
CROSS-COMPONENT COMMUNICATION AND TASK SCHEDULING IN AN ANDROID OS APPLICATION

Jeremy Iamurri
March 30, 2012
Design Team 3

Abstract

This application note covers the basics of how to do cross-component communication within a single Android app, such as from a Service to an Activity or vice versa, and how to schedule tasks between these components. The note will first discuss the various aspects of using the Android Software Development Kit (SDK) for this particular subject, and then give a basic demonstration on how to set up and perform the necessary tasks. The overall goal of this application note is to provide a basic understanding of how to program an application in Eclipse with the built in functionality from both Java and the Android SDK. This application will combine both the cross-component communication and the task scheduling.

Keywords: Java, Service, Activity, Scheduling, Broadcast, Android, Android SDK

Background

Android's SDK provides quite a few options for handling the ability to communicate between multiple components, or even multiple processes or applications. The easiest to use, which will be covered in the example later on in this note, is Intent calls between the main Activity, which is the main program component of the application, and either another Activity. These Intents are objects that contain a command and, optionally, can contain a Bundle object of additional data. While this certainly does simplify the process, it has a security hole that could cause data to leak due to the way Android handles these Intent objects.

Specifically, when an Intent is sent out to the OS, it contains a filter string that another component, called a Broadcast Receiver, must watch for. Any currently running activity that contains a Listener with this particular filter will be able to process the data in the Intent, which could clearly be a problem. To prevent these security holes, Android 3.0 introduced a new object, called the Local Broadcast Manager. This object has been backported to earlier versions of Android as well, due to the obvious significance it has. However, for the purposes of this application note, the Local Broadcast Manager will not be used.

The other important aspect of this application note is task scheduling, which can be done in a multitude of ways. One particular method is actually a native Java object called a Timer. The Timer object, upon creation and scheduling, will automatically create a thread to run a set of tasks sequentially. This Timer can be set to either run at a specific date and time, wait a predetermined number of milliseconds before the thread executes, and optionally set a delay between multiple executions.

How to Combine Broadcast Intents with the Timer

Step 1 – Setting up the Activity

In this example of code snippets, an application Activity will bind and activate a Service component. This Service component, based on task scheduling done with a Timer object, will return an Intent back to the main Activity for further processing. Please note this example will assume that a developer has some experience with the Android SDK and is capable of setting up it within Eclipse or another IDE. However, URLs will be provided in the Appendix on any subjects referenced, but not discussed in great detail, in this example.

Within this Android application there will be two major components: the Activity, which most likely is actually an extended class, that drives the program and receives data, and the Service, also extended, that drives the task scheduling. Within the Activity, there will be multiple functions that have the flag `@Override` just before them. As one would expect, the `@Override` flag forces the Activity to run an alternate version of the following function instead of the default one provided by the base class. **Section 1, Figure 1** in the Appendix provides the basic flow chart of an Activity's life-cycle.

The four most important functions in the Activity related to this activity are `onCreate`, `onResume`, `onPause`, and `onDestroy`. The `onCreate` and `onDestroy` functions are what bind and unbind, respectively, the Service from the Activity. See the code in **Section 1, Figure 2** of the Appendix for how binding and unbinding looks. The `doBindService` and `doUnbindService` functions must be called in `onCreate` and `onDestroy`, respectively, as **Section 1, Figure 3** will show. Of course, in order to keep the binding connection alive, the Service will not be unbound from the Activity until the Activity calls the `onDestroy` function.

The `onResume` and `onPause` functions handle starting and stopping another important sub-component of the Activity, the Broadcast Receiver. To start off, the receiver class must be created and prepared for use. **Section 1, Figure 4** gives an example. Note that in this particular example, the Receiver was used as a subclass of the Activity. In the code for this extended Broadcast Receiver class, one can see that the Intent object contains extra data in a Bundle. As previously mentioned, Intents can contain data to be used in other areas. Of course, the Receiver must be activated or deactivated in order to function. This is done by registering, or deregistering, the Receiver object, as seen in **Section 1, Figure 5**. The previously mentioned filter for Intents can be seen as well in **Figure 5**. This string is what the Receiver will be looking for whenever a broadcast is sent by an Intent.

Step 2 – Preparing the Service

Much like the Activity, it is ideal to extend the Android Service class in order to provide any additional functionality desired. There is one major sub-component used in this service, the Timer. As a clarification, the primary reason for using a Service instead of an Activity is to allow for multi-process code, though in the complete source code of this example, every function runs in a single process. The Timer, however, is multi-threaded in its application.

As can be seen in **Section 1, Figure 6**, a Timer object must be declared and then scheduled. Elsewhere in the service, most likely also in the onCreate function, it is necessary to set up a list or vector of dates, along with the data to be processed. These functions were omitted for the sake of readability. After declaring the Timer object, the schedule function is used to, as expected, schedule tasks. It is recommended to extend the Timer Task class, also seen in **Figure 6**, to allow for additional data, and pass this, along with the scheduled Date object, or the desired wait time in milliseconds.

Upon reaching the desired time, the run function in the Timer Task object will be called, and, in this example's case, creates an Intent with the same Intent filter as the Receiver in the Activity, and then broadcasts it. The Receiver will detect the broadcast and appropriately handle the embedded data, as discussed in Step 1. After completing the broadcast, this particular example sends a notification to the user (see Section 3 of the Appendix for a link to how this is done), and then terminates the Timer Task's thread with the cancel function call.

Conclusion



In this application note, a method for handling scheduled cross-component communication was discussed. With this code, and an understanding of the basics of an Android application, a developer should be able to schedule tasks with ease, such as for a social network application. There are, of course, other possible ways of handling the Service side, including using Intent Services and Bound Services. Links to those two specialized types of services are included in Section 3 of the Appendix. Regardless, the code is quite extendable due to the flexibility of both Java and the Android SDK, and should prove to be a useful guide.

Appendix

Section 1: Figures and Diagrams

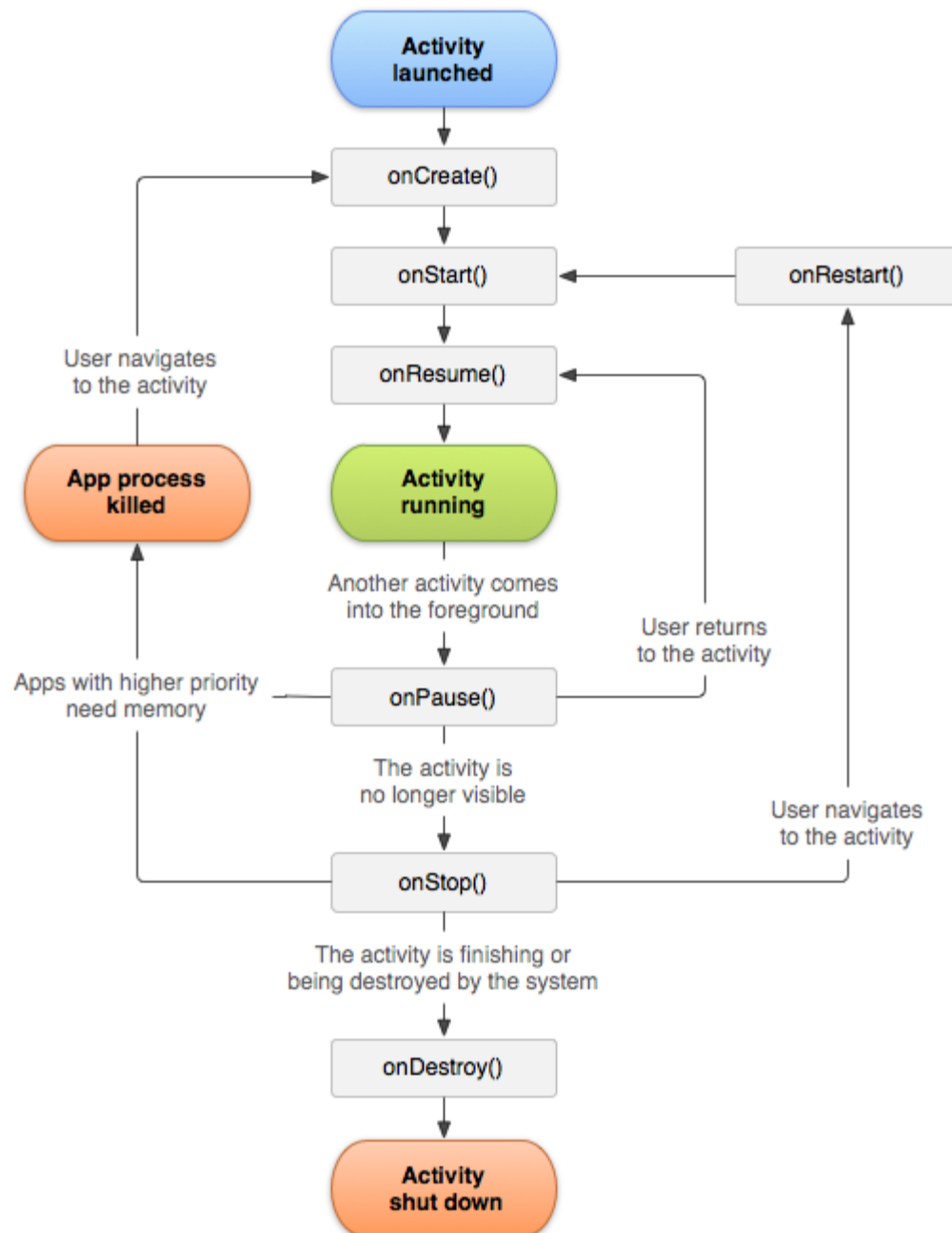


Figure 1: Android Activity Life-cycle

Appendix

```
private CameraPositioningService mService;
private ServiceConnection mConnection = new ServiceConnection() {
    public void onServiceConnected(ComponentName className, IBinder service) {
        // Get the service object
        mService = ((CameraPositioningService.LocalBinder)service).getService();

        // do any extra setup that requires your Service
    }

    public void onServiceDisconnected(ComponentName className) {
        // As our service is in the same process, this should never be called
        mService = null;
    }
};

private boolean mIsBound = false;

void doBindService() {
    // Establish a connection with the service. We use an explicit
    // class name because we want a specific service implementation that
    // we know will be running in our own process (and thus won't be
    // supporting component replacement by other applications).
    if (!mIsBound)
    {
        bindService(new Intent(CameraPositioning.this,
            CameraPositioningService.class), mConnection, Context.BIND_AUTO_CREATE);
        mIsBound = true;
    }
}

void doUnbindService() {
    if (mIsBound) {
        // Detach our existing connection.
        unbindService(mConnection);
        mIsBound = false;
    }
}
```

Figure 2: Setting up a service connection

Appendix

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    doBindService();

    // Do the rest of onCreate
}

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

    doUnbindService();

    // Do the rest of onDestroy if needed
}
```

Figure 3: Calling binding/unbinding

```
public class IncomingMessage extends BroadcastReceiver {

    @Override
    public void onReceive(Context context, Intent intent) {
        // TODO Auto-generated method stub
        String action = intent.getAction();

        if (action.equalsIgnoreCase("edu.msu.camerapositioning.UPDATELOCATION"))
        {
            Bundle extra = intent.getExtras();

            double[] vals = extra.getDoubleArray("edu.msu.camerapositioning.coords");

            setTarget(vals[0], vals[1], vals[2]);
        }
    }
}

private IncomingMessage mReceiver = new IncomingMessage();
```

Figure 4: BroadcastReceiver extended class example

Appendix

```
@Override
protected void onResume() {
    super.onResume();

    registerReceiver(mReceiver, new IntentFilter("edu.msu.camerapositioning.UPDATELOCATION"));
}

@Override
protected void onPause() {
    super.onPause();

    unregisterReceiver(mReceiver);
}
```

Figure 5: Registering/deregistering BroadcastReceiver

Appendix

```
private Timer mTimer = new Timer();

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

    // The schedule function takes in a TimerTask object, or an extended class variant
    // It also can either take in a Date object to run at, or an integer of milliseconds
    // to wait
    mTimer.schedule(new CameraTask(data),data.getDateToRun());
}

class CameraTask extends TimerTask
{
    // Private data for the extended class
    // This data is used for the eventual run call
    private LocationData mTaskData;

    public CameraTask(LocationData data)
    {
        mTaskData = data;
    }

    // This function will be called when the scheduled time has been met
    @Override
    public void run() {
        // Note the intent that's the same as the filter from the Activity
        Intent i = new Intent("edu.msu.camerapositioning.UPDATELOCATION");
        i = i.putExtra("edu.msu.camerapositioning.coords", mTaskData.getCoords());
        sendBroadcast(i);

        SimpleDateFormat df = new SimpleDateFormat("yyyy-MM-dd H:mm:ss");

        if (mTaskData.getNextDate() != null)
            showNotification(true,"Next run: " + df.format(mTaskData.getNextDate()));
        else
            showNotification(true,"Next run: None");

        // The cancel function only necessary if doing a single run
        // It terminates the current Timer thread
        mTimer.cancel();
    }
}
```

Figure 6: Necessary functions and setup for Service

Appendix

Section 2: References

Android Activity: <http://developer.android.com/reference/android/app/Activity.html>

Android Service: <http://developer.android.com/reference/android/app/Service.html>

Android Broadcast Receiver:
<http://developer.android.com/reference/android/content/BroadcastReceiver.html>

Java Timer: <http://docs.oracle.com/javase/1.4.2/docs/api/java/util/Timer.html>

Section 3: Additional Reading

Android Local Broadcast Manager:
<http://developer.android.com/reference/android/support/v4/content/LocalBroadcastManager.html>

Installing the Android SDK:
<http://developer.android.com/sdk/installing.html>

ADT Plugin for Eclipse (Particularly the Installing the ADT Plugin section):
<http://developer.android.com/sdk/eclipse-adt.html>

Status Bar Notifications:
<http://developer.android.com/guide/topics/ui/notifiers/notifications.html>

Bound Services: <http://developer.android.com/guide/topics/fundamentals/bound-services.html>

Intent Service: <http://developer.android.com/reference/android/app/IntentService.html>