CHAPTER 14

Advanced Service Patterns

In *The Busy Coder's Guide to Android Development*, we covered how to create and consume services and covered some basic service patterns. However, services can certainly do more than what is covered in those introductory patterns. In this chapter, we will examine some more powerful options for services, including remote services and using services in the role of "cron jobs" or "scheduled tasks".

Remote Services

By default, services are used within the application that publishes them. However, it is possible to expose services for other applications to take advantage of. These are basically inter-process editions of the binding pattern and command patterns outlined in *The Busy Coder's Guide to Android Development*.

We start with an explanation of the inter-process communication (IPC) mechanism offered in Android for allowing services to work with clients in other applications. Then, we move onto the steps to allow a client to connect to a remote service, before describing how to turn an ordinary service into a remote one. We then look at how one can implement a callback system to allow services, through IPC, to pass information back to clients. After noting the possibility of binder errors, we wrap by examining other ways to get results from remote services, back to clients, without going through binding.

When IPC Attacks!

Services will tend to offer IPC as a means of interacting with activities or other Android components. Each service declares what methods it is making available over IPC; those methods are then available for other components to call, with Android handling all the messy details involved with making method calls across component or process boundaries.

The guts of this, from the standpoint of the developer, is expressed in AIDL: the Android Interface Description Language. If you have used IPC mechanisms like COM, CORBA, or the like, you will recognize the notion of IDL. AIDL describes the public IPC interface, and Android supplies tools to build the client and server side of that interface.

With that in mind, let's take a look at AIDL and IPC.

Write the AIDL

IDLs are frequently written in a "language-neutral" syntax. AIDL, on the other hand, looks a lot like a Java interface. For example, here is some AIDL:

```
package com.commonsware.android.advservice;

// Declare the interface.
interface IScript {
  void executeScript(String script);
}
```

As with a Java interface, you declare a package at the top. As with a Java interface, the methods are wrapped in an interface declaration (`interface IScript { ... }`). And, as with a Java interface, you list the methods you are making available.

The differences, though, are critical.

First, not every Java type can be used as a parameter. Your choices are:

- Primitive values (`int`, `float`, `double`, `boolean`, etc.)
- `String` and `CharSequence`
- `List` and `Map` (from `java.util`)
- Any other AIDL-defined interfaces
- Any Java classes that implement the `Parcelable` interface, which is Android's flavor of serialization

In the case of the latter two categories, you need to include `import` statements referencing the names of the classes or interfaces that you are using (e.g., `import com.commonsware.android.ISomething`). This is true even if these classes are in your own package – you have to import them anyway.

Next, parameters can be classified as `in`, `out`, or `inout`. Values that are `out` or `inout` can be changed by the service and those changes will be propagated back to the client. Primitives (e.g., `int`) can only be `in`; we included in for the AIDL for `enable()` just for illustration purposes.

Also, you cannot throw any exceptions. You will need to catch all exceptions in your code, deal with them, and return failure indications some other way (e.g., error code return values).

Name your AIDL files with the `.aidl` extension and place them in the proper directory based on the package name.

When you build your project, either via an IDE or via Ant, the `aidl` utility from the Android SDK will translate your AIDL into a server stub and a client proxy.

Implement the Interface

Given the AIDL-created server stub, now you need to implement the service, either directly in the stub, or by routing the stub implementation to other methods you have already written.

The mechanics of this are fairly straightforward:

- Create a private instance of the AIDL-generated `.Stub` class (e.g., `IScript.Stub`)
- Implement methods matching up with each of the methods you placed in the AIDL
- Return this private instance from your `onBind()` method in the `Service` subclass

Note that AIDL IPC calls are synchronous, and so the caller is blocked until the IPC method returns. Hence, your services need to be quick about their work.

We will see examples of service stubs later in this chapter.

A Consumer Economy

Of course, we need to have a client for AIDL-defined services, lest these services feel lonely.

Bound for Success

To use an AIDL-defined service, you first need to create an instance of your own `ServiceConnection` class. `ServiceConnection`, as the name suggests, represents your connection to the service for the purposes of making IPC calls.

Your `ServiceConnection` subclass needs to implement two methods:

1. `onServiceConnected()`, which is called once your activity is bound to the service
2. `onServiceDisconnected()`, which is called if your connection ends normally, such as you unbinding your activity from the service

Each of those methods receives a `ComponentName`, which simply identifies the service you connected to. More importantly, `onServiceConnected()` receives an `IBinder` instance, which is your gateway to the IPC interface. You will want to convert the `IBinder` into an instance of your AIDL interface class, so you can use IPC as if you were calling regular methods on a regular Java class (`IScript.Stub.asInterface(binder)`).

To actually hook your activity to the service, call `bindService()` on the activity:

```
bindService(new Intent("com.commonsware.android.advservice.IScript"),
            svcConn, Context.BIND_AUTO_CREATE);
```

The `bindService()` method takes three parameters:

1. An `Intent` representing the service you wish to invoke
2. Your `ServiceConnection` instance
3. A set of flags – most times, you will want to pass in `BIND_AUTO_CREATE`, which will start up the service if it is not already running

After your `bindService()` call, your `onServiceConnected()` callback in the `ServiceConnection` will eventually be invoked, at which time your connection is ready for use.

Request for Service

Once your service interface object is ready (`IScript.Stub.asInterface(binder)`), you can start calling methods on it as you need to. In fact, if you disabled some widgets awaiting the connection, now is a fine time to re-enable them.

However, you will want to trap two exceptions. One is `DeadObjectException` – if this is raised, your service connection terminated unexpectedly. In this case, you should unwind your use of the service, perhaps by calling `onServiceDisconnected()` manually, as shown above. The other is `RemoteException`, which is a more general-purpose exception indicating a

cross-process communications problem. Again, you should probably cease your use of the service.

Getting Unbound

When you are done with the IPC interface, call `unbindService()`, passing in the `ServiceConnection`. Eventually, your connection's `onServiceDisconnected()` callback will be invoked, at which point you should null out your interface object, disable relevant widgets, or otherwise flag yourself as no longer being able to use the service.

You can always reconnect to the service, via `bindService()`, if you need to use it again.

Service From Afar

Everything from the preceding two sections could be used by local services. In fact, that prose originally appeared in *The Busy Coder's Guide to Android Development* specifically in the context of local services. However, AIDL adds a fair bit of overhead, which is not necessary with local services. After all, AIDL is designed to marshal its parameters and transport them across process boundaries, which is why there are so many quirky rules about what you can and cannot pass as parameters to your AIDL-defined APIs.

So, given our AIDL description, let us examine some implementations, specifically for remote services.

Our sample applications – shown in the `AdvServices/RemoteService` and `AdvServices/RemoteClient` sample projects – convert our Beanshell demo from *The Busy Coder's Guide to Android Development* into a remote service. If you actually wanted to use scripting in an Android application, with scripts loaded off of the Internet, isolating their execution into a service might not be a bad idea. In the service, those scripts are sandboxed, only able to access files and APIs available to that service. The scripts cannot access your own application's databases, for example. If the script-executing

service is kept tightly controlled, it minimizes the mischief a rogue script could possibly do.

Service Names

To bind to a service's AIDL-defined API, you need to craft an Intent that can identify the service in question. In the case of a local service, that Intent can use the local approach of directly referencing the service class.

Obviously, that is not possible in a remote service case, where the service class is not in the same process, and may not even be known by name to the client.

When you define a service to be used by remote, you need to add an intent-filter element to your service declaration in the manifest, indicating how you want that service to be referred to by clients. The manifest for `RemoteService` is shown below:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest android:versionCode="1"
          android:versionName="1.0"
          package="com.commonsware.android.advservice"
          xmlns:android="http://schemas.android.com/apk/res/android">

  <uses-sdk android:minSdkVersion="3"
            android:targetSdkVersion="6" />
  <supports-screens android:largeScreens="false"
                    android:normalScreens="true"
                    android:smallScreens="false" />
  <application android:icon="@drawable/cw"
               android:label="@string/app_name">
    <service android:name=".BshService">
      <intent-filter>
        <action android:name="com.commonsware.android.advservice.IScript" />
      </intent-filter>
    </service>
  </application>
</manifest>
```

Here, we say that the service can be identified by the name `com.commonsware.android.advservice.IScript`. So long as the client uses this name to identify the service, it can bind to that service's API.

In this case, the name is not an implementation, but the AIDL API, as you will see below. In effect, this means that so long as some service exists on the device that implements this API, the client will be able to bind to something.

The Service

Beyond the manifest, the service implementation is not too unusual. There is the AIDL interface, `IScript`:

```
package com.commonsware.android.advservice;

// Declare the interface.
interface IScript {
  void executeScript(String script);
}
```

And there is the actual service class itself, `BshService`:

```
package com.commonsware.android.advservice;

import android.app.Service;
import android.content.Intent;
import android.os.IBinder;
import android.util.Log;
import bsh.Interpreter;

public class BshService extends Service {
  private final IScript.Stub binder=new IScript.Stub() {
    public void executeScript(String script) {
      executeScriptImpl(script);
    }
  };
  private Interpreter i=new Interpreter();

  @Override
  public void onCreate() {
    super.onCreate();

    try {
      i.set("context", this);
    }
    catch (bsh.EvalError e) {
      Log.e("BshService", "Error executing script", e);
    }
  }
```

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

  @Override
  public void onDestroy() {
    super.onDestroy();
  }

  private void executeScriptImpl(String script) {
    try {
      i.eval(script);
    }
    catch (bsh.EvalError e) {
      Log.e("BshService", "Error executing script", e);
    }
  }
}
```

If you have seen the service and Beanshell samples in *The Busy Coder's Guide to Android Development* then this implementation will seem familiar. The biggest thing to note is that the service returns no result and handles any errors locally. Hence, the client will not get any response back from the script – the script will just run. In a real implementation, this would be silly, and we will work to rectify this later in this chapter.

Also note that, in this implementation, the script is executed directly by the service on the calling thread. One might think this is not a problem, since the service is in its own process and, therefore, cannot possibly be using the client's UI thread. However, AIDL IPC calls are synchronous, so the client will still block waiting for the script to be executed. This too will be corrected later in this chapter.

The Client

The client – `BshServiceDemo` out of `AdvServices/RemoteClient` – is a fairly straight-forward mashup of the service and Beanshell clients, with two twists:

```
package com.commonsware.android.advservice.client;

import android.app.Activity;
```

```
import android.app.AlertDialog;
import android.content.ComponentName;
import android.content.Context;
import android.content.Intent;
import android.content.ServiceConnection;
import android.os.Bundle;
import android.os.IBinder;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;
import com.commonsware.android.advservice.IScript;

public class BshServiceDemo extends Activity {
  private IScript service=null;
  private ServiceConnection svcConn=new ServiceConnection() {
    public void onServiceConnected(ComponentName className,
                                   IBinder binder) {
      service=IScript.Stub.asInterface(binder);
    }

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

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

    Button btn=(Button)findViewById(R.id.eval);
    final EditText script=(EditText)findViewById(R.id.script);

    btn.setOnClickListener(new View.OnClickListener() {
      public void onClick(View view) {
        String src=script.getText().toString();

        try {
          service.executeScript(src);
        }
        catch (android.os.RemoteException e) {
          AlertDialog.Builder builder=
                    new AlertDialog.Builder(BshServiceDemo.this);

          builder
            .setTitle("Exception!")
            .setMessage(e.toString())
            .setPositiveButton("OK", null)
            .show();
        }
      }
    });

    bindService(new Intent("com.commonsware.android.advservice.IScript"),
```

```
                svcConn, Context.BIND_AUTO_CREATE);
  }

  @Override
  public void onDestroy() {
    super.onDestroy();

    unbindService(svcConn);
  }
}
```

One twist is that the client needs its own copy of `IScript.aidl`. After all, it is a totally separate application, and therefore does not share source code with the service. In a production environment, we might craft and distribute a JAR file that contains the `IScript` classes, so both client and service can work off the same definition (see the upcoming chapter on reusable components). For now, we will just have a copy of the AIDL.

Then, the `bindService()` call uses a slightly different `Intent`, one that references the name the service is registered under, and that is the glue that allows the client to find the matching service.

If you compile both applications and upload them to the device, then start up the client, you can enter in Beanshell code and have it be executed by the service. Note, though, that you cannot perform UI operations (e.g., raise a `Toast`) from the service. If you choose some script that is long-running, you will see that the Go! button is blocked until the script is complete:

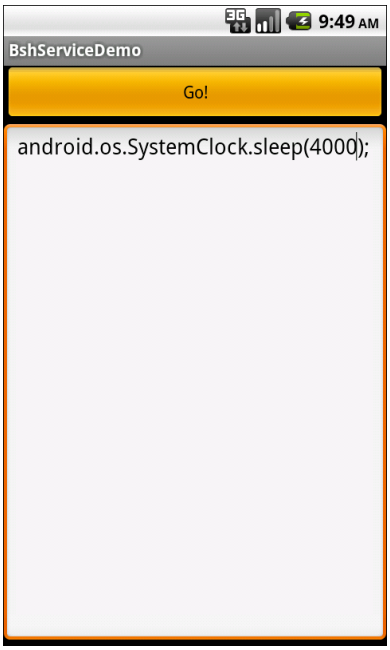

Figure 60. The BshServiceDemo application, running a long script

Servicing the Service

The preceding section outlined two flaws in the implementation of the Beanshell remote service:

1. The client received no results from the script execution
2. The client blocked waiting for the script to complete

If we were not worried about the blocking-call issue, we could simply have the `executeScript()` exported API return some sort of result (e.g., `toString()` on the result of the Beanshell `eval()` call). However, that would not solve the fact that calls to service APIs are synchronous even for remote services.

Another approach would be to pass some sort of callback object with `executeScript()`, such that the server could run the script asynchronously and invoke the callback on success or failure. This, though, implies that there is some way to have the activity export an API to the service.

Fortunately, this is eminently doable, as you will see in this section, and the accompanying samples (`AdvServices/RemoteServiceEx` and `AdvServices/RemoteClientEx`).

Callbacks via AIDL

AIDL does not have any concept of direction. It just knows interfaces and stub implementations. In the preceding example, we used AIDL to have the service flesh out the stub implementation and have the client access the service via the AIDL-defined interface. However, there is nothing magic about services implementing and clients accessing – it is equally possible to reverse matters and have the client implement something the service uses via an interface.

So, for example, we could create an `IScriptResult.aidl` file:

```
package com.commonsware.android.advservice;

// Declare the interface.
interface IScriptResult {
  void success(String result);
  void failure(String error);
}
```

Then, we can augment `IScript` itself, to pass an `IScriptResult` with `executeScript()`:

```
package com.commonsware.android.advservice;

import com.commonsware.android.advservice.IScriptResult;

// Declare the interface.
interface IScript {
  void executeScript(String script, IScriptResult cb);
}
```

Notice that we need to specifically import `IScriptResult`, just like we might import some "regular" Java interface. And, as before, we need to make sure the client and the server are working off of the same AIDL definitions, so these two AIDL files need to be replicated across each project.

But other than that one little twist, this is all that is required, at the AIDL level, to have the client pass a callback object to the service: define the AIDL for the callback and add it as a parameter to some service API call.

Of course, there is a little more work to do on the client and server side to make use of this callback object.

Revising the Client

On the client, we need to implement an `IScriptResult`. On `success()`, we can do something like raise a `Toast`; on `failure()`, we can perhaps show an `AlertDialog`.

The catch is that we cannot be certain we are being called on the UI thread in our callback object.

So, the safest way to do that is to make the callback object use something like `runOnUiThread()` to ensure the results are displayed on the UI thread:

```
private final IScriptResult.Stub callback=new IScriptResult.Stub() {
  public void success(final String result) {
    runOnUiThread(new Runnable() {
      public void run() {
        successImpl(result);
      }
    });
  }

  public void failure(final String error) {
    runOnUiThread(new Runnable() {
      public void run() {
        failureImpl(error);
      }
    });
  }
};

private void successImpl(String result) {
  Toast
    .makeText(BshServiceDemo.this, result, Toast.LENGTH_LONG)
    .show();
}

private void failureImpl(String error) {
```

```
  AlertDialog.Builder builder=
            new AlertDialog.Builder(BshServiceDemo.this);

  builder
    .setTitle("Exception!")
    .setMessage(error)
    .setPositiveButton("OK", null)
    .show();
}
```

And, of course, we need to update our call to `executeScript()` to pass the callback object to the remote service:

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

  Button btn=(Button)findViewById(R.id.eval);
  final EditText script=(EditText)findViewById(R.id.script);

  btn.setOnClickListener(new View.OnClickListener() {
    public void onClick(View view) {
      String src=script.getText().toString();

      try {
        service.executeScript(src, callback);
      }
      catch (android.os.RemoteException e) {
        failureImpl(e.toString());
      }
    }
  });

  bindService(new Intent("com.commonsware.android.advservice.IScript"),
              svcConn, Context.BIND_AUTO_CREATE);
}
```

Revising the Service

The service also needs changing, to both execute the scripts asynchronously and use the supplied callback object for the end results of the script's execution.

`BshService` from `AdvServices/RemoteServiceEx` uses the `LinkedBlockingQueue` pattern to manage a background thread. An `ExecuteScriptJob` wraps up the script and callback; when the job is eventually processed, it uses the

callback to supply the results of the `eval()` (on success) or the message of the `Exception` (on failure):

```
package com.commonsware.android.advservice;

import android.app.Service;
import android.content.Intent;
import android.os.IBinder;
import android.util.Log;
import java.util.concurrent.LinkedBlockingQueue;
import bsh.Interpreter;

public class BshService extends Service {
  private final IScript.Stub binder=new IScript.Stub() {
    public void executeScript(String script, IScriptResult cb) {
      executeScriptImpl(script, cb);
    }
  };
  private Interpreter i=new Interpreter();
  private LinkedBlockingQueue<Job> q=new LinkedBlockingQueue<Job>();

  @Override
  public void onCreate() {
    super.onCreate();

    new Thread(qProcessor).start();

    try {
      i.set("context", this);
    }
    catch (bsh.EvalError e) {
      Log.e("BshService", "Error executing script", e);
    }
  }

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

  @Override
  public void onDestroy() {
    super.onDestroy();

    q.add(new KillJob());
  }

  private void executeScriptImpl(String script,
                                 IScriptResult cb) {
    q.add(new ExecuteScriptJob(script, cb));
  }

  Runnable qProcessor=new Runnable() {
```

```
  public void run() {
    while (true) {
      try {
        Job j=q.take();

        if (j.stopThread()) {
          break;
        }
        else {
          j.process();
        }
      }
      catch (InterruptedException e) {
        break;
      }
    }
  }
};

class Job {
  boolean stopThread() {
    return(false);
  }

  void process() {
    // no-op
  }
}

class KillJob extends Job {
  @Override
  boolean stopThread() {
    return(true);
  }
}

class ExecuteScriptJob extends Job {
  IScriptResult cb;
  String script;

  ExecuteScriptJob(String script, IScriptResult cb) {
    this.script=script;
    this.cb=cb;
  }

  void process() {
    try {
      cb.success(i.eval(script).toString());
    }
    catch (Throwable e) {
      Log.e("BshService", "Error executing script", e);

      try {
        cb.failure(e.getMessage());
```

```
        }
        catch (Throwable t) {
          Log.e("BshService",
                "Error returning exception to client",
                t);
        }
      }
    }
  }
}
```

Notice that the service's own API just needs the `IScriptResult` parameter, which can be passed around and used like any other Java object. The fact that it happens to cause calls to be made synchronously back to the remote client is invisible to the service.

The net result is that the client can call the service and get its results without tying up the client's UI thread.

You may be wondering why we do not simply use an `AsyncTask`. The reason is that remote service methods exposed by AIDL are not invoked on the main application thread – one of the few places in Android where Android calls your code from a background thread. An `AsyncTask` expects to be created on the main application thread.

The Bind That Fails

Sometimes, a call to `bindService()` will fail for some reason. The most common cause will be an invalid `Intent` – for example, you might be trying to bind to a `Service` that you failed to register in the manifest. The `bindService()` method returns a `boolean` value indicating whether or not there was an immediate problem, so you can take appropriate steps.

For local services, this is usually just a coding problem. For remote services, though, it could be that the service you are trying to work with has not been installed on the device. You have two approaches for dealing with this:

1. You can watch for `bindService()` to return `false` and assume that means the service is not installed

2. You can use introspection to see if the service is indeed installed before you even try calling `bindService()`

We will look at introspection techniques later in this book.

If the Binding Is Too Tight

Sometimes, binding is more than you really need.

Sending data to a remote service is easy, even without binding. Just package some data in `Intent` extras and use that `Intent` in a `startService()` call. The remote service can grab those extras and operate on that data. This works best with an `IntentService`, which does three things to assist with this pattern:

1. It passes the `Intents`, with their extras, to your code in `onHandleIntent()` on a background thread, so you can take as long as you want to process them
2. It queues up `Intents`, so if another one arrives while you are working on a previous one, there is no problem
3. It automatically shuts down the service when there is no more work to be done

The biggest issue is getting results back to the client. There is no possibility of a callback if there is no binding.

Fortunately, Android offers some alternatives that work nicely with this approach.

Private Broadcasts

The concept of a "private broadcast" may seem like an oxymoron, but it is something available to you in Android.

Sending a broadcast `Intent` is fairly easy – create the `Intent` and call `sendBroadcast()`. However, by default, any application could field a `BroadcastReceiver` to watch for your broadcast. This may or may not concern you.

If you feel that "spies" could be troublesome, you can call `setPackage()` on your `Intent`, to limit the distribution of the broadcast. With `setPackage()`, only components in the named application will be able to receive the broadcast. You can even arrange to send the name of the package via an extra to the remote service, so the service does not need to know the name of the package in advance.

Pending Results

Another way for a remote service to send data back to your activity is via `createPendingResult()`. This is a method on `Activity` that gives you a `PendingIntent` set up to trigger `onActivityResult()` in your activity. In essence, this is the underpinnings behind `startActivityForResult()` and `setResult()`. You create the `PendingIntent` with `createPendingResult()` and pass it in an `Intent` extra to the remote service. The remote service can call `send()` on the `PendingIntent`, supplying an `Intent` with return data, just like `setResult()` would do in an activity started via `startActivityForResult()`. In your activity's `onActivityResult()`, you would get and inspect the returned `Intent`.

This works nicely for activities, but this mechanism does not work for other components. Hence, you cannot use this technique for one service calling another remote service, for example.

BshService, Revisited

Let us take a closer look at those two techniques, as implemented in `AdvServices/RemoteClientUnbound` and `AdvServices/RemoteServiceUnbound`. These versions of the Beanshell sample are designed to demonstrate both private broadcasts and pending results.

AlarmManager: Making the Services Run On Time

A common question when doing Android development is "where do I set up `cron` jobs?"

The `cron` utility – popular in Linux – is a way of scheduling work to be done periodically. You teach `cron` what to run and when to run it (e.g., weekdays at noon), and `cron` takes care of the rest. Since Android has a Linux kernel at its heart, one might think that `cron` might literally be available.

While `cron` itself is not, Android does have a system service named `AlarmManager` which fills a similar role. You give it a `PendingIntent` and a time (and optionally a period for repeating) and it will fire off the `Intent` as needed. By this mechanism, you can get a similar effect to `cron`.

There is one small catch, though: Android is designed to run on mobile devices, particularly ones powered by all-too-tiny batteries. If you want your periodic tasks to be run even if the device is "asleep", you will need to take a fair number of extra steps, mostly stemming around the concept of the `WakeLock`.

The WakefulIntentService Pattern

Most times, if you are bothering to get control on a periodic basis, you will want to do so even when the device is asleep. For example, if you are writing an email client, you will want to go get new emails even if the device is asleep, so the user has all of the emails immediately upon the next time the device wakes up. You might even want to raise a `Notification` based upon the arrived emails.

Alarms that wake up the device are possible, but tricky, so we will examine `AlarmManager` in the context of this scenario. And, to make that work, we are going to use the `WakefulIntentService` – another of the CommonsWare Android Components, available as open source for you to use. In particular, we will be looking at the demo project from the `WakefulIntentService`

GitHub project, in addition to the implementation of `WakefulIntentService` itself, specifically looking at these from the standpoint of using `AlarmManager` for scheduled tasks.

Note that to use `WakefulIntentService` you will need the `WAKE_LOCK` permission in your application, and if you are using the `AlarmListener` approach described in this chapter, you will also need the `RECEIVE_BOOT_COMPLETED` permission.

Step #1: Create an Alarm Listener

The `WakefulIntentService` offers an `AlarmListener` interface. If you create one and register it properly, the `WakefulIntentService` will handle much of the details of arranging to schedule your alarms whenever they need to be scheduled, plus do the actual periodic work that you need the alarms for.

An `AlarmListener` needs to implement three methods:

- `scheduleAlarms()`, which is where you will teach Android when an alarm is supposed to go off
- `sendWakefulWork()`, which is where you tell `WakefulIntentService` what should occur when an alarm goes off
- `getMaxAge()`, where you indicate how long of a time between alarms should elapse before `WakefulIntentService` assumes that the alarms were lost and need to be re-scheduled

For example, from the `WakefulIntentService` demo project, here is an `AlarmListener` implementation named `AppListener`:

```
package com.commonsware.cwac.wakeful.demo;

import android.app.AlarmManager;
import android.app.PendingIntent;
import android.content.Context;
import android.os.SystemClock;
import com.commonsware.cwac.wakeful.WakefulIntentService;

public class AppListener implements WakefulIntentService.AlarmListener {
  public void scheduleAlarms(AlarmManager mgr, PendingIntent pi,
```

```
                            Context ctxt) {
    mgr.setInexactRepeating(AlarmManager.ELAPSED_REALTIME_WAKEUP,
                            SystemClock.elapsedRealtime()+60000,
                            AlarmManager.INTERVAL_FIFTEEN_MINUTES, pi);
  }

  public void sendWakefulWork(Context ctxt) {
    WakefulIntentService.sendWakefulWork(ctxt, AppService.class);
  }

  public long getMaxAge() {
    return(AlarmManager.INTERVAL_FIFTEEN_MINUTES*2);
  }
}
```

Your `scheduleAlarms()` method will be passed everything you should need to define when you want to get control again in the future. Mostly, this is in the form of an `AlarmManager`, to save you from having to call `getSystemService()` on a `Context` to get an `AlarmManager` instance. `AlarmManager` has three methods for defining when your scheduled tasks should run:

- `setRepeating()`, where you stipulate when the first time is you want your task to run, and how frequently thereafter
- `setInexactRepeating()`, which does the same thing but gives Android the flexibility to slightly tweak your schedule for more efficiency
- `set()`, which is for "one-shot" alarms, where you want a task to run once at a designated time

The `AlarmListener` pattern is for use with the first two approaches, not for one-shot alarms via `set()`.

Each of these methods takes a "type" of alarm. There are four types, based upon two orthogonal axes of decision:

- Do you want the device to wake up out of sleep mode to perform the task?
- Do you want to specify the time for the first task in terms of actual real world time, or simply an amount of time from now?

There are four values, therefore, you can choose from for the first parameter:

- `ELAPSED_REALTIME` indicates you want to specify a time relative to `SystemClock.elapsedRealtime()` (a good choice for specifying a time relative to now), and you do not want to wake up the device for the alarms
- `ELAPSED_REALTIME_WAKEUP` is the same as the above one, except that you do want the device to wake up out of sleep mode
- `RTC` indicates that you want to specify a time relative to `System.currentTimeMillis()` (a good choice for specifying a real-world time, as you can get a suitable value for this out of a `Calendar` object), and you do not want to wake up the device for the alarms
- `RTC_WAKEUP` is the same as the above one, except that you do want the device to wake up out of sleep mode

So, in the above sample, we are using `setInexactRepeating()`, indicating that we want the first event to occur one minute (60,000 milliseconds) from now, then recur every 15 minutes or so.

The `sendWakefulWork()` method is where you indicate to `WakefulIntentService` what is to be done when each alarm goes off. Most times, you will simply turn around and call `sendWakefulWork()` on `WakefulIntentService` itself, identifying your custom subclass of `WakefulIntentService`, where your business logic will reside. In the above sample, our business logic is in `AppService`, which we will examine shortly.

The `getMaxAge()` method... takes a bit of explaining, which we will do later in this chapter. For the moment, take it on faith that a value of twice your period is a reasonable approach.

Register the AlarmReceiver

The `WakefulIntentService` library supplies a class named `AlarmReceiver`. You will need to add this to your manifest via a `<receiver>` element, to get

control when the device wakes up. For example, here is the manifest from the demo project:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.cwac.wakeful.demo"
  android:versionCode="4"
  android:versionName="1.0"
  android:installLocation="auto">

  <uses-sdk
    android:minSdkVersion="3"
    android:targetSdkVersion="6"/>

  <supports-screens
    android:largeScreens="false"
    android:normalScreens="true"
    android:smallScreens="false"/>

  <uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED"/>
  <uses-permission android:name="android.permission.WAKE_LOCK"/>

  <application android:label="@string/app_name">
    <receiver android:name="com.commonsware.cwac.wakeful.AlarmReceiver">
      <intent-filter>
        <action android:name="android.intent.action.BOOT_COMPLETED"/>
      </intent-filter>

      <meta-data
        android:name="com.commonsware.cwac.wakeful"
        android:resource="@xml/wakeful"/>
    </receiver>

    <service android:name=".AppService">
    </service>

    <activity
      android:label="@string/app_name"
      android:name=".DemoActivity"
      android:theme="@android:style/Theme.NoDisplay">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>
  </application>

</manifest>
```

Our `<receiver>` points to the `com.commonsware.cwac.wakeful.AlarmReceiver` class and ties it to a `BOOT_COMPLETED` action. This requires a `<uses-permission>`

element to request the RECEIVE_BOOT_COMPLETED permission, so users will know at install time that we need to get control when the device turns on.

Our <receiver> element also has a <meta-data> element. These elements are used to supply more information about a component than the manifest XML calls for. In our case, we are indicating that there is additional data, known as com.commonsware.cwac.wakeful, available via a wakeful XML resource (/res/xml/wakeful.xml). That resource contains a single WakefulIntentService element:

```
<WakefulIntentService
  listener="com.commonsware.cwac.wakeful.demo.AppListener"
/>
```

The one attribute in that element, listener, points to our AlarmListener implementation from the previous step.

Step #2: Do Your Wakeful Work

Our AppService will get control in a method named doWakefulWork(). The doWakefulWork() method has similar semantics to the onHandleIntent() of a regular IntentService – we get control in a background thread, and the service will shut down once the method returns if there is no other outstanding work. The difference is that WakefulIntentService will keep the device awake while doWakefulWork() is doing its work.

In this case, AppService just logs a message to LogCat:

```
package com.commonsware.cwac.wakeful.demo;

import android.content.Intent;
import android.util.Log;
import com.commonsware.cwac.wakeful.WakefulIntentService;

public class AppService extends WakefulIntentService {
  public AppService() {
    super("AppService");
  }

  @Override
  protected void doWakefulWork(Intent intent) {
```

```
    Log.i("AppService", "I'm awake! I'm awake! (yawn)");
  }
}
```

And that's it. Those two steps – plus `WakefulIntentService` – is all you need to get control on a periodic basis to do work, waking up the phone as needed.

Of course, we have not yet scheduled any alarms. To understand how to do that, we need to first understand *when* to do that.

When Alarms Come and Go

When your app is initially installed by the user, none of your code is immediately executed. Hence, you will not have had a chance to register any alarms. So, the first place you need to think about registering your alarms is when your launcher activity is executed. In the `WakefulIntentService` demo, there is a `DemoActivity` that does just that:

```
package com.commonsware.cwac.wakeful.demo;

import android.app.Activity;
import android.os.Bundle;
import android.os.StrictMode;
import android.widget.Toast;
import com.commonsware.cwac.wakeful.WakefulIntentService;

public class DemoActivity extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    StrictMode.setThreadPolicy(new StrictMode.ThreadPolicy.Builder()
    .detectAll()
    .penaltyLog()
    .build());


    WakefulIntentService.scheduleAlarms(new AppListener(),
                                        this, false);

    Toast.makeText(this, "Alarms active!",
                   Toast.LENGTH_LONG).show();

    finish();
```

```
  }
}
```

The `scheduleAlarms()` method on `WakefulIntentService` takes an implementation of your `AlarmListener`, a `Context`, and an optional boolean flag indicating whether or not the alarms should be forced to be scheduled, even if they probably are already active. By default, this is set to `true`, though our `DemoActivity` has it as `false`.

So, the first time your activity is executed, the alarms will be scheduled, following the rules set down in your `AlarmListener`. In an ideal world, this is all you would need. If this were an ideal world, though, this part of the book would not need to be written. Since you are reading it, you can tell that this is not an ideal world.

So, when else do we need to schedule alarms?

On a Reboot

A Unix-style cron daemon will pick up jobs where it left off after a reboot. Similarly, Windows "Scheduled Tasks" will resume after a reboot.

On the other hand, `AlarmManager` in Android is wiped clean, starting with zero scheduled alarms. The argument that we have been given is that the core Android team does not want apps to schedule alarms and "forget" about them, causing them to pile up and slow down the device. While this is a noble objective, it does cause some annoyance, as we need to get control at boot time and arrange to reschedule the alarms.

After Being Force-Stopped

Users can go into the Settings application, find your application, and click a "Force Stop" button. When they do this, your app moves into a "stoppped" state. Nothing of your code will run again – including any `BroadcastReceiver` components that you have defined – until the user proactively runs one of your activities again, typically from the launcher. Hence, any user that

force-stops your app will cause you to not receive any scheduled alarms ever again, as the existing alarms are removed and you will not be able to get control again at boot time.

This is why we are looking at scheduling the alarms every time the `DemoActivity` runs. Not only might it be the very first time the user has launched our activity, but it might be the very first time the user has launched our activity after being force-stopped.

Fortunately, `WakefulIntentService` handles these scenarios for you, so long as you call `scheduleAlarms()` whenever your app is launched. This is where `getMaxAge()` comes into play – if `WakefulIntentService` detects that an alarm has not gone off in the number of milliseconds you return from `getMaxAge()`, it assumes that the user force-stopped your application and it schedules the alarms again.

The "Wakeful" of WakefulIntentService

Now, let us take a look "under the covers" to see how `WakefulIntentService` actually keeps the device awake long enough for `doWakefulWork()` to do its, um, wakeful work.

Concept of WakeLocks

Most of the time in Android, you are developing code that will run while the user is actually using the device. Activities, for example, only really make sense when the device is fully awake and the user is tapping on the screen or keyboard.

Particularly with scheduled background tasks, though, you need to bear in mind that the device will eventually "go to sleep". In full sleep mode, the display, main CPU, and keyboard are all powered off, to maximize battery life. Only on a low-level system event, like an incoming phone call, will anything wake up.

Another thing that will partially wake up the phone is an `Intent` raised by the `AlarmManager`. So long as broadcast receivers are processing that `Intent`, the `AlarmManager` ensures the CPU will be running (though the screen and keyboard are still off). Once the broadcast receivers are done, the `AlarmManager` lets the device go back to sleep.

You can achieve the same effect in your code via a `WakeLock`, obtained via the `PowerManager` system service. When you acquire a "partial `WakeLock`" (`PARTIAL_WAKE_LOCK`), you prevent the CPU from going back to sleep until you release said `WakeLock`. By proper use of a partial `WakeLock`, you can ensure the CPU will not get shut off while you are trying to do background work, while still allowing the device to sleep most of the time, in between alarm events.

However, using a `WakeLock` is a bit tricky, particularly when responding to an alarm `Intent`, as we will see in the next few sections.

The WakeLock Problem

For a `_WAKEUP` alarm, the `AlarmManager` will arrange for the device to stay awake, via a `WakeLock`, for as long as the `BroadcastReceiver`'s `onReceive()` method is executing. For some situations, that may be all that is needed. However, `onReceive()` is called on the main application thread, and Android will kill off the receiver if it takes too long.

Your natural inclination in this case is to have the `BroadcastReceiver` arrange for a `Service` to do the long-running work on a background thread, since `BroadcastReceiver` objects should not be starting their own threads. Perhaps you would use an `IntentService`, which packages up this "start a `Service` to do some work in the background" pattern. And, given the preceding section, you might try acquiring a partial `WakeLock` at the beginning of the work and release it at the end of the work, so the CPU will keep running while your `IntentService` does its thing.

This strategy will work...some of the time.

The problem is that there is a gap in `WakeLock` coverage, as depicted in the following diagram:

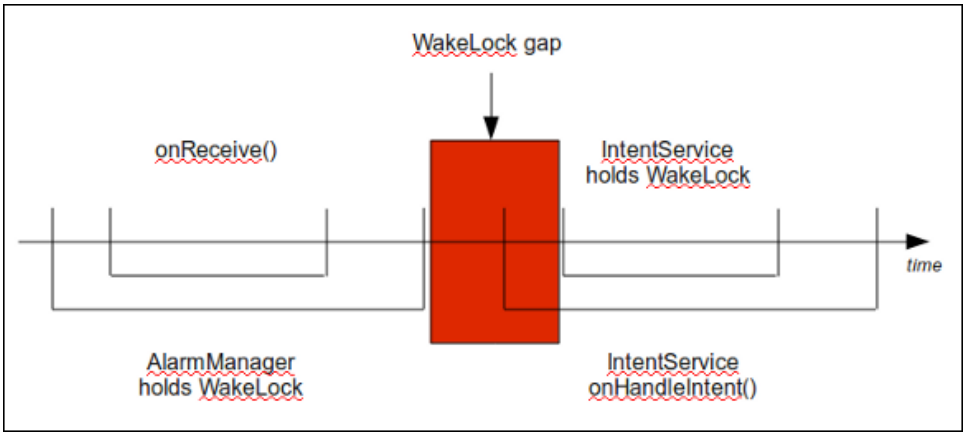

Figure 61. The WakeLock gap

The `BroadcastReceiver` will call `startService()` to send work to the `IntentService`, but that service will not start up until after `onReceive()` ends. As a result, there is a window of time between the end of `onReceive()` and when your `IntentService` can acquire its own `WakeLock`. During that window, the device might fall back asleep. Sometimes it will, sometimes it will not.

What you need to do, instead, is arrange for overlapping `WakeLock` instances. You need to acquire a `WakeLock` in your `BroadcastReceiver`, during the `onReceive()` execution, and hold onto that `WakeLock` until the work is completed by the `IntentService`:

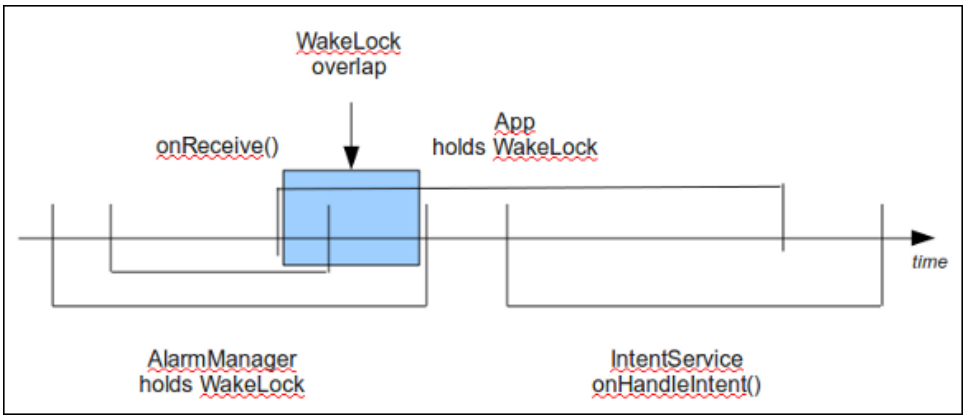

Figure 62. The WakeLock overlap

Then you are assured that the device will stay awake as long as the work remains to be done.

WakefulIntentService and WakeLocks

By now, you have noticed that the `WakefulIntentService` recipe does not have you manage your own `WakeLock`. That is because `WakefulIntentService` handles it for you. One reason why `WakefulIntentService` exists is to manage that `WakeLock`, because `WakeLocks` suffer from one major problem: they are not `Parcelable`, and therefore cannot be passed in an `Intent` extra. Hence, for our `BroadcastReceiver` and our `WakefulIntentService` to use the same `WakeLock`, they have to be shared via a static data member... which is icky. `WakefulIntentService` is designed to hide this icky part from you, so you do not have to worry about it.

Inside WakefulIntentService

With all that behind us, now we can take a look at some pieces of how `WakefulIntentService` performs its magic.

scheduleAlarms()

Our `DemoActivity` called `scheduleAlarms()`. The job of `scheduleAlarms()` is to get your `AlarmListener` to schedule the alarms, if it appears that this is necessary:

```
public static void scheduleAlarms(AlarmListener listener,
                                  Context ctxt) {
  scheduleAlarms(listener, ctxt, true);
}

public static void scheduleAlarms(AlarmListener listener,
                                  Context ctxt,
                                  boolean force) {
  SharedPreferences prefs=ctxt.getSharedPreferences(NAME, 0);
  long lastAlarm=prefs.getLong(LAST_ALARM, 0);

  if (lastAlarm==0 || force ||
      (System.currentTimeMillis()>lastAlarm &&
          System.currentTimeMillis()-lastAlarm>listener.getMaxAge())) {
    AlarmManager mgr=(AlarmManager)ctxt.getSystemService(Context.ALARM_SERVICE);
    Intent i=new Intent(ctxt, AlarmReceiver.class);
    PendingIntent pi=PendingIntent.getBroadcast(ctxt, 0,
                                                i, 0);

    listener.scheduleAlarms(mgr, pi, ctxt);
  }
}
```

The `scheduleAlarms()` method will think that scheduling alarms is necessary if:

- You forced it via passing `true` to the `force` parameter, or
- If it cannot find the last time the alarm went off, via a value stored in a service-specific `SharedPreferences` object, or
- If the last time the alarm went off is longer than your desired maximum age, suggesting that perhaps the user force-stopped your app

In principle, we could just always schedule the alarms. If you call `setRepeating()` (or kin) on `AlarmManager` for a `PendingIntent`, any existing alarm for an equivalent `PendingIntent` is replaced. Here, "equivalent" means that the underlying Intent has the same action, component, categories, and MIME type.

The downside of always scheduling the alarms is that if you use `setRepeating()`, the alarm will next go off at your designated time, which may be far sooner than the existing alarm would call for. For example, suppose that you want to get control once per day. If you call `setRepeating()` or `setInexactRepeating()` every time the user launches your main activity, you will trigger the alarm to go off each of those times, in addition to the once per day. Using `setRepeating()` with a specific time of the day (e.g., 3am) will not have that effect, but `setRepeating()` with "do the first one a minute from now" would.

AlarmReceiver

The `WakefulIntentService`-supplied `AlarmReceiver` will get control when the device powers on (via our manifest entry) and when the alarm goes off. At those times, `onReceive()` will be called:

```
@Override
 public void onReceive(Context ctxt, Intent intent) {
   AlarmListener listener=getListener(ctxt);

   if (listener!=null) {
     if (intent.getAction()==null) {
       SharedPreferences
prefs=ctxt.getSharedPreferences(WakefulIntentService.NAME, 0);

       prefs
         .edit()
         .putLong(WakefulIntentService.LAST_ALARM, System.currentTimeMillis())
         .commit();

       listener.sendWakefulWork(ctxt);
     }
     else {
       WakefulIntentService.scheduleAlarms(listener, ctxt, true);
     }
   }
 }
```

If the `Intent` that was broadcast does not have an action, we assume this is the broadcast from `AlarmManager`, and so we call `sendWakefulWork()` on your `AlarmListener` after updating our private `SharedPreferences` to note that the alarm went off (for use by `scheduleAlarms()`). If, however, the action is not-

null, that means we are being started from the `BOOT_COMPLETED` broadcast, in which case we call `scheduleAlarms()` on your `AlarmListener`.

Where does the `AlarmListener` object come from? A private `getListener()` method that takes advantage of the `<meta-data>` from the manifest:

```
@SuppressWarnings("unchecked")
private WakefulIntentService.AlarmListener getListener(Context ctxt) {
  PackageManager pm=ctxt.getPackageManager();
  ComponentName cn=new ComponentName(ctxt, getClass());

  try {
    ActivityInfo ai=pm.getReceiverInfo(cn,
                                       PackageManager.GET_META_DATA);
    XmlResourceParser xpp=ai.loadXmlMetaData(pm,
                                             WAKEFUL_META_DATA);

    while (xpp.getEventType()!=XmlPullParser.END_DOCUMENT) {
      if (xpp.getEventType()==XmlPullParser.START_TAG) {
        if (xpp.getName().equals("WakefulIntentService")) {
          String clsName=xpp.getAttributeValue(null, "listener");
          Class<AlarmListener> cls=(Class<AlarmListener>)Class.forName(clsName);

          return(cls.newInstance());
        }
      }

      xpp.next();
    }
  }
  catch (NameNotFoundException e) {
    Log.e(getClass().getName(), "Cannot find own info???", e);
  }
  catch (XmlPullParserException e) {
    Log.e(getClass().getName(), "Malformed metadata resource XML", e);
  }
  catch (IOException e) {
    Log.e(getClass().getName(), "Could not read resource XML", e);
  }
  catch (ClassNotFoundException e) {
    Log.e(getClass().getName(), "Listener class not found", e);
  }
  catch (IllegalAccessException e) {
    Log.e(getClass().getName(), "Listener is not public or lacks public
constructor", e);
  }
  catch (InstantiationException e) {
    Log.e(getClass().getName(), "Could not create instance of listener", e);
  }
```

```
  return(null);
}
```

To read data from a `<meta-data>` element, you use `PackageManager` to get information about your `BroadcastReceiver` via `getReceiverInfo()` and the `ActivityInfo` object it returns. You can ask the `ActivityInfo` object to get you the metadata for a given name via the `loadXmlMetaData()` method, which returns an `XmlResourceParser`. This is an implementation of the `XmlPullParser` interface, so you can start iterating over the parse events until you encounter your desired element. In this case, we scan for the start-tag event for the `WakefulIntentService` element, then read out the listener attribute, use `Class.forName()` to find the class, and call `newInstance()` on the `Class` object to create an instance of your `AlarmListener`.

An alternative approach would be to have you subclass `AlarmReceiver` and override `getListener()`, and you are certainly welcome to do that if the `<meta-data>` approach causes you difficulty.

So, the net of `AlarmReceiver` is that we allow you to schedule your alarms (if the device is freshly booted) or we allow you to do your wakeful work (if the alarm is from `AlarmManager`).

Command Processing

`AlarmReceiver`, when triggered by `AlarmManager`, detects that there is no supplied action and, in that case, passes control to `WakefulIntentService` via `sendWakefulWork()`. You can call this method yourself from other, non-alarm scenarios where you want work to be done that can be completed without the device falling asleep.

There are two flavors of `sendWakefulWork()`. Both take a `Context` – the difference is in the second parameter. One takes a `Class` object, identifying the `WakefulIntentService` subclass that should be invoked to do the wakeful work. This is a convenience method, wrapping around the other flavor of `sendWakefulWork()` that takes an `Intent`.

Either flavor of `sendWakefulWork()` on `WakefulIntentService` eventually routes to a `getLock()` method:

```
static final String NAME="com.commonsware.cwac.wakeful.WakefulIntentService";
static final String LAST_ALARM="lastAlarm";
private static volatile PowerManager.WakeLock lockStatic=null;

synchronized private static PowerManager.WakeLock getLock(Context context) {
  if (lockStatic==null) {
    PowerManager
mgr=(PowerManager)context.getSystemService(Context.POWER_SERVICE);

    lockStatic=mgr.newWakeLock(PowerManager.PARTIAL_WAKE_LOCK,
                               NAME);
    lockStatic.setReferenceCounted(true);
  }

  return(lockStatic);
}

public static void sendWakefulWork(Context ctxt, Intent i) {
  getLock(ctxt.getApplicationContext()).acquire();
  ctxt.startService(i);
}
```

The `getLock()` implementation lazy-creates our `WakeLock` by getting the `PowerManager`, creating a new partial `WakeLock`, and setting it to be reference counted (meaning if it is acquired several times, it takes a corresponding number of `release()` calls to truly release the lock). If we have already retrieved the `WakeLock` in a previous invocation, we reuse the same lock.

Back in `AlarmReceiver`, up until this point, the CPU was running because `AlarmManager` held a partial `WakeLock`. Now, the CPU is running because both `AlarmManager` *and* `WakefulIntentService` hold a partial `WakeLock`.

Then, `sendWakefulWork()` starts up our service and exits. Since this is the only thing `onReceive()` was doing in `AlarmReceiver`, `onReceive()` exits. Notably, `AlarmReceiver` does not release the `WakeLock` it acquired via `sendWakefulWork()`. This is important, as we need to ensure that the service can get its work done while the CPU is running. Had we released the `WakeLock` before returning, it is possible that the device would fall back asleep before our service had a chance to acquire a fresh `WakeLock`. This is one of the keys of using `WakeLock` successfully – as needed, use overlapping

`WakeLock` instances to ensure constant coverage as you pass from component to component.

Now, our service will start up and be able to do something, while the CPU is running due to our acquired `WakeLock`.

So, `WakefulIntentService` will now get control, under an active `WakeLock`. Since it is an `IntentService` subclass, `onHandleIntent()` is called. Here, we just route control to the subclass' implementation of an abstract `doWakefulWork()` method, ensuring that we release the `WakeLock` when the work is done, even if a `RuntimeException` is raised:

```
@Override
final protected void onHandleIntent(Intent intent) {
  try {
    doWakefulWork(intent);
  }
  finally {
    getLock(this.getApplicationContext()).release();
  }
}
```

As a result, each piece of work that gets sent to the `WakefulIntentService` will acquire a `WakeLock` via `sendWakefulWork()` and will release that `WakeLock` when `doWakefulWork()` ends. Once that `WakeLock` is fully released, the device can fall back asleep.

Background Data Setting

Users can check or uncheck a checkbox in the Settings application that indicates if they want applications to use the Internet in the background. Services employing `AlarmManager` should honor this setting.

To find out whether background data is allowed, use the `ConnectivityManager` system service and call `getBackgroundDataSetting()`. For example, your alarm-triggered `BroadcastReceiver` could check this before bothering to arrange for the `IntentService` (or `WakefulIntentService`) to do work.

You can also register a `BroadcastReceiver` to watch for the `ACTION_BACKGROUND_DATA_SETTING_CHANGED` broadcast, also defined on `ConnectivityManager`. For example, you could elect to completely cancel your alarm if the background data setting is flipped to `false`.

The "Everlasting Service" Anti-Pattern

One anti-pattern that is all too prevalent in Android is the "everlasting service". Such a service is started via `startService()` and never stops – the component starting it does not stop it and it does not stop itself via `stopSelf()`.

Why is this an anti-pattern?

- The service takes up memory all of the time. This is bad in its own right if the service is not continuously delivering sufficient value to be worth the memory.
- Users, fearing services that sap their device's CPU or RAM, may attack the service with so-called "task killer" applications or may terminate the service via the Settings app, thereby defeating your original goal.
- Android itself, due to user frustration with sloppy developers, will terminate services it deems ill-used, particularly ones that have run for quite some time.

Occasionally, an everlasting service is the right solution. Take a VOIP client, for example. A VOIP client usually needs to hold an open socket with the VOIP server to know about incoming calls. The only way to continuously watch for incoming calls is to continuously hold open the socket. The only component capable of doing that would be a service, so the service would have to continuously run.

However, in the case of a VOIP client, or a music player, the user is the one specifically requesting the service to run forever. By using `startForeground()`, a service can ensure it will not be stopped due to old age for cases like this.

As a counter-example, imagine an email client. The client wishes to check for new email messages periodically. The right solution for this is the `AlarmManager` pattern described earlier in this chapter. The anti-pattern would have a service running constantly, spending most of its time waiting for the polling period to elapse (e.g., via `Thread.sleep()`). There is no value to the user in taking up RAM to watch the clock tick. Such services should be rewritten to use `AlarmManager`.

Most of the time, though, it appears that services are simply leaked. That is one advantage of using `AlarmManager` and an `IntentService` – it is difficult to leak the service, causing it to run indefinitely. In fact, `IntentService` in general is a great implementation to use whenever you use the command pattern, as it ensures that the service will shut down eventually. If you use a regular service, be sure to shut it down when it is no longer actively delivering value to the user.