast.LENGTH_LONG).show();
                // Código que queremos ejecutar
                // a petición de la otra aplicación
                // ...
        }

}
```

Sin olvidar declararlo en el `AndroidManifest.xml`, antes de cerrar la etiqueta de `</application>`:

```
<receiver android:name="MiBroadcastReceiver">
```

En este ejemplo se utilizará un `PendingIntent` para que el Alarm Manager pueda ejecutar el código de nuestro `BroadcastReceiver`. Este código pertenecería a algún método de la `Activity`:

```
Intent intent = new Intent(this, MiBroadcastReceiver.class);
PendingIntent pendingIntent = PendingIntent.getBroadcast(
                this.getApplicationContext(), 0, intent, 0);
AlarmManager alarmManager = (AlarmManager)
                                        getSystemService(ALARM_SERVICE);
alarmManager.set(AlarmManager.RTC_WAKEUP,
```

```
        System.currentTimeMillis()+5000, pendingIntent);
```

Android ofrece una serie de servicios predefinidos a los que se accede a través del método `getSystemService(String)`. La cadena debe ser una de las siguientes constantes: `WINDOW_SERVICE`, `LAYOUT_INFLATER_SERVICE`, `ACTIVITY_SERVICE`, `POWER_SERVICE`, `ALARM_SERVICE`, `NOTIFICATION_SERVICE`, `KEYGUARD_SERVICE`, `LOCATION_SERVICE`, `SEARCH_SERVICE`, `VIBRATOR_SERVICE`, `CONNECTIVITY_SERVICE`, `WIFI_SERVICE`, `INPUT_METHOD_SERVICE`, `UI_MODE_SERVICE`, `DOWNLOAD_SERVICE`.

En general los servicios obtenidos a través de esta API pueden estar muy relacionados con el contexto (`Context`) en el cuál fueron obtenidos, por tanto no conviene compartirlos con contextos diferentes (actividades, servicios, aplicaciones, proveedores).

3.1. AlarmManager para programar servicios

`AlarmManager` no sólo se puede utilizar para programar otros servicios sino que en la mayoría de los casos **se debe** utilizar. Un ejemplo sería el de programar un servicio que comprueba si hay correo electrónico o RSS. Se trata de una tarea periódica y el servicio no tiene por qué estar en ejecución todo el tiempo. De hecho un servicio que sólo se inicia con `startService()` y nunca se finaliza con `stopService()` se considera un "antipatrón de diseño" en Android.

Si un servicio cumple ese antipatrón, es posible que Android lo mate en algún momento. Si un servicio de verdad requiere estar todo el tiempo en ejecución, como por ejemplo, uno de voz IP (require estar conectado y a la escucha todo el tiempo), entonces habría que iniciarlo como servicio foreground para que Android no lo mate nunca.

El `AlarmManager` es la analogía del cron de Unix. La diferencia importante es que cron siempre continúa con su anterior estado mientras que `AlarmManager` empieza en blanco cada vez que arranca. Por tanto estamos obligados a volver a registrar nuestros servicios durante el arranque.

4. Comunicación entre procesos

La comunicación entre procesos se puede realizar tanto invocando métodos como pasando objetos con información. Para pasar información de una actividad a un servicio se pueden utilizar los `BroadcastReceiver`, pasando información extra, `Bundle`, en el `Intent` que se utiliza para iniciar un servicio, o bien haciendo un binding al servicio.

Vamos a empezar por el binding de una actividad a un servicio:

4.1. Atar actividades a servicios

Un tipo posible de comunicación es el de invocar los métodos de un servicio.

Atar o enlazar (to bind) una actividad a un servicio consiste en mantener una referencia a la instancia del servicio, permitiendo a la actividad realizar llamadas a métodos del servicio igual que se harían a cualquier otra clase accesible desde la actividad.

Para que un serivicio de soporte a binding hay que implementar su método `onBind()`. Éste devolverá una clase binder que debe implementar la interfaz `IBinder`. La implementación nos obliga a definir el método `getService()` del binder. En este método devolveremos la instancia al servicio.

```
public class MiServicio extends Service {

        private final IBinder binder = new MiBinder();

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

        public class MiBinder extends Binder {
                MiServicio getService() {
                        return MiServicio.this;
                }
        }

        // ...
}
```

La conexión entre el servicio y la actividad es representada por un objeto de clase `ServiceConnection`. Hay que implementar una nueva clase hija, sobrecargando el método `onServiceConnected()` y `onServiceDisconnected()` para poder obtener la referencia al servicio una vez que se ha establecido la conexión:

```
private MiServicio servicio; //La referencia al servicio

private ServiceConnection serviceConnection =
                                new ServiceConnection() {
        public void onServiceConnected(
                ComponentName className, IBinder service) {
                servicio = ((MiServicio.MiBinder)service).getService();
        }
        public void onServiceConnected(ComponentName className) {
                servicio = null;
        }
};
```

Finalmente, para realizar el binding hay que pasarle el intent correspondiente al método `Activity.bindService()`. El intent servirá para poder seleccionar qué servicio devolver. Tendríamos el siguiente código en `Activity.onCreate()`:

```
Intent intent = new Intent(MiActividad.this,
                                                MiServicio.this);
bindService(intent, serviceConnection,
                                                Conext.BIND_AUTO_CREATE);
```

Una vez establecido el binding, todos los métodos y propiedades públicos del servicio

estarán disponibles a través del objeto `servicio` del ejemplo.

4.2. Inter Process Communication

En Android cada aplicación se ejecuta en su propia "caja de arena" y no comparte la memoria con otras aplicaciones o procesos. Para comunicarse entre procesos Android implementa el mecanismo IPC, Inter Process Communication. Su protocolo requiere codificar y descodificar los distintos tipos de datos y para facilitar esta parte, Android ofrece la posibilidad de definir los tipos de datos con AIDL, Android Interface Definition Language. De esta manera para comunicarse entre dos aplicaciones o procesos hay que definir el AIDL, a partir de éste implementar un Stub para la comunicación con un servicio remoto y por último dar al cliente acceso al servicio remoto.

La interfaz de datos se define en un archivo .aidl. Por ejemplo el siguiente archivo es `/src/es/ua/jtech/IServicio.aidl`:

```
package es.ua.jtech;

interface IServicio {
        // Los valores pueden ser: in, out, inout.
        String saludo(in String nombre, in String apellidos);
}
```

Una vez creado el anterior archivo, la herramienta Eclipse+AIDL generará un archivo .java, para el ejemplo sería el `/src/es/ua/jtech/IServicio.java`

En nuestro servicio implementaremos el método `onBind()` para que devuelva un stub que cumple la interfaz anteriormente definida.

```
public class Servicio extends Service{
        @Override
        public IBinder onBind(Intent intent) {
                return new IServicio.Stub() {
                        public int saludo(String nombre, String apellidos)
throws RemoteException {
                                return "Hola, "+ nombre + " " + apellidos;
                        }
                };
        }
        // ...
}
```

Ahora hay que obtener acceso al servicio desde la aplicación a través de `ServiceConnection`. Dentro de nuestra `Activity` tendríamos:

```
public class MiActividad extends Activity {
        IServicio servicio;
        MiServicioConnection connection;

        class MiServicioConnection implements ServiceConnection {

                public void onServiceConnected(ComponentName name,
```

```
                                                IBinder service) {
                        servicio = IServicio.Stub.asInterface(
                                                (IBinder) service);
                }

                public void onServiceDisconnected(
                                        ComponentName name) {
                        servicio = null;
                }

        }

        @Override
        public void onCreate(Bundle savedInstanceState) {
                super.onCreate(savedInstanceState);
                setContentView(R.layout.main);

                connection = new MiServicioConnection();

                // El intent se crea así porque
                // la clase podría estar en otra
                // aplicación:
                Intent intent = new Intent();
                intent.setClassName("es.ua.jtech",
                        es.ua.jtech.Servicio.class.getName());

                bindService(intent, connection,
                        Context.BIND_AUTO_CREATE);

                String saludo = servicio.saludo(
                                                "Boyan", "Bonev");
                // ...
        }

        @Override
        protected void onDestroy() {
                unbindService(connection);
        }

}
```

Los tipos que se permiten en AIDL son:

- Valores primitivos como int, float, double, boolean, etc.
- `String` y `CharSequence`
- `java.util.List` y `java.util.Map`
- Intefaces definidos en AIDL, requiere definir el import
- Clases Java que implementen la interfaz `Parcelable` que Android usa para permitir la serialización. También requiere definir el import

Si el servicio está en otro proyecto necesitaremos declararlo con un intent filter en su Manifest

```
<service android:export="true" name="es.ua.jtech.Servicio">
  <intent-filter>
    <action android:name="es.ua.jtech.IServicio"/>
  </intent-filter>
</service>
```

y crear el intent con ese nombre de acción:

```
Intent intent = new Intent("es.ua.jtech.IServicio");
```

4.3. Otras formas de comunicación

4.3.1. Broadcast privados

Enviar un `Intent` a través del método `sendBroadcast()` es sencillo pero por defecto cualquier aplicación podría tener un `BroadcastReceiver` que obtenga la información de nuestro intent. Para evitarlo se puede utilizar el método `Intent.setPackage()` que restringirá el broadcast a determinado paquete.

4.3.2. PendingResult

Otra manera de que un servicio envíe información a una actividad es a través del método `createPendingResult()`. Se trata de un método que permite a un `PendingIntent` disparar el método `Activity.onActivityResult`. Es decir, el servicio remoto llamaría a `send()` con un `PendingIntent` con la información, de manera análoga al `setResult()` que ejecuta una actividad que fue llamada con `startActivityForResult()`. Dentro de `Activity.onActivityResult` se procesaría la información del `Intent`.

Este es un mecanismo que sólo funciona con actividades.