Android Architecture Tutorial: Developing an App with a Background Service (using IPC)

Jun 1st, 2010 at 1:50 pm

Intro

Android is a wonderful platform. It has a rich API that allows you, a developer, to easily implement things that other platforms do not support at all or they need you to bend over backwards to get what you want.

The ability to develop a service that runs in the background and does something when the user is not actively working with your app is a great option for application authors. Examples of usage include mail or IM apps that send you a notification whenever there is a new message (we'll discuss notifications and how to program them correctly in another post), background music players, background downloads and so on.

Yes, there is a dark side to Android background services too – too many services doing too much stuff in the background will slow down the phone and suck the battery. However it's not quite what I wanted to bring up in this post, so maybe we'll come back to resource management *etiquette* in yet another post.

Problem Definition

What we will develop here is a dummy application that contains the following components:

1. A **background service** doing something important but not immediately visible to the user
2. **One or more activities** that, once launched by the user, **talk** to the service, **control** the service and **present some information that the service provides**

The architecture also has the following characteristics:

- The service will run in a **separate process**, allowing the platform to manage its lifecycle and resources separately from the activities
- The service is supposed to **run all the time** (there are other valid usage patterns for services, but we won't discuss that here)
- The UI (activities) will get updates from the service in a **callback** or **passive mode**, reacting to service events **rather than polling the service for updates**. This is very important in terms of effective architecture, so please keep it in mind for now
- Since the service will run in a different process, we need to use **IPC** to talk to it from activities. IPC stands for **interprocess communication** and is not an Android-specific term but Android has its own implementation (as we will see)
- The service will do all the thinking (**business logic**) while the UI will be designed to be **as thin and dumb as possible**. We'll keep the architecture well-layered
- The service will be configured to **start during system boot-up**

In terms of functionality, our service will grab the latest **#android** tweets using the twitter search API. The activity will show the latest tweets grabbed by the service when you launch them, and update automatically. Yes, it is a pretty artificial use case, but later you can add notifications and turn the app into "Android News Notifier", and then put it on the market. 😃

I will not paste all the boilerplate code here, only the most important pieces. The complete project source is totally available at the bottom of the post.

Step 1: Developing the Background Service

First we want to understand what our background service will do and what API operations it will have for its clients (activities). Basically, we want the background service to connect to twitter and download the latest tweets labeled with #android every minute (60 seconds). We can use the straightforward Android facilities such as Timer and HttpClient to do what we want.

The first classes I create are **Tweet** and **TweetSearchResult**. They will represent the results of twitter search. They will be our domain model classes, created and kept by the service and presented by the activities.

In order to be able to pass instances of these classes via IPC (between the service and the activities), we have to make them Parcelable. That's basically the same as **Serializable** on Java SE but parceling is a more lightweight facility that Android architects preferred for its speed. On the other hand, parceling requires some manual work from you. (There is a way to still use Serializable if you want to but you are officially discouraged to do so.) I won't discuss all the details of parceling right here, so please read the Parcelable doc and make sure you understand it well before we continue – or ask your question in comments. 😃

Now that our domain model is ready (and it is generally a good idea to start with domain model classes), we can move on and build a stub for our service. First, we create a class that extends the Android Service class:

?

```java
public class TweetCollectorService extends Service {
  @Override
  public IBinder onBind(Intent intent) {
    // TODO Auto-generated method stub
    return null;
  }
}
```

This is what you get in Eclipse if you make a new class that extends **Service**. **onBind()** is the method that handles incoming IPC connections. We will implement it later. For now, we want to develop the basic lifecycle methods: **onCreate()** and **onDestroy()**:

?

```java
public class TweetCollectorService extends Service {

  private static final String TAG = TweetCollectorService.class.getSimpleName();

  private Timer timer;

  private TimerTask updateTask = new TimerTask() {
    @Override
    public void run() {
      Log.i(TAG, "Timer task doing work");
    }
  };

  @Override
  public IBinder onBind(Intent intent) {
    // TODO Auto-generated method stub
    return null;
  }

  @Override
  public void onCreate() {
    super.onCreate();
    Log.i(TAG, "Service creating");

    timer = new Timer("TweetCollectorTimer");
    timer.schedule(updateTask, 1000L, 60 * 1000L);
  }

  @Override
  public void onDestroy() {
    super.onDestroy();
    Log.i(TAG, "Service destroying");

    timer.cancel();
    timer = null;
  }
}
```

As you can easily figure out, the idea is to create and start our timer-based work in the **onCreate()** method and stop and clean up in the **onDestroy()** method. We are almost ready to test our background service, but first we need to register it in our **AndroidManifest.xml** file:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
  xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.mindtherobot.samples.tweetservice"
  android:versionCode="1"
  android:versionName="1.0">
  <application
    android:icon="@drawable/icon"
    android:label="@string/app_name">
    <activity
      android:name=".TweetViewActivity"
      android:label="@string/app_name">
      <intent-filter>
        <action
          android:name="android.intent.action.MAIN" />
        <category
          android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>
    <!-- Here's what we add -->
    <service
      android:name=".TweetCollectorService"
      android:process=":remote">
      <intent-filter>
        <action
          android:name="com.mindtherobot.samples.tweetservice.TweetCollectorService" />
      </intent-filter>
    </service>
  </application>
  <uses-sdk
    android:minSdkVersion="3" />
  <uses-permission
    android:name="android.permission.INTERNET" />
</manifest>
```

(As you can see, when creating the project, I created a dummy activity that I called **TweetViewActivity**. We will use it later on.)

The **android:process=":remote"** attribute tells Android to run our service in a separate process from the rest of the application. Why do we want to do that? We want the service to have a separate lifecycle and a separate resource pool from the rest of application components (activities etc.), since it is going to run all the time while other components will only exist for short periods of time when the user will be actively using them. This decision has the consequence that we're now required to use IPC (read this part of Android docs if you're curious about other choices you could have in different situations).

At this point though, there is still no way for us to test the service. The reason is that services and other application components are never activated without a reason (called an **intent**) on Android. At the end of this article, we will add a **BroadcastReceiver** that will start the service at system boot-up, but for now let's make our activity also start the service whenever the activity is created. (It's a good idea to leave it that way in the production code too for situations when the service did not start at boot-up for some reason, or it died afterwards.)

The following part…

```
<intent-filter>
  <action
    android:name="com.mindtherobot.samples.tweetservice.TweetCollectorService" />
</intent-filter>
```

…defines an **action name** which is just like an ID that we will give to Android when we want to start our service. Without an action name, we can't build an intent to start our service from other components. Using a fully-qualified class name for the action name is a good idea.

Now, let's modify the activity code to start our service whenever the activity is created. It's OK if this will be called when the service is already running – the request will be ignored by Android (services are not instantiated more than once):

```
public class TweetViewActivity extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        /* This is the important part */
        startService(new Intent(TweetCollectorService.class.getName()));
    }
}
```

As you can see, we create an Intent that takes the fully-qualified service class name as the parameter. This will match the **intent-filter** in **AndroidManifest.xml** that we just added, and launch the service if it's not yet running. (Note: **startService()** is a method of the Context mega-class. Thus the **startService()** and other service-related methods are available whenever a **Context** is available – we will also use them in our **BroadcastReceiver** later.)

Now we're ready to give our service a test! Just launch the application in Eclipse. As soon as the activity shows up on the screen, you should start seeing log messages from our service in **LogCat**:

```
06-01 09:41:25.733: INFO/TweetCollectorService(21716): Service creating
06-01 09:41:26.764: INFO/TweetCollectorService(21716): Timer task doing work
06-01 09:42:26.764: INFO/TweetCollectorService(21716): Timer task doing work
```

Beautiful! The service starts and runs fine. Now close the activity (using the “bent arrow” button) and watch how the service is still running even though the activity is dead now. You can even start doing something else, like browsing the web, but the service will still be running:

```
06-01 09:44:26.764: INFO/TweetCollectorService(21716): Timer task doing work
```

We just did a great thing – made a mobile application that runs in the background. Isn’t Android cool?

At this point I will add some “meat” to the service. I won’t cover in detail how it is done (you can check the full sources at the bottom of the article), but the timer task will now actually retrieve data from twitter and store the latest search result in a field of the service class that is called **latestSearchResult**:

```java
private final Object latestSearchResultLock = new Object();

private TweetSearchResult latestSearchResult = new TweetSearchResult();
```

We need to wrap all operations with **latestSearchResult** in..

```java
synchronized (latestSearchResultLock) {
    /* operation here */
}
```

…because one thread might be trying to update the result while another one might be serving it for an IPC client. In a word, we need to synchronize here.

Finally, just a short description how to make the service start at boot-up. First, the **BootReceiver** class:

```java
public class BootReceiver extends BroadcastReceiver {
  @Override
  public void onReceive(Context context, Intent intent) {
        Intent serviceIntent = new Intent(TweetCollectorService.class.getName());
        context.startService(serviceIntent);
  }
}
```

Second, changes in **AndroidManifest.xml** (within the **application** element):

```xml
<receiver
  android:name=".BootReceiver">
  <intent-filter>
    <action
      android:name="android.intent.action.BOOT_COMPLETED">
    </action>
  </intent-filter>
</receiver>
```

And a permission in the same file:

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

Now it’s time to expose our service for IPC clients!

Step 2: Exposing the Service for IPC

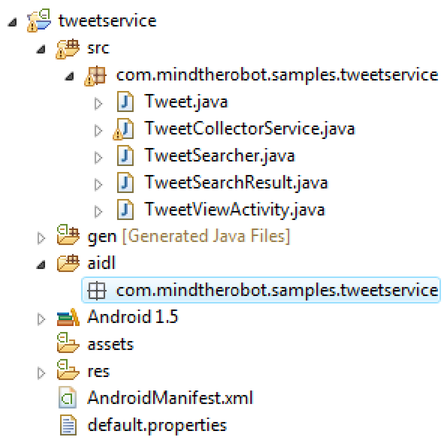
As I mentioned above, **IPC** stands for **Interprocess Communication**. In fact, this term is used on various platforms, and even on a single platform you can communicate multiple processes in various ways.

On Android, the term IPC is used for a specific, platform-supported facility that allows you to connect various components that might be parts of different application, or of the same application but running in separate processes. There are fundamental rules of using IPC on Android:

- Interfaces that you want to expose for IPC should be described in special files using the **AIDL language** (which is similar to Java and almost straightforward – don’t worry)
- If you want to pass non-primitive objects (such as **Tweet** instances), you have to make them **Parcelable** and **declare them in AIDL as well**
- IPC connecting and data transfer **can be considered fast** (unlike, for example, network connections). (This means that in most cases you can do IPC requests from the UI thread without putting each one into an AsyncTask.)
- IPC connections are initiated via **Intents** just like all other component interaction on Android
- You can never assume anything about an IPC connection – it might get lost at any time, it might be unsuccessful etc. – **you have to handle that gracefully**

Let’s start with creating the AIDL definitions for our service API. First of all, we need to create **a new source folder** in Eclipse called **aidl**. We can’t just put AIDL files into the **src** folder along with Java files due to a bug in the Android tools for Eclipse (you can read more about it here).

The directory structure will look like this:



```
tweetservice
  src
    com.mindtherobot.samples.tweetservice
      Tweet.java
      TweetCollectorService.java
      TweetSearcher.java
      TweetSearchResult.java
      TweetViewActivity.java
  gen [Generated Java Files]
  aidl
    com.mindtherobot.samples.tweetservice
  Android 1.5
  assets
  res
  AndroidManifest.xml
  default.properties
```

aidl source folder

As you can see, I created the same package in **aidl** as I use in **src**. AIDL files are automatically converted to Java by ADT and I want the resulting Java classes to reside in the same package.

Now, there are three AIDL files I put into that package:

TweetSearchResult.aidl

```
package com.mindtherobot.samples.tweetservice;

parcelable TweetSearchResult;
```

Basically, we just need to introduce our **TweetSearchResult** type to the AIDL compiler (which works undercover) since we are going to pass instances of that class in our IPC operations.

Next, the listener: **TweetCollectorListener.aidl**. We will allow the IPC clients to be notified about tweet updates rather than have them poll the service. This is a much cooler and architecturally beneficial solution than polling:

```
package com.mindtherobot.samples.tweetservice;

interface TweetCollectorListener {

  void handleTweetsUpdated();
}
```

And, finally, the API endpoint: **TweetCollectorApi.aidl**

```
package com.mindtherobot.samples.tweetservice;

import com.mindtherobot.samples.tweetservice.TweetSearchResult;
import com.mindtherobot.samples.tweetservice.TweetCollectorListener;

interface TweetCollectorApi {

  TweetSearchResult getLatestSearchResult();

  void addListener(TweetCollectorListener listener);

  void removeListener(TweetCollectorListener listener);
}
```

As you can see, in AIDL you have to import names even from the same package. Not a big deal though.

Note: There is a bit more to AIDL syntax than I described here. Be sure to read the corresponding Android doc page before you try to develop something more complex than this example.

Now, if everything is fine and you have no errors in your project, you should be able to peek into the **gen** folder and see the auto-generated Java classes that correspond to your AIDL files. You should not edit that auto-generated stuff, but we will use those classes in our code as superclasses, like they are supposed to be used.

Those auto-generated classes might also add some warnings to your project. I don't think there's much you can do even if you hate warnings as much as I do. So let's just sigh and live on.

OK, now it's time to use our AIDL-generated code and finally expose some API! Here's the first change to the **TweetCollectorService** – adding some fields:

```
private List<TweetCollectorListener> listeners = new ArrayList<TweetCollectorListener>();

private TweetCollectorApi.Stub apiEndpoint = new TweetCollectorApi.Stub() {

  @Override
  public TweetSearchResult getLatestSearchResult() throws RemoteException {
    synchronized (latestSearchResultLock) {
      return latestSearchResult;
    }
  }

  @Override
  public void addListener(TweetCollectorListener listener)
      throws RemoteException {

    synchronized (listeners) {
      listeners.add(listener);
    }
  }

  @Override
  public void removeListener(TweetCollectorListener listener)
      throws RemoteException {

    synchronized (listeners) {
      listeners.remove(listener);
    }
  }

};
```

The **listeners** field is the container for API listeners that we will notify on tweet update.

The **apiEndpoint** field below is actually the code that will be called when IPC clients will invoke the external interface. It's pretty straightforward what we do there, just remember to sync everything since IPC requests are processed from separate threads. Also, notice we're extending **TweetCollectorApi.Stub**. That's an AIDL convention – the abstract base **Stub** class is generated for you.

Now, we should not forget to notify the listeners whenever tweets are updated, so we add the following to our **updateTask** code:

```
synchronized (listeners) {
  for (TweetCollectorListener listener : listeners) {
    try {
      listener.handleTweetsUpdated();
    } catch (RemoteException e) {
      Log.w(TAG, "Failed to notify listener " + listener, e);
    }
  }
}
```

We need to catch **RemoteException** whenever we invoke remote methods. In fact, here we have two-way IPC – the clients can call the service (**getLatestSearchResult()**), but the service calls its clients too, via the listeners! This is a powerful pattern that is not always considered when designing APIs.

Now, we need to implement the **onBind()** method so that it actually accepts IPC connections:

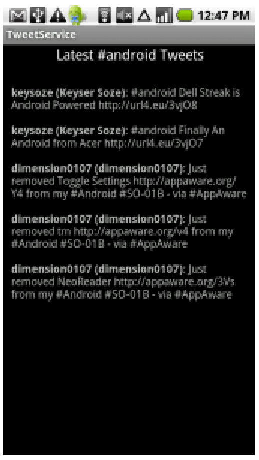
```java
@Override
public IBinder onBind(Intent intent) {
  if (TweetCollectorService.class.getName().equals(intent.getAction())) {
    Log.d(TAG, "Bound by intent " + intent);
    return apiEndpoint;
  } else {
    return null;
  }
}
```

Note that we get the client's intent here. This can be used to expose multiple APIs from the same service.

Now the service is ready to accept IPC clients. Let's implement the activity.

Step 3: Turning the Activity into an IPC Client

This is what our **TweetViewActivity** will finally look like (click to enlarge):


12:47 PM
TweetService
Latest #android Tweets

keysoze (Keyser Soze): #android Dell Streak is Android Powered http://url4.eu/3vjO8

keysoze (Keyser Soze): #android Finally An Android from Acer http://url4.eu/3vjO7

dimension0107 (dimension0107): Just removed Toggle Settings http://appaware.org/Y4 from my #Android #SO-01B - via #AppAware

dimension0107 (dimension0107): Just removed tm http://appaware.org/v4 from my #Android #SO-01B - via #AppAware

dimension0107 (dimension0107): Just removed NeoReader http://appaware.org/3Vs from my #Android #SO-01B - via #AppAware

Note the tweets are retrieved by the service

So what we want to do is:

1. Create an IPC connection to the service when the activity is created (and be sure to close it after the activity is destroyed)
2. After the connection is open, show the tweets in the UI
3. Register the activity as a listener to the service so that the service will ping it whenever tweets are updated; when it does, update the UI

Activities have an interesting lifecycle that's worth studying, but here we will use only the most important lifecycle hooks: **onCreate()** and **onDestroy()**. Note that they are similar to their counterparts in services, but activities have a different lifecycle. For example, **onCreate** and **onDestroy()** will typically be called when you change the device orientation from landscape to portrait and vice-versa.

Anyway, here is the implementation of those methods in our **TweetViewActivity**:

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

      handler = new Handler(); // handler will be bound to the current thread (UI)

      tweetView = (TextView) findViewById(R.id.tweet_view);

      Intent intent = new Intent(TweetCollectorService.class.getName());

      // start the service explicitly.
      // otherwise it will only run while the IPC connection is up.
      startService(intent);

      bindService(intent, serviceConnection, 0);

      Log.i(TAG, "Activity created");
  }

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

  try {
    api.removeListener(collectorListener);
    unbindService(serviceConnection);
  } catch (Throwable t) {
    // catch any issues, typical for destroy routines
    // even if we failed to destroy something, we need to continue destroying
    Log.w(TAG, "Failed to unbind from the service", t);
  }

  Log.i(TAG, "Activity destroyed");
}
```

And here are the fields mentioned in the methods above:

```
private ServiceConnection serviceConnection = new ServiceConnection() {
  @Override
  public void onServiceConnected(ComponentName name, IBinder service) {
    Log.i(TAG, "Service connection established");

    // that's how we get the client side of the IPC connection
    api = TweetCollectorApi.Stub.asInterface(service);
    try {
      api.addListener(collectorListener);
    } catch (RemoteException e) {
      Log.e(TAG, "Failed to add listener", e);
    }

    updateTweetView();
  }

  @Override
  public void onServiceDisconnected(ComponentName name) {
    Log.i(TAG, "Service connection closed");
  }
};

private TweetCollectorApi api;

private TextView tweetView;

private Handler handler;

private TweetCollectorListener.Stub collectorListener = new TweetCollectorListener.Stub() {
  @Override
  public void handleTweetsUpdated() throws RemoteException {
    updateTweetView();
  }
};
```

So what happens here?

The following line creates the IPC connection to the service:

```
bindService(intent, serviceConnection, 0);
```

serviceConnection serves as a callback that tells when the connection is open. Only after its **onServiceConnected()** method has been invoked you can start working with the IPC connection (which is represented by the **api** field – note how we use the AIDL-generated interface here).

We need to call **unbindService()** in the **onDestroy()** method to explicitly drop the IPC connection.

There are multiple places where we actually use the API interface: when registering and unregistering the listener (**onServiceConnected()** and **onDestroy()**) and here in the **updateTweetView()** method:

```
TweetSearchResult result = api.getLatestSearchResult();
```

That is probably the apogee of this tutorial – being able to retrieve data from a service that runs in a separate process.

OK now. I'm not going to cover the UI details so just read the code to understand how the UI works if it's not clear.

Conclusion

We built a pretty nice structure for a background service based app with activities connected via IPC.
As discussed at the beginning of the article, you can use it for various apps and do cool things now.

Do not let the seemingly large amount of effort to build the IPC architecture overwhelm you. Neither be scared of not understanding how it works. IPC is the most effective, scalable and correct way to connect Android components to each other, and it's actually quite easy to use too, once you've done that a single time.

Be sure to ask your questions in the comments below. Don't be afraid to point me to any mistakes you find either 🙂

Attachment: the complete project source

Note: *This is the first Mind The Robot post. I hope it was useful enough. Any feedback is welcome, and stay tuned for new articles, tutorials and other good Android stuff. You can learn more about MTR here.*

Tags: activity, android, apps, architecture, development, ipc, service, tutorial

In General. You can leave a response, or trackback from your site.