Chapter 14

Writing Android Services

In a previous chapter (p 67) we introduced the concept of a Service in Android. There is nothing mysterious about them; they are merely tasks that run in the background, independent of the main Activity. Note that, although they are separate components, they still run in context of the main Activity process and any operation that is CPU intensive or will block (like network operations) should be run in a separate thread.

To define a new service, your class needs to extend the class `Service`. Let's look at some skeleton code (*this is for Android 2.0 or later, for previous versions see the `onStart()` method*):

```
public class NewService extends Service {

    @Override
    public void onCreate() {
        ...
    }

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

    @Override
    public int onStartCommand(Intent intent, int flags,
                              int startId) {
        return Service.START_STICKY;
    }
}
```

As usual, we have the `onCreate` API, which is called when the service is first created. The `onBind` API is called to return the communication channel to the service. The service may return `null` if clients can not bind to the service, i.e. will only start or stop the service via `startService` or `stopService` and not call `bindService`. Although overriding the `onStartCommand` API is optional, it is often the place where services will start background processing. Note that this API may be called several times

during a Service lifecycle. The flags parameter is used to discover how the service was started, with

- `START_FLAG_REDELIVERY` – indicates that the Intent parameter is a redelivery, or
- `START_FLAG_RETRY` – indicates that the Service was restarted after an abnormal termination.

Implementing the (deprecated) `onStart` API in Android SDK 2.0 or later is equivalent to the `onStartCommand` implementation in the code snippet on the previous page.

The Service must also be registered in the Android manifest. To do this we use the `<service>` element tag. You probably also want to include an Intent Filter in order to specify which services your Service provides. We will cover this a bit later in this chapter.

Communicating with Services

Although it's perfectly possible and valid to have a service running in the background without any direct communication channel to the foreground Activity, you probably want to interact with the Service in some way to control its operation or retrieve some status of information.

In some of these cases you want to bind to the Service, as opposed to just starting it, which maintains a reference to it, allowing the client to make method calls to the Service. In this chapter we will define a new remote Service that allows clients to call methods exported via an implemented interface.

The client interaction when binding to a Service is slightly different. The connection is represented as a `ServiceConnection`, where you will need to override the `onServiceConnected` and `onServiceDisconnected` methods to retrieve a reference when the connection is established, for example:

```
/* Our service */
IDmsFriends fService;

/**
 * Connection for interacting with the service.
 */
private ServiceConnection mFriendsConnection =
                                    new ServiceConnection() {
    public void onServiceConnected(ComponentName className,
                                   IBinder service) {
```

```
            // Create the connection to our service.
            fService = IDmsFriends.Stub.asInterface(service);
        }

        public void onServiceDisconnected(
                                        ComponentName className) {
            fService = null;
        }
    };
```

To bind to the service the client calls `bindService`, and once bound, the public interface is accessible through the object instance retrieve in the `onServiceConnected` handler (`fService` in the example above). Note that all methods are executed synchronously (the local method blocks until the remote method finishes), even if there is no return value.

Defining the interface in AIDL

On the Android platform, one process can not normally access the memory of another process. So to talk, they need to decompose their objects into primitives that the operating system can understand, and "marshall" the object across that process boundary for you. Luckily Android provides the AIDL tool to make this a lot easier it for you.

AIDL (Android Interface Definition Language) is an IDL language used to generate code that enables two processes on an Android-powered device to communicate using interprocess communication (IPC). So in our case we have code in one process (the Activity) that needs to call methods on an object in another process (our Service). To do this we will use AIDL to generate code to marshall the parameters between them.

AIDL is an IPC mechanism similar to COM or Corba, but lighter weight. It uses a proxy class to pass values between the client and the implementation.

The following steps are required to implement an IPC service using AIDL.

1. Create your `.aidl` file - This file defines an interface (`MyInterface.aidl`) that defines the methods and fields available to a client.

2. Add the `.aidl` file to the makefile – if using the Eclipse ADT Plugin, this is all done for you. Android includes the compiler, called `aidl`, in the `tools/` directory.

3. Implement the interface methods - The AIDL compiler creates an interface in Java from your AIDL interface. This interface has an inner abstract class named Stub that inherits the interface (and implements a few additional methods necessary for the IPC call).

4. Expose the interface to clients - For a service, we override `Service.onBind()` to return an instance of the class that implements the interface.

Figure 23 shows how the Android generated classes fit together:

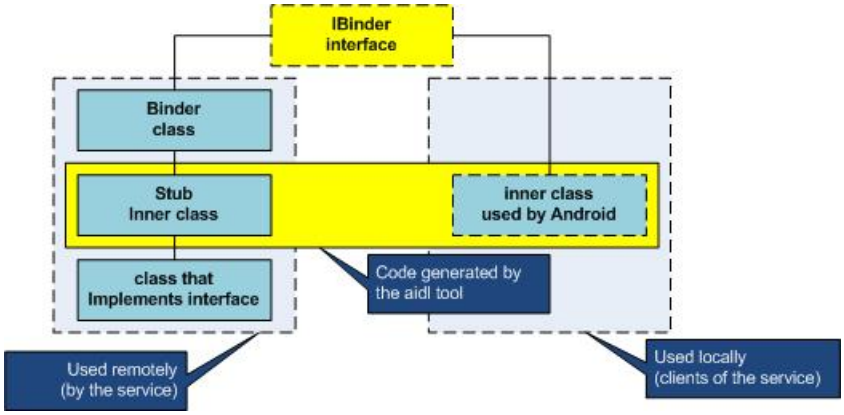

Figure 23 : Android IPC inner classes

The IPC mechanism works as follows: You declare the RPC interface you want to implement using the simple IDL (interface definition language). Using that declaration, you run the `aidl` tool which generates a Java interface definition that must be made available to both the local and the remote process.

Typically, the remote process would be managed by a service and would have both the interface file generated by the `aidl` tool and the Stub subclass which implements the actual RPC methods. On the other hand, the clients of the service would have only the interface file generated by the `aidl` tool.

The WhereIsJohny Service

To see how this all fit together, lets add a Service to our application that implements a well defined interface through which the clients can control and query the service.

We'll start with the AIDL definition.