Audience

This article is for Android programmers who are already familiar with the use of Activities, Intents and Services
(http://developer.android.com/reference/android/app/Service.html).

Introduction

This article presents a few methods of communication between a Service and an Activity.

While running, a program often needs the result of a side calculation without halting its other processes. One method of providing this information is to create a service which performs the side calculation, and returns the result to the main program. Following are different examples of methods of communication between services and activities.

Broadcasts and BroadcastReceivers

The simplest solution uses Broadcasts and BroadcastReceivers. However, this method has the following disadvantages:

The message sender does not know who will receive the data (it broadcasts the message).

Broadcasts consume a lot of bandwidth and are not suited for high traffic.

Each broadcast is actually an Intent with additional (optional) data. Each of these Intents is created for a specific action. This action allows the intended recipient to distinguish the received broadcasts from other data, and listen for only certain broadcasts. To limit the access of the broadcast to only the desired application, permissions need to be set in the sender sendBroadcast(Intent intent, String permission);. The receiver needs to use the same permission in order to be able to successfully intercept the intent.

First, we create a sample permission in the manifest file:

```
<permission
        android:name="com.sprc.sample.permission.ALLOW"
        android:label="@string/permission"
        android:protectionLevel="normal" >
    </permission>
```

Code Sample 1

Then, we need to use it.

```
<uses-permission android:name="com.sprc.sample.permission.ALLOW" />
```

Code Sample 2

The actual sending of the broadcast with the given permission will be shown a bit later.

The Action is a string consisting of a package name and some human-readable text. The Service class defines four of such actions used in this example:

```
public static final String ACTION_START = "com.sprc.sample.broadcast.action.START";
public static final String ACTION_STOP =
"com.sprc.sample.broadcast.action.STOP";

public static final String ACTION_TO_SERVICE =
"com.sprc.sample.broadcast.action.TO_SERVICE";

public static final String ACTION_FROM_SERVICE =
"com.sprc.sample.broadcast.action.FROM_SERVICE";
```

Code Sample 3

To set up a Service and an Activity, we use a BroadcastReceiver implementing an onReceive(Context, Intent) method. This method will process the data received.

Broadcast Activity

This activity receives the key “data” from the Intent, and prints its associated value to the TextView:

```
/**

 * Receiver for messages sent from Service to Activity.

 */

private BroadcastReceiver mReceiver = new BroadcastReceiver() {

             @Override

             public void onReceive(Context context, Intent intent) {

                          final TextView responseFromService = (TextView)
findViewById(R.id.responseFromService);

                          responseFromService.setText(intent.getCharSequenceExtra("data"));

             }

};
```

Code Sample 4

To register a receiver as an onCreate() method to listen for specific broadcasts, this example creates a filter for intents with the action ACTION_FROM_SERVICE:

```
public void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

…

// Message handling - from service:

final IntentFilter myFilter = new

IntentFilter(BroadcastService.ACTION_FROM_SERVICE);

registerReceiver(mReceiver, myFilter);

}
```

Code Sample 5

The receiver must be unregistered in onDestroy() method:

```
public void onDestroy() {

             stopService(new Intent(BroadcastService.ACTION_STOP));

             unregisterReceiver(mReceiver);

             super.onDestroy();

}
```

Code Sample 6

The Context.sendBroadcast() method can be used to send data from an Activity to a Service:

```
void sendToService(CharSequence text) {

             Log.d("BroadcastActivity", "Sending message to service: " + text);

             final Intent intent = new Intent(BroadcastService.ACTION_TO_SERVICE);

             intent.putExtra("data", text);

             sendBroadcast(intent);

}
}
```

Code Sample 7

Broadcast Service

In the Service, the receiver broadcasts a response consisting of received text and a message:

```
/**

 * Receiver for messages sent from Activity to Service.

 */

private BroadcastReceiver mReceiver = new BroadcastReceiver() {

             @Override

             public void onReceive(Context context, Intent intent) {

                          Log.d("BroadcastService", "Service received: " +

intent.getCharSequenceExtra("data"));

                          sendToActivity("Who's there? You wrote: " +

intent.getCharSequenceExtra("data"));

             }

};
```

Code Sample 8

This receiver must also be registered in the onCreate() method. This receiver will listen for a different action than that of the Activity's receiver:

```
public void onCreate() {

             super.onCreate();

             final IntentFilter myFilter = new IntentFilter(ACTION_TO_SERVICE);

             registerReceiver(mReceiver, myFilter);

}
```

Code Sample 9

… and unregistered in onDestroy():

```
public void onDestroy() {

             unregisterReceiver(mReceiver);
```

```
            super.onDestroy();

}
```

The method sendToActivity() is very similar to Activity's method sendToService(), only we send this intent with our previously created permission (the earlier intent was sent globally, this one is sent only to recipients using our permission):

```
void sendToActivity(CharSequence text) {

            Log.d("BroadcastService", "Sending message to activity: " + text);

            final Intent intent = new Intent(BroadcastService.ACTION_FROM_SERVICE);

            intent.putExtra("data", text);

            sendBroadcast(intent, com.sprc.sample.broadcast.Manifest.permission.ALLOW);

}
```

Code Sample 10

The source code for this example is in attachment.

Binder

For Activities and Services used in one application the "Binder" solution is recommended. After binding with the service, the Binder will allow the activity (or other class bound to the service) to call any public method of the service.

Binder Service

First the service must define a class extending the Binder class and create an instance of it. In the following example, it is called LocalBinder. Reference to this instance will be exchanged with an activity. The LocalBinder may return the instance of enclosing Service class or contain public methods that Activity could call.

In our example the LocalBinder will just return the reference to BinderService instance:

```
public class BinderService extends Service {

…

…

/**

 * Local Binder returned when binding with Activity.
 * This binder will return the enclosing BinderService instance.

 */

public class LocalBinder extends Binder {

             /**

              * Return enclosing BinderService instance

              */

             BinderService getService() {

                          return BinderService.this;

             }

}

private final IBinder mBinder = new LocalBinder();

…

}
```

Code Sample 11

When client (activity) will bind to the service we will return the instance of LocalBinder:

```
@Override

public IBinder onBind(Intent intent) {

             Log.d("BinderService", "Binding...");

             return mBinder;

}



The service will also define a public method which could be called by client. In our example
the method is a simple String concatenation:
```

```
/**

 * Public method which can be called from bound clients.

 *

 * This method will return a String as a response to passed query

 */

public String getResponse(CharSequence query) {

             return "Who's there? You wrote: " + query;

}
```

Code Sample 12

Binder Activity

Activity will hold a reference to the service:

```
public class BinderActivity extends Activity {

…

/**

 * Reference to our bound service.

 */

BinderService mService = null;

boolean mServiceConnected = false;

…

}
```

Code Sample 13

The field mService is initialized on successful connection to the service. This is done by an instance of class ServiceConnection, an object for managing the connection:

```
/**
```

```
 * Class for interacting with the main interface of the service.

 */

private ServiceConnection mConn = new ServiceConnection() {

             @Override

             public void onServiceConnected(ComponentName className, IBinder binder) {

                          Log.d("BinderActivity", "Connected to service.");

                          mService = ((BinderService.LocalBinder) binder).getService();

                          mServiceConnected = true;

             }



             /**

              * Connection dropped.

              */

             @Override

             public void onServiceDisconnected(ComponentName className) {

                          Log.d("BinderActivity", "Disconnected from service.");

                          mService = null;

                          mServiceConnected = false;

             }

};
```

Code Sample 14

In ServiceConnection we override the method onServiceConnected(). This method is called after successful connection to the service. The onServiceConnected() calls LocalBinder.getService() to obtain the reference to the service. The Local Binder Service was described in the previous section “Binder Service”.

For an Activity to communicate with a service, the activity must first bind to the service, for example in the onStart() method:

```
@Override

public void onStart() {

             super.onStart();

             bindService(new Intent(this, BinderService.class), mConn,
Context.BIND_AUTO_CREATE);

             startService(new Intent(this, BinderService.class));

}
```

Code Sample 15

It must also unbind in the onStop() method:

```
@Override

protected void onStop() {

             super.onStop();

             if (mServiceConnected) {

                          unbindService(mConn);

                          stopService(new Intent(this, BinderService.class));

                          mServiceConnected = false;

             }

}
```

After successfully binding to the service, the activity obtains a valid reference to the BinderService instance. Communication calls a method from the service, for example:

```
final EditText textToService = (EditText) findViewById(R.id.textToService);
final Button sendTextToService = (Button) findViewById(R.id.sendTextToService);

final TextView responseFromService = (TextView)

findViewById(R.id.responseFromService);

sendTextToService.setOnClickListener(new View.OnClickListener() {

             @Override

             public void onClick(View v) {

                          if (mService != null) {

                                       String response = mService.getResponse((CharSequence)

textToService.getText());

                                       responseFromService.setText(response);

                          }

             }

});
```

Code Sample 16

The source code for this example is in attachment.

Messenger

The recommended method of communication between Activities and Services for different applications' Inter-Process Communication (IPC) is by using Messengers. In this case, the service defines a handler for different incoming messages. This handler is then used in Messenger for transferring data via messages.

To get bi-directional communication, the activity also has to register a Messenger and Handler. This Messenger is passed to the service in one of messages. Without this the service wouldn't know to who it should respond.

Message

The Activity and Service send Message objects to each other. Each message has a type, in the following example it is "MESSAGE_TYPE_TEXT", and an optional Bundle is attached to it.

```
Bundle data = new Bundle();
```

```
data.putCharSequence("data", text);

Message msg = Message.obtain(null, MESSAGE_TYPE_TEXT);

msg.setData(data);
```
Code Sample 17

Messenger Service

Since we want the service to be a remote one, we must modify its manifest file.
First of all, we add two lines in <service />

```
android:exported="true"

android:process=":remote"
```
Code Sample 18

This will enable us to view the service from outside. But to run it, we need to setup an action, also in the <service /> section.

```
<intent-filter>

    <action android:name="com.sprc.sample.service.StartService" />

</intent-filter>
```
Code Sample 19

We'll use this action in the activity to call on the service.

The full manifest file can be found in the attachment.

In this service, an instance of the Messenger class is necessary to process incoming messages. It will also be the binder between activities and the service. The service in the onBind() method must return this messenger:

```
final Messenger mMessenger = new Messenger(new IncomingHandler());

…

@Override

public IBinder onBind(Intent intent) {

             Log.d("MessengerService", "Binding messenger...");

             return mMessenger.getBinder();

}
```

Code Sample 20

In the class IncomingHandler we override the method handleMessage(). In this method we parse the message according to its type. The type of message is a simple integer constant:

```
/**

 * Message type: register the activity's messenger for receiving responses

 * from Service. We assume only one activity can be registered at one time.

 */

public static final int MESSAGE_TYPE_REGISTER = 1;

/**

 * Message type: text sent Activity<->Service

 */

public static final int MESSAGE_TYPE_TEXT = 2;



Messenger mResponseMessenger = null;



class IncomingHandler extends Handler {

             @Override

             public void handleMessage(Message msg) {

                          switch (msg.what) {

                          case MESSAGE_TYPE_TEXT:

                                       Bundle b = msg.getData();

                                       if (b != null) {

                                                    sendToActivity("Who's there? You wrote:
```

```
" +

b.getCharSequence("data"));

                                        } else {

                                                      sendToActivity("Who's there? Speak!");

                                        }

                                        break;

                           case MESSAGE_TYPE_REGISTER:

                                        Log.d("MessengerService", "Registered Activity's
Messenger.");

                                        mResponseMessenger = msg.replyTo;

                                        break;

                           default:

                                        super.handleMessage(msg);

                           }

             }

}
```

Code Sample 21

If the handler receives a message of type MESSAGE_TYPE_REGISTER, it will store the “replyTo” from the message in the “mResponseMessenger” field. This will be the Messenger on the activity’s side to which will be sending our responses from service.

If handler receives a message of type MESSAGE_TYPE_TEXT he will respond to the activity in the method sendToActivity():

```
void sendToActivity(CharSequence text) {

             if (mResponseMessenger == null) {

                           Log.d("MessengerService", "Cannot send message to activity - no
activity registered to this service.");

             } else {

                           Log.d("MessengerService", "Sending message to activity: " +
text);

                           Bundle data = new Bundle();

                           data.putCharSequence("data", text);

                           Message msg = Message.obtain(null, MESSAGE_TYPE_TEXT);

                           msg.setData(data);

                           try {

                                        mResponseMessenger.send(msg);

                           } catch (RemoteException e) {

                                        // We always have to trap RemoteException

// (DeadObjectException

                                        // is thrown if the target Handler no longer exists)

                                        e.printStackTrace();

                           }

             }

}
```

Code Sample 22

In the sendToActivity() method we create a new Message object, add a Bundle to it, and send it via the “mResponseMessenger” field. The call to mResponseMessenger.send() must be surrounded by a try-catch (Remote Exception) statement.

Messenger Activity

Similar to the service, we declare an IncomingHandler class to handle messages sent from services to activities:

```
class IncomingHandler extends Handler {

             @Override

             public void handleMessage(Message msg) {

                          if (msg.what == MessengerService.MESSAGE_TYPE_TEXT) {

                                      Bundle b = msg.getData();

                                      CharSequence text = null;

                                      if (b != null) {

                                                   text = b.getCharSequence("data");

                                      } else {

                                                   text = "Service responded with empty 
message";

                                      }

                                      Log.d("MessengerActivity", "Response: " + text);

                                      final TextView responseFromService = (TextView) 
findViewById(R.id.responseFromService);

                                      responseFromService.setText(text);

                          } else {

                                      super.handleMessage(msg);

                          }

             }

}



/**
```

```
 * Messenger used for receiving responses from service.

 */

final Messenger mMessenger = new Messenger(new IncomingHandler());
```

Code Sample 23

The Messenger using this handler will only handle MESSAGE_TYPE_TEXT messages. When such messages arrive from a service, the activity updates the TextView with the contents of the Bundle associated with message.

Next, the activity requires a ServiceConnection object for managing the connection with service:

```
/**

 * Messenger used for communicating with service.

 */

Messenger mService = null;



boolean mServiceConnected = false;



/**

 * Class for interacting with the main interface of the service.

 */

private ServiceConnection mConn = new ServiceConnection() {

             @Override

             public void onServiceConnected(ComponentName className, IBinder service) {

                          mService = new Messenger(service);

                          mServiceConnected = true;
                          // Register our messenger also on Service side:

                          Message msg = Message.obtain(null,

MessengerService.MESSAGE_TYPE_REGISTER);

                          msg.replyTo = mMessenger;

                          try {

                                      mService.send(msg);

                          } catch (RemoteException e) {

                                      // We always have to trap RemoteException

// (DeadObjectException

                                      // is thrown if the target Handler no longer exists)

                                      e.printStackTrace();

                          }

             }

             /**

              * Connection dropped.

              */

             @Override

             public void onServiceDisconnected(ComponentName className) {

                          Log.d("MessengerActivity", "Disconnected from service.");

                          mService = null;

                          mServiceConnected = false;
             }

};
```

Code Sample 24

In ServiceConnection we override the method onServiceConnected(). This method will be called after successful connection to the service. This is the moment to:

Create a messenger from the passed IBinder object. This Messenger will be used for sending messages to the service.

Send a registration message to the service (message of type MESSAGE_TYPE_REGISTER) with “replyTo” set to the previously created Messenger/IncomingHandler.

In the onStart() method the Activity binds to the service (passing this ServiceConnection object and the action described in the Service manifest) and starts it:

```
@Override

public void onStart() {

             super.onStart();

             bindService(new Intent("com.sprc.sample.service.StartService"), mConn,
Context.BIND_AUTO_CREATE);

}
```

Code Sample 25

Similarly it will unbind and stop the service in onStop():

```
@Override

protected void onStop() {

             super.onStop();

             if (mServiceConnected) {

                          unbindService(mConn);

                          stopService(new Intent(this, MessengerService.class));

                          mServiceConnected = false;
             }

}
```

Code Sample 26

Finally, the method sendToService(). It is very similar to the previously described method sendToActivity() on the service side:

```
/**

 * Sends message with text stored in bundle extra data ("data" key).

 *

 * @param text

 *            text to send

 */

void sendToService(CharSequence text) {

             if (mServiceConnected) {

                          Message msg = Message.obtain(null,

MessengerService.MESSAGE_TYPE_TEXT);

                          Bundle b = new Bundle();

                          b.putCharSequence("data", text);

                          msg.setData(b);

                          try {

                                       mService.send(msg);

                          } catch (RemoteException e) {

                                      // We always have to trap RemoteException

// (DeadObjectException
                                        // is thrown if the target Handler no longer exists)
                                        e.printStackTrace();
                            }
              } else {
                            Log.d("MessengerActivity", "Cannot send - not connected
to service.");
              }
}
```

Code Sample 27

ref:http://developer.samsung.com/android/technical-docs/Effective-communication-between-Service-and-Activity